**Week 1 – Algorithms Data Structure**

**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

**SOLUTION:**

**1)Why data structures and algorithms are essential:**

Efficient **data structures** allow quick access, modification, and deletion of inventory items.

**Algorithms** help minimize processing time when searching or managing items in large datasets (e.g., O(1) lookup vs O(n)).

**Suitable data structures:**

* Dictionary<int, Product> (HashMap) – For fast lookup using productId as key.
* List<Product> – Simple list if you only need to iterate or don't require quick search.

For large inventories, **Dictionary is best** for O(1) operations.

**2)Create Project:**

dotnet new console -n InventorySystem

cd InventorySystem

**3)Program Implementation:**

using System;

using System.Collections.Generic;

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 id, string name, int quantity, double price)

    {

        ProductId = id;

        ProductName = name;

        Quantity = quantity;

        Price = price;

    }

    public override string ToString()

    {

        return $"ID: {ProductId}, Name: {ProductName}, Qty: {Quantity}, Price: {Price}";

    }

}

class InventoryManager

{

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

    public void AddProduct(Product product)

    {

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

        {

            inventory[product.ProductId] = product;

            Console.WriteLine(" Product added.");

        }

        else

        {

            Console.WriteLine(" Product with this ID already exists.");

        }

    }

    public void UpdateProduct(int id, string name, int qty, double price)

    {

        if (inventory.ContainsKey(id))

        {

            var prod = inventory[id];

            prod.ProductName = name;

            prod.Quantity = qty;

            prod.Price = price;

            Console.WriteLine(" Product updated.");

        }

        else

        {

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

        }

    }

    public void DeleteProduct(int id)

    {

        if (inventory.Remove(id))

            Console.WriteLine(" Product deleted.");

        else

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

    }

    public void DisplayInventory()

    {

        if (inventory.Count == 0)

        {

            Console.WriteLine(" No products in inventory.");

            return;

        }

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

        foreach (var item in inventory.Values)

        {

            Console.WriteLine(item);

        }

    }

}

class Program

{

    static void Main(string[] args)

    {

        InventoryManager manager = new();

        while (true)

        {

            Console.WriteLine("\n========= Inventory Menu =========");

            Console.WriteLine("1. Add Product");

            Console.WriteLine("2. Update Product");

            Console.WriteLine("3. Delete Product");

            Console.WriteLine("4. Display Inventory");

            Console.WriteLine("5. Exit");

            Console.Write("Choose an option: ");

            string? input = Console.ReadLine();

            if (!int.TryParse(input, out int option))

            {

                Console.WriteLine(" Invalid input. Please enter a number.");

                continue;

            }

            switch (option)

            {

                case 1:

                    Console.Write("Enter ID: ");

                    if (!int.TryParse(Console.ReadLine(), out int id))

                    {

                        Console.WriteLine(" Invalid ID.");

                        break;

                    }

                    Console.Write("Enter Name: ");

                    string? name = Console.ReadLine();

                    if (string.IsNullOrWhiteSpace(name))

                    {

                        Console.WriteLine(" Invalid name.");

                        break;

                    }

                    Console.Write("Enter Quantity: ");

                    if (!int.TryParse(Console.ReadLine(), out int qty))

                    {

                        Console.WriteLine(" Invalid quantity.");

                        break;

                    }

                    Console.Write("Enter Price: ");

                    if (!double.TryParse(Console.ReadLine(), out double price))

                    {

                        Console.WriteLine(" Invalid price.");

                        break;

                    }

                    manager.AddProduct(new Product(id, name, qty, price));

                    break;

                case 2:

                    Console.Write("Enter ID to update: ");

                    if (!int.TryParse(Console.ReadLine(), out int uid))

                    {

                        Console.WriteLine(" Invalid ID.");

                        break;

                    }

                    Console.Write("Enter New Name: ");

                    string? uname = Console.ReadLine();

                    if (string.IsNullOrWhiteSpace(uname))

                    {

                        Console.WriteLine(" Invalid name.");

                        break;

                    }

                    Console.Write("Enter New Quantity: ");

                    if (!int.TryParse(Console.ReadLine(), out int uqty))

                    {

                        Console.WriteLine(" Invalid quantity.");

                        break;

                    }

                    Console.Write("Enter New Price: ");

                    if (!double.TryParse(Console.ReadLine(), out double uprice))

                    {

                        Console.WriteLine(" Invalid price.");

                        break;

                    }

                    manager.UpdateProduct(uid, uname, uqty, uprice);

                    break;

                case 3:

                    Console.Write("Enter ID to delete: ");

                    if (!int.TryParse(Console.ReadLine(), out int did))

                    {

                        Console.WriteLine(" Invalid ID.");

                        break;

                    }

                    manager.DeleteProduct(did);

                    break;

                case 4:

                    manager.DisplayInventory();

                    break;

                case 5:

                    Console.WriteLine(" Exiting the system...");

                    return;

                default:

                    Console.WriteLine(" Invalid choice.");

                    break;

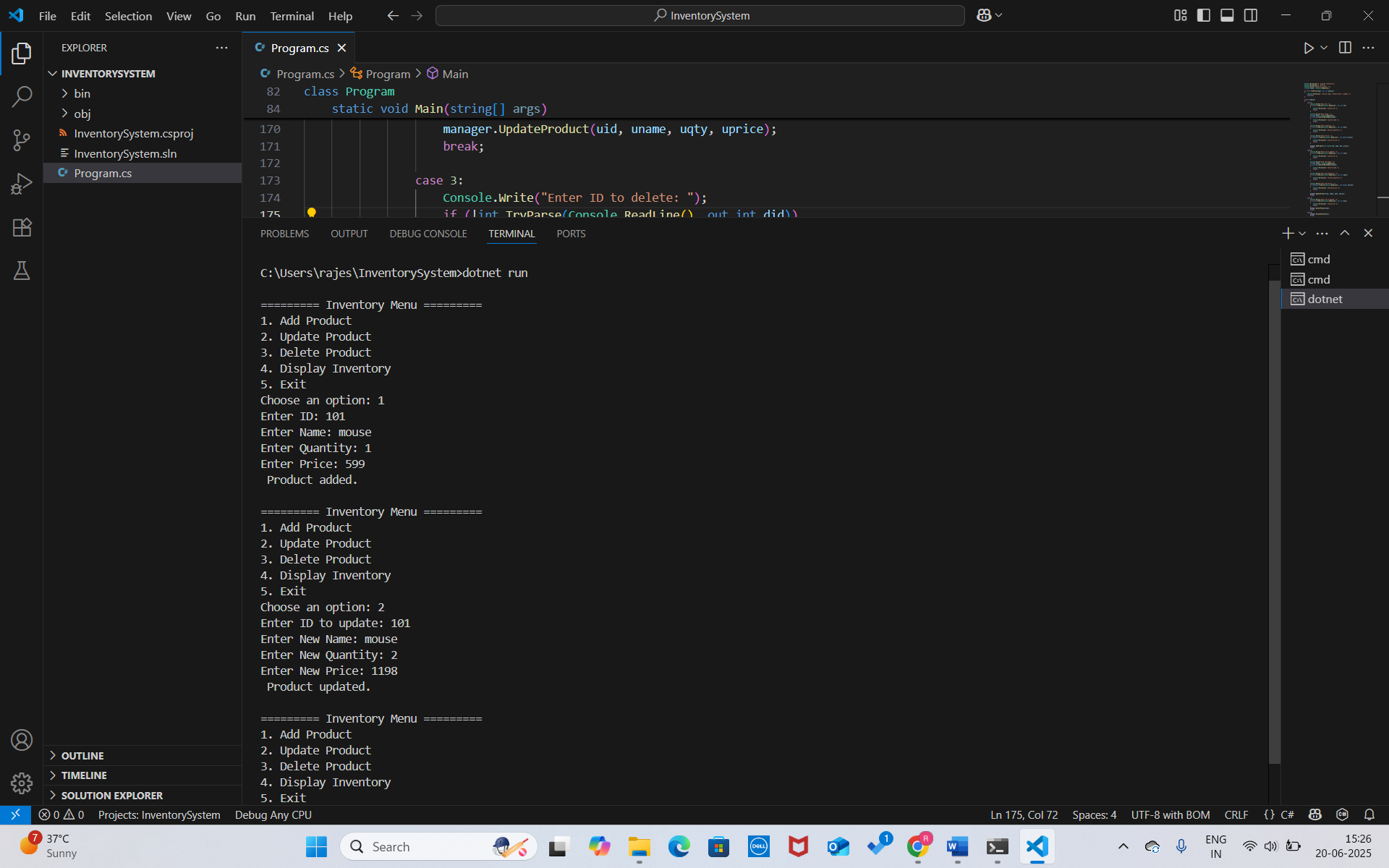
            }

        }

    }

}

**OUTPUT:**



**4)Analysis:**

**TIME COMPLEXITY OF OPERATION:**

| **Operation** | **Code Method** | **Time Complexity** | **Explanation** |
| --- | --- | --- | --- |
| **Add Product** | AddProduct() | **O(1)** average, **O(n)** worst | Adding to a dictionary is on average constant time. In rare cases (like hash collisions or rehashing), it may degrade. |
| **Update Product** | UpdateProduct() | **O(1)** | Lookup by key and update fields – both are constant time operations. |
| **Delete Product** | DeleteProduct() | **O(1)** | Removing by key is a constant time operation. |

**DISCUSSION:**

**Key Choice Optimization:**

* We use ProductId as the key in Dictionary<int, Product> which is an integer.
* Integers are fast to hash and compare — ideal for dictionary performance.

**Avoid Duplicate Keys:**

* Before adding, we check ContainsKey(productId) to prevent overwrite.
* This preserves unique product entries and prevents unnecessary computation.

**Bulk Operations:**

* For large inventory files, consider using Dictionary<int, Product> with AddRange pattern (using loop) for bulk load efficiency.

**Thread-Safety:**

* If you scale to multi-threaded apps, use ConcurrentDictionary instead of Dictionary.

**Memory Optimization:**

* Dictionary has a memory overhead due to hashing and resizing. If memory is a concern and lookups are not frequent, consider using List<Product> + manual search.

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Solution:**

**1)Big O Notation**

Big O describes how the **execution time** or **space requirements** of an algorithm grow relative to input size (**n**).

It helps developers compare the **efficiency** of algorithms regardless of hardware.

In search algorithms, **linear search** has a best-case time of **O(1)** (if the item is first), and a worst-case of **O(n)** (item not found or last). **Binary search**, which requires sorted data, also has a best-case of **O(1)** (middle match), but is more efficient overall with **O(log n)** time for both average and worst cases. This makes binary search much faster and more suitable for large, sorted datasets.

**2)Product Class**

public class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public string Category { get; set; }

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

{

ProductId = id;

ProductName = name;

Category = category;

}

public override string ToString()

{

return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";

}

}

**3)Implementation:**

using System;

public class Product

{

public int ProductId { get; set; }

    public string ProductName { get; set; }

    public string Category { get; set; }

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

    {

        ProductId = id;

        ProductName = name;

        Category = category;

    }

    public override string ToString()

    {

        return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";

    }

}

class ECommerceSearch

{

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

    {

        foreach (var product in products)

        {

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

                return product;

        }

        return null;

    }

    public 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[]

        {

            new Product(101, "Laptop", "Electronics"),

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

            new Product(103, "Shoes", "Fashion"),

            new Product(104, "Watch", "Accessories"),

            new Product(105, "T-shirt", "Fashion")

        };

        Array.Sort(products, (p1, p2) => p1.ProductName.CompareTo(p2.ProductName));

        Console.Write("Enter product name to search: ");

        string? name = Console.ReadLine();

        Console.WriteLine("\n Linear Search Result:");

        var linearResult = LinearSearch(products, name!);

        Console.WriteLine(linearResult != null ? linearResult.ToString() : "Product not found");

        Console.WriteLine("\n Binary Search Result:");

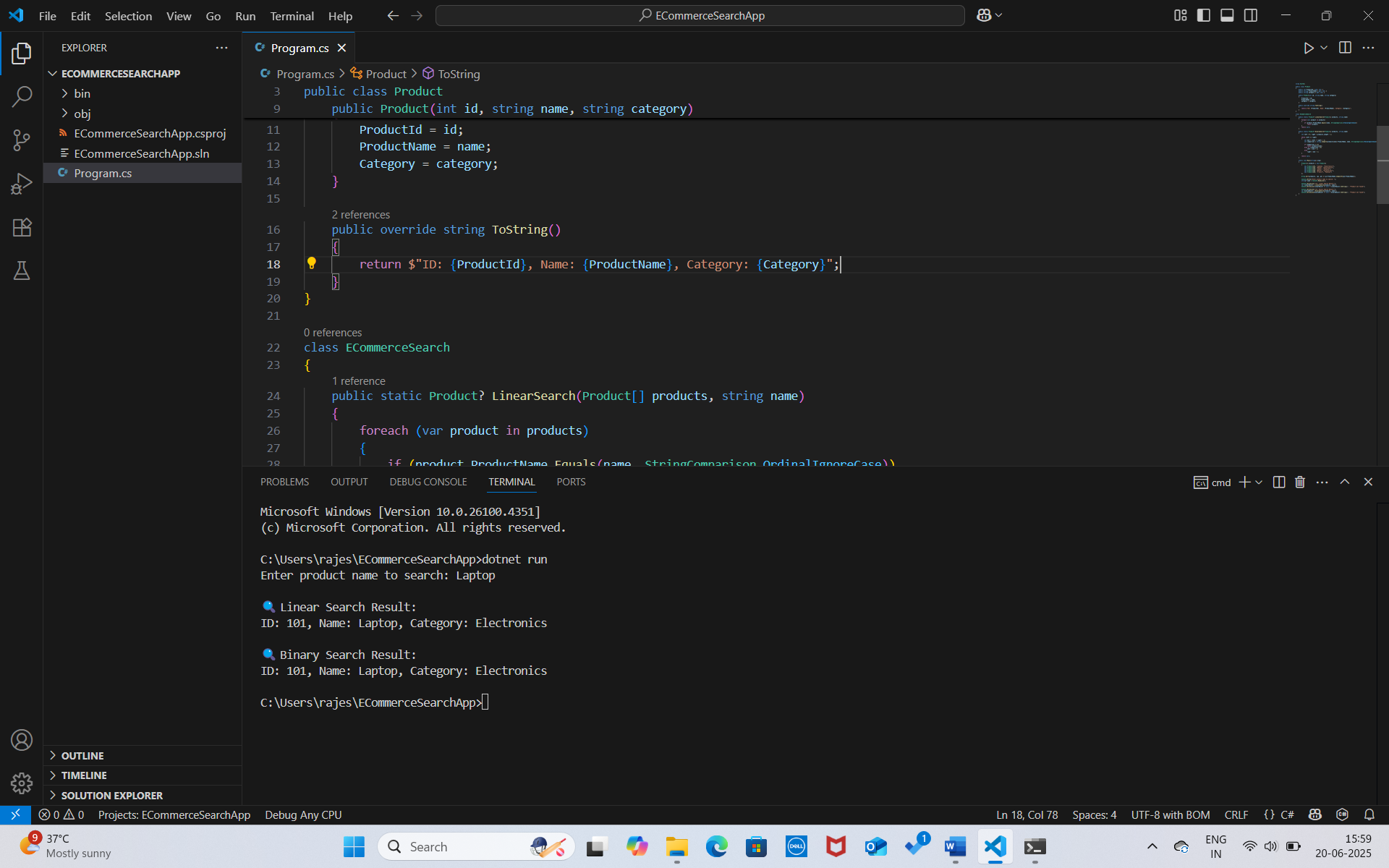
        var binaryResult = BinarySearch(products, name!);

        Console.WriteLine(binaryResult != null ? binaryResult.ToString() : "Product not found");

    }

}

**OUTPUT :**



**4)Analysis:**

**Time Complexity Comparison**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Requires Sorted Data?** |
| --- | --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) | No |
| **Binary Search** | O(1) | O(log n) | O(log n) | Yes |

For a **small or unsorted dataset**, use **Linear Search** (simple and flexible).

For a **large e-commerce platform** with **frequent search operations**, use **Binary Search**:

Requires maintaining a sorted array (can be done at load time).

Provides **logarithmic time** lookups, which are much faster for thousands of products.

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**SOLUTION:**

**1)Sorting Algorithms:**

**Bubble Sort** repeatedly swaps adjacent elements if they’re in the wrong order. It’s simple but inefficient, with a time complexity of **O(n²)** in average and worst cases, though it can be **O(n)** in the best case when the array is already sorted.

**Insertion Sort** builds a sorted array one item at a time. It’s efficient for small or nearly sorted datasets, with average and worst-case complexity of **O(n²)**.

**Quick Sort** is a fast divide-and-conquer algorithm that selects a pivot to partition the array. It performs very well in practice, with an average time complexity of **O(n log n)** and a worst case of **O(n²)** (rare and avoidable with good pivot selection).

**Merge Sort** also uses divide and conquer, splitting and merging sorted halves. It consistently runs in **O(n log n)** time but requires additional memory for merging.

**2)Order Class**

public class Order

{

public int OrderId { get; set; }

public string CustomerName { get; set; }

public double TotalPrice { get; set; }

public Order(int id, string name, double price)

{

OrderId = id;

CustomerName = name;

TotalPrice = price;

}

public override string ToString()

{

return $"OrderID: {OrderId}, Customer: {CustomerName}, Total: {TotalPrice}";

}

}

**3)Program:**

using System;

public class Order

{

    public int OrderId { get; set; }

    public string CustomerName { get; set; }

    public double TotalPrice { get; set; }

    public Order(int id, string name, double price)

    {

        OrderId = id;

        CustomerName = name;

        TotalPrice = price;

    }

    public override string ToString()

    {

        return $"OrderID: {OrderId}, Customer: {CustomerName}, Total: {TotalPrice}";

    }

}

class OrderSorter

{

    public static void BubbleSort(Order[] orders)

    {

        int n = orders.Length;

        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(Order[] orders, int low, int high)

    {

        if (low < high)

        {

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

            QuickSort(orders, low, pivotIndex - 1);

            QuickSort(orders, pivotIndex + 1, high);

        }

    }

    private static int Partition(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++;

                (orders[i], orders[j]) = (orders[j], orders[i]);

            }

        }

        (orders[i + 1], orders[high]) = (orders[high], orders[i + 1]);

        return i + 1;

    }

    public static void PrintOrders(Order[] orders)

    {

        foreach (var order in orders)

        {

            Console.WriteLine(order);

        }

    }

    static void Main(string[] args)

    {

        Order[] orders = new Order[]

        {

            new Order(1, "Alice", 500.00),

            new Order(2, "Bob", 1200.00),

            new Order(3, "Charlie", 300.00),

            new Order(4, "Diana", 800.00)

        };

        Console.WriteLine("Original Orders:");

        PrintOrders(orders);

        Console.WriteLine("\nSorted using Bubble Sort:");

        BubbleSort(orders);

        PrintOrders(orders);

        orders = new Order[]

        {

            new Order(1, "Alice", 500.00),

            new Order(2, "Bob", 1200.00),

            new Order(3, "Charlie", 300.00),

            new Order(4, "Diana", 800.00)

        };

        Console.WriteLine("\nSorted using Quick Sort:");

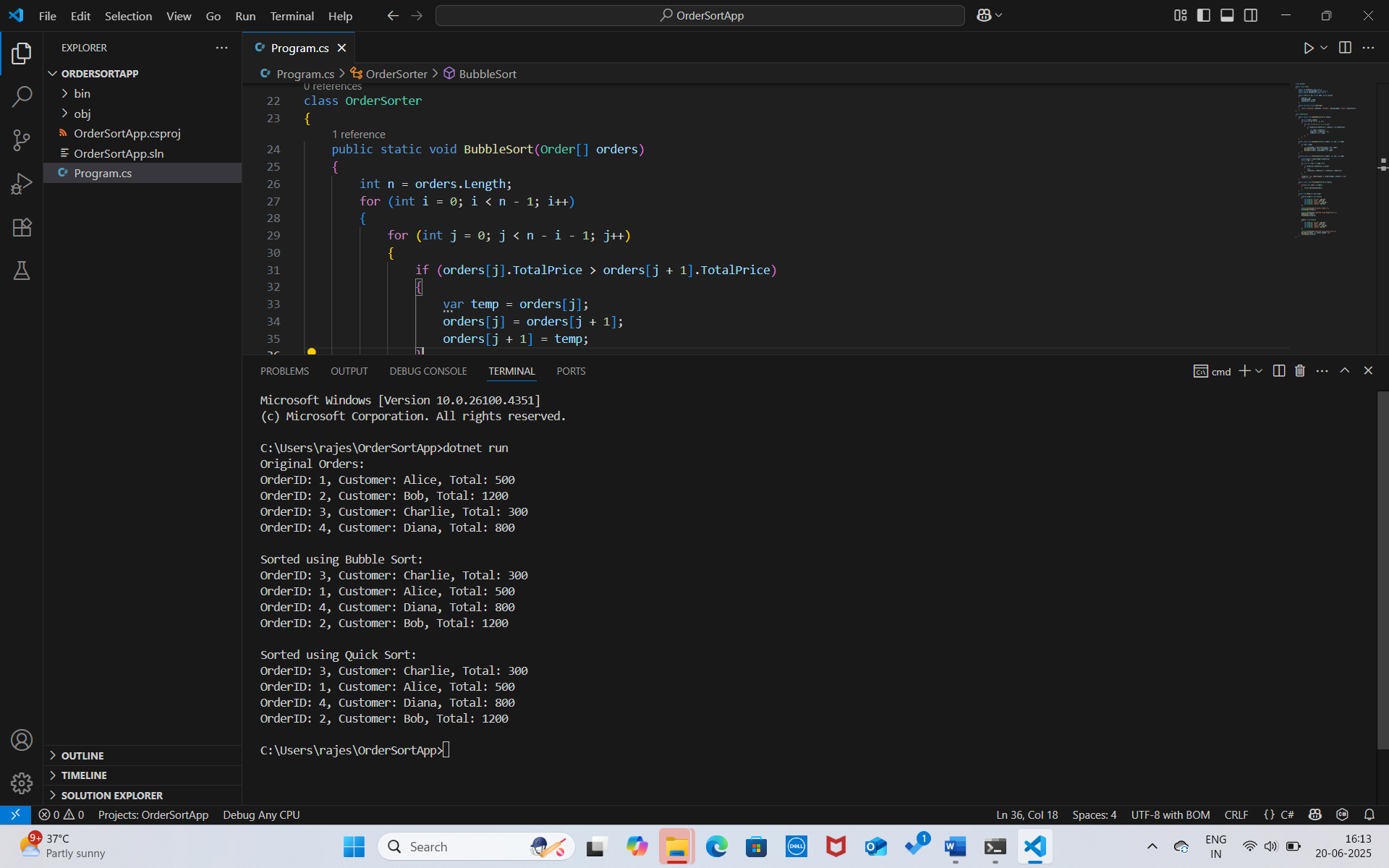
        QuickSort(orders, 0, orders.Length - 1);

        PrintOrders(orders);

    }

}

**OUTPUT :**



**4)Analysis:**

**Time Complexity Comparison**

| **Algorithm** | **Best** | **Average** | **Worst** |
| --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) |

**Why Quick Sort is Preferred**

**Much faster** for large datasets due to O(n log n) average time.

Uses less memory than Merge Sort.

Bubble Sort is too slow for real-world use — only good for educational purposes or very small datasets.

**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**SOLUTION:**

**1)Array Representation:**

In C#, arrays are stored in **contiguous memory blocks**, allowing each element to be accessed directly using its **index** in **O(1)** time. However, arrays are **fixed in size**, meaning their length cannot be changed once declared.

**Advantages:**

Fast access by index (O(1))

Simple and memory-efficient for small/known datasets

Good for static data structures

**2)** **Setup: Employee Class**

public class Employee

{

public int EmployeeId { get; set; }

public string Name { get; set; }

public string Position { get; set; }

public double Salary { get; set; }

public Employee(int id, string name, string position, double salary)

{

EmployeeId = id;

Name = name;

Position = position;

Salary = salary;

}

public override string ToString()

{

return $"ID: {EmployeeId}, Name: {Name}, Position: {Position}, Salary: {Salary}";

}

}

**3)Program :**

using System;

public class Employee

{

    public int EmployeeId { get; set; }

    public string Name { get; set; }

    public string Position { get; set; }

    public double Salary { get; set; }

    public Employee(int id, string name, string position, double salary)

    {

        EmployeeId = id;

        Name = name;

        Position = position;

        Salary = salary;

    }

    public override string ToString()

    {

        return $"ID: {EmployeeId}, Name: {Name}, Position: {Position}, Salary: {Salary}";

    }

}

class EmployeeManager

{

    static int capacity = 100;

    static Employee[] employees = new Employee[capacity];

    static int count = 0;

    public static void AddEmployee(Employee emp)

    {

        if (count < capacity)

        {

            employees[count++] = emp;

            Console.WriteLine(" Employee added.");

        }

        else

        {

            Console.WriteLine(" Employee list is full.");

        }

    }

    public static void TraverseEmployees()

    {

        Console.WriteLine(" Employee List:");

        for (int i = 0; i < count; i++)

        {

            Console.WriteLine(employees[i]);

        }

    }

    public static Employee? SearchEmployee(int id)

    {

        for (int i = 0; i < count; i++)

        {

            if (employees[i].EmployeeId == id)

                return employees[i];

        }

        return null;

    }

    public static void DeleteEmployee(int id)

    {

        int index = -1;

        for (int i = 0; i < count; i++)

        {

            if (employees[i].EmployeeId == id)

            {

                index = i;

                break;

            }

        }

        if (index != -1)

        {

            for (int j = index; j < count - 1; j++)

            {

                employees[j] = employees[j + 1];

            }

            employees[--count] = null!;

            Console.WriteLine("Employee deleted.");

        }

        else

        {

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

        }

    }

    static void Main(string[] args)

    {

        AddEmployee(new Employee(101, "Alice", "Manager", 75000));

        AddEmployee(new Employee(102, "Bob", "Engineer", 55000));

        AddEmployee(new Employee(103, "Charlie", "HR", 45000));

        TraverseEmployees();

        Console.WriteLine("\n Searching for Employee ID 102:");

        var emp = SearchEmployee(102);

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

        Console.WriteLine("\n Deleting Employee ID 102:");

        DeleteEmployee(102);

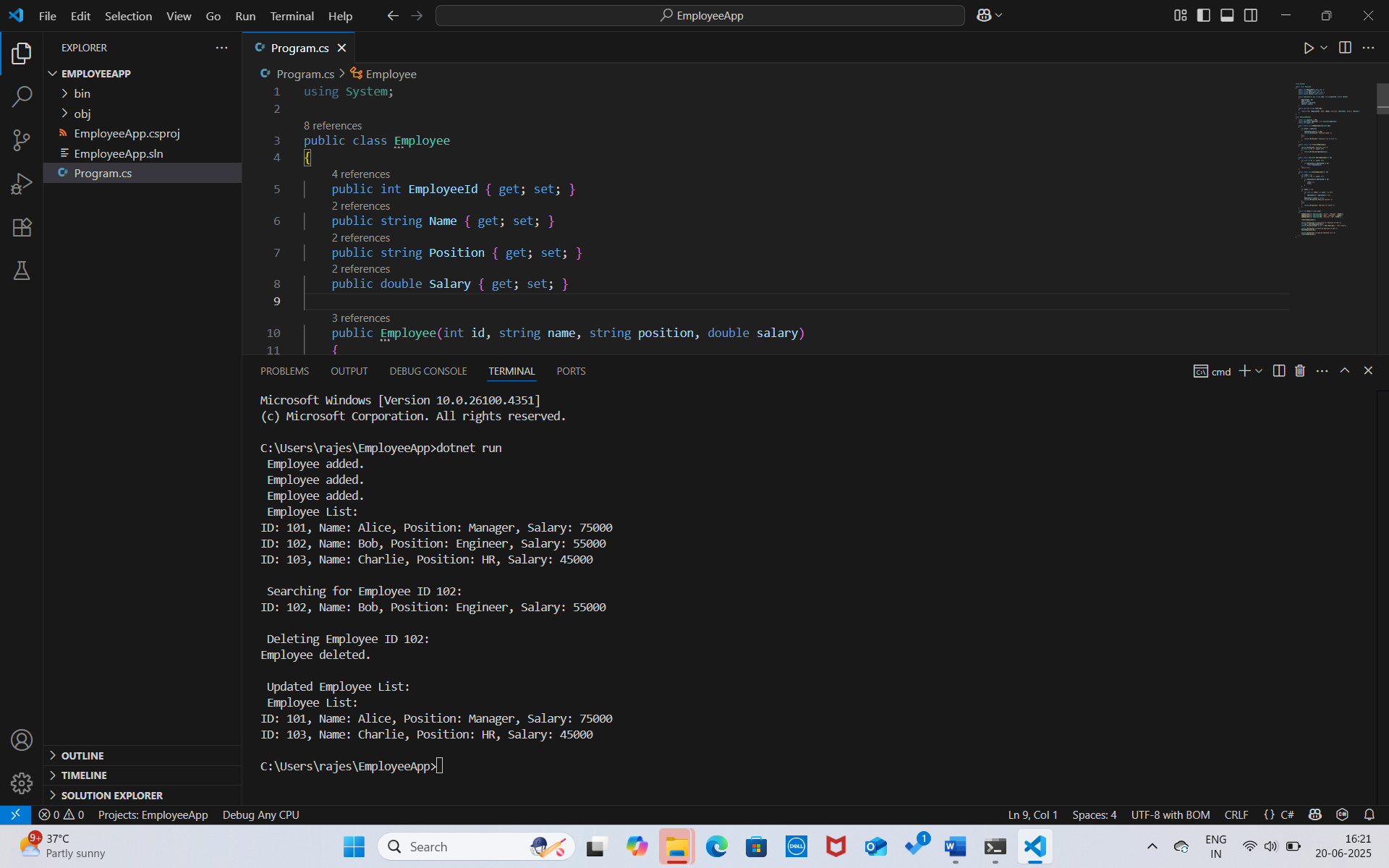
        Console.WriteLine("\n Updated Employee List:");

        TraverseEmployees();

    }

}

**OUTPUT :**



**4)Analysis:**

| **Operation** | **Time Complexity** |
| --- | --- |
| **Add** | O(1) |
| **Traverse** | O(n) |
| **Search** | O(n) (linear search) |
| **Delete** | O(n) (shift elements) |

**Limitations of Arrays**

**Fixed size**: Must predefine capacity.

**Insertion/Deletion** at specific indices requires shifting → slower for large datasets.

Not ideal for dynamic collections where frequent resizing is needed.

**When to Use Arrays**

When you know the **maximum size**.

When you need **fast access by index**.

When data is relatively static or small.

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**SOLUTION:**

**1)Understand Linked Lists**

A **singly linked list** is a linear data structure where each node points to the next node, allowing one-way (forward) traversal. It is efficient for insertion or deletion at the head or a known position and uses less memory compared to a doubly linked list.

A **doubly linked list** allows two-way traversal since each node has both next and prev pointers. It provides easier insertion and deletion at both ends or in the middle but uses slightly more memory due to the extra pointer.

**Different types of single and doubly linked lists:**

**Singly Linked List**

Each **node** contains:

**Data**

A pointer/reference to the **next node**.

**Traversal** is only possible in **one direction** — from head to tail.

**Operations** (insert, delete) at the head are fast (O(1)), but to access or delete elements in the middle, you must traverse (O(n)).

**Doubly Linked List**

Each **node** contains:

**Data**

A pointer to the **next node**

A pointer to the **previous node**

Supports **two-way traversal** — forward and backward.

**2)** **Task Class**

public class Task

{

public int TaskId { get; set; }

public string TaskName { get; set; }

public string Status { get; set; }

public Task(int id, string name, string status)

{

TaskId = id;

TaskName = name;

Status = status;

}

public override string ToString()

{

return $"TaskID: {TaskId}, Name: {TaskName}, Status: {Status}";

}

}

**3)Program:**

using System;

public class Task

{

    public int TaskId { get; set; }

    public string TaskName { get; set; }

    public string Status { get; set; }

    public Task(int id, string name, string status)

    {

        TaskId = id;

        TaskName = name;

        Status = status;

    }

    public override string ToString()

    {

        return $"TaskID: {TaskId}, Name: {TaskName}, Status: {Status}";

    }

}

public class TaskNode

{

    public Task TaskData { get; set; }

    public TaskNode? Next { get; set; }

    public TaskNode(Task task)

    {

        TaskData = task;

        Next = null;

    }

}

public class TaskLinkedList

{

    private TaskNode? head;

    public void AddTask(Task task)

    {

        TaskNode newNode = new TaskNode(task);

        if (head == null)

        {

            head = newNode;

        }

        else

        {

            TaskNode current = head;

            while (current.Next != null)

                current = current.Next;

            current.Next = newNode;

        }

        Console.WriteLine(" Task added.");

    }

    public void TraverseTasks()

    {

        Console.WriteLine("\n Task List:");

        TaskNode? current = head;

        while (current != null)

        {

            Console.WriteLine(current.TaskData);

            current = current.Next;

        }

    }

    public Task? SearchTask(int id)

    {

        TaskNode? current = head;

        while (current != null)

        {

            if (current.TaskData.TaskId == id)

                return current.TaskData;

            current = current.Next;

        }

        return null;

    }

    public void DeleteTask(int id)

    {

        if (head == null) return;

        if (head.TaskData.TaskId == id)

        {

            head = head.Next;

            Console.WriteLine(" Task deleted.");

            return;

        }

        TaskNode? current = head;

        while (current.Next != null && current.Next.TaskData.TaskId != id)

        {

            current = current.Next;

        }

        if (current.Next != null)

        {

            current.Next = current.Next.Next;

            Console.WriteLine(" Task deleted.");

        }

        else

        {

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

        }

    }

}

class Program

{

    static void Main(string[] args)

    {

        TaskLinkedList taskList = new TaskLinkedList();

        taskList.AddTask(new Task(1, "Design UI", "Pending"));

        taskList.AddTask(new Task(2, "Develop API", "In Progress"));

        taskList.AddTask(new Task(3, "Testing", "Not Started"));

        taskList.TraverseTasks();

        Console.WriteLine("\n Searching Task ID 2:");

        var task = taskList.SearchTask(2);

        Console.WriteLine(task != null ? task.ToString() : " Not found");

        Console.WriteLine("\n Deleting Task ID 2:");

        taskList.DeleteTask(2);

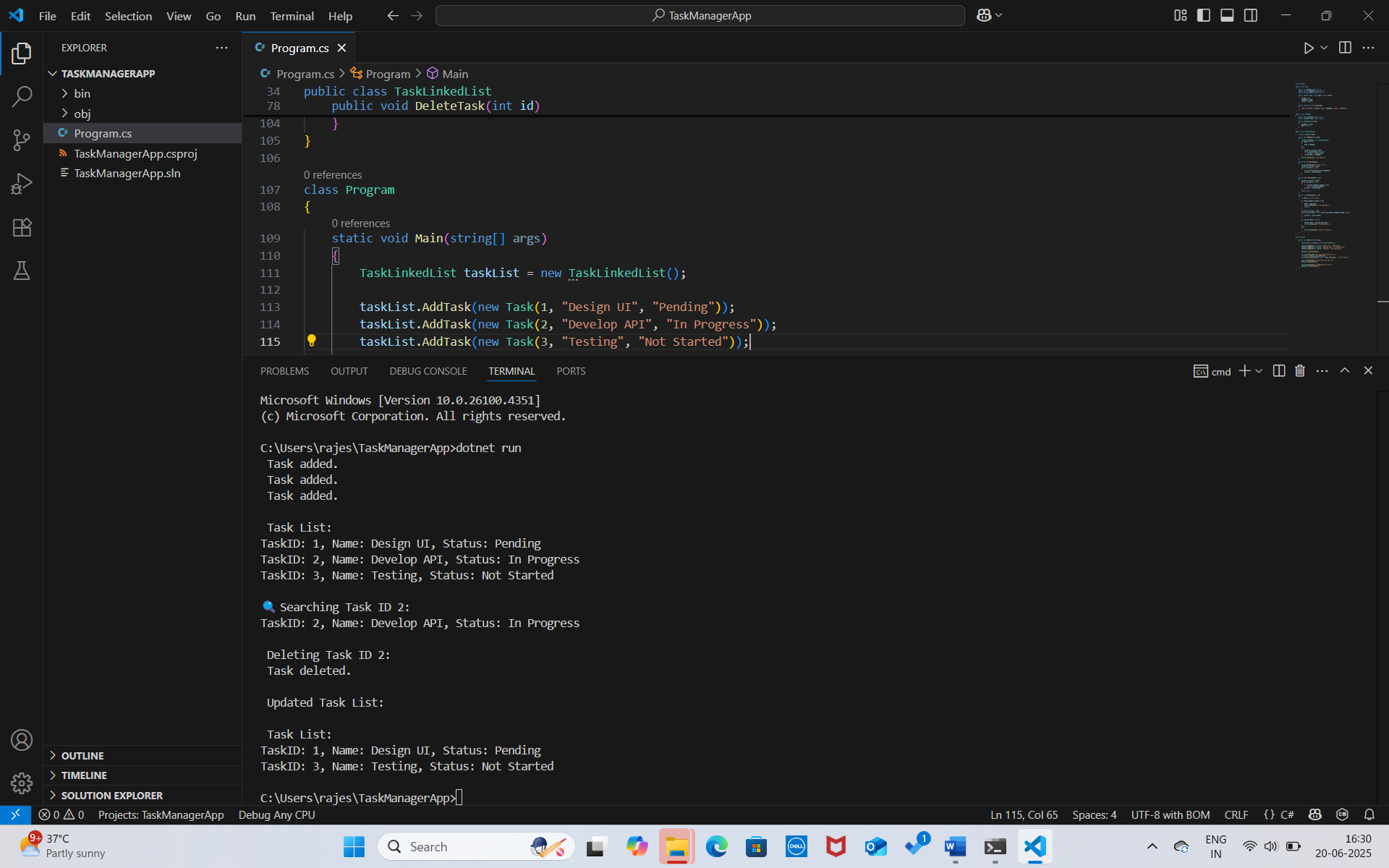
        Console.WriteLine("\n Updated Task List:");

        taskList.TraverseTasks();

    }

}

**OUTPUT :**



**4)Analysis:**

**Time Complexity**

| **Operation** | **Time Complexity** |
| --- | --- |
| **Add (at end)** | O(n) (need to traverse) |
| **Add (at head)** | O(1) |
| **Search** | O(n) |
| **Traverse** | O(n) |
| **Delete (by value)** | O(n) (need to find and unlink node) |

**Advantages of Linked Lists over Arrays for Dynamic Data**

**Dynamic size**: Linked lists can grow or shrink at runtime without predefined size.

**Efficient insertions/deletions**: Especially at the head or middle, since there's no need to shift elements like in arrays.

**Memory efficiency**: No need to allocate extra unused space upfront (unlike arrays).

**Better for unpredictable workloads**: Ideal for applications where the number of elements changes frequently (e.g., task queues, schedulers).

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**Steps:**

1. **Understand Search Algorithms:**
   * Explain linear search and binary search algorithms.
2. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

**SOLUTION:**

**1)Search Algorithms:**

**Linear search** checks each element one by one and works on unsorted data. It's simple but less efficient for large datasets, with a time complexity of **O(1)** in the best case and **O(n)** in the worst.

**Binary search** works only on **sorted data**, dividing the search space in half each time. It’s much faster for large datasets, with a time complexity of **O(log n)** on average and worst case.

**2)Book Class:**

public class Book

{

public int BookId { get; set; }

public string Title { get; set; }

public string Author { get; set; }

public Book(int id, string title, string author)

{

BookId = id;

Title = title;

Author = author;

}

public override string ToString()

{

return $"ID: {BookId}, Title: {Title}, Author: {Author}";

}

}

**3)Program:**

using System;

public class Book

{

    public int BookId { get; set; }

    public string Title { get; set; }

    public string Author { get; set; }

    public Book(int id, string title, string author)

    {

        BookId = id;

        Title = title;

        Author = author;

    }

    public override string ToString()

    {

        return $"ID: {BookId}, Title: {Title}, Author: {Author}";

    }

}

class LibraryManager

{

    public static Book? LinearSearch(Book[] books, string title)

    {

        foreach (var book in books)

        {

            if (book.Title.Equals(title, StringComparison.OrdinalIgnoreCase))

                return book;

        }

        return null;

    }

    public static Book? BinarySearch(Book[] books, string title)

    {

        int left = 0;

        int right = books.Length - 1;

        while (left <= right)

        {

            int mid = (left + right) / 2;

            int compare = string.Compare(books[mid].Title, title, true);

            if (compare == 0)

                return books[mid];

            else if (compare < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

    static void Main(string[] args)

    {

        Book[] books = new Book[]

        {

            new Book(101, "Algorithms", "Cormen"),

            new Book(102, "Clean Code", "Robert Martin"),

            new Book(103, "Design Patterns", "Erich Gamma"),

            new Book(104, "Intro to AI", "Stuart Russell")

        };

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

        var result1 = LinearSearch(books, "Design Patterns");

        Console.WriteLine(result1 != null ? result1.ToString() : " Book not found.");

        Array.Sort(books, (a, b) => a.Title.CompareTo(b.Title));

        Console.WriteLine("\n Binary Search for 'Clean Code':");

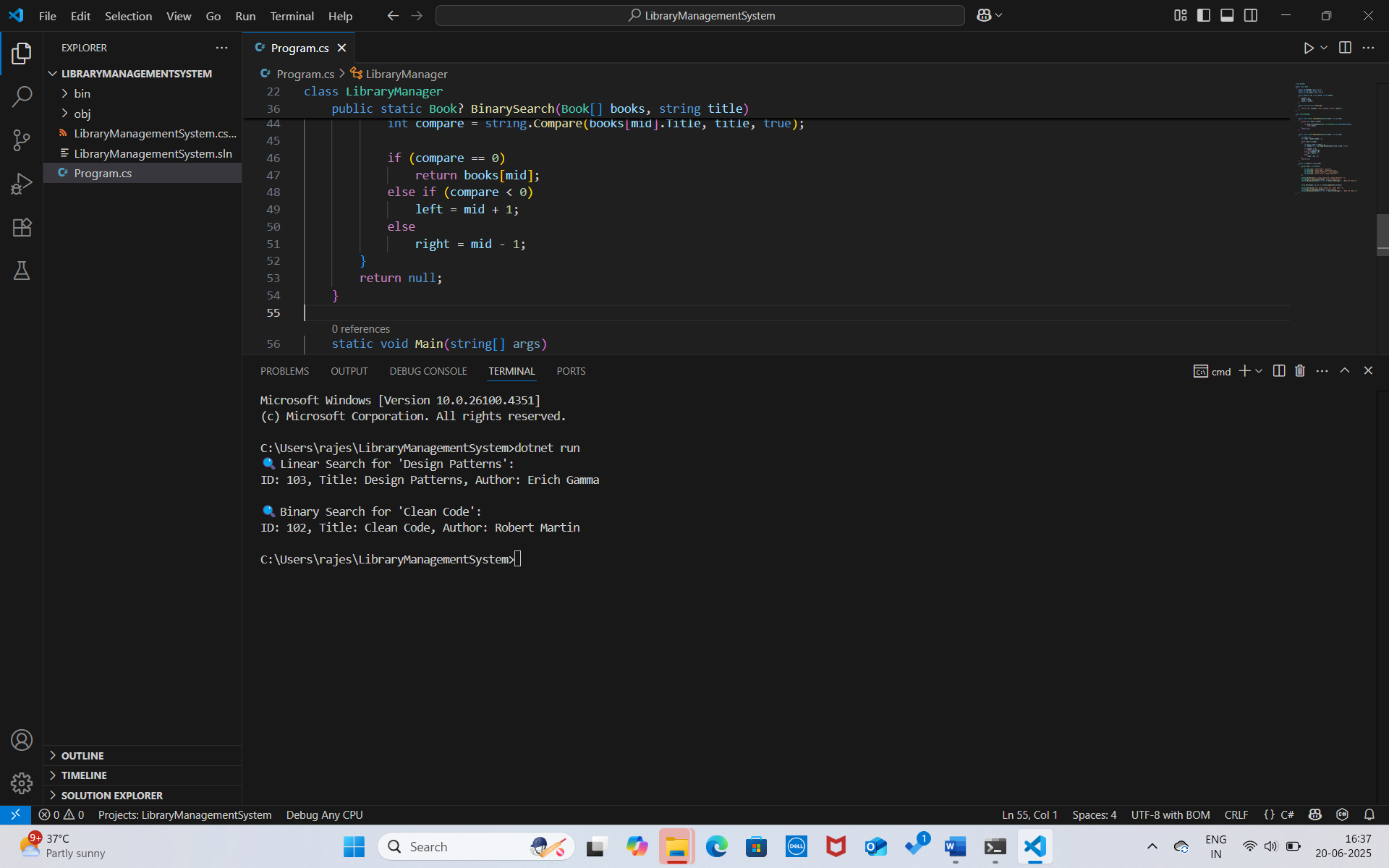
        var result2 = BinarySearch(books, "Clean Code");

        Console.WriteLine(result2 != null ? result2.ToString() : " Book not found.");

    }

}

**OUTPUT :**



**4)Analysis:**

**Time Complexity Comparison**

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

**When to Use**

Use **linear search** for **small or unsorted** datasets.

Use **binary search** for **large sorted** datasets for better performance.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**SOLUTION:**

**1)Recursive Algorithms:**

**Recursion** is a technique where a function calls itself to solve smaller instances of a problem. It simplifies problems that follow a repetitive pattern, such as computing future values based on previous years. Recursion works well when each result depends directly on the result of a smaller subproblem.

**2)Setup:**

**Recursive Method for Forecasting**

We predict future value using:

FutureValue(n)=FutureValue(n−1)×(1+growthRate)\text{FutureValue}(n) = \text{FutureValue}(n - 1) \times (1 + \text{growthRate})FutureValue(n)=FutureValue(n−1)×(1+growthRate)

Where:

* initialValue = investment amount
* growthRate = annual growth (e.g., 0.08 for 8%)
* n = number of years

**3)Program:**

using System;

class Program

{

    public static double CalculateFutureValue(double initialValue, double growthRate, int years)

    {

        if (years == 0)

            return initialValue;

        return CalculateFutureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

    }

    static void Main(string[] args)

    {

        double initial = 10000;

        double rate = 0.08;

        int years = 5;

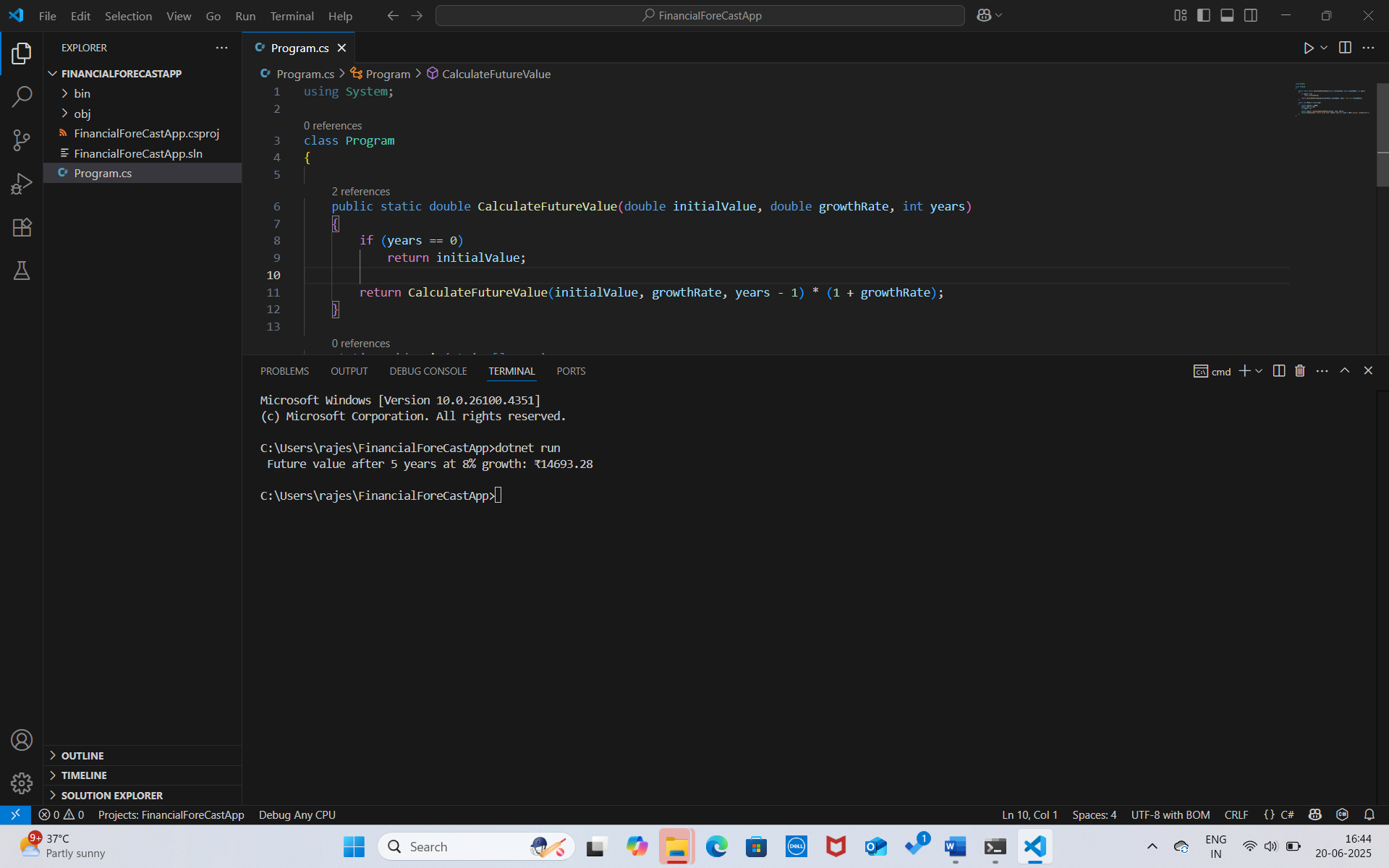
        double result = CalculateFutureValue(initial, rate, years);

        Console.WriteLine($" Future value after {years} years at {rate \* 100}% growth: ₹{result:F2}");

    }

}

**OUTPUT :**



**4)Analysis:**

**Time Complexity:**

Each call makes one smaller call → **O(n)** where n = number of years.

**Optimization:**

Avoid recomputation by using:

**Memoization** (store intermediate values)

Or use an **iterative approach** (more efficient and stack-safe):