**WEEK 1**

**Data Structures And Algorithms**

**Name: Rajesh A  
Superset ID: 6384182**

**Exercise 1: Inventory Management System(.NET)**

**Scenario:**

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

### What I Learned

I learned how to build a basic inventory management system in C# using classes and collections. This program demonstrates adding, updating, deleting, and displaying products using a dictionary for efficient item access by product ID.

**CODE:**

using System;

using System.Collections.Generic;

namespace InventoryManagement

{

class Product

{

public int ProductId;

public string ProductName;

public int Quantity;

public double Price;

}

class Inventory

{

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

public void AddProduct(Product product) => products[product.ProductId] = product;

public void UpdateProduct(int id, Product updatedProduct) => products[id] = updatedProduct;

public void DeleteProduct(int id) => products.Remove(id);

public void DisplayInventory()

{

foreach (var p in products.Values)

Console.WriteLine($"{p.ProductId} - {p.ProductName} - {p.Quantity} - {p.Price}");

}

}

class Program

{

static void Main()

{

Inventory inventory = new();

inventory.AddProduct(new Product { ProductId = 1, ProductName = "Mouse", Quantity = 50, Price = 300 });

inventory.AddProduct(new Product { ProductId = 2, ProductName = "Keyboard", Quantity = 20, Price = 600 });

inventory.UpdateProduct(1, new Product { ProductId = 1, ProductName = "Wireless Mouse", Quantity = 40, Price = 500 });

inventory.DeleteProduct(2);

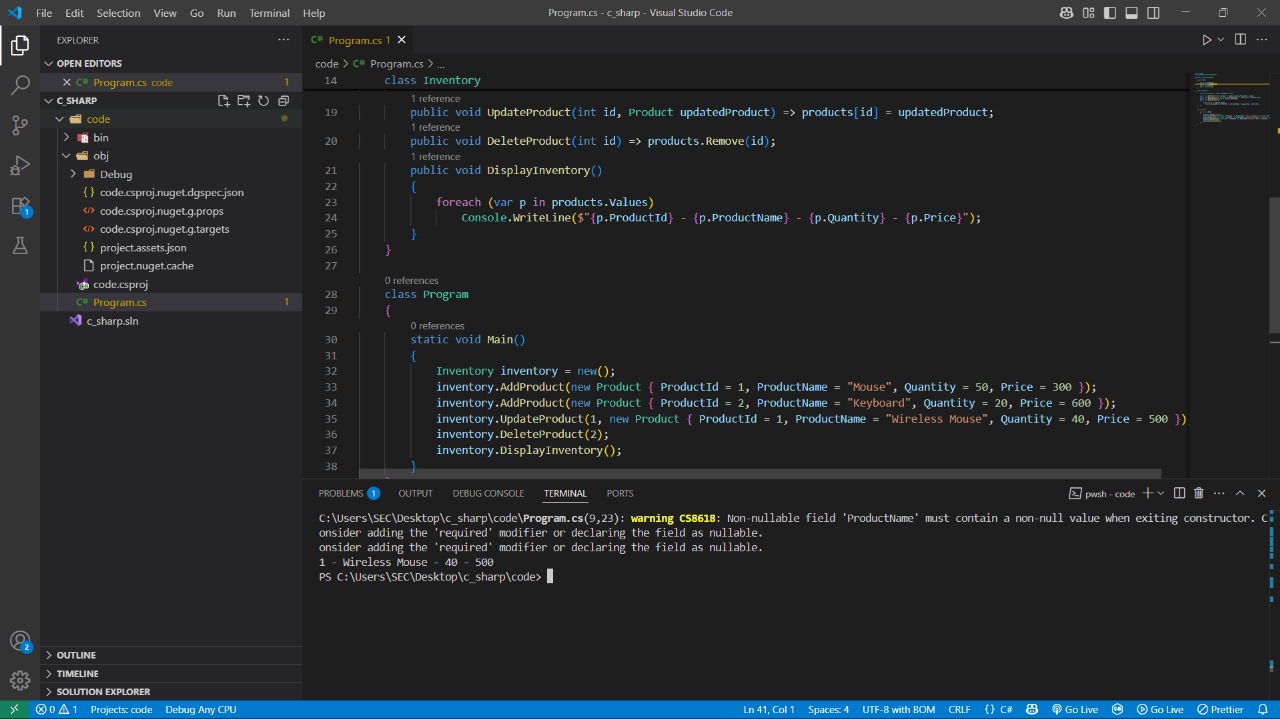
inventory.DisplayInventory();

}

}

}

**OUTPUT:**



### Conclusion

The inventory system performs basic product management operations efficiently using a dictionary for fast access and updates.

**Exercise 2: E-commerce Platform Search Function(.NET)**

**Scenario:**

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

### What I Learned

I learned how to implement both linear and binary search algorithms in C# to find a product by name. Binary search requires the array to be sorted, offering better performance than linear search on large datasets.

**CODE:**

namespace SearchFunction

{

class Product

{

public int ProductId;

public string ProductName;

public string Category;

}

class Program2

{

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

{

for (int i = 0; i < products.Length; i++)

if (products[i].ProductName == name) return i;

return -1;

}

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

{

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

while (left <= right)

{

int mid = (left + right) / 2;

int cmp = string.Compare(products[mid].ProductName, name);

if (cmp == 0) return mid;

if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return -1;

}

static void Main()

{

Product[] products = {

new Product { ProductId = 1, ProductName = "Camera", Category = "Electronics" },

new Product { ProductId = 2, ProductName = "Phone", Category = "Electronics" },

new Product { ProductId = 3, ProductName = "TV", Category = "Electronics" }

};

Array.Sort(products, (a, b) => string.Compare(a.ProductName, b.ProductName));

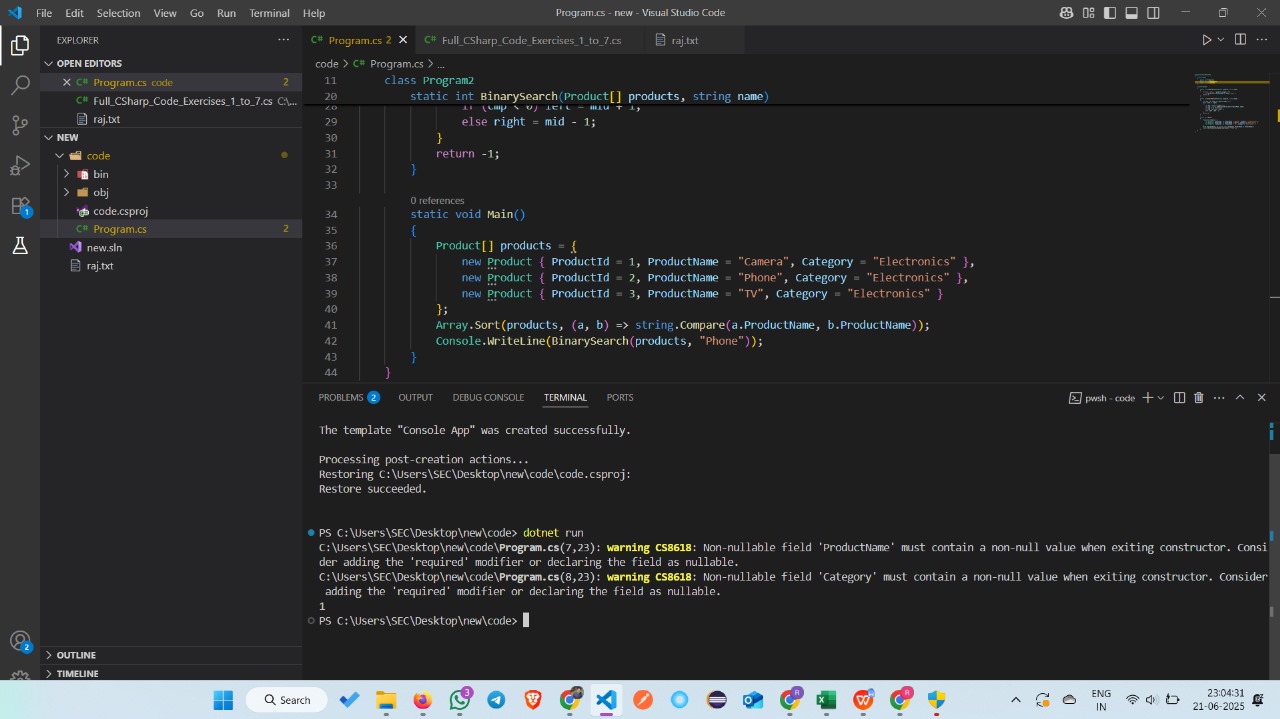
Console.WriteLine(BinarySearch(products, "Phone"));

}

}

}

**OUTPUT:**



### Conclusion

Binary search provides a faster way to locate a product in a sorted list compared to linear search.

**Exercise 3: Sorting Customer Orders(.NET)**

**Scenario:**

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

### What I Learned

I learned how to implement the Quick Sort algorithm in C# to sort a list of orders based on their total price. This algorithm improves performance for large datasets by using a divide-and-conquer approach to sort elements efficiently in-place.

**CODE:**

namespace SortingOrders

{

class Order

{

public int OrderId;

public required string CustomerName;

public double TotalPrice;

}

class Program3

{

static void QuickSort(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);

}

}

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;

}

static void Main()

{

Order[] orders = {

new Order { OrderId = 1, CustomerName = "A", TotalPrice = 500 },

new Order { OrderId = 2, CustomerName = "B", TotalPrice = 1500 },

new Order { OrderId = 3, CustomerName = "C", TotalPrice = 1000 }

};

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

foreach (var o in orders)

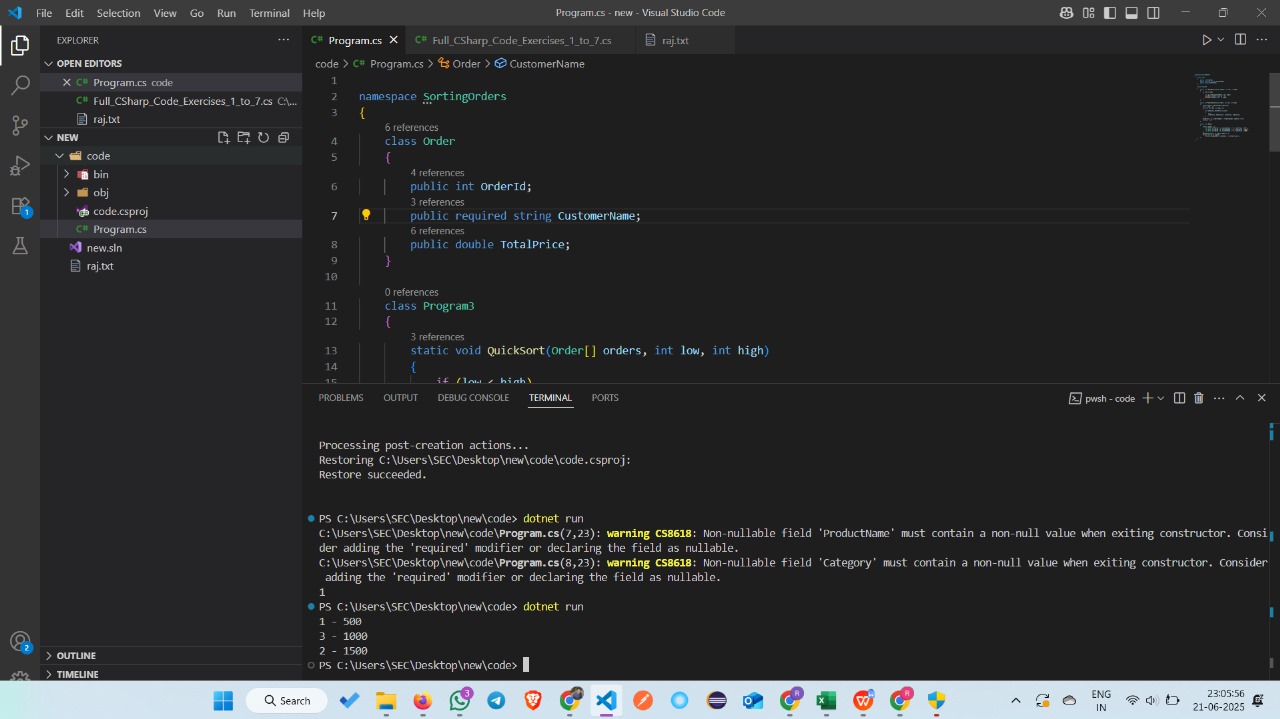
Console.WriteLine($"{o.OrderId} - {o.TotalPrice}");

}

}

}

**OUTPUT:**



### Conclusion

Quick Sort efficiently arranged the orders by total price using recursion and partitioning, making it ideal for large datasets with better average-case performance.

**Exercise 4: Employee Management System(.NET)**

**Scenario:**

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

### What I Learned

I learned how to create and manage an array of employee records in C#. This program allows adding employees, displaying basic information, and searching for a specific employee by ID using a simple loop.

**CODE:**

namespace EmployeeSystem

{

class Employee

{

public int EmployeeId;

public string Name;

public string Position;

public double Salary;

}

class Program4

{

static void Main()

{

Employee[] employees = new Employee[100];

int count = 0;

employees[count++] = new Employee { EmployeeId = 1, Name = "John", Position = "Manager", Salary = 70000 };

employees[count++] = new Employee { EmployeeId = 2, Name = "Jane", Position = "Clerk", Salary = 40000 };

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

Console.WriteLine($"{employees[i].EmployeeId} - {employees[i].Name}");

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

if (employees[i].EmployeeId == 1)

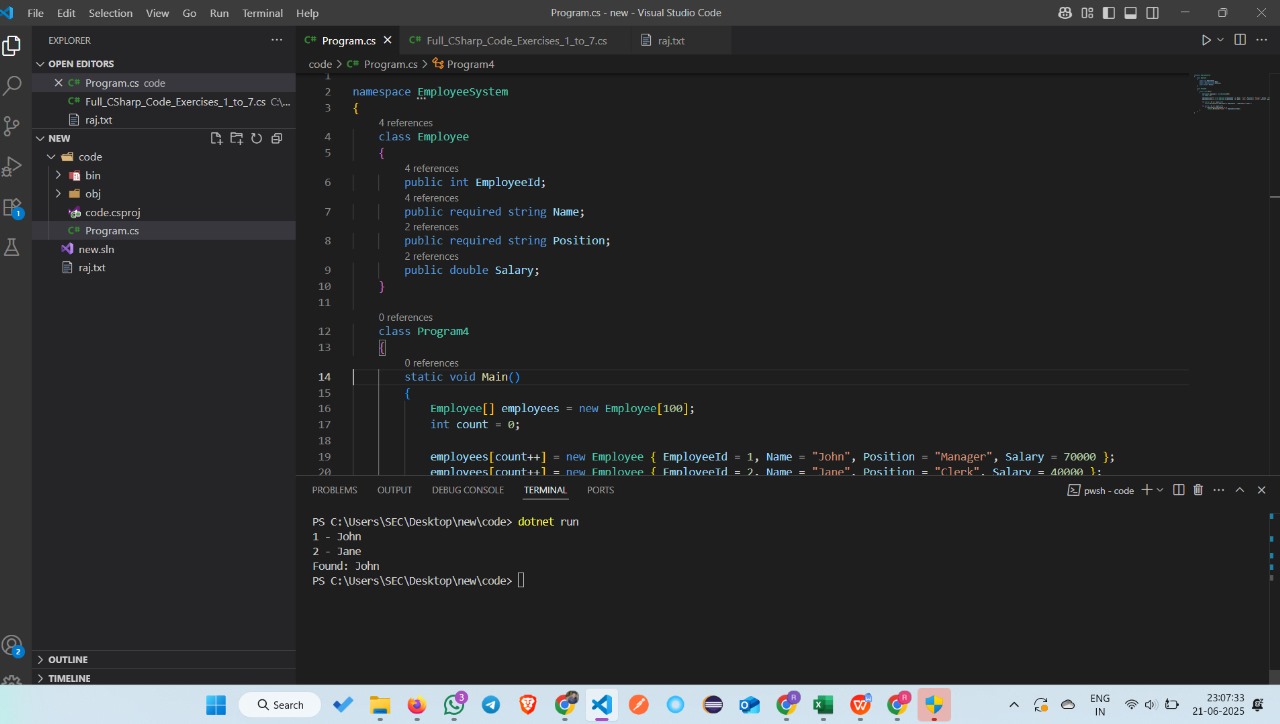
Console.WriteLine("Found: " + employees[i].Name);

}

}

}

**OUTPUT:**



### Conclusion

This program provides basic employee record handling with add, display, and search functionalities using a static array.

**Exercise 5: Task Management System(.NET)**

**Scenario:**

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

### What I Learned:

I learned how to use a singly linked list to manage tasks in C#. The program demonstrates adding tasks to the front, traversing all tasks, searching a task by ID, and deleting a task from the list dynamically using reference links.

**CODE:**

using System;

class Task

{

public int TaskId;

public required string TaskName;

public required string Status;

public required Task Next;

}

class TaskList

{

private Task? head;

public void AddTask(int id, string name, string status)

{

Task newTask = new Task { TaskId = id, TaskName = name, Status = status, Next = head };

head = newTask;

}

public void Traverse()

{

Task current = head;

while (current != null)

{

Console.WriteLine($"{current.TaskId} - {current.TaskName} - {current.Status}");

current = current.Next;

}

}

public Task Search(int id)

{

Task current = head;

while (current != null)

{

if (current.TaskId == id) return current;

current = current.Next;

}

return null;

}

public void Delete(int id)

{

Task current = head, prev = null;

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

{

prev = current;

current = current.Next;

}

if (current == null) return;

if (prev == null) head = head.Next;

else prev.Next = current.Next;

}

}

class Program5

{

static void Main()

{

TaskList list = new TaskList();

list.AddTask(1, "Design", "Pending");

list.AddTask(2, "Development", "In Progress");

list.AddTask(3, "Testing", "Not Started");

list.Traverse();

Console.WriteLine("After Deleting Task 2");

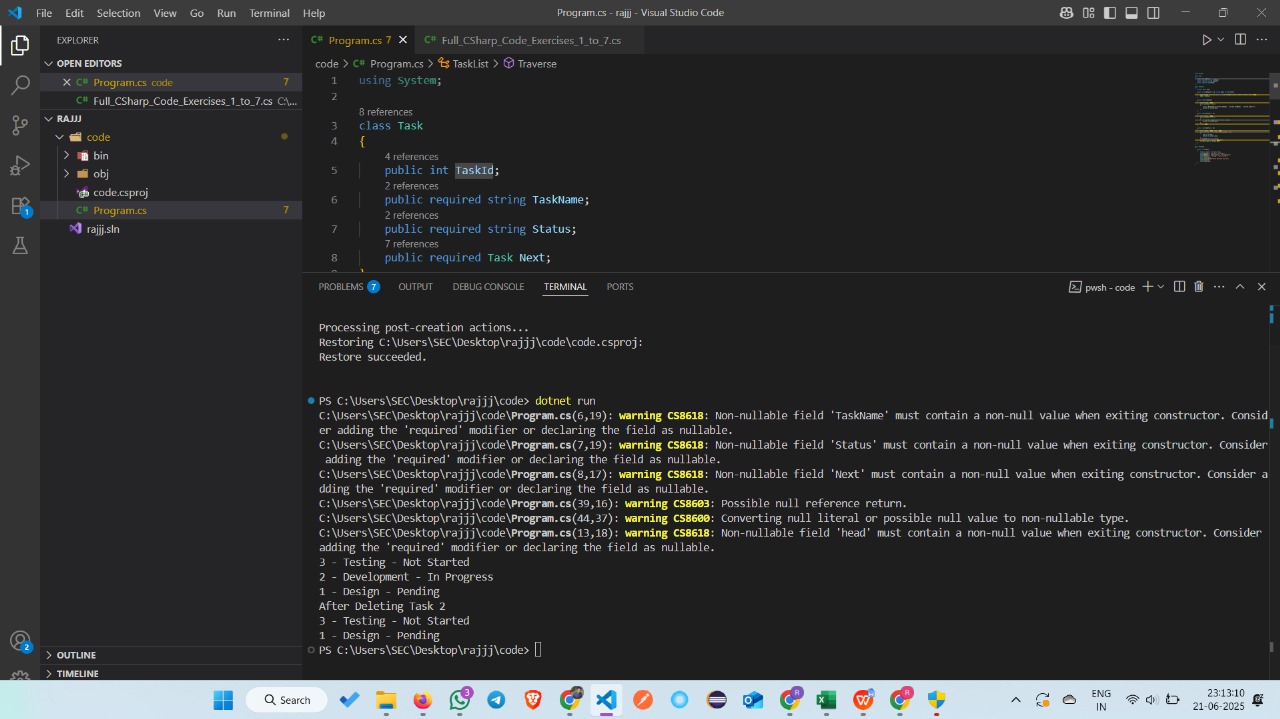
list.Delete(2);

list.Traverse();

}

}

**OUTPUT:**



### Conclusion:

The program efficiently handles dynamic task operations like insertion, deletion, and searching using a singly linked list.

**Exercise 6: Library Management System(.NET)**

**Scenario:**

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

### What I Learned:

I learned how to implement both linear and binary search algorithms in C# to find books by title. Linear search works on unsorted arrays, while binary search requires sorting and offers better performance for large datasets.

**CODE:**

using System;

class Book

{

public int BookId;

public string Title;

public string Author;

}

class Program6

{

static int LinearSearch(Book[] books, string title)

{

for (int i = 0; i < books.Length; i++)

if (books[i].Title == title) return i;

return -1;

}

static int BinarySearch(Book[] books, string title)

{

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

while (left <= right)

{

int mid = (left + right) / 2;

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

if (cmp == 0) return mid;

if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return -1;

}

static void Main()

{

Book[] books = {

new Book { BookId = 1, Title = "Algorithms", Author = "Cormen" },

new Book { BookId = 2, Title = "C#", Author = "Anders" },

new Book { BookId = 3, Title = "Java", Author = "Gosling" }

};

Console.WriteLine("Linear Search: " + LinearSearch(books, "Java"));

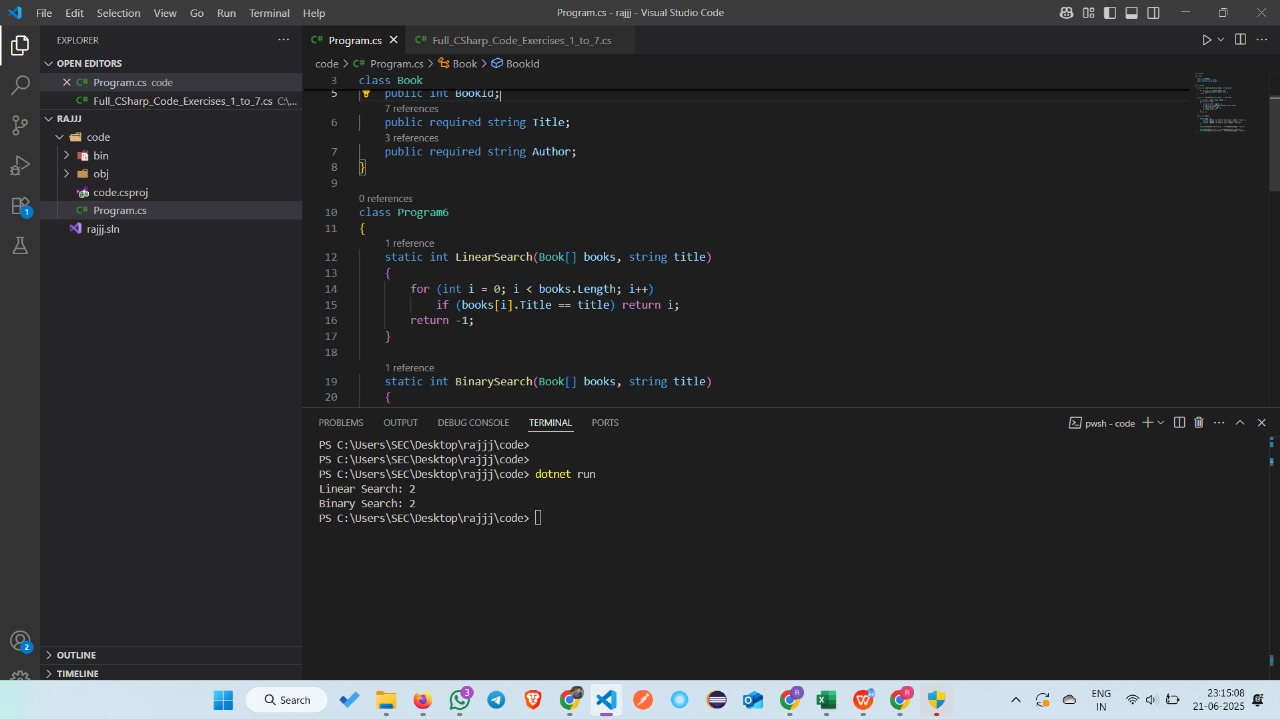
Array.Sort(books, (a, b) => string.Compare(a.Title, b.Title));

Console.WriteLine("Binary Search: " + BinarySearch(books, "Java"));

}

}

**OUTPUT:**



### Conclusion

This program demonstrates effective searching in book data using linear and binary search, highlighting the importance of sorting for efficient binary lookup.

**Exercise 7: Financial Forecasting(.NET)**

**Scenario:**

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

### What I Learned

I learned how to use both recursion and memoization to predict investment growth over time. Recursion recalculates values repeatedly, while memoization improves efficiency by storing intermediate results to avoid redundant computations.

**CODE:**

using System;

class Program7

{

static double PredictRecursive(int year, double rate)

{

if (year == 0) return 1000;

return PredictRecursive(year - 1, rate) \* (1 + rate);

}

static double PredictMemo(int year, double rate, double[] memo)

{

if (memo[year] != 0) return memo[year];

if (year == 0) return memo[year] = 1000;

return memo[year] = PredictMemo(year - 1, rate, memo) \* (1 + rate);

}

static void Main()

{

Console.WriteLine("Recursive Value: " + PredictRecursive(5, 0.1));

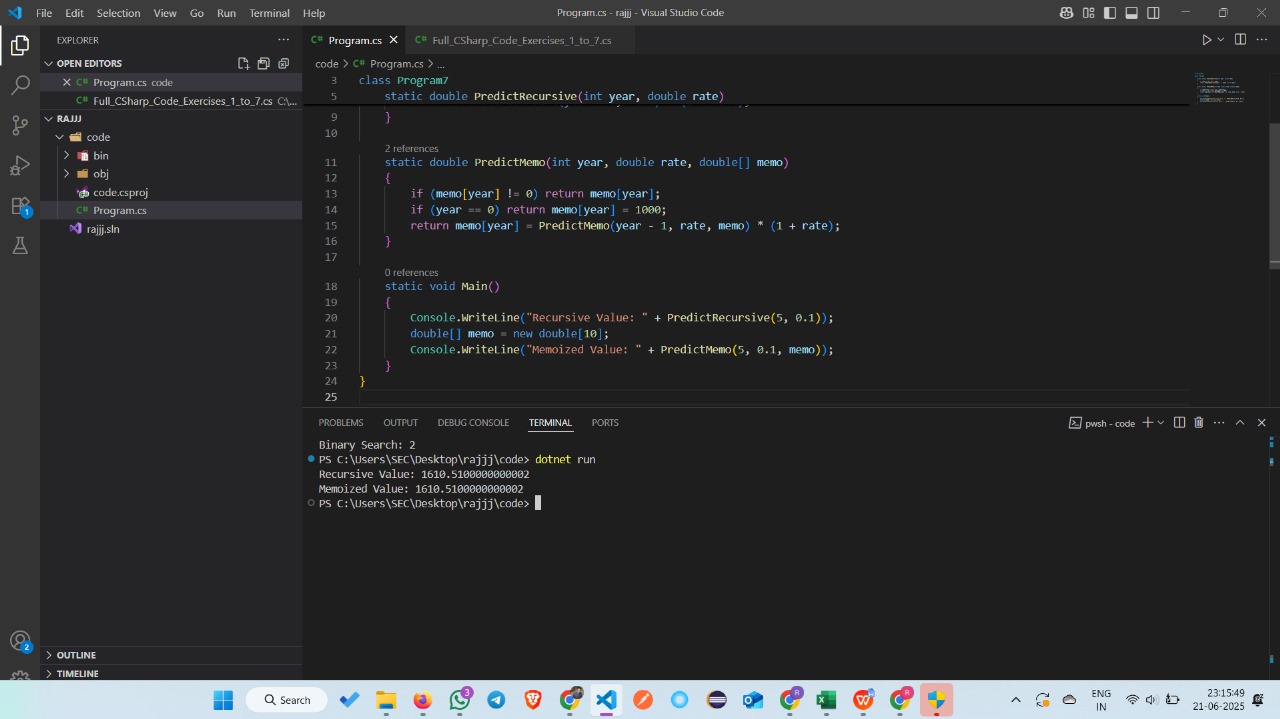
double[] memo = new double[10];

Console.WriteLine("Memoized Value: " + PredictMemo(5, 0.1, memo));

}

}

**OUTPUT:**



### Conclusion:

Memoization significantly reduces the number of recursive calls by caching results, making the investment prediction more efficient.

**Thank you**