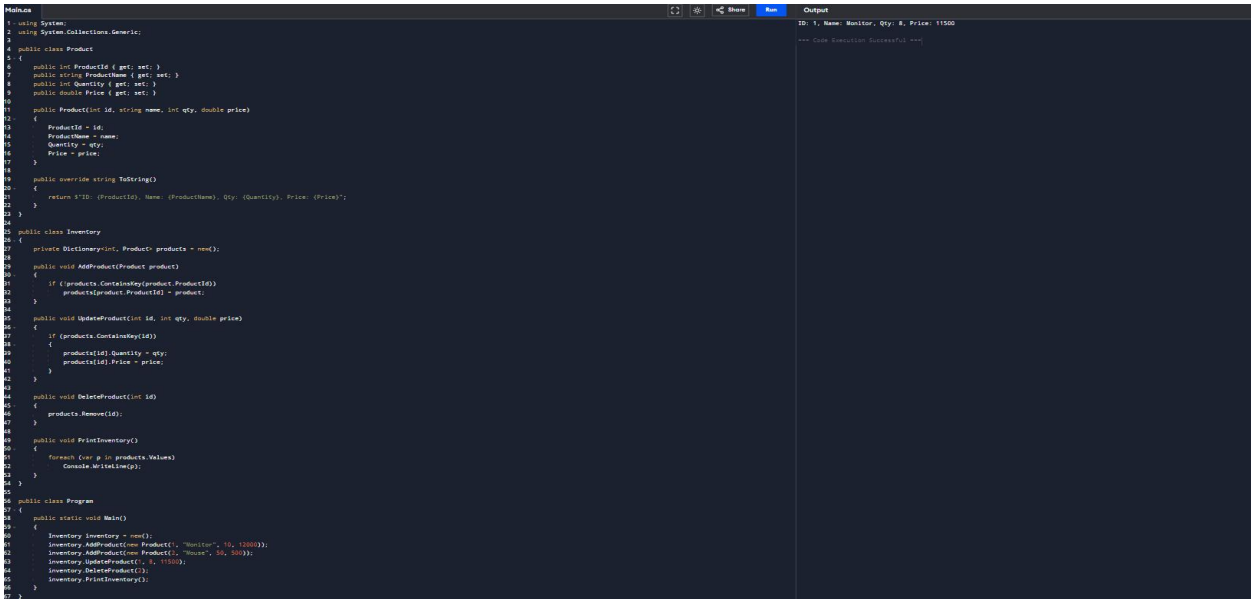
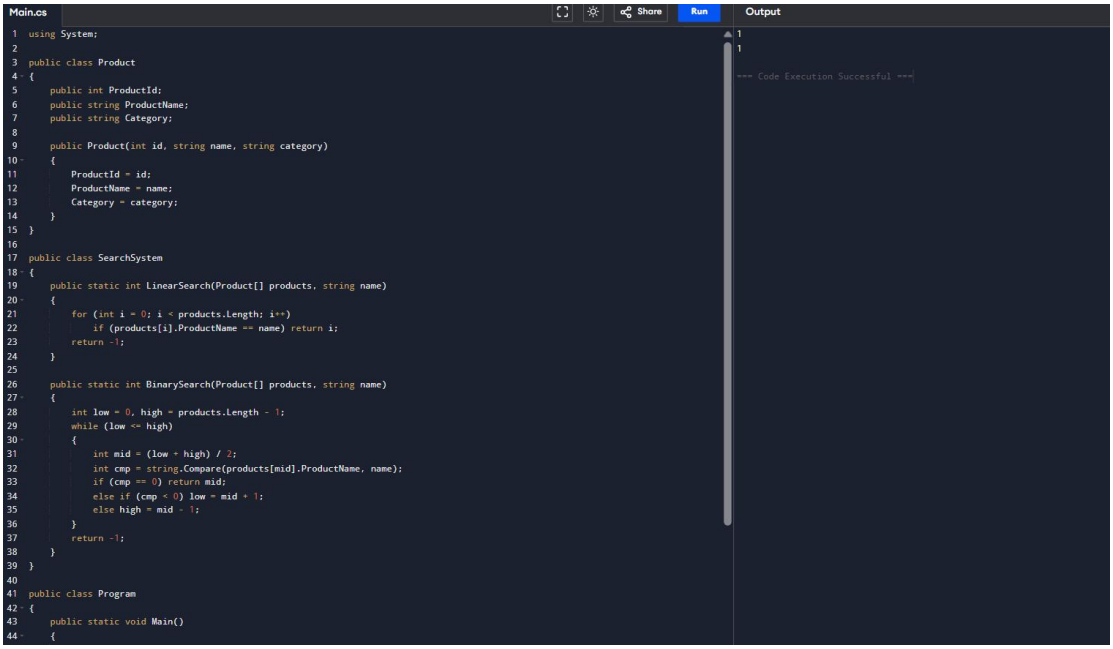
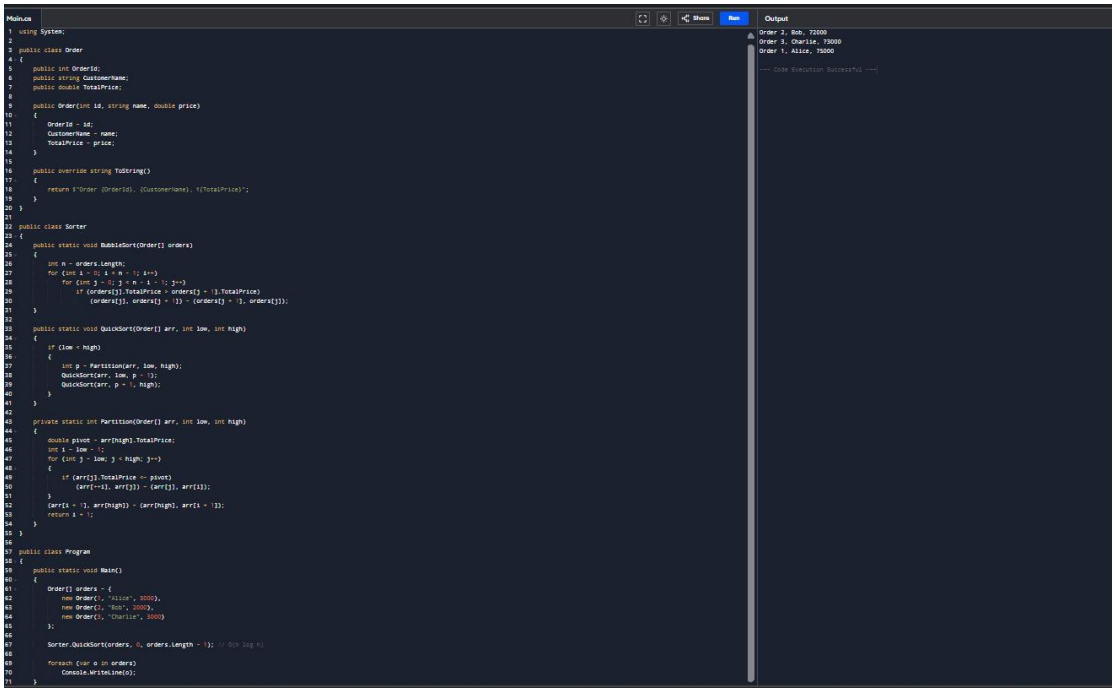
Exercise 1:



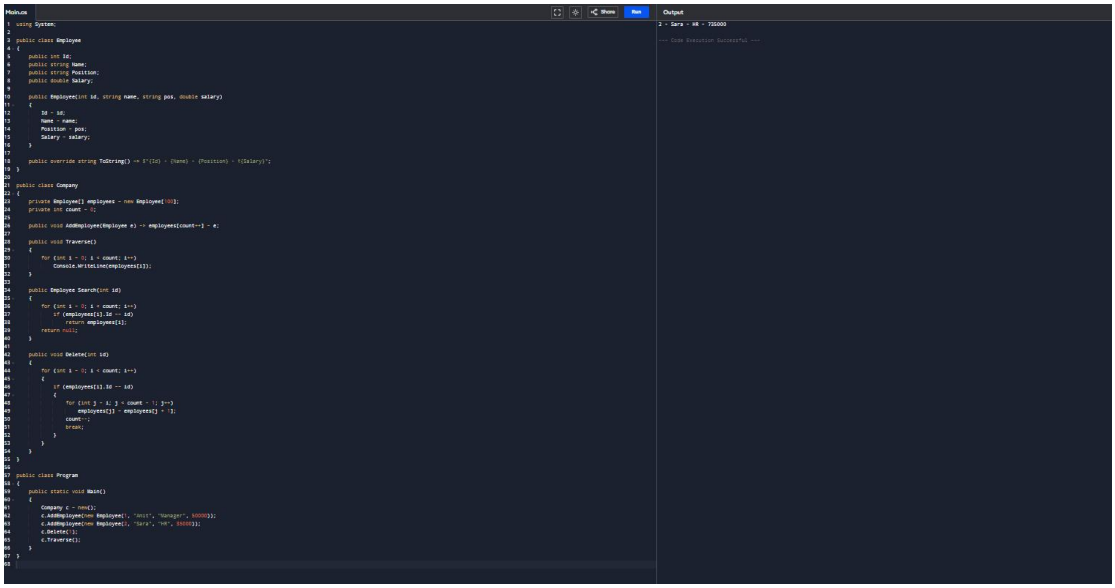
Exercise 2:



Exercise 3:



Exercise 4:



Exercise 5:

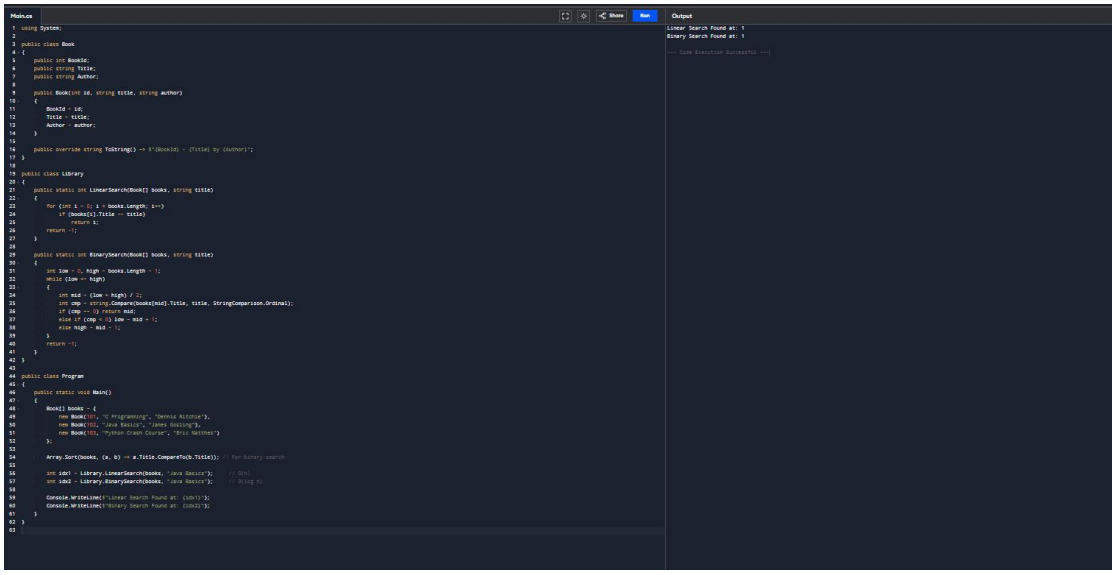
```
using System;

public class Task
{
    public int TaskId;
    public string TaskName;
    public string Status;
    public Task Next;

    public Task(int id, string name, string status)
    {
        TaskId = id;
        TaskName = name;
        Status = status;
        Next = null;
    }

    public override string ToString() => $"ID: {TaskId}, Name: {TaskName}, Status: {Status}";
}

public class TaskLinkedList
{
    private Task head;

    public void AddTask(Task task)
    {
        if (head == null)
        {
            head = task;
        }
        else
        {
            Task temp = head;
            while (temp.Next != null)
                temp = temp.Next;
            temp.Next = task;
        }
    }

    public Task SearchTask(int id)
    {
        Task temp = head;
        while (temp != null)
        {
            if (temp.TaskId == id)
                return temp;
            temp = temp.Next;
        }
        return null;
    }

    public void TraverseTasks()
    {
        Task temp = head;
        while (temp != null)
        {
```

```
            Console.WriteLine(temp);
            temp = temp.Next;
        }
    }

    public void DeleteTask(int id)
    {
        if (head == null) return;

        if (head.TaskId == id)
        {
            head = head.Next;
            return;
        }

        Task temp = head;
        while (temp.Next != null && temp.Next.TaskId != id)
            temp = temp.Next;

        if (temp.Next != null)
            temp.Next = temp.Next.Next;
    }
}

public class Program
{
    public static void Main()
    {
        TaskLinkedList list = new();
        list.AddTask(new Task(1, "Design UI", "Pending"));
        list.AddTask(new Task(2, "Fix Bugs", "In Progress"));
        list.TraverseTasks();

        Console.WriteLine("After Deleting Task 1:");
        list.DeleteTask(1);
        list.TraverseTasks();
    }
}
```

Output

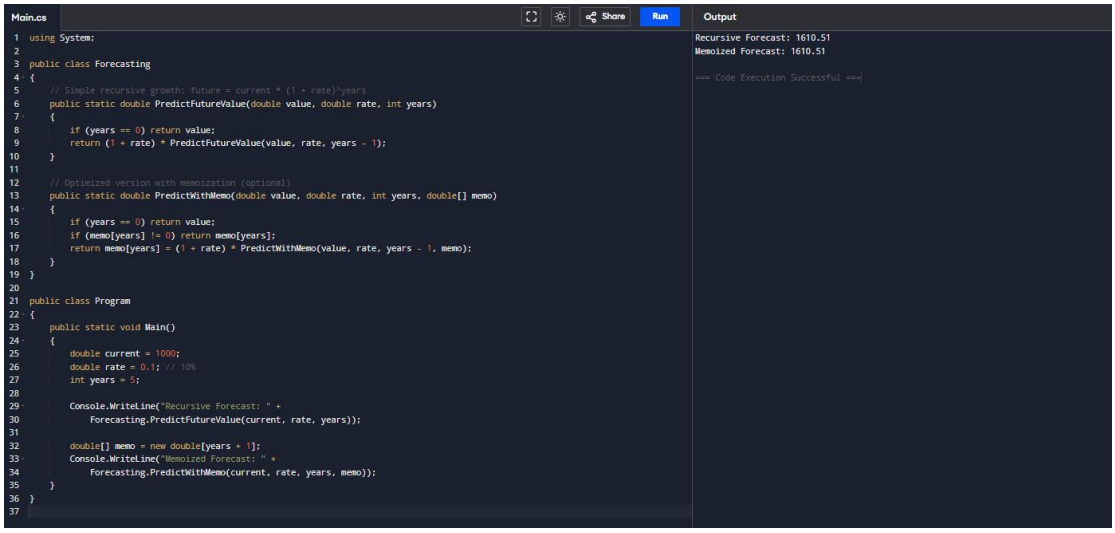
```
ID: 1, Name: Design UI, Status: Pending
ID: 2, Name: Fix Bugs, Status: In Progress
After Deleting Task 1:
ID: 2, Name: Fix Bugs, Status: In Progress

=== Code Execution Successful ===
```

Exercise 6:




```
using System;

public class Book
{
    public int BookId;
    public string Title;
    public string Author;

    public Book(int id, string title, string author)
    {
        BookId = id;
        Title = title;
        Author = author;
    }

    public override string ToString() => $"{BookId} - {Title} by {Author}";
}

public class Library
{
    public static int LinearSearch(Book[] books, string title)
    {
        for (int i = 0; i < books.Length; i++)
            if (books[i].Title == title)
                return i;
        return -1;
    }

    public static int BinarySearch(Book[] books, string title)
    {
        int low = 0, high = books.Length - 1;
        while (low <= high)
        {
            int mid = (low + high) / 2;
            int cmp = string.Compare(books[mid].Title, title, StringComparison.Ordinal);
            if (cmp == 0) return mid;
            else if (cmp < 0) low = mid + 1;
            else high = mid - 1;
        }
        return -1;
    }
}

public class Program
{
    public static void Main()
    {
        Book[] books = {
            new Book(101, "C Programming", "Dennis Ritchie"),
            new Book(102, "Java Basics", "James Gosling"),
            new Book(103, "Python Crash Course", "Eric Matthes")
        };

        Array.Sort(books, (a, b) => a.Title.CompareTo(b.Title));  // For binary search

        int idx1 = Library.LinearSearch(books, "Java Basics");   // O(n)
        int idx2 = Library.BinarySearch(books, "Java Basics");   // O(log n)

        Console.WriteLine($"Linear Search Found at: {idx1}");
        Console.WriteLine($"Binary Search Found at: {idx2}");
    }
}
```

Exercise 7:




```
using System;

public class Forecasting
{
    // Simple recursive growth: future = current * (1 + rate)^years
    public static double PredictFutureValue(double value, double rate, int years)
    {
        if (years == 0) return value;
        return (1 + rate) * PredictFutureValue(value, rate, years - 1);
    }

    // Optimized version with memoization (optional)
    public static double PredictWithMemo(double value, double rate, int years, double[] memo)
    {
        if (years == 0) return value;
        if (memo[years] != 0) return memo[years];
        return memo[years] = (1 + rate) * PredictWithMemo(value, rate, years - 1, memo);
    }
}

public class Program
{
    public static void Main()
    {
        double current = 1000;
        double rate = 0.1; // 10%
        int years = 5;

        Console.WriteLine("Recursive Forecast: " +
            Forecasting.PredictFutureValue(current, rate, years));

        double[] memo = new double[years + 1];
        Console.WriteLine("Memoized Forecast: " +
            Forecasting.PredictWithMemo(current, rate, years, memo));
    }
}
```