Advanced Debugging with Microsoft Copilot

Objective:

Use Microsoft Copilot to assist in identifying and fixing advanced issues in a code block. This task will allow you to utilize AI-assisted debugging to enhance your problem-solving capabilities, focusing on areas like optimizing loops, input validation, and edge-case handling.

Step-by-Step Code Construction:

1. Initial Review of the Code: Start by reviewing the provided code, which contains issues related to loop inefficiencies, incomplete input validation, and incorrect handling of edge cases. Before making any changes, analyze the code manually to identify potential problem areas.
2. Using Copilot to Identify Logical Errors:
   - Activate Microsoft Copilot and load the code into your IDE.
   - Ask Copilot for suggestions on improving the code, such as pointing out inefficient loops, validation gaps, or redundant operations.
   - Use Copilot's recommendations to pinpoint errors you may have missed, especially regarding how the loop processes each item in a list.
3. *Example prompt*: "Copilot, suggest improvements for this loop to handle all task scenarios."
4. Optimizing Loops with Copilot:
   - Once Copilot provides suggestions, focus on the loop structure.
   - Use Copilot to help you rewrite the loop, ensuring it properly handles all items in a list without unnecessary iterations.
   - Ensure that the loop now works efficiently, especially when dealing with empty lists or lists that are near the boundary of the program's constraints.
5. *Example prompt*: "How can I optimize this loop for better performance?"
6. Improving Input Validation: Input validation in this code can be made more robust with Copilot's assistance.

- Use Copilot to identify weak points in the current validation logic, especially handling inputs that are non-numeric, out-of-range, or edge cases like empty strings.
- Implement the improvements Copilot suggests and test how they handle various invalid input scenarios.

7. *Example prompt*: "Can you suggest how to strengthen this input validation block?"
8. Handling Edge Cases with Copilot's Suggestions: Often, code can break when handling unusual inputs or unexpected user actions (e.g., trying to mark a non-existent task). Use Copilot to recommend how to better manage these edge cases.
    - Ask Copilot to help you handle cases such as no tasks being present, marking a task that doesn't exist, or exiting the program unexpectedly.
    - Review and implement Copilot's suggestions, ensuring that the program gracefully handles all edge cases.
9. Final Testing and Validation: After implementing Copilot's suggestions, run multiple tests to ensure all edge cases and validation scenarios are handled correctly.
    - Test the program with valid and invalid inputs.
    - Ensure the loop runs efficiently for both small and large datasets.
    - Confirm that input validation prevents crashes or incorrect behavior.

Code:

```
class Program
{
    static void Main()
    {
        List<string> tasks = new List<string>();
        bool exit = false;

        while (!exit)
        {
            ShowMenu();
            Console.Write("Enter your choice: ");

            if (int.TryParse(Console.ReadLine(), out int choice))
            {
                switch (choice)
                {
                    case 1:
                        ViewTasks(tasks);
                        break;
                    case 2:
                        AddTask(tasks);
                        break;
                    case 3:
                        MarkTaskComplete(tasks);
                        break;
                    case 4:
                        exit = true;
                        Console.WriteLine("Goodbye!");
                        break;
                    default:
                        Console.WriteLine("Invalid option. Please try
again.");
                        break;
                }
            }
            else
            {
                Console.WriteLine("Please enter a valid number.");
            }

            Console.WriteLine();
        }
    }

    static void ShowMenu()
    {
        Console.WriteLine("=== TO-DO LIST MENU ===");
        Console.WriteLine("1. View Tasks");
        Console.WriteLine("2. Add Task");
        Console.WriteLine("3. Mark Task Complete");
        Console.WriteLine("4. Exit");
    }

    static void ViewTasks(List<string> tasks)
    {
        Console.WriteLine("\nYour Tasks:");

        if (tasks.Count == 0)
        {
            Console.WriteLine("No tasks available.");
        }
        else
        {
            for (int i = 0; i < tasks.Count; i++)
            {
                Console.WriteLine($"{i + 1}. {tasks[i]}");
            }
        }
    }

    static void AddTask(List<string> tasks)
    {
        Console.Write("Enter the task: ");
        string? input = Console.ReadLine();
        string task = input != null ? input.Trim() : string.Empty;

        if (string.IsNullOrEmpty(task))
        {
            Console.WriteLine("Task cannot be empty.");
        }
        else
        {
            tasks.Add(task);
            Console.WriteLine("Task added.");
        }
    }

    static void MarkTaskComplete(List<string> tasks)
    {
        if (tasks.Count == 0)
        {
            Console.WriteLine("No tasks to mark complete.");
            return;
        }

        Console.Write("Enter the task number to mark complete: ");
        if (int.TryParse(Console.ReadLine(), out int taskNumber) &&
            taskNumber > 0 && taskNumber <= tasks.Count)
        {
            string currentTask = tasks[taskNumber - 1];
            if (currentTask.EndsWith(" [Complete]"))
            {
                Console.WriteLine("This task is already marked as
complete.");
            }
            else
            {
                tasks[taskNumber - 1] += " [Complete]";
                Console.WriteLine("Task marked as complete.");
            }
        }
        else
        {
            Console.WriteLine("Invalid task number.");
        }
    }
}
```