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

**Understanding Asymptotic Notation**

**Explain Big O notation and how it helps in analyzing algorithms.**

ANS:

Big O Notation: Big O notation is a mathematical representation used to describe the upper bound of the time complexity of an algorithm. It provides a high-level understanding of the algorithm's performance in terms of input size . The notation expresses how the runtime or space requirements grow as the input size increases.

For example:

O(1): Constant time - the algorithm's performance is not affected by the size of the input.

O(n): Linear time - the performance grows linearly with the input size.

O(logn): Logarithmic time - the performance grows logarithmically as the input size increases.

**Describe the best, average, and worst-case scenarios for search operations.**

ANS:

***Linear Search***

Linear Search is a simple search algorithm that checks each element in the array sequentially until the target element is found or the end of the array is reached.

1. **Best Case:**

Scenario: The target element is the first element in the array.

Operations: The algorithm performs only one comparison.

Time Complexity: O(1)

**2.Average Case:**

Scenario: The target element is located somewhere in the middle of the array.

Operations: On average, the algorithm will check half of the elements in the array.

Time Complexity: O(n) (where n is the number of elements in the array)

**3.Worst Case:**

Scenario: The target element is the last element in the array or is not present at all.

Operations: The algorithm checks all elements in the array.

Time Complexity: O(n)

***Binary Search***

Binary Search is a more efficient search algorithm that works on sorted arrays. It repeatedly divides the search interval in half.

**1.Best Case:**

Scenario: The target element is the middle element of the array.

Operations: The algorithm finds the target in the first comparison.

Time Complexity: O(1)

**2.Average Case:**

Scenario: The target element is located somewhere in the array, not necessarily in the middle.

Operations: The algorithm will perform a logarithmic number of comparisons, as it halves the search space with each step.

Time Complexity: O(logn)

**3.Worst Case:**

Scenario: The target element is not present in the array or is located at one of the ends.

Operations: The algorithm will continue halving the search space until it narrows down to a single element.

Time Complexity: O(logn)

**CODE:**

using System;

public class Program

{

public static void Main(string[] args)

{

Product[] products = new Product[] {

new Product(1, "Redmi Note 7 pro", "Electronics"),

new Product(2, "ClassMate 160 pages Notebook", "Stationary"),

new Product(3, "One Plus Ear Buds", "Electronics"),

new Product(4, "Red Silk Curtain", "Home Decorations"),

new Product(5, "Boat Rockerz 255 Pro+", "Electronics"),

new Product(6, "Faber-Castell Color Pencils", "Stationary"),

new Product(7, "Samsung Galaxy S22", "Electronics"),

new Product(8, "Ikea Wooden Study Table", "Furniture"),

new Product(9, "Usha Mist Air Fan", "Home Appliances"),

new Product(10, "Raymond Men’s Formal Shirt", "Clothing"),

new Product(11, "Dell Inspiron 15 Laptop", "Electronics"),

new Product(12, "Havells LED Bulb 9W", "Home Appliances"),

new Product(13, "Parker Ink Pen", "Stationary"),

new Product(14, "Bombay Dyeing Bedsheet - King Size", "Home Decorations"),

new Product(15, "Nike Revolution 6 Shoes", "Footwear"),

new Product(16, "Wildcraft Backpack 35L", "Accessories"),

new Product(17, "Canon Pixma Inkjet Printer", "Electronics"),

new Product(18, "Prestige Pressure Cooker 5L", "Kitchen Appliances"),

new Product(19, "Adidas Sports T-shirt", "Clothing"),

new Product(20, "Philips Beard Trimmer", "Personal Care"),

new Product(21, "Redmi Note 8 pro", "Electronics"),

new Product(22, "Redmi Note 9 pro", "Electronics"),

};

Console.WriteLine("Welcome to E-Commerce Shopping Platform");

Console.WriteLine("1. Search By Product Name (partial match)");

Console.WriteLine("2. Search for Product ID");

Console.WriteLine("3. Search for Products in a Category (partial match)");

int choice;

if (int.TryParse(Console.ReadLine(), out choice))

{

if (choice == 1)

{

Console.WriteLine("Enter name of the product:");

string input = Console.ReadLine();

SearchByNamePartial(products, input);

}

else if (choice == 2)

{

Console.WriteLine("Enter the ID of the product:");

string idInput = Console.ReadLine();

int id;

if (int.TryParse(idInput, out id))

{

BinarySearchByID(products, id);

}

else

{

Console.WriteLine("Invalid product ID entered.");

}

}

else if (choice == 3)

{

Console.WriteLine("Enter the name of the Category:");

string categoryName = Console.ReadLine();

SearchByCategoryPartial(products, categoryName);

}

else

{

Console.WriteLine("Invalid choice. Please select 1, 2, or 3.");

}

}

else

{

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

}

Console.WriteLine("Press Enter to exit...");

Console.ReadLine();

}

public static void SearchByNamePartial(Product[] products, string namePart)

{

bool found = false;

foreach (var product in products)

{

if (product.productName.IndexOf(namePart, StringComparison.OrdinalIgnoreCase) >= 0)

{

Console.WriteLine($"Product: {product.productName} (ID: {product.productID}, Category: {product.Category}) is Available");

found = true;

}

}

if (!found)

{

Console.WriteLine("No matching products found.");

}

}

public static void SearchByCategoryPartial(Product[] products, string categoryPart)

{

bool found = false;

foreach (var product in products)

{

if (product.Category.IndexOf(categoryPart, StringComparison.OrdinalIgnoreCase) >= 0)

{

Console.WriteLine($"Product: {product.productName} (ID: {product.productID}, Category: {product.Category}) is Available");

found = true;

}

}

if (!found)

{

Console.WriteLine("No products found in this category.");

}

}

public static void BinarySearchByID(Product[] products, int id)

{

Array.Sort(products, (a, b) => a.productID.CompareTo(b.productID));

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

int mid = left + (right - left) / 2;

if (products[mid].productID == id)

{

Console.WriteLine($"Product: {products[mid].productName} (ID: {products[mid].productID}, Category: {products[mid].Category}) is Available");

return;

}

else if (products[mid].productID < id)

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

Console.WriteLine("Product Not Available");

}

}

public class Product

{

public int productID { get; set; }

public string productName { get; set; }

public string Category { get; set; }

public Product(int productID, string productName, string category)

{

this.productID = productID;

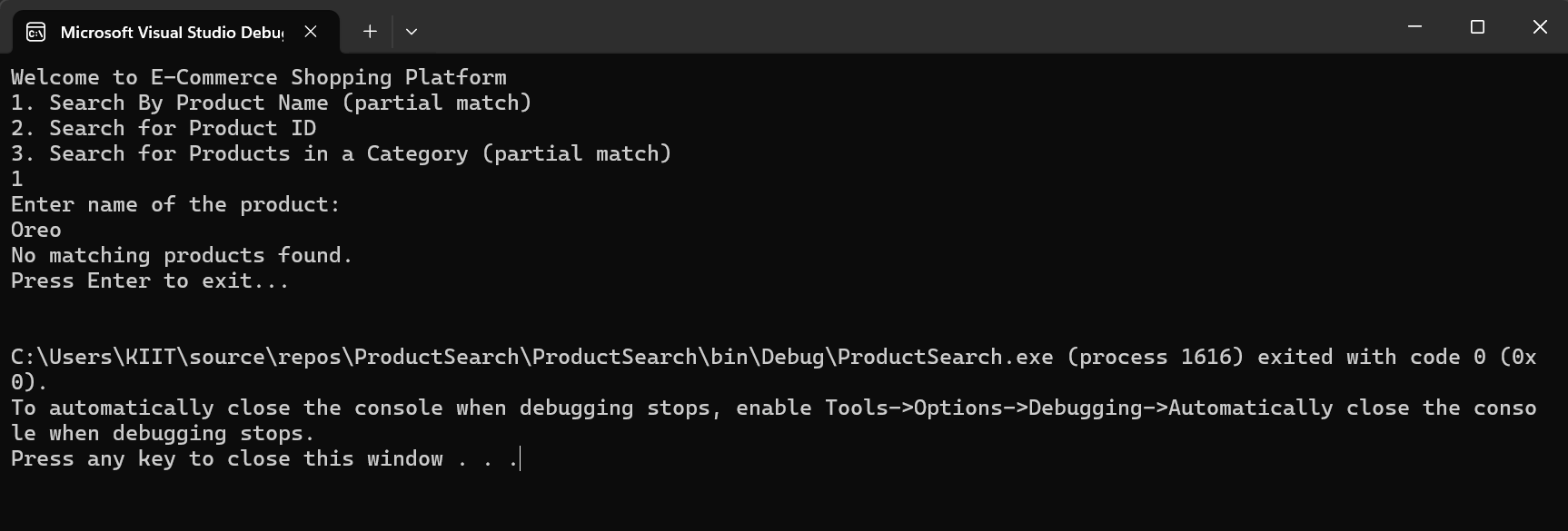
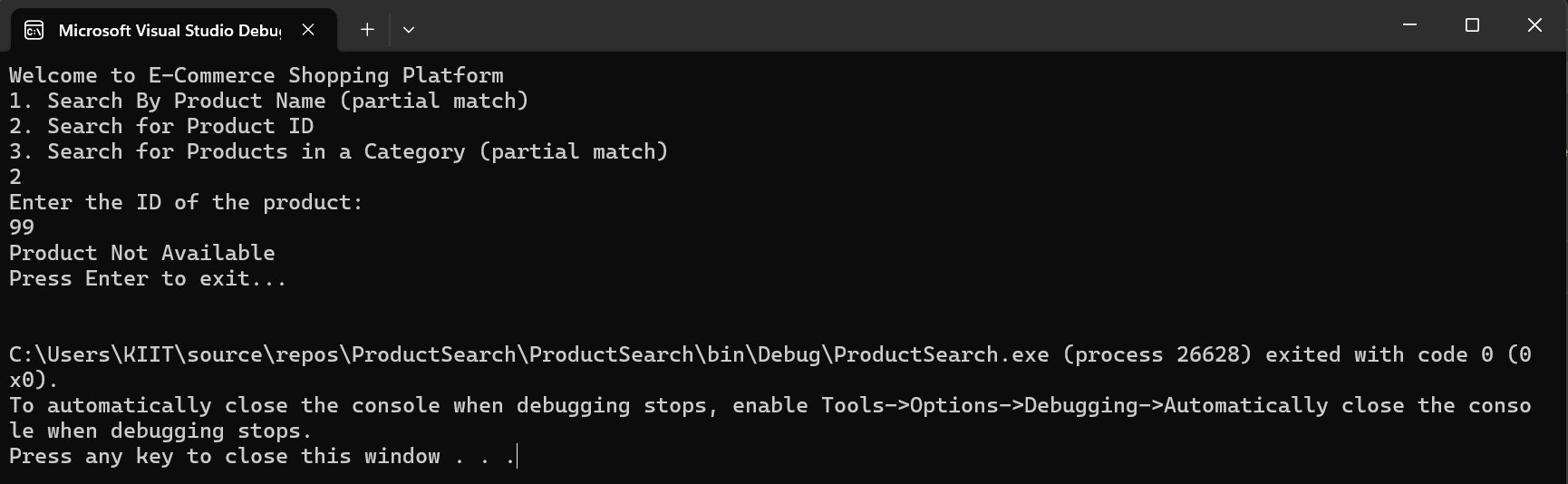
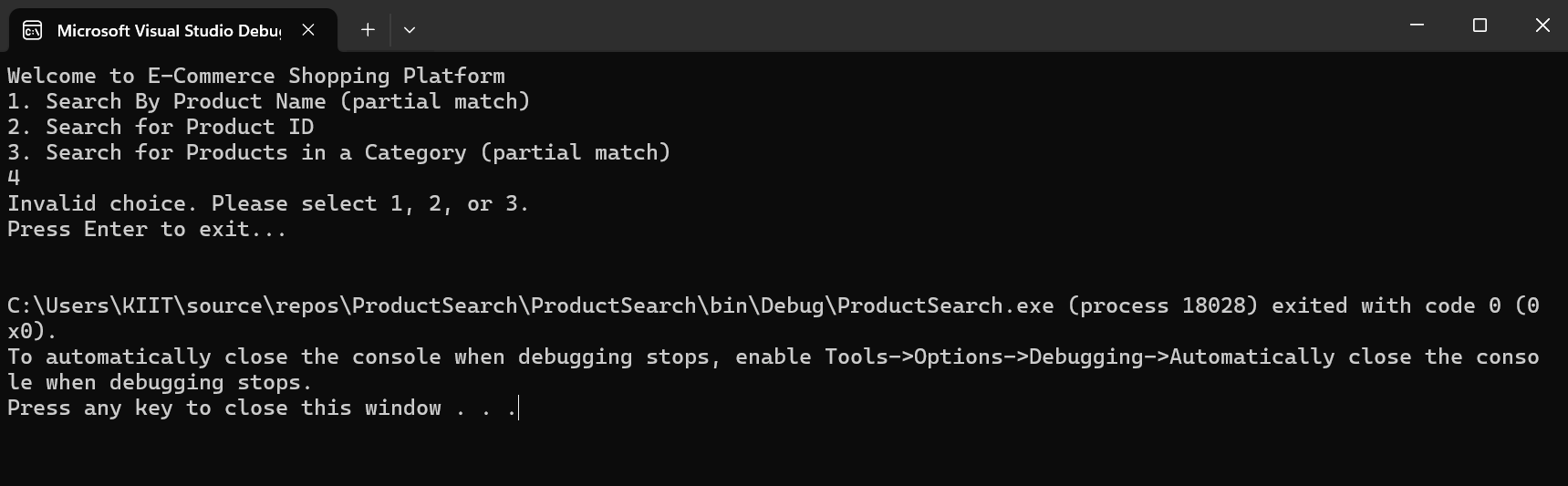
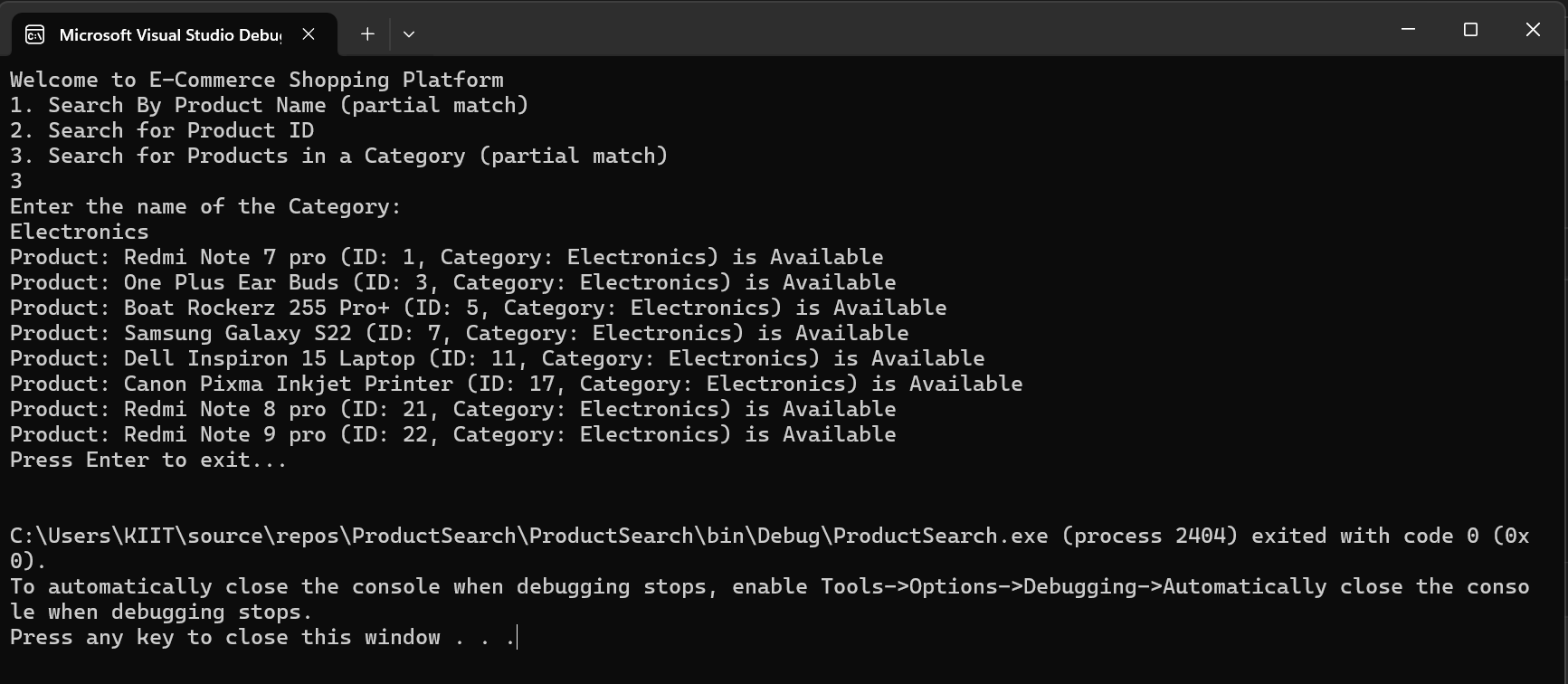
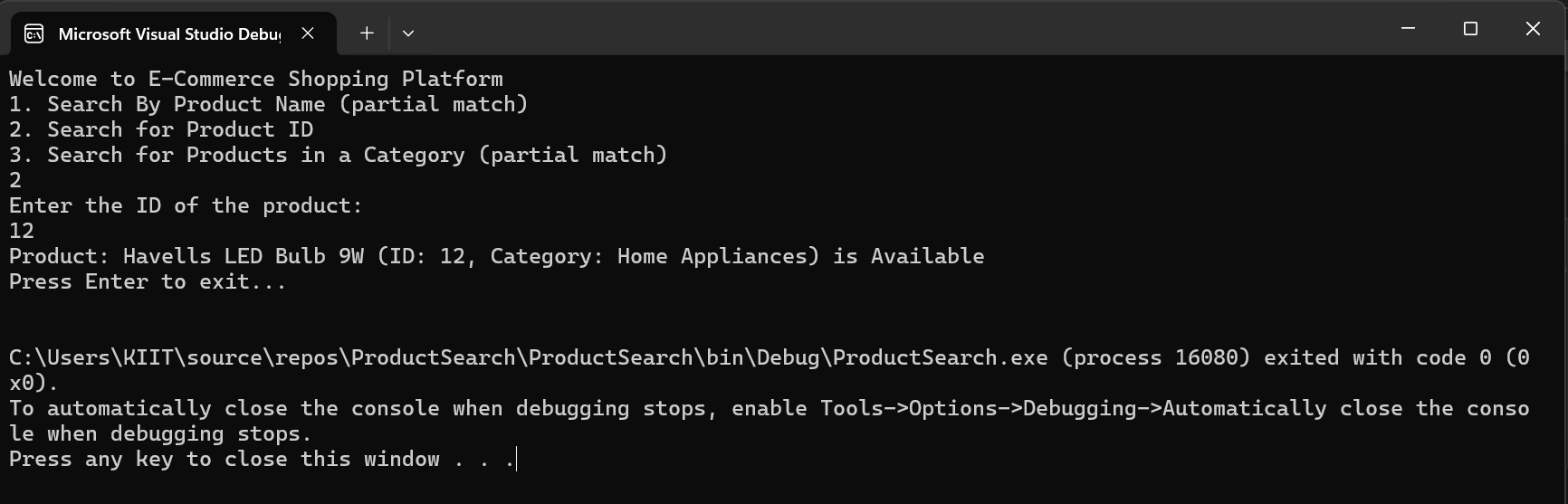
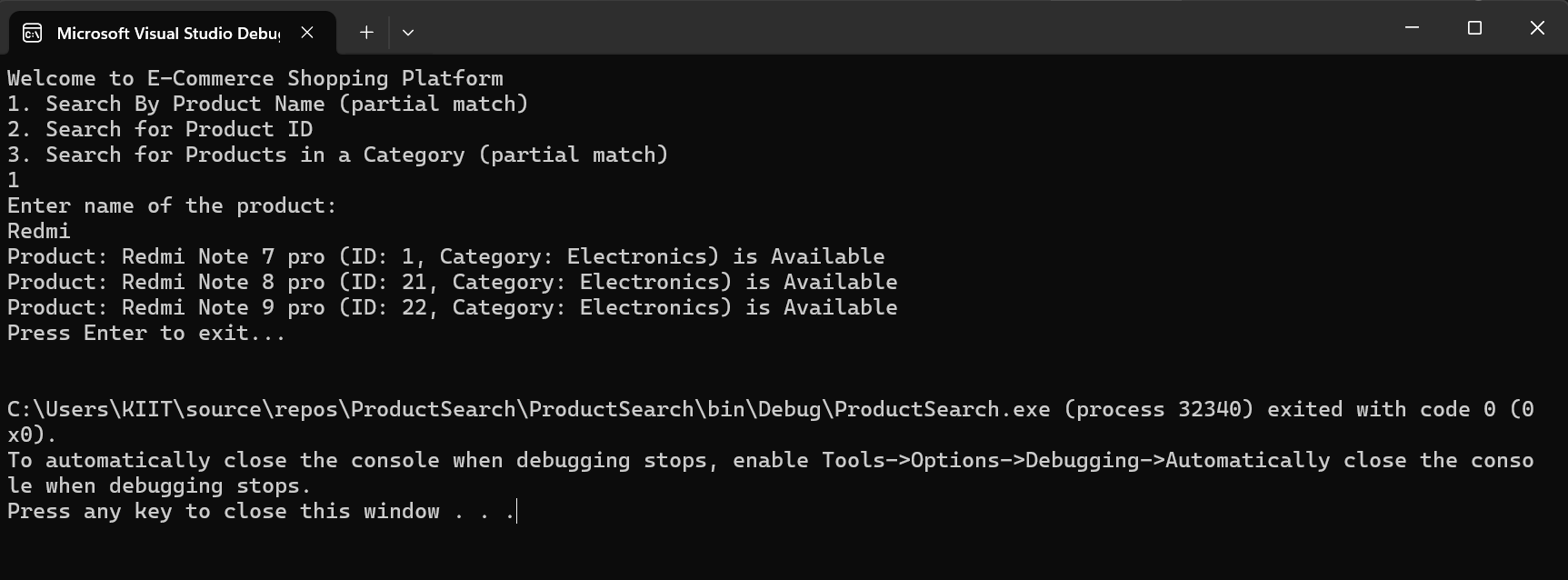
this.productName = productName;

this.Category = category;

}

}

**OUTPUT:**

****

**ANALYSIS:**

1. **Compare the time complexity of linear and binary search algorithms**

ANS: Linear Search is the simplest search algorithm. It goes through each element in a list one by one until it finds the desired item or reaches the end. Its time complexity in the best case is O(1) if the target element is at the very beginning. In the worst case, it is O(n), where n is the number of elements in the list — meaning every element may need to be checked. The average case is also O(n/2), which simplifies to O(n) in asymptotic notation.

Binary Search, on the other hand, is a much more efficient algorithm, but it works only on sorted data. It repeatedly divides the search interval in half. If the value of the search key is less than the middle element, the algorithm continues in the left half; otherwise, it goes to the right half. The time complexity of binary search in the best case is O(1) (if the middle element is the match), and in both the average and worst case, it is O(log n). This is significantly faster than linear search for large datasets.

1. **Discuss which algorithm is more suitable for your platform and why.**

ANS: In my e-commerce application, linear search is more suitable for searching by product name and category, because users often enter partial keywords or browse through multiple items. Linear search allows flexible, case-insensitive, and partial matches, which is ideal for such scenarios where the data isn’t sorted.

On the other hand, binary search is best for product ID searches. Since product IDs are unique and can be sorted easily, binary search provides much faster and more efficient lookups, especially as the number of products grows.

**Exercise 7: Financial Forecasting**

**Understand Recursive Algorithms:**

**Recursion:** Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It simplifies complex problems by breaking them down into simpler subproblems. Each recursive call typically works towards a base case, which is a condition that stops the recursion

**CODE:**

using System;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter The Present Sum:");

int presentSum = int.Parse(Console.ReadLine());

Console.WriteLine("Enter the rate in percent (e.g., 5 for 5%):");

double rate = double.Parse(Console.ReadLine()) / 100;

Console.WriteLine("Enter the Time Period (in years):");

int time = int.Parse(Console.ReadLine());

double futureValue = Forecasting(presentSum, rate, time);

Console.WriteLine($"Forecasted Future Value after {time} years: Rs{futureValue:F2}");

}

public static double Forecasting(double presentSum, double rate, int time)

{

if (time == 0)

{

return presentSum;

}

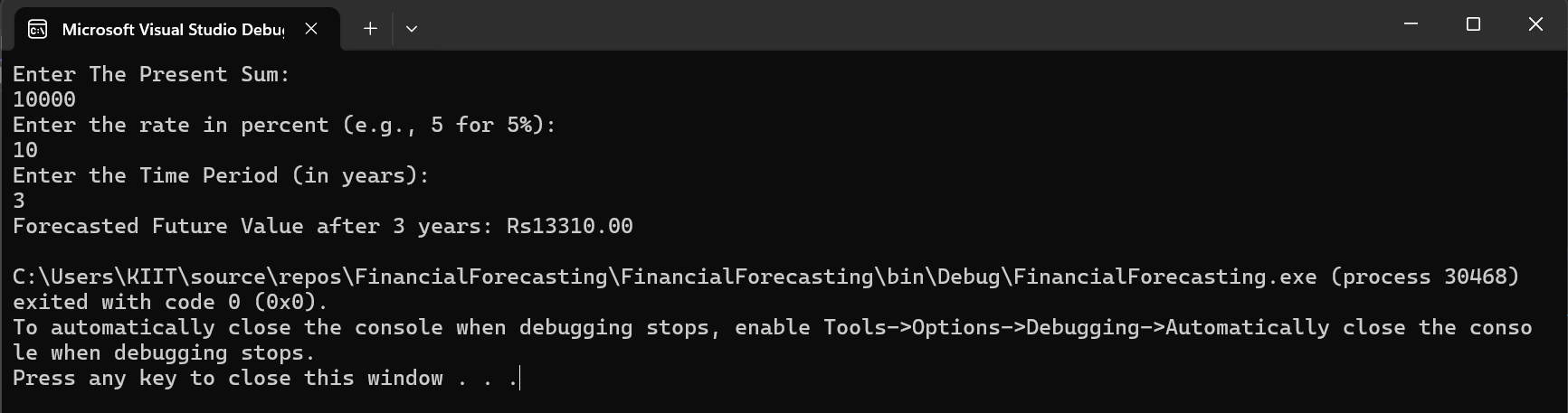
return Forecasting(presentSum \* (1 + rate), rate, time - 1);

}

}

}

**OUTPUT:**



**ANALYSIS**

**Discuss the time complexity of your recursive algorithm.**

ANS: Time Complexity of the Recursive Algorithm

The recursive forecasting method calls itself once for each year. So, if you are forecasting for n years, the function makes n recursive calls.

Thus, the time complexity is:

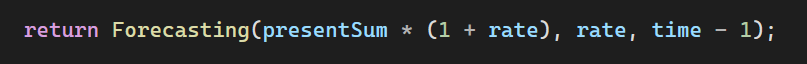
O(n) — linear time complexity, where n is the number of years.

This is because each recursive call performs a simple multiplication and then moves to the next year until the base case (year = 0) is reached.

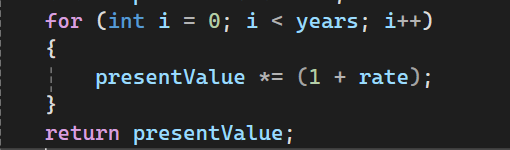
**Explain how to optimize the recursive solution to avoid excessive computation.**

ANS: To avoid excessive computation and memory usage, the recursive solution can be **converted into an iterative solution**. Iteration eliminates the need for multiple stack frames, making the program more efficient and safer for large inputs.

Instead of:



We can use:



This improves:

Space complexity from O(n) to O(1)

Avoids recursion depth limits

Performs the same logic in a more efficient way