## ****Exercise 2: E-Commerce Platform Search Function****

### ****Understanding Big O Notation****

**Big O Notation** is a theoretical framework used to describe the time or space complexity of an algorithm in terms of the size of the input. It highlights how efficiently an algorithm scales and helps compare multiple solutions without executing the code.

**Key Examples:**

**O(1)** – Constant time: Execution time is fixed regardless of input size.

**O(n)** – Linear time: Execution time grows in direct proportion to input size.

**O(log n)** – Logarithmic time: Execution time increases slowly as input size grows.

### ****Best, Average, and Worst-Case Scenarios in Searching****

#### ****Linear Search****

A straightforward method where each item in the list is checked one by one.

**Best Case:** Target is found at the first position.  
⮞ **Time Complexity:** O(1)

**Average Case:** Target appears midway through the list.  
⮞ **Time Complexity:** O(n)

**Worst Case:** Target is at the end or missing entirely.  
⮞ **Time Complexity:** O(n)

#### ****Binary Search****

A more optimized method that requires the input to be sorted. The list is divided into halves repeatedly.

**Best Case:** Match found at the central index.  
⮞ **Time Complexity:** O(1)

**Average Case:** Several divisions needed to locate the element.  
⮞ **Time Complexity:** O(log n)

**Worst Case:** Element is missing or found after complete division.  
⮞ **Time Complexity:** O(log n)

### ****C# Implementation: E-Commerce Search****

using System;

public class Program

{

public static void Main(string[] args)

{

Product[] inventory = {

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

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

// ... other products ...

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

};

Console.WriteLine("Welcome to Our E-Commerce Search Engine");

Console.WriteLine("1. Find Product by Name");

Console.WriteLine("2. Locate Product by ID");

Console.WriteLine("3. Search Products by Category");

int option;

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

{

switch (option)

{

case 1:

Console.WriteLine("Enter product keyword:");

SearchByName(inventory, Console.ReadLine());

break;

case 2:

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

int id;

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

BinarySearchByID(inventory, id);

else

Console.WriteLine("Invalid ID.");

break;

case 3:

Console.WriteLine("Enter category name:");

SearchByCategory(inventory, Console.ReadLine());

break;

default:

Console.WriteLine("Invalid selection.");

break;

}

}

}

static void SearchByName(Product[] list, string keyword)

{

bool match = false;

foreach (var item in list)

{

if (item.productName.IndexOf(keyword, StringComparison.OrdinalIgnoreCase) >= 0)

{

Console.WriteLine($"{item.productName} (ID: {item.productID}) in {item.Category}");

match = true;

}

}

if (!match)

Console.WriteLine("No product matches found.");

}

static void SearchByCategory(Product[] list, string category)

{

bool match = false;

foreach (var item in list)

{

if (item.Category.IndexOf(category, StringComparison.OrdinalIgnoreCase) >= 0)

{

Console.WriteLine($"{item.productName} (ID: {item.productID}) in {item.Category}");

match = true;

}

}

if (!match)

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

}

static void BinarySearchByID(Product[] list, int id)

{

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

int low = 0, high = list.Length - 1;

while (low <= high)

{

int mid = (low + high) / 2;

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

{

Console.WriteLine($"{list[mid].productName} (ID: {list[mid].productID}) in {list[mid].Category}");

return;

}

if (list[mid].productID < id)

low = mid + 1;

else

high = mid - 1;

}

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

}

}

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;

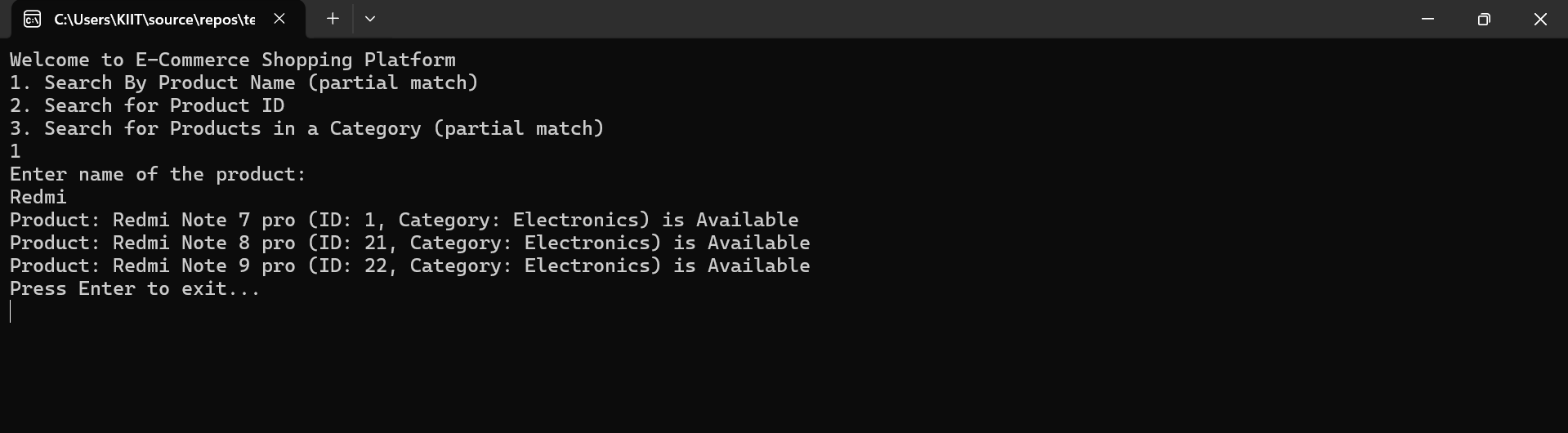
}

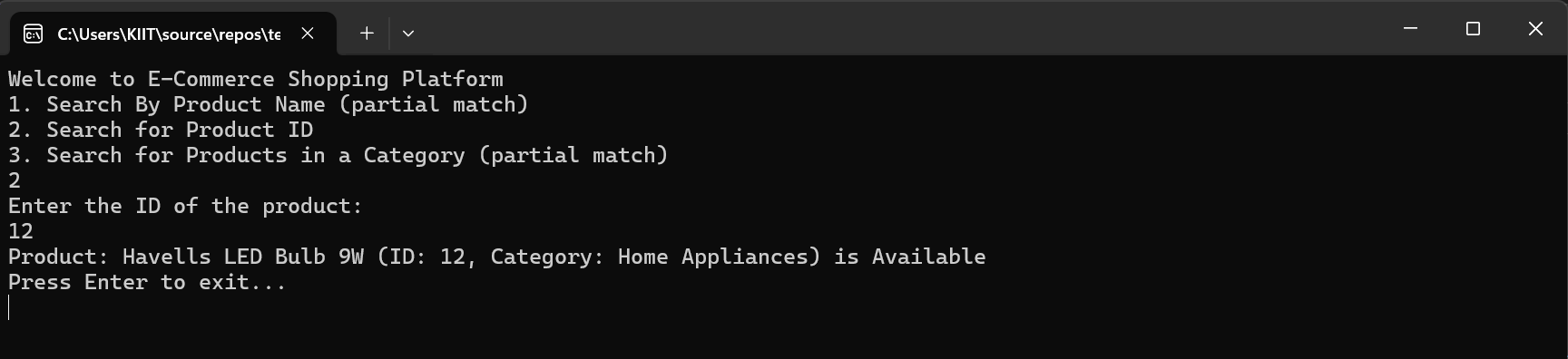
}

### ****Comparative Time Complexity Analysis****

**Linear Search:**  
✔ Simple but inefficient for large datasets.  
❌ O(n) in both average and worst cases.

**Binary Search:**  
✔ Highly efficient for sorted data like product IDs.  
❌ O(log n) time, but requires sorting beforehand.





**Conclusion:**  
For flexible keyword searches (product name/category), linear search is better. For fast lookup by ID, binary search is optimal.

## ****Exercise 7: Forecasting Future Value (Recursion)****

### ****Understanding Recursion****

Recursion involves a function repeatedly calling itself to solve smaller sub-problems until a base condition is met. This approach is particularly useful for problems that have a naturally recursive structure like compound interest forecasting.

### ****C# Implementation: Recursive Forecasting****

using System;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter present amount:");

double amount = double.Parse(Console.ReadLine());

Console.WriteLine("Enter annual interest rate (%):");

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

Console.WriteLine("Enter forecast period (years):");

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

double future = Forecast(amount, rate, years);

Console.WriteLine($"Future value after {years} years: Rs {future:F2}");

}

static double Forecast(double amount, double rate, int years)

{

if (years == 0)

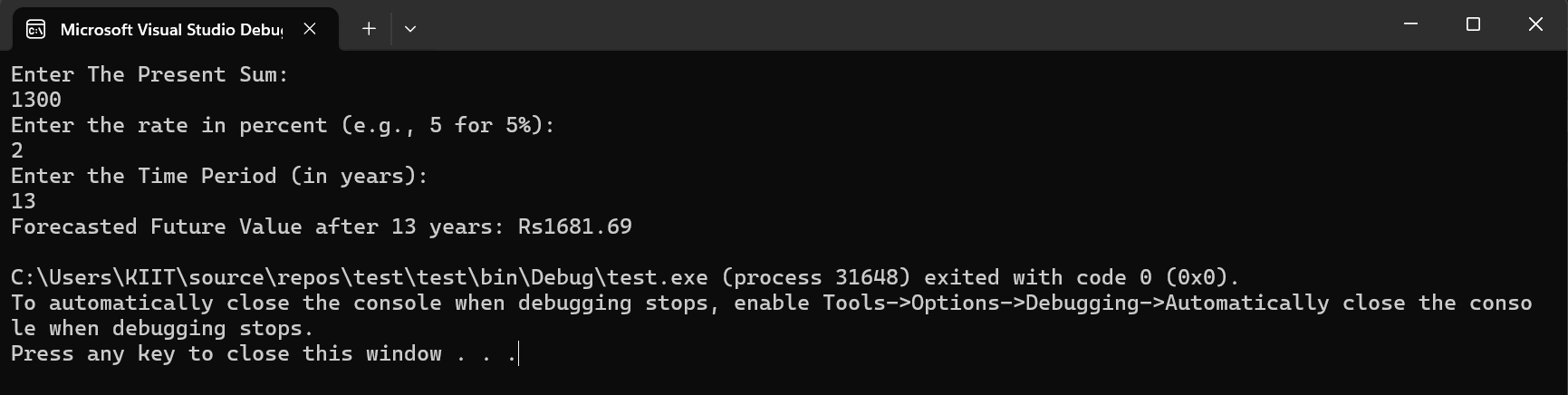
return amount;

return Forecast(amount \* (1 + rate), rate, years - 1);

}

}

}



### ****Complexity and Optimization****

**Time Complexity:**  
The recursive solution performs one function call per year, resulting in **O(n)** time complexity, where n is the number of years.

**Optimization Recommendation:**  
Recursion can be converted into a loop to make it more efficient in terms of space and avoid stack overflow errors:

static double ForecastIterative(double amount, double rate, int years)

{

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

amount \*= (1 + rate);

return amount;

}

✅ Space complexity improves from **O(n)** to **O(1)**

✅ Handles large input sizes safely

✅ Execution is generally faster due to the elimination of recursive overhead