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

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

Product.cs

namespace EcommerceSearchApp

{

public class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public string Category { get; set; }

public Product() { }

public Product(int id, string name, string category)

{

ProductId = id;

ProductName = name;

Category = category;

}

public override string ToString()

{

return $"[{ProductId}] {ProductName} - {Category}";

}

}

}

Program.cs

// See https://aka.ms/new-console-template for more information

// Console.WriteLine("Hello, World!");

using System;

using System.IO;

using System.Linq;

using System.Text.Json;

using System.Collections.Generic;

namespace EcommerceSearchApp

{

class Program

{

static void Main(string[] args)

{

string filePath = "products.json";

// Load products from JSON file

Product[] products = LoadProductsFromJson(filePath);

if (products.Length == 0)

{

Console.WriteLine("No products found in JSON file.");

return;

}

Console.Write("Enter Product ID to search: ");

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

Console.WriteLine("\n--- Linear Search ---");

Product foundLinear = LinearSearch(products, searchId);

Console.WriteLine(foundLinear != null ? foundLinear.ToString() : "Product not found");

Console.WriteLine("\n--- Binary Search ---");

var sortedProducts = products.OrderBy(p => p.ProductId).ToArray();

Product foundBinary = BinarySearch(sortedProducts, searchId, 0, sortedProducts.Length - 1);

Console.WriteLine(foundBinary != null ? foundBinary.ToString() : "Product not found");

}

// Place this method here (inside Program class, outside Main)

static Product[] LoadProductsFromJson(string path)

{

if (!File.Exists(path))

return Array.Empty<Product>();

string json = File.ReadAllText(path);

return JsonSerializer.Deserialize<Product[]>(json) ?? Array.Empty<Product>();

}

static Product LinearSearch(Product[] products, int productId)

{

foreach (var product in products)

{

if (product.ProductId == productId)

return product;

}

return null;

}

static Product BinarySearch(Product[] products, int productId, int left, int right)

{

while (left <= right)

{

int mid = (left + right) / 2;

if (products[mid].ProductId == productId)

return products[mid];

else if (products[mid].ProductId < productId)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

}

products.json

[

{

"ProductId": 101,

"ProductName": "iPhone 14",

"Category": "Electronics"

},

{

"ProductId": 102,

"ProductName": "Samsung TV",

"Category": "Electronics"

},

{

"ProductId": 103,

"ProductName": "Nike Shoes",

"Category": "Footwear"

},

{

"ProductId": 104,

"ProductName": "Dell Laptop",

"Category": "Computers"

},

{

"ProductId": 105,

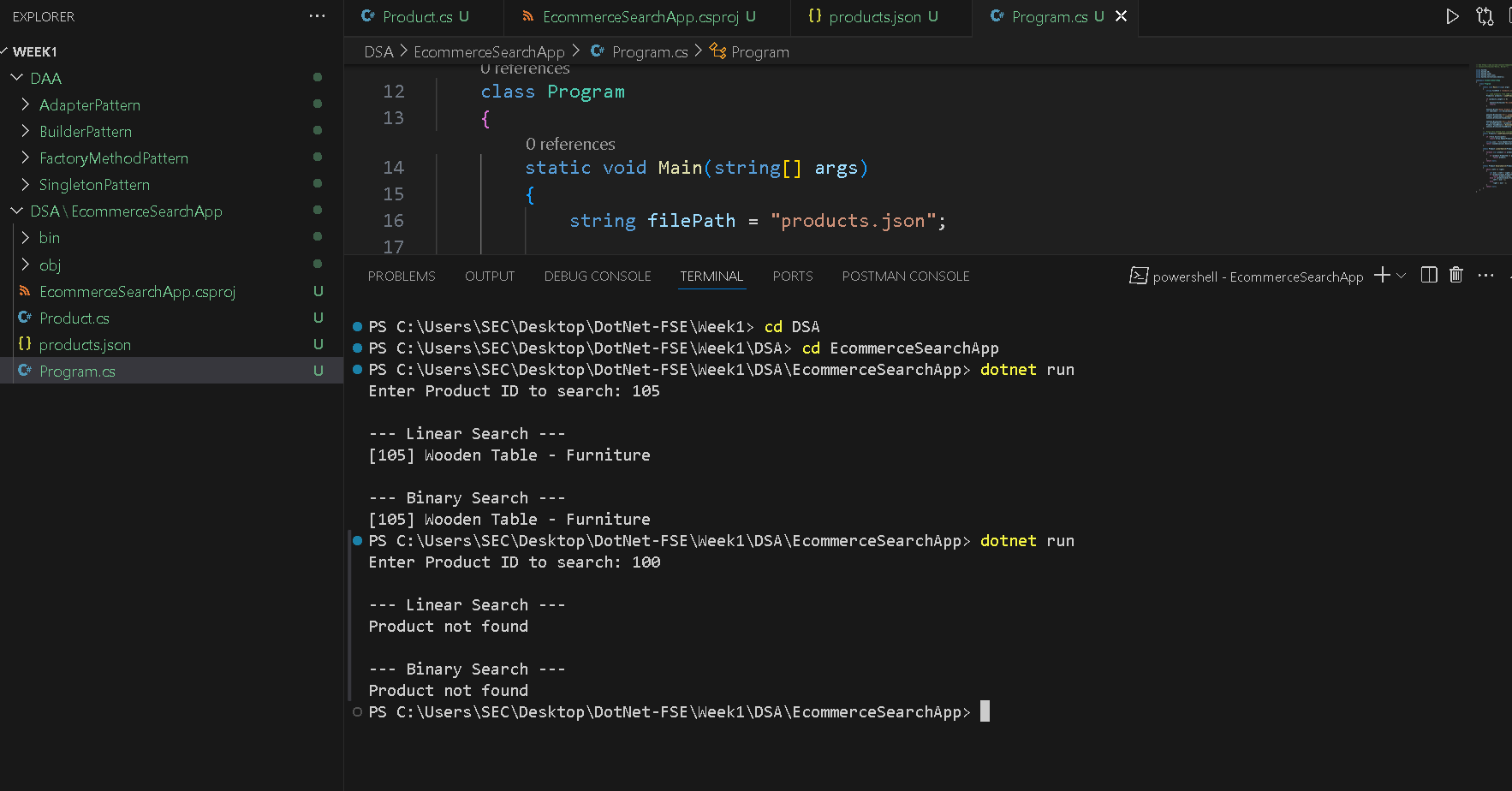
"ProductName": "Wooden Table",

"Category": "Furniture"

}

]

OUTPUT SCREENSHOT:



**Exercise 7: Financial Forecasting**

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

Program.cs

using System;

using System.Collections.Generic;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

double initialValue = 1000; // Starting amount

double annualGrowthRate = 0.10; // 10% per year

int years = 5;

Console.WriteLine("Recursive Financial Forecasting Tool");

Console.WriteLine($"Initial Value: {initialValue}, Growth Rate: {annualGrowthRate \* 100}%");

double forecast = ForecastRecursive(initialValue, annualGrowthRate, years);

Console.WriteLine($"\nForecasted Value after {years} years: ₹{forecast:F2}");

// Optimized version with memoization

var memo = new Dictionary<int, double>();

double optimizedForecast = ForecastMemo(initialValue, annualGrowthRate, years, memo);

Console.WriteLine($"[Optimized] Forecasted Value: ₹{optimizedForecast:F2}");

}

// Recursive method

static double ForecastRecursive(double value, double rate, int years)

{

if (years == 0)

return value;

return ForecastRecursive(value, rate, years - 1) \* (1 + rate);

}

// Optimized recursive method (Memoization)

static double ForecastMemo(double value, double rate, int years, Dictionary<int, double> memo)

{

if (years == 0)

return value;

if (memo.ContainsKey(years))

return memo[years];

double result = ForecastMemo(value, rate, years - 1, memo) \* (1 + rate);

memo[years] = result;

return result;

}

}

}

OUTPUT SCREENSHOT:

