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

*Explanation:*

Big O notation is a mathematical representation used to describe the time and space complexity of an algorithm. It characterizes the upper bound of an algorithm's growth rate as the input size increases — essentially telling us how the algorithm performs at scale.

Time Complexity:

- Best Case: The scenario where the algorithm performs the least work.

- Average Case: The expected performance for a random input.

- Worst Case: The scenario where the algorithm performs the most work.

For example:

- Linear Search: Best: O(1), Average: O(n), Worst: O(n)

- Binary Search: Best: O(1), Average: O(log n), Worst: O(log n)

*Code:*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

public class Product

{

public int ProductId { get; set; }

public string? ProductName { get; set; }

public string? Category { get; set; }

}

public interface ISearchStrategy

{

Product? Search(Product[] products, int productId);

}

public class LinearSearch : ISearchStrategy

{

public Product? Search(Product[] products, int productId)

{

return products.FirstOrDefault(p => p.ProductId == productId);

}

}

public class BinarySearch : ISearchStrategy

{

public Product Search(Product[] products, int productId)

{

int left = 0, right = products.Length - 1;

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;

}

}

public class ProductSearchService

{

private readonly ISearchStrategy \_strategy;

public ProductSearchService(ISearchStrategy strategy)

{

\_strategy = strategy;

}

public Product Search(Product[] products, int productId)

{

return \_strategy.Search(products, productId);

}

}

class Program

{

static void Main()

{

var products = new Product[]

{

new Product { ProductId = 101, ProductName = "Laptop", Category = "Electronics" },

new Product { ProductId = 102, ProductName = "Mouse", Category = "Accessories" },

new Product { ProductId = 103, ProductName = "Keyboard", Category = "Accessories" }

};

Array.Sort(products, (a, b) => a.ProductId.CompareTo(b.ProductId)); // for BinarySearch

ISearchStrategy strategy = new BinarySearch(); // can switch to LinearSearch

var service = new ProductSearchService(strategy);

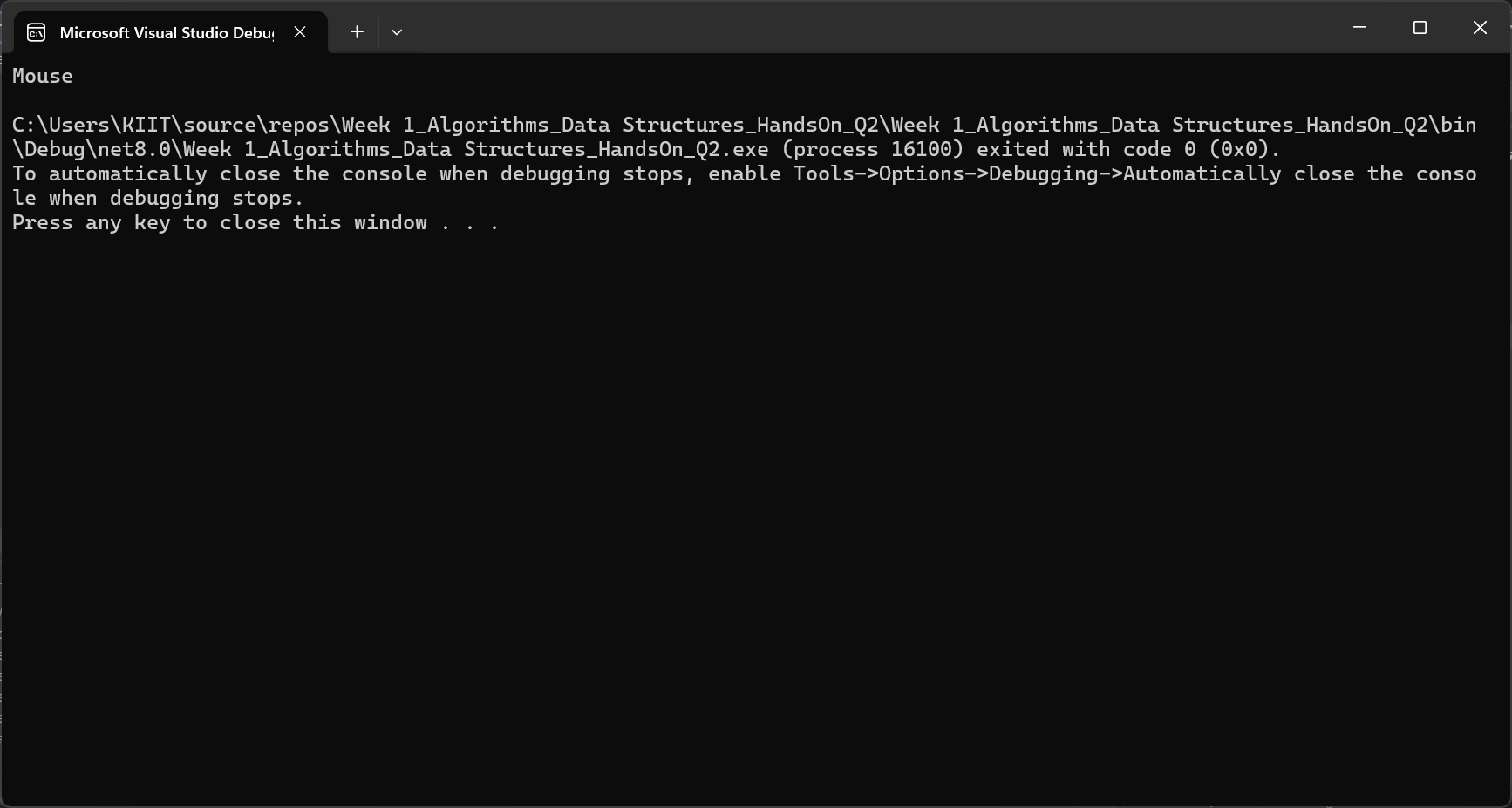
var result = service.Search(products, 102);

Console.WriteLine(result?.ProductName ?? "Product not found");

}

}

*OUTPUT:*

****

*Analysis:*

Linear Search:

- Time Complexity: O(n)

- Use when data is small or unsorted..

Binary Search:

- Time Complexity: O(log n)

- Requires sorted input but much faster for large data sets.

Conclusion:

Binary Search is more efficient for sorted product lists, making it more suitable for real-world e-commerce platforms.

**Exercise 7: Financial Forecasting**

*Explanation:*

Recursion is a technique where a function calls itself to solve smaller subproblems. It simplifies code for repetitive or mathematical problems, especially those based on recurrence patterns like growth or factorial. It reduces manual logic by allowing a function to call itself .

- Base Case: Defines the condition when recursion ends.

- Recursive Case: Breaks the problem down and calls itself.

*Code:*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

public interface IForecastStrategy

{

double Forecast(double initialValue, double growthRate, int years);

}

public class RecursiveForecast : IForecastStrategy

{

public double Forecast(double initialValue, double growthRate, int years)

{

if (years == 0) return initialValue;

return Forecast(initialValue \* (1 + growthRate), growthRate, years - 1);

}

}

public class IterativeForecast : IForecastStrategy

{

public double Forecast(double initialValue, double growthRate, int years)

{

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

initialValue \*= (1 + growthRate);

return initialValue;

}

}

public class ForecastService

{

private readonly IForecastStrategy \_strategy;

public ForecastService(IForecastStrategy strategy)

{

\_strategy = strategy;

}

public double Predict(double initialValue, double growthRate, int years)

{

return \_strategy.Forecast(initialValue, growthRate, years);

}

}

class Program

{

static void Main()

{

IForecastStrategy strategy = new RecursiveForecast(); // or IterativeForecast

var service = new ForecastService(strategy);

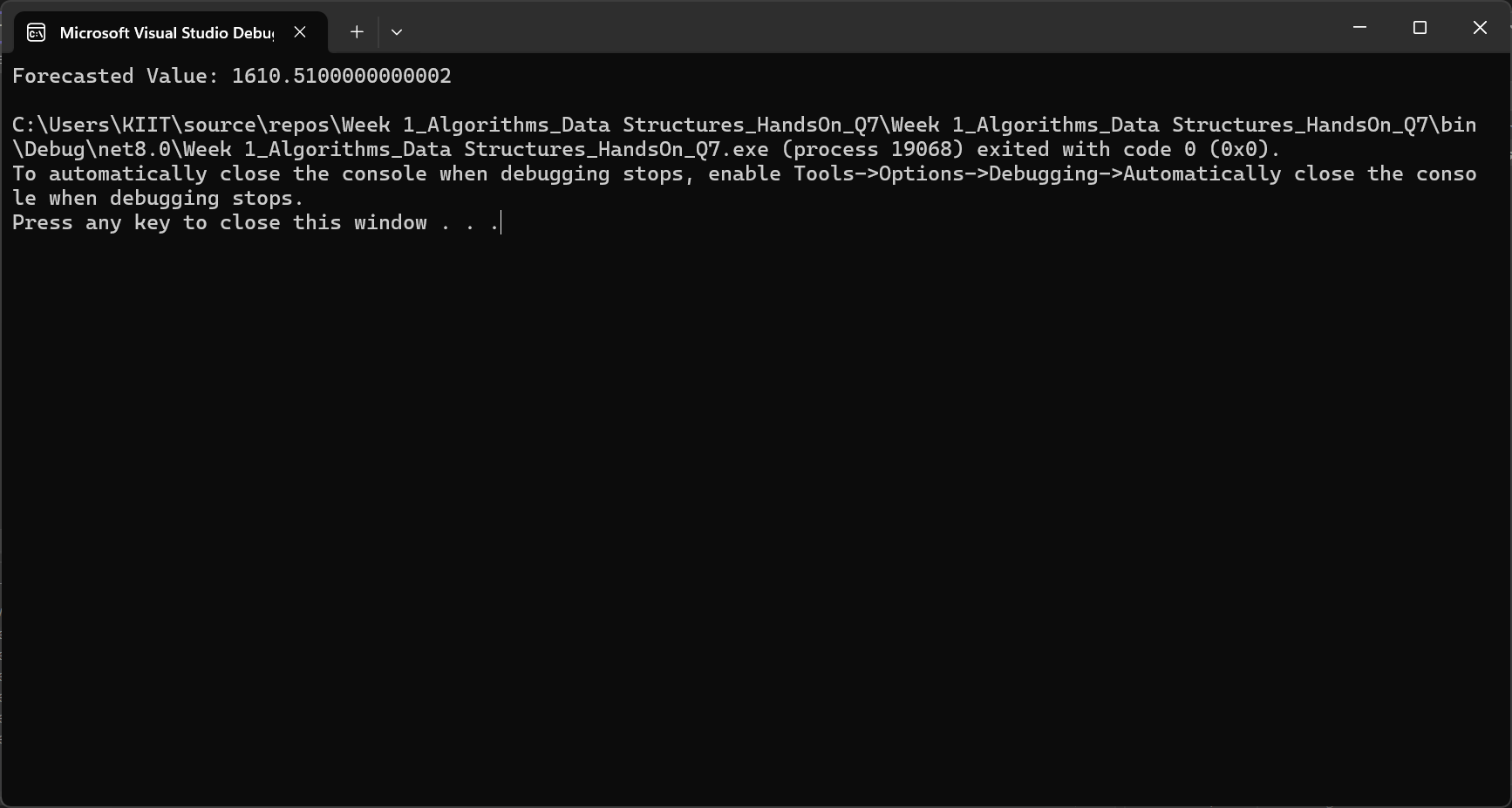
double result = service.Predict(1000, 0.10, 5);

Console.WriteLine($"Forecasted Value: {result}");

}

}

*OUTPUT:*



*Analysis:*

Time Complexity:

* Basic recursion: O(n) time, O(n) space.
* Memoized Recursive: O(n) time, O(n) space (avoids recalculations).

Optimization Strategy:

* Use memoization to store and reuse results of previously calculated years.
* Prevents redundant computation in recursive branches.
* Handles large input sizes more safely and efficiently.

Conclusion:

Recursion is well-suited for financial forecasting based on patterns. Memoization further improves performance for practical use.