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

Big O notation helps to measure time and space complexity of an algorithms as the size of the input increases. It is therefore used to compare the efficiency of different algorithms.

* *Linear Search :*

Best Case :  **O (1)** [The element is found at the first position]

Worst Case : **O (n)** [The element is approximately in the middle]

Average Case : **O (n)** [The element is at the end or not present]

* *Binary Search :*

Best Case : **O (1)** [The element is at the middle position of the first check]

Worst Case : **O (log n)** [The element is not found at any divisions]

Average Case : **O (log n)** [The element is found after few divisions]

**Product Class :**

namespace ECommercePlatformSearchFunction

{

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;

}

}

}

**Search Algorithms Implementation :**

using System;

namespace ECommercePlatformSearchFunction

{

public class SearchAlgorithms

{

public static Product LinearSearch(Product[] products, int targetId)

{

foreach (var product in products)

{

if (product.productId == targetId)

return product;

}

return null;

}

public static Product BinarySearch(Product[] products, int targetId)

{

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

while (left <= right)

{

int mid = (left + right) / 2;

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

return products[mid];

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

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

}

**Element Searching and time complexity measurements :**

using System;

using System.Diagnostics;

using System.Linq;

namespace ECommercePlatformSearchFunction

{

class Program

{

static void Main(string[] args)

{

Product[] products = new Product[]

{

new Product(101, "Pens", "Stationary"),

new Product(102, "Jeans", "Garments"),

new Product(103, "Shoes", "Fashion"),

new Product(104, "Cookies", "Food"),

new Product(105, "Keyboard", "Electronics")

};

Console.WriteLine("Linear Search for ID 104:");

Stopwatch sw1 = Stopwatch.StartNew();

Product result1 = SearchAlgorithms.LinearSearch(products, 104);

sw1.Stop();

Console.WriteLine(result1 != null ? result1.productName : "Product not found");

Console.WriteLine($"Time taken (Linear Search): ({sw1.Elapsed.TotalMilliseconds} ms)\n");

var sorted = products.OrderBy(p => p.productId).ToArray();

Console.WriteLine("Binary Search for ID 104:");

Stopwatch sw2 = Stopwatch.StartNew();

Product result2 = SearchAlgorithms.BinarySearch(sorted, 104);

sw2.Stop();

Console.WriteLine(result2 != null ? result2.productName : "Product not found");

Console.WriteLine($"Time taken (Binary Search): ({sw2.Elapsed.TotalMilliseconds} ms)");

}

}

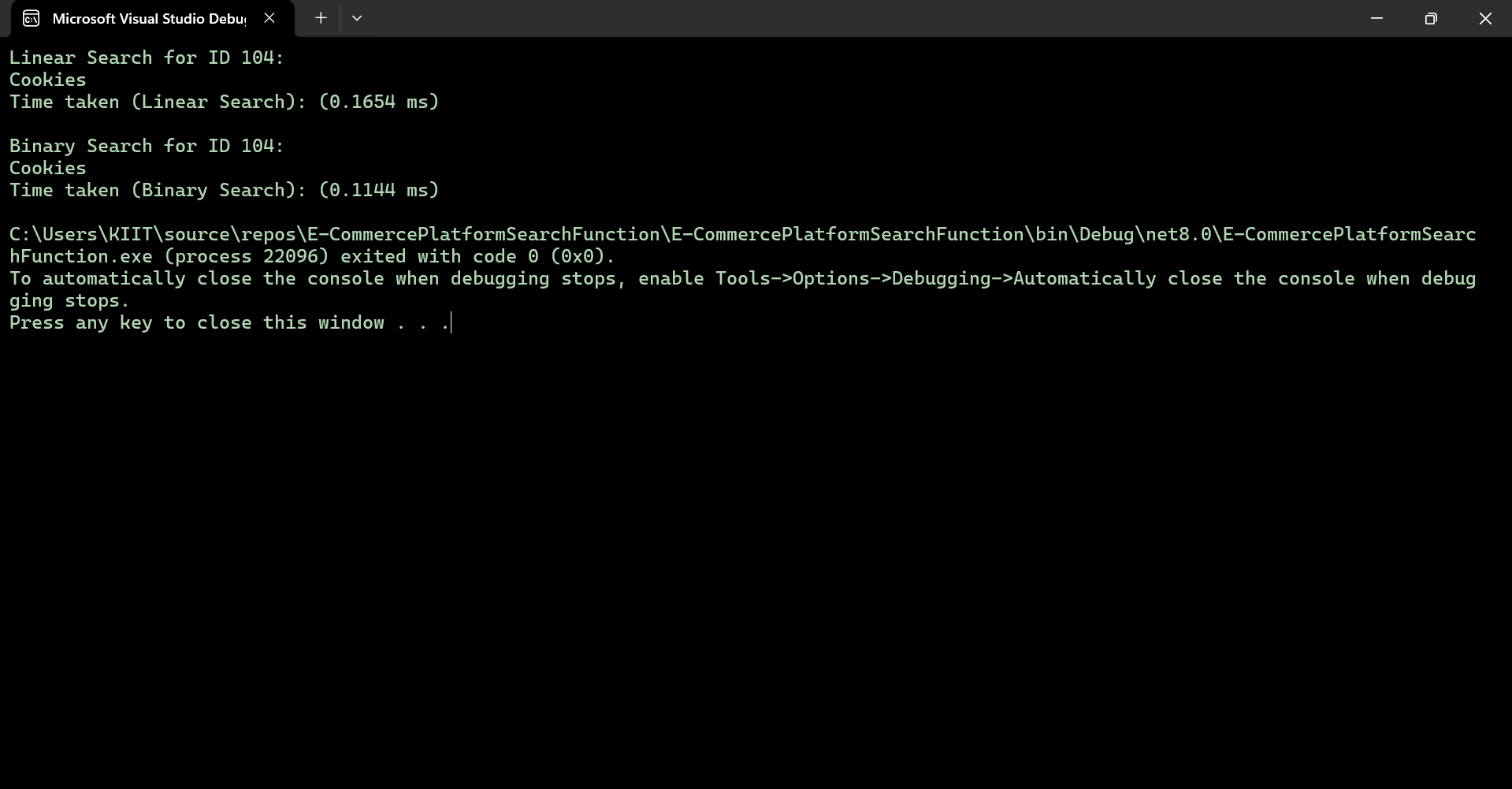
}

* The time complexity of linear search is O(n), as the element is approximately at the middle.
* The time complexity of binary search is O(log n), as the element is approximately at the middle.

For my platform, Binary Search is more suitable as I have searched by IDs and it is significantly faster than linear search.

1. commerece platforms can maintain the sorted order required in binary search.

**OUTPUT :**

****

**Exercise 7: Financial Forecasting**

Recursion is a process where a function calls itself directly or indirectly.

It breaks down a problem into smaller sub-problems and each one of them can be solved easily.

A base case is set and each subproblem must reach that base case to be solved.

**FinancialForecast method (recursive):**

using System;

namespace FinancialForecasting

{

public class FinancialForecast

{

public static double PredictFutureValueRecursive(double currentValue, double growthRate, int years)

{

if (years == 0)

return currentValue;

return PredictFutureValueRecursive(currentValue, growthRate, years - 1) \* (1 + growthRate);

}

}

}

**OptimizedForecast method (iterative):**

using System;

namespace FinancialForecasting

{

public class OptimizedForecast

{

public static double PredictFutureValueIterative(double currentValue, double growthRate, int years)

{

double result = currentValue;

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

{

result \*= (1 + growthRate);

}

return result;

}

}

}

**Implementation of recursive and optimized(iterative) algorithms :**

using System;

using System.Diagnostics;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

double currentValue = 100000;

double growthRate = 0.08;

int years = 5;

Console.WriteLine("Financial Forecasting\n");

//recursive

Stopwatch swRecursive = Stopwatch.StartNew();

double recursiveValue = FinancialForecast.PredictFutureValueRecursive(currentValue, growthRate, years);

swRecursive.Stop();

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

Console.WriteLine($"Time taken (recursive): ({swRecursive.Elapsed.TotalMilliseconds:F6} ms)\n");

//iterative

Stopwatch swIterative = Stopwatch.StartNew();

double iterativeValue = OptimizedForecast.PredictFutureValueIterative(currentValue, growthRate, years);

swIterative.Stop();

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

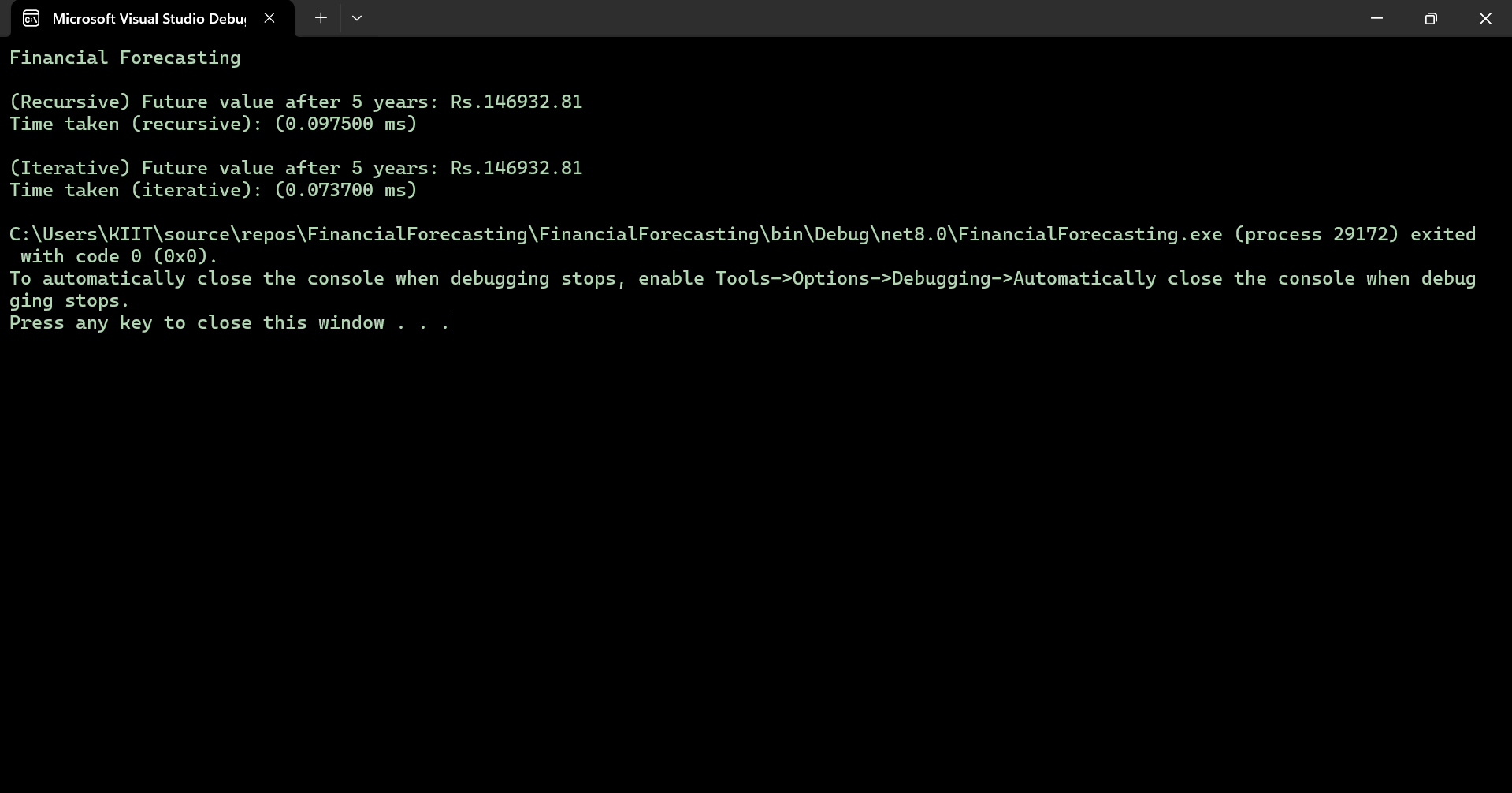
Console.WriteLine($"Time taken (iterative): ({swIterative.Elapsed.TotalMilliseconds:F6} ms)");

}

}

}

**OUTPUT :**

****

* The time complexity of the recursive algorithm is O(n) , as the time will increase, the depth of recursion becomes larger and the time increases and more memory will be consumed.
* The recursive solution can be optimized to avoid excessive computation by applying iterative solution. It does not cann the entire stack, rather it calls each value one by one using a loop.