# Algorithms & Data Structure Exercise Solutions

## Exercise 2: E-commerce Platform Search Function

### Code & Output Screenshots

*Product.cs*

|  |
| --- |
| *using System;*  *using System.Collections.Generic;*  *using System.Linq;*  *using System.Text;*  *using System.Threading.Tasks;*    *namespace ECommercePlatformSearchFunctionality.Entity*  *{*  *public class Product : IComparable<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;*  *}*    *public int CompareTo(Product other)*  *{*  *if(other == null) return 1;*    *return this.productName.CompareTo(other.productName);*  *}*    *public override string ToString()*  *{*  *return $"Id: {productId}, Product Name: {productName}, Category: {category}";*  *}*  *}*  *}* |

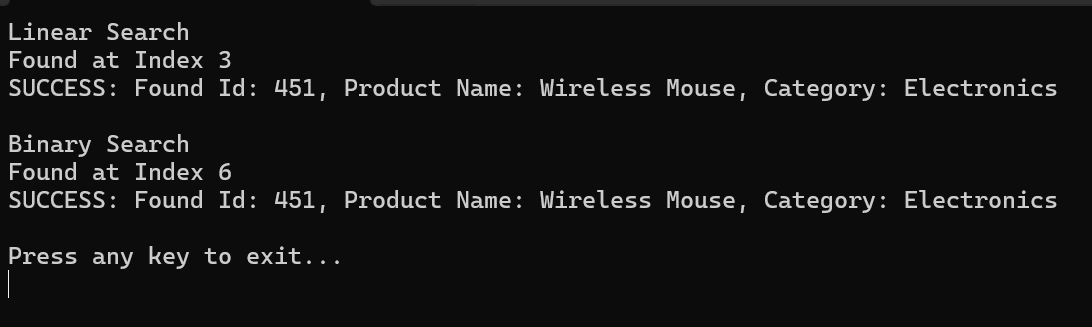
*SearchService.cs*

|  |
| --- |
| *using ECommercePlatformSearchFunctionality.Entity;*  *using System;*  *using System.Collections.Generic;*  *using System.Linq;*  *using System.Text;*  *using System.Threading.Tasks;*    *namespace ECommercePlatformSearchFunctionality.Service*  *{*  *public static class SearchService*  *{*  *public static Product? LinearSearch(List<Product> products, string targetProductName)*  *{*  *for (int i = 0; i < products.Count; i++)*  *{*  *if (products[i].productName.Equals(targetProductName, StringComparison.OrdinalIgnoreCase))*  *{*  *Console.WriteLine($"Found at Index {i}");*  *return products[i];*  *}*  *}*  *return null;*  *}*    *public static Product? BinarySearch(List<Product> products, string targetProductName)*  *{*  *int left = 0;*  *int right = products.Count - 1;*  *int mid = left + (right - left) / 2;*    *while(left <= right)*  *{*  *Product midProduct = products[mid];*  *int comparision = string.Compare(midProduct.productName, targetProductName, StringComparison.OrdinalIgnoreCase);*    *if(comparision == 0)*  *{*  *Console.WriteLine($"Found at Index {mid}");*  *return midProduct;*  *}else if(comparision < 0)*  *{*  *left = mid + 1;*  *}*  *else*  *{*  *right = mid - 1;*  *}*  *mid = left + (right - left) / 2;*  *}*    *return null;*  *}*  *}*  *}* |

*Program.cs*

|  |
| --- |
| *using ECommercePlatformSearchFunctionality.Entity;*  *using ECommercePlatformSearchFunctionality.Service;*  *using System;*    *public class Program*  *{*  *public static void Main(string[] args)*  *{*  *var products = new List<Product>*  *{*  *new Product(101, "Laptop", "Electronics"),*  *new Product(205, "Coffee Maker", "Home Goods"),*  *new Product(312, "T-Shirt", "Apparel"),*  *new Product(451, "Wireless Mouse", "Electronics"),*  *new Product(142, "Bookshelf", "Furniture"),*  *new Product(589, "Running Shoes", "Apparel"),*  *new Product(601, "Desk Lamp", "Home Goods")*  *};*    *string target = "Wireless Mouse";*  *Product? foundProduct;*    *Console.WriteLine("Linear Search");*    *foundProduct = SearchService.LinearSearch(products, target);*    *if (foundProduct != null)*  *{*  *Console.WriteLine($"SUCCESS: Found {foundProduct}");*  *}*  *else*  *{*  *Console.WriteLine($"FAILURE: Product '{target}' not found.");*  *}*    *Console.WriteLine("\nBinary Search");*    *var sortedProducts = new List<Product>*  *{*  *new Product(142, "Bookshelf", "Furniture"),*  *new Product(205, "Coffee Maker", "Home Goods"),*  *new Product(601, "Desk Lamp", "Home Goods"),*  *new Product(101, "Laptop", "Electronics"),*  *new Product(589, "Running Shoes", "Apparel"),*  *new Product(312, "T-Shirt", "Apparel"),*  *new Product(451, "Wireless Mouse", "Electronics")*  *};*    *foundProduct = SearchService.BinarySearch(sortedProducts, target);*    *if (foundProduct != null)*  *{*  *Console.WriteLine($"SUCCESS: Found {foundProduct}");*  *}*  *else*  *{*  *Console.WriteLine($"FAILURE: Product '{target}' not found.");*  *}*  *Console.WriteLine("\nPress any key to exit...");*  *Console.ReadKey();*  *}*  *}* |

*Output*



### *Theoretical answers for the questions for Exercise 2*

## **Understand Asymptotic Notation:**

Q1. **Explain Big O notation and how it helps in analyzing algorithms.**  
*ans:* Big O notation describes the upper bound of an algorithm’s runtime or space usage in terms of input size n. It tells how well the algorithm scales as the size of the data increases.  
It doesn’t measure exact time, but rather the rate of growth — which is essential for comparing algorithms, especially on large inputs.

Q2. **Describe the best, average, and worst-case scenarios for search operations.**  
*ans:* The best, average and worst-case scenarios for Linear and Binary Searches are:  
*Linear Search (Unsorted Data)*  
Best Case: O(1) → Element found at the beginning.  
Average Case: O(n/2) ≈ O(n) → Element in the middle or randomly located.  
Worst Case: O(n) → Element not present or at the end.

*Binary Search (Sorted Data)*  
Best Case: O(1) → Element found in the first middle comparison.  
Average Case: O(log n) → Element found in one of the mid divisions.  
Worst Case: O(log n) → Element not present, requires full depth traversal.

## **Analysis:**

Q1.**Compare the time complexity of linear and binary search algorithms.**  
*ans:* The direct comparision of the searche algorithms:  
***Linear Search***  
*Time Complexity (Worst-Case)*: O(N) Linear  
*Working*: Checks each product one by one, from the beginning to the end.

***Binary Search***  
*Time Complexity (Worst-Case)*: O(log N) Logarithmic  
*Working*: Jumps to the middle, decides if the target is in the left or right half, and repeats.  
*Requirement*: The data must be sorted first (e.g., by productName or productId).

Q2. **Discuss which algorithm is more suitable for your platform and why.**  
*ans*: *Binary Search* is moe suitable algorithm for this platform, due to its optimal Time Complexity, i.e. O(log N) making it better than linear search that takes, O(N).  
For an e-commerce platform fast search is crucial for user experience.

## Exercise 7: Financial Forecasting

### Code & Output Screenshots

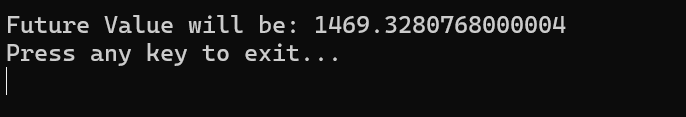
*FinancialForecasting.cs*

|  |
| --- |
| *using System;*  *using System.Collections.Generic;*  *using System.Linq;*  *using System.Text;*  *using System.Threading.Tasks;*    *namespace FinancialForecasting*  *{*  *public class FinancialForecaster*  *{*  *public static double Calculate(double p, double r, int t)*  *{*  *if (t == 0) return p;*    *double prevYearVal = Calculate(p, r, t - 1);*  *return prevYearVal \* (1 + r);*  *}*  *}*  *}* |

*Program.cs*

|  |
| --- |
| *using System;*    *namespace FinancialForecasting*  *{*  *class Program*  *{*  *public static void Main(string[] args)*  *{*  *double initialInvestment = 1000;*  *double annualRate = 0.08;*  *int investmentYears = 5;*    *double futureVal = FinancialForecaster.Calculate(initialInvestment, annualRate, investmentYears);*    *Console.WriteLine($"Future Value will be: {futureVal}");*      *Console.WriteLine("Press any key to exit...");*  *Console.ReadKey();*  *}*  *}*  *}* |

*Output*



### *Theoretical answers for the questions for Exercise 7*

*Understand Recursive Algorithms:*

*Q1) Explain the concept of recursion and how it can simplify certain problems.*

*Ans:* Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It simplifies problems by breaking them down into base cases (simple, directly solvable cases) and recursive cases (where the function calls itself with a smaller input). This approach is especially useful for problems that have a natural hierarchical or repetitive structure.

For example, calculating factorial *n!n*! is naturally recursive:

* Base case: *0!=1*0!=1
* Recursive case: *n!=n×(n−1)!n*!=*n*×(*n*−1)!

In financial forecasting, recursion can simplify calculations like predicting future values based on past values and growth rates.

*Analysis:*

*Q1) Discuss the time complexity of your recursive algorithm.*

*Ans:* The time complexity of this recursive algorithm is O(n) because it makes one recursive call per period until it reaches the base case.

*Q2) Explain how to optimize the recursive solution to avoid excessive computation*

*Ans:* Recursive algorithms can be inefficient if they recompute the same values multiple times. Although this example does not have overlapping subproblems, in more complex recursive forecasting models, repeated calculations can occur.

To optimize:

* Use Memoization: Store results of subproblems in a cache (e.g., a dictionary) so that repeated calls with the same parameters return the cached result instead of recomputing.
* Alternatively, use Dynamic Programming (DP): Convert the recursive approach into an iterative one that builds up the solution from the base case, avoiding recursion overhead.