**Name: Deepanshu Rathore**

**Superset ID: 6358199**  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
Week 1  
Exercise 2: E-commerce Platform Search Function  
  
*Big O notation can be thought of as a way to describe how the time or work required by an algorithm grows when the size of the input grows. In other words, it gives a shorthand for the worst‐case behavior of an algorithm as the number of items increases. When we say that an algorithm is O(n), we mean that doubling the number of items will about double the time it takes. If something is O(log n), then doubling the items only adds a small constant amount of extra work. This helps us compare different approaches without getting lost in machine‐specific details.*

*When talking about search operations, we often consider three scenarios. In the best case, the item we look for happens to be right at the start of the list, so we find it immediately. The average case imagines that the item is somewhere in the middle on average, so we check roughly half of the items before finding it. The worst case is when the item is at the very end or not in the list at all, which means we check every single entry before concluding. These cases give us a realistic sense of how an algorithm might perform in everyday use versus in a perfectly lucky or unlucky situation.*

*To set up the data structure for searching, I would create a class named Product. Inside this class there would be an integer field called productId to hold a unique number for each product. Next, I would include a string field named productName so that the search can match by name or keywords. Finally, there would be another string field called category to allow searches filtered by type, such as “electronics” or “clothing.” With these three attributes, the search feature can look for products by their ID, name, or category.*

*Comparing linear search and binary search by their time complexity shows a clear difference. Linear search looks at items one by one from start to finish, so in the worst case it takes time proportional to n, written O(n). Binary search, on the other hand, requires the list to be sorted first. It repeatedly cuts the search interval in half, so its worst‐case time is proportional to log n, written O(log n). This means binary search becomes much faster than linear search once you have a large collection of products.*

*For an e-commerce platform, binary search tends to be more suitable if the product list is kept in sorted order, for example by productId or by name. Since the catalog can be very large, the lower time complexity of O(log n) makes searches respond quickly even with thousands or millions of items. Linear search might still be used for very small lists or when the data cannot be sorted easily, but in most cases the performance benefit of binary search makes it the better choice.*

Code:  
using System;

using System.Diagnostics;

public class Product

{

    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public string Category { get; set; }

    public Product(int productId, string productName, string category)

    {

        ProductId = productId;

        ProductName = productName;

        Category = category;

    }

    public override string ToString()

    {

        return $"Product ID: {ProductId}, Name: {ProductName}, Category: {Category}";

    }

}

public class SearchAlgorithms

{

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

    {

        for (int i = 0; i < products.Length; i++)

        {

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

            {

                return products[i];

            }

        }

        return null;

    }

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

    {

        int left = 0;

        int right = products.Length - 1;

        while (left <= right)

        {

            int mid = left + (right - left) / 2;

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

            {

                return products[mid];

            }

            if (products[mid].ProductId < productId)

            {

                left = mid + 1;

            }

            else

            {

                right = mid - 1;

            }

        }

        return null;

    }

}

public class Program

{

    static void Main(string[] args)

    {

        Console.WriteLine("E-commerce Platform Search Function");

        // create products array

        Product[] products = new Product[10000];

        for (int i = 0; i < products.Length; i++)

        {

            string category = i % 5 == 0 ? "Electronics" : i % 4 == 0 ? "Clothing" : i % 3 == 0 ? "Books" : i % 2 == 0 ? "Home" : "Sports";

            products[i] = new Product(i + 1, $"Product {i + 1}", category);

        }

        // create a sorted copy for binary search

        Product[] sortedProducts = new Product[products.Length];

        Array.Copy(products, sortedProducts, products.Length);

        // Note: Products are already sorted by ProductId

        // search for products

        int searchId = 8765; // Product to search for

        Console.WriteLine("\nSearching for Product ID: " + searchId);

        // measure Linear Search time

        Stopwatch linearStopwatch = new Stopwatch();

        linearStopwatch.Start();

        Product linearResult = SearchAlgorithms.LinearSearch(products, searchId);

        linearStopwatch.Stop();

        // measure Binary Search time

        Stopwatch binaryStopwatch = new Stopwatch();

        binaryStopwatch.Start();

        Product binaryResult = SearchAlgorithms.BinarySearch(sortedProducts, searchId);

        binaryStopwatch.Stop();

        // display results

        Console.WriteLine("\nLinear Search Result:");

        Console.WriteLine($"Found: {linearResult}");

        Console.WriteLine($"Time taken: {linearStopwatch.ElapsedTicks} ticks");

        Console.WriteLine("\nBinary Search Result:");

        Console.WriteLine($"Found: {binaryResult}");

        Console.WriteLine($"Time taken: {binaryStopwatch.ElapsedTicks} ticks");

        Console.WriteLine("\nTest completed. Press any key to exit.");

        Console.ReadKey();

    }

}

Output:  
A screenshot of a computer

AI-generated content may be incorrect.

Exercise 7: Financial Forecasting  
  
*Recursion is a way of defining a problem so that its solution depends on solutions to smaller instances of the same problem. In simple terms, a function calls itself with a reduced version of the input until it reaches a condition where it can return an answer directly. This “base case” stops the calls from going on forever. By breaking a complex task into identical but smaller tasks, recursion can make certain solutions clearer and more elegant than trying to track every step in a single, flat sequence of instructions.*

*For setting up a recursive method for future-value calculation, one imagines that the value at time n is the value at time n − 1 multiplied by a growth factor that depends on the rate for period n. In this view the base case is the known value at period 0, and each step extends the known history by applying one more growth rate. This way, if you know today’s value, you can express tomorrow’s value in terms of today’s, and the same pattern applies until you reach the target period.*

*In the actual recursive implementation one would write a routine that takes two inputs: the initial amount and the list of growth rates up to the desired period. The routine first checks if it has reached period 0. If so, it returns the initial amount. Otherwise, it removes the last rate from the list, calls itself on the shorter list to get the value up to the previous period, and then multiplies that result by one plus the removed rate. This structure mirrors the mathematical definition:*

FutureValue(n) = FutureValue(n − 1) x (1 + rate\_n)

with FutureValue(0) = initialAmount.

*Finally, when we consider time complexity, each call to the recursive function makes exactly one further recursive call until it hits the base case. That means for n periods the function will call itself n + 1 times, giving a time complexity on the order of n. However, since each call waits for its recursive child to complete before doing its own multiplication, the overall work is linear but the call stack grows to depth n. To avoid repeated work if, for example, one wanted values at many periods without restarting from zero each time, one could use memoization to store previously computed results. Alternatively, converting the recursion into an iterative loop eliminates the overhead of many function calls and a deep call stack, achieving the same linear-time behavior with constant additional space.*  
  
Code:  
using System;

public class FinancialForecasting

{

public static double RecursiveGrowthForecast(double initialValue, double growthRate, int periods)

{

if (periods == 0)

{

return initialValue;

}

return RecursiveGrowthForecast(initialValue, growthRate, periods - 1) \* (1 + growthRate);

}

public static double MemoizedGrowthForecast(double initialValue, double growthRate, int periods, Dictionary<int, double>? memo = null)

{

if (memo == null)

{

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

}

if (periods == 0)

{

return initialValue;

}

if (memo.ContainsKey(periods))

{

return memo[periods];

}

memo[periods] = MemoizedGrowthForecast(initialValue, growthRate, periods - 1, memo) \* (1 + growthRate);

return memo[periods];

}

public static double TailRecursiveGrowthForecast(double initialValue, double growthRate, int periods)

{

return TailRecursiveHelper(initialValue, growthRate, periods, initialValue);

}

private static double TailRecursiveHelper(double initialValue, double growthRate, int periodsRemaining, double currentValue)

{

if (periodsRemaining == 0)

{

return currentValue;

}

return TailRecursiveHelper(initialValue, growthRate, periodsRemaining - 1, currentValue \* (1 + growthRate));

}

}

public class Program

{

static void Main(string[] args)

{

Console.WriteLine("Financial Forecasting Tool");

double initialInvestment = 10000;

double annualGrowthRate = 0.07; // 7% annual growth

Console.WriteLine($"Initial Investment: ${initialInvestment}");

Console.WriteLine($"Annual Growth Rate: {annualGrowthRate \* 100}%");

Console.WriteLine();

Console.WriteLine("Future Value Forecasts:");

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

for (int year = 1; year <= 30; year += 5)

{

Console.WriteLine($"Year {year}:");

var stopwatch = new System.Diagnostics.Stopwatch();

stopwatch.Start();

double recursiveValue = FinancialForecasting.RecursiveGrowthForecast(initialInvestment, annualGrowthRate, year);

stopwatch.Stop();

long recursiveTime = stopwatch.ElapsedTicks;

stopwatch.Restart();

double memoizedValue = FinancialForecasting.MemoizedGrowthForecast(initialInvestment, annualGrowthRate, year);

stopwatch.Stop();

long memoizedTime = stopwatch.ElapsedTicks;

stopwatch.Restart();

double tailRecursiveValue = FinancialForecasting.TailRecursiveGrowthForecast(initialInvestment, annualGrowthRate, year);

stopwatch.Stop();

long tailRecursiveTime = stopwatch.ElapsedTicks;

Console.WriteLine($" Simple Recursive: ${recursiveValue:F2} (Time: {recursiveTime} ticks)");

Console.WriteLine($" Memoized: ${memoizedValue:F2} (Time: {memoizedTime} ticks)");

Console.WriteLine($" Tail Recursive: ${tailRecursiveValue:F2} (Time: {tailRecursiveTime} ticks)");

Console.WriteLine();

}

// compare algorithm performance for a larger value

int largeYear = 45;

Console.WriteLine($"Performance comparison for {largeYear} years:");

var performanceStopwatch = new System.Diagnostics.Stopwatch();

performanceStopwatch.Start();

double memoizedLargeValue = FinancialForecasting.MemoizedGrowthForecast(initialInvestment, annualGrowthRate, largeYear);

performanceStopwatch.Stop();

long memoizedLargeTime = performanceStopwatch.ElapsedTicks;

performanceStopwatch.Restart();

double tailRecursiveLargeValue = FinancialForecasting.TailRecursiveGrowthForecast(initialInvestment, annualGrowthRate, largeYear);

performanceStopwatch.Stop();

long tailRecursiveLargeTime = performanceStopwatch.ElapsedTicks;

Console.WriteLine($" Memoized: ${memoizedLargeValue:F2} (Time: {memoizedLargeTime} ticks)");

Console.WriteLine($" Tail Recursive: ${tailRecursiveLargeValue:F2} (Time: {tailRecursiveLargeTime} ticks)");

// simple recursive would be too slow for this value

Console.WriteLine(" Simple Recursive: Too slow to calculate");

Console.WriteLine("\nPress any key to exit.");

Console.ReadKey();

}  
}  
Output:  
A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.