WEEK 1 DATA STRUCTURE AND ALGORITHMS

**Question:**

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

**Scenario:**

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

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

Answers:

**Big o notation:**

**Big O** as a tool that tells us how slow or fast a program can get as it handles more and more data

**O(1)** means: No matter how big the list is, the time stays the same. Super fast

**O(n)** means: The time grows linearly as the number of items grows. Double the items = double the time.

**O(log n)** means: Even as the list grows, time increases slowly. That’s what **binary search** gives us, and it’s why it’s good.

**Best Case**: Minimum time required.

**Average Case**: Expected time over all inputs.

**Worst Case**: Maximum time required.

**Analysis:**

1. Linear Search:

Time Complexity: O(n)

Best Case: O(1) (if the item is found at the beginning)

Worst Case: O(n) (if the item is last or not present)

Use Case: When the product list is small or unsorted.

2. Binary Search:

Time Complexity: O(log n)

Best Case: O(1) (if the item is exactly in the middle)

Worst Case: O(log n)

Requirement: The product list must be sorted.

**Code:**

Search.java

import java.util.Arrays;

import java.util.Comparator;

public class Search {

    public static int linearSearch(Product[] products, String productName) {

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

            if (products[i].getProductName().equals(productName)) {

                return i;

            }

        }

        return -1;

    }

    public static int binarySearch(Product[] products, String productName) {

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

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

            int comparison = products[mid].getProductName().compareTo(productName);

            if (comparison == 0) {

                return mid;

            }

            if (comparison < 0) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return -1;

    }

    public static void main(String[] args) {

        Product[] products = {

            new Product("P4", "Product D", "Category Y"),

            new Product("P2", "Product B", "Category X"),

            new Product("P5", "Product E", "Category Y"),

            new Product("P1", "Product A", "Category X"),

            new Product("P3", "Product C", "Category X")

        };

        String searchTarget = "Product C";

        System.out.println("Searching for: '" + searchTarget + "'\n");

        System.out.println("Initial unsorted array: " + Arrays.toString(products));

        int indexBeforeSorting = linearSearch(products, searchTarget);

        if (indexBeforeSorting != -1) {

            System.out.println("Before sorting, found at index: " + indexBeforeSorting);

        } else {

            System.out.println("Product not found before sorting.");

        }

        Arrays.sort(products, Comparator.comparing(Product::getProductName));

        System.out.println("Array after sorting:  " + Arrays.toString(products));

        int indexAfterSorting = binarySearch(products, searchTarget);

        if (indexAfterSorting != -1) {

            System.out.println("After sorting, found at index: " + indexAfterSorting);

        } else {

            System.out.println("Product not found after sorting.");

        }

    }

}

Product.java

public class Product {

    private String productId;

    private String productName;

    private String category;

    public Product(String productId, String productName, String category) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    public String getProductId() {

        return productId;

    }

    public String getProductName() {

        return productName;

    }

    public String getCategory() {

        return category;

    }

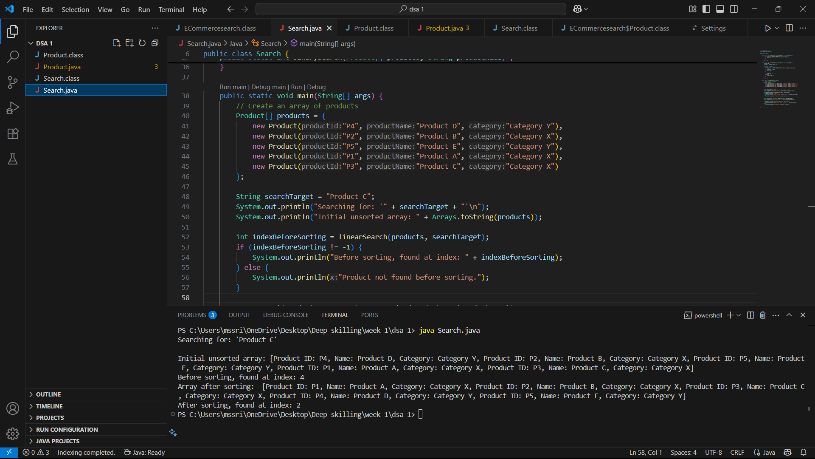
    public String toString() {

        return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

    }

}

Output:



**Question:**

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

Answers:

Recursion is a powerful programming technique where a method calls itself to solve a problem. It works by breaking a complex problem down into smaller, more manageable sub-problems that are identical in nature to the original problem.

Code:

FinancialForecasting.java

public class FinancialForecasting {

    public static double calculateFutureValue(double presentValue, double growthRate, int years) {

        double futureValue = presentValue;

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

            futureValue = futureValue \* (1 + growthRate);

        }

        return futureValue;

    }

    public static void main(String[] args) {

        double initialInvestment = 25000.0;

        double annualGrowthRate = 0.06;

        int forecastYears = 10;

        System.out.println("Initial Investment: $" + initialInvestment);

        System.out.println("Annual Growth Rate: " + (annualGrowthRate \* 100) + "%");

        System.out.println("Forecasting for: " + forecastYears + " years\n");

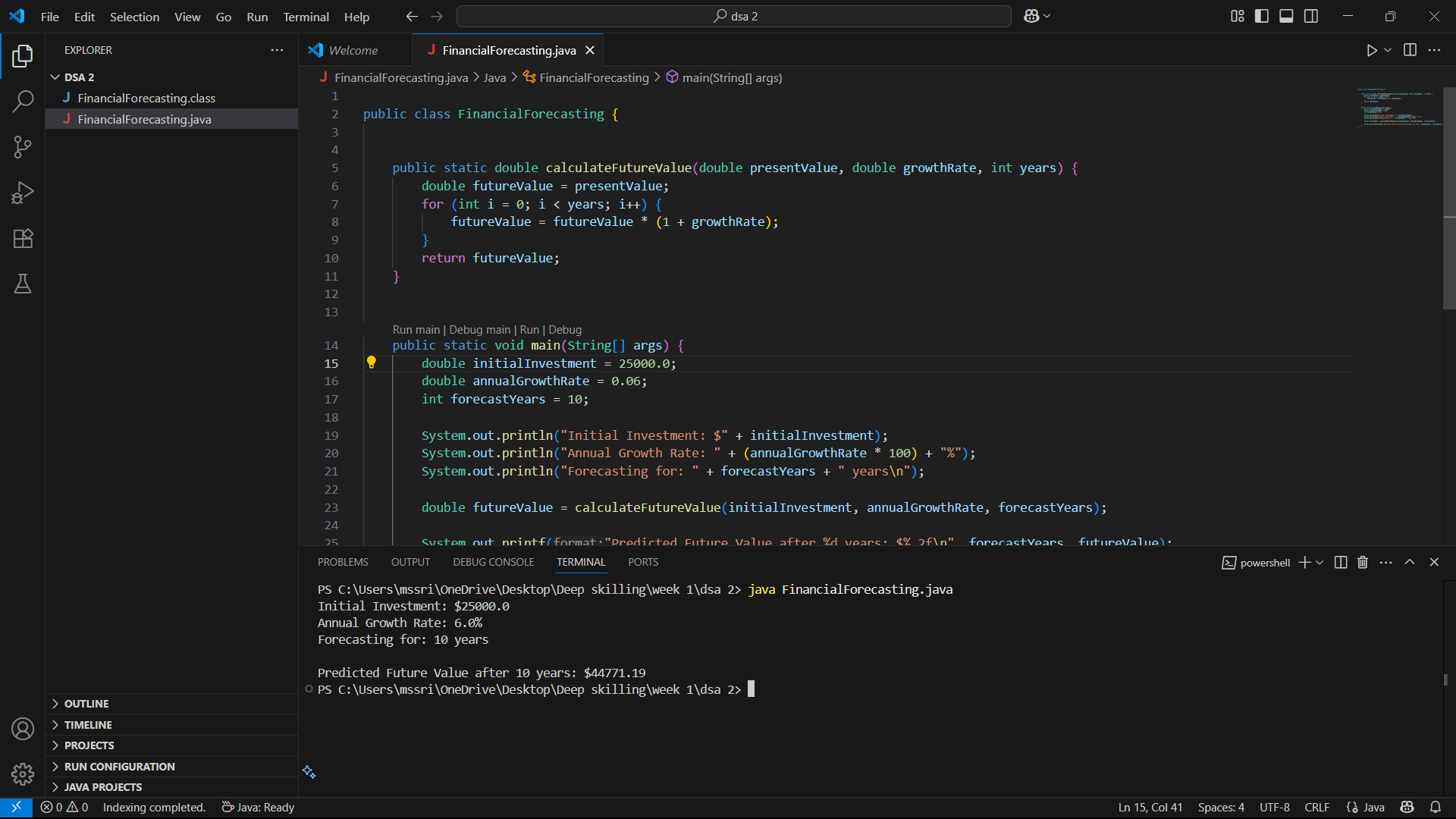
        double futureValue = calculateFutureValue(initialInvestment, annualGrowthRate, forecastYears);

        System.out.printf("Predicted Future Value after %d years: $%.2f\n", forecastYears, futureValue);

    }

}

Output:



Analysis:

**Time Complexity:**

The time complexity of this iterative algorithm is **O(n)**, where 'n' is the number of years.  The core of the calculation is a for loop that runs exactly 'n' times. Inside the loop, a single multiplication is performed, which is a constant time operation, O(1). Therefore, the total time is directly proportional to the number of years, resulting in a linear time complexity of O(n).