**WEEK-1**

**Data Structures and Algorithms HandsOn**

**Exercise 1: Inventory Management System:**

Warehouses deals with thousands of products in that kind of situation efficient search, update and deletion plays a crucial role. Poor data structures leads to slow performance.

**Suitable Data Structures:**

**ArrayList :** Good for sequential storage but search and update is slow it takes O(n) Time Complexity

**HashMap :** It is used for fast access it takes O(n) Time Complexity

**TreeMap :** Maintain sorted order by key. Useful where sorted order is required. It takes O(log n) Time Complexity.

**Code:**

**package InventoryManagementSystem;**

import java.util.HashMap;

import java.util.Map;

class Product {

    int productId;

    String productName;

    int quantity;

    double price;

    public Product(int productId, String productName, int quantity, double price) {

        this.productId = productId;

        this.productName = productName;

        this.quantity = quantity;

        this.price = price;

    }

    public void displayProduct() {

        System.out.println("ID: " + productId);

        System.out.println("Name: " + productName);

        System.out.println("Quantity: " + quantity);

        System.out.println("Price: " + price);

        System.out.println();

    }

}

class Inventory {

    private Map<Integer, Product> inventory = new HashMap<>();

    public void addProduct(Product product) {

        inventory.put(product.productId, product);

        System.out.println("Product added");

    }

    public void updateProduct(int productId, String name, int quantity, double price) {

        if (inventory.containsKey(productId)) {

            Product p = inventory.get(productId);

            p.productName = name;

            p.quantity = quantity;

            p.price = price;

            System.out.println("Product updated");

        } else {

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

        }

    }

    public void deleteProduct(int productId) {

        if (inventory.remove(productId) != null) {

            System.out.println("Product deleted.");

        } else {

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

        }

    }

    public void displayAllProducts() {

        for (Product p : inventory.values()) {

            p.displayProduct();

        }

    }

}

public class InventoryManagementSystem {

    public static void main(String[] args) {

        Inventory inventory = new Inventory();

        inventory.addProduct(new Product(1, "Laptop", 10, 60000));

        inventory.addProduct(new Product(2, "Mouse", 50, 500));

        inventory.displayAllProducts();

        inventory.updateProduct(1, "Gaming Laptop", 8, 85000);

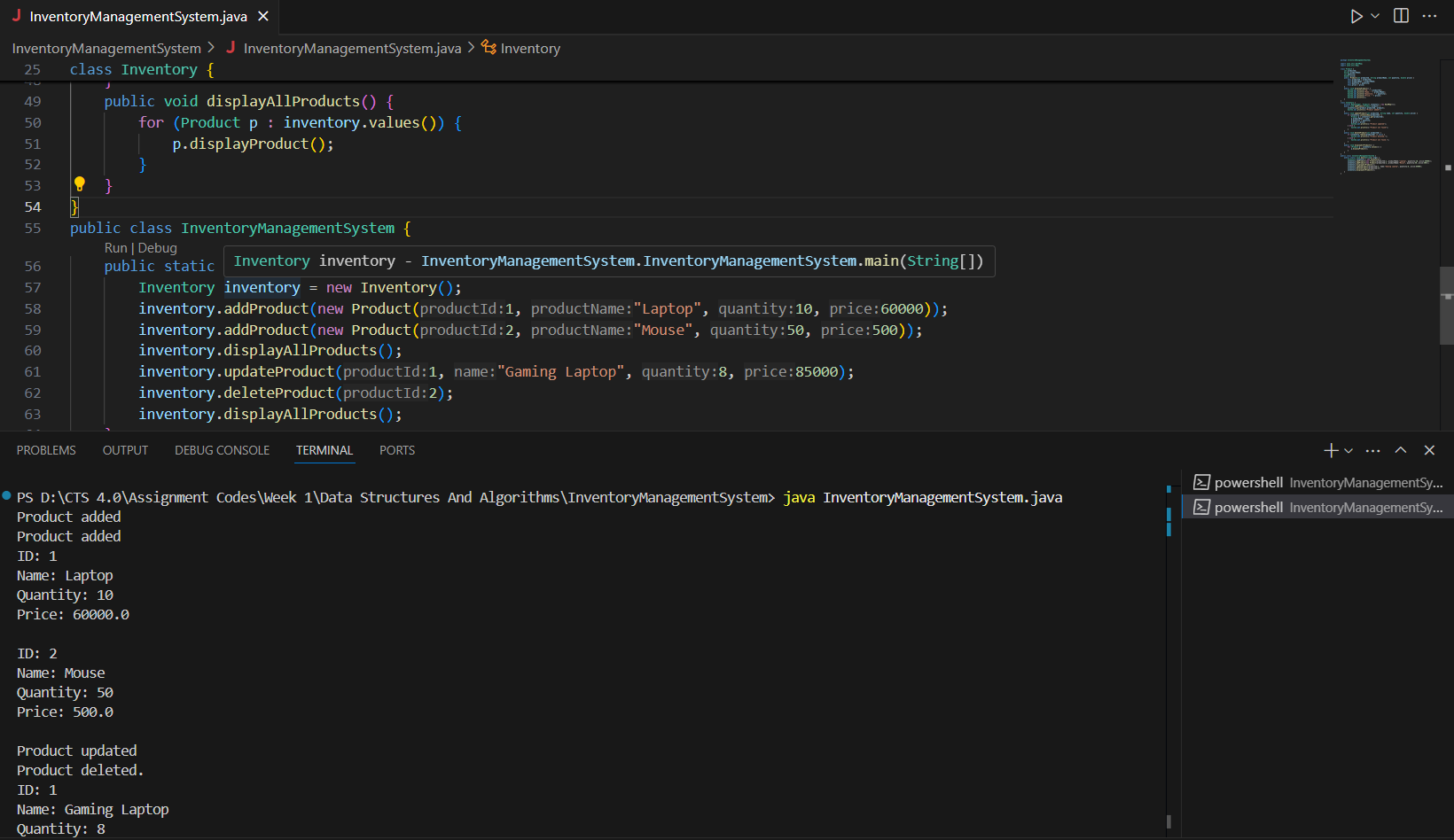
        inventory.deleteProduct(2);

        inventory.displayAllProducts();

    }

}

**Output:**



**Time Complexity Analysis:**

Add Product – O(1) Constant time for insertion by key.

Update Product – O(1) Direct update of details by key.

Delete Product – O(1) Constant time for deleting a product by accessing through key.

Display All – O(n) Must iterate through all products.

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

**Big O Notation:**

Mathematical notation used to describe the performance or complexity of an algorithm. Specifically how its runtime or memory usage increases as the input size increases.

**Search operations:**

There are two types of searching methods  
 i) Linear Search

ii) Binary Search

**i)Linear Search:**

Best Case : O(1)

Average Case : O(n)

Worst Case : O(n)

**ii)Binary Search:**

Best Case : O(1)

Average Case : O(log n)

Worst Case : O(log n)

**Code:**

package EcommerceSearch;

import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearch {

    static class Product{

        int productId;

        String productName;

        String category;

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

            this.productId = productId;

            this.productName = productName;

            this.category = category;

        }

        public String display(){

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

        }

    }

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

        for(Product p:products){

            if(p.productName.equalsIgnoreCase(target)){

                return p;

            }

        }

        return null;

    }

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

        int low = 0;

        int high = products.length-1;

        while(low<=high){

            int mid = (low+high)/2;

            int value = products[mid].productName.compareToIgnoreCase(target);

            if(value == 0){

                return products[mid];

            }

            else if(value<0){

                low = mid+1;

            }

            else{

                high = mid-1;

            }

        }

        return null;

    }

    public static void sort(Product[] products){

        Arrays.sort(products,Comparator.comparing(p->p.productName.toLowerCase()));

    }

    public static void main(String[] args) {

        Product[] products = {new Product(1, "Mobile", "Electronics"),

        new Product(2, "Laptop", "Electronics"),

        new Product(3, "Watch", "Accessories"),

        new Product(4, "Dress", "Fashion")};

        String target = "Dress";

        Product search1 = linearSearch(products, target);

        System.out.println("Linear Search : "+(search1!=null?search1.display():"Product Not found"));

        sort(products);

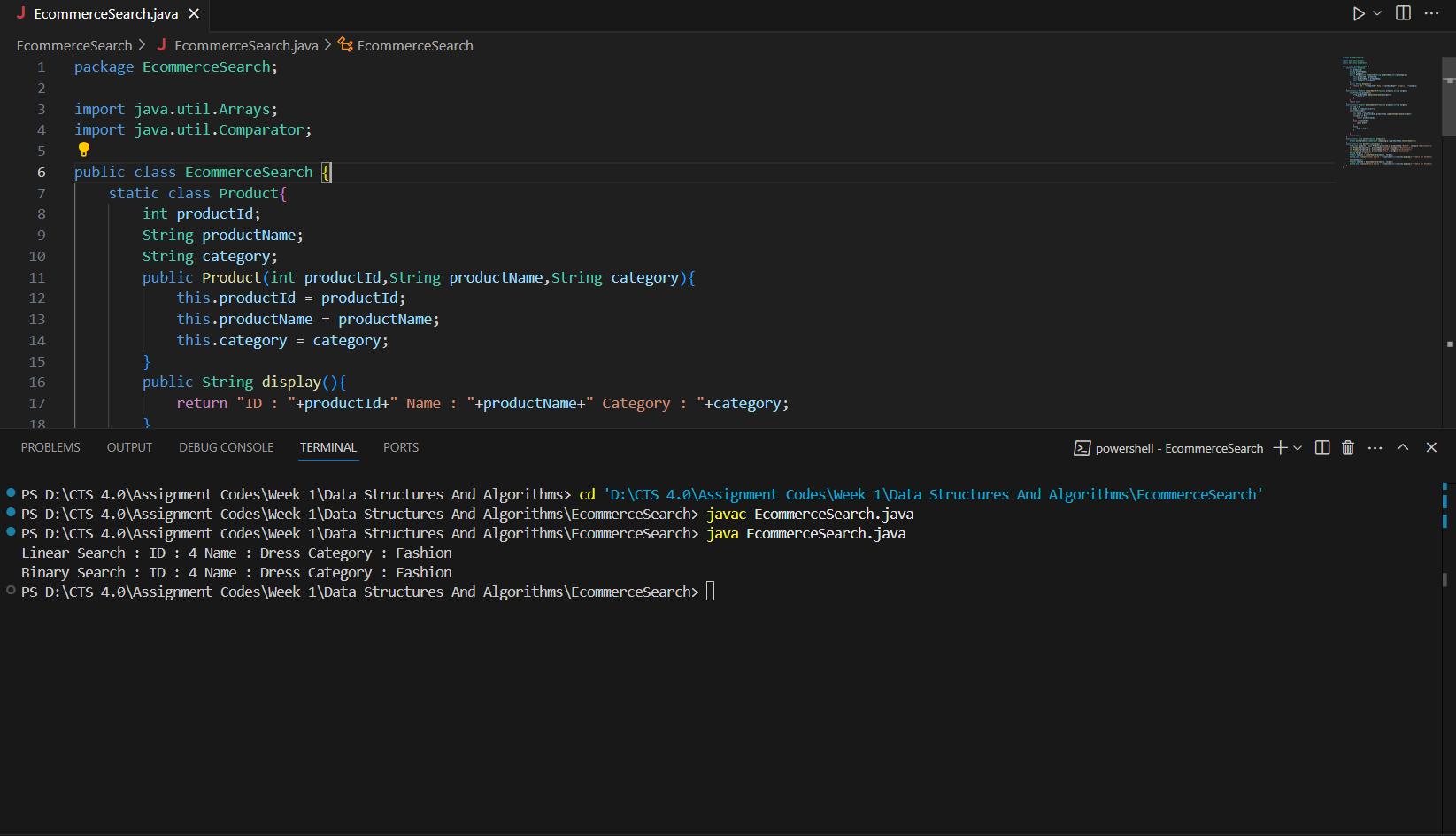
        Product search2 = binarySearch(products, target);

        System.out.println("Binary Search : "+(search2!=null?search2.display():"Product Not Found"));

    }

}

**Output:**

****

**Linear Search:**

Time Complexity – O(n)

Space Complexity – O(1)

**Binary Search:**

Time Complexity – O(log n)

Space Complexity – O(1)

Linear Search is simple and works on unsorted data. Whereas binary search is more efficient for large datasets but requires sorted data.

For e-commerce platforms with large data sets Binary Search is better for fast performance. So, binary search is good algorithm for searching approach in an E-commerce platforms.

**Exercise 3: Sorting Customer Orders**

**Sorting Algorithms:**

**Bubble Sort :** Repeatedly compares adjacent elements and swaps them if they are in the wrong order.

**Insertion Sort :** Builds the sorted array one element at a time by inserting elements into their correct position.

**Quick Sort :** A divide-and-conquer algorithm. Selects a pivot, partitions the array, and recursively sorts subarrays.

**Merge Sort :** Divides array into halves, recursively sorts, and merges sorted halves.

**Code:**

package Sorting;

public class Sorting {

    static class Order {

        String orderId;

        String customerName;

        double totalPrice;

        public Order(String orderId, String customerName, double totalPrice) {

            this.orderId = orderId;

            this.customerName = customerName;

            this.totalPrice = totalPrice;

        }

        public void printOrder() {

            System.out.println(orderId + "  " + customerName + "  " + totalPrice);

        }

    }

    public static void bubbleSort(Order[] orders) {

        int n = orders.length;

        for (int i = 0; i < n - 1; i++) {

            for (int j = 0; j < n - 1 - i; j++) {

                if (orders[j].totalPrice > orders[j + 1].totalPrice) {

                    Order temp = orders[j];

                    orders[j] = orders[j + 1];

                    orders[j + 1] = temp;

                }

            }

        }

    }

    public static void quickSort(Order[] orders, int low, int high) {

        if (low < high) {

            int pi = partition(orders, low, high);

            quickSort(orders, low, pi - 1);

            quickSort(orders, pi + 1, high);

        }

    }

    private static int partition(Order[] orders, int low, int high) {

        double pivot = orders[high].totalPrice;

        int i = low - 1;

        for (int j = low; j < high; j++) {

            if (orders[j].totalPrice <= pivot) {

                i++;

                Order temp = orders[i];

                orders[i] = orders[j];

                orders[j] = temp;

            }

        }

        Order temp = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp;

        return i + 1;

    }

    public static void printOrders(Order[] orders) {

        for (Order order : orders) {

            order.printOrder();

        }

    }

    public static void main(String[] args) {

        Order[] orders = {

            new Order("1", "ABC", 200),

            new Order("2", "DEF", 300),

            new Order("3", "GHI", 450),

            new Order("4", "JKL", 550),

            new Order("5", "MNO", 650)

        };

        System.out.println("Original Orders:");

        printOrders(orders);

        bubbleSort(orders);

        System.out.println("After Bubble Sort by totalPrice: ");

        printOrders(orders);

        orders = new Order[] {

             new Order("1", "ABC", 200),

            new Order("2", "DEF", 300),

            new Order("3", "GHI", 450),

            new Order("4", "JKL", 550),

            new Order("5", "MNO", 650)

        };

        quickSort(orders, 0, orders.length - 1);

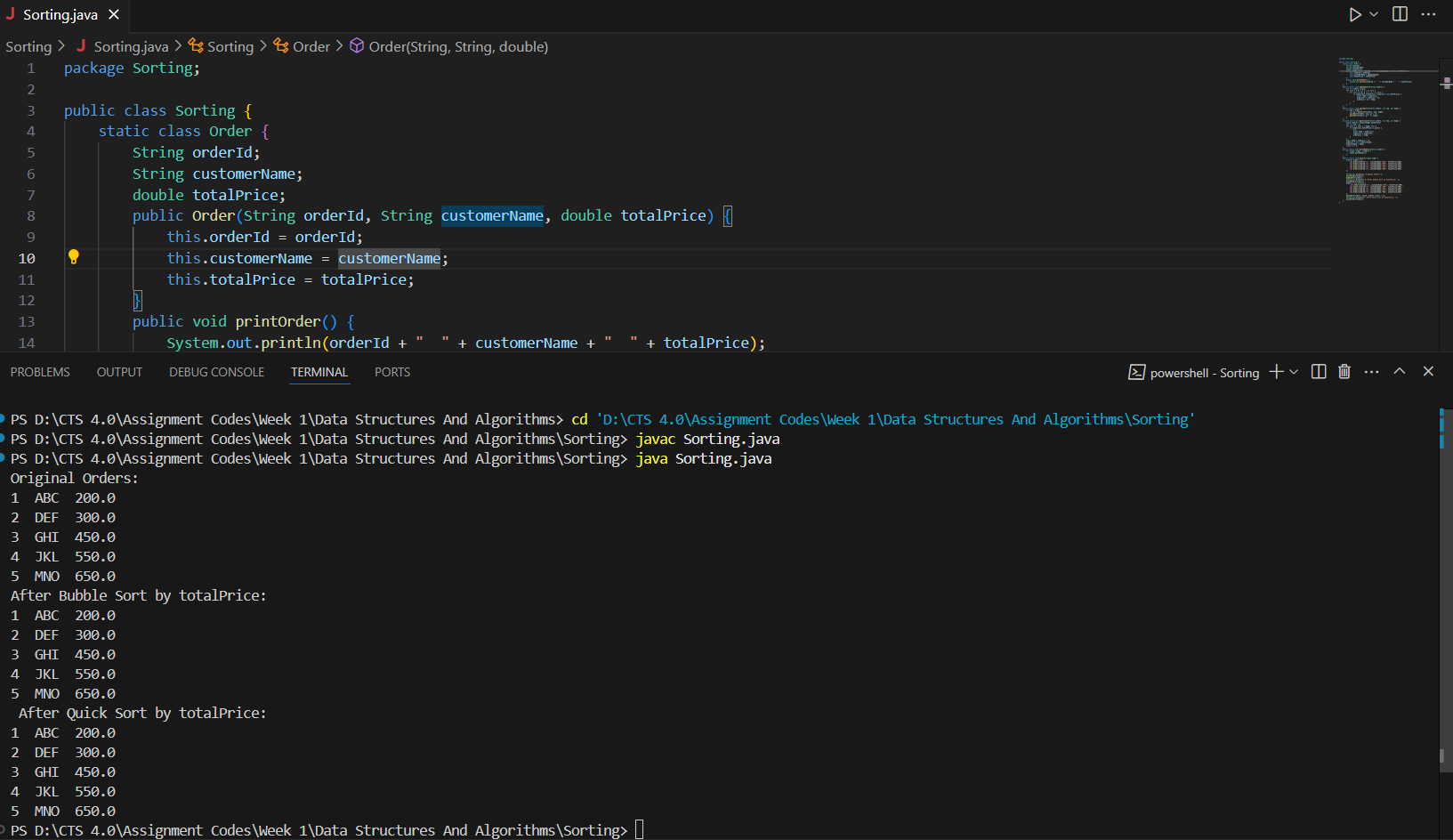
        System.out.println(" After Quick Sort by totalPrice: ");

        printOrders(orders);

    }

}

**Output:**

****

**Time Complexity and Performance:**

**Bubble Sort:**

Time Complexity:

i)Best Case – O(n)

ii)Average Case – O(n^2)

iii)Worst Case – O(n^2)

Space Complexity – O(1)

**Quick Sort:**

Time Complexity:

i)Best Case – O(n log n)

ii)Average Case – O(n log n)

iii)Worst Case – O(n^2)

Space Complexity – O(log n)

Bubble Sort is simple but inefficient for large datasets. Quick sort is generally faster because it reduces the number of comparison via divide and conquer approach. So, quick sort is preferred.

**Exercise 4: Employee Management System**

Arrays are contiguous blocks of memory where each element is stored one after another. The address of the first element and an index allow quick access to any element in O(1) time using base + index \* size.

**Advantages of Array:**

Fast random access using index-O(1).

Simple to use for fixed-size data.

Better cache locality, enhancing performance.

**Code:**

package EmplayeeManagementSystem;

public class EmployeeManagementSystem {

    static class Employee {

        int employeeId;

        String name;

        String position;

        double salary;

        public Employee(int employeeId, String name, String position, double salary) {

            this.employeeId = employeeId;

            this.name = name;

            this.position = position;

            this.salary = salary;

        }

        public void printDetails() {

            System.out.println(employeeId + "  " + name + "  " + position + "  " + salary);

        }

    }

    static final int max = 100;

    static Employee[] employees = new Employee[max];

    static int count = 0;

    public static void addEmployee(Employee emp) {

        if (count < max) {

            employees[count++] = emp;

            System.out.println("Employee added : " + emp.name);

        } else {

            System.out.println("Cannot add more employees Array is full.");

        }

    }

    public static void searchEmployee(int id) {

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

            if (employees[i].employeeId == id) {

                System.out.println("Employee Found:");

                employees[i].printDetails();

                return;

            }

        }

        System.out.println("Employee with ID " + id + " not found.");

    }

    public static void displayEmployees() {

        if (count == 0) {

            System.out.println("No employees to display.");

            return;

        }

        System.out.println("Employee List:");

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

            employees[i].printDetails();

        }

    }

    public static void deleteEmployee(int id) {

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

            if (employees[i].employeeId == id) {

                for (int j = i; j < count - 1; j++) {

                    employees[j] = employees[j + 1];

                }

                employees[--count] = null;

                System.out.println("Employee with ID " + id + " deleted.");

                return;

            }

        }

        System.out.println("Employee with ID " + id + " not found.");

    }

    public static void main(String[] args) {

        addEmployee(new Employee(201, "ABC", "Manager", 80000));

        addEmployee(new Employee(202, "DEF", "Developer", 60000));

        addEmployee(new Employee(203, "GHI", "Designer", 55000));

        System.out.println();

        displayEmployees();

        System.out.println("Searching for employee with ID 202:");

        searchEmployee(202);

        System.out.println();

        System.out.println("Deleting employee with ID 201:");

        deleteEmployee(201);

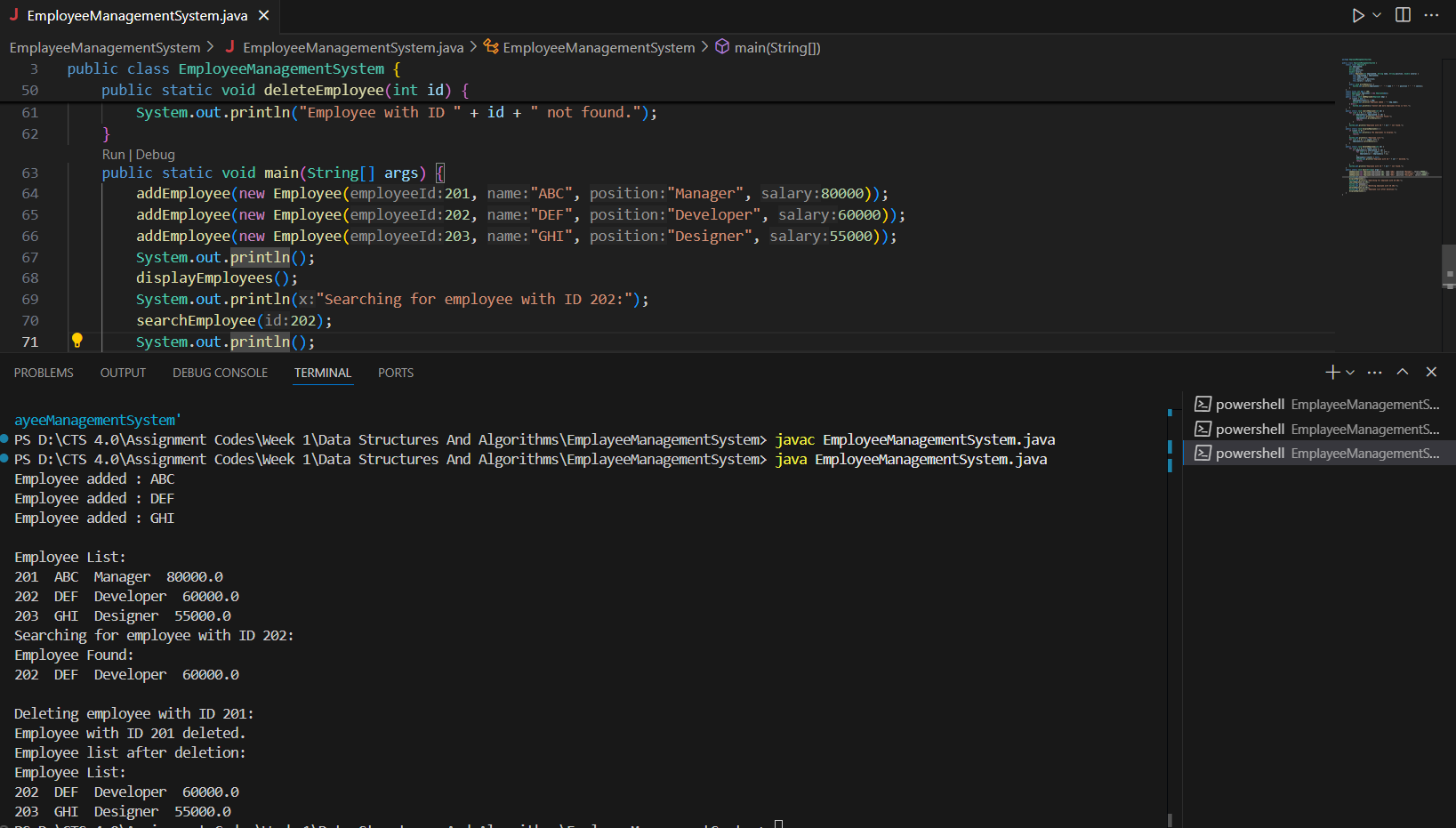
        System.out.println("Employee list after deletion:");

        displayEmployees();

    }

}

**Output:**

****

**Time Complexity of Operations:**

Add-O(1)

Search- O(n)

Traverse- O(n)

Delete- O(n)

**Limitations of Arrays:**

Fixed Size cannot dynamically grow or shrink.

Inefficient Deletion/Insertion shifting elements takes time.

Wasted Space pre allocated space may go unused.

**When to Use Arrays:**

When the number of elements is known and fixed.

When fast index-based access is required.

When memory usage is predictable.

**Exercise 5: Task Management System**

**Understand Linked Lists**

**Singly Linked List:**

Each node contains data and a reference to the next node.

Traversal is possible in only one direction.

Efficient insertion/deletion at the beginning.

Used when memory efficiency is more important than bidirectional navigation.

**Doubly Linked List:**

Each node contains data, a reference to the next node, and a reference to the previous node.

Allows traversal in both directions.

More flexible but requires extra memory per node.

**Code:**

package TaskManagementSystem;

public class TaskManagementSystem {

    static class Task {

        int taskId;

        String taskName;

        String status;

        Task next;

        public Task(int taskId, String taskName, String status) {

            this.taskId = taskId;

            this.taskName = taskName;

            this.status = status;

            this.next = null;

        }

        public void printTask() {

            System.out.println(taskId + "  " + taskName + "  " + status);

        }

    }

    static class TaskLinkedList {

        Task head = null;

        public void addTask(int taskId, String taskName, String status) {

            Task newTask = new Task(taskId, taskName, status);

            if (head == null) {

                head = newTask;

            } else {

                Task temp = head;

                while (temp.next != null) {

                    temp = temp.next;

                }

                temp.next = newTask;

            }

            System.out.println("Task added: " + taskName);

        }

        public void searchTask(int taskId) {

            Task temp = head;

            while (temp != null) {

                if (temp.taskId == taskId) {

                    System.out.println("Task Found:");

                    temp.printTask();

                    return;

                }

                temp = temp.next;

            }

            System.out.println("Task with ID " + taskId + " not found.");

        }

        public void displayTasks() {

            if (head == null) {

                System.out.println("No tasks available.");

                return;

            }

            System.out.println("Task List:");

            Task temp = head;

            while (temp != null) {

                temp.printTask();

                temp = temp.next;

            }

        }

        public void deleteTask(int taskId) {

            if (head == null) {

                System.out.println("List is empty.");

                return;

            }

            if (head.taskId == taskId) {

                head = head.next;

                System.out.println("Task with ID " + taskId + " deleted.");

                return;

            }

            Task current = head;

            Task previous = null;

            while (current != null && current.taskId != taskId) {

                previous = current;

                current = current.next;

            }

            if (current == null) {

                System.out.println("Task with ID " + taskId + " not found.");

            } else {

                previous.next = current.next;

                System.out.println("Task with ID " + taskId + " deleted.");

            }

        }

    }

    public static void main(String[] args) {

        TaskLinkedList taskList = new TaskLinkedList();

        taskList.addTask(1, "Front End", "Pending");

        taskList.addTask(2, "Back End", "In Progress");

        taskList.addTask(3, "Database", "Not Started");

        System.out.println();

        taskList.displayTasks();

        System.out.println();

        System.out.println("Searching for Task ID 2:");

        taskList.searchTask(2);

        System.out.println();

        System.out.println("Deleting Task ID 1:");

        taskList.deleteTask(1);

        System.out.println();

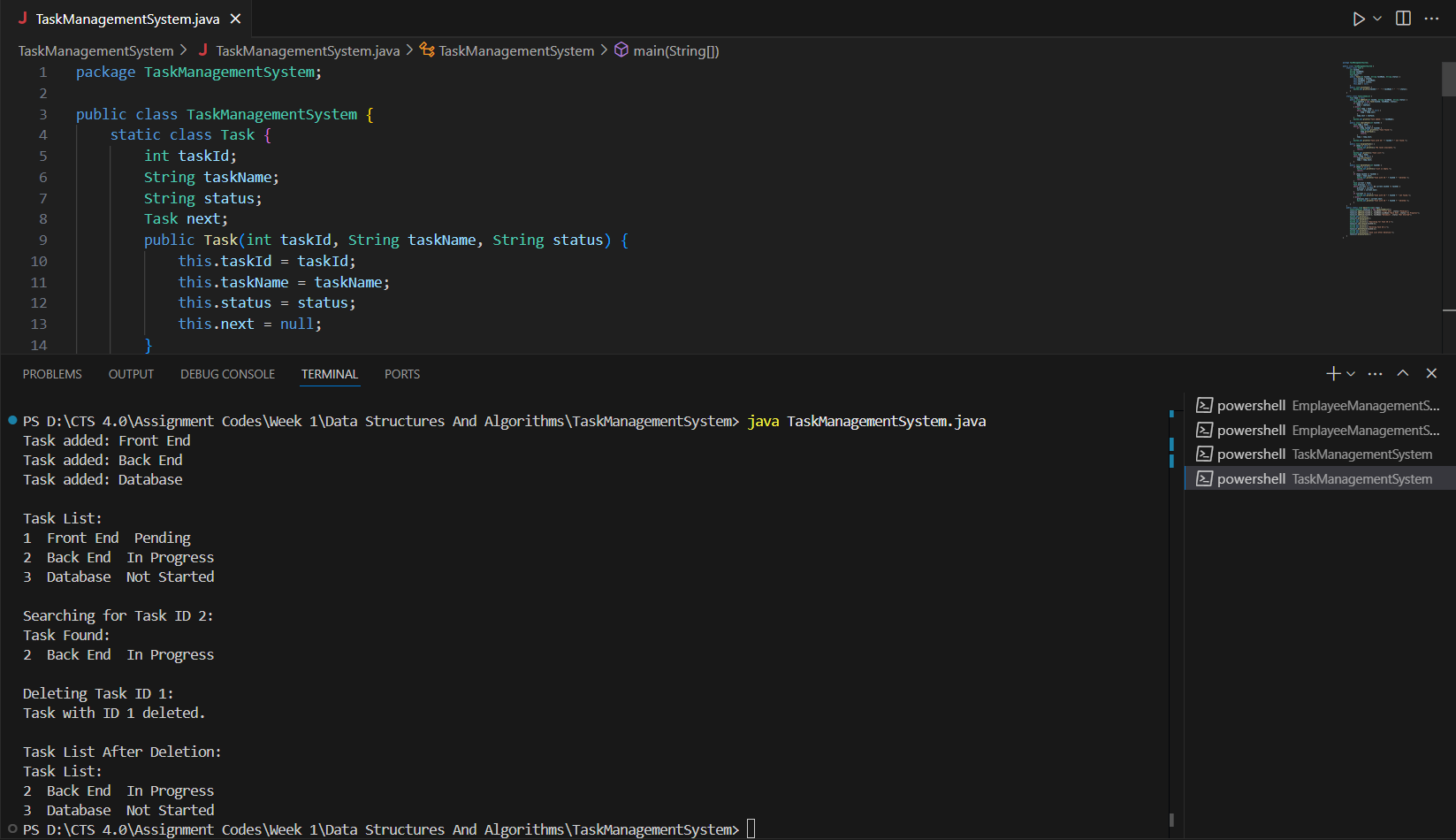
        System.out.println("Task List After Deletion:");

        taskList.displayTasks();

    }

}

**Output:**



**Exercise 6: Library Management System**

**Linear Search:**

Checks each element one by one until the target is found or end of list is reached.

**Binary Search:**

Repeatedly divides the sorted list in half to search for the target.

**Code:**

package LibraryManagementSystem;

import java.util.Arrays;

import java.util.Comparator;

public class LibraryManagementSystem {

    static class Book {

        int bookId;

        String title;

        String author;

        public Book(int bookId, String title, String author) {

            this.bookId = bookId;

            this.title = title;

            this.author = author;

        }

        public void printDetails() {

            System.out.println(bookId + "  " + title + "  " + author);

        }

    }

    static Book[] books;

    public static void linearSearchByTitle(String title) {

        boolean found = false;

        for (Book book : books) {

            if (book.title.equalsIgnoreCase(title)) {

                System.out.println("Book Found using Linear Search:");

                book.printDetails();

                found = true;

                break;

            }

        }

        if (!found) {

            System.out.println("Book not found using Linear Search.");

        }

    }

    public static void binarySearchByTitle(String title) {

        int left = 0;

        int right = books.length - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

            int comparison = title.compareToIgnoreCase(books[mid].title);

            if (comparison == 0) {

                System.out.println("Book Found using Binary Search:");

                books[mid].printDetails();

                return;

            } else if (comparison < 0) {

                right = mid - 1;

            } else {

                left = mid + 1;

            }

        }

        System.out.println("Book not found using Binary Search.");

    }

    public static void sortBooksByTitle() {

        Arrays.sort(books, Comparator.comparing(book -> book.title.toLowerCase()));

    }

    public static void main(String[] args) {

        books = new Book[] {

            new Book(101, "ABC", "DEF"),

            new Book(102, "GHI", "JKL"),

            new Book(103, "MNO", "PQR"),

            new Book(104, "STU", "VWX"),

        };

        System.out.println("Linear Search:");

        linearSearchByTitle("GHI");

        System.out.println();

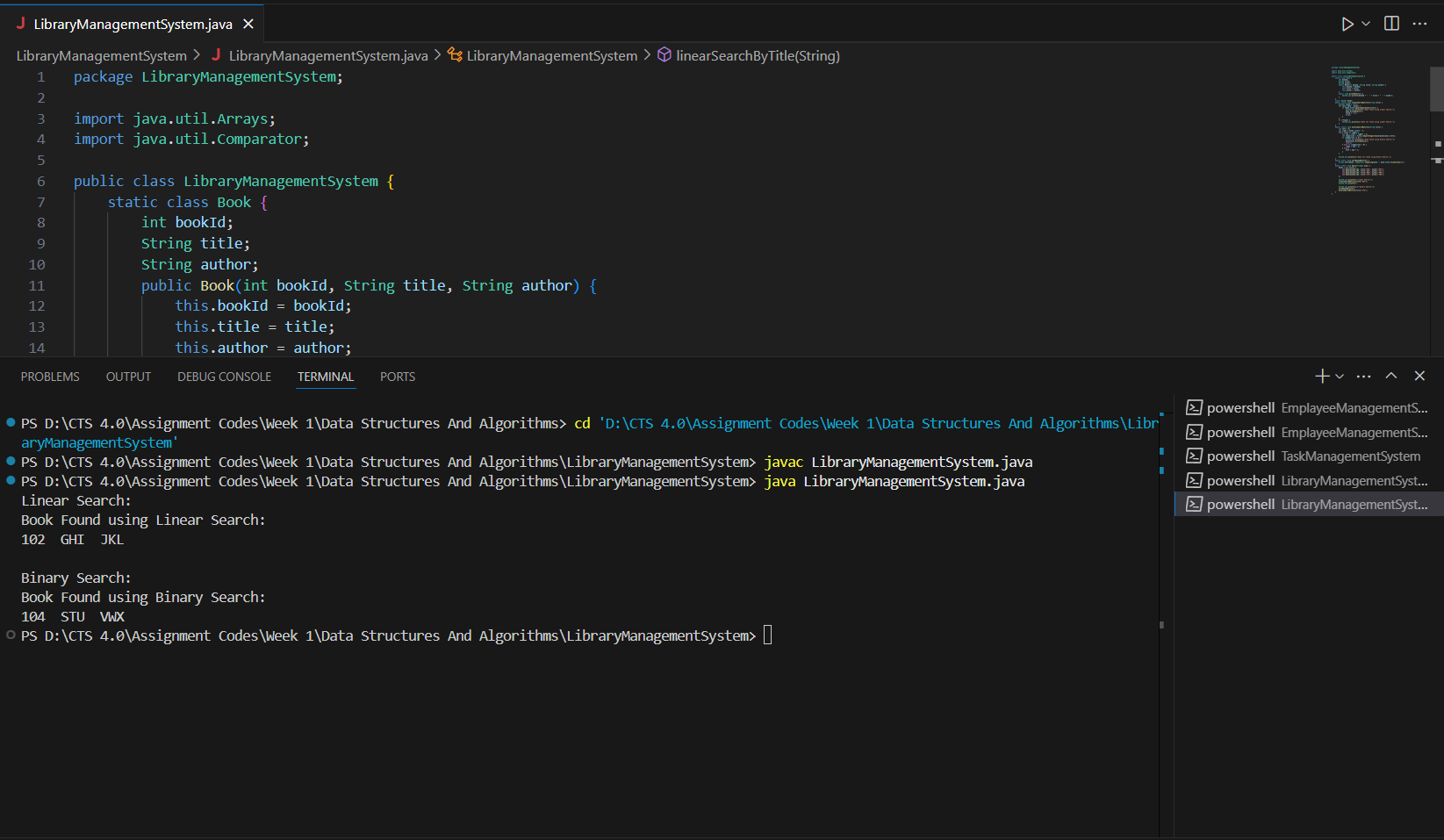
        System.out.println("Binary Search:");

        sortBooksByTitle();

        binarySearchByTitle("STU");

    }

}

**Output:**  


**Time Complexity:**

**i)Linear Search:**

Best Case : O(1)

Average Case : O(n)

Worst Case : O(n)

**ii)Binary Search:**

Best Case : O(1)

Average Case : O(log n)

Worst Case : O(log n)

**Use Linear Search:**

When the list is unsorted or small

When insertion/deletion happens frequently and sorting is costly

**Use Binary Search:**

When the list is **already sorted**

When you have a **large static dataset**

**Exercise 7: Financial Forecasting**

**Recursion:**

Recursion is a programming technique where a method calls itself to solve a smaller subproblem. It simplifies problems that have repetitive or self similar structure such as calculating factorial, Fibonacci numbers or compound interest over time.

**Code:**

package FinancialForecasting;

public class FinancialForecasting {

    public static double futureValue(double initial, double growth, int years) {

        if (years == 0) {

            return initial;

        }

        return futureValue(initial,growth,years-1)\*(1+growth);

    }

    public static void main(String[] args) {

        double initial = 10000;

        double growth = 0.07;

        int years = 5;

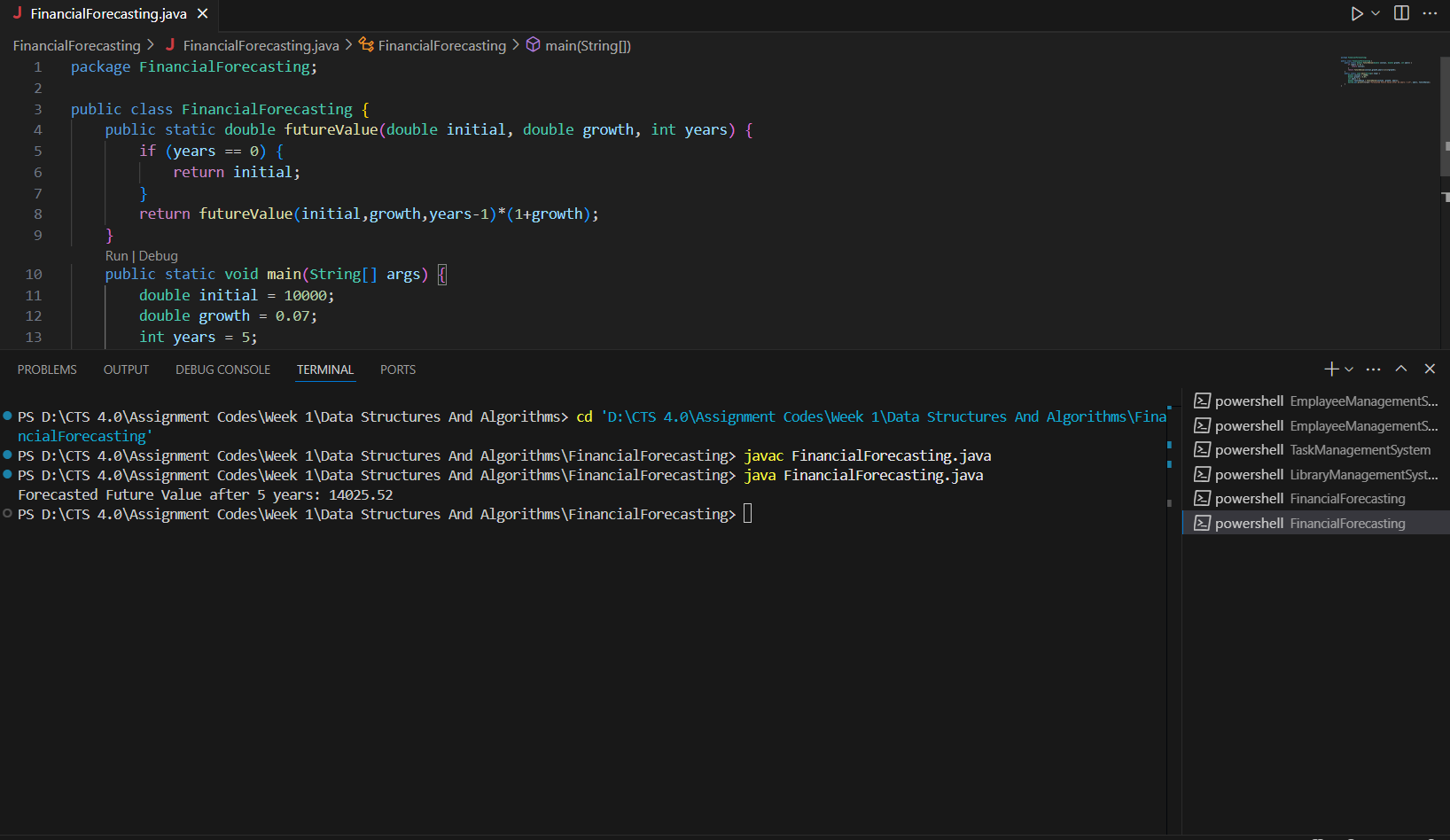
        double futureValue = futureValue(initial, growth, years);

        System.out.printf("Forecasted Future Value after %d years: %.2f", years, futureValue);

    }

}

**Output:**

****

Time Complexity – O(n)

Space Complexity – O(n)

**Why Optimization is Needed?**

Although the recursion works here deeper or repeated recursion without caching may lead to

Redundant calculations.

Stack overflow for large n.

**Optimization Technique:**

Use memoization which store already computed values to avoid repeated calls.

For even better performance with low overhead, consider using iteration instead of recursion.