**Student Name :- Dharan Gowda H E Track:-Java FSE**

**Exercise 1: Inventory Management System**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

Why Data Structures and Algorithms Are Important:

1. Large warehouses handle thousands of products, making efficient search, insert, and delete operations crucial.
2. Proper data structures ensure fast access, reduce memory usage, and improve overall system performance.
3. Algorithms optimize how operations are executed, especially in scenarios like bulk updates, sorting, or filtering.

Suitable Data Structures:

1. ArrayList – Good for ordered lists, but searching by productId is O(n).
2. HashMap<String, Product> – Ideal for key-based access (productId as key), giving O(1) time for add, update, delete, and search.
3. TreeMap<String, Product> – Maintains sorted order by key, useful if sorted traversal is needed, with O(log n) operations.

import java.util.HashMap;

public class Exercise1 {

    // Static inner class Product

    public static class Product {

        private String productId;

        private String productName;

        private int quantity;

        private double price;

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

            this.productId = productId;

            this.productName = productName;

            this.quantity = quantity;

            this.price = price;

        }

        public String getProductId() { return productId; }

        public String getProductName() { return productName; }

        public int getQuantity() { return quantity; }

        public double getPrice() { return price; }

        public void setProductName(String productName) { this.productName = productName; }

        public void setQuantity(int quantity) { this.quantity = quantity; }

        public void setPrice(double price) { this.price = price; }

        @Override

        public String toString() {

            return "[" + productId + "] " + productName + " - Qty: " + quantity + ", Price: " + price;

        }

    }

    // Static inner class Inventory

    public static class Inventory {

        private HashMap<String, Product> products = new HashMap<>();

        public void addProduct(Product product) {

            products.put(product.getProductId(), product);

        }

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

            Product product = products.get(productId);

            if (product != null) {

                product.setQuantity(quantity);

                product.setPrice(price);

                return true;

            }

            return false;

        }

        public boolean deleteProduct(String productId) {

            return products.remove(productId) != null;

        }

        public Product getProduct(String productId) {

            return products.get(productId);

        }

        public void displayInventory() {

            for (Product product : products.values()) {

                System.out.println(product);

            }

        }

    }

    // Main method

    public static void main(String[] args) {

        Inventory inventory = new Inventory();

        inventory.addProduct(new Product("P001", "Laptop", 10, 799.99));

        inventory.addProduct(new Product("P002", "Mouse", 50, 15.99));

        inventory.addProduct(new Product("P003", "Keyboard", 30, 25.49));

        System.out.println("Initial Inventory:");

        inventory.displayInventory();

        inventory.updateProduct("P001", 8, 749.99);  // Update Laptop

        inventory.deleteProduct("P002");             // Delete Mouse

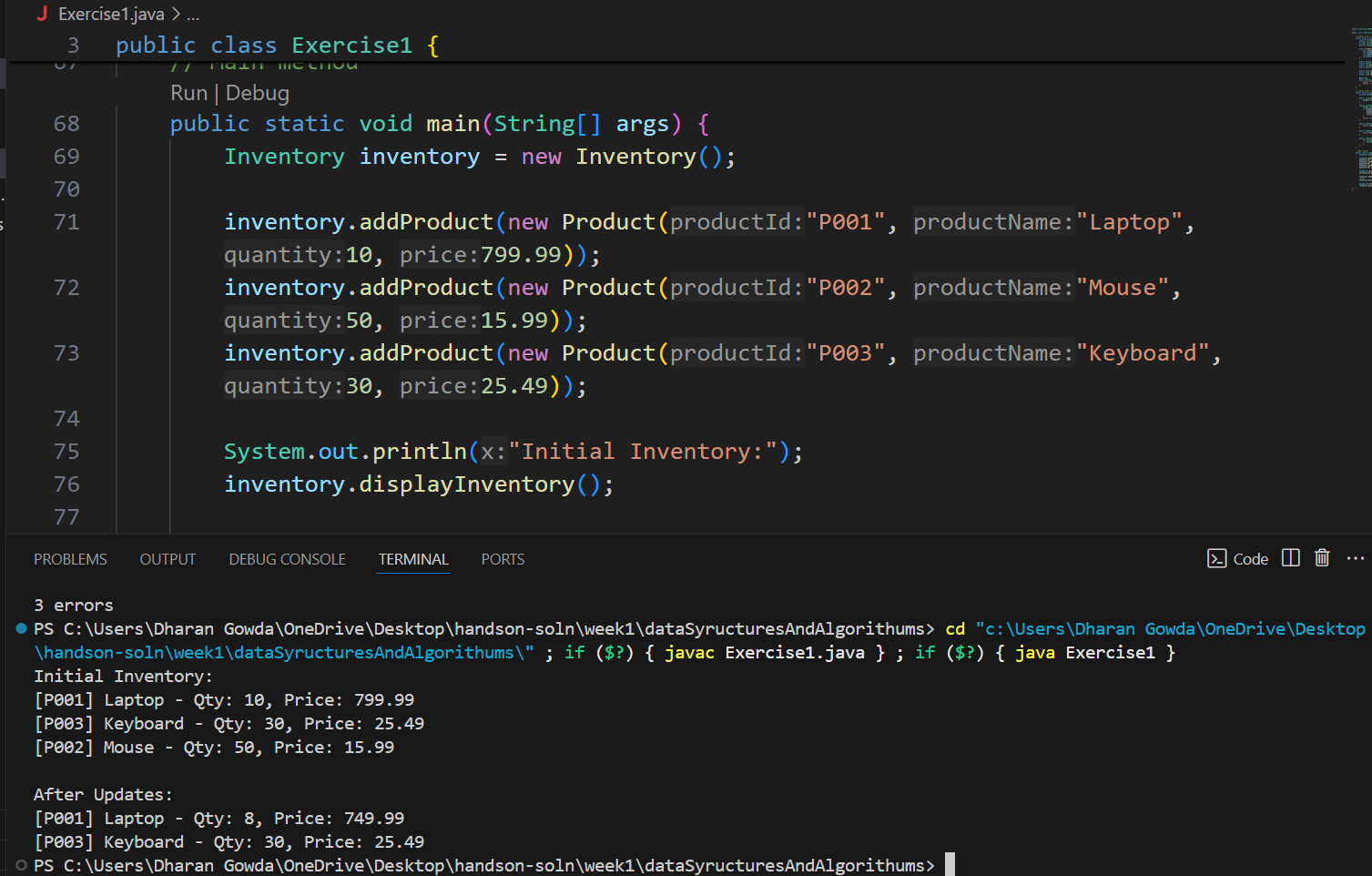
        System.out.println("\nAfter Updates:");

        inventory.displayInventory();

    }

}

**OUTPUT:-**



**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.

**1. Understand Asymptotic Notation**

Big O Notation:

* Big O notation describes the upper bound of an algorithm’s running time as input size grows.
* It helps compare algorithms based on their scalability and performance.

Search Complexity Scenarios**:**

| Case | Linear Search (Unsorted) | Binary Search (Sorted) |
| --- | --- | --- |
| Best Case | O(1) – | O(1) – | |
| Average Case | O(n/2) ≈ O(n) | O(log n) |
| Worst Case | O(n) | O(log n) |

**CODE :-**

import java.util.Arrays;

import java.util.Comparator;

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; }

    @Override

    public String toString() {

        return "[" + productId + "] " + productName + " (" + category + ")";

    }

}

public class EcommerceSearch {

    // Linear Search

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

        for (Product product : products) {

            if (product.getProductName().equalsIgnoreCase(targetName)) {

                return product;

            }

        }

        return null;

    }

    // Binary Search

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

        int low = 0, high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int cmp = products[mid].getProductName().compareToIgnoreCase(targetName);

            if (cmp == 0) return products[mid];

            else if (cmp < 0) low = mid + 1;

            else high = mid - 1;

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] products = {

            new Product("P001", "Laptop", "Electronics"),

            new Product("P002", "Mouse", "Electronics"),

            new Product("P003", "Shirt", "Clothing"),

            new Product("P004", "Book", "Stationery"),

            new Product("P005", "Keyboard", "Electronics")

        };

        // Linear Search

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

        Product result1 = linearSearch(products, "Shirt");

        System.out.println(result1 != null ? result1 : "Product not found");

        // Sort before binary search

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

        // Binary Search

        System.out.println("\nBinary Search (after sorting):");

        Product result2 = binarySearch(products, "Shirt");

        System.out.println(result2 != null ? result2 : "Product not found");

    } import java.util.Arrays;

import java.util.Comparator;

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; }

    @Override

    public String toString() {

        return "[" + productId + "] " + productName + " (" + category + ")";

    }

}

public class EcommerceSearch {

    // Linear Search

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

        for (Product product : products) {

            if (product.getProductName().equalsIgnoreCase(targetName)) {

                return product;

            }

        }

        return null;

    }

    // Binary Search

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

        int low = 0, high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int cmp = products[mid].getProductName().compareToIgnoreCase(targetName);

            if (cmp == 0) return products[mid];

            else if (cmp < 0) low = mid + 1;

            else high = mid - 1;

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] products = {

            new Product("P001", "Laptop", "Electronics"),

            new Product("P002", "Mouse", "Electronics"),

            new Product("P003", "Shirt", "Clothing"),

            new Product("P004", "Book", "Stationery"),

            new Product("P005", "Keyboard", "Electronics")

        };

        // Linear Search

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

        Product result1 = linearSearch(products, "Shirt");

        System.out.println(result1 != null ? result1 : "Product not found");

        // Sort before binary search

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

        // Binary Search

        System.out.println("\nBinary Search (after sorting):");

        Product result2 = binarySearch(products, "Shirt");

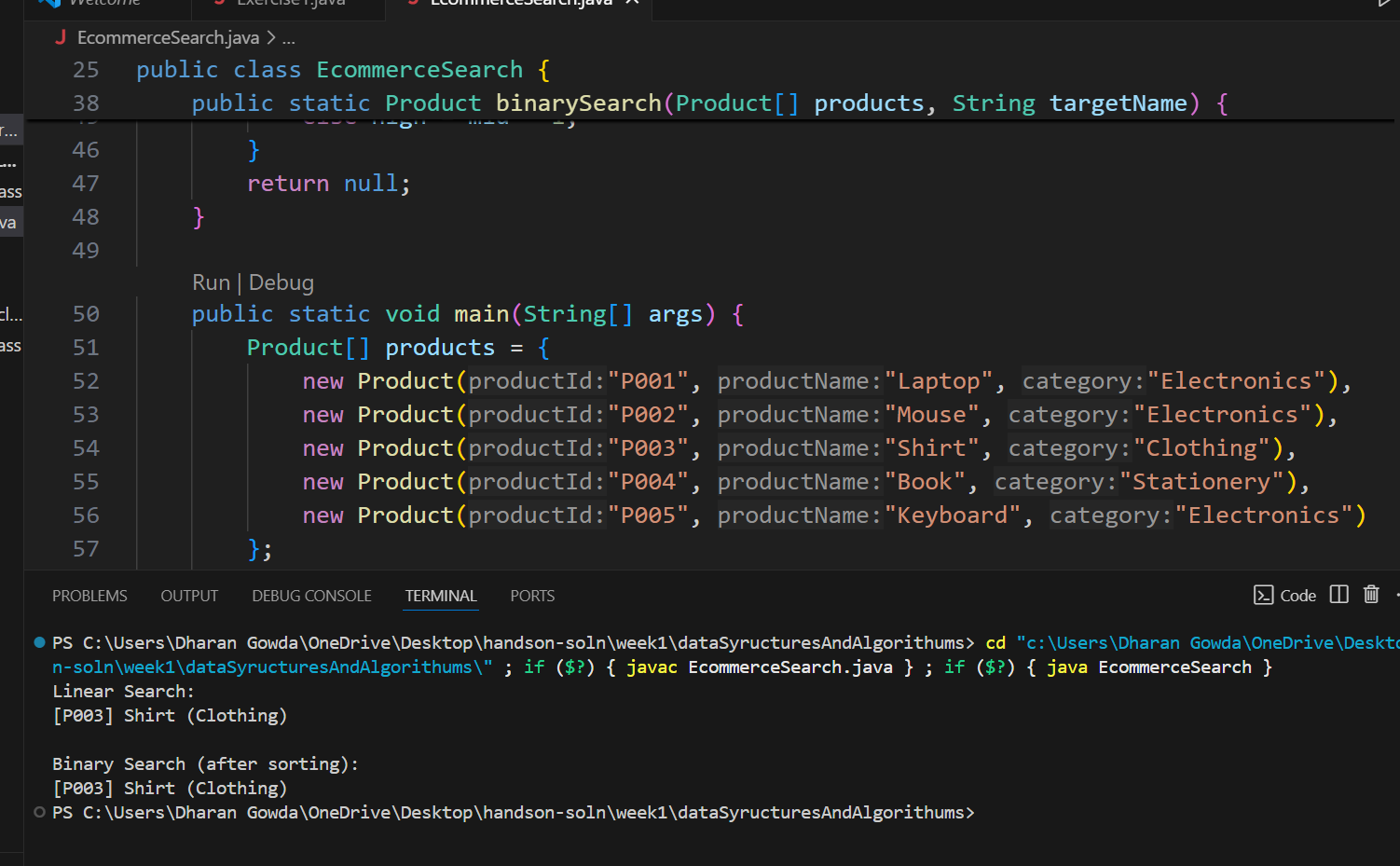
        System.out.println(result2 != null ? result2 : "Product not found");

    }

}

}

**Output:-**

****

**Analysis**

| Algorithm | Time Complexity | Suitable When |
| --- | --- | --- |
| Linear Search | O(n) | Data is unsorted or small |
| Binary Search | O(log n) | Data is sorted |

**Exercise 3: Sorting Customer Orders**

**1. Understand Sorting Algorithms**

🔹 Bubble Sort:

* Repeatedly compares adjacent elements and swaps if they’re in the wrong order.
* Simple but inefficient for large datasets.

🔹 Insertion Sort:

* Builds a sorted list one element at a time by inserting into the correct position.
* Efficient for small or nearly sorted data.

🔹 Quick Sort:

* Uses divide and conquer.
* Selects a pivot and partitions elements into less and greater than pivot.
* Fastest in practice for large, unsorted datasets.

🔹 Merge Sort:

* Recursively divides array into halves, sorts and merges them.
* Stable and predictable O(n log n), but requires extra space

**Code:-**

import java.util.Arrays;

class Order {

    private String orderId;

    private String customerName;

    private double totalPrice;

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

        this.orderId = orderId;

        this.customerName = customerName;

        this.totalPrice = totalPrice;

    }

    public double getTotalPrice() {

        return totalPrice;

    }

    @Override

    public String toString() {

        return "[" + orderId + "] " + customerName + " - ₹" + totalPrice;

    }

}

public class OrderSorting {

    // Bubble Sort

    public static void bubbleSort(Order[] orders) {

        int n = orders.length;

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

            boolean swapped = false;

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

                if (orders[j].getTotalPrice() < orders[j + 1].getTotalPrice()) {

                    Order temp = orders[j];

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

                    orders[j + 1] = temp;

                    swapped = true;

                }

            }

            if (!swapped) break;

        }

    }

    // Quick Sort

    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].getTotalPrice();

        int i = low - 1;

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

            if (orders[j].getTotalPrice() > 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(String title, Order[] orders) {

        System.out.println(title);

        for (Order order : orders) {

            System.out.println(order);

        }

        System.out.println();

    }

    public static void main(String[] args) {

        Order[] originalOrders = {

            new Order("O001", "Alice", 2999.99),

            new Order("O002", "Bob", 499.50),

            new Order("O003", "Charlie", 7999.75),

            new Order("O004", "Diana", 1599.00),

            new Order("O005", "Eve", 10500.00)

        };

        // Bubble Sort

        Order[] bubbleSorted = Arrays.copyOf(originalOrders, originalOrders.length);

        bubbleSort(bubbleSorted);

        printOrders("Orders sorted by Bubble Sort (High to Low Total Price):", bubbleSorted);

        // Quick Sort

        Order[] quickSorted = Arrays.copyOf(originalOrders, originalOrders.length);

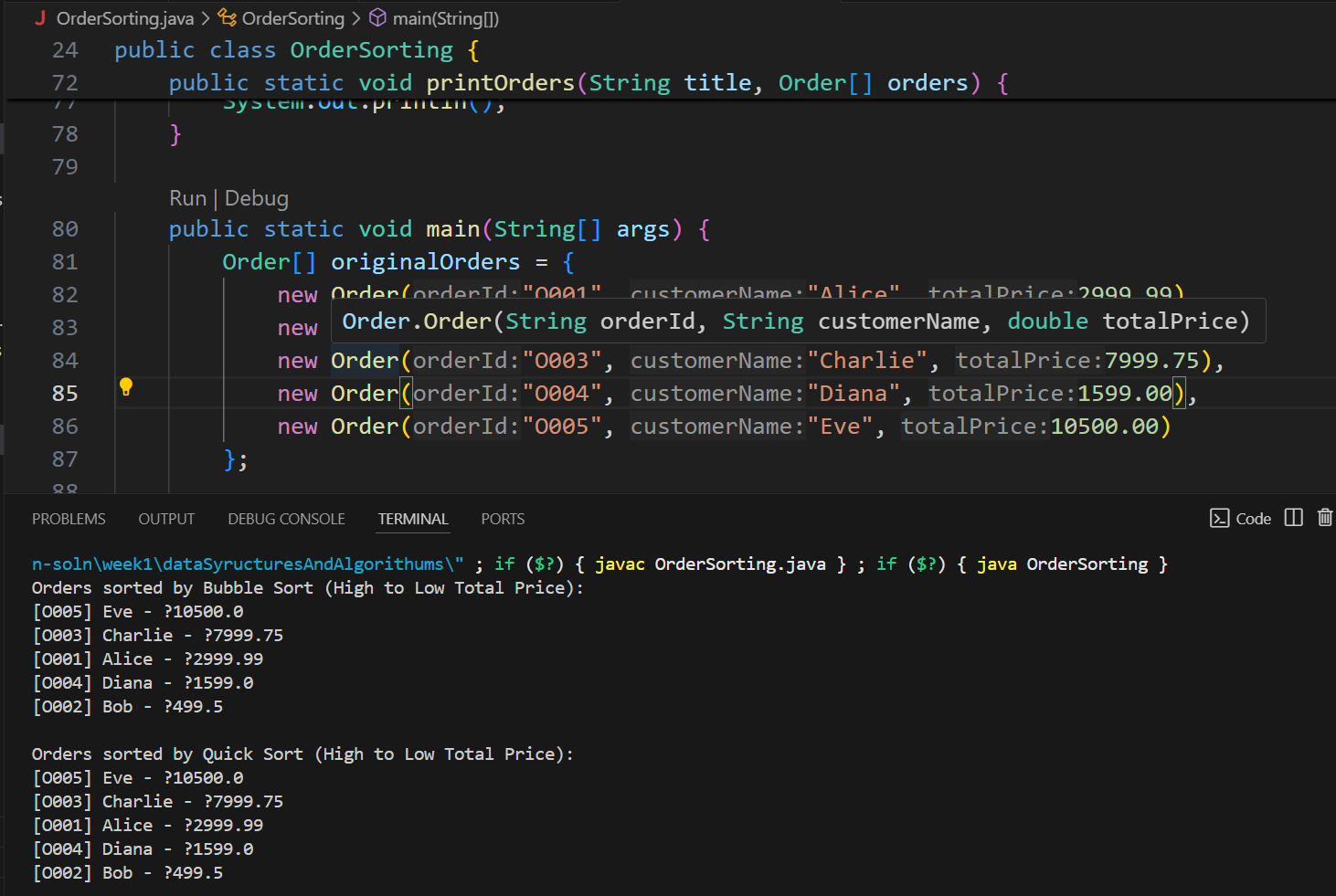
        quickSort(quickSorted, 0, quickSorted.length - 1);

        printOrders("Orders sorted by Quick Sort (High to Low Total Price):", quickSorted);

    }

}

**Output:-**

****

**Sorting Algorithm Summary:**

**1. Bubble Sort**

* Repeatedly swaps adjacent elements if they are in the wrong order.
* Time Complexity:
  + Best: O(n) (already sorted)
  + Average/Worst: O(n²)
* Simple but inefficient for large datasets.

**2. Quick Sort**

* Divide and conquer: selects a pivot and partitions the array.
* Time Complexity:
  + Best/Average: O(n log n)
  + Worst: O(n²) (rare, depends on pivot)
* Much faster in practice, used widely in standard libraries.

**Quick Sort is Preferred:**

* Faster average-case performance.
* Less swapping and fewer passes.
* More scalable for real-world data.

**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**1. Understand Array Representation**

**How Arrays Work in Memory:**

* Arrays are stored in contiguous memory blocks.
* Each element is accessed via an index, starting from 0.
* Access time is constant time O(1) since the address is calculated directly.

**Advantages of Arrays:**

* Fast access via indexing.
* Fixed size leads to predictable memory usage.
* Easy to traverse using loops.

**Code:-**

class Employee {

private int employeeId;

private String name;

private String position;

private 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 int getEmployeeId() {

return employeeId;

}

@Override

public String toString() {

return "[" + employeeId + "] " + name + " - " + position + " - ₹" + salary;

}

}

public class EmployeeManager {

private Employee[] employees;

private int count;

public EmployeeManager(int size) {

employees = new Employee[size];

count = 0;

}

// Add employee

public boolean addEmployee(Employee e) {

if (count < employees.length) {

employees[count++] = e;

return true;

} else {

System.out.println("Employee array is full!");

return false;

}

}

// Search employee by ID

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

// Traverse and display all employees

public void traverseEmployees() {

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

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

System.out.println(employees[i]);

}

}

// Delete employee by ID

public boolean deleteEmployee(int id) {

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

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

// Shift elements to the left

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

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

}

employees[--count] = null;

return true;

}

}

return false;

}

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager(5);

// Adding employees

manager.addEmployee(new Employee(101, "Alice", "Manager", 60000));

manager.addEmployee(new Employee(102, "Bob", "Developer", 50000));

manager.addEmployee(new Employee(103, "Charlie", "Designer", 45000));

// Traversing employees

manager.traverseEmployees();

// Searching for an employee

Employee found = manager.searchEmployee(102);

System.out.println("\nSearch Result: " + (found != null ? found : "Employee not found"));

// Deleting an employee

boolean deleted = manager.deleteEmployee(102);

System.out.println("\nDelete Result: " + (deleted ? "Deleted Successfully" : "Employee not found"));

// Traversing after deletion

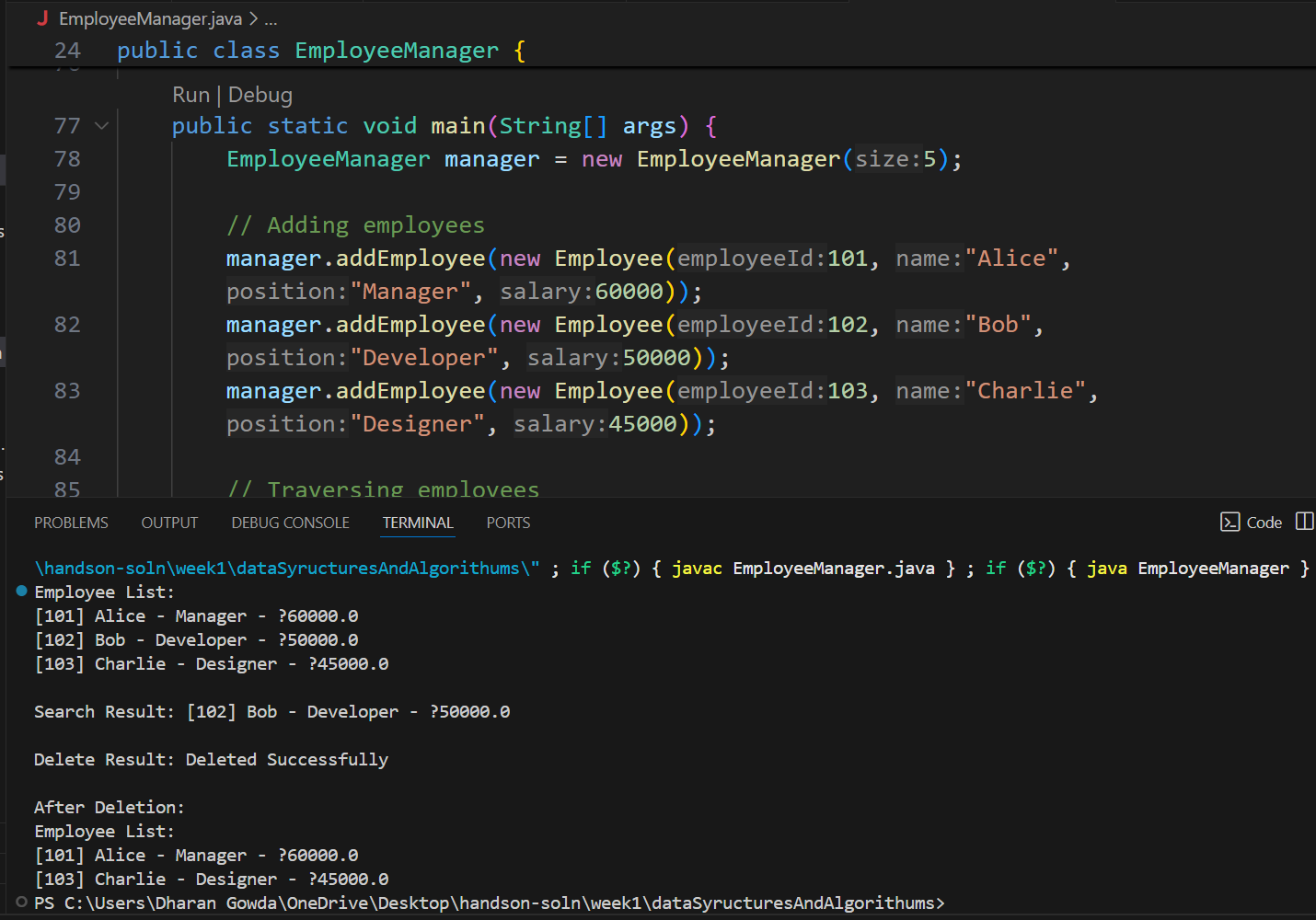
System.out.println("\nAfter Deletion:");

manager.traverseEmployees();

}

}

**Output:-**

****

**Analysis: Time Complexity**

| Operation | Time Complexity | Explanation |
| --- | --- | --- |
| Add | O(1) (amortized) | If index is known, just insert. |
| Search | O(n) | Linear scan to find by ID. |
| Traverse | O(n) | Visit each element. |
| Delete | O(n) | Search + shift elements. |

**Limitations of Arrays:**

* Fixed size — you must predefine capacity.
* Inserting/deleting in the middle is costly due to shifting.
* Not dynamic like ArrayList or LinkedList.

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**1). Understand Linked Lists**

**i) Singly Linked List (SLL):**

* Each node stores data and a reference to the next node.
* Traversal is one-way (forward only).
* Uses less memory than doubly linked list**.**

**ii) Doubly Linked List (DLL):**

* Each node stores data, reference to next and previous nodes.
* Allows forward and backward traversal.
* Slightly more memory and overhead than singly linked list

**CODE:-**

class Task {

int taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

@Override

public String toString() {

return "[" + taskId + "] " + taskName + " - " + status;

}

}

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

class TaskLinkedList {

private TaskNode head;

// Add task at end

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

// Search task by ID

public Task searchTask(int taskId) {

TaskNode current = head;

while (current != null) {

if (current.task.taskId == taskId) {

return current.task;

}

current = current.next;

}

return null;

}

// Traverse and display all tasks

public void traverseTasks() {

TaskNode current = head;

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

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

// Delete task by ID

public boolean deleteTask(int taskId) {

if (head == null) return false;

if (head.task.taskId == taskId) {

head = head.next;

return true;

}

TaskNode current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

return true;

}

return false;

}

}

public class TaskManager {

public static void main(String[] args) {

TaskLinkedList taskList = new TaskLinkedList();

taskList.addTask(new Task(1, "Design UI", "Pending"));

taskList.addTask(new Task(2, "Implement Backend", "Pending"));

taskList.addTask(new Task(3, "Test Features", "Completed"));

taskList.traverseTasks();

// Search

Task found = taskList.searchTask(2);

System.out.println("\nSearch Result: " + (found != null ? found : "Task not found"));

// Delete

boolean deleted = taskList.deleteTask(2);

System.out.println("\nDelete Result: " + (deleted ? "Deleted successfully" : "Task not found"));

// Traverse after deletion

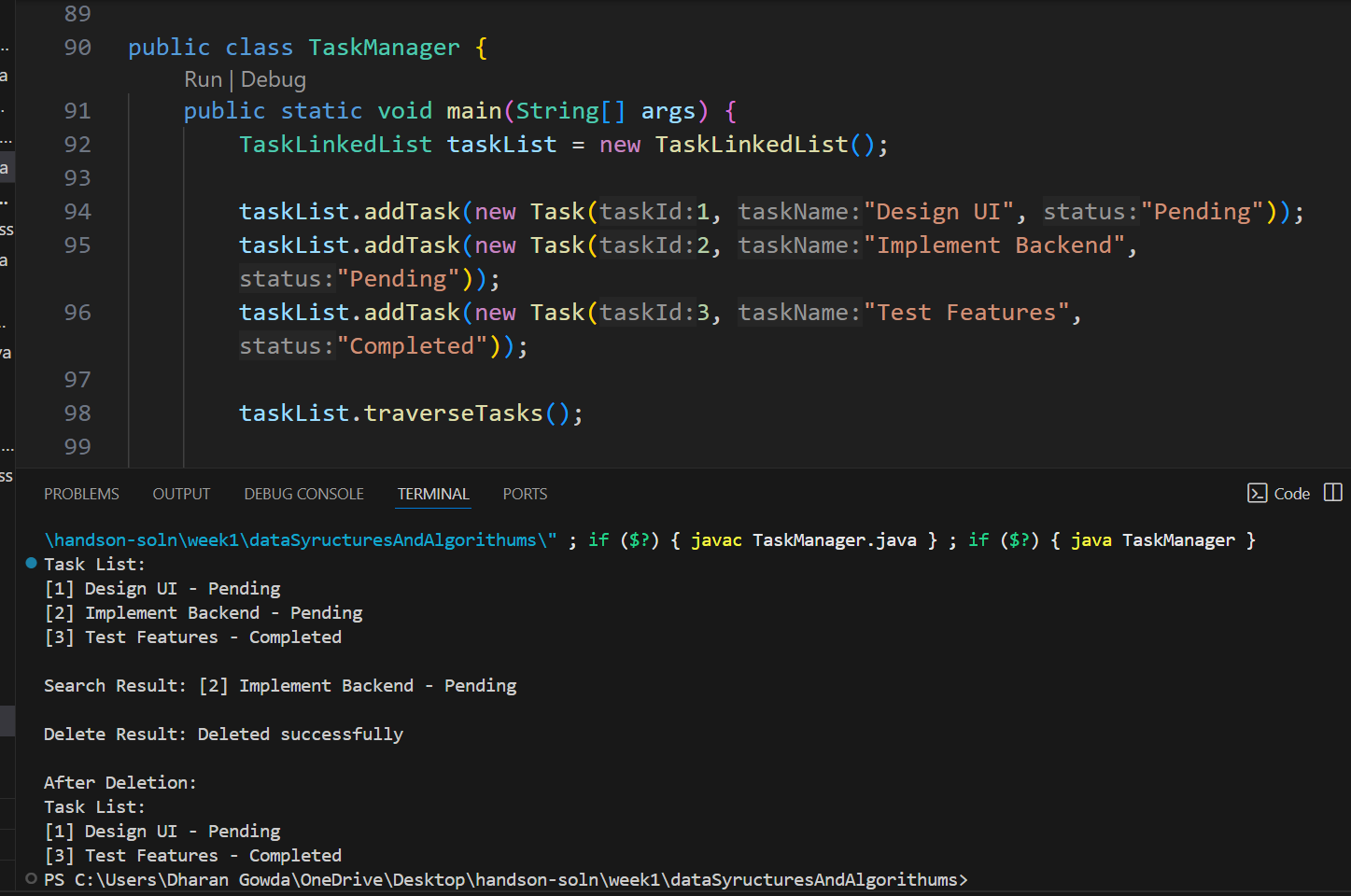
System.out.println("\nAfter Deletion:");

taskList.traverseTasks();

}

}

**Output:-**

****

| Operation | Time Complexity | Explanation |
| --- | --- | --- |
| Add | O(1) (at head) or O(n) (at end) | Need to find position |
| Search | O(n) | Linear scan through list |
| Traverse | O(n) | Visit every node |
| Delete | O(n) | Need to locate node and unlink it |

**Linked List Advantages Over Arrays:**

* Dynamic size — no need to predefine size.
* Easier insertion and deletion (no shifting).
* Efficient for frequent adds/removes in middle or ends.

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**1. Understand Search Algorithms**

**Linear Search:**

* Checks each element one by one.
* Works on unsorted data.
* Simple and safe, but inefficient for large datasets.

**Binary Search:**

* Works only on **sorted** data.
* Repeatedly divides the search space in half.
* Very efficient for large, sorted datasets.

**Code:-**

import java.util.Arrays;

import java.util.Comparator;

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 String getTitle() {

return title;

}

@Override

public String toString() {

return "[" + bookId + "] " + title + " by " + author;

}

}

public class BookSearch {

// Linear Search by title

public static Book linearSearch(Book[] books, String title) {

for (Book book : books) {

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

return book;

}

}

return null;

}

// Binary Search by title (assuming sorted)

public static Book binarySearch(Book[] books, String title) {

int low = 0, high = books.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) return books[mid];

else if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void printSearchResult(String method, Book result) {

System.out.println(method + " Result: " + (result != null ? result : "Book not found"));

}

public static void main(String[] args) {

Book[] books = {

new Book(1, "Java Basics", "Alice"),

new Book(2, "Data Structures", "Bob"),

new Book(3, "Operating Systems", "Charlie"),

new Book(4, "Networking", "Diana"),

new Book(5, "Algorithms", "Eve")

};

// Linear Search

Book linearResult = linearSearch(books, "Networking");

printSearchResult("Linear Search", linearResult);

// Sort books by title before binary search

Arrays.sort(books, Comparator.comparing(Book::getTitle));

// Binary Search

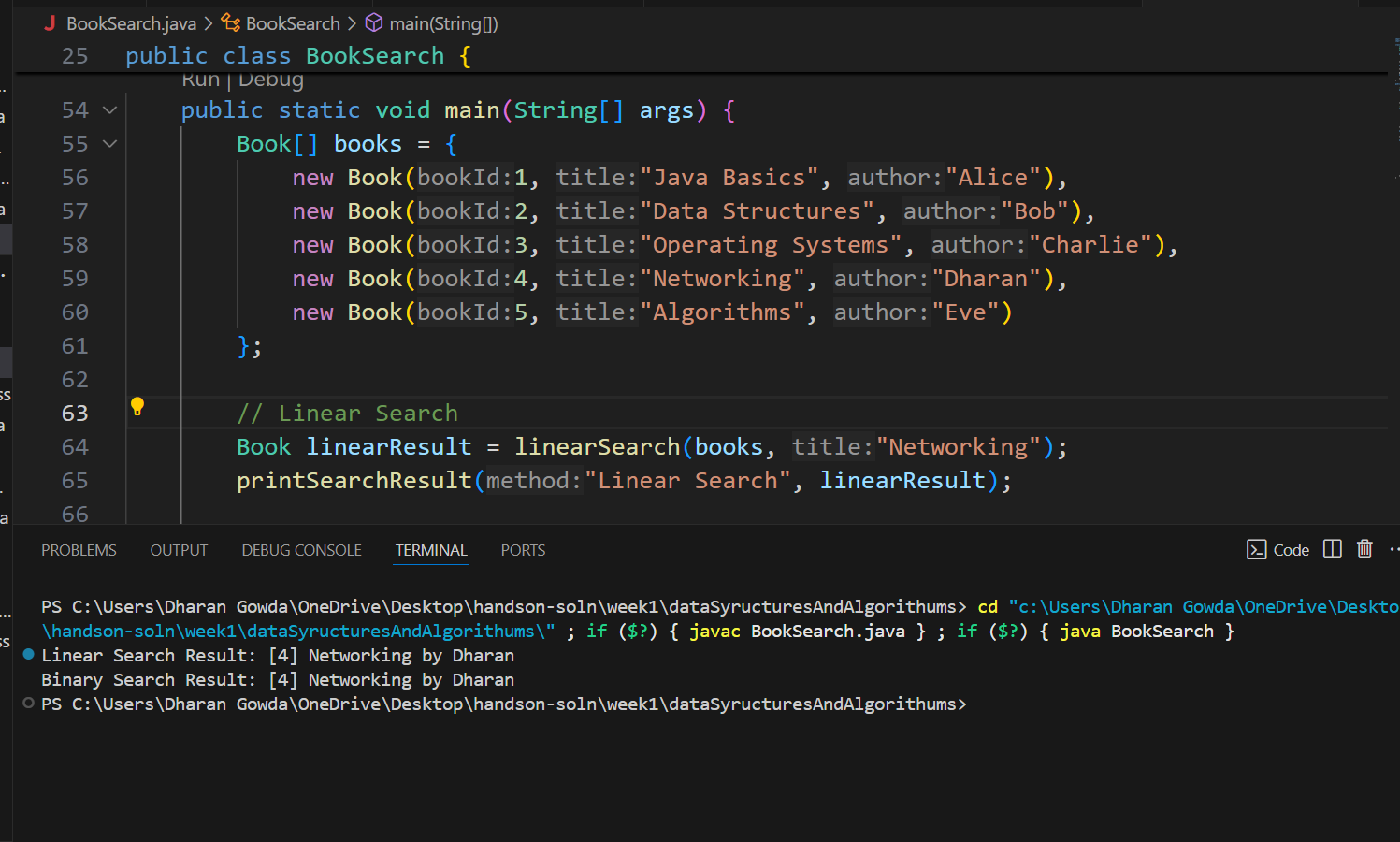
Book binaryResult = binarySearch(books, "Networking");

printSearchResult("Binary Search", binaryResult);

}

}

**Output:-**

****

**Analysis**

| Algorithm | Time Complexity | Best Case | Average | Worst Case |
| --- | --- | --- | --- | --- |
| Linear Search | O(n) | O(1) | O(n/2) | O(n) |
| Binary Search | O(log n) | O(1) | O(log n) | O(log n) |

**When to Use**:

* linear search: for unsorted or small datasets.
* binary search: for sorted and large datasets.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**1. Understand Recursive Algorithms**

**What is Recursion?**

* A function calls itself to solve a smaller subproblem.
* Requires a base case to terminate.
* Useful in problems with repetitive, nested structure, like factorial, Fibonacci, or financial projections.

**CODE:-**

public class FutureValueCalculator {

// Recursive function to calculate future value

public static double calculateFutureValue(double principal, double rate, int years) {

if (years == 0) return principal;

return calculateFutureValue(principal \* (1 + rate), rate, years - 1);

}

// Optimized Iterative Version

public static double calculateFutureValueIterative(double principal, double rate, int years) {

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

principal \*= (1 + rate);

}

return principal;

}

public static void main(String[] args) {

double principal = 10000; // ₹10,000

double rate = 0.08; // 8% annual growth

int years = 5; // Over 5 years

double futureValueRecursive = calculateFutureValue(principal, rate, years);

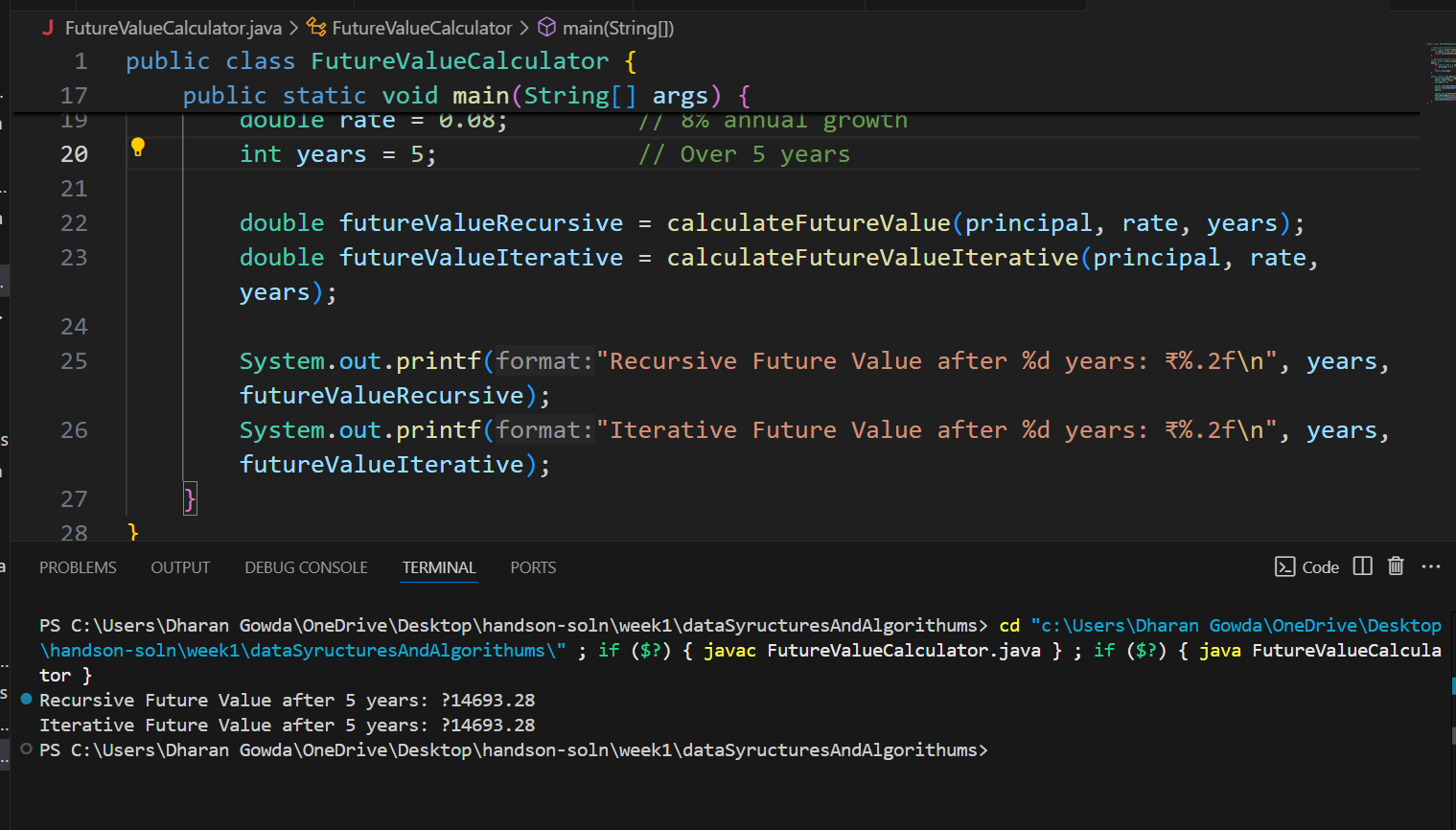
double futureValueIterative = calculateFutureValueIterative(principal, rate, years);

System.out.printf("Recursive Future Value after %d years: ₹%.2f\n", years, futureValueRecursive);

System.out.printf("Iterative Future Value after %d years: ₹%.2f\n", years, futureValueIterative);

}}

**OUTPUT:-**

****

**4. Analysis & Optimization**

**Time Complexity:**

* O(n) — one call per year until years == 0.

**Drawbacks:**

* Recursive calls consume stack memory.
* For large years, can lead to stack overflow.

**Optimization – Tail Recursion or Iteration:**

* Iterative version is more efficient in Java (no tail call optimization).

| **Approach** | **Time Complexity** | **Space Complexity** | **Notes** |
| --- | --- | --- | --- |
| Recursive | O(n) | O(n) | Simple but risk of stack overflow |
| Iterative | O(n) | O(1) | Safer and more efficient |