**Exercise 1: Inventory Management System**

**Scenario:**

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

**Explain why data structures and algorithms are essential in handling large inventories.**

* Efficient Retrieval: You may have thousands of products. A poor choice of structure can make search/update slow.
* Faster Operations: Efficient algorithms minimize operation time (e.g., updating stock in O(1) instead of O(n)).
* Optimized Memory Usage: Choosing the right data structure saves memory and avoids duplication or unnecessary complexity.

**Discuss the types of data structures suitable for this problem.**

**1. ArrayList :** Best for small inventories or when you need to access items sequentially.

* **Pros:** Easy to iterate using loops (for or for-each) and Maintains insertionorder
* **Cons:** Searching or updating a product by ID takes O(n) time (linear search) and Not efficient for large data sets

**2. HashMap<Integer, Product> :** Ideal for large inventories where fast lookup, update, or delete is needed by product ID

* **Pros:** O(1) time for add, search, update, and delete and Uses keys (productId) to quickly access values (Product objects).
* **Cons:** Slightly higher memory usage due to internal hashing mechanism.

**Project name : InventorySystem**

**Product.java**

public 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;

}

@Override

public String toString() {

return "ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: $" + price;

}

}

**InventoryManager.java**

import java.util.HashMap;

public class InventoryManager {

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

public void addProduct(Product product) {

inventory.put(product.productId, product);

System.out.println("Product added: ID " + product.productId + " - " + product.productName);

}

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

Product product = inventory.get(productId);

if (product != null) {

product.quantity = quantity;

product.price = price;

System.out.println("Product updated: ID " + productId +

" now has Quantity " + quantity + " and Price $" + price);

} else {

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

}

}

public void deleteProduct(int productId) {

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

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

} else {

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

}

}

public void displayInventory() {

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

System.out.println(p);

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

Product p1 = new Product(101, "Laptop", 10, 75000);

Product p2 = new Product(102, "Mouse", 50, 500);

Product p3 = new Product(103, "Keyboard", 20, 1500);

manager.addProduct(p1);

manager.addProduct(p2);

manager.addProduct(p3);

System.out.println("\n Current Inventory:");

manager.displayInventory();

System.out.println("\nUpdating product with ID 102...");

manager.updateProduct(102, 60, 450);

System.out.println("\nDeleting product with ID 101...");

manager.deleteProduct(101);

System.out.println("\n Inventory after update and delete:");

manager.displayInventory();

System.out.println("\nAttempting to update non-existent product ID 999...");

manager.updateProduct(999, 5, 100);

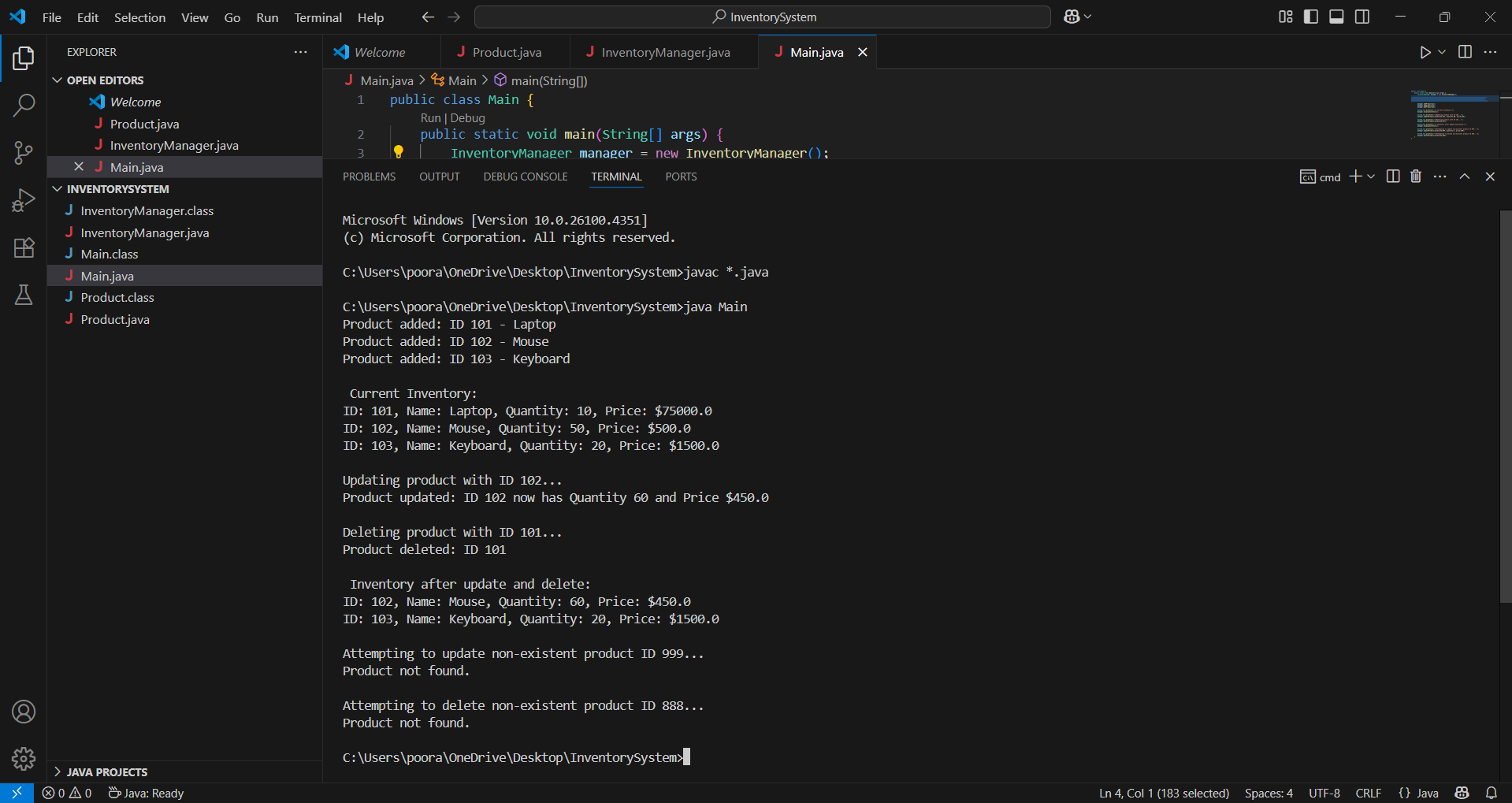
System.out.println("\nAttempting to delete non-existent product ID 888...");

manager.deleteProduct(888);

}

}

**Output screenshot:**

****

**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 or space requirement in terms of input size n. It helps to:  
- Analyze performance regardless of hardware.  
- Compare efficiency of different algorithms.  
- Focus on scalability.

**🔹 Best, Average, and Worst-Case Scenarios:**

• Best Case (Ω): The item is found immediately.  
• Average Case (Θ): The item is found after scanning half the collection.  
• Worst Case (O): The item is not found or is last in the collection.

**2. Setup**

Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

public class Product {

String productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String toString() {

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

}

}

**3. Implementation:**

🔹 **Linear Search Implementation:**

**// ProductSerach.java**

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

for (Product p : products) {

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

return p;

}

}

return null;

}

🔹 **Binary Search Implementation (on sorted array):**

**// ProductSearch.java**

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

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

int left = 0, right = products.length - 1;

while (left <= right) {

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

int compare = products[mid].productName.compareToIgnoreCase(targetName);

if (compare == 0) {

return products[mid];

} else if (compare < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

**4.Analysis**

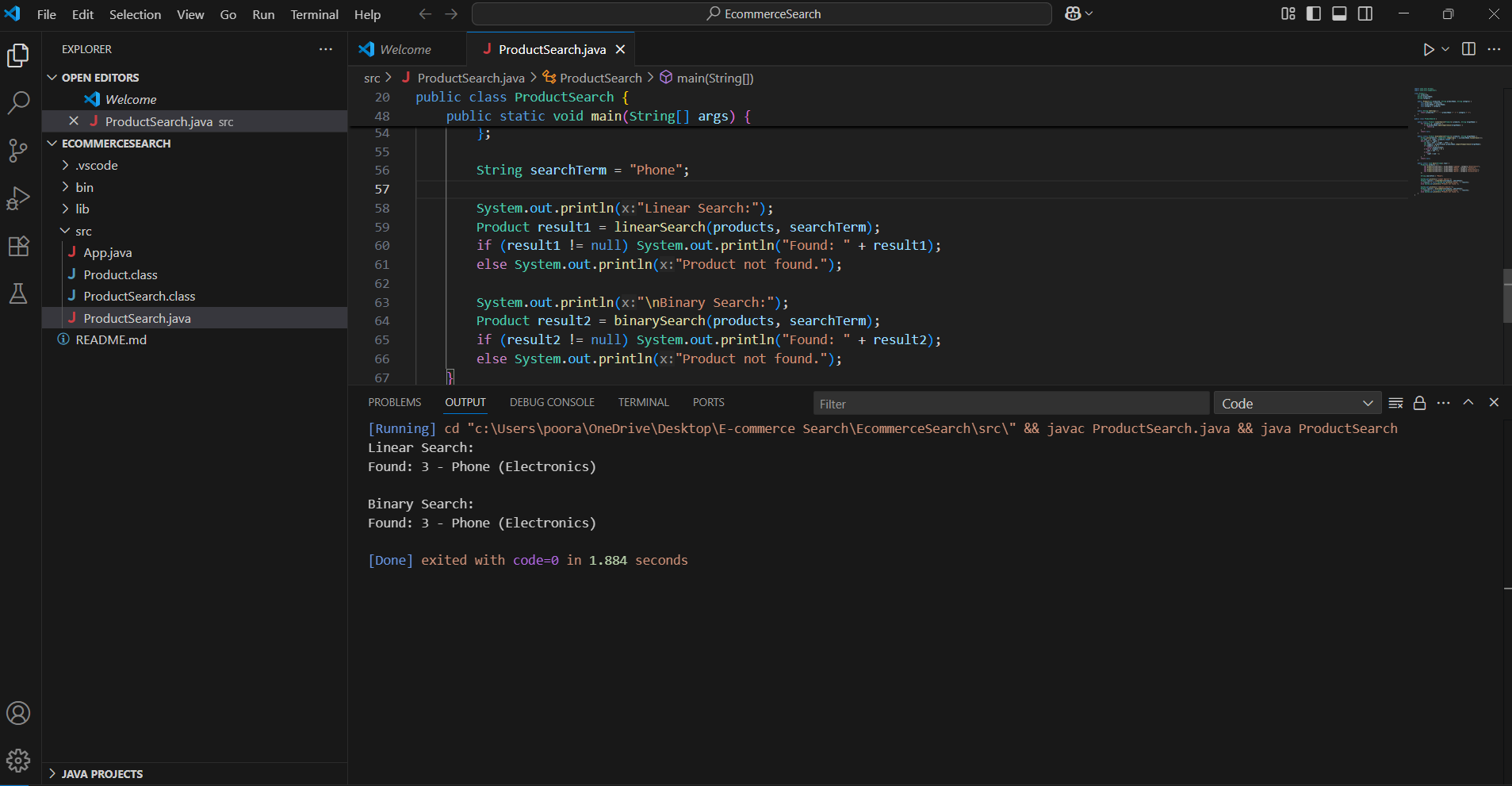
🔹 **Time Complexity Comparison:**

• Linear Search: Best Case – O(1), Average/Worst Case – O(n)  
• Binary Search: Best Case – O(1), Average/Worst Case – O(log n)

🔹 **Recommendation:**

We an use Binary Search for large, sorted datasets to achieve better performance and Linear Search for small or unsorted collections.

**5. Output screenshot**



**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).**

**1)Bubble Sort**

* It repeatedly compares adjacent elements and swaps them if they're in the wrong order.
* Simple but slow.
* Time Complexity: Best: O(n) (already sorted)

Average/Worst: O(n²)

**2)Insertion Sort**

* It is useful to build the final sorted array one item at a time.
* Good for small or nearly sorted data.
* Time Complexity: Best: O(n)

Average/Worst: O(n²)

**3)Quick Sort**

* Divide-and-conquer: Chooses a pivot, partitions the array, recursively sorts the subarrays.
* Fast and widely used.
* Time Complexity: Best/Average: O(n log n)

Worst: O(n²) (rare, occurs with bad pivot choice)

**4)Merge Sort**

* Also divide-and-conquer but splits array completely, then merges.
* Stable and predictable time.
* Time Complexity: Always O(n log n)

**Project name : CustomerOrderSorting**

**Order.java**

public class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

@Override

public String toString() {

return "Order ID: " + orderId + ", Customer: " + customerName + ", Total: $" + totalPrice;

}

}

**SortUtil.java**

public class SortUtil {

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].totalPrice < orders[j + 1].totalPrice) {

Order temp = orders[j];

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

orders[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break;

}

}

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;

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 300.0),

new Order(2, "Bob", 150.0),

new Order(3, "Charlie", 700.0),

new Order(4, "Diana", 450.0)

};

Order[] bubbleOrders = orders.clone();

Order[] quickOrders = orders.clone();

// Bubble Sort

SortUtil.bubbleSort(bubbleOrders);

System.out.println(" Orders Sorted by Bubble Sort:");

for (Order o : bubbleOrders) System.out.println(o);

// Quick Sort

SortUtil.quickSort(quickOrders, 0, quickOrders.length - 1);

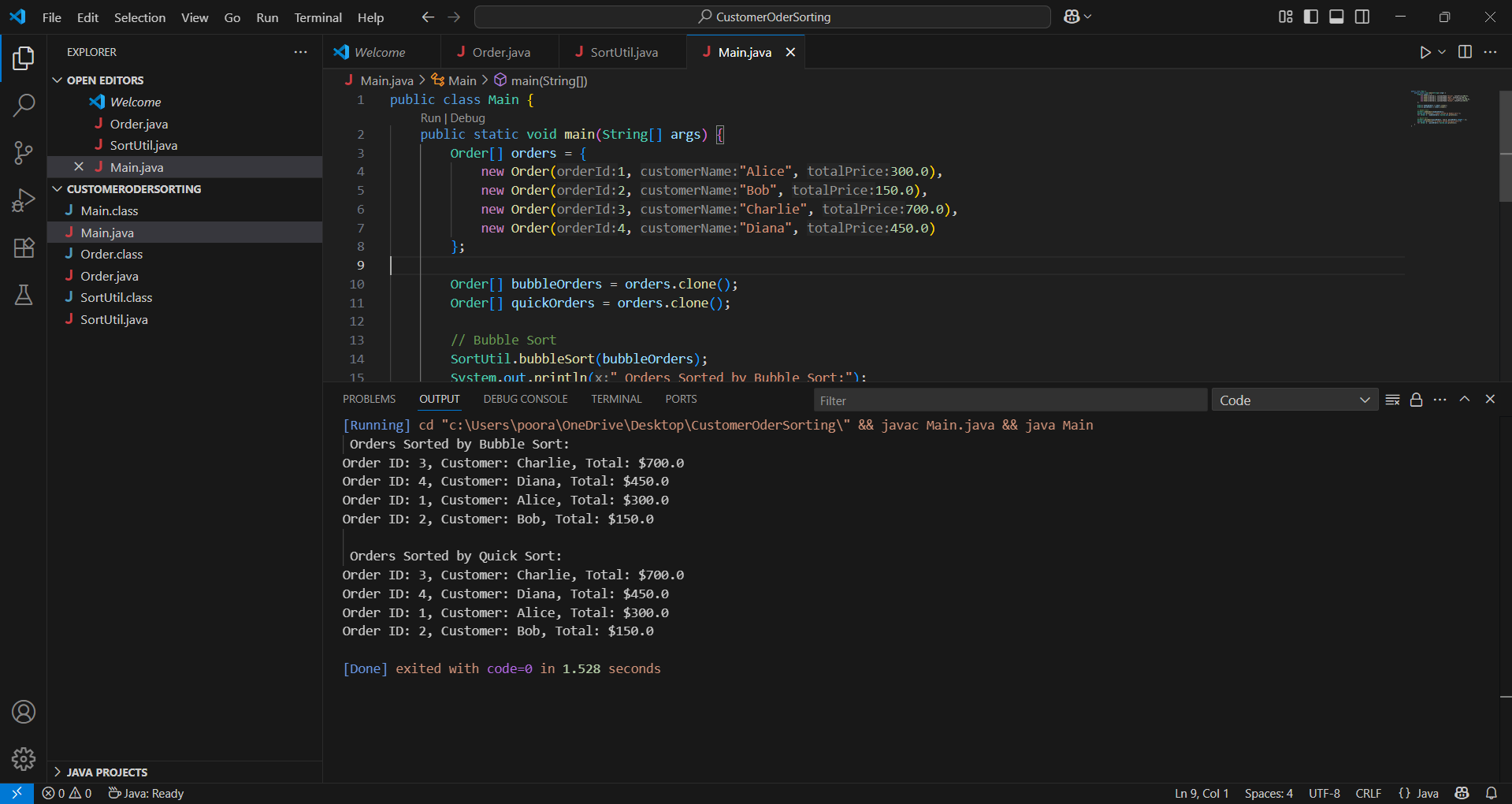
System.out.println("\n Orders Sorted by Quick Sort:");

for (Order o : quickOrders) System.out.println(o);

}

}

**Output screenshot:**

****

**Exercise 4: Employee Management System**

**Scenario:**

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

**Explain how arrays are represented in memory and their advantages.**

* Contiguous memory allocation: All elements are stored in consecutive memory locations.
* Index-based access: Every element can be accessed in O(1) time using its index.

### **Advantages of Arrays**:

* **Fast access** using index
* **Simple structure**, easy to implement
* **Memory-efficient** for fixed-size data

**Project name : EmployeeManagementSystem**

**Employee.java**

public 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;

}

@Override

public String toString() {

return "ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: $" + salary;

}

}

**EmployeeManager.java**

public class EmployeeManager {

private Employee[] employees;

private int size;

public EmployeeManager(int capacity) {

employees = new Employee[capacity];

size = 0;

}

public void addEmployee(Employee emp) {

if (size < employees.length) {

employees[size++] = emp;

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

} else {

System.out.println(" Employee list is full.");

}

}

public Employee searchEmployee(int empId) {

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

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

return employees[i];

}

}

return null;

}

public void displayEmployees() {

System.out.println("\n Employee Records:");

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

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

}

}

public void deleteEmployee(int empId) {

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

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

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

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

}

employees[--size] = null;

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

return;

}

}

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager(5);

// Add 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));

manager.addEmployee(new Employee(104, "Diana", "Tester", 40000));

manager.addEmployee(new Employee(105, "Eve", "HR", 48000));

// Display all employees

manager.displayEmployees();

// Search for an employee

System.out.println("\n Searching for employee with ID 103:");

Employee found = manager.searchEmployee(103);

System.out.println(found != null ? found : "Not found.");

// Delete an employee

System.out.println("\n Deleting employee with ID 102:");

manager.deleteEmployee(102);

// Try to delete a non-existing employee

System.out.println("\n Trying to delete employee with ID 200:");

manager.deleteEmployee(200);

// Final state of employee list

manager.displayEmployees();

// Try to add more than capacity

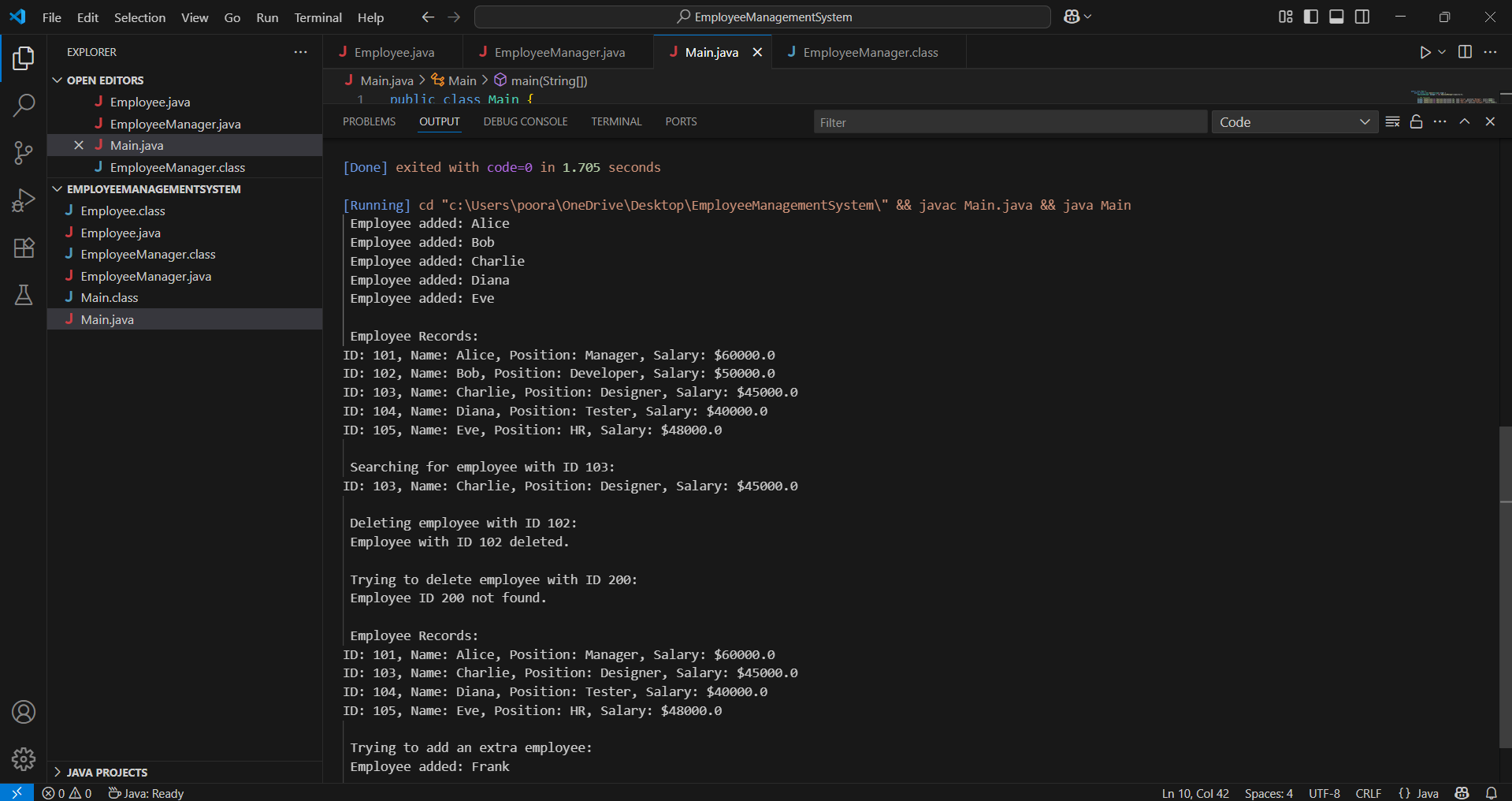
System.out.println("\n Trying to add an extra employee:");

manager.addEmployee(new Employee(106, "Frank", "Intern", 25000));

}

}

**Output screenshot:**

****

**Exercise 5: Task Management System**

**Scenario:**

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

**Explain the different types of linked lists (Singly Linked List, Doubly Linked List).**

**1)Singly Linked List**

* In a singly linked list, each node contains data and a reference (or pointer) to the next node in the sequence.
* It allows forward-only traversal.
* It is lightweight and simple to implement.

**2)Doubly Linked List**

* In a doubly linked list, each node has two references: one to the next node and one to the previous node.
* This allows for bi-directional traversal, meaning you can go forward and backward through the list.
* It uses more memory due to the extra pointer.

**Project name : Task Management System**

**Task.java**

public 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 "Task ID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

**TaskManager.java**

class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.next = null;

}

}

public class TaskManager {

private Node head;

public void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

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

}

public Task searchTask(int taskId) {

Node temp = head;

while (temp != null) {

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

return temp.task;

}

temp = temp.next;

}

return null;

}

public void displayTasks() {

Node temp = head;

if (temp == null) {

System.out.println(" Task list is empty.");

return;

}

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

while (temp != null) {

System.out.println(temp.task);

temp = temp.next;

}

}

public void deleteTask(int taskId) {

if (head == null) {

System.out.println(" Task list is empty.");

return;

}

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

head = head.next;

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

return;

}

Node prev = head;

Node curr = head.next;

while (curr != null) {

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

prev.next = curr.next;

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

return;

}

prev = curr;

curr = curr.next;

}

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

TaskManager manager = new TaskManager();

// Add tasks

manager.addTask(new Task(101, "Design UI", "Pending"));

manager.addTask(new Task(102, "Develop Backend", "In Progress"));

manager.addTask(new Task(103, "Testing", "Pending"));

manager.addTask(new Task(104, "Deployment", "Not Started"));

// Display all tasks

manager.displayTasks();

// Search for a task

System.out.println("\n Searching for Task ID 102:");

Task t = manager.searchTask(102);

System.out.println(t != null ? t : "Task not found.");

// Delete a task

System.out.println("\n Deleting Task ID 103:");

manager.deleteTask(103);

// Try to delete a non-existent task

System.out.println("\n Deleting Task ID 999:");

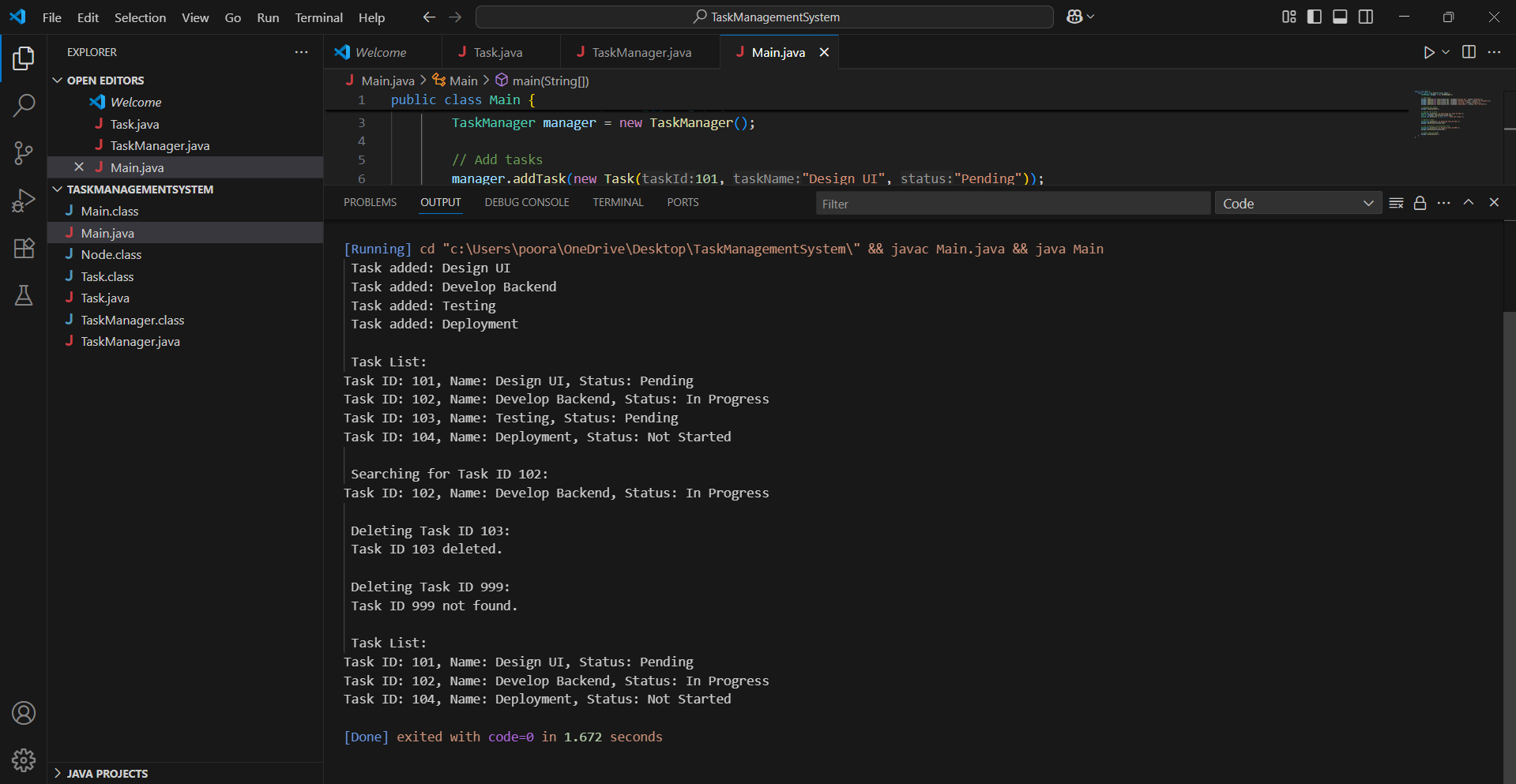
manager.deleteTask(999);

manager.displayTasks();

}

}

**Output screenshot:**

****

**Exercise 6: Library Management System**

**Scenario:**

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

**Explain linear search and binary search algorithms.**

**1)Linear Search**

* Linear search checks each element one by one until the target is found or the list ends.
* It does not require sorted data.

**Time Complexity:**

* Best-case: O(1) → if the item is at the start
* Worst-case: O(n) → if the item is at the end or not present
* Average-case: O(n)

**2)Binary Search**

* Binary search only works on sorted arrays or lists.
* It divides the array in half and eliminates half the search space in each step.

**Time Complexity:**

* Best-case: O(1) → if the middle is the target
* Worst-case: O(log n)
* Average-case: O(log n)

**Project name : Library Management System**

**Book.java**

public 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;

}

@Override

public String toString() {

return "Book ID: " + bookId + ", Title: " + title + ", Author: " + author;

}

}

**LibraryManager.java**

import java.util.Arrays;

import java.util.Comparator;

public class LibraryManager {

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

for (Book book : books) {

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

return book;

}

}

return null;

}

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

int left = 0, right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0)

return books[mid];

else if (cmp < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

public static void sortBooksByTitle(Book[] books) {

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

}

public static void displayBooks(Book[] books) {

System.out.println("\n Book List:");

for (Book b : books) {

System.out.println(b);

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Book[] books = {

new Book(1, "Java Programming", "James Gosling"),

new Book(2, "C Programming", "Dennis Ritchie"),

new Book(3, "Python Crash Course", "Eric Matthes"),

new Book(4, "Data Structures", "Mark Allen Weiss"),

new Book(5, "Algorithms", "Robert Sedgewick")

};

// Display unsorted books

LibraryManager.displayBooks(books);

// Linear search (works on unsorted data)

System.out.println("\n Linear Search: 'Python Crash Course'");

Book result1 = LibraryManager.linearSearch(books, "Python Crash Course");

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

// Sort books for binary search

LibraryManager.sortBooksByTitle(books);

System.out.println("\n Sorted Book List (for Binary Search):");

LibraryManager.displayBooks(books);

// Binary search (on sorted data)

System.out.println("\n Binary Search: 'Data Structures'");

Book result2 = LibraryManager.binarySearch(books, "Data Structures");

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

// Try searching a book that doesn't exist

System.out.println("\n Binary Search: 'Machine Learning'");

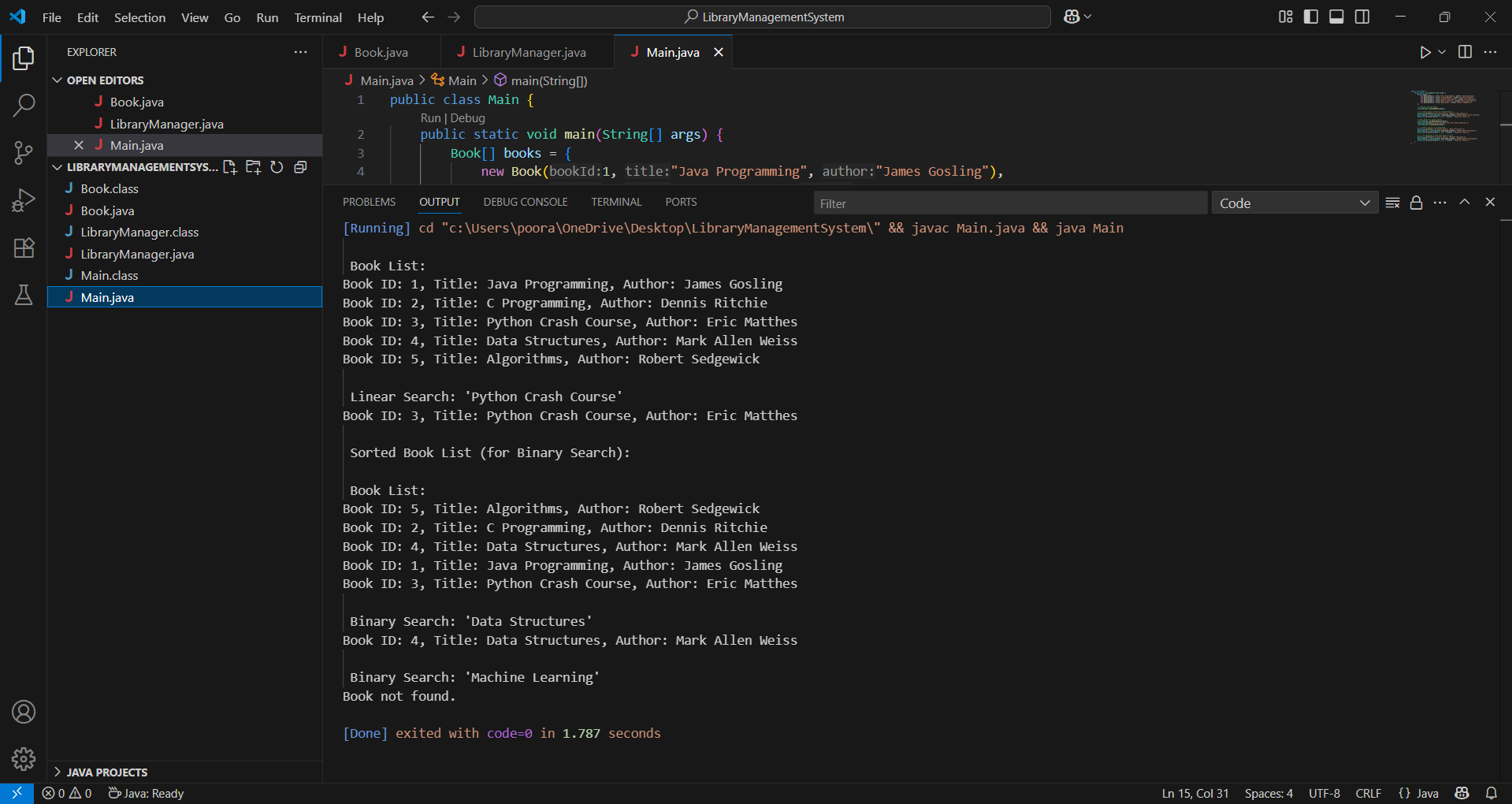
Book result3 = LibraryManager.binarySearch(books, "Machine Learning");

System.out.println(result3 != null ? result3 : "Book not found.");

}

}

**Output screenshot:**

****

**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**

**🔹 The concept of recursion and how it can simplify certain problems:**  
 • Recursion is a programming technique in which a function repeatedly calls itself to solve smaller pieces of a larger problem.  
 • In forecasting, this is helpful because each future value depends on the previous year's value — a naturally recursive relationship.  
 • Instead of writing nested loops, recursion offers a cleaner way to model step-by-step annual changes. It simplifies the problem logic and makes the code easier to understand when dealing with repetitive calculations.

1. **Setup**

🔹 **A method to calculate the future value using a recursive approach:**

**// Forecast.java**

public static double calculateFutureValue(double currentValue, double annualGrowthRate, int years) {

if (years == 0) {

return currentValue;

}

return calculateFutureValue(currentValue, annualGrowthRate, years - 1) \* (1 + annualGrowthRate);

}

}

1. **Implementation:**

🔹 **Implementation of a recursive algorithm to predict future values based on past growth rates:**

**// ForecastTest.java**

public class ForecastTest {

public static void main(String[] args) {

double startingAmount = 1500.0;

double growthRate = 0.07;

int totalYears = 6;

double result = Forecast.calculateFutureValue(startingAmount, growthRate, totalYears);

System.out.printf("Estimated value after %d years: ₹%.2f\n", totalYears, result);

}

}

1. **Analysis:**

**🔹 Time complexity of the recursive algorithm:**

* The time complexity is **O(n)**, where n is the number of years to forecast.
* Each recursive call handles a single year, so the function is called n times in total.

**🔹 How to optimize the recursive solution to avoid excessive computation:**

1. **Use Iteration Instead:**  
   Recursion in this case is straightforward, but using a loop (for or while) is more efficient and avoids potential stack overflow errors.
2. **Avoid Redundant Calls:**  
   Although this example does not contain repeated subproblems, in more complex models, **memoization** (caching results) can be used to save time.
3. **Output screenshot:**

