**ALGORITHMS\_ DATA STRUCTURES**

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

Big O notation describes the upper bound of an algorithm's runtime as the input size (n) grows. It helps us understand how algorithms scale and which are more efficient for large datasets.

|  |  |  |
| --- | --- | --- |
| **Big O** | **Name** | **Example** |
| O(1) | Constant | Accessing an array index |
| O(log n) | Logarithmic | Binary search |
| O(n) | Linear | Linear search |
| O(n log n) | Log-linear | Merge sort |
| O(n²) | Quadratic | Bubble sort |

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

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;

}

}

import java.util.Arrays;

import java.util.Comparator;

public class ECommerceSearch {

// Linear Search

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

for (Product p : products) {

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

return p;

}

}

return null;

}

// Binary Search

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

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

while (left <= right) {

int mid = (left + right) / 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;

}

**Search Algorithms**

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Fashion"),

new Product(3, "Book", "Education"),

new Product(4, "Phone", "Electronics"),

new Product(5, "T-shirt", "Fashion")

};

// Linear Search

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

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

System.out.println(result1 != null ? "Found: " + result1.productName : "Not found");

// Sort for Binary Search

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

// Binary Search

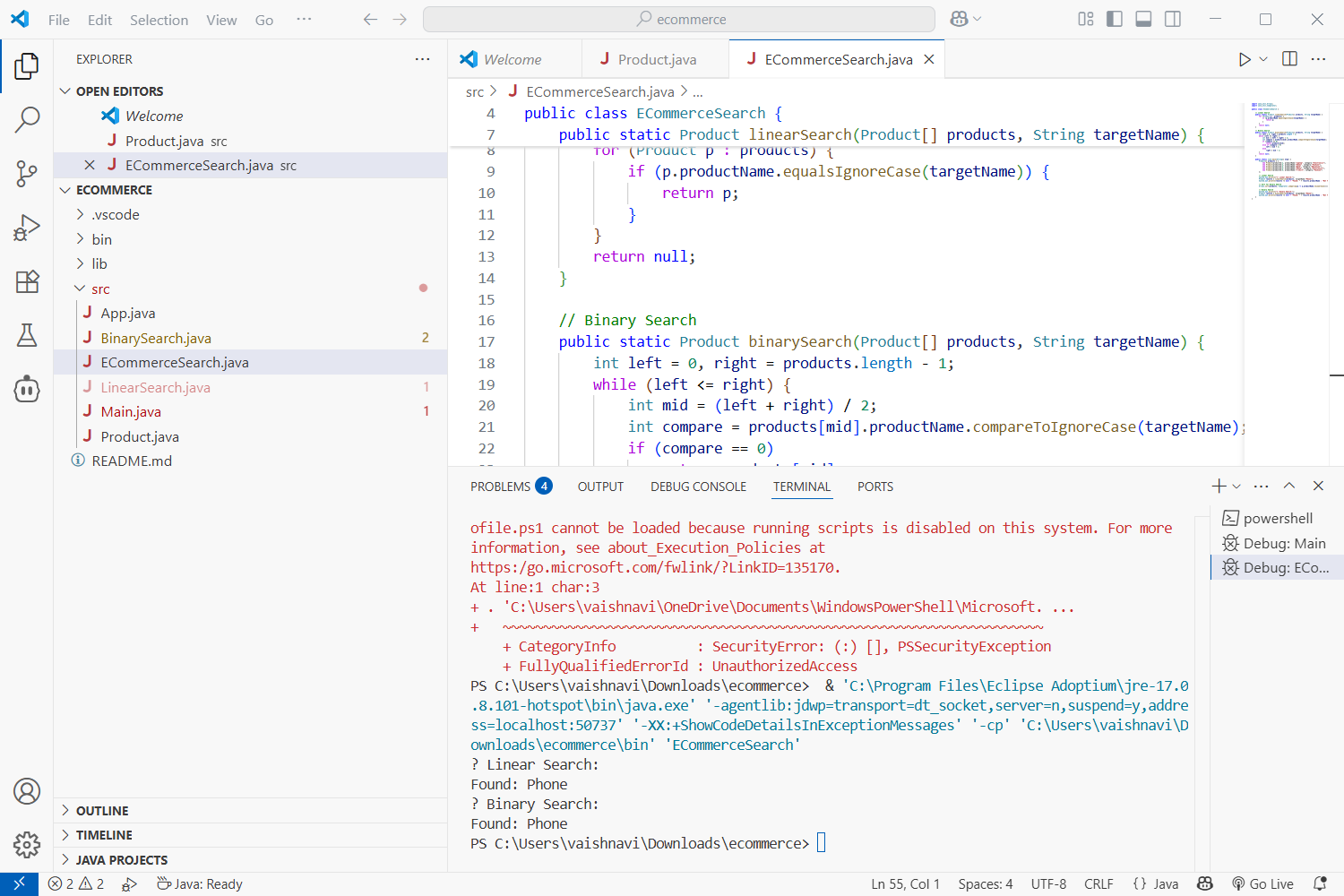
System.out.println("Binary Search:");

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

System.out.println(result2 != null ? "Found: " + result2.productName : "Not found");

}

}



**Analysis**

Linear Search -Works on unsorted lists

Binary Search -Efficient but requires sorted array

Use **Binary Search** for faster results on **sorted product lists**, especially with large datasets.

**Exercise 7: Financial Forecasting**

Recursion is a programming technique where a function calls itself to solve a problem by breaking it into smaller sub-problems**.**

➤ Key Characteristics:

* Base case: Stops the recursion.
* Recursive case: Calls itself with smaller input.

➤ Example Use Cases:

* Factorial calculation
* Fibonacci series
* Tree traversal
* Financial projections (like compound growth)

We want to predict the **future value** of an investment using a **compound growth rate**:

➤ Formula:

FV=PV×(1+r)nFV = PV \times (1 + r)^nFV=PV×(1+r)n

Where:

* FV = Future Value
* PV = Present Value
* r = growth rate (as a decimal)
* n = number of years

We'll use a **recursive approach** to compute this.

public class FinancialForecasting {

// Recursive method to calculate future value

public static double futureValueRecursive(double presentValue, double rate, int years) {

if (years == 0) {

return presentValue; // base case

}

return (1 + rate) \* futureValueRecursive(presentValue, rate, years - 1);

}

public static void main(String[] args) {

double presentValue = 10000; // Rs. 10,000

double rate = 0.08; // 8% annual growth

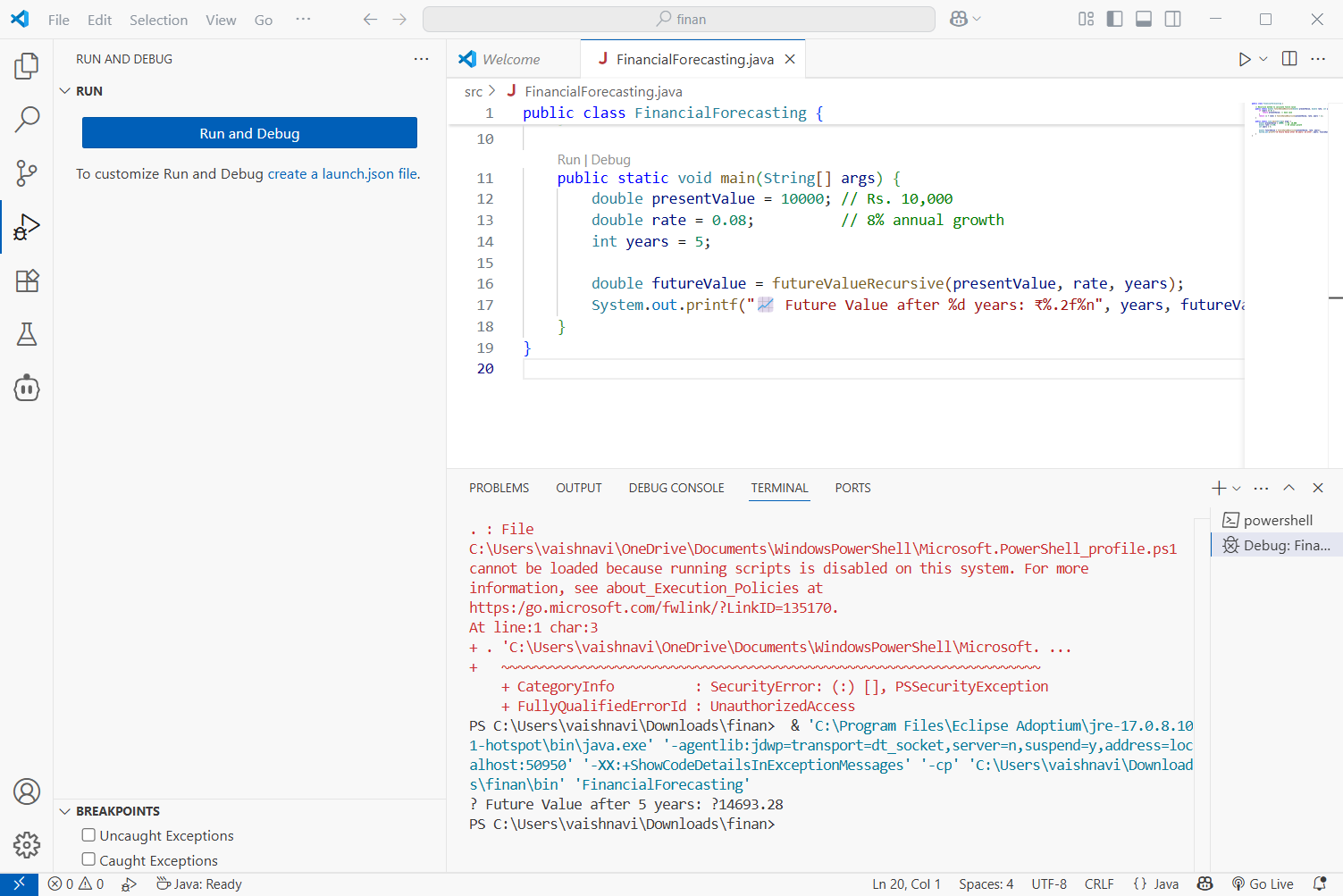
int years = 5;

double futureValue = futureValueRecursive(presentValue, rate, years);

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

}

}



**Analysis of the Recursive Solution**

➤ Time Complexity:

* Each recursive call reduces years by 1.
* So for n years, the function makes n calls.
* Time Complexity: O(n)

➤ Space Complexity:

* Due to the recursive call stack: O(n)

➤ Pros:

* Simple and clean implementation.
* Easy to understand for small input sizes.

➤ Cons:

* Not efficient for very large n.
* May cause StackOverflowError in Java for deep recursion (Java has a limited call stack).

**Optimizing Recursive Solution**

1. **Use Iteration Instead** (preferred for large n):

public static double futureValueIterative(double presentValue, double rate, int years) {

double futureValue = presentValue;

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

futureValue \*= (1 + rate);

}

return futureValue;

}

* **Time Complexity: O(n)**
* **Space Complexity: O(1)** – no recursion stack.

1. **Use Memoization** (not very helpful here since results are not reused, but good in other recursive problems like Fibonacci).

**Summary**

We implemented a recursive approach to predict the future value of an investment using the compound growth formula. The recursive method multiplies the present value by (1 + rate) repeatedly for each year until the base case (year 0) is reached. This approach is **simple and easy to understand**, especially for small numbers of years.

However, recursion has limitations: its **time complexity is O(n)**, and it also uses **O(n) space** because each function call is stored in memory. For larger input sizes (like 10,000 years), it can lead to stack overflow errors due to too many recursive calls.

To optimize this, we can use an **iterative solution**, which computes the future value using a loop. It still takes O(n) time but uses **constant memory (O(1))**, making it more efficient and safe for large data.

In some cases, using a **closed-form formula** (like FV = PV × (1 + r)^n) with Math.pow() is the best choice as it runs in **constant time and space**.

**Exercise 1: Inventory Management System**

* A warehouse inventory can have **thousands of items**.
* Efficient data handling ensures **fast searching, updating, and deleting**.
* Choosing the right data structure improves performance and memory usage.

Suitable Data Structures:

| **Structure** | **Use Case** |
| --- | --- |
| ArrayList- | Simple lists, slower lookup (O(n)) |
| HashMap- | Fast lookup, insert, update, and delete by ID (O(1)) |
|  |  |

import java.util.HashMap;

import java.util.Scanner;

class Product {

int productId;

String productName;

int quantity;

double price;

Product(int id, String name, int qty, double price) {

this.productId = id;

this.productName = name;

this.quantity = qty;

this.price = price;

}

@Override

public String toString() {

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

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

}

}

public class InventorySystem {

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

public static void addProduct(Product p) {

inventory.put(p.productId, p);

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

}

public static void updateProduct(int id, int qty, double price) {

if (inventory.containsKey(id)) {

Product p = inventory.get(id);

p.quantity = qty;

p.price = price;

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

} else {

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

}

}

public static void deleteProduct(int id) {

if (inventory.remove(id) != null)

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

else

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

}

public static void main(String[] args) {

addProduct(new Product(101, "Mouse", 50, 499.99));

addProduct(new Product(102, "Keyboard", 30, 899.50));

updateProduct(101, 45, 479.99);

deleteProduct(102);

// Display inventory

System.out.println("\ Inventory List:");

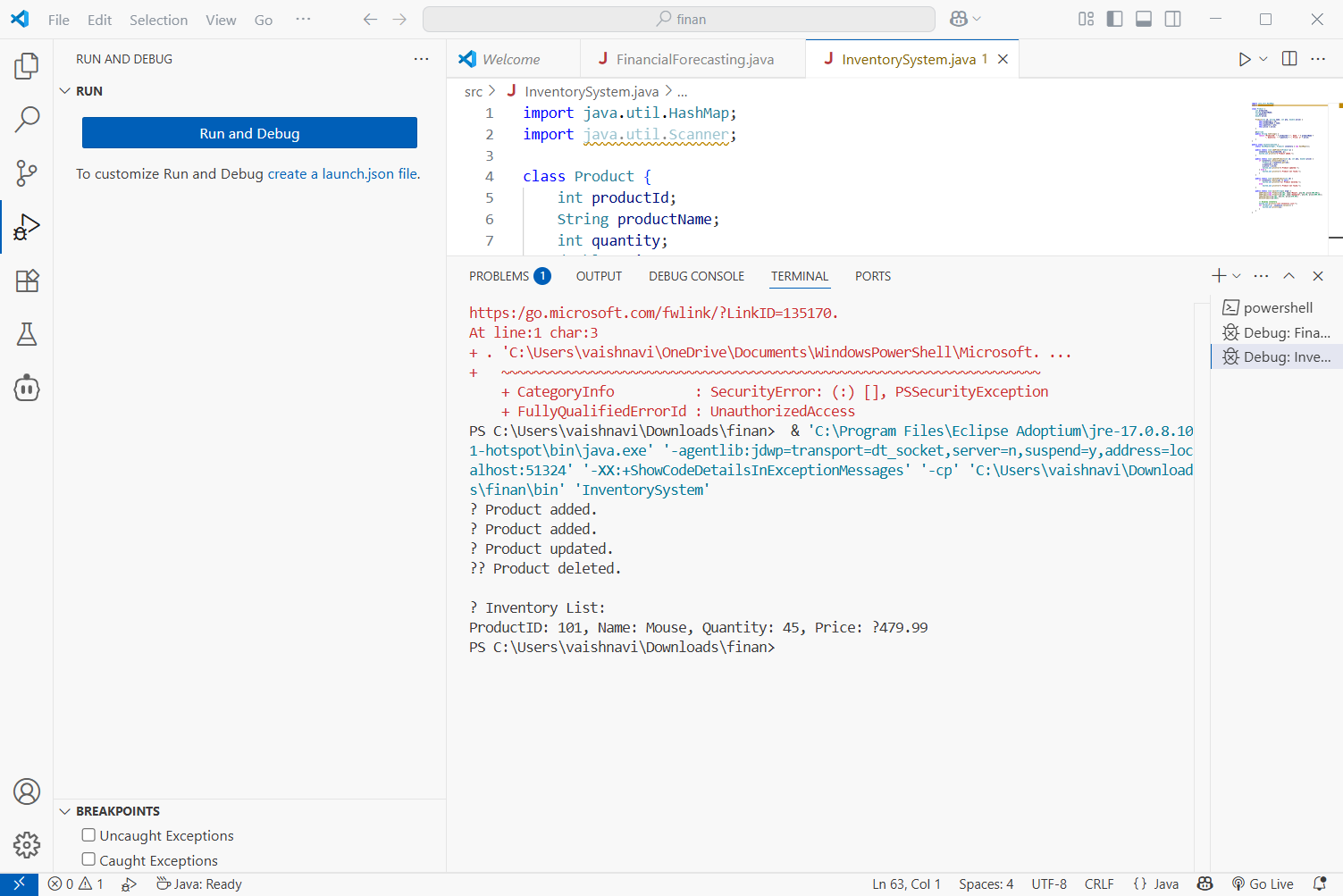
for (Product p : inventory.values()) {

System.out.println(p);

}

}

}



**Analysis**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O(1) | HashMap inserts are constant time |
| Update | O(1) | Direct access via key (productId) |
| Delete | O(1) | Removes based on key directly |

**Exercise 3: Sorting Customer Orders**

➤ Bubble Sort

* Repeatedly swaps adjacent elements if they’re in the wrong order.
* Best case: O(n) (when already sorted)
* Average/Worst case: O(n²)
* Simple but slow

➤ Insertion Sort

* Builds sorted array one item at a time.
* Best case: O(n), Worst: O(n²)
* Works well for small or nearly sorted arrays

➤ Merge Sort

* Divides the array and merges sorted halves.
* Always O(n log n) time.
* Uses extra memory

➤ Quick Sort

* Picks a pivot, partitions the array, and recursively sorts.
* Best/Average: O(n log n), Worst: O(n²)
* Efficient and widely used

**Order Class**

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 "OrderID: " + orderId + ", Customer: " + customerName + ", Total: ₹" + totalPrice;

}

}

**OrderSorted**

public class OrderSorter {

// Bubble Sort

public static void bubbleSort(Order[] orders) {

int n = orders.length;

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

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;

}

}

}

}

// 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].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;

}

// Display Orders

public static void display(Order[] orders) {

for (Order o : orders) {

System.out.println(o);

}

}

public static void main(String[] args) {

Order[] orders = {

new Order(201, "Aarav", 4500.75),

new Order(202, "Bhavya", 1200.50),

new Order(203, "Chirag", 7800.00),

new Order(204, "Devika", 3000.00)

};

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

display(orders);

System.out.println("\nBubble Sorted Orders (Low to High):");

bubbleSort(orders);

display(orders);

// Resetting array for Quick Sort

orders = new Order[]{

new Order(201, "Aarav", 4500.75),

new Order(202, "Bhavya", 1200.50),

new Order(203, "Chirag", 7800.00),

new Order(204, "Devika", 3000.00)

};

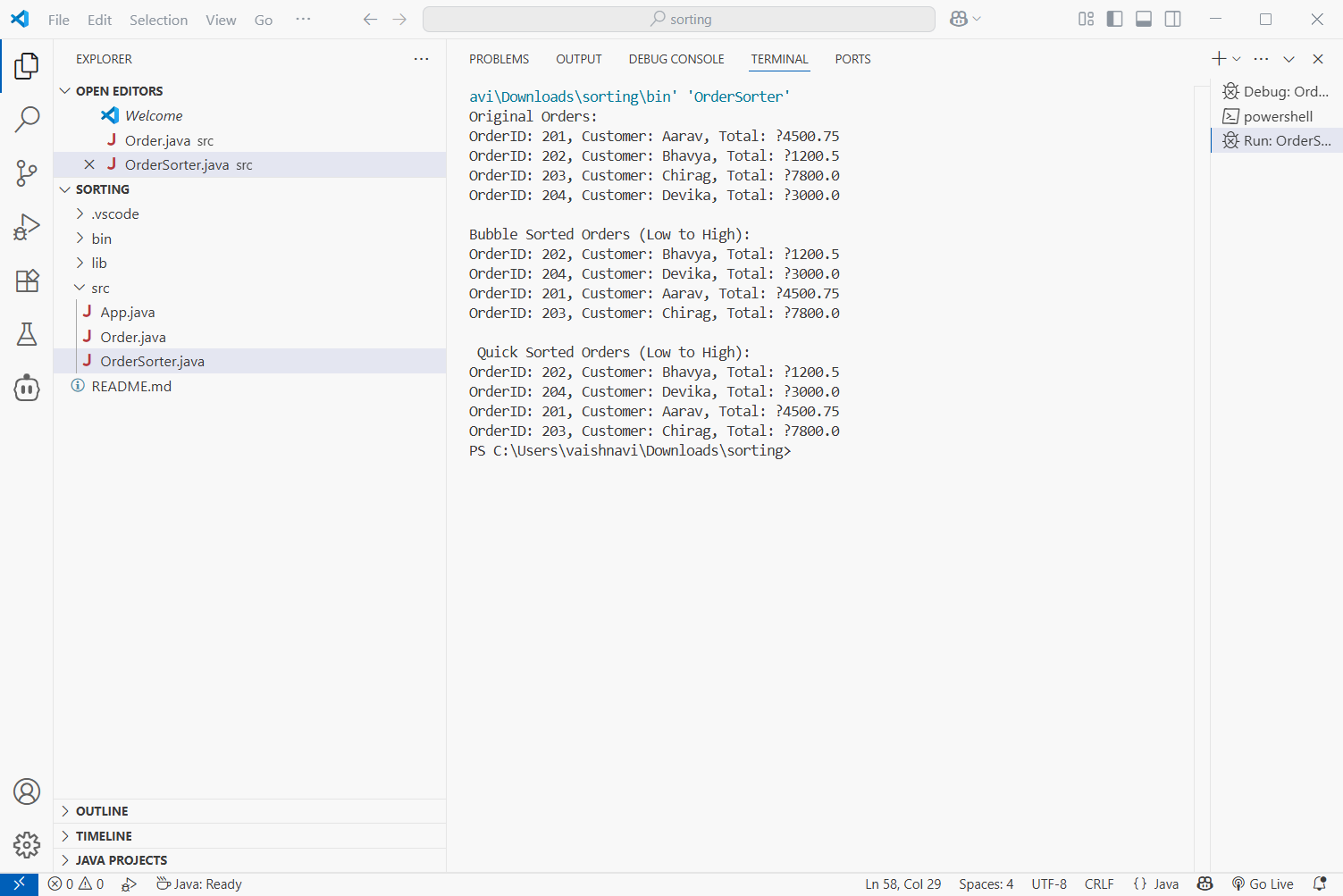
System.out.println("\nQuick Sorted Orders (Low to High):");

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

display(orders);

}

}



**Comparison**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) | Easy, slow on large data |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) | Fast, widely used, in-place |

**Quick Sort**

* Much faster on average for large datasets.
* Bubble Sort is inefficient and mostly for learning.

**Exercise 4: Employee Management System**

➤ Arrays are Representation in Memory:

* Arrays are stored in contiguous memory locations.
* Each element is accessed using an index (starts at 0).
* In Java, arrays are objects with a fixed size, defined at creation.

➤ Advantages:

* Fast access using index → Time Complexity: O(1)
* Efficient memory use (if size is known)
* Simple to implement

**Employee Class**

class Employee {

int employeeId;

String name;

String position;

double salary;

Employee(int id, String name, String position, double salary) {

this.employeeId = id;

this.name = name;

this.position = position;

this.salary = salary;

}

@Override

public String toString() {

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

}

}

**EmployeeManagement Class**

public class EmployeeManagement {

static final int MAX\_EMPLOYEES = 100;

static Employee[] employees = new Employee[MAX\_EMPLOYEES];

static int count = 0;

// Add an employee

public static void addEmployee(Employee e) {

if (count < MAX\_EMPLOYEES) {

employees[count++] = e;

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

} else {

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

}

}

// Search employee by ID

public static void searchEmployee(int id) {

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

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

System.out.println(" Found: " + employees[i]);

return;

}

}

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

}

// Traverse all employees

public static void displayAll() {

System.out.println(" All Employees:");

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

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

}

}

// Delete employee by ID

public static void deleteEmployee(int id) {

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

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

// Shift elements left

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

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

}

employees[--count] = null;

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

return;

}

}

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

}

public static void main(String[] args) {

addEmployee(new Employee (1, "Aarav", "Manager", 75000));

addEmployee(new Employee(2, "Bhavya", "Developer", 55000));

addEmployee (new Employee (3, "Chirag", "Analyst", 50000));

displayAll ();

searchEmployee (2);

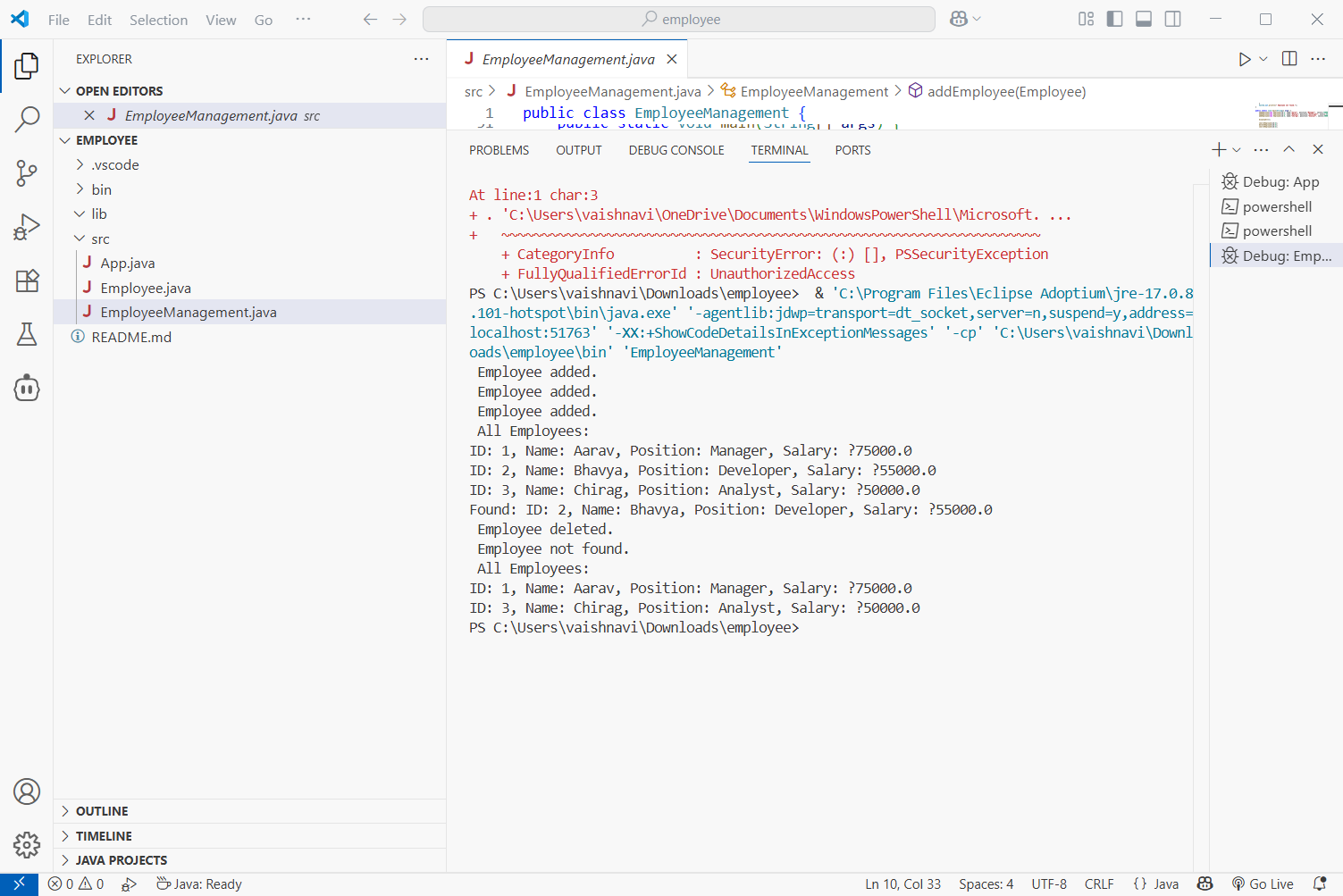
deleteEmployee (2);

searchEmployee (2);

displayAll ();

}

}



**Time Complexity Analysis**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O ( 1) | Insert at the end |
| Search | O(n) | Linear search |
| Traverse | O(n) | Visit each employee |
| Delete | O(n) | Search + shift elements after deletion |

**Limitations of Arrays**

➤ Limitations:

* Fixed size: Must define size in advance
* Inefficient deletions: Requires shifting elements
* No dynamic resizing

➤ When to Use Arrays:

* When data size is known in advance
* When fast index-based access is required
* For memory-efficient storage of simple structures

**Exercise 5: Task Management System**

➤ Types of Linked Lists:

| Type | Description |
| --- | --- |
| Singly Linked List | Each node points to the next node only. One direction traversal. |
| Doubly Linked List | Each node has pointers to both next and previous nodes. |

➤ Use of Linked Lists

* Arrays have a fixed size; resizing is costly.
* Linked lists offer dynamic memory allocation.
* Efficient insertions/deletions at beginning or middle.

**Task.java**

class Task {

int taskId;

String taskName;

String status;

Task next;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

this.next = null;

}

@Override

public String toString() {

return "TaskID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

**TaskMangaer.java**

public class TaskManager {

    static Task head = null;

    // Add task at end

    public static 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);

    }

    // Search task by ID

    public static void searchTask(int taskId) {

        Task temp = head;

        while (temp != null) {

            if (temp.taskId == taskId) {

                System.out.println("Found: " + temp);

                return;

            }

            temp = temp.next;

        }

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

    }

    // Traverse and display all tasks

    public static void displayTasks() {

        if (head == null) {

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

            return;

        }

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

        Task temp = head;

        while (temp != null) {

            System.out.println(temp);

            temp = temp.next;

        }

    }

    // Delete task by ID

    public static void deleteTask(int taskId) {

        if (head == null) {

            System.out.println(" No tasks to delete.");

            return;

        }

        if (head.taskId == taskId) {

            head = head.next;

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

            return;

        }

        Task current = head, prev = null;

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

            prev = current;

            current = current.next;

        }

        if (current == null) {

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

        } else {

            prev.next = current.next;

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

        }

    }

    public static void main(String[] args) {

        addTask(1, "Design UI", "Pending");

        addTask(2, "Develop Backend", "In Progress");

        addTask(3, "Test Features", "Pending");

        displayTasks();

        searchTask(2);

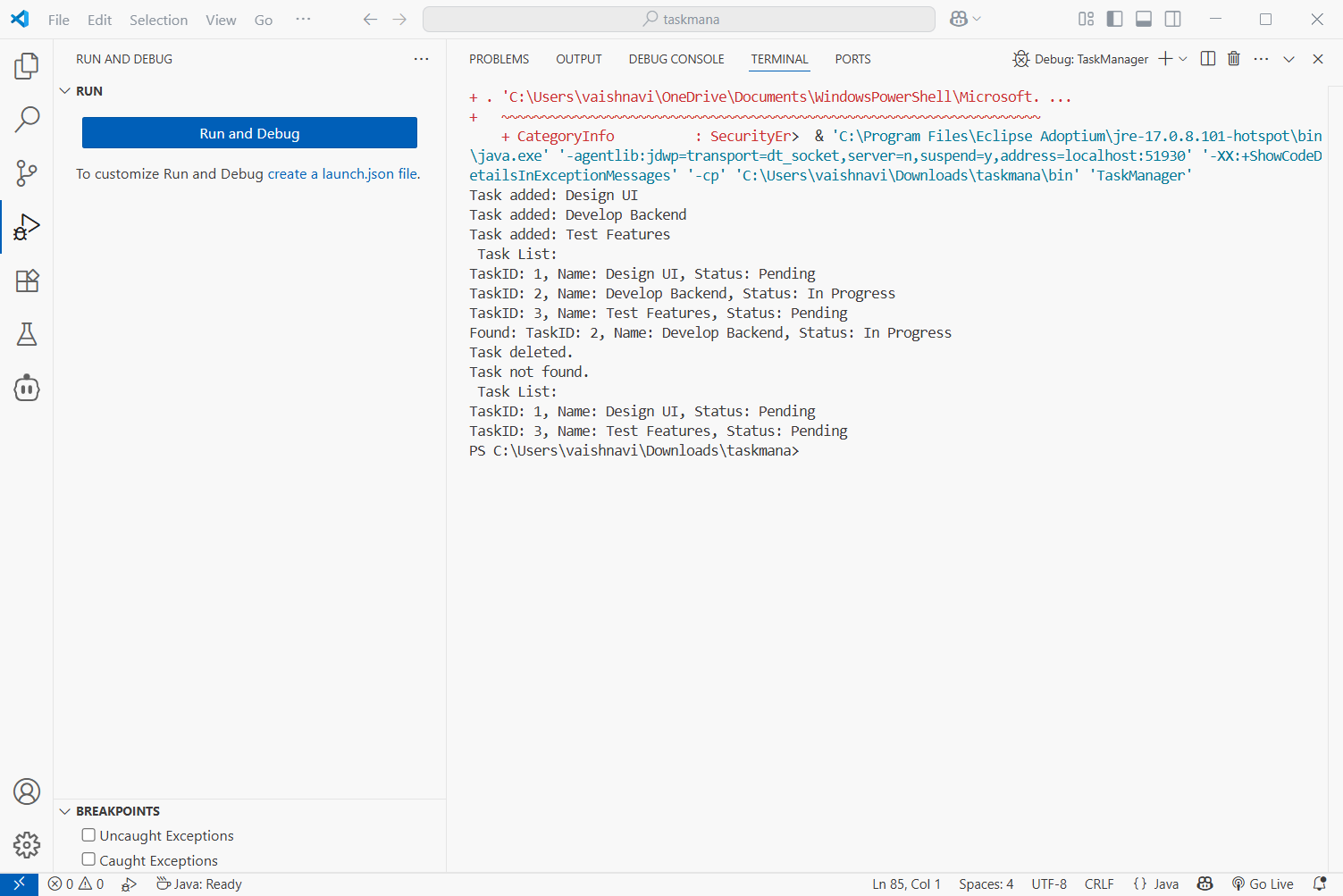
        deleteTask(2);

        searchTask(2);

        displayTasks();

    }

}



**Analysis**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O(n) | Traverse to the end |
| Search | O(n) | Linear search |
| Traverse | O(n) | Visit each node |
| Delete | O(n) | Need to search + adjust pointers |

➤ Advantages of Linked Lists Over Arrays:

* Dynamic size: no need to declare size beforehand.
* Efficient insert/delete at head or middle.
* No shifting needed unlike arrays.

Exercise 6: Library Management System

➤ Linear Search

* Goes through each element one by one.
* Time Complexity:
  + Best: O(1)
  + Average/Worst: O(n)

➤ Binary Search

* Requires sorted data.
* Repeatedly divides the list and compares the middle element.
* Time Complexity:
  + Best: O(1)
  + Average/Worst: O(log n)

**Book.java**

class Book {

int bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

@Override

public String toString() {

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

}

}

**LibraryManagement**

import java.util.Arrays;

import java.util.Comparator;

public class LibraryManagement {

// Linear Search by title

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

for (Book b : books) {

if (b.title.equalsIgnoreCase(title)) {

return b;

}

}

return null;

}

// Binary Search by title (Assumes sorted)

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

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

else if (comp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Book[] books = {

new Book(101, "Data Structures", "Mark Allen"),

new Book(102, "Algorithms", "Cormen"),

new Book(103, "Artificial Intelligence", "Stuart Russell"),

new Book(104, "Computer Networks", "Tanenbaum")

};

// Linear Search

System.out.println("Linear Search for 'Algorithms':");

Book found1 = linearSearch(books, "Algorithms");

System.out.println(found1 != null ? found1 : "Book not found.");

// Sort books by title for Binary Search

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

// Binary Search

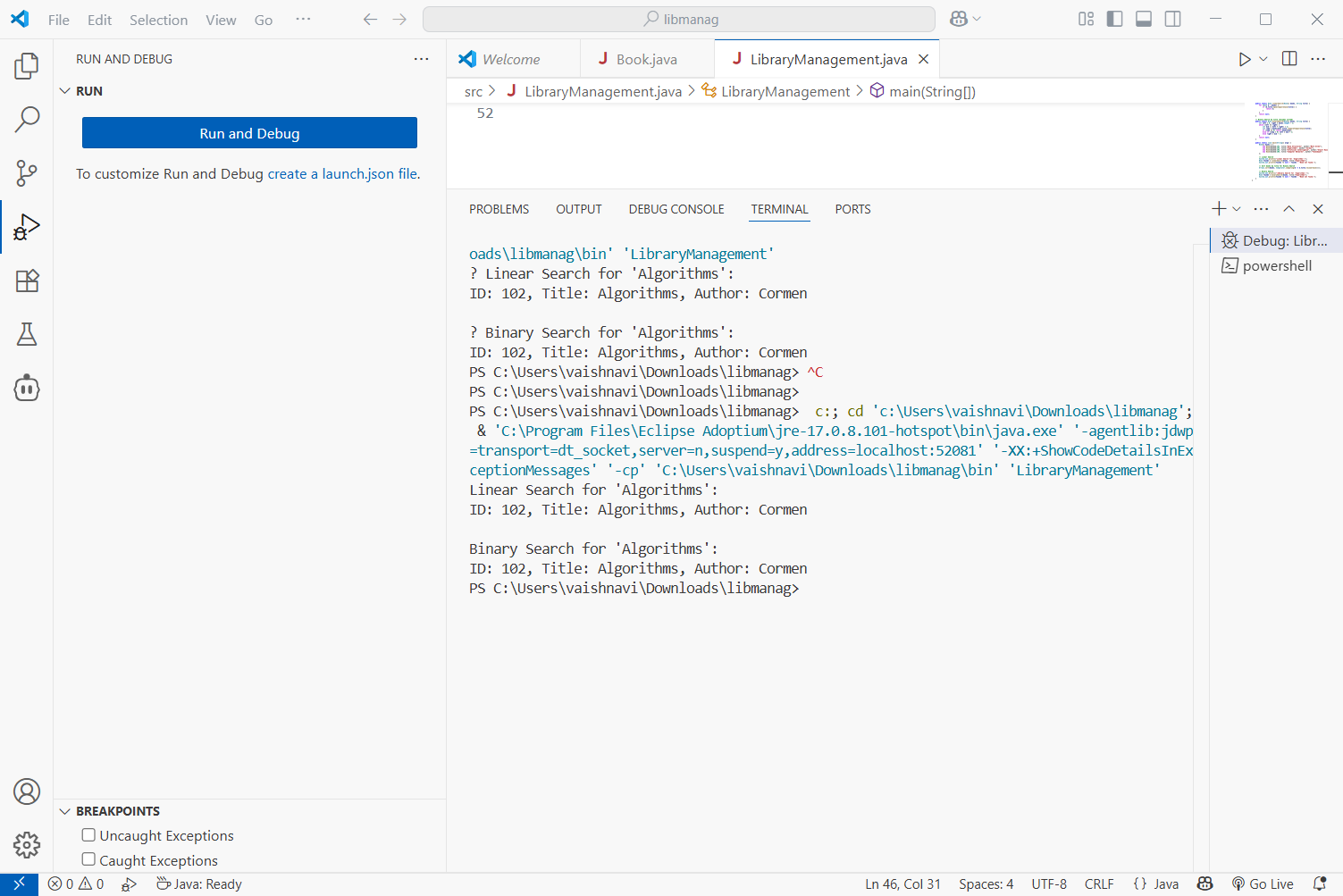
System.out.println("\nBinary Search for 'Algorithms':");

Book found2 = binarySearch(books, "Algorithms");

System.out.println(found2 != null ? found2 : "Book not found.");

}

}



**Analysis**

| **Algorithm** | **Time Complexity** | **Use Case** |
| --- | --- | --- |
| Linear Search | O(n) | Use when data is **unsorted** |
| Binary Search | O(log n) | Use when data is **sorted** |

* **Linear Search** is best for small datasets or if sorting is not practical.
* **Binary Search** is preferred for **large sorted datasets** for faster lookup.