**PAKHI SHARMA**

**Exercise 1: Inventory Management System**

**Scenario:**

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

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

**CODE**

**import java.util.\*;**

**class Product {**

**int productId;**

**String productName;**

**int quantity;**

**double price;**

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

**this.productId = id;**

**this.productName = name;**

**this.quantity = qty;**

**this.price = price;**

**}**

**}**

**class Inventory {**

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

**public void addProduct(Product p) {**

**products.put(p.productId, p);**

**}**

**public void updateProduct(Product p) {**

**products.put(p.productId, p);**

**}**

**public void deleteProduct(int id) {**

**products.remove(id);**

**}**

**public Product getProduct(int id) {**

**return products.get(id);**

**}**

**public void displayAll() {**

**for (Product p : products.values()) {**

**System.out.println(p.productId + " " + p.productName + " " + p.quantity + " " + p.price);**

**}**

**}**

**}**

**public class Main {**

**public static void main(String[] args) {**

**Inventory inv = new Inventory();**

**inv.addProduct(new Product(1, "Mouse", 50, 499.0));**

**inv.addProduct(new Product(2, "Keyboard", 30, 899.0));**

**inv.updateProduct(new Product(2, "Mechanical Keyboard", 20, 1299.0));**

**inv.displayAll();**

**inv.deleteProduct(1);**

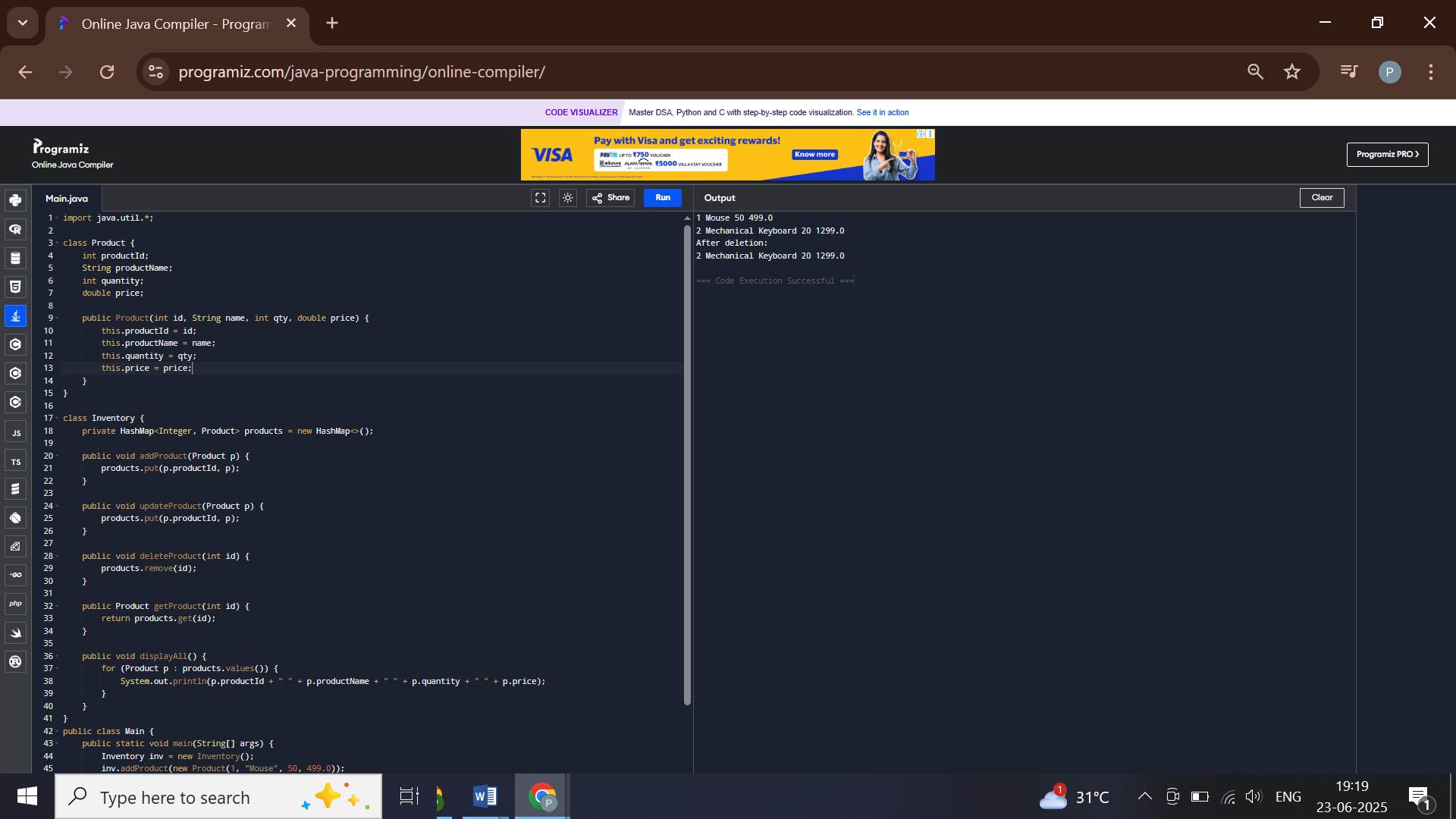
**System.out.println("After deletion:");**

**inv.displayAll();**

**}**

**}**

**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.

**Steps:**

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

**CODE**

**import java.util.Arrays;**

**import java.util.Comparator;**

**class Product {**

**int productId;**

**String productName;**

**String category;**

**public Product(int id, String name, String cat) {**

**this.productId = id;**

**this.productName = name;**

**this.category = cat;**

**}**

**}**

**public class Main {**

**public static int linearSearch(Product[] arr, String name) {**

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

**if (arr[i].productName.equals(name)) return i;**

**}**

**return -1;**

**}**

**public static int binarySearch(Product[] arr, String name, int low, int high) {**

**if (low <= high) {**

**int mid = (low + high) / 2;**

**int cmp = arr[mid].productName.compareTo(name);**

**if (cmp == 0) return mid;**

**else if (cmp > 0) return binarySearch(arr, name, low, mid - 1);**

**else return binarySearch(arr, name, mid + 1, high);**

**}**

**return -1;**

**}**

**public static void main(String[] args) {**

**Product[] products = {**

**new Product(1, "Charger", "Electronics"),**

**new Product(2, "Headphones", "Audio"),**

**new Product(3, "Keyboard", "Computer"),**

**new Product(4, "Mouse", "Computer")**

**};**

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

**int linearIndex = linearSearch(products, "Keyboard");**

**System.out.println("Found at index: " + linearIndex);**

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

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

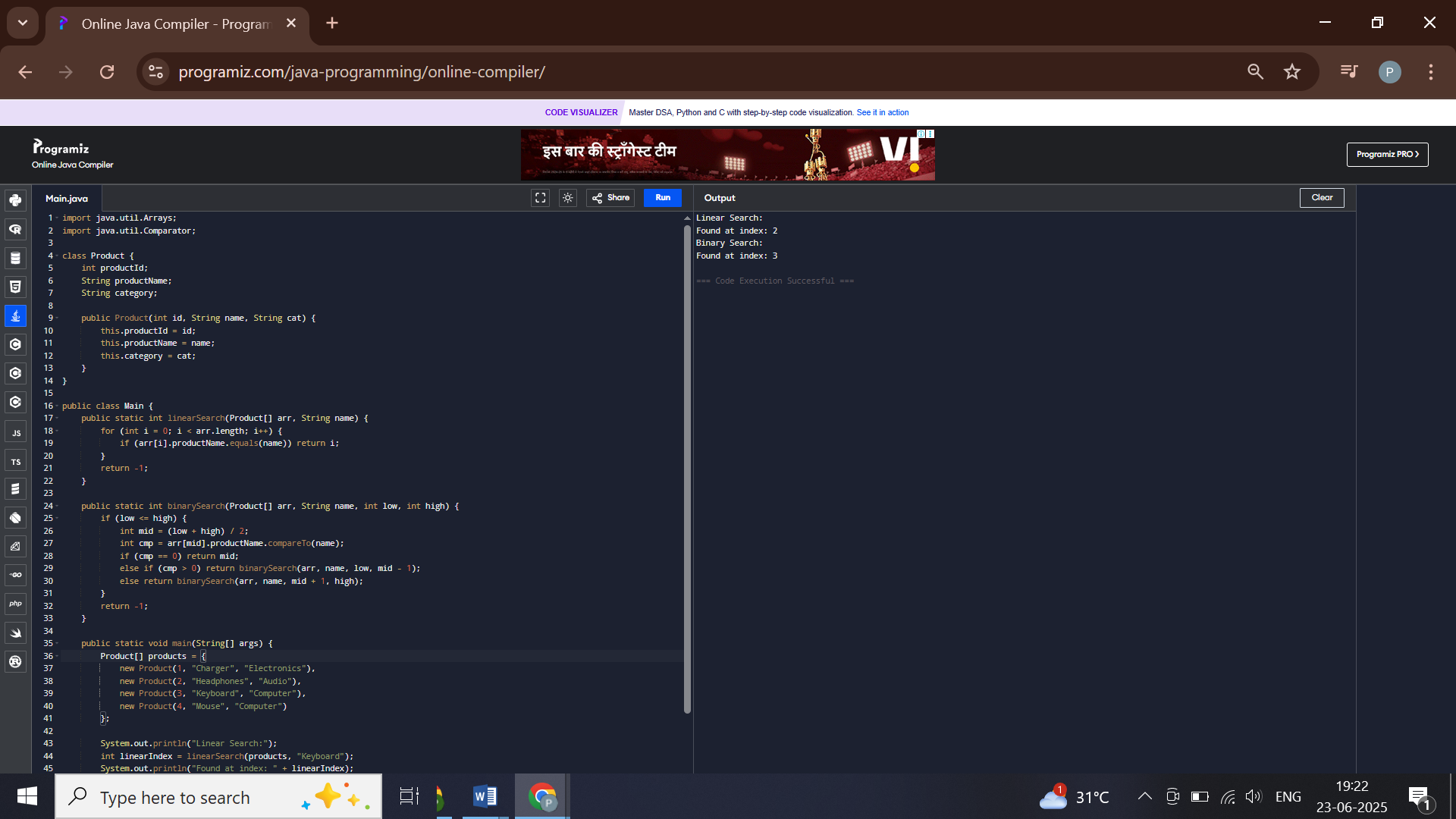
**int binaryIndex = binarySearch(products, "Mouse", 0, products.length - 1);**

**System.out.println("Found at index: " + binaryIndex);**

**}**

**}**

**OUTPUT**



**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.

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**CODE**

**class Order {**

**int orderId;**

**String customerName;**

**double totalPrice;**

**public Order(int id, String name, double price) {**

**this.orderId = id;**

**this.customerName = name;**

**this.totalPrice = price;**

**}**

**}**

**public class Main {**

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

**if (low < high) {**

**int pi = partition(arr, low, high);**

**quickSort(arr, low, pi - 1);**

**quickSort(arr, pi + 1, high);**

**}**

**}**

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

**double pivot = arr[high].totalPrice;**

**int i = low - 1;**

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

**if (arr[j].totalPrice < pivot) {**

**i++;**

**Order temp = arr[i];**

**arr[i] = arr[j];**

**arr[j] = temp;**

**}**

**}**

**Order temp = arr[i + 1];**

**arr[i + 1] = arr[high];**

**arr[high] = temp;**

**return i + 1;**

**}**

**public static void main(String[] args) {**

**Order[] orders = {**

**new Order(1, "Alice", 2500.0),**

**new Order(2, "Bob", 1200.0),**

**new Order(3, "Charlie", 4000.0)**

**};**

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

**for (Order o : orders) {**

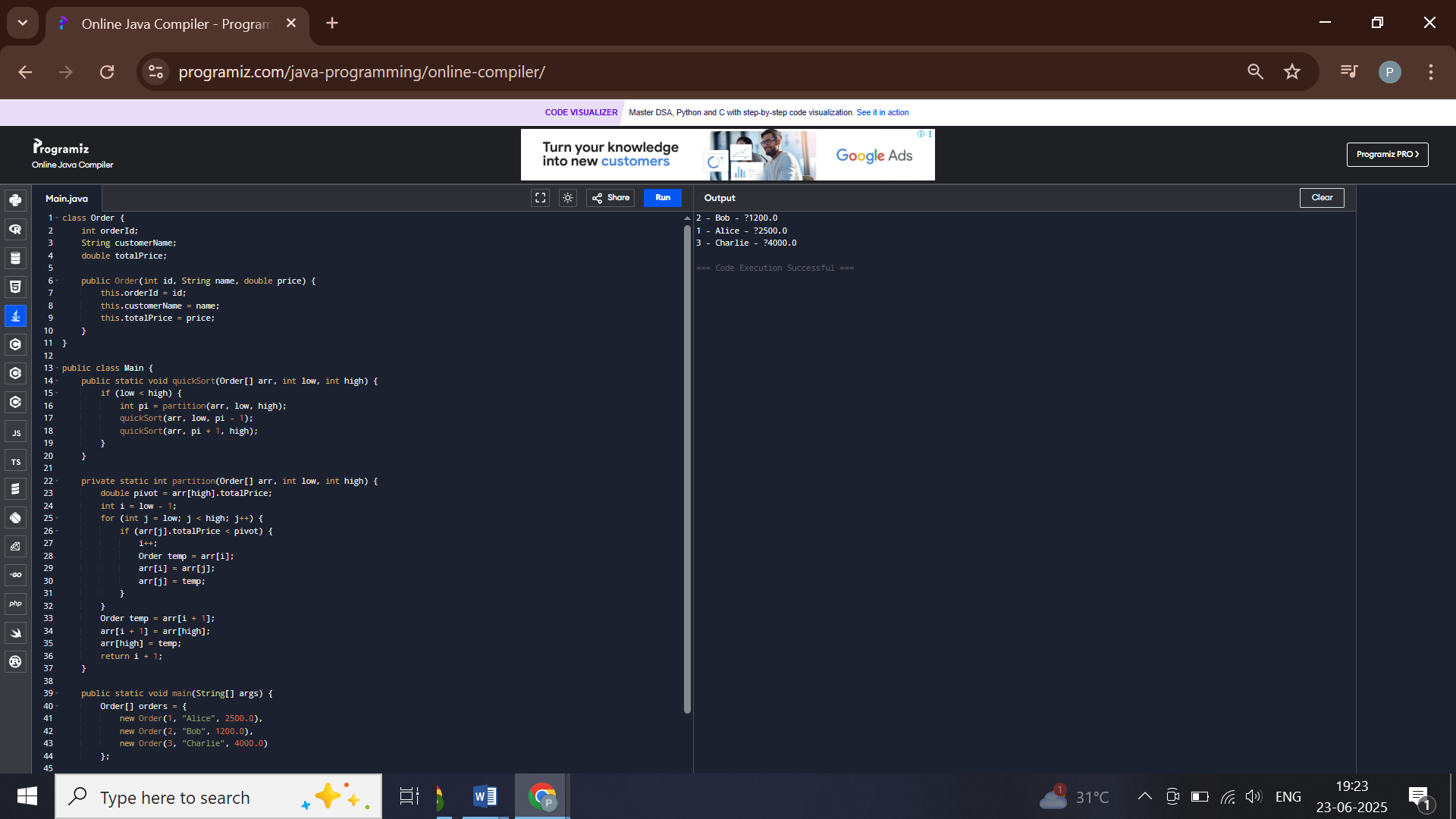
**System.out.println(o.orderId + " - " + o.customerName + " - ₹" + o.totalPrice);**

**}**

**}**

**}**

**OUTPUT**



**Exercise 4: Employee Management System**

**Scenario:**

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

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**CODE**

**class Employee {**

**int employeeId;**

**String name;**

**String position;**

**double salary;**

**public Employee(int id, String name, String pos, double sal) {**

**this.employeeId = id;**

**this.name = name;**

**this.position = pos;**

**this.salary = sal;**

**}**

**}**

**public class Main {**

**static Employee[] employees = new Employee[100];**

**static int count = 0;**

**public static void addEmployee(Employee e) {**

**if (count < employees.length) {**

**employees[count++] = e;**

**}**

**}**

**public static Employee searchEmployee(int id) {**

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

**if (employees[i].employeeId == id) return employees[i];**

**}**

**return null;**

**}**

**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;**

**break;**

**}**

**}**

**}**

**public static void listEmployees() {**

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

**Employee e = employees[i];**

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

**}**

**}**

**public static void main(String[] args) {**

**addEmployee(new Employee(101, "Pakhi", "Manager", 50000));**

**addEmployee(new Employee(102, "Lakshya", "Analyst", 30000));**

**listEmployees();**

**System.out.println("After deletion:");**

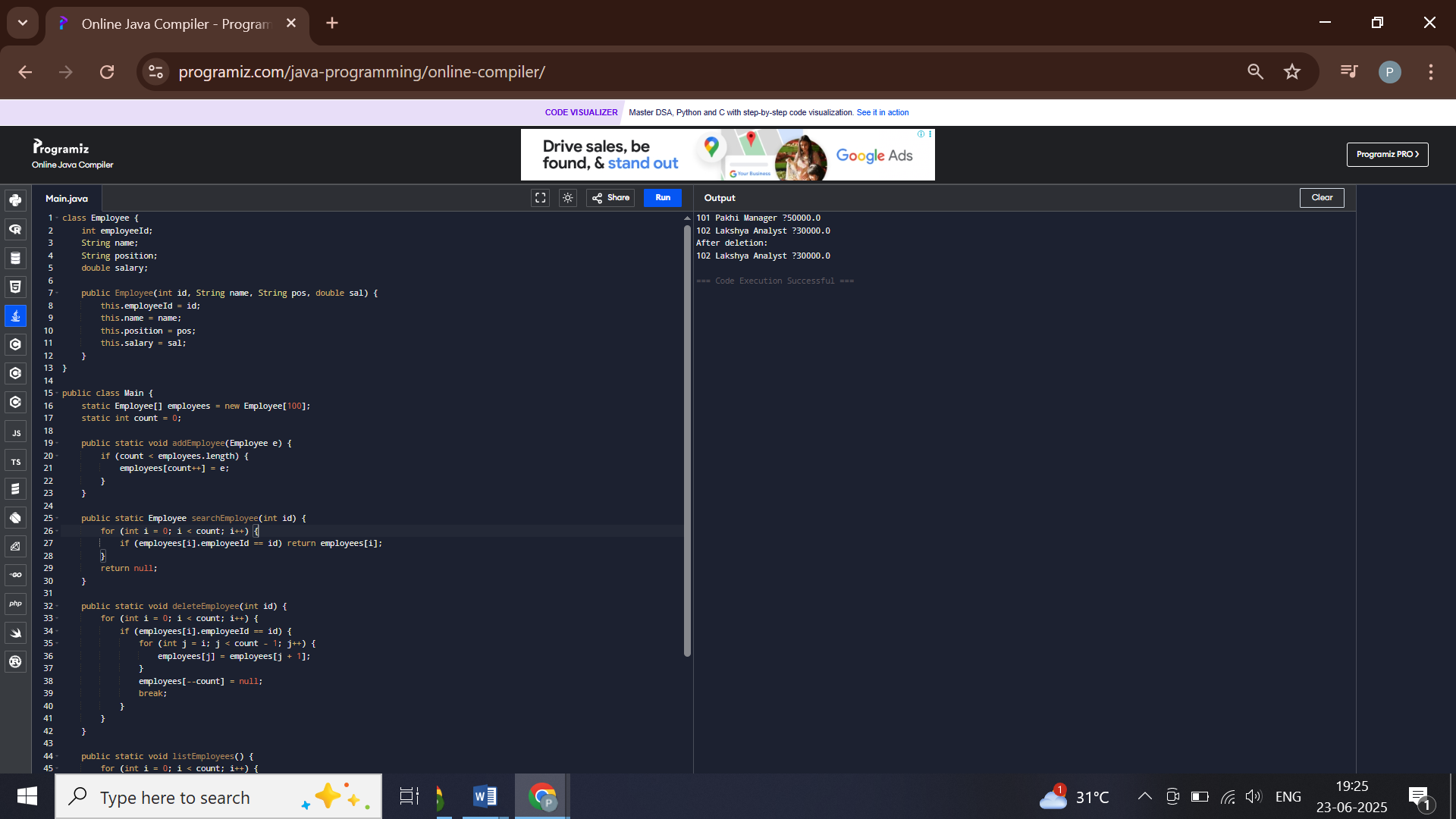
**deleteEmployee(101);**

**listEmployees();**

**}**

**}**

**OUTPUT**



**Exercise 5: Task Management System**

**Scenario:**

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

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**CODE**

**class Task {**

**int taskId;**

**String taskName;**

**String status;**

**Task next;**

**public Task(int id, String name, String status) {**

**this.taskId = id;**

**this.taskName = name;**

**this.status = status;**

**}**

**}**

**public class Main {**

**static Task head = null;**

**public static void addTask(Task task) {**

**if (head == null) {**

**head = task;**

**} else {**

**Task temp = head;**

**while (temp.next != null) temp = temp.next;**

**temp.next = task;**

**}**

**}**

**public static void deleteTask(int id) {**

**if (head == null) return;**

**if (head.taskId == id) {**

**head = head.next;**

**return;**

**}**

**Task prev = head, curr = head.next;**

**while (curr != null) {**

**if (curr.taskId == id) {**

**prev.next = curr.next;**

**return;**

**}**

**prev = curr;**

**curr = curr.next;**

**}**

**}**

**public static Task searchTask(int id) {**

**Task temp = head;**

**while (temp != null) {**

**if (temp.taskId == id) return temp;**

**temp = temp.next;**

**}**

**return null;**

**}**

**public static void listTasks() {**

**Task temp = head;**

**while (temp != null) {**

**System.out.println(temp.taskId + " " + temp.taskName + " [" + temp.status + "]");**

**temp = temp.next;**

**}**

**}**

**public static void main(String[] args) {**

**addTask(new Task(1, "Write report", "Pending"));**

**addTask(new Task(2, "Review PR", "In Progress"));**

**listTasks();**

**System.out.println("After deleting task 1:");**

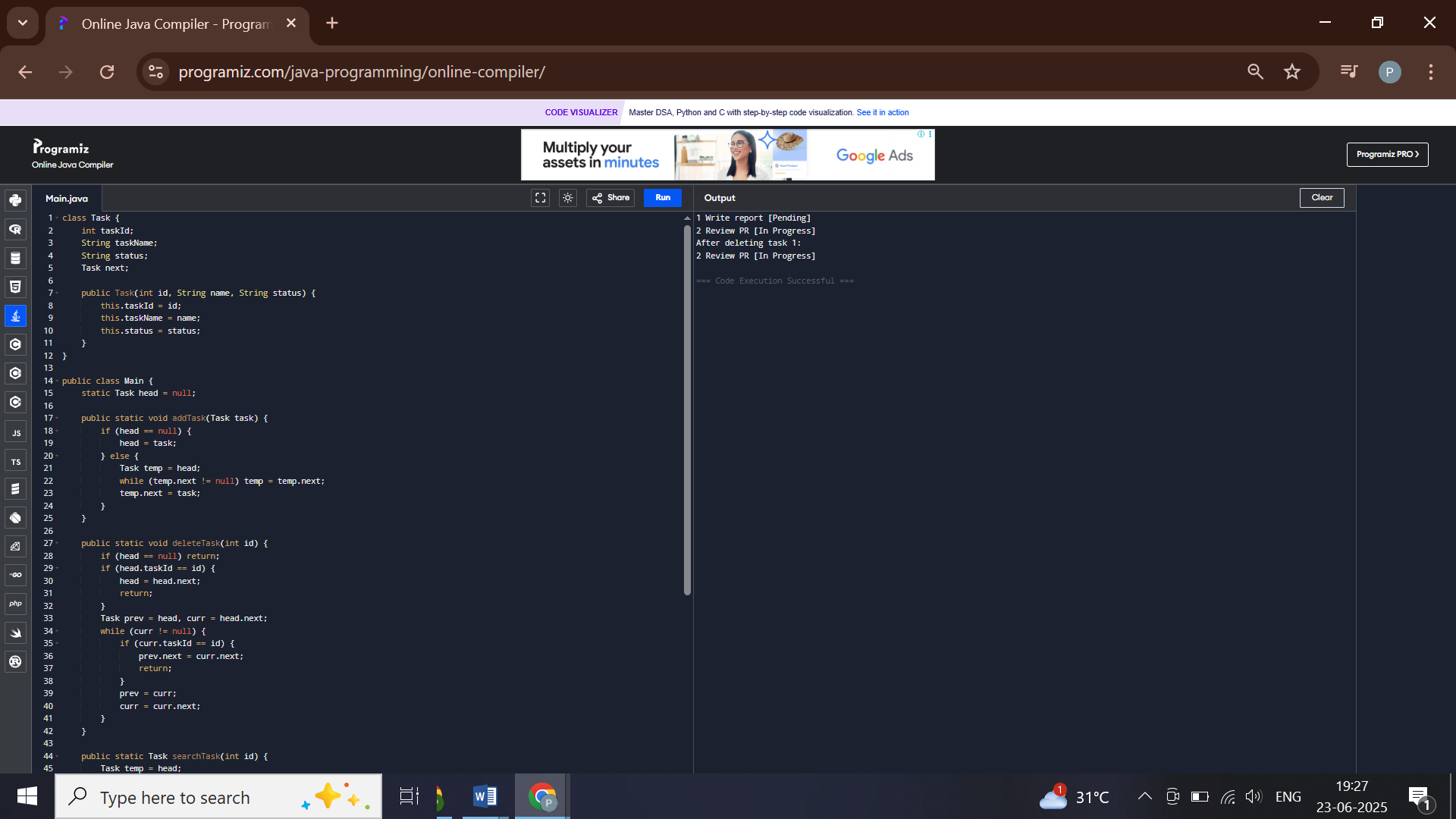
**deleteTask(1);**

**listTasks();**

**}**

**}**

**OUTPUT**



**Exercise 6: Library Management System**

**Scenario:**

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

**Steps:**

1. **Understand Search Algorithms:**
   * Explain linear search and binary search algorithms.
2. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

**CODE**

**class Book {**

**int bookId;**

**String title;**

**String author;**

**public Book(int id, String title, String author) {**

**this.bookId = id;**

**this.title = title;**

**this.author = author;**

**}**

**}**

**public class Main {**

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

**for (Book b : books) {**

**if (b.title.equals(title)) return b;**

**}**

**return null;**

**}**

**public static Book binarySearch(Book[] books, String title, int low, int high) {**

**if (low <= high) {**

**int mid = (low + high) / 2;**

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

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

**else if (cmp > 0) return binarySearch(books, title, low, mid - 1);**

**else return binarySearch(books, title, mid + 1, high);**

**}**

**return null;**

**}**

**public static void main(String[] args) {**

**Book[] books = {**

**new Book(1, "Algorithms", "Cormen"),**

**new Book(2, "Database Systems", "Ramakrishnan"),**

**new Book(3, "Java Programming", "Herbert Schildt")**

**};**

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

**Book result1 = linearSearch(books, "Java Programming");**

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

**java.util.Arrays.sort(books, java.util.Comparator.comparing(b -> b.title));**

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

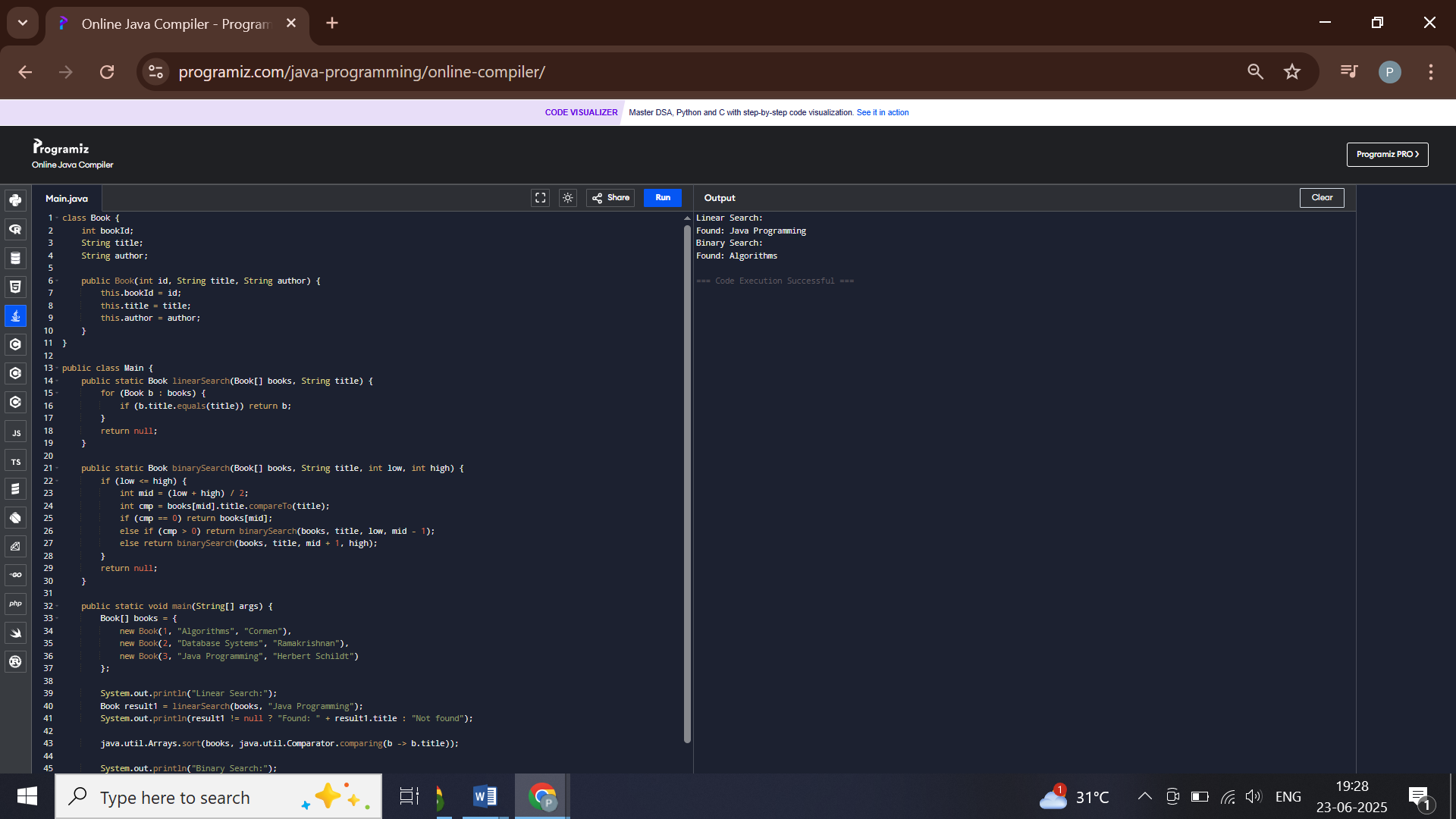
**Book result2 = binarySearch(books, "Algorithms", 0, books.length - 1);**

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

**}**

**}**

**OUTPUT**



**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

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

**CODE**

**public class Main {**

**// Recursive function to calculate future value**

**public static double futureValue(int year, double value, double growthRate) {**

**if (year == 0) return value;**

**return (1 + growthRate) \* futureValue(year - 1, value, growthRate);**

**}**

**// Optimized with memoization**

**public static double futureValueMemo(int year, double value, double growthRate, double[] memo) {**

**if (year == 0) return value;**

**if (memo[year] != 0) return memo[year];**

**memo[year] = (1 + growthRate) \* futureValueMemo(year - 1, value, growthRate, memo);**

**return memo[year];**

**}**

**public static void main(String[] args) {**

**int years = 5;**

**double start = 10000;**

**double rate = 0.10;**

**System.out.println("Recursive (no memo): " + futureValue(years, start, rate));**

**double[] memo = new double[years + 1];**

**System.out.println("Recursive with memo: " + futureValueMemo(years, start, rate, memo));**

**}**

**}**

**OUTPUT**

