**ALGORITHM DATASTRUCTURES SOLUTIONS:**

**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.*

1. *Setup:*

* *Create a new project for the inventory management system.*

1. *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.*

1. *Analysis:*

* *Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.*
* *Discuss how you can optimize these operations.*

**UNDERSTAND THE PROBLEM:**

**Explain why data structures and algorithms are essential in handling large inventories.**In an inventory management system, a large number of products are handled. Operations like search, add, update, and delete need to be performed frequently. Without using proper data structures and algorithms, these operations can become slow and inefficient as data grows. Data structures help in organizing and managing data efficiently, while algorithms ensure the operations are optimized. This improves performance, scalability, and accuracy.

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

* **HashMap:** Best suited for fast access using productId as key. It provides average-case O(1) time for add, search, and delete.
* **ArrayList:** Useful for ordered data or index-based access, but searching or deleting by productId takes O(n) time.
* **TreeMap:** Maintains sorted keys and provides O(log n) time for basic operations. Suitable when sorted output is required.
* **LinkedList**: Rarely suitable for inventory systems unless insertions and deletions are frequent and order matters.

**Conclusion:** HashMap is the best choice for fast product lookup using productId.

**SOLUTION:**

import java.util.\*;

class ProductDetails {

    String productName; int price;int quantity;

    ProductDetails(String productName, int price, int quantity) {

        this.productName = productName;  this.price = price;  this.quantity = quantity; }}

class InventoryManager {

    static HashMap<Integer, ProductDetails> warHouse = new HashMap<>();

     int productId;

    String productName;

    int price;

    int quantity;

    InventoryManager(int productId, String productName, int price, int quantity) {

        this.productId = productId;

        this.productName = productName;

        this.price = price;

        this.quantity = quantity;

}

    InventoryManager(int productId) {

        this.productId = productId;

}

    void add() {

        if (warHouse.containsKey(productId)) {

            System.out.println("Product with this ID already exists. Use update instead.");

        } else {

            warHouse.put(productId, new ProductDetails(productName, price, quantity));

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

}

}

    void update() {

        if (warHouse.containsKey(productId)) {

            warHouse.put(productId, new ProductDetails(productName, price, quantity));

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

        } else {

            System.out.println("Product not found. Use add to create a new product.");

        }

}

    void delete() {

        if (warHouse.containsKey(productId)) {

            warHouse.remove(productId);

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

        } else {

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

        }

}

}

public class Main {

    public static void main(String[] args) {

        Scanner obj = new Scanner(System.in);

        while (true) {

            System.out.println("\nEnter command (add, update, delete, view, exit): ");

            String cmd = obj.next();

             if (cmd.equalsIgnoreCase("add") || cmd.equalsIgnoreCase("update")) {

                System.out.print("Enter Product ID: ");

                int productId = obj.nextInt();

                System.out.print("Enter Product Name: ");

                String productName = obj.next();

                System.out.print("Enter Product Price: ");

                int price = obj.nextInt();

                System.out.print("Enter Product Quantity: ");

                int quantity = obj.nextInt();

                InventoryManager inv = new InventoryManager(productId, productName, price, quantity);

                if (cmd.equalsIgnoreCase("add")) {

                    inv.add();

                } else {

                    inv.update();}

            } else if (cmd.equalsIgnoreCase("delete")) {

                System.out.print("Enter Product ID to delete: ");

                int productId = obj.nextInt();

                InventoryManager inv = new InventoryManager(productId);

                inv.delete();

            } else if (cmd.equalsIgnoreCase("view")) {

                if (InventoryManager.warHouse.isEmpty()) {

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

                } else {

                    for (Map.Entry<Integer, ProductDetails> entry : InventoryManager.warHouse.entrySet()) {

                        int id = entry.getKey();

                        ProductDetails details = entry.getValue();

                        System.out.println("Product ID: " + id + ", Name: " + details.productName + ", Price: " + details.price + ", Quantity: " + details.quantity);

 }

}

             } else if (cmd.equalsIgnoreCase("exit")) {

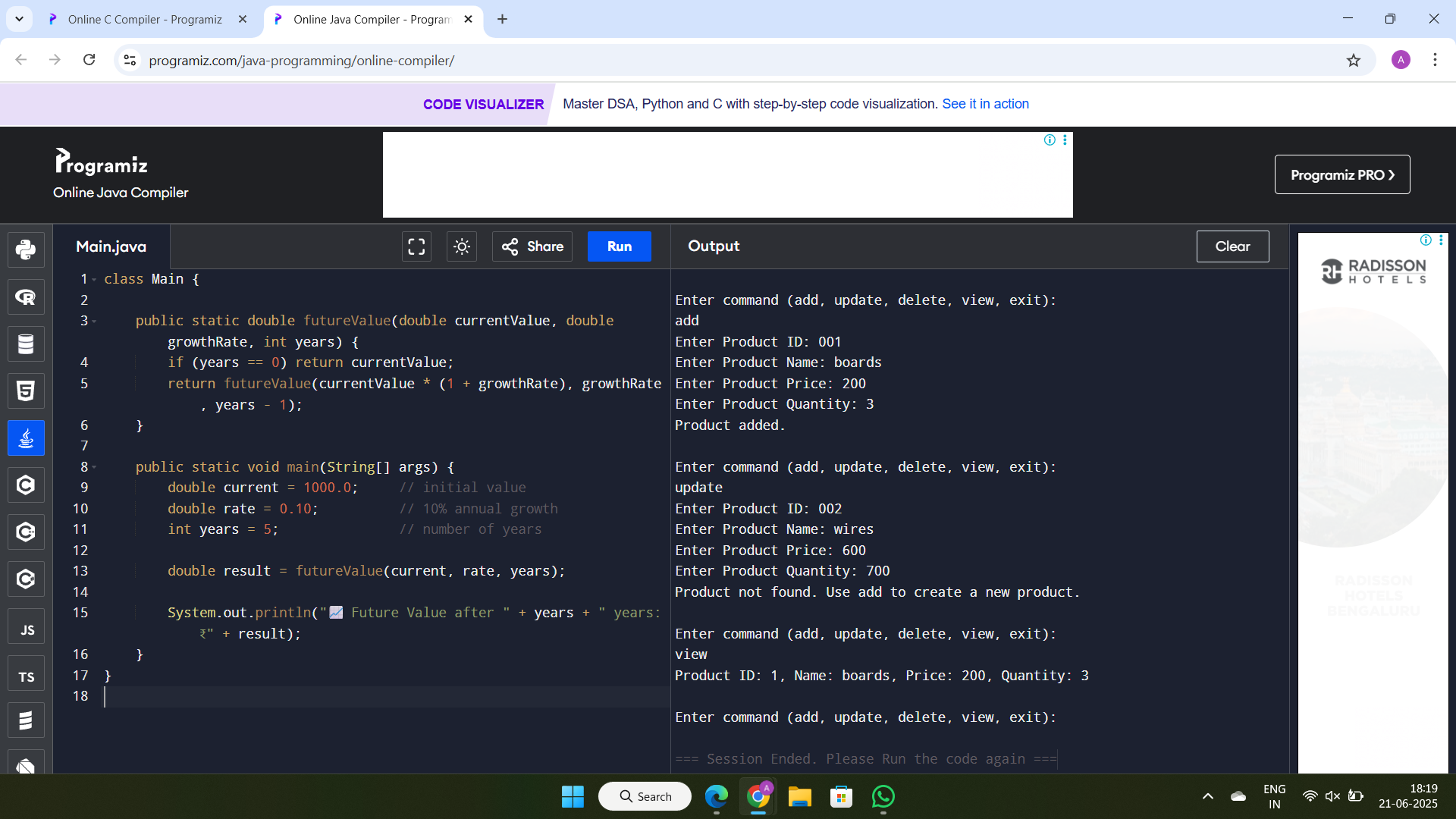
                System.out.println("Exiting...");

                break;

            } else {

                System.out.println("Unknown command.");}}

        obj.close();}}

**OUTPUT:**

**ANALYSIS:**

**Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.**

* Add Product : put(productId, product) - O(1) average time
* Update Product :get(productId) and set values - O(1)
* Delete Product :remove(productId) -O(1)
* View Inventory: iterate values() -O(n)

**Discuss how you can optimize these operations.**

* Use HashMap for productId-based access.
* Use TreeMap or sorting when sorted output is needed.
* Maintain secondary indexes (e.g., Map<String, List<Product>>) for name/category-based search.
* Cache frequently accessed products.
* Avoid using ArrayList when frequent search or delete is needed by ID.

**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.*

1. *Setup:*

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

1. *Implementation:*

* *Implement linear search and binary search algorithms.*
* *Store products in an array for linear search and a sorted array for binary search.*

1. *Analysis:*

* *Compare the time complexity of linear and binary search algorithms.*
* *Discuss which algorithm is more suitable for your platform and why.*

**UNDERSTAND ASYMPTOTIC NOTATION:**

**Explain Big O notation and how it helps in analyzing algorithms.**Big O notation is a mathematical way to describe the upper bound of an algorithm's running time or space requirement in terms of input size (n). It helps compare the efficiency of algorithms without worrying about hardware or programming language. It shows how quickly runtime grows as input size increases, allowing us to select the most efficient algorithm for large inputs.

**Common examples:**

* O(1): Constant time
* O(n): Linear time
* O(log n): Logarithmic time
* O(n²): Quadratic time

**Describe the best, average, and worst-case scenarios for search operations**

* **Best Case:** The condition is met immediately. For example, in linear search, the element is found at the first position - O(1).
* **Average Case:** The element is found somewhere in the middle. For linear search, it takes about n/2 comparisons - O(n).
* **Worst Case:** The element is at the end or not present at all. Linear search compares all elements - O(n).

**In contrast, binary search has:**

* **Best Case:** O(1) (element is at the middle)
* **Average and Worst Case:** O(log n) (search space halves each time)

**SOLUTION:**

import java.util.\*;

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

}

public class Main {

static Product linearSearch(Product[] products, int id) {

for (Product p : products) {

if (p.productId == id) return p;

}

return null;

}

static Product binarySearch(Product[] products, int id) {

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

while (left <= right) {

int mid = (left + right) / 2;

if (products[mid].productId == id) return products[mid];

else if (products[mid].productId < id) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Shoes", "Fashion"),

new Product(2, "Book", "Stationery"),

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

new Product(4, "Bag", "Fashion"),

new Product(5, "Mobile", "Electronics")

};

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

Scanner sc = new Scanner(System.in);

System.out.println("Choose search method:");

System.out.println("1. Linear Search");

System.out.println("2. Binary Search");

System.out.print("Enter choice (1 or 2): ");

int choice = sc.nextInt();

System.out.print("Enter ProductId to search: ");

int id = sc.nextInt();

Product result = null;

if (choice == 1) result = linearSearch(products, id);

else if (choice == 2) result = binarySearch(products, id);

else {

System.out.println("Invalid choice.");

sc.close();

return;

}

if (result != null)

System.out.println("Product found: " + result.productName + " (" + result.category + ")");

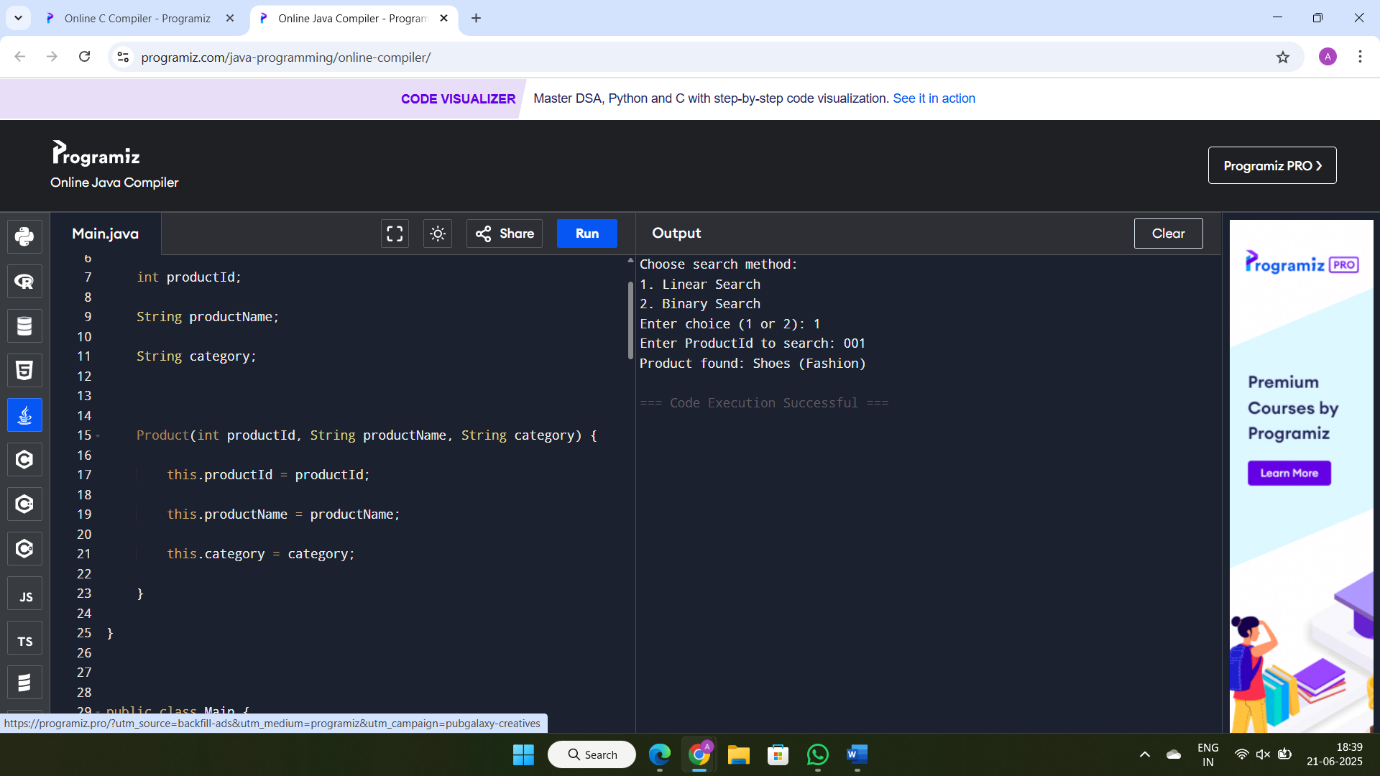
else

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

sc.close();

}

}

**OUTPUT:**

**ANALYSIS:**

**Time complexity:**

* Linear Search - O(n)
* Binary Search - O(log n) (requires sorted array)

Binary search is more suitable for large e-commerce platforms because it reduces the number of comparisons significantly. It performs faster but needs sorted data. If the product list is sorted or can be kept sorted, binary search is the best choice. Linear search is simpler but slower as data grows.

**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).*

1. *Setup:*

* *Create a class Order with attributes like orderId, customerName, and totalPrice.*

1. *Implementation:*

* *Implement Bubble Sort to sort orders by totalPrice.*
* *Implement Quick Sort to sort orders by totalPrice.*

1. *Analysis:*

* *Compare the performance (time complexity) of Bubble Sort and Quick Sort.*
* *Discuss why Quick Sort is generally preferred over Bubble Sort.*

**Understand Sorting Algorithms:**  
o Bubble Sort compares and swaps adjacent elements, simple but slow - O(n²).  
o Insertion Sort builds a sorted list one item at a time - O(n²).  
o Quick Sort uses divide and conquer by selecting a pivot - O(n log n) average.  
o Merge Sort also uses divide and conquer, stable and consistent - O(n log n).

**SOLUTION:**

import java.util.\*;

class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

public class Main {

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;

}

}

}

}

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);

}

}

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;

}

static void printOrders(Order[] orders) {

for (Order o : orders) {

System.out.println("OrderID: " + o.orderId + ", Customer: " + o.customerName + ", Total: " + o.totalPrice);

}

}

public static void main(String[] args) {

Order[] orders = {

new Order(101, "Alice", 450.0),

new Order(102, "Bob", 300.0),

new Order(103, "Charlie", 700.0),

new Order(104, "Diana", 150.0)

};

Scanner sc = new Scanner(System.in);

System.out.println("Choose sorting method:");

System.out.println("1. Bubble Sort");

System.out.println("2. Quick Sort");

System.out.print("Enter choice (1 or 2): ");

int choice = sc.nextInt();

if (choice == 1) {

bubbleSort(orders);

} else if (choice == 2) {

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

} else {

System.out.println("Invalid choice.");

sc.close();

return;

}

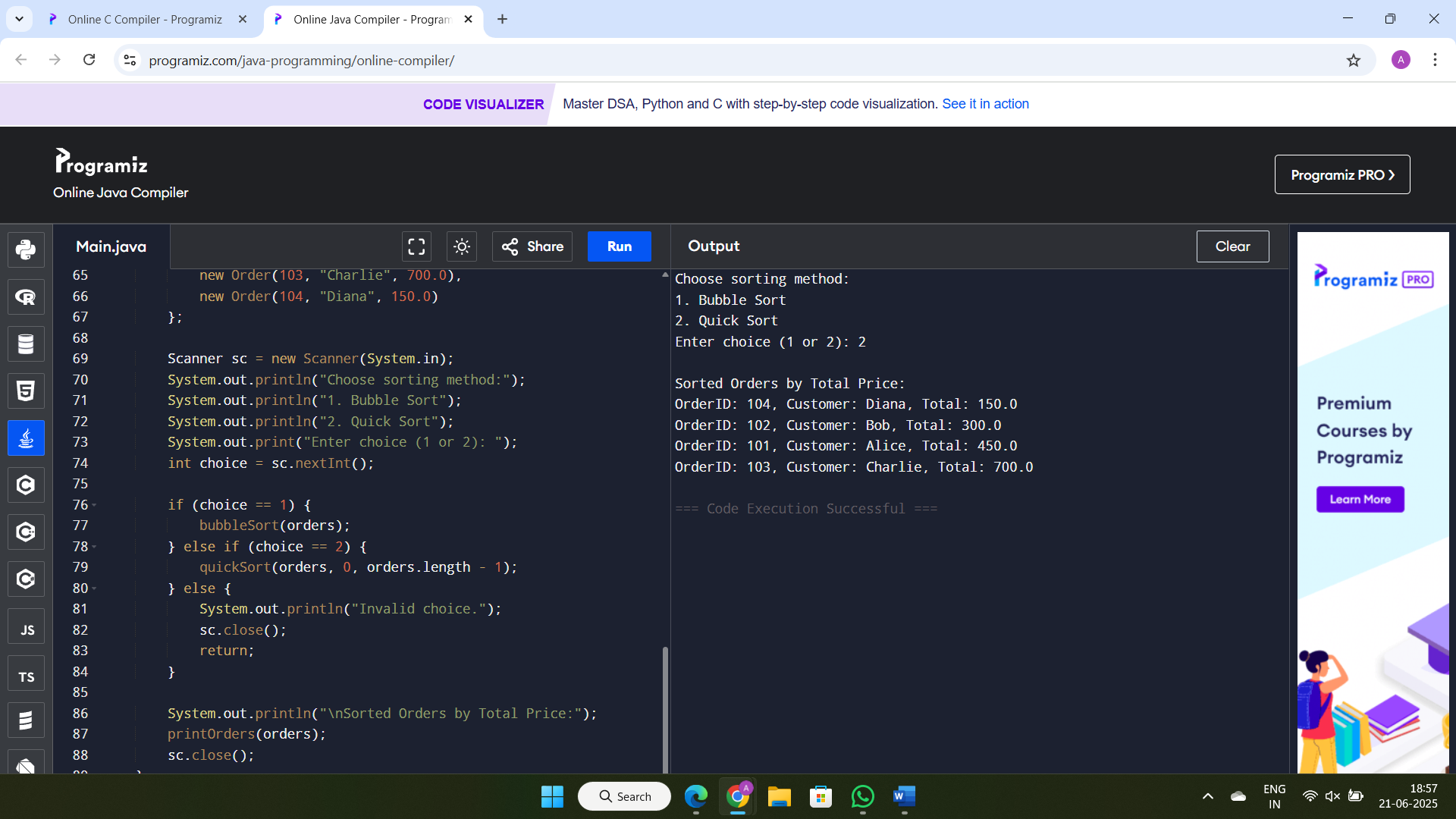
System.out.println("\nSorted Orders by Total Price:");

printOrders(orders);

sc.close();

}

}

**OUTPUT:**

**ANALYSIS:**

o Bubble Sort - O(n²), slow for large data  
o Quick Sort - O(n log n) average, O(n²) worst-case but very fast in practice  
o Quick Sort is preferred because it’s faster, uses less memory than Merge Sort, and handles large datasets efficiently. Bubble Sort is simple but not suitable for production use due to its slowness.

**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.*

1. *Setup:*

* *Create a class Employee with attributes like employeeId, name, position, and salary.*

1. *Implementation:*

* *Use an array to store employee records.*
* *Implement methods to add, search, traverse, and delete employees in the array.*

1. *Analysis:*

* *Analyze the time complexity of each operation (add, search, traverse, delete).*
* *Discuss the limitations of arrays and when to use them.*

**UNDERSTAND ARRAY REPRESENTATION:**

Arrays are stored in contiguous memory locations, meaning all elements are placed one after another in memory. This allows direct access to any element using its index, resulting in constant time (O(1)) retrieval. Arrays are simple to use, offer fast indexing, and consume less memory compared to some dynamic data structures. However, the main drawback of arrays is their fixed size once declared, the size cannot be changed. Additionally, insertions or deletions in the middle of the array can be costly, as they require shifting elements to maintain order, which takes linear time (O(n)).

**SOLUTION:**

import java.util.\*;

class Employee {

    int employeeId;

    String name;

    String position;

    double salary;

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

        this.employeeId = employeeId;

        this.name = name;

        this.position = position;

        this.salary = salary;

    }

}

public class Main {

    static void addEmployee(Employee[] employees, int index, Employee newEmployee) {

        if (index >= 0 && index < employees.length) {

            employees[index] = newEmployee;

            System.out.println("Employee added at position " + index);

        } else {

            System.out.println("Index out of bounds. Cannot add employee.");

        }

    }

    static Employee searchEmployee(Employee[] employees, int id) {

        for (Employee e : employees) {

            if (e != null && e.employeeId == id)

                return e;

        }

        return null;

    }

    static void deleteEmployee(Employee[] employees, int id) {

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

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

                employees[i] = null;

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

                return;

            }

        }

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

    }

    static void viewAllEmployees(Employee[] employees) {

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

        boolean found = false;

        for (Employee e : employees) {

            if (e != null) {

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

                found = true;

            }

        }

        if (!found) System.out.println("No employees found.");

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        Employee[] employees = new Employee[5];

        while (true) {

            System.out.println("\nMenu:");

            System.out.println("1. Add Employee");

            System.out.println("2. Search Employee");

            System.out.println("3. Delete Employee");

            System.out.println("4. View All Employees");

            System.out.println("5. Exit");

            System.out.print("Enter choice: ");

            int choice = sc.nextInt();

            if (choice == 1) {

                System.out.print("Enter Index (0-4): ");

                int index = sc.nextInt();

                System.out.print("Enter Employee ID: ");

                int id = sc.nextInt();

                sc.nextLine();

                System.out.print("Enter Name: ");

                String name = sc.nextLine();

                System.out.print("Enter Position: ");

                String position = sc.nextLine();

                System.out.print("Enter Salary: ");

                double salary = sc.nextDouble();

                addEmployee(employees, index, new Employee(id, name, position, salary));

            } else if (choice == 2) {

                System.out.print("Enter Employee ID to search: ");

                int id = sc.nextInt();

                Employee emp = searchEmployee(employees, id);

                if (emp != null) {

                    System.out.println("Found: " + emp.employeeId + " " + emp.name + " " + emp.position + " $" + emp.salary);

                } else {

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

                }

            } else if (choice == 3) {

                System.out.print("Enter Employee ID to delete: ");

                int id = sc.nextInt();

                deleteEmployee(employees, id);

            } else if (choice == 4) {

                viewAllEmployees(employees);

            } else if (choice == 5) {

                System.out.println("Exiting...");

                break;

            } else {

                System.out.println("Invalid choice.");

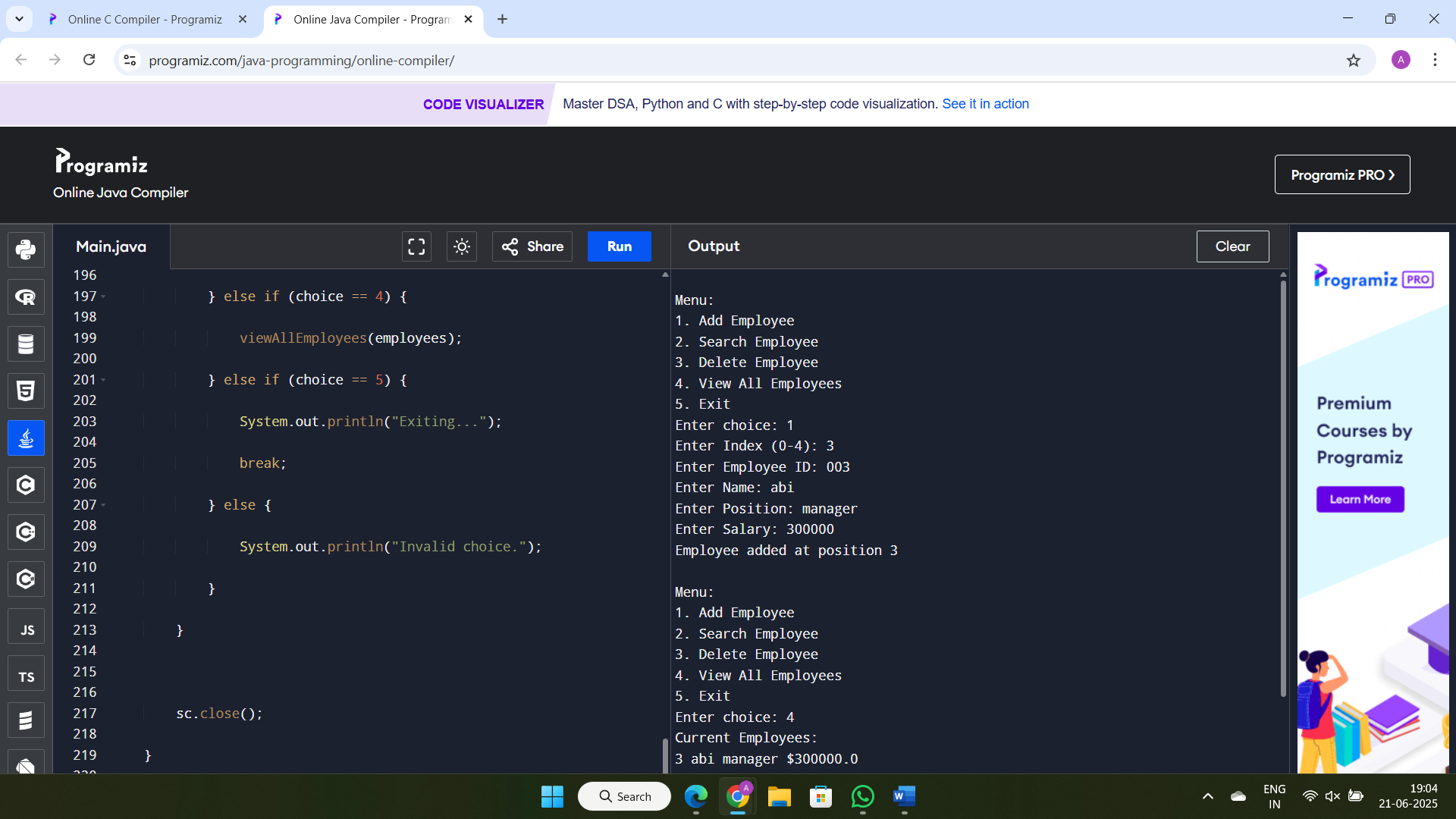
            }

        }

        sc.close();

    }

}

**OUTPUT:**

**ANALYSIS:**

**Time complexity:**  
 Add - O(1) if space is available  
 Search - O(n) in worst case  
 Traverse - O(n)  
 Delete - O(n) due to shifting elements

**Limitation:**

Fixed size, costly insert/delete operations.  
Use arrays when the number of employees is small and mostly append or read-only operations are needed. For dynamic operations, use 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.*

*Steps:*

1. *Understand Linked Lists:*

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

1. *Setup:*

* *Create a class Task with attributes like taskId, taskName, and status.*

1. *Implementation:*

* *Implement a singly linked list to manage tasks.*
* *Implement methods to add, search, traverse, and delete tasks in the linked list.*

1. *Analysis:*

* *Analyze the time complexity of each operation.*
* *Discuss the advantages of linked lists over arrays for dynamic data.*

**UNDERSTAND LINKED LISTS:**

o **Singly Linked List**: Each node contains data and a reference to the next node. It allows forward traversal only. It's simple, uses less memory, and is efficient for inserting or deleting elements at the beginning or middle. However, backward traversal is not possible, and deletion from the end requires full traversal.

o **Doubly Linked List**: Each node contains data, a reference to the next node, and a reference to the previous node. This allows traversal in both directions. It provides more flexibility in insertion and deletion operations (especially backward deletions), but it consumes more memory and slightly more processing due to maintaining two pointers per node.

**SOLUTION:**

import java.util.Scanner;

class Task {

    int taskId;

    String description;

    boolean isDone;

    Task next;

     Task(int taskId, String description) {

        this.taskId = taskId;

        this.description = description;

        this.isDone = false;

        this.next = null;

    }

}

class TaskList {

    Task head;

    void addTask(int taskId, String description) {

        Task newTask = new Task(taskId, description);

        if (head == null) {

            head = newTask;

        } else {

            Task temp = head;

            while (temp.next != null) {

                temp = temp.next;

            }

            temp.next = newTask;

        }

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

    }

    void viewAllTasks() {

        Task temp = head;

        if (temp == null) {

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

            return;

        }

        while (temp != null) {

            System.out.println(temp.taskId + " " + temp.description + " Completed: " + temp.isDone);

            temp = temp.next;

        } }

    void completeTask(int id) {

        Task temp = head;

        while (temp != null) {

            if (temp.taskId == id) {

                temp.isDone = true;

                System.out.println("Task " + id + " marked as completed.");

                return;

            }

            temp = temp.next;

        }

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

    }

    void deleteTask(int id) {

        if (head == null) {

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

            return;

        }

        if (head.taskId == id) {

            head = head.next;

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

            return;

        }

        Task prev = null;

        Task current = head;

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

            prev = current;

            current = current.next;

        }

        if (current == null) {

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

        } else {

            prev.next = current.next;

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

        }

    }

    Task searchTask(int id) {

        Task temp = head;

        while (temp != null) {

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

            temp = temp.next;

        }

        return null;

    }

}

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        TaskList taskList = new TaskList();

        int choice;

        do {

            System.out.println("\n--- Task Manager ---");

            System.out.println("1. Add Task");

            System.out.println("2. View All Tasks");

            System.out.println("3. Complete Task");

            System.out.println("4. Delete Task");

            System.out.println("5. Search Task");

            System.out.println("6. Exit");

            System.out.print("Enter your choice: ");

            choice = sc.nextInt();

            sc.nextLine();

            switch (choice) {

                case 1:

                    System.out.print("Enter Task ID: ");

                    int id = sc.nextInt();

                    sc.nextLine(); // consume newline

                    System.out.print("Enter Task Description: ");

                    String desc = sc.nextLine();

                    taskList.addTask(id, desc);

                    break;

                case 2:

                    taskList.viewAllTasks();

                    break;

                case 3:

                    System.out.print("Enter Task ID to mark as completed: ");

                    int compId = sc.nextInt();

                    taskList.completeTask(compId);

                    break;

                case 4:

                    System.out.print("Enter Task ID to delete: ");

                    int delId = sc.nextInt();

                    taskList.deleteTask(delId);

                    break;

                case 5:

                    System.out.print("Enter Task ID to search: ");

                    int searchId = sc.nextInt();

                    Task found = taskList.searchTask(searchId);

                    if (found != null)

                        System.out.println("Found: " + found.taskId + " " + found.description + " Completed: " + found.isDone);

                    else

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

                    break;

                case 6:

                    System.out.println("Exiting Task Manager...");

                    break;

                default:

                    System.out.println("Invalid choice.");

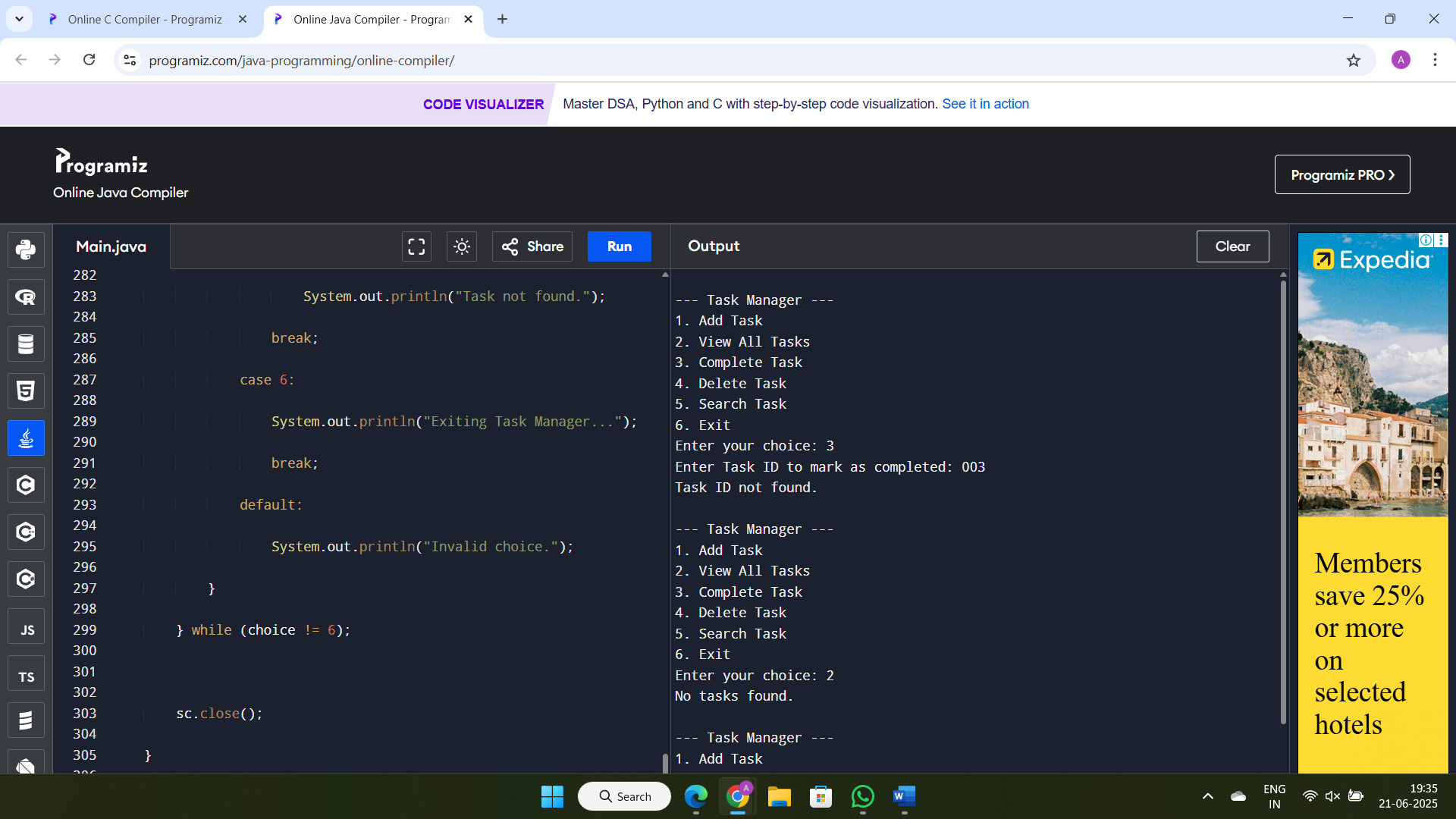
            }

        } while (choice != 6);

        sc.close();

    }

}

**OUTPUT:**

**ANALYSIS:**

**Time Complexitiy:**

o Add - O(n) at end, O(1) at head  
o Search - O(n)  
o Traverse - O(n)  
o Delete - O(n) in worst case

**Advantages:**

o Linked lists allow flexible memory usage and grow dynamically as needed.  
o Insertion and deletion in linked lists are efficient because no shifting of elements is required.  
o Arrays have a fixed size and may waste or run out of memory.  
o Arrays require shifting elements during insert/delete, which increases time complexity.  
o Linked lists are ideal when the number of elements changes frequently or is not known in advance.

**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.*

1. *Setup:*

* *Create a class Book with attributes like bookId, title, and author.*

1. *Implementation:*

* *Implement linear search to find books by title.*
* *Implement binary search to find books by title (assuming the list is sorted).*

1. *Analysis:*

* *Compare the time complexity of linear and binary search.*
* *Discuss when to use each algorithm based on the data set size and order.*

**UNDERSTAND SEARCH ALGORITHMS:**

o **Linear Search**:  
– Goes through each element in the list one by one.  
– It does not require the list to be sorted.  
– It is simple to implement and works well for small datasets.  
– Time complexity:  
 • Best case - O(1) (element at the beginning)  
 • Average case - O(n/2) ≈ O(n)  
 • Worst case - O(n) (element at the end or not present)

o **Binary Search**:  
– Works by repeatedly dividing the sorted list in half to find the target.  
– Requires the data to be sorted in advance.  
– Much faster for large datasets compared to linear search.  
– Time complexity:  
 • Best case - O(1) (element is at the middle)  
 • Average/Worst case - O(log n)  
– Not suitable for unsorted data unless sorting is done before searching.

**SOLUTION:**

import java.util.\*;

class Book {

    int bookId;

    String title;

    String author;

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

        this.bookId = bookId;

        this.title = title;

        this.author = author;

    } }

public class Main {

    static void addBook(List<Book> library, Book book) {

        library.add(book); }

    static void deleteBook(List<Book> library, int id) {

        library.removeIf(b -> b.bookId == id);}

    static void viewAllBooks(List<Book> library) {

        for (Book b : library) {

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

        } }

    static Book linearSearchByTitle(List<Book> library, String title) {

        for (Book b : library) {

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

                return b;}}

        return null;  }

    static Book binarySearchByTitle(List<Book> library, String title) {

        Collections.sort(library, Comparator.comparing(b -> b.title.toLowerCase()));

        int left = 0, right = library.size() - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

            String midTitle = library.get(mid).title.toLowerCase();

            int cmp = midTitle.compareTo(title.toLowerCase());

            if (cmp == 0) return library.get(mid);

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

            else right = mid - 1; }

        return null;}

    public static void main(String[] args) {

        List<Book> library = new ArrayList<>();

        addBook(library, new Book(1, "1984", "George Orwell"));

        addBook(library, new Book(2, "To Kill a Mockingbird", "Harper Lee"));

        addBook(library, new Book(3, "Brave New World", "Aldous Huxley"));

        addBook(library, new Book(4, "Fahrenheit 451", "Ray Bradbury"));

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

        viewAllBooks(library);

        System.out.println("\nLinear Search Result:");

        Book foundLinear = linearSearchByTitle(library, "Brave New World");

        if (foundLinear != null)

            System.out.println("Found: " + foundLinear.title + " by " + foundLinear.author);

        else

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

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

        Book foundBinary = binarySearchByTitle(library, "Brave New World");

        if (foundBinary != null)

            System.out.println("Found: " + foundBinary.title + " by " + foundBinary.author);

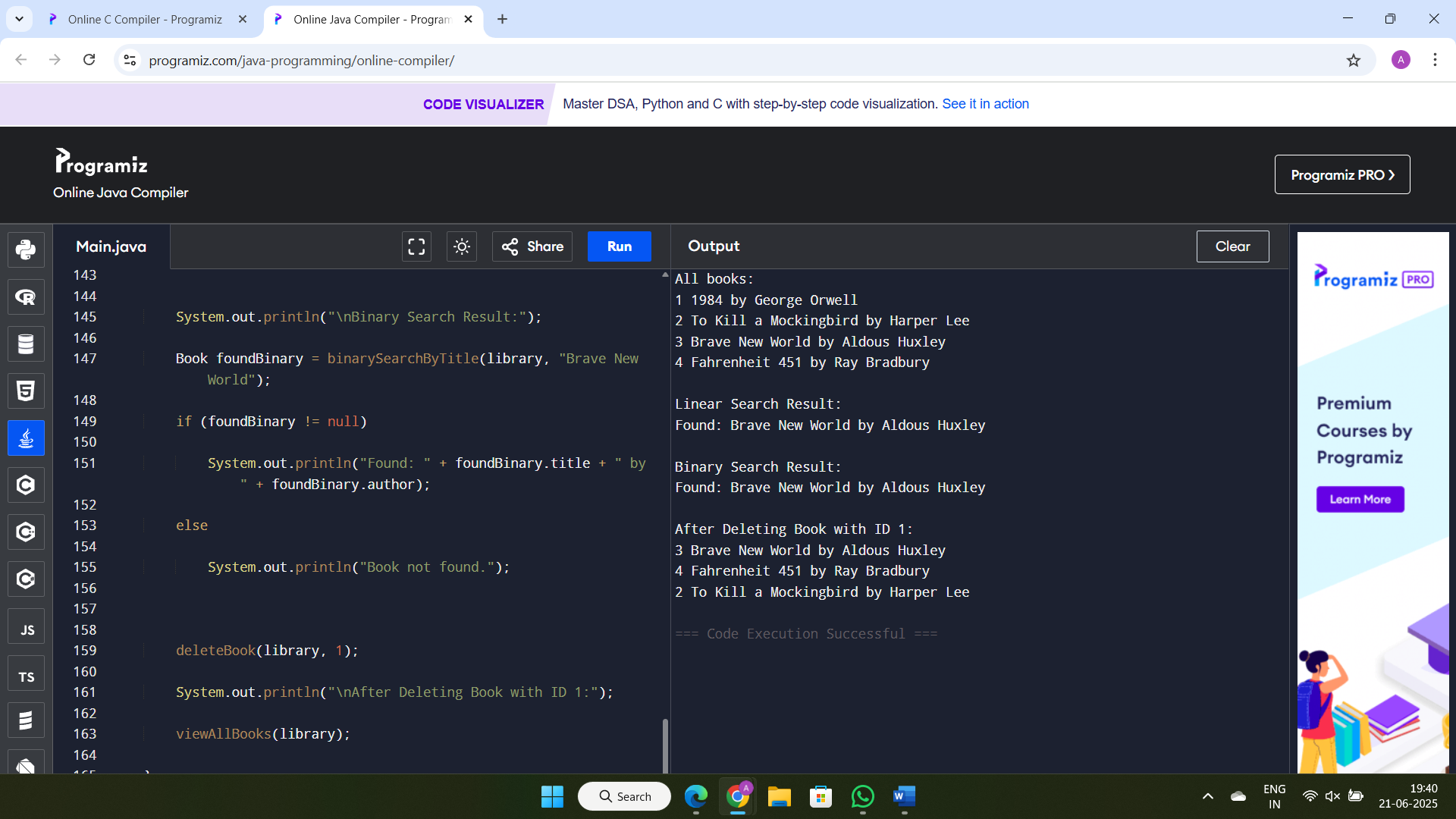
        else

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

        deleteBook(library, 1);

        System.out.println("\nAfter Deleting Book with ID 1:");

        viewAllBooks(library);}}

**OUTPUT:**

**ANALYSIS:**

| **Criteria** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Time Complexity | O(n) | O(log n) |
| Data Requirement | Unsorted or sorted | Sorted only |
| Best Case | O(1) | O(1) |
| Worst Case | O(n) | O(log n) |
| Simplicity | Very simple | Slightly complex |
| Suitable for | Small or unsorted data | Large and sorted data |

**o Use Linear Search:**  
– When the data set is small  
– When the data is unsorted  
– When implementation needs to be simple  
– When searches are rare or one-time

**o Use Binary Search:**  
– When the data set is large  
– When the data is already sorted  
– When fast search performance is needed  
– When multiple searches will be performed on the same data

**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.*

1. *Setup:*

* *Create a method to calculate the future value using a recursive approach.*

1. *Implementation:*

* *Implement a recursive algorithm to predict future values based on past growth rates.*

1. *Analysis:*

* *Discuss the time complexity of your recursive algorithm.*
* *Explain how to optimize the recursive solution to avoid excessive computation.*

**UNDERSTAND RECURSIVE ALGORITHMS:**

o Recursion is a technique where a method calls itself to solve smaller instances of a problem.  
o It helps break down complex problems into simpler sub-problems, making the solution easier to understand.  
o Recursion is ideal for problems like factorial, Fibonacci, tree traversal, and divide-and-conquer scenarios.  
o A recursive method must have a base condition to stop the calls and avoid infinite loops.

**SOLUTION:**

import java.util.\*;

class Transaction {

int transactionId;

double amount;

Transaction(int transactionId, double amount) {

this.transactionId = transactionId;

this.amount = amount;

} }

class Main {

static void addTransaction(List<Transaction> transactions, Transaction transaction) {

transactions.add(transaction);

}

static double total(List<Transaction> transactions) {

double sum = 0;

for (Transaction t : transactions) {

sum += t.amount;

}

return sum;

}

static void viewAll(List<Transaction> transactions) {

for (Transaction t : transactions) {

System.out.println(t.transactionId + " " + t.amount);

} }

static double forecast(double currentValue, double rate, int years) {

if (years == 0) return currentValue;

return forecast(currentValue \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

List<Transaction> transactions = new ArrayList<>();

addTransaction(transactions, new Transaction(1, 500.0));

addTransaction(transactions, new Transaction(2, -200.0));

viewAll(transactions);

double current = total(transactions);

System.out.println("Total: " + current);

double growthRate = 0.1; // 10%

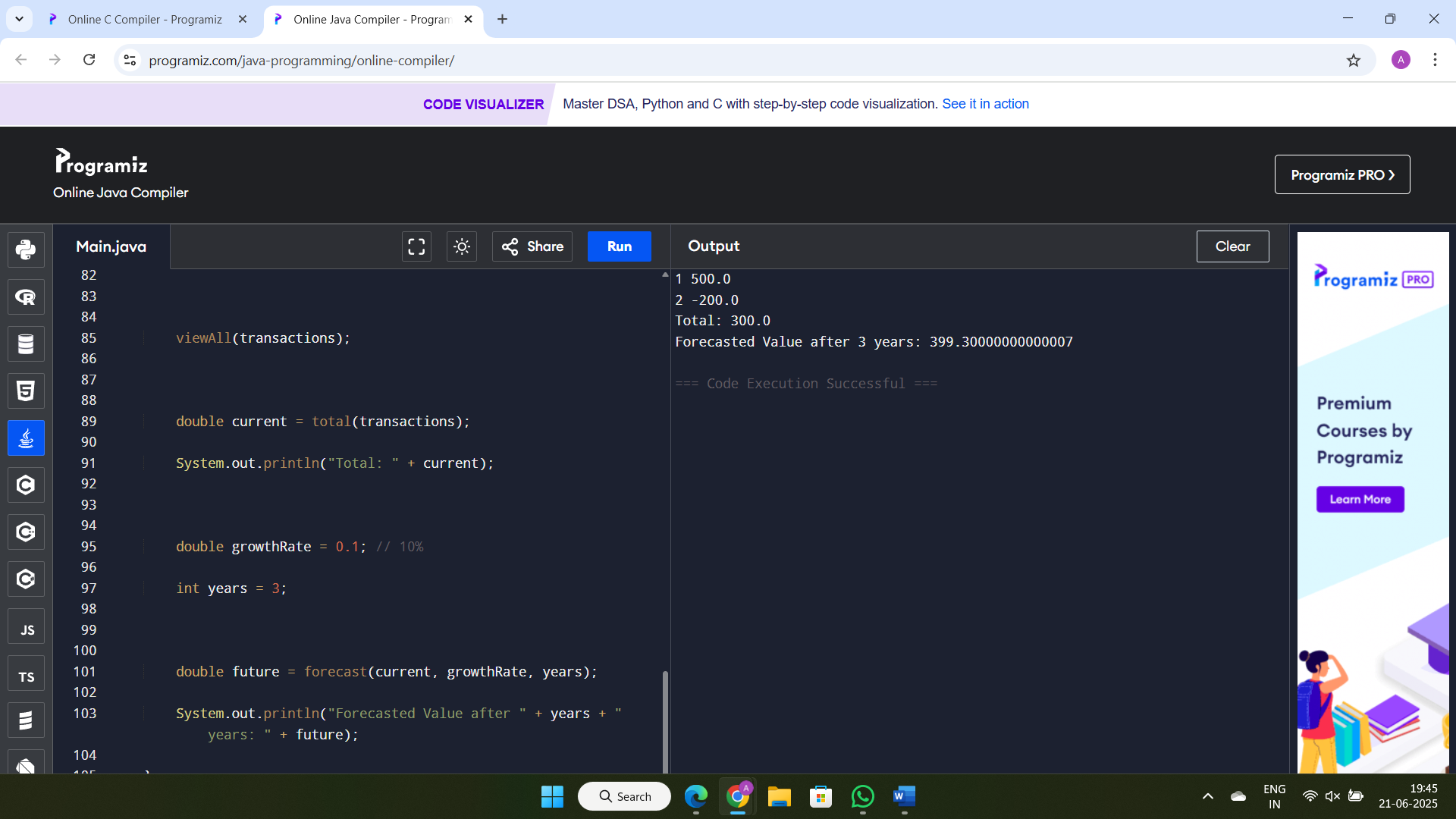
int years = 3;

double future = forecast(current, growthRate, years);

System.out.println("Forecasted Value after " + years + " years: " + future);

}

}

**OUTPUT:**

**ANALYSIS:**

o Time complexity - O(n), where n is the number of years (recursive calls).  
o Recursive calls are linear and each call processes one step.

**Optimization:**  
– For this problem, recursion is already efficient as it only involves one recursive call per step.  
– To optimize deeper recursion, use tail recursion if supported by the compiler or convert to iteration.  
– For complex recursive problems (like Fibonacci), use memorization to store and reuse computed values and avoid repeated calculations.