WEEK 1: DATA STRUCTURES AND ALGORITHMS

# Exercise 1: Inventory Management System

**Scenario:** To develop an inventory management system for a warehouse with efficient data storage and retrieval.

**Understand the Problem:**

**Why Data Structures and Algorithms are Essential in Handling Large Inventories**

Data structures and algorithms are crucial in managing large inventories for several reasons:

1. **Efficiency**: Proper data structures enable the efficient data storage, retrieval, and manipulation of data. Algorithms ensure that operations like searching, adding, updating, and deleting items are performed optimally.
2. **Scalability**: As inventory size grows, the efficient data structures and algorithms help us maintain performance and prevent slow process progress.
3. **Memory Management**: Optimized data structures help utilize memory effectively, avoiding wastage and ensuring that the system can handle large datasets.
4. **Complexity Management**: They help manage the inherent complexity of large datasets, making it easier to implement and maintain the system.

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

The suitable types of Data Structures for Inventory Management System are as follows:

1. **ArrayList**: Provides dynamic arrays that can grow as needed. Good for scenarios where the number of items is variable, but accessing and iterating through the list is frequent.
2. **HashMap**: Also known as Hash Table, provides efficient key-value pair storage. Excellent for quick lookups, additions, and deletions based on unique identifiers like product IDs, etc.
3. **Binary Search Tree (BST)**: Allows for sorted storage and efficient in-order traversal. Suitable for scenarios requiring ordered data.
4. **Linked List**: Useful for constant-time insertions and deletions. It provides linear-time access, which might be a drawback for large datasets.

Here, choosing the data structure HashMap would be more appropriate in case of time complexity and for also insertion, deletion, and updating.

**Setup:**

**Creation of product class**

public class Item {

private String itemId;

private String itemName;

private int quantity;

private double price;

//constructor

public Item(String itemId, String itemName, int quantity, double price) {

this.itemId = itemId;

this.itemName = itemName;

this.quantity = quantity;

this.price = price;

}

// Getters and setters for the item class attributes

public String getItemId() {

return itemId;

}

public void setItemId(String itemId) {

this.itemId = itemId;

}

public String getItemName() {

return itemName;

}

public void setItemName(String itemName) {

this.itemName = itemName;

}

public int getQuantity() {

return quantity;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

}

import java.util.HashMap;

public class InventoryManager {

private HashMap<String, Item> inventory;

public InventoryManager() {

inventory = new HashMap<>();

}

public void addItem(Item item) {

inventory.put(item.getItemId(), item);

}

//To Update an item in the inventory

public void updateItem(Item item) {

if (inventory.containsKey(item.getItemId())) {

inventory.put(item.getItemId(), item);

} else {

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

}

}

public void deleteItem(String itemId) {

if (inventory.containsKey(itemId)) {

inventory.remove(itemId);

} else {

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

}

}

//To Display all items in the inventory

public void displayItems() {

for (Item item : inventory.values()) {

System.out.println("Item ID: " + item.getItemId());

System.out.println("Item Name: " + item.getItemName());

System.out.println("Quantity: " + item.getQuantity());

System.out.println("Price: " + item.getPrice());

System.out.println();

}

}

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

// Adding items

Item item1 = new Item("I001", "Item\_no\_1", 50, 999.99);

Item item2 = new Item("I002", "Item\_no\_2", 30, 899.99);

Item item3 = new Item("I003", "Item\_no\_3", 100, 349.99);

Item item4 = new Item("I004", "Item\_no\_4", 25, 1249.99);

Item item5 = new Item("I005", "Item\_no\_5", 15, 2399.99);

manager.addItem(item1);

manager.addItem(item2);

manager.addItem(item3);

manager.addItem(item4);

manager.addItem(item5);

// To Display items

manager.displayItems();

// To Update an item

Item updatedItem1 = new Item("I001", "Item\_no\_1", 60, 949.99);

manager.updateItem(updatedItem1);

// To Display items after update

manager.displayItems();

// For Deleting an item

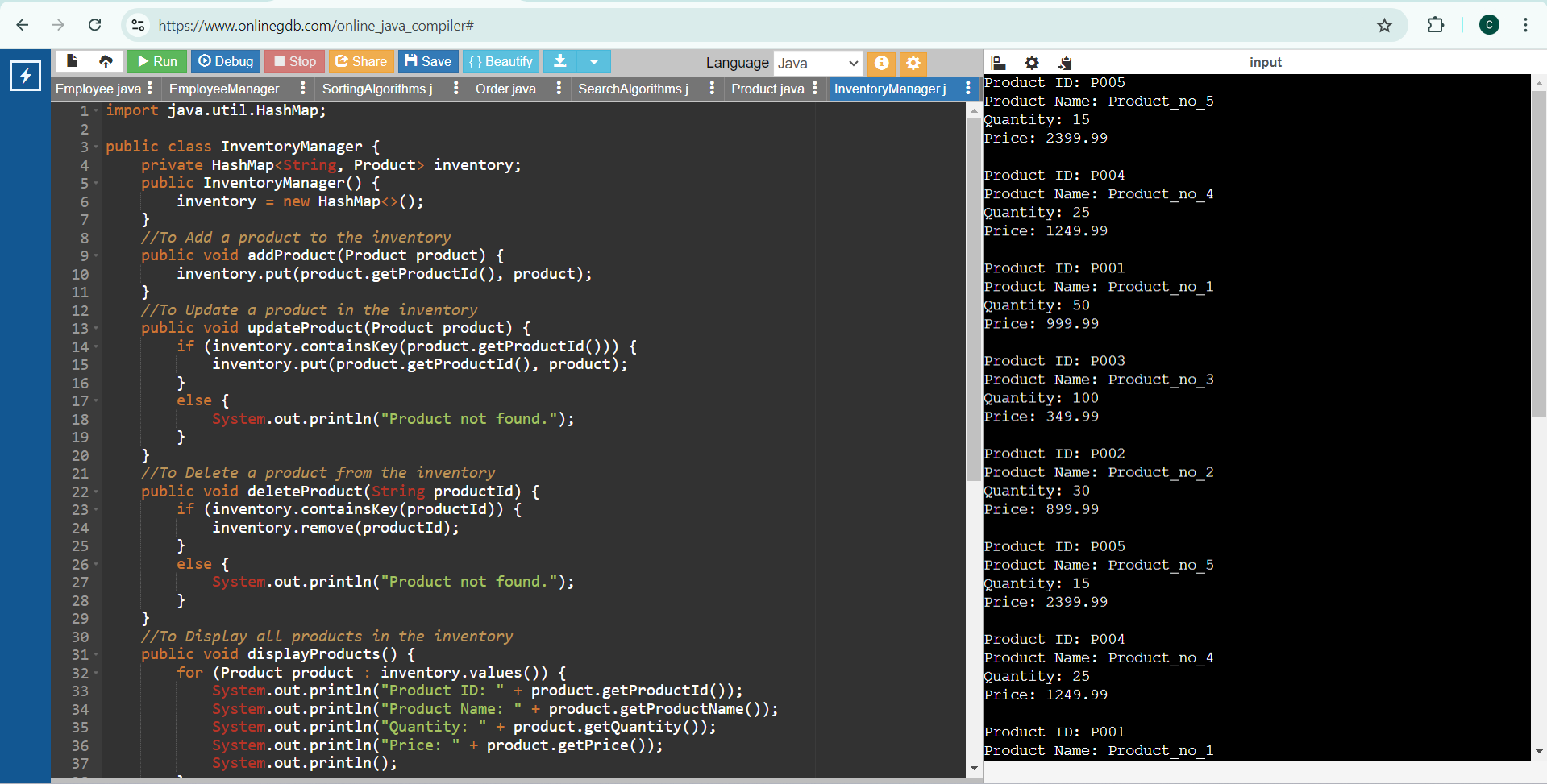
manager.deleteItem("I002");

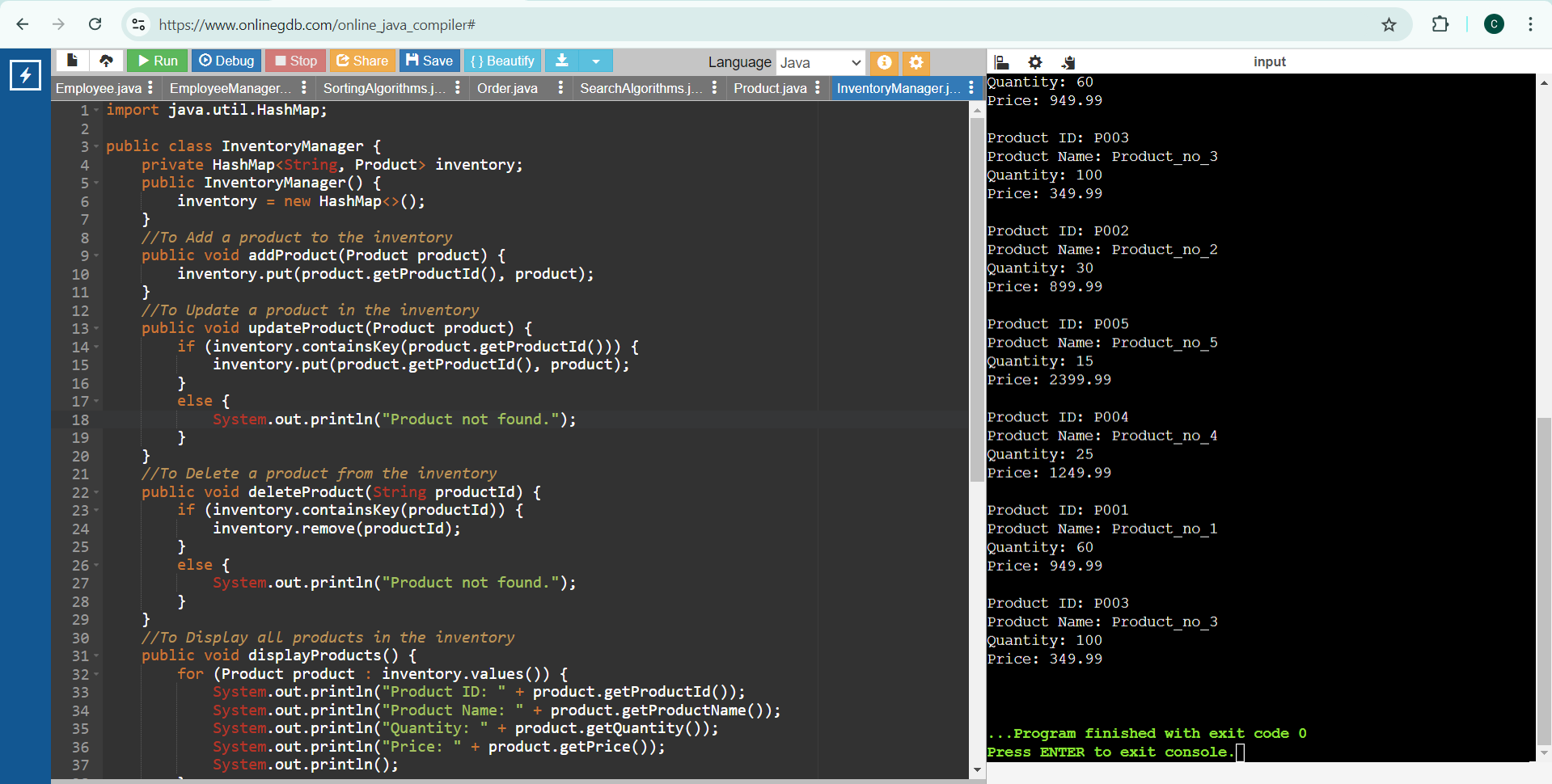
// To Display items after deletion

manager.displayItems();

}

}**Output:**

****

****

**Time Complexity Analysis:**

1. **Add Product:**

* Time Complexity: O(1)
  + Inserting a product into a HashMap is O(1) due to the constant time complexity of hash-based data structures.

1. **Update Product:**
   * Time Complexity: O(1)
   * Updating a product in a HashMap is also O(1) since it involves accessing the element by key and replacing the value.
2. **Delete Product:**
   * Time Complexity: O(1)
   * Removing a product from a HashMap is O(1) as it involves finding the element by key and deleting it.

**Optimization:**

To optimize the above inventory management system, we should ensure that the `HashMap` is sized properly to avoid frequent rehashing, and also adjust the load factor for a balance between performance and memory usage.

For concurrent access, we can also consider using `ConcurrentHashMap` to prevent the chances of occurrence of synchronization issues.

# Exercise 2: E-commerce Platform Search Function

**Scenario:** To work on the search functionality of an e-commerce platform with optimized performance.

**Understand Asymptotic Notation:**

**Explain Big O notation and how it helps in analyzing algorithms.**

Big O notation is a mathematical representation used to describe the upper bound of an algorithm's running time or space requirements in terms of the size of the input data. It also helps in analyzing the efficiency of algorithms by providing an approximation of the worst-case scenario in terms scales as the input size increases. This allows developers to predict performance and make informed decisions about which algorithms to use.

Big O Notation helps in analyzing the Algorithms

* **Performance Prediction**: Predicts how an algorithm scales with input size, guiding suitability for large datasets.
* **Algorithm Comparison**: Standardizes efficiency comparison, e.g., O(n log n) vs. O(n^2).
* **Scalability**: Assesses how well an algorithm handles larger inputs.
* **Optimization**: Guides code optimization by highlighting less efficient algorithms.
* **Worst-Case Analysis**: Ensures the system can handle the algorithm's maximum resource needs.

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

**Best, Average, and Worst-Case Scenarios for Search Operations**

1. **Best Case**: This is the best-case scenario where the search operation completes in the shortest possible time, usually when the desired element is at the beginning of the collection.
2. **Average Case**: The scenario of average case, represents a typical run where the position of the desired element is uniformly distributed in the list of elements.
3. **Worst Case**: The scenario of worst case is where the search operation takes the longest time, this is usually considered when the desired element is at the end of the collection or not present in the list at all.

**Setup:**

**Create a product class**

public class Product {

private String productId;

private String productName;

private String category;

//constructor

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

//Setter and Getter methods

public String getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

public override String toString() {

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

}

}

**Implementation:**

import java.util.Arrays;

import java.util.Comparator;

public class SearchAlgorithms {

// Linear Search

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

for (Product product : products) {

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

return product;

}

}

return null;

}

// Binary Search

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

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

int comparison = products[mid].getProductName().compareToIgnoreCase(productName);

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

new Product("P003", "Tablet", "Electronics"),

new Product("P004", "Monitor", "Electronics"),

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

};

// Linear search demonstration

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

String searchName = "Tablet";

Product foundProduct = SearchAlgorithms.linearSearch(products, searchName);

if (foundProduct != null) {

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

} else {

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

}

// Binary search demonstration

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

searchName = "Monitor";

foundProduct = SearchAlgorithms.binarySearch(products, searchName);

if (foundProduct != null) {

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

} else {

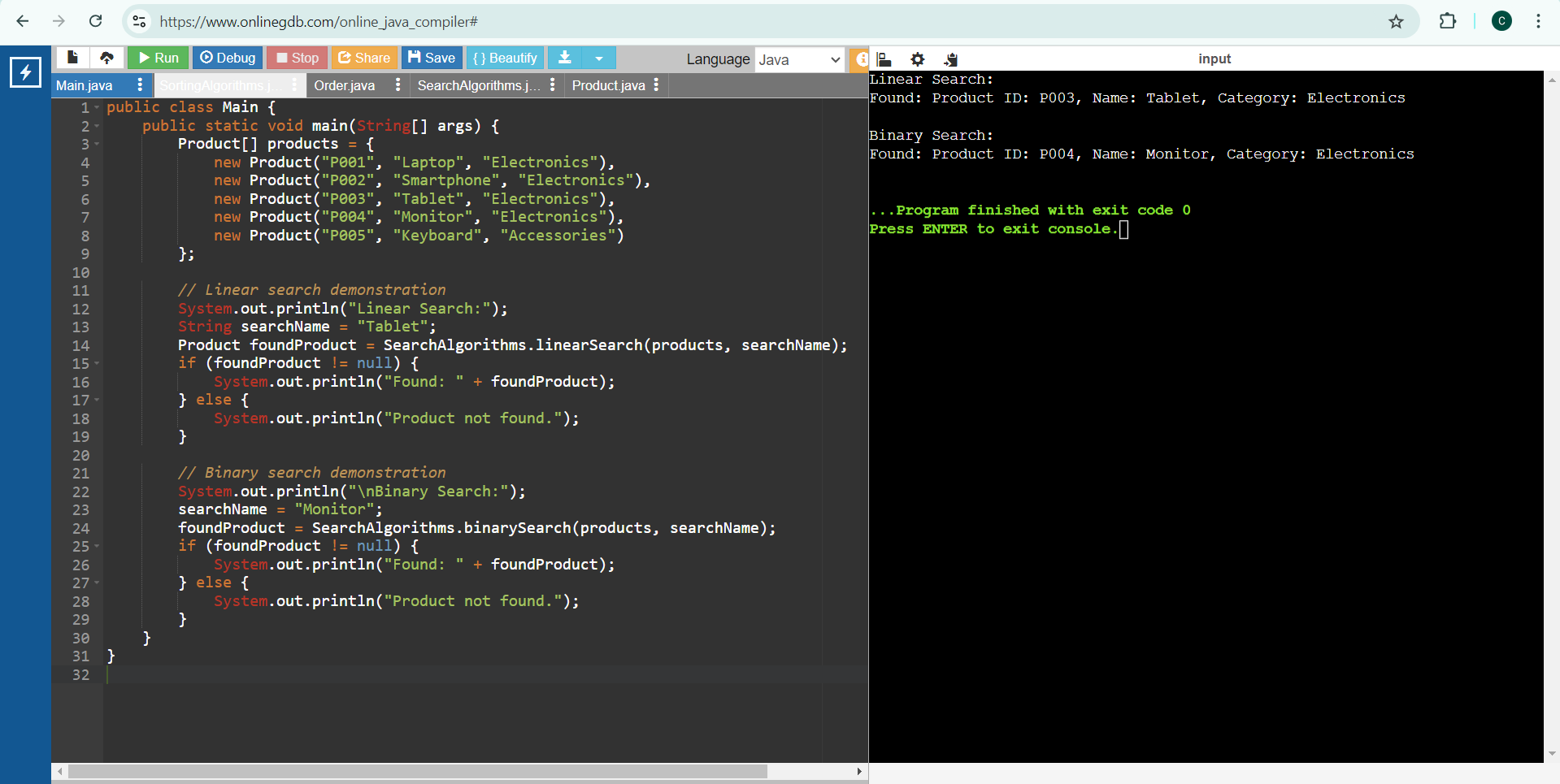
System.out.println("Product not found.");

}

}

}

**Output:**



**Time Complexity:**

**Linear Search:**

* Best Case: O(1) (when the element is at the beginning)
* Average Case: O(n)
* Worst Case: O(n)

**Binary Search:**

* Best Case: O(1) (when the element is at the middle)
* Average Case: O(log n)
* Worst Case: O(log n)

**Suitable Algorithm for this Platform:**

Binary search is more suitable for the e-commerce platform because it has a lower time complexity of O(log n) compared to linear search O(n) for large datasets. However, it requires the dataset to be sorted. If the dataset is not sorted or frequently updated, linear search might be simpler to implement initially but less efficient for larger datasets.

# Exercise 3: Sorting Customer Orders

**Scenario:** To sort customer orders by their total price on an e-commerce platform which helps in prioritizing high-value orders.

**Understand Sorting Algorithms:**

**Bubble Sort**

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

* **Time Complexity**: O(n^2) in the average and worst case, O(n) in the best case
* **Space Complexity**: O(1)
* **Stability**: Stable

**Insertion Sort**

Insertion Sort builds the final sorted array one item at a time. It takes each element from the input and inserts it into the correct position within the already sorted part of the array.

* **Time Complexity**: O(n^2) in the average and worst case, O(n) in the best case
* **Space Complexity**: O(1)
* **Stability**: Stable

**Quick Sort**

Quick Sort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

* **Time Complexity**: O(n log n) on average, O(n^2) in the worst case.
* **Space Complexity**: O(log n)
* **Stability**: Not stable

**Merge Sort**

Merge Sort is also a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts them, and then merges the two sorted halves.

* **Time Complexity**: O(n log n)
* **Space Complexity**: O(n)
* **Stability**: Stable

**Setup:**

**Create a class Order**

public class Purchase {

private String purchaseId;

private String clientName;

private double totalAmount;

public Purchase(String purchaseId, String clientName, double totalAmount) {

this.purchaseId = purchaseId;

this.clientName = clientName;

this.totalAmount = totalAmount;

}

public String getPurchaseId() {

return purchaseId;

}

public String getClientName() {

return clientName;

}

public double getTotalAmount() {

return totalAmount;

}

@Override

public String toString() {

return "Purchase ID: " + purchaseId + ", Client Name: " + clientName + ", Total Amount: $" + totalAmount;

}

}

public class SortingAlgorithms {

public static void bubbleSort(Purchase[] purchases) {

int n = purchases.length;

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

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

if (purchases[j].getTotalAmount() > purchases[j + 1].getTotalAmount()) {

// Swap purchases[j] and purchases[j + 1]

Purchase temp = purchases[j];

purchases[j] = purchases[j + 1];

purchases[j + 1] = temp;

}

}

}

}

public static void quickSort(Purchase[] purchases, int low, int high) {

if (low < high) {

int pi = partition(purchases, low, high);

quickSort(purchases, low, pi - 1);

quickSort(purchases, pi + 1, high);

}

}

private static int partition(Purchase[] purchases, int low, int high) {

double pivot = purchases[high].getTotalAmount();

int i = (low - 1);

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

if (purchases[j].getTotalAmount() <= pivot) {

i++;

// Swap purchases[i] and purchases[j]

Purchase temp = purchases[i];

purchases[i] = purchases[j];

purchases[j] = temp;

}

}

Purchase temp = purchases[i + 1];

purchases[i + 1] = purchases[high];

purchases[high] = temp;

return i + 1;

}

}

public class Main {

public static void main(String[] args) {

Purchase[] purchases = {

new Purchase("P001", "Alice", 250.50),

new Purchase("P002", "Bob", 150.75),

new Purchase("P003", "Charlie", 300.10),

new Purchase("P004", "David", 175.20),

new Purchase("P005", "Eve", 210.80)

};

System.out.println("Before Bubble Sort:");

printPurchases(purchases);

SortingAlgorithms.bubbleSort(purchases);

System.out.println("\nAfter Bubble Sort:");

printPurchases(purchases);

purchases = new Purchase[]{

new Purchase("P001", "Alice", 250.50),

new Purchase("P002", "Bob", 150.75),

new Purchase("P003", "Charlie", 300.10),

new Purchase("P004", "David", 175.20),

new Purchase("P005", "Eve", 210.80)

};

System.out.println("\nBefore Quick Sort:");

printPurchases(purchases);

SortingAlgorithms.quickSort(purchases, 0, purchases.length - 1);

System.out.println("\nAfter Quick Sort:");

printPurchases(purchases);

}

public static void printPurchases(Purchase[] purchases) {

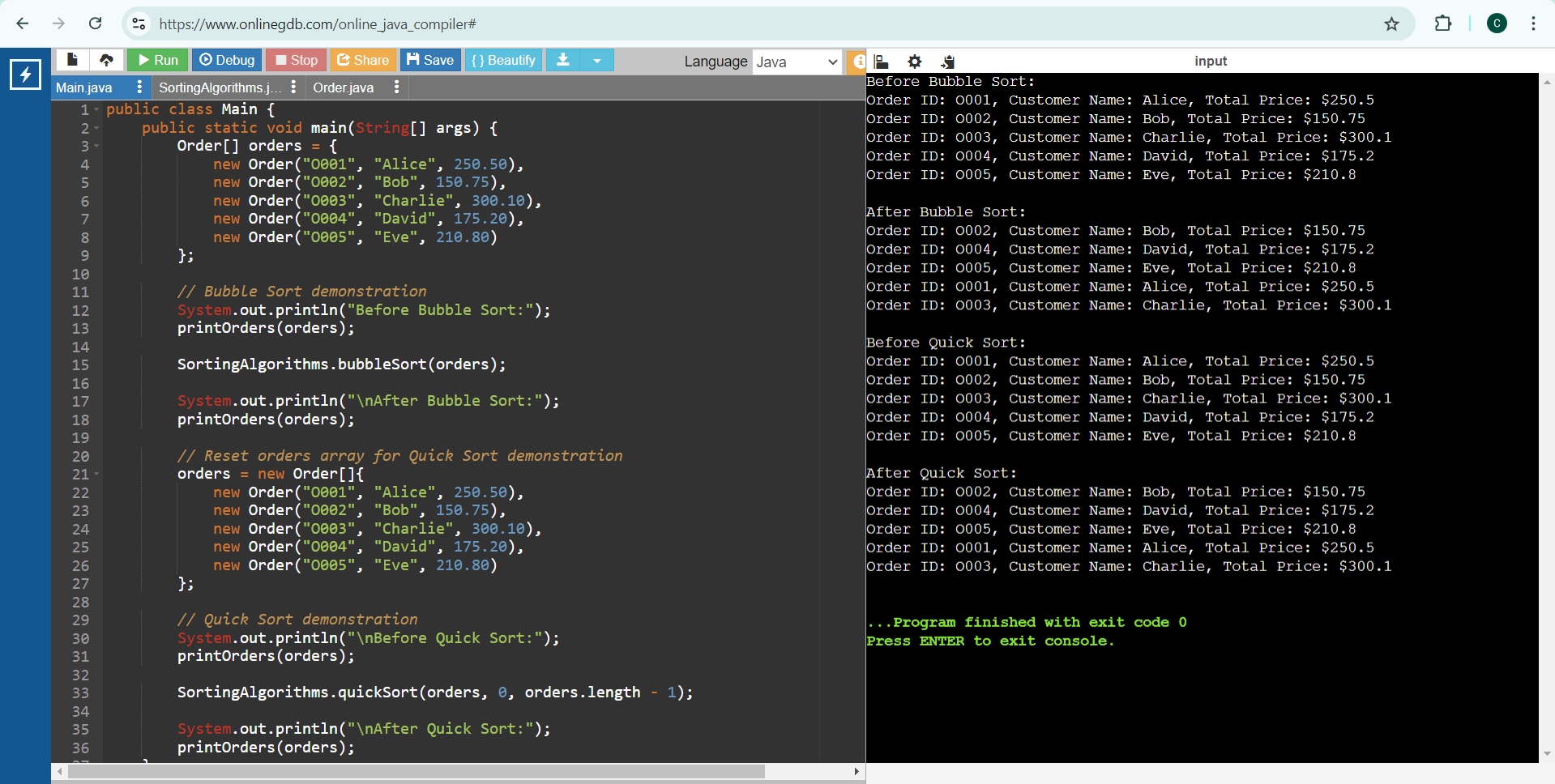
for (Purchase purchase : purchases) {

System.out.println(purchase);

}

}

}**Output:**

****

**Analysis**

**Time Complexity Comparison**

* Bubble Sort:
* Best Case: O(n)
* Average Case: O(n^2)
* Worst Case: O(n^2)
* Quick Sort:
* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n^2)

**Quick Sort is Preferred Over Bubble Sort**

Quick Sort is generally preferred over Bubble Sort because it has a much better average-case time complexity of O(n log n) compared to Bubble Sort's O(n^2).

Even though Quick Sort can degrade to O(n^2) in the worst case, this can be mitigated with good pivot selection strategies, such as choosing the median or using randomization.

Quick Sort also tends to have better cache performance and is more efficient in practice, making it more suitable for sorting large datasets on an e-commerce platform.

# Exercise 4: Employee Management System

**Scenario:** To develop an employee management system for a company and efficiently manage employee records.

**Understand Array Representation:**

**Arrays are Represented in Memory**

Arrays are a fundamental data structure in Java, where elements are stored in contiguous memory locations. This arrangement provides efficient indexing and quick access to elements.

* Contiguous Memory Allocation: Elements are stored in adjacent memory blocks, allowing direct access via an index.
* Fixed Size: The size of an array is defined at the time of its creation and cannot be changed.

**Advantages:**

* Fast Access: O(1) time complexity for accessing elements by index.
* Simplicity: Easy to use and understand, with a straightforward syntax.
* Memory Efficiency: Efficient memory usage due to contiguous allocation.

**Setup:**

**Create a class Employee**

public class Staff {

private int staffId;

private String fullName;

private String jobTitle;

private double annualSalary;

public Staff(int staffId, String fullName, String jobTitle, double annualSalary) {

this.staffId = staffId;

this.fullName = fullName;

this.jobTitle = jobTitle;

this.annualSalary = annualSalary;

}

public int getStaffId() {

return staffId;

}

public String getFullName() {

return fullName;

}

public String getJobTitle() {

return jobTitle;

}

public double getAnnualSalary() {

return annualSalary;

}

@Override

public String toString() {

return "Staff ID: " + staffId + ", Name: " + fullName + ", Job Title: " + jobTitle + ", Salary: $" + annualSalary;

}

}

public class StaffManager {

private Staff[] staffList;

private int count;

public StaffManager(int capacity) {

staffList = new Staff[capacity];

count = 0;

}

public void addStaff(Staff staff) {

if (count < staffList.length) {

staffList[count++] = staff;

} else {

System.out.println("Array is full. Cannot add more staff.");

}

}

public Staff searchStaffById(int staffId) {

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

if (staffList[i].getStaffId() == staffId) {

return staffList[i];

}

}

return null;

}

public void traverseStaff() {

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

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

}

}

public boolean deleteStaffById(int staffId) {

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

if (staffList[i].getStaffId() == staffId) {

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

staffList[j] = staffList[j + 1];

}

staffList[--count] = null;

return true;

}

}

return false;

}

public static void main(String[] args) {

StaffManager manager = new StaffManager(10);

manager.addStaff(new Staff(1, "Alice", "Developer", 70000));

manager.addStaff(new Staff(2, "Bob", "Manager", 85000));

manager.addStaff(new Staff(3, "Charlie", "Analyst", 60000));

manager.addStaff(new Staff(4, "David", "Designer", 65000));

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

manager.traverseStaff();

System.out.println("\nSearching for staff with ID 3:");

Staff staff = manager.searchStaffById(3);

if (staff != null) {

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

} else {

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

}

System.out.println("\nDeleting staff with ID 2:");

boolean isDeleted = manager.deleteStaffById(2);

System.out.println("Deleted: " + isDeleted);

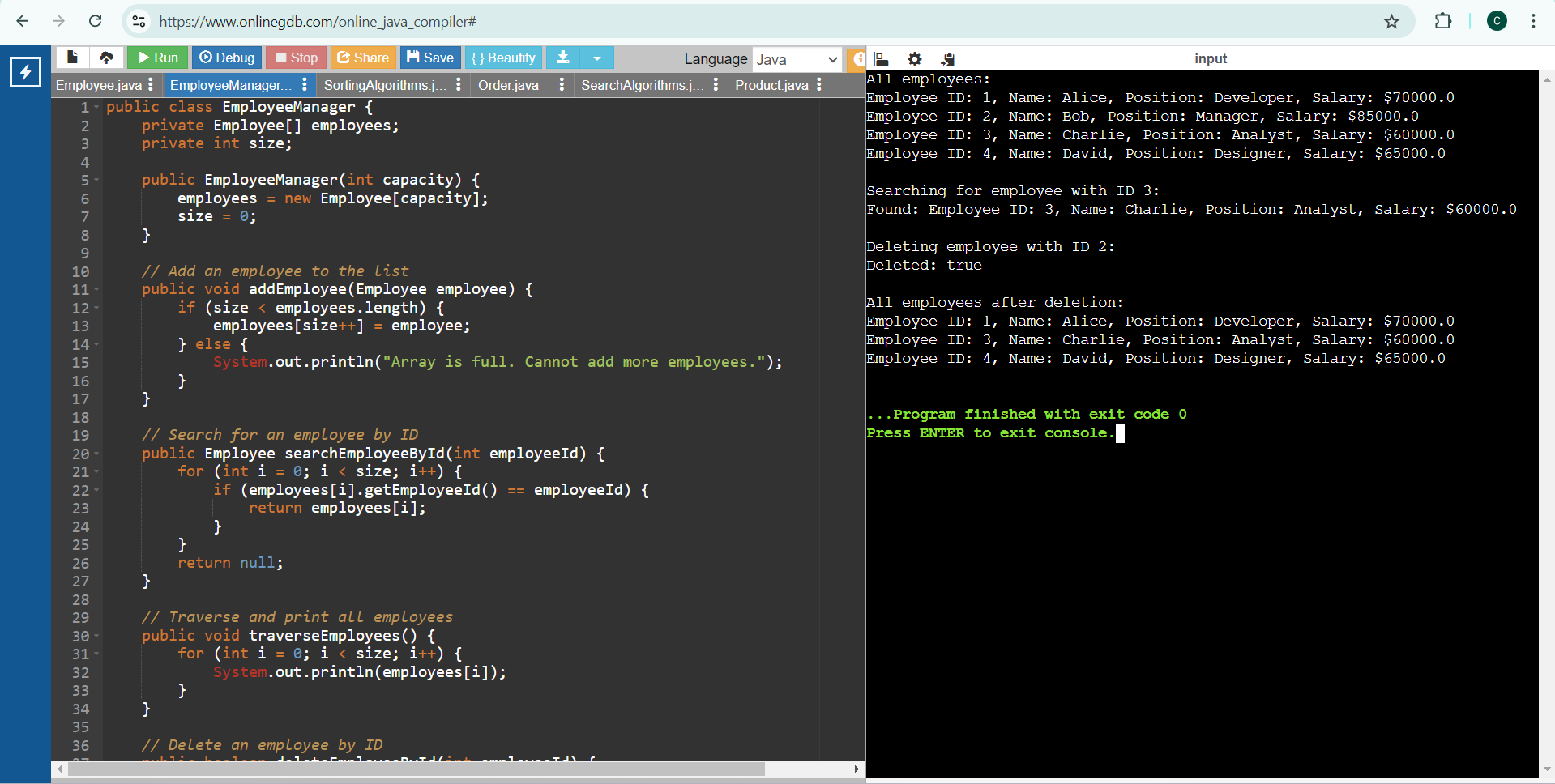
System.out.println("\nAll staff after deletion:");

manager.traverseStaff();

}

}

**Output:**

****

**Analysis:**

**Time Complexity of Operations**

* Add Employee:
* Time Complexity: O(1)
* Search Employee by ID:
* Time Complexity: O(n)
* Traverse Employees:
  + Time Complexity: O(n)
* Delete Employee by ID:
  + Time Complexity: O(n)

**Limitations of Arrays and When to Use Them**

* **Fixed Size**: Arrays have a fixed size, making them unsuitable when the number of elements is unknown or changes frequently.
* **Inefficient for Frequent Insertions/Deletions:** Operations like insertion and deletion are costly (O(n)) compared to dynamic data structures like ArrayList or LinkedList.
* **When to Use Arrays:**
* When the number of elements is known and fixed.
* For applications requiring fast access to elements by index.
* When memory efficiency and performance of access are critical.

# Exercise 5: Task Management System

**Scenario:** To develop a task management system where tasks need to be added, deleted, and traversed efficiently.

**Understand Linked Lists**

Linked lists offer better management of dynamic data due to their flexible size and efficient insertions/deletions, despite having a higher time complexity for search operations compared to arrays.

**Types of Linked Lists;**

* **Singly Linked List:**
* **Structure**: Each node contains data and a reference to the next node.
* **Traversal:** Can only traverse in one direction (forward).
* **Advantages:** Simple implementation, uses less memory compared to doubly linked lists.
* **Doubly Linked List:**
* **Structure:** Each node contains data, a reference to the next node, and a reference to the previous node.
* **Traversal:** Can traverse in both directions (forward and backward).
* **Advantages:** Easier to implement certain operations (like deletion) and more flexible traversal.

**Setup**

**Create a class Task**

public class Assignment {

private int assignmentId;

private String assignmentName;

private String status;

public Assignment(int assignmentId, String assignmentName, String status) {

this.assignmentId = assignmentId;

this.assignmentName = assignmentName;

this.status = status;

}

public int getAssignmentId() {

return assignmentId;

}

public String getAssignmentName() {

return assignmentName;

}

public String getStatus() {

return status;

}

@Override

public String toString() {

return "Assignment ID: " + assignmentId + ", Assignment Name: " + assignmentName + ", Status: " + status;

}

}

public class Node {

Assignment assignment;

Node next;

public Node(Assignment assignment) {

this.assignment = assignment;

this.next = null;

}

}

public class AssignmentLinkedList {

private Node head;

public AssignmentLinkedList() {

this.head = null;

}

public void addAssignment(Assignment assignment) {

Node newNode = new Node(assignment);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

public Assignment searchAssignmentById(int assignmentId) {

Node current = head;

while (current != null) {

if (current.assignment.getAssignmentId() == assignmentId) {

return current.assignment;

}

current = current.next;

}

return null;

}

public void traverseAssignments() {

Node current = head;

while (current != null) {

System.out.println(current.assignment);

current = current.next;

}

}

public boolean deleteAssignmentById(int assignmentId) {

if (head == null) return false;

if (head.assignment.getAssignmentId() == assignmentId) {

head = head.next;

return true;

}

Node current = head;

while (current.next != null && current.next.assignment.getAssignmentId() != assignmentId) {

current = current.next;

}

if (current.next == null) return false;

current.next = current.next.next;

return true;

}

}

public class Main {

public static void main(String[] args) {

AssignmentLinkedList assignmentList = new AssignmentLinkedList();

assignmentList.addAssignment(new Assignment(1, "Design UI", "In Progress"));

assignmentList.addAssignment(new Assignment(2, "Develop Backend", "Not Started"));

assignmentList.addAssignment(new Assignment(3, "Write Tests", "Not Started"));

assignmentList.addAssignment(new Assignment(4, "Deploy Application", "Completed"));

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

assignmentList.traverseAssignments();

System.out.println("\nSearching for assignment with ID 3:");

Assignment assignment = assignmentList.searchAssignmentById(3);

if (assignment != null) {

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

} else {

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

}

System.out.println("\nDeleting assignment with ID 2:");

boolean isDeleted = assignmentList.deleteAssignmentById(2);

System.out.println("Deleted: " + isDeleted);

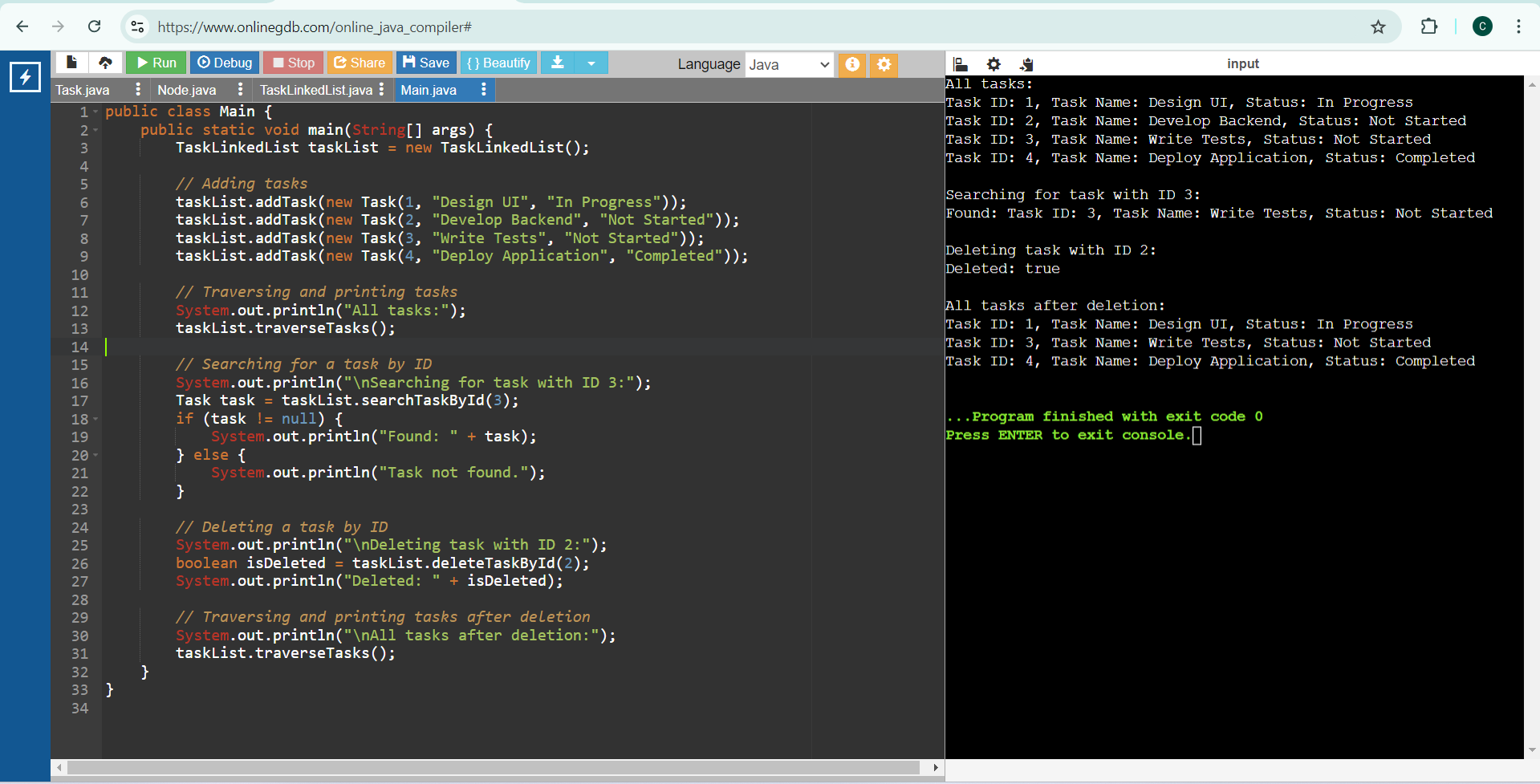
System.out.println("\nAll assignments after deletion:");

assignmentList.traverseAssignments();

}

}

**Output:**

****

**Analysis:**

**Time Complexity of Operations:**

* + **Add Task:**
  + Time Complexity: O(n)
  + **Search Task by ID:**
    - Time Complexity: O(n)
  + **Traverse Tasks:**
    - Time Complexity: O(n)
  + **Delete Task by ID:**
* Time Complexity: O(n)

**Advantages of Linked Lists Over Arrays for Dynamic Data**

* Dynamic Size: Linked lists can grow and shrink dynamically, unlike arrays that have a fixed size.
* Efficient Insertions/Deletions: Insertions and deletions can be more efficient (O(1)) if the position is known, as there's no need to shift elements.
* Memory Usage: Linked lists use memory more efficiently for dynamic data as they allocate memory as needed, whereas arrays may allocate more memory than necessary.
* Flexibility: Linked lists provide more flexibility with dynamic data structures, making them more suitable for tasks where the size of the dataset changes frequently.

# Exercise 6: Library Management System

**Scenario:** To develop a library management system where users can search for books by title or author.

**Understand Search Algorithms**

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

**Linear Search:**

Linear search is a simple search algorithm that checks every element in the list sequentially until the desired element is found or the list ends.

* Time Complexity: O(n)
* Space Complexity: O(1)
* Best Case: O(1) (if the element is at the beginning)
* Worst Case: O(n) (if the element is at the end or not present)
* Use Case: Suitable for unsorted or small lists.

**Binary Search:**

Binary search is a more efficient search algorithm for sorted lists. It repeatedly divides the search interval in half, comparing the middle element with the target value.

* Time Complexity: O(log n)
* Space Complexity: O(1)
* Best Case: O(1) (if the middle element is the target)
* Worst Case: O(log n) (if the element is not present)
* Use Case: Suitable for large, sorted lists.

Linear search is straightforward and suitable for small or unsorted datasets, while binary search is more efficient for larger, sorted datasets due to its significantly lower time complexity.

**Setup & Implementation:**

**Create a class Book**

public class Book {

private int bookId;

private String title;

private String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

public int getBookId() {

return bookId;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

@Override

public String toString() {

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

}

}**Output:**

****

**Analysis**

**Time Complexity of Search Algorithms**

**Linear Search:**

* Best Case: O(1)
* Average Case: O(n)
* Worst Case: O(n)
* Space Complexity: O(1)

**Binary Search:**

* Best Case: O(1)
* Average Case: O(log n)
* Worst Case: O(log n)
* Space Complexity: O(1)

**When to Use Each Algorithm**

**Linear Search:**

* Use for unsorted or small datasets.
* Simple to implement and does not require sorting.
* Efficient for cases where the dataset size is small or the target element is frequently near the beginning.

**Binary Search:**

* Use for large, sorted datasets.
* Much more efficient for large datasets due to its O(log n) time complexity.
* Requires the list to be sorted, adding an additional step if the data is not already sorted.

# Exercise 7: Financial Forecasting

**Scenario:** To develop a financial forecasting tool that predicts future values based on past data.

**Understand Recursive Algorithms**

**Concept of Recursion**

Recursion is a technique where a function calls itself to solve smaller instances of the same problem. It can simplify complex problems by breaking them down into more manageable subproblems.

* **Base Case**: The condition under which the recursion stops.
* **Recursive Case**: The part of the function where it calls itself with a smaller or simpler input.
* **Advantages**:
  + Simplifies code for problems that have repetitive structures.
  + Often more intuitive for problems like tree traversal, factorial calculation, etc.
* **Disadvantages**:
  + Can lead to excessive memory use due to function call stack.
  + Potential for stack overflow if not properly controlled.

**Setup**

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

public class FinancialForecasting {

// Recursive method to calculate future value

public static double predictFutureValue(double presentValue, double growthRate, int periods) {

// Base case: if no periods left, return present value

if (periods == 0) {

return presentValue;

}

// Recursive case: apply growth rate to present value and reduce the period

return predictFutureValue(presentValue \* (1 + growthRate), growthRate, periods - 1);

}

**Implementation**

**Recursive Algorithm to Predict Future Values**

public class FinancialForecasting {

public static double calculateFutureAmount(double initialAmount, double rateOfGrowth, int numberOfPeriods) {

if (numberOfPeriods == 0) {

return initialAmount;

}

return calculateFutureAmount(initialAmount \* (1 + rateOfGrowth), rateOfGrowth, numberOfPeriods - 1);

}

public static void main(String[] args) {

double initialAmount = 1000.0;

double rateOfGrowth = 0.05; // 5% growth rate

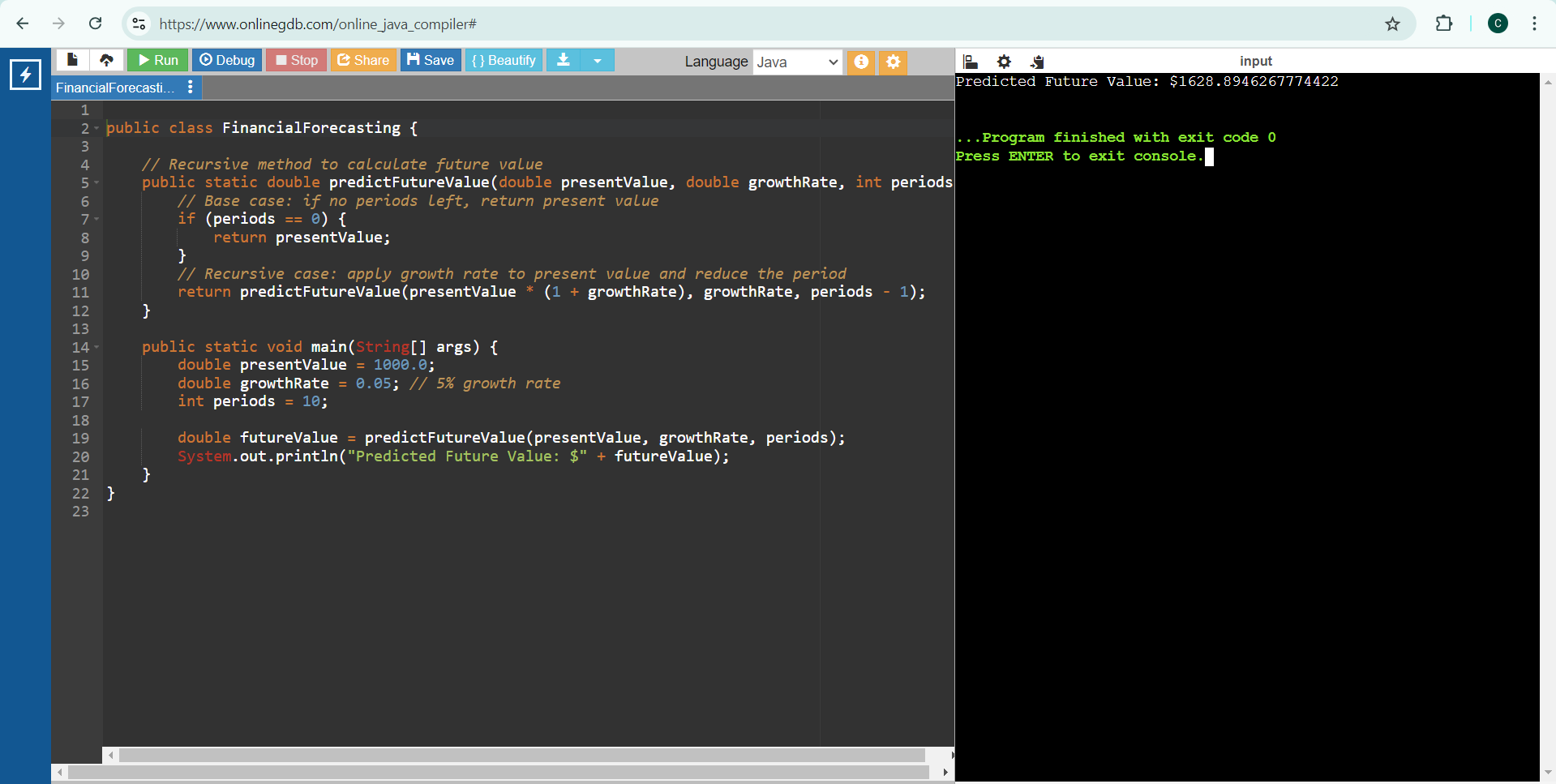
int numberOfPeriods = 10;

double futureAmount = calculateFutureAmount(initialAmount, rateOfGrowth, numberOfPeriods);

System.out.println("Predicted Future Amount: $" + futureAmount);

}

}**Output:**



**Analysis**

**Time Complexity of Recursive Algorithm**

* **Time Complexity**: O(n), where n is the number of periods.
  + Each call to the function handles one period, leading to n recursive calls.
* **Space Complexity**: O(n), due to the function call stack. Each recursive call adds a new frame to the stack.

**Optimizing the Recursive Solution**

To avoid excessive computation and potential stack overflow, we can use memoization to store previously computed results. However, in this simple growth rate model, memoization is not necessary as each step only depends on the previous step and does not repeat subproblems.