**Exercise 1: Inventory Management System**

**Scenario:**

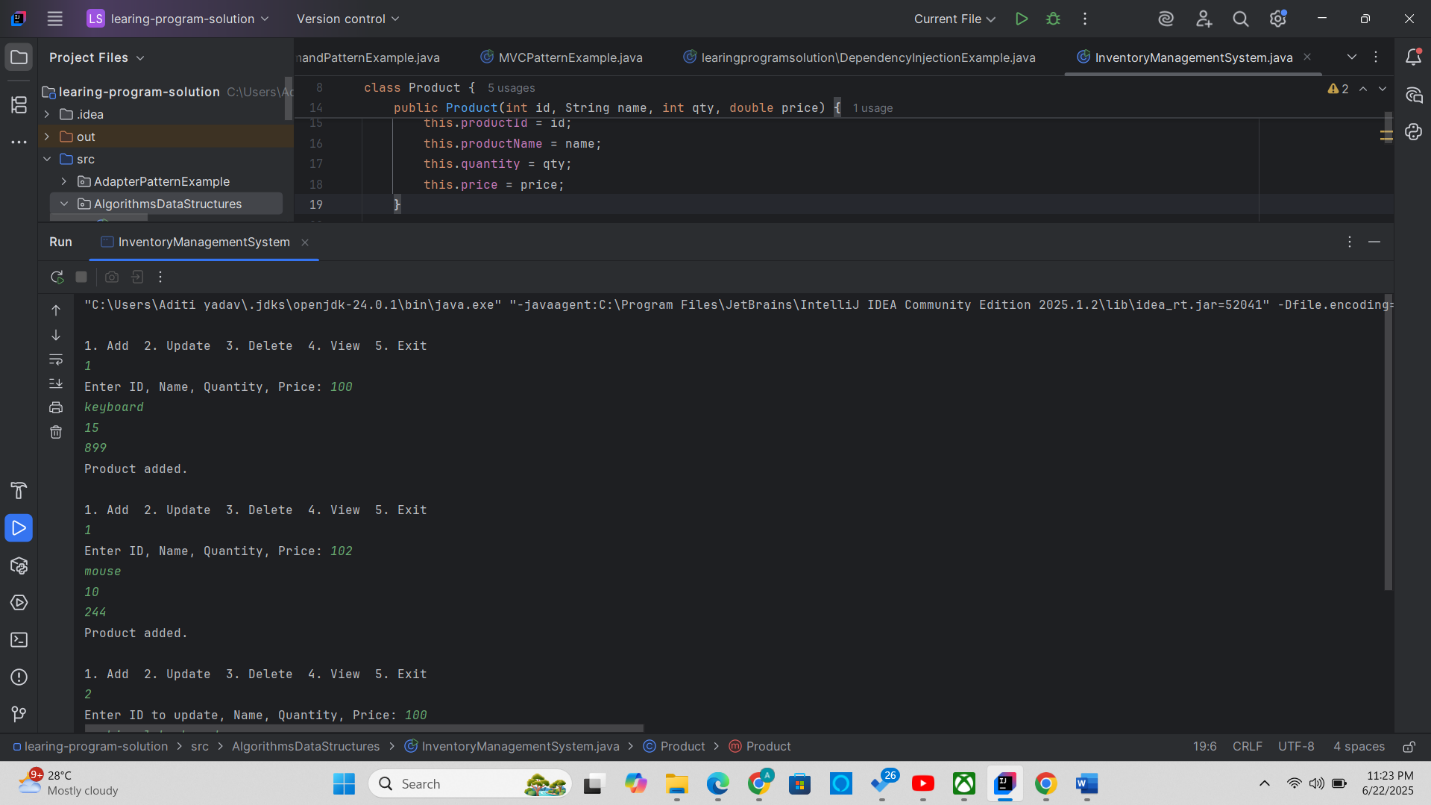
You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**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.

**Solution:**

package AlgorithmsDataStructures;  
  
import java.util.HashMap;  
import java.util.Map;  
import java.util.Scanner;  
  
public class InventoryManagementSystem {  
 private Map<Integer, Product> inventory;  
  
 public InventoryManagementSystem() {  
 inventory = new HashMap<>();  
 }  
  
 public void addProduct(Product p) {  
 inventory.put(p.productId, p);  
 System.*out*.println("Product added.");  
 }  
  
 public void updateProduct(int id, String name, int qty, double price, String category) {  
 if (inventory.containsKey(id)) {  
 Product p = inventory.get(id);  
 p.productName = name;  
 p.quantity = qty;  
 p.price = price;  
 p.category = category;  
 System.*out*.println("Product updated.");  
 } else {  
 System.*out*.println(“Product not found.");  
 }  
 }  
  
 public void deleteProduct(int id) {  
 if (inventory.remove(id) != null) {  
 System.*out*.println("Product deleted.");  
 } else {  
 System.*out*.println("❌ Product not found.");  
 }  
 }  
  
 public void displayInventory() {  
 if (inventory.isEmpty()) {  
 System.*out*.println("Inventory is empty.");  
 } else {  
 for (Product p : inventory.values()) {  
 System.*out*.println(p);  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 InventoryManagementSystem ims = new InventoryManagementSystem();  
 Scanner sc = new Scanner(System.*in*);  
  
 while (true) {  
 System.*out*.println("\n1. Add 2. Update 3. Delete 4. View 5. Exit");  
 System.*out*.print("Enter your choice: ");  
 int choice = sc.nextInt();  
  
 switch (choice) {  
 case 1 -> {  
 System.*out*.print("Enter ID, Name, Quantity, Price, Category: ");  
 int id = sc.nextInt();  
 String name = sc.next();  
 int qty = sc.nextInt();  
 double price = sc.nextDouble();  
 String category = sc.next();  
 ims.addProduct(new Product(id, name, qty, price, category));  
 }  
 case 2 -> {  
 System.*out*.print("Enter ID to update, New Name, Quantity, Price, Category: ");  
 int id = sc.nextInt();  
 String name = sc.next();  
 int qty = sc.nextInt();  
 double price = sc.nextDouble();  
 String category = sc.next();  
 ims.updateProduct(id, name, qty, price, category);  
 }  
 case 3 -> {  
 System.*out*.print("Enter ID to delete: ");  
 ims.deleteProduct(sc.nextInt());  
 }  
 case 4 -> ims.displayInventory();  
 case 5 -> {  
 System.*out*.println("Exiting.");  
 sc.close();  
 return;  
 }  
 default -> System.*out*.println("⚠️ Invalid choice.");  
 }  
 }  
 }  
}

****

|  |
| --- |
|  |

|  | **HashMap** |
| --- | --- |
| **Search by ID** | **O(1) – Direct access via key** |
| **Add Product** | **O(1)** |
| **Update Product** | **O(1)** |
| **Delete Product** | **O(1)** |

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

**Scenario:**

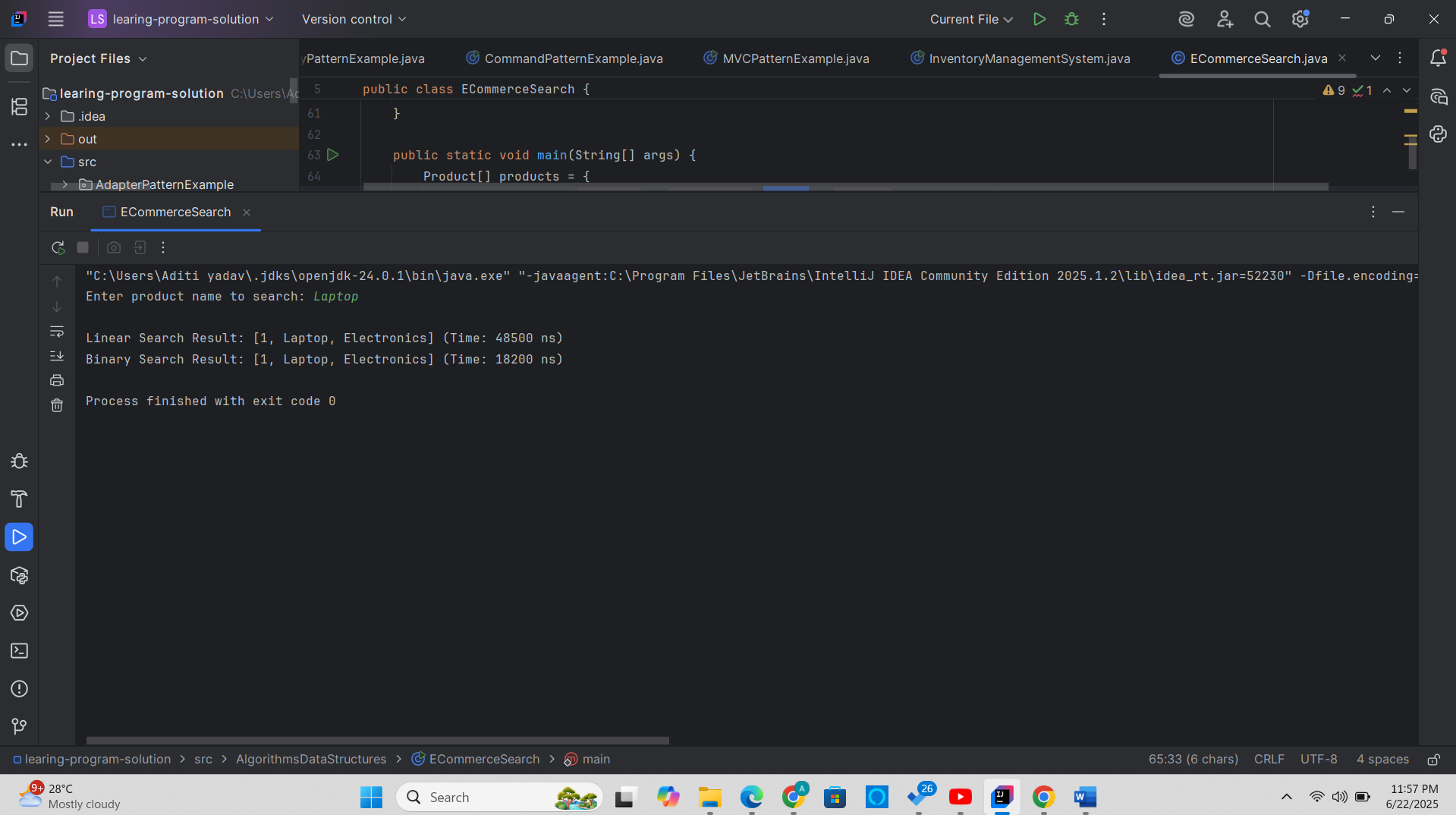
You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**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.

**Solution:**

package AlgorithmsDataStructures;  
  
import java.util.\*;  
  
public class ECommerceSearch {  
  
  
 static class Product {  
 private int productId;  
 private String productName;  
 private String category;  
  
 public Product(int productId, String productName, String category) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 }  
  
 public int getProductId() {  
 return productId;  
 }  
  
 public String getProductName() {  
 return productName;  
 }  
  
 public String getCategory() {  
 return category;  
 }  
  
 @Override  
 public String toString() {  
 return "[" + productId + ", " + productName + ", " + category + "]";  
 }  
 }  
  
  
 public static Product linearSearch(Product[] products, String targetName) {  
 for (Product p : products) {  
 if (p.getProductName().equalsIgnoreCase(targetName)) {  
 return p;  
 }  
 }  
 return null;  
 }  
  
 public static Product binarySearch(Product[] products, String targetName) {  
 int low = 0, high = products.length - 1;  
 while (low <= high) {  
 int mid = low + (high - low) / 2;  
 int cmp = products[mid].getProductName().compareToIgnoreCase(targetName);  
 if (cmp == 0)  
 return products[mid];  
 else if (cmp < 0)  
 low = mid + 1;  
 else  
 high = mid - 1;  
 }  
 return null;  
 }  
  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(1, "Laptop", "Electronics"),  
 new Product(2, "Shirt", "Apparel"),  
 new Product(3, "Watch", "Accessories"),  
 new Product(4, "Phone", "Electronics"),  
 new Product(5, "Shoes", "Footwear")  
 };  
  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.print("Enter product name to search: ");  
 String query = sc.nextLine();  
  
 long start = System.*nanoTime*();  
 Product linearResult = *linearSearch*(products, query);  
 long linearTime = System.*nanoTime*() - start;  
  
  
 Arrays.*sort*(products, Comparator.*comparing*(Product::getProductName));  
  
 start = System.*nanoTime*();  
 Product binaryResult = *binarySearch*(products, query);  
 long binaryTime = System.*nanoTime*() - start;  
  
 System.*out*.println("\nLinear Search Result: " + (linearResult != null ? linearResult : "Not Found") + " (Time: " + linearTime + " ns)");  
 System.*out*.println("Binary Search Result: " + (binaryResult != null ? binaryResult : "Not Found") + " (Time: " + binaryTime + " ns)");  
  
 sc.close();  
 }  
}



**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.

**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.

**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.

**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.

**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.

package AlgorithmsDataStructures;  
  
public class FinancialForecasting {  
  
 public static double forecast(double currentValue, double growthRate, int years) {  
 if (years == 0) {  
 return currentValue;  
 }  
 return *forecast*(currentValue, growthRate, years - 1) \* (1 + growthRate);  
 }  
  
   
 public static double forecastMemo(double currentValue, double growthRate, int years, double[] memo) {  
 if (years == 0) return currentValue;  
 if (memo[years] != 0) return memo[years];  
 memo[years] = *forecastMemo*(currentValue, growthRate, years - 1, memo) \* (1 + growthRate);  
 return memo[years];  
 }  
  
 public static void main(String[] args) {  
 double currentValue = 10000.0;   
 double growthRate = 0.08;   
 int years = 5;   
  
 System.*out*.println("Recursive Forecast:");  
 double futureValue = *forecast*(currentValue, growthRate, years);  
 System.*out*.printf("Future value after %d years: %.2f\n", years, futureValue);  
  
 System.*out*.println("\nMemoized Recursive Forecast:");  
 double[] memo = new double[years + 1];  
 double futureMemo = *forecastMemo*(currentValue, growthRate, years, memo);  
 System.*out*.printf("Future value after %d years (memoized): %.2f\n", years, futureMemo);  
 }  
}

