**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.

Main.java

import java.util.\*;

public class Main {

    private static InventoryManager inventoryManager;

    private static Scanner scanner;

    public static void main(String[] args) {

        inventoryManager = new InventoryManager();

        scanner = new Scanner(System.in);

        // Add some sample data

        initializeSampleData();

        // Start interactive menu

        showMainMenu();

    }

    private static void initializeSampleData() {

        System.out.println("Initializing sample data...");

        Product[] sampleProducts = {

            new Product("P001", "Laptop", 10, 999.99),

            new Product("P002", "Mouse", 50, 29.99),

            new Product("P003", "Keyboard", 30, 79.99),

            new Product("P004", "Monitor", 15, 299.99),

            new Product("P005", "Webcam", 8, 89.99),

            new Product("P006", "Headphones", 25, 149.99),

            new Product("P007", "USB Cable", 100, 12.99),

            new Product("P008", "External Hard Drive", 5, 129.99),

            new Product("P009", "Wireless Router", 12, 79.99),

            new Product("P010", "Tablet", 7, 349.99)

        };

        for (Product product : sampleProducts) {

            inventoryManager.addProduct(product);

        }

        System.out.println("Sample data initialized!\n");

    }

    private static void showMainMenu() {

        while (true) {

            System.out.println("\n===== INVENTORY MANAGEMENT SYSTEM =====");

            System.out.println("1. Display All Products");

            System.out.println("2. Add New Product");

            System.out.println("3. Update Product");

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

            System.out.println("5. Search Product by ID");

            System.out.println("6. Search Products by Name");

            System.out.println("7. View Low Stock Products");

            System.out.println("8. View Products Sorted by Name");

            System.out.println("9. View Products Sorted by Quantity");

            System.out.println("10. View Inventory Statistics");

            System.out.println("11. Bulk Update Quantities");

            System.out.println("12. Performance Test");

            System.out.println("0. Exit");

            System.out.print("Choose an option: ");

            int choice = getIntInput();

            switch (choice) {

                case 1:

                    inventoryManager.displayAllProducts();

                    break;

                case 2:

                    addNewProduct();

                    break;

                case 3:

                    updateProduct();

                    break;

                case 4:

                    deleteProduct();

                    break;

                case 5:

                    searchProductById();

                    break;

                case 6:

                    searchProductsByName();

                    break;

                case 7:

                    viewLowStockProducts();

                    break;

                case 8:

                    viewProductsSortedByName();

                    break;

                case 9:

                    viewProductsSortedByQuantity();

                    break;

                case 10:

                    viewInventoryStatistics();

                    break;

                case 11:

                    bulkUpdateQuantities();

                    break;

                case 12:

                    performanceTest();

                    break;

                case 0:

                    System.out.println("Thank you for using Inventory Management System!");

                    return;

                default:

                    System.out.println("Invalid option. Please try again.");

            }

        }

    }

    private static void addNewProduct() {

        System.out.println("\n--- Add New Product ---");

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

        String id = scanner.nextLine().trim();

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

        String name = scanner.nextLine().trim();

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

        int quantity = getIntInput();

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

        double price = getDoubleInput();

        try {

            Product product = new Product(id, name, quantity, price);

            inventoryManager.addProduct(product);

        } catch (IllegalArgumentException e) {

            System.out.println("Error: " + e.getMessage());

        }

    }

    private static void updateProduct() {

        System.out.println("\n--- Update Product ---");

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

        String id = scanner.nextLine().trim();

        System.out.print("Enter new name (or press Enter to keep current): ");

        String newName = scanner.nextLine().trim();

        if (newName.isEmpty()) newName = null;

        System.out.print("Enter new quantity (or -1 to keep current): ");

        int newQuantity = getIntInput();

        System.out.print("Enter new price (or -1 to keep current): ");

        double newPrice = getDoubleInput();

        inventoryManager.updateProduct(id, newName, newQuantity, newPrice);

    }

    private static void deleteProduct() {

        System.out.println("\n--- Delete Product ---");

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

        String id = scanner.nextLine().trim();

        inventoryManager.deleteProduct(id);

    }

    private static void searchProductById() {

        System.out.println("\n--- Search Product by ID ---");

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

        String id = scanner.nextLine().trim();

        Product product = inventoryManager.getProduct(id);

        if (product != null) {

            System.out.println("Product found: " + product);

        } else {

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

        }

    }

    private static void searchProductsByName() {

        System.out.println("\n--- Search Products by Name ---");

        System.out.print("Enter search term: ");

        String searchTerm = scanner.nextLine().trim();

        List<Product> results = inventoryManager.searchProductsByName(searchTerm);

        if (results.isEmpty()) {

            System.out.println("No products found matching: " + searchTerm);

        } else {

            System.out.println("Found " + results.size() + " product(s):");

            results.forEach(System.out::println);

        }

    }

    private static void viewLowStockProducts() {

        System.out.println("\n--- Low Stock Products ---");

        System.out.print("Enter stock threshold: ");

        int threshold = getIntInput();

        List<Product> lowStockProducts = inventoryManager.getLowStockProducts(threshold);

        if (lowStockProducts.isEmpty()) {

            System.out.println("No products with stock <= " + threshold);

        } else {

            System.out.println("Products with stock <= " + threshold + ":");

            lowStockProducts.forEach(System.out::println);

        }

    }

    private static void viewProductsSortedByName() {

        System.out.println("\n--- Products Sorted by Name ---");

        List<Product> sortedProducts = inventoryManager.getAllProductsSortedByName();

        sortedProducts.forEach(System.out::println);

    }

    private static void viewProductsSortedByQuantity() {

        System.out.println("\n--- Products Sorted by Quantity ---");

        List<Product> sortedProducts = inventoryManager.getAllProductsSortedByQuantity();

        sortedProducts.forEach(System.out::println);

    }

    private static void viewInventoryStatistics() {

        System.out.println("\n--- Inventory Statistics ---");

        System.out.println("Total Products: " + inventoryManager.size());

        System.out.println("Total Inventory Value: $" + String.format("%.2f", inventoryManager.getTotalInventoryValue()));

        List<Product> lowStock = inventoryManager.getLowStockProducts(10);

        System.out.println("Products with stock <= 10: " + lowStock.size());

        if (!lowStock.isEmpty()) {

            System.out.println("Low stock products:");

            lowStock.forEach(p -> System.out.println("  " + p.getProductName() + " (Qty: " + p.getQuantity() + ")"));

        }

    }

    private static void bulkUpdateQuantities() {

        System.out.println("\n--- Bulk Update Quantities ---");

        Map<String, Integer> updates = new HashMap<>();

        System.out.println("Enter product updates (enter 'done' to finish):");

        while (true) {

            System.out.print("Product ID (or 'done'): ");

            String id = scanner.nextLine().trim();

            if (id.equalsIgnoreCase("done")) break;

            System.out.print("New quantity: ");

            int quantity = getIntInput();

            updates.put(id, quantity);

        }

        if (!updates.isEmpty()) {

            inventoryManager.bulkUpdateQuantity(updates);

        }

    }

    private static void performanceTest() {

        System.out.println("\n--- Performance Test ---");

        System.out.println("Testing operations on current inventory...");

        long startTime, endTime;

        // Test search performance

        startTime = System.nanoTime();

        for (int i = 0; i < 1000; i++) {

            inventoryManager.getProduct("P001");  // HashMap lookup

        }

        endTime = System.nanoTime();

        System.out.println("1000 ID lookups took: " + (endTime - startTime) / 1\_000\_000.0 + " ms");

        // Test name search performance

        startTime = System.nanoTime();

        inventoryManager.searchProductsByName("Laptop");

        endTime = System.nanoTime();

        System.out.println("Name search took: " + (endTime - startTime) / 1\_000\_000.0 + " ms");

        // Test sorting performance

        startTime = System.nanoTime();

        inventoryManager.getAllProductsSortedByName();

        endTime = System.nanoTime();

        System.out.println("Sorting by name took: " + (endTime - startTime) / 1\_000\_000.0 + " ms");

    }

    // Utility methods for input handling

    private static int getIntInput() {

        while (true) {

            try {

                int value = Integer.parseInt(scanner.nextLine().trim());

                return value;

            } catch (NumberFormatException e) {

                System.out.print("Invalid input. Please enter a number: ");

            }

        }

    }

    private static double getDoubleInput() {

        while (true) {

            try {

                double value = Double.parseDouble(scanner.nextLine().trim());

                return value;

            } catch (NumberFormatException e) {

                System.out.print("Invalid input. Please enter a number: ");

            }

        }

    }

}

Product.java

public class Product {

    private String productId;

    private String productName;

    private int quantity;

    private double price;

    // Constructor

    public Product(String productId, String productName, int quantity, double price) {

        if (productId == null || productId.trim().isEmpty()) {

            throw new IllegalArgumentException("Product ID cannot be null or empty");

        }

        if (productName == null || productName.trim().isEmpty()) {

            throw new IllegalArgumentException("Product name cannot be null or empty");

        }

        if (quantity < 0) {

            throw new IllegalArgumentException("Quantity cannot be negative");

        }

        if (price < 0) {

            throw new IllegalArgumentException("Price cannot be negative");

        }

        this.productId = productId.trim();

        this.productName = productName.trim();

        this.quantity = quantity;

        this.price = price;

    }

    // Getters

    public String getProductId() {

        return productId;

    }

    public String getProductName() {

        return productName;

    }

    public int getQuantity() {

        return quantity;

    }

    public double getPrice() {

        return price;

    }

    // Setters with validation

    public void setProductName(String productName) {

        if (productName == null || productName.trim().isEmpty()) {

            throw new IllegalArgumentException("Product name cannot be null or empty");

        }

        this.productName = productName.trim();

    }

    public void setQuantity(int quantity) {

        if (quantity < 0) {

            throw new IllegalArgumentException("Quantity cannot be negative");

        }

        this.quantity = quantity;

    }

    public void setPrice(double price) {

        if (price < 0) {

            throw new IllegalArgumentException("Price cannot be negative");

        }

        this.price = price;

    }

    // Utility methods

    public double getTotalValue() {

        return quantity \* price;

    }

    public boolean isInStock() {

        return quantity > 0;

    }

    @Override

    public String toString() {

        return String.format("Product{ID='%s', Name='%s', Quantity=%d, Price=%.2f, Value=%.2f}",

                productId, productName, quantity, price, getTotalValue());

    }

    @Override

    public boolean equals(Object obj) {

        if (this == obj) return true;

        if (obj == null || getClass() != obj.getClass()) return false;

        Product product = (Product) obj;

        return productId.equals(product.productId);

    }

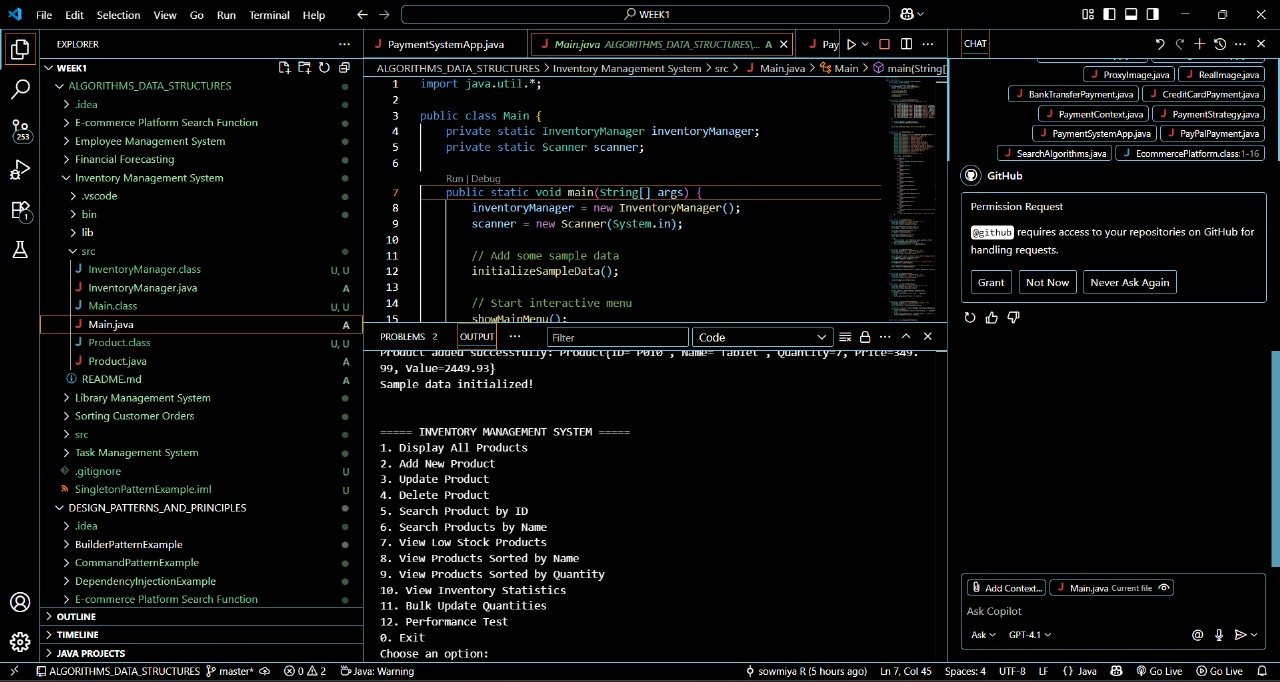
    @Override

    public int hashCode() {

        return productId.hashCode();

    }

}



**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.

Product.class

// Source code is decompiled from a .class file using FernFlower decompiler.

public class Product implements Comparable<Product> {

   private String productId;

   private String productName;

   private String category;

   private double price;

   private int stock;

   private double rating;

   public Product(String var1, String var2, String var3, double var4, int var6, double var7) {

      if (var1 != null && !var1.trim().isEmpty()) {

         if (var2 != null && !var2.trim().isEmpty()) {

            if (var3 != null && !var3.trim().isEmpty()) {

               if (var4 < 0.0) {

                  throw new IllegalArgumentException("Price cannot be negative");

               } else if (var6 < 0) {

                  throw new IllegalArgumentException("Stock cannot be negative");

               } else if (!(var7 < 0.0) && !(var7 > 5.0)) {

                  this.productId = var1.trim();

                  this.productName = var2.trim();

                  this.category = var3.trim();

                  this.price = var4;

                  this.stock = var6;

                  this.rating = var7;

               } else {

                  throw new IllegalArgumentException("Rating must be between 0 and 5");

               }

            } else {

               throw new IllegalArgumentException("Category cannot be null or empty");

            }

         } else {

            throw new IllegalArgumentException("Product name cannot be null or empty");

         }

      } else {

         throw new IllegalArgumentException("Product ID cannot be null or empty");

      }

   }

   public String getProductId() {

      return this.productId;

   }

   public String getProductName() {

      return this.productName;

   }

   public String getCategory() {

      return this.category;

   }

   public double getPrice() {

      return this.price;

   }

   public int getStock() {

      return this.stock;

   }

   public double getRating() {

      return this.rating;

   }

   public void setPrice(double var1) {

      if (var1 < 0.0) {

         throw new IllegalArgumentException("Price cannot be negative");

      } else {

         this.price = var1;

      }

   }

   public void setStock(int var1) {

      if (var1 < 0) {

         throw new IllegalArgumentException("Stock cannot be negative");

      } else {

         this.stock = var1;

      }

   }

   public void setRating(double var1) {

      if (!(var1 < 0.0) && !(var1 > 5.0)) {

         this.rating = var1;

      } else {

         throw new IllegalArgumentException("Rating must be between 0 and 5");

      }

   }

  public boolean isInStock() {

      return this.stock > 0;

   }

   public boolean isHighRated() {

      return this.rating >= 4.0;

   }

   public String getPriceRange() {

      if (this.price < 50.0) {

         return "Budget";

      } else {

         return this.price < 200.0 ? "Mid-range" : "Premium";

      }

   }

   public int compareTo(Product var1) {

      return this.productName.compareToIgnoreCase(var1.productName);

   }

   public static int compareById(Product var0, Product var1) {

      return var0.productId.compareToIgnoreCase(var1.productId);

   }

   public static int compareByPrice(Product var0, Product var1) {

      return Double.compare(var0.price, var1.price);

   }

   public static int compareByRating(Product var0, Product var1) {

      return Double.compare(var1.rating, var0.rating);

   }

   public static int compareByCategory(Product var0, Product var1) {

      return var0.category.compareToIgnoreCase(var1.category);

   }

   public String toString() {

      return String.format("Product{ID='%s', Name='%s', Category='%s', Price=$%.2f, Stock=%d, Rating=%.1f}", this.productId, this.productName, this.category, this.price, this.stock, this.rating);

   }

   public boolean equals(Object var1) {

      if (this == var1) {

         return true;

      } else if (var1 != null && this.getClass() == var1.getClass()) {

         Product var2 = (Product)var1;

         return this.productId.equals(var2.productId);

      } else {

         return false;

      }

   }

   public int hashCode() {

      return this.productId.hashCode();

   }

   public String toSearchResult() {

      return String.format("%-8s | %-25s | %-12s | $%-8.2f | Stock: %-3d | Rating: %.1f", this.productId, this.productName.length() > 25 ? this.productName.substring(0, 22) + "..." : this.productName, this.category, this.price, this.stock, this.rating);

   }

}

Search Algorithms

// Source code is decompiled from a .class file using FernFlower decompiler.

public class SearchAlgorithms$SearchResult {

   private Product product;

   private int comparisons;

   private long executionTimeNanos;

   private String algorithm;

   private boolean found;

   public SearchAlgorithms$SearchResult(Product var1, int var2, long var3, String var5, boolean var6) {

      this.product = var1;

      this.comparisons = var2;

      this.executionTimeNanos = var3;

      this.algorithm = var5;

      this.found = var6;

   }

   public Product getProduct() {

      return this.product;

   }

   public int getComparisons() {

      return this.comparisons;

   }

   public long getExecutionTimeNanos() {

      return this.executionTimeNanos;

   }

   public double getExecutionTimeMillis() {

      return (double)this.executionTimeNanos / 1000000.0;

   }

   public String getAlgorithm() {

      return this.algorithm;

 }

   public boolean isFound() {

      return this.found;

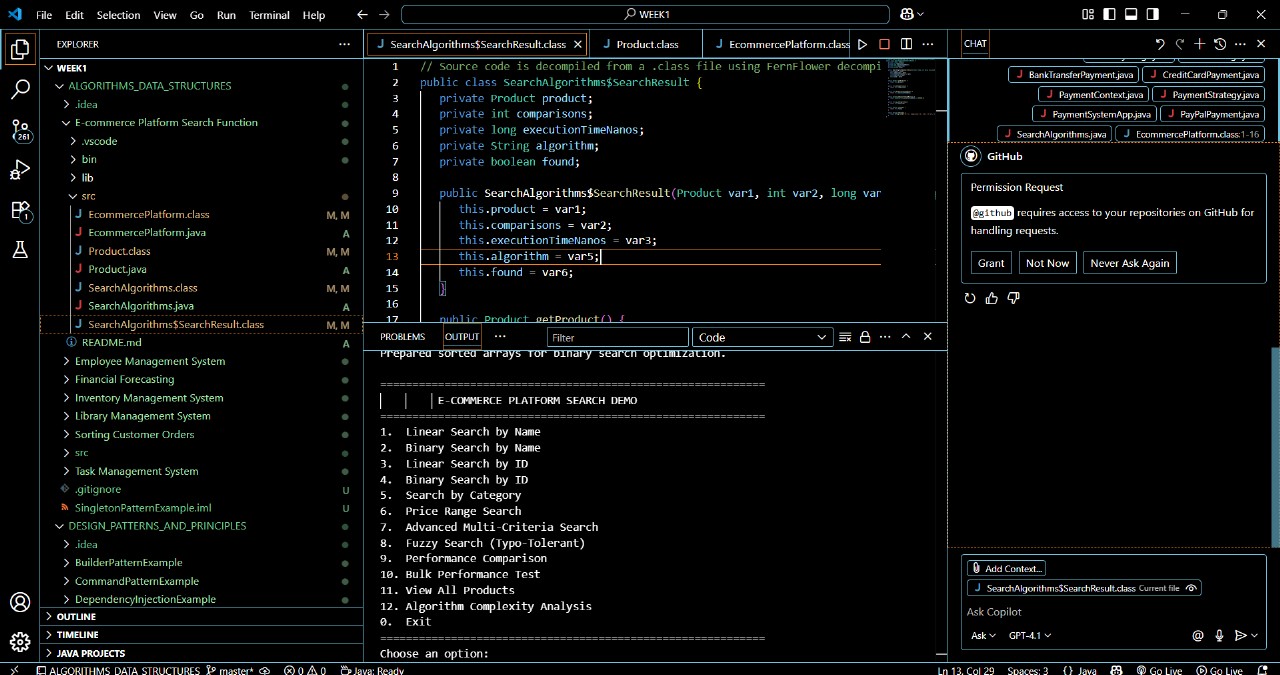
   }

   public String toString() {

      return String.format("%s: %s | Comparisons: %d | Time: %.3f ms", this.algorithm, this.found ? "FOUND" : "NOT FOUND", this.comparisons, this.getExecutionTimeMillis());

   }

}



**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.

Order Sorting System.java

import java.util.\*;

import java.util.concurrent.ThreadLocalRandom;

class Order {

    private String orderId;

    private String customerName;

    private double totalPrice;

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

        this.orderId = orderId;

        this.customerName = customerName;

        this.totalPrice = totalPrice;

    }

    // Getters

    public String getOrderId() { return orderId; }

    public String getCustomerName() { return customerName; }

    public double getTotalPrice() { return totalPrice; }

    // Setters

    public void setOrderId(String orderId) { this.orderId = orderId; }

    public void setCustomerName(String customerName) { this.customerName = customerName; }

    public void setTotalPrice(double totalPrice) { this.totalPrice = totalPrice; }

    @Override

    public String toString() {

        return String.format("Order(%s, %s, $%.2f)", orderId, customerName, totalPrice);

    }

    @Override

    public boolean equals(Object obj) {

        if (this == obj) return true;

        if (obj == null || getClass() != obj.getClass()) return false;

        Order order = (Order) obj;

        return Double.compare(order.totalPrice, totalPrice) == 0 &&

               Objects.equals(orderId, order.orderId) &&

               Objects.equals(customerName, order.customerName);

    }

    @Override

    public int hashCode() {

        return Objects.hash(orderId, customerName, totalPrice);

    }

}

class OrderSorter {

    /\*\*

     \* Bubble Sort implementation for sorting orders by total price

     \* Time Complexity: O(n²)

     \* Space Complexity: O(1)

     \*/

    public static List<Order> bubbleSort(List<Order> orders) {

        List<Order> sortedOrders = new ArrayList<>(orders); // Create a copy

        int n = sortedOrders.size();

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

            boolean swapped = false; // Flag to optimize - if no swaps occur, list is sorted

            // Last i elements are already in place

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

                // Compare adjacent elements

                if (sortedOrders.get(j).getTotalPrice() > sortedOrders.get(j + 1).getTotalPrice()) {

                    // Swap elements

                    Collections.swap(sortedOrders, j, j + 1);

                    swapped = true;

                }

            }

            // If no swapping occurred, array is sorted

            if (!swapped) {

                break;

            }

        }

        return sortedOrders;

    }

    public static List<Order> quickSort(List<Order> orders) {

        List<Order> sortedOrders = new ArrayList<>(orders); // Create a copy

        quickSortRecursive(sortedOrders, 0, sortedOrders.size() - 1);

        return sortedOrders;

    }

    private static void quickSortRecursive(List<Order> orders, int low, int high) {

        if (low < high) {

            // Partition the array and get pivot index

            int pivotIndex = partition(orders, low, high);

            // Recursively sort elements before and after partition

            quickSortRecursive(orders, low, pivotIndex - 1);

            quickSortRecursive(orders, pivotIndex + 1, high);

        }

    }

    private static int partition(List<Order> orders, int low, int high) {

        // Choose the rightmost element as pivot

        double pivot = orders.get(high).getTotalPrice();

        // Index of smaller element (indicates right position of pivot)

        int i = low - 1;

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

            // If current element is smaller than or equal to pivot

            if (orders.get(j).getTotalPrice() <= pivot) {

                i++;

                Collections.swap(orders, i, j);

            }

        }

        // Place pivot in correct position

        Collections.swap(orders, i + 1, high);

        return i + 1;

    }

    public static List<Order> insertionSort(List<Order> orders) {

        List<Order> sortedOrders = new ArrayList<>(orders);

        for (int i = 1; i < sortedOrders.size(); i++) {

            Order key = sortedOrders.get(i);

            int j = i - 1;

            // Move elements greater than key one position ahead

            while (j >= 0 && sortedOrders.get(j).getTotalPrice() > key.getTotalPrice()) {

                sortedOrders.set(j + 1, sortedOrders.get(j));

                j--;

            }

            sortedOrders.set(j + 1, key);

        }

        return sortedOrders;

    }

    public static List<Order> mergeSort(List<Order> orders) {

        if (orders.size() <= 1) {

            return new ArrayList<>(orders);

        }

        // Divide the array into two halves

        int mid = orders.size() / 2;

        List<Order> left = mergeSort(orders.subList(0, mid));

        List<Order> right = mergeSort(orders.subList(mid, orders.size()));

        // Merge the sorted halves

        return merge(left, right);

    }

    private static List<Order> merge(List<Order> left, List<Order> right) {

        List<Order> result = new ArrayList<>();

        int i = 0, j = 0;

        // Compare elements from both arrays and merge

        while (i < left.size() && j < right.size()) {

            if (left.get(i).getTotalPrice() <= right.get(j).getTotalPrice()) {

                result.add(left.get(i));

                i++;

            } else {

                result.add(right.get(j));

                j++;

            }

        }

        // Add remaining elements

        while (i < left.size()) {

            result.add(left.get(i));

            i++;

        }

        while (j < right.size()) {

            result.add(right.get(j));

            j++;

        }

        return result;

    }

}

class PerformanceAnalyzer {

    /\*\*

     \* Generate sample orders with random data

     \*/

    public static List<Order> generateSampleOrders(int count) {

        List<Order> orders = new ArrayList<>();

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

            String orderId = String.format("ORD%04d", i + 1);

            String customerName = "Customer\_" + (i + 1);

            double totalPrice = Math.round(ThreadLocalRandom.current().nextDouble(10.0, 1000.0) \* 100.0) / 100.0;

            orders.add(new Order(orderId, customerName, totalPrice));

        }

        return orders;

    }

    public static class SortingResult {

        public List<Order> sortedOrders;

        public long executionTime; // in nanoseconds

        public SortingResult(List<Order> sortedOrders, long executionTime) {

            this.sortedOrders = sortedOrders;

            this.executionTime = executionTime;

        }

        public double getExecutionTimeInSeconds() {

            return executionTime / 1\_000\_000\_000.0;

        }

    }

    @FunctionalInterface

    public interface SortingAlgorithm {

        List<Order> sort(List<Order> orders);

    }

    public static SortingResult measureSortingTime(SortingAlgorithm algorithm, List<Order> orders) {

        long startTime = System.nanoTime();

        List<Order> sortedOrders = algorithm.sort(orders);

        long endTime = System.nanoTime();

        long executionTime = endTime - startTime;

        return new SortingResult(sortedOrders, executionTime);

    }

    public static void compareAlgorithms(List<Order> orders) {

        System.out.println("\n" + "=".repeat(60));

        System.out.printf("PERFORMANCE COMPARISON - %d orders%n", orders.size());

        System.out.println("=".repeat(60));

        Map<String, SortingAlgorithm> algorithms = new LinkedHashMap<>();

        algorithms.put("Bubble Sort", OrderSorter::bubbleSort);

        algorithms.put("Insertion Sort", OrderSorter::insertionSort);

        algorithms.put("Quick Sort", OrderSorter::quickSort);

        algorithms.put("Merge Sort", OrderSorter::mergeSort);

        List<String> results = new ArrayList<>();

        String fastest = "";

        double fastestTime = Double.MAX\_VALUE;

        for (Map.Entry<String, SortingAlgorithm> entry : algorithms.entrySet()) {

            String name = entry.getKey();

            SortingAlgorithm algorithm = entry.getValue();

            try {

                SortingResult result = measureSortingTime(algorithm, orders);

                double timeInSeconds = result.getExecutionTimeInSeconds();

                System.out.printf("%-15s: %.6f seconds%n", name, timeInSeconds);

                if (timeInSeconds < fastestTime) {

                    fastestTime = timeInSeconds;

                    fastest = name;

                }

            } catch (Exception e) {

                System.out.printf("%-15s: Error - %s%n", name, e.getMessage());

            }

        }

        System.out.println("=".repeat(60));

        if (!fastest.isEmpty()) {

            System.out.printf("Fastest Algorithm: %s (%.6f seconds)%n", fastest, fastestTime);

        }

    }

    public static boolean verifySorting(List<Order> orders) {

        for (int i = 0; i < orders.size() - 1; i++) {

            if (orders.get(i).getTotalPrice() > orders.get(i + 1).getTotalPrice()) {

                return false;

            }

        }

        return true;

    }

}

public class OrderSortingSystem {

    public static void explainAlgorithms() {

        System.out.println("\n" + "=".repeat(70));

        System.out.println("SORTING ALGORITHMS EXPLANATION");

        System.out.println("=".repeat(70));

        Map<String, Map<String, String>> explanations = new LinkedHashMap<>();

        // Bubble Sort

        Map<String, String> bubbleSort = new HashMap<>();

        bubbleSort.put("description", "Repeatedly steps through the list, compares adjacent elements and swaps them if they're in wrong order.");

        bubbleSort.put("time\_complexity", "O(n²) - worst and average case, O(n) - best case");

        bubbleSort.put("space\_complexity", "O(1)");

        bubbleSort.put("pros", "Simple to understand and implement, Stable sorting algorithm, In-place sorting");

        bubbleSort.put("cons", "Very inefficient for large datasets, O(n²) time complexity, More swaps compared to other algorithms");

        explanations.put("Bubble Sort", bubbleSort);

        // Insertion Sort

        Map<String, String> insertionSort = new HashMap<>();

        insertionSort.put("description", "Builds the final sorted array one item at a time, inserting each element in its correct position.");

        insertionSort.put("time\_complexity", "O(n²) - worst and average case, O(n) - best case");

        insertionSort.put("space\_complexity", "O(1)");

        insertionSort.put("pros", "Simple implementation, Efficient for small datasets, Stable and in-place, Online algorithm");

        insertionSort.put("cons", "Inefficient for large datasets, More writes than selection sort");

        explanations.put("Insertion Sort", insertionSort);

        // Quick Sort

        Map<String, String> quickSort = new HashMap<>();

        quickSort.put("description", "Divides array into partitions around a pivot, recursively sorts partitions.");

        quickSort.put("time\_complexity", "O(n log n) - average case, O(n²) - worst case");

        quickSort.put("space\_complexity", "O(log n) - average case");

        quickSort.put("pros", "Generally faster than other O(n²) algorithms, In-place sorting, Cache efficient");

        quickSort.put("cons", "Worst case O(n²), Not stable, Performance depends on pivot selection");

        explanations.put("Quick Sort", quickSort);

        // Merge Sort

        Map<String, String> mergeSort = new HashMap<>();

        mergeSort.put("description", "Divides array into halves, recursively sorts them, then merges sorted halves.");

        mergeSort.put("time\_complexity", "O(n log n) - all cases");

        mergeSort.put("space\_complexity", "O(n)");

        mergeSort.put("pros", "Guaranteed O(n log n), Stable sorting, Predictable performance");

        mergeSort.put("cons", "Requires additional memory, Not in-place, Overhead for small arrays");

        explanations.put("Merge Sort", mergeSort);

        for (Map.Entry<String, Map<String, String>> entry : explanations.entrySet()) {

            String algorithm = entry.getKey();

            Map<String, String> details = entry.getValue();

            System.out.println("\n" + algorithm.toUpperCase() + ":");

            System.out.println("Description: " + details.get("description"));

            System.out.println("Time Complexity: " + details.get("time\_complexity"));

            System.out.println("Space Complexity: " + details.get("space\_complexity"));

            System.out.println("Pros: " + details.get("pros"));

            System.out.println("Cons: " + details.get("cons"));

            System.out.println("-".repeat(70));

        }

    }

    public static void whyQuickSortPreferred() {

        System.out.println("\n" + "=".repeat(70));

        System.out.println("WHY QUICK SORT IS PREFERRED OVER BUBBLE SORT");

        System.out.println("=".repeat(70));

        String[][] comparisonPoints = {

            {"Time Complexity", "O(n²) - always quadratic", "O(n log n) average, O(n²) worst case", "Quick Sort - much better average performance"},

            {"Practical Performance", "Very slow for large datasets", "Fast for most real-world scenarios", "Quick Sort - significantly faster in practice"},

            {"Scalability", "Becomes unusable with large data", "Scales well with increasing data size", "Quick Sort - handles large datasets efficiently"},

            {"Industry Usage", "Only used for educational purposes", "Widely used in production systems", "Quick Sort - industry standard"},

            {"Adaptability", "Performance doesn't improve with partially sorted data", "Can be optimized for different scenarios", "Quick Sort - more adaptable"}

        };

        for (String[] point : comparisonPoints) {

            System.out.println("\n" + point[0] + ":");

            System.out.println("  Bubble Sort: " + point[1]);

            System.out.println("  Quick Sort:  " + point[2]);

            System.out.println("  Winner: " + point[3]);

        }

        System.out.println("\n" + "=".repeat(70));

        System.out.println("CONCLUSION:");

        System.out.println("Quick Sort is preferred because:");

        System.out.println("1. Much better average-case time complexity O(n log n) vs O(n²)");

        System.out.println("2. Significantly faster in practice for real-world data");

        System.out.println("3. Scales well with large datasets");

        System.out.println("4. Used in production systems and standard libraries");

        System.out.println("5. Can be optimized further with techniques like randomized pivots");

        System.out.println("=".repeat(70));

    }

    public static void demonstrateSorting() {

        System.out.println("CUSTOMER ORDER SORTING SYSTEM");

        System.out.println("=".repeat(50));

        // Generate sample orders

        List<Order> orders = PerformanceAnalyzer.generateSampleOrders(10);

        System.out.println("\nOriginal Orders:");

        for (Order order : orders) {

            System.out.println("  " + order);

        }

        // Demonstrate Bubble Sort

        System.out.println("\n" + "=".repeat(50));

        System.out.println("BUBBLE SORT RESULTS:");

        System.out.println("=".repeat(50));

        PerformanceAnalyzer.SortingResult bubbleResult = PerformanceAnalyzer.measureSortingTime(

            OrderSorter::bubbleSort, orders

        );

        for (Order order : bubbleResult.sortedOrders) {

            System.out.println("  " + order);

        }

        System.out.printf("Execution Time: %.6f seconds%n", bubbleResult.getExecutionTimeInSeconds());

        System.out.println("Correctly Sorted: " + PerformanceAnalyzer.verifySorting(bubbleResult.sortedOrders));

        // Demonstrate Quick Sort

        System.out.println("\n" + "=".repeat(50));

        System.out.println("QUICK SORT RESULTS:");

        System.out.println("=".repeat(50));

        PerformanceAnalyzer.SortingResult quickResult = PerformanceAnalyzer.measureSortingTime(

            OrderSorter::quickSort, orders

        );

        for (Order order : quickResult.sortedOrders) {

            System.out.println("  " + order);

        }

        System.out.printf("Execution Time: %.6f seconds%n", quickResult.getExecutionTimeInSeconds());

        System.out.println("Correctly Sorted: " + PerformanceAnalyzer.verifySorting(quickResult.sortedOrders));

        // Performance comparison with larger dataset

        System.out.println("\n" + "=".repeat(50));

        System.out.println("PERFORMANCE ANALYSIS");

        System.out.println("=".repeat(50));

        // Test with different sizes

        int[] sizes = {100, 500, 1000};

        for (int size : sizes) {

            List<Order> testOrders = PerformanceAnalyzer.generateSampleOrders(size);

            PerformanceAnalyzer.compareAlgorithms(testOrders);

        }

    }

    public static void main(String[] args) {

        // Run the demonstration

        explainAlgorithms();

        demonstrateSorting();

        whyQuickSortPreferred();

        // Interactive section

        System.out.println("\n" + "=".repeat(50));

        System.out.println("INTERACTIVE TESTING");

        System.out.println("=".repeat(50));

        System.out.println("You can now test the algorithms with custom data!");

        System.out.println("Example usage:");

        System.out.println("List<Order> orders = PerformanceAnalyzer.generateSampleOrders(50);");

        System.out.println("List<Order> sortedOrders = OrderSorter.quickSort(orders);");

        System.out.println("PerformanceAnalyzer.compareAlgorithms(orders);");

        // Demonstrate with user input simulation

        Scanner scanner = new Scanner(System.in);

        System.out.println("\nWould you like to test with a custom dataset size? (y/n):");

        try {

            String input = scanner.nextLine();

            if (input.toLowerCase().startsWith("y")) {

                System.out.print("Enter the number of orders to generate: ");

                int customSize = scanner.nextInt();

                if (customSize > 0 && customSize <= 10000) {

                    List<Order> customOrders = PerformanceAnalyzer.generateSampleOrders(customSize);

                    System.out.println("\nTesting with " + customSize + " orders:");

                    PerformanceAnalyzer.compareAlgorithms(customOrders);

                } else {

                    System.out.println("Please enter a number between 1 and 10000.");

                }

            }

        } catch (Exception e) {

            System.out.println("Invalid input. Ending program.");

        } finally {

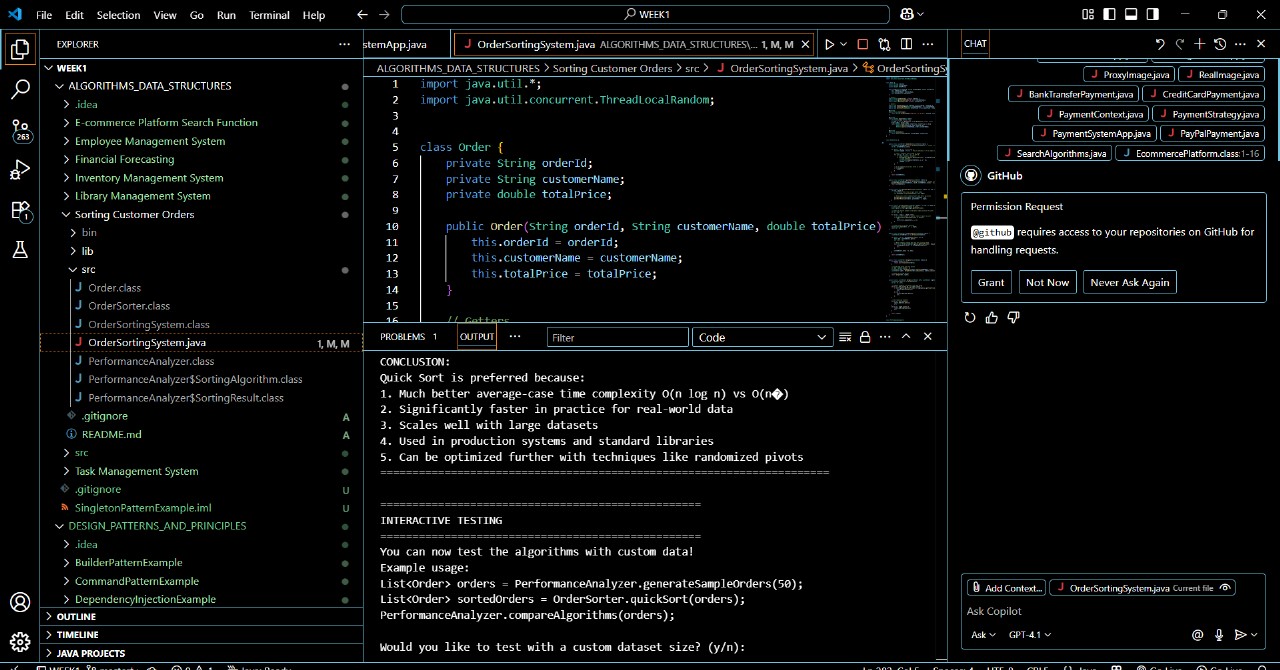
            scanner.close();

        }

        System.out.println("\nThank you for using the Customer Order Sorting System!");

    }

}



**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.

Employee.class

import java.util.\*;

import java.util.concurrent.ThreadLocalRandom;

class Order {

    private String orderId;

    private String customerName;

    private double totalPrice;

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

        this.orderId = orderId;

        this.customerName = customerName;

        this.totalPrice = totalPrice;

    }

    // Getters

    public String getOrderId() { return orderId; }

    public String getCustomerName() { return customerName; }

    public double getTotalPrice() { return totalPrice; }

    // Setters

    public void setOrderId(String orderId) { this.orderId = orderId; }

    public void setCustomerName(String customerName) { this.customerName = customerName; }

    public void setTotalPrice(double totalPrice) { this.totalPrice = totalPrice; }

    @Override

    public String toString() {

        return String.format("Order(%s, %s, $%.2f)", orderId, customerName, totalPrice);

    }

    @Override

    public boolean equals(Object obj) {

        if (this == obj) return true;

        if (obj == null || getClass() != obj.getClass()) return false;

        Order order = (Order) obj;

        return Double.compare(order.totalPrice, totalPrice) == 0 &&

               Objects.equals(orderId, order.orderId) &&

               Objects.equals(customerName, order.customerName);

    }

    @Override

    public int hashCode() {

        return Objects.hash(orderId, customerName, totalPrice);

    }

}

class OrderSorter {

    public static List<Order> bubbleSort(List<Order> orders) {

        List<Order> sortedOrders = new ArrayList<>(orders); // Create a copy

        int n = sortedOrders.size();

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

            boolean swapped = false; // Flag to optimize - if no swaps occur, list is sorted

            // Last i elements are already in place

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

                // Compare adjacent elements

                if (sortedOrders.get(j).getTotalPrice() > sortedOrders.get(j + 1).getTotalPrice()) {

                    // Swap elements

                    Collections.swap(sortedOrders, j, j + 1);

                    swapped = true;

                }

            }

            // If no swapping occurred, array is sorted

            if (!swapped) {

                break;

            }

        }

        return sortedOrders;

    }

    public static List<Order> quickSort(List<Order> orders) {

        List<Order> sortedOrders = new ArrayList<>(orders); // Create a copy

        quickSortRecursive(sortedOrders, 0, sortedOrders.size() - 1);

        return sortedOrders;

    }

    private static void quickSortRecursive(List<Order> orders, int low, int high) {

        if (low < high) {

            // Partition the array and get pivot index

            int pivotIndex = partition(orders, low, high);

            // Recursively sort elements before and after partition

            quickSortRecursive(orders, low, pivotIndex - 1);

            quickSortRecursive(orders, pivotIndex + 1, high);

        }

    }

    private static int partition(List<Order> orders, int low, int high) {

        // Choose the rightmost element as pivot

        double pivot = orders.get(high).getTotalPrice();

        // Index of smaller element (indicates right position of pivot)

        int i = low - 1;

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

            // If current element is smaller than or equal to pivot

            if (orders.get(j).getTotalPrice() <= pivot) {

                i++;

                Collections.swap(orders, i, j);

            }

        }

        // Place pivot in correct position

        Collections.swap(orders, i + 1, high);

        return i + 1;

    }

    public static List<Order> insertionSort(List<Order> orders) {

        List<Order> sortedOrders = new ArrayList<>(orders);

        for (int i = 1; i < sortedOrders.size(); i++) {

            Order key = sortedOrders.get(i);

            int j = i - 1;

            // Move elements greater than key one position ahead

            while (j >= 0 && sortedOrders.get(j).getTotalPrice() > key.getTotalPrice()) {

                sortedOrders.set(j + 1, sortedOrders.get(j));

                j--;

            }

            sortedOrders.set(j + 1, key);

        }

        return sortedOrders;

    }

    public static List<Order> mergeSort(List<Order> orders) {

        if (orders.size() <= 1) {

            return new ArrayList<>(orders);

        }

        // Divide the array into two halves

        int mid = orders.size() / 2;

        List<Order> left = mergeSort(orders.subList(0, mid));

        List<Order> right = mergeSort(orders.subList(mid, orders.size()));

        // Merge the sorted halves

        return merge(left, right);

    }

    private static List<Order> merge(List<Order> left, List<Order> right) {

        List<Order> result = new ArrayList<>();

        int i = 0, j = 0;

        // Compare elements from both arrays and merge

        while (i < left.size() && j < right.size()) {

            if (left.get(i).getTotalPrice() <= right.get(j).getTotalPrice()) {

                result.add(left.get(i));

                i++;

            } else {

                result.add(right.get(j));

                j++;

            }

        }

        // Add remaining elements

        while (i < left.size()) {

            result.add(left.get(i));

            i++;

        }

        while (j < right.size()) {

            result.add(right.get(j));

            j++;

        }

        return result;

    }

}

class PerformanceAnalyzer {

    public static List<Order> generateSampleOrders(int count) {

        List<Order> orders = new ArrayList<>();

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

            String orderId = String.format("ORD%04d", i + 1);

            String customerName = "Customer\_" + (i + 1);

            double totalPrice = Math.round(ThreadLocalRandom.current().nextDouble(10.0, 1000.0) \* 100.0) / 100.0;

            orders.add(new Order(orderId, customerName, totalPrice));

        }

        return orders;

    }

    public static class SortingResult {

        public List<Order> sortedOrders;

        public long executionTime; // in nanoseconds

        public SortingResult(List<Order> sortedOrders, long executionTime) {

            this.sortedOrders = sortedOrders;

            this.executionTime = executionTime;

        }

        public double getExecutionTimeInSeconds() {

            return executionTime / 1\_000\_000\_000.0;

        }

    }

    @FunctionalInterface

    public interface SortingAlgorithm {

        List<Order> sort(List<Order> orders);

    }

    public static SortingResult measureSortingTime(SortingAlgorithm algorithm, List<Order> orders) {

        long startTime = System.nanoTime();

        List<Order> sortedOrders = algorithm.sort(orders);

        long endTime = System.nanoTime();

        long executionTime = endTime - startTime;

        return new SortingResult(sortedOrders, executionTime);

    }

    public static void compareAlgorithms(List<Order> orders) {

        System.out.println("\n" + "=".repeat(60));

        System.out.printf("PERFORMANCE COMPARISON - %d orders%n", orders.size());

        System.out.println("=".repeat(60));

        Map<String, SortingAlgorithm> algorithms = new LinkedHashMap<>();

        algorithms.put("Bubble Sort", OrderSorter::bubbleSort);

        algorithms.put("Insertion Sort", OrderSorter::insertionSort);

        algorithms.put("Quick Sort", OrderSorter::quickSort);

        algorithms.put("Merge Sort", OrderSorter::mergeSort);

        List<String> results = new ArrayList<>();

        String fastest = "";

        double fastestTime = Double.MAX\_VALUE;

        for (Map.Entry<String, SortingAlgorithm> entry : algorithms.entrySet()) {

            String name = entry.getKey();

            SortingAlgorithm algorithm = entry.getValue();

            try {

                SortingResult result = measureSortingTime(algorithm, orders);

                double timeInSeconds = result.getExecutionTimeInSeconds();

                System.out.printf("%-15s: %.6f seconds%n", name, timeInSeconds);

                if (timeInSeconds < fastestTime) {

                    fastestTime = timeInSeconds;

                    fastest = name;

                }

            } catch (Exception e) {

                System.out.printf("%-15s: Error - %s%n", name, e.getMessage());

            }

        }

        System.out.println("=".repeat(60));

        if (!fastest.isEmpty()) {

            System.out.printf("Fastest Algorithm: %s (%.6f seconds)%n", fastest, fastestTime);

        }

    }

    public static boolean verifySorting(List<Order> orders) {

        for (int i = 0; i < orders.size() - 1; i++) {

            if (orders.get(i).getTotalPrice() > orders.get(i + 1).getTotalPrice()) {

                return false;

            }

        }

        return true;

    }

}

public class OrderSortingSystem {

    public static void explainAlgorithms() {

        System.out.println("\n" + "=".repeat(70));

        System.out.println("SORTING ALGORITHMS EXPLANATION");

        System.out.println("=".repeat(70));

        Map<String, Map<String, String>> explanations = new LinkedHashMap<>();

        // Bubble Sort

        Map<String, String> bubbleSort = new HashMap<>();

        bubbleSort.put("description", "Repeatedly steps through the list, compares adjacent elements and swaps them if they're in wrong order.");

        bubbleSort.put("time\_complexity", "O(n²) - worst and average case, O(n) - best case");

        bubbleSort.put("space\_complexity", "O(1)");

        bubbleSort.put("pros", "Simple to understand and implement, Stable sorting algorithm, In-place sorting");

        bubbleSort.put("cons", "Very inefficient for large datasets, O(n²) time complexity, More swaps compared to other algorithms");

        explanations.put("Bubble Sort", bubbleSort);

        // Insertion Sort

        Map<String, String> insertionSort = new HashMap<>();

        insertionSort.put("description", "Builds the final sorted array one item at a time, inserting each element in its correct position.");

        insertionSort.put("time\_complexity", "O(n²) - worst and average case, O(n) - best case");

        insertionSort.put("space\_complexity", "O(1)");

        insertionSort.put("pros", "Simple implementation, Efficient for small datasets, Stable and in-place, Online algorithm");

        insertionSort.put("cons", "Inefficient for large datasets, More writes than selection sort");

        explanations.put("Insertion Sort", insertionSort);

        // Quick Sort

        Map<String, String> quickSort = new HashMap<>();

        quickSort.put("description", "Divides array into partitions around a pivot, recursively sorts partitions.");

        quickSort.put("time\_complexity", "O(n log n) - average case, O(n²) - worst case");

        quickSort.put("space\_complexity", "O(log n) - average case");

        quickSort.put("pros", "Generally faster than other O(n²) algorithms, In-place sorting, Cache efficient");

        quickSort.put("cons", "Worst case O(n²), Not stable, Performance depends on pivot selection");

        explanations.put("Quick Sort", quickSort);

        // Merge Sort

        Map<String, String> mergeSort = new HashMap<>();

        mergeSort.put("description", "Divides array into halves, recursively sorts them, then merges sorted halves.");

        mergeSort.put("time\_complexity", "O(n log n) - all cases");

        mergeSort.put("space\_complexity", "O(n)");

        mergeSort.put("pros", "Guaranteed O(n log n), Stable sorting, Predictable performance");

        mergeSort.put("cons", "Requires additional memory, Not in-place, Overhead for small arrays");

        explanations.put("Merge Sort", mergeSort);

        for (Map.Entry<String, Map<String, String>> entry : explanations.entrySet()) {

            String algorithm = entry.getKey();

            Map<String, String> details = entry.getValue();

            System.out.println("\n" + algorithm.toUpperCase() + ":");

            System.out.println("Description: " + details.get("description"));

            System.out.println("Time Complexity: " + details.get("time\_complexity"));

            System.out.println("Space Complexity: " + details.get("space\_complexity"));

            System.out.println("Pros: " + details.get("pros"));

            System.out.println("Cons: " + details.get("cons"));

            System.out.println("-".repeat(70));

        }

    }

    public static void whyQuickSortPreferred() {

        System.out.println("\n" + "=".repeat(70));

        System.out.println("WHY QUICK SORT IS PREFERRED OVER BUBBLE SORT");

        System.out.println("=".repeat(70));

        String[][] comparisonPoints = {

            {"Time Complexity", "O(n²) - always quadratic", "O(n log n) average, O(n²) worst case", "Quick Sort - much better average performance"},

            {"Practical Performance", "Very slow for large datasets", "Fast for most real-world scenarios", "Quick Sort - significantly faster in practice"},

            {"Scalability", "Becomes unusable with large data", "Scales well with increasing data size", "Quick Sort - handles large datasets efficiently"},

            {"Industry Usage", "Only used for educational purposes", "Widely used in production systems", "Quick Sort - industry standard"},

            {"Adaptability", "Performance doesn't improve with partially sorted data", "Can be optimized for different scenarios", "Quick Sort - more adaptable"}

        };

        for (String[] point : comparisonPoints) {

            System.out.println("\n" + point[0] + ":");

            System.out.println("  Bubble Sort: " + point[1]);

            System.out.println("  Quick Sort:  " + point[2]);

            System.out.println("  Winner: " + point[3]);

        }

        System.out.println("\n" + "=".repeat(70));

        System.out.println("CONCLUSION:");

        System.out.println("Quick Sort is preferred because:");

        System.out.println("1. Much better average-case time complexity O(n log n) vs O(n²)");

        System.out.println("2. Significantly faster in practice for real-world data");

        System.out.println("3. Scales well with large datasets");

        System.out.println("4. Used in production systems and standard libraries");

        System.out.println("5. Can be optimized further with techniques like randomized pivots");

        System.out.println("=".repeat(70));

    }

    public static void demonstrateSorting() {

        System.out.println("CUSTOMER ORDER SORTING SYSTEM");

        System.out.println("=".repeat(50));

        // Generate sample orders

        List<Order> orders = PerformanceAnalyzer.generateSampleOrders(10);

        System.out.println("\nOriginal Orders:");

        for (Order order : orders) {

            System.out.println("  " + order);

        }

        // Demonstrate Bubble Sort

        System.out.println("\n" + "=".repeat(50));

        System.out.println("BUBBLE SORT RESULTS:");

        System.out.println("=".repeat(50));

        PerformanceAnalyzer.SortingResult bubbleResult = PerformanceAnalyzer.measureSortingTime(

            OrderSorter::bubbleSort, orders

        );

        for (Order order : bubbleResult.sortedOrders) {

            System.out.println("  " + order);

        }

        System.out.printf("Execution Time: %.6f seconds%n", bubbleResult.getExecutionTimeInSeconds());

        System.out.println("Correctly Sorted: " + PerformanceAnalyzer.verifySorting(bubbleResult.sortedOrders));

        // Demonstrate Quick Sort

        System.out.println("\n" + "=".repeat(50));

        System.out.println("QUICK SORT RESULTS:");

        System.out.println("=".repeat(50));

        PerformanceAnalyzer.SortingResult quickResult = PerformanceAnalyzer.measureSortingTime(

            OrderSorter::quickSort, orders

        );

        for (Order order : quickResult.sortedOrders) {

            System.out.println("  " + order);

        }

        System.out.printf("Execution Time: %.6f seconds%n", quickResult.getExecutionTimeInSeconds());

        System.out.println("Correctly Sorted: " + PerformanceAnalyzer.verifySorting(quickResult.sortedOrders));

        // Performance comparison with larger dataset

        System.out.println("\n" + "=".repeat(50));

        System.out.println("PERFORMANCE ANALYSIS");

        System.out.println("=".repeat(50));

        // Test with different sizes

        int[] sizes = {100, 500, 1000};

        for (int size : sizes) {

            List<Order> testOrders = PerformanceAnalyzer.generateSampleOrders(size);

            PerformanceAnalyzer.compareAlgorithms(testOrders);

        }

    }

    public static void main(String[] args) {

        // Run the demonstration

        explainAlgorithms();

        demonstrateSorting();

        whyQuickSortPreferred();

        // Interactive section

        System.out.println("\n" + "=".repeat(50));

        System.out.println("INTERACTIVE TESTING");

        System.out.println("=".repeat(50));

        System.out.println("You can now test the algorithms with custom data!");

        System.out.println("Example usage:");

        System.out.println("List<Order> orders = PerformanceAnalyzer.generateSampleOrders(50);");

        System.out.println("List<Order> sortedOrders = OrderSorter.quickSort(orders);");

        System.out.println("PerformanceAnalyzer.compareAlgorithms(orders);");

        // Demonstrate with user input simulation

        Scanner scanner = new Scanner(System.in);

        System.out.println("\nWould you like to test with a custom dataset size? (y/n):");

        try {

            String input = scanner.nextLine();

            if (input.toLowerCase().startsWith("y")) {

                System.out.print("Enter the number of orders to generate: ");

                int customSize = scanner.nextInt();

                if (customSize > 0 && customSize <= 10000) {

                    List<Order> customOrders = PerformanceAnalyzer.generateSampleOrders(customSize);

                    System.out.println("\nTesting with " + customSize + " orders:");

                    PerformanceAnalyzer.compareAlgorithms(customOrders);

                } else {

                    System.out.println("Please enter a number between 1 and 10000.");

                }

            }

        } catch (Exception e) {

            System.out.println("Invalid input. Ending program.");

        } finally {

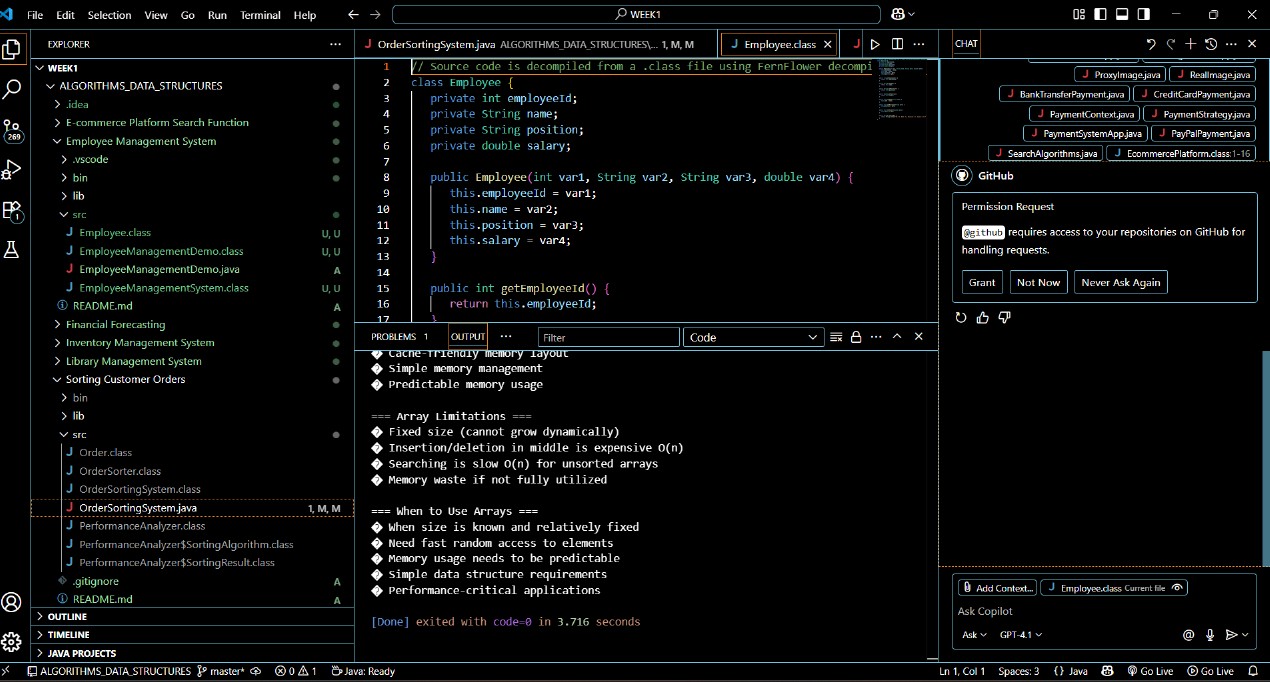
            scanner.close();

        }

        System.out.println("\nThank you for using the Customer Order Sorting System!");

    }

}



**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.

Task managementsytem.java

public class TaskManagementSystem {

    public static void main(String[] args) {

        TaskLinkedList taskList = new TaskLinkedList();

        System.out.println("=== Task Management System Demo ===\n");

        // Adding tasks

        System.out.println("1. Adding Tasks:");

        taskList.addTask(101, "Complete Project Documentation", "In Progress");

        taskList.addTask(102, "Review Code", "Pending");

        taskList.addTask(103, "Fix Bug #234", "Completed");

        taskList.addTaskAtEnd(104, "Prepare Presentation", "Not Started");

        taskList.addTaskAtEnd(105, "Update Database", "In Progress");

        // Traversing tasks

        System.out.println("\n2. Traversing All Tasks:");

        taskList.traverseTasks();

        // Searching tasks

        System.out.println("\n3. Searching Tasks:");

        Task foundTask = taskList.searchTask(103);

        if (foundTask != null) {

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

        } else {

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

        }

        // Search by status

        taskList.searchTasksByStatus("In Progress");

        // Updating task status

        System.out.println("\n4. Updating Task Status:");

        taskList.updateTaskStatus(102, "Completed");

        taskList.updateTaskStatus(104, "In Progress");

        // Display updated list

        taskList.traverseTasks();

        // Deleting tasks

        System.out.println("\n5. Deleting Tasks:");

        taskList.deleteTask(103);

        taskList.deleteTask(999); // Non-existent task

        // Final list

        System.out.println("\n6. Final Task List:");

        taskList.traverseTasks();

        // Performance demonstration

        System.out.println("\n7. Performance Info:");

        System.out.println("Total tasks: " + taskList.getSize());

        System.out.println("Is empty: " + taskList.isEmpty());

        // Clear all tasks

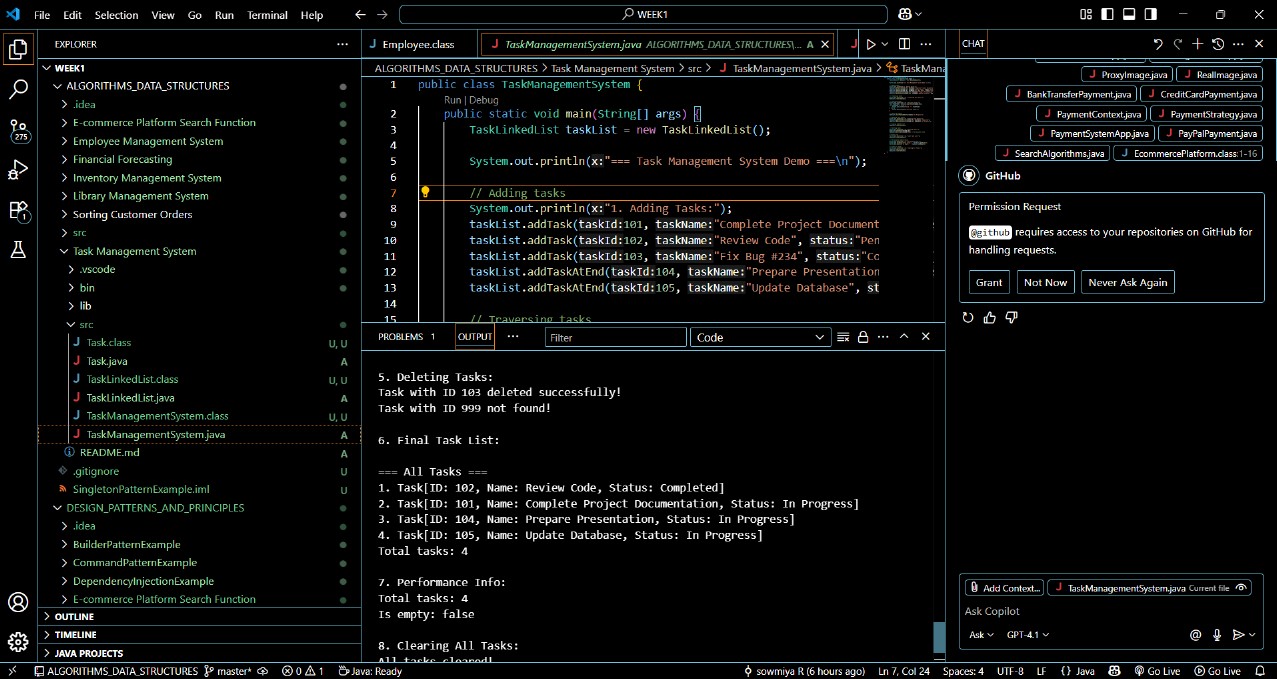
        System.out.println("\n8. Clearing All Tasks:");

        taskList.clearAllTasks();

        taskList.traverseTasks();

    }

}



**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.

LibraryDemo.java

public class LibraryDemo {

    public static void main(String[] args) {

        LibraryManagementSystem library = new LibraryManagementSystem();

        System.out.println("=== LIBRARY MANAGEMENT SYSTEM DEMO ===\n");

        // Adding sample books

        System.out.println("1. ADDING BOOKS TO LIBRARY:");

        System.out.println("-".repeat(40));

        library.addBook(new Book(1, "To Kill a Mockingbird", "Harper Lee", "Fiction", 1960));

        library.addBook(new Book(2, "1984", "George Orwell", "Dystopian Fiction", 1949));

        library.addBook(new Book(3, "Pride and Prejudice", "Jane Austen", "Romance", 1813));

        library.addBook(new Book(4, "The Great Gatsby", "F. Scott Fitzgerald", "Fiction", 1925));

        library.addBook(new Book(5, "Animal Farm", "George Orwell", "Political Satire", 1945));

        library.addBook(new Book(6, "Lord of the Flies", "William Golding", "Fiction", 1954));

        library.addBook(new Book(7, "Brave New World", "Aldous Huxley", "Science Fiction", 1932));

        library.addBook(new Book(8, "The Catcher in the Rye", "J.D. Salinger", "Fiction", 1951));

        library.addBook(new Book(9, "Of Mice and Men", "John Steinbeck", "Fiction", 1937));

        library.addBook(new Book(10, "The Hobbit", "J.R.R. Tolkien", "Fantasy", 1937));

        // Display all books

        library.displayAllBooks();

        // LINEAR SEARCH DEMONSTRATIONS

        System.out.println("\n2. LINEAR SEARCH DEMONSTRATIONS:");

        System.out.println("-".repeat(40));

        // Search by title

        SearchResult result1 = library.linearSearchByTitle("1984");

        library.displaySearchResults(result1, "Title: '1984'");

        // Search by author

        SearchResult result2 = library.linearSearchByAuthor("George Orwell");

        library.displaySearchResults(result2, "Author: 'George Orwell'");

        // BINARY SEARCH DEMONSTRATIONS

        System.out.println("\n3. BINARY SEARCH DEMONSTRATIONS:");

        System.out.println("-".repeat(40));

        // Search by title

        SearchResult result3 = library.binarySearchByTitle("The Great Gatsby");

        library.displaySearchResults(result3, "Title: 'The Great Gatsby'");

        // Search by author

        SearchResult result4 = library.binarySearchByAuthor("Jane Austen");

        library.displaySearchResults(result4, "Author: 'Jane Austen'");

        // PERFORMANCE COMPARISON

        System.out.println("\n4. PERFORMANCE COMPARISON:");

        System.out.println("-".repeat(40));

        library.compareSearchPerformance("Animal Farm");

        library.compareSearchPerformance("Non-existent Book");

        // ADVANCED SEARCH

        System.out.println("\n5. ADVANCED SEARCH DEMONSTRATION:");

        System.out.println("-".repeat(40));

        SearchResult advancedResult = library.advancedSearch(null, "George Orwell", null, null);

        library.displaySearchResults(advancedResult, "All books by George Orwell");

        // Search by genre

        SearchResult genreResult = library.advancedSearch(null, null, "Fiction", null);

        library.displaySearchResults(genreResult, "All Fiction books");

        // ALGORITHM ANALYSIS

        System.out.println("\n6. ALGORITHM ANALYSIS:");

        System.out.println("-".repeat(40));

        printAlgorithmAnalysis();

        // INTERACTIVE DEMONSTRATIONS

        System.out.println("\n7. INTERACTIVE DEMONSTRATIONS:");

        System.out.println("-".repeat(40));

        demonstrateWorstCase(library);

        demonstrateBestCase(library);

    }

    private static void printAlgorithmAnalysis() {

        System.out.println("LINEAR SEARCH:");

        System.out.println("• Time Complexity: O(n)");

        System.out.println("• Space Complexity: O(1)");

        System.out.println("• Works on unsorted data");

        System.out.println("• Best for small datasets or unsorted data");

        System.out.println("• Guaranteed to find all matches");

        System.out.println("\nBINARY SEARCH:");

        System.out.println("• Time Complexity: O(log n)");

        System.out.println("• Space Complexity: O(1)");

        System.out.println("• Requires sorted data");

        System.out.println("• Best for large, sorted datasets");

        System.out.println("• Much faster for large datasets");

        System.out.println("\nWHEN TO USE EACH:");

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

        System.out.println("• Small datasets (< 100 items)");

        System.out.println("• Unsorted data");

        System.out.println("• When you need to find all occurrences");

        System.out.println("• When sorting cost is too high");

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

        System.out.println("• Large datasets (> 1000 items)");

        System.out.println("• When data is already sorted");

        System.out.println("• When search operations are frequent");

        System.out.println("• When performance is critical");

    }

    private static void demonstrateWorstCase(LibraryManagementSystem library) {

        System.out.println("\nWORST CASE SCENARIOS:");

        System.out.println("• Linear Search: Target is the last element or doesn't exist");

        System.out.println("• Binary Search: Requires maximum log(n) comparisons");

        // Demonstrate searching for non-existent book (worst case for linear)

        SearchResult worstLinear = library.linearSearchByTitle("Non-existent Book");

        SearchResult worstBinary = library.binarySearchByTitle("Non-existent Book");

        System.out.println("\nSearching for 'Non-existent Book':");

        System.out.printf("Linear Search: %d comparisons%n", worstLinear.getComparisons());

        System.out.printf("Binary Search: %d comparisons%n", worstBinary.getComparisons());

    }

    private static void demonstrateBestCase(LibraryManagementSystem library) {

        System.out.println("\nBEST CASE SCENARIOS:");

        System.out.println("• Linear Search: Target is the first element");

        System.out.println("• Binary Search: Target is the middle element");

        // For demonstration, we'll search for the first book added

        SearchResult bestLinear = library.linearSearchByTitle("The Hobbit"); // Last added, first in list

        SearchResult bestBinary = library.binarySearchByTitle("The Hobbit");

        System.out.println("\nSearching for 'The Hobbit' (first in unsorted list):");

        System.out.printf("Linear Search: %d comparisons%n", bestLinear.getComparisons());

        System.out.printf("Binary Search: %d comparisons%n", bestBinary.getComparisons());

    }

    public static void demonstrateScalability() {

        System.out.println("\n=== SCALABILITY DEMONSTRATION ===");

        System.out.println("Dataset Size | Linear Search | Binary Search | Efficiency Gain");

        System.out.println("-------------|---------------|---------------|----------------");

        int[] sizes = {10, 100, 1000, 10000, 100000};

        for (int size : sizes) {

            int linearComparisons = size; // Worst case for linear

            int binaryComparisons = (int) Math.ceil(Math.log(size) / Math.log(2)); // Worst case for binary

            double efficiency = (double) linearComparisons / binaryComparisons;

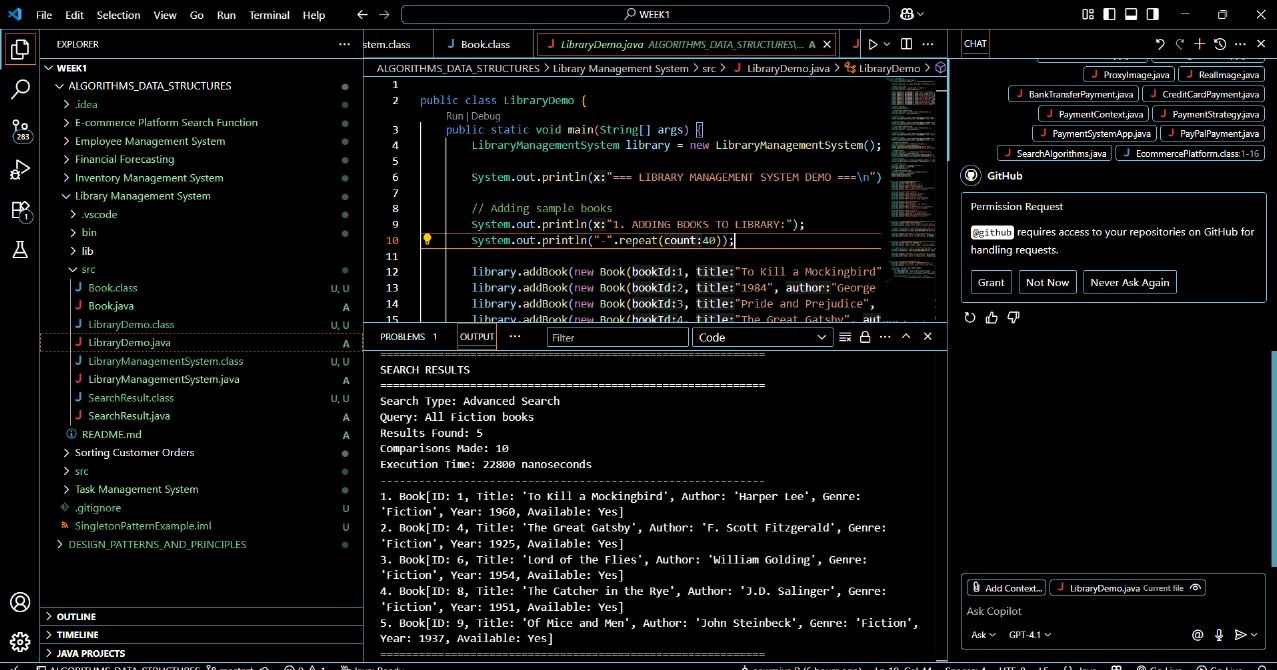
            System.out.printf("%-12d | %-13d | %-13d | %.1fx%n",

                            size, linearComparisons, binaryComparisons, efficiency);

        }

    }

}



**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.

FinancialForecastingDemo.java

import java.util.\*;

import java.text.DecimalFormat;

// Class to represent financial data points

class FinancialDataPoint {

    private int period;

    private double value;

    private double growthRate;

    public FinancialDataPoint(int period, double value, double growthRate) {

        this.period = period;

        this.value = value;

        this.growthRate = growthRate;

    }

    // Getters

    public int getPeriod() { return period; }

    public double getValue() { return value; }

    public double getGrowthRate() { return growthRate; }

    @Override

    public String toString() {

        return String.format("Period %d: $%.2f (Growth: %.2f%%)",

                           period, value, growthRate \* 100);

    }

}

// Main Financial Forecasting System

class FinancialForecastingSystem {

    private static final DecimalFormat df = new DecimalFormat("#,##0.00");

    private Map<String, Double> memoizationCache = new HashMap<>();

    private int recursiveCallCount = 0;

    private int memoizedCallCount = 0;

    public double calculateCompoundInterestRecursive(double principal, double rate, int years) {

        recursiveCallCount++;

        // Base case: no more years

        if (years == 0) {

            return principal;

        }

        // Recursive case: calculate for (years - 1) and apply interest

        return calculateCompoundInterestRecursive(principal \* (1 + rate), rate, years - 1);

    }

    public double calculateCompoundInterestMemoized(double principal, double rate, int years) {

        String key = principal + "," + rate + "," + years;

        if (memoizationCache.containsKey(key)) {

            return memoizationCache.get(key);

        }

        memoizedCallCount++;

        // Base case

        if (years == 0) {

            memoizationCache.put(key, principal);

            return principal;

        }

        // Recursive case with memoization

        double result = calculateCompoundInterestMemoized(principal \* (1 + rate), rate, years - 1);

        memoizationCache.put(key, result);

        return result;

    }

    public double fibonacciGrowthPrediction(double baseValue, int period) {

        recursiveCallCount++;

        // Base cases

        if (period <= 0) return baseValue;

        if (period == 1) return baseValue \* 1.1; // 10% growth

        if (period == 2) return baseValue \* 1.2; // 20% growth

        // Recursive case: growth follows Fibonacci pattern

        return fibonacciGrowthPrediction(baseValue, period - 1) \* 0.6 +

               fibonacciGrowthPrediction(baseValue, period - 2) \* 0.4;

    }

    private Map<Integer, Double> fibCache = new HashMap<>();

    public double fibonacciGrowthMemoized(double baseValue, int period) {

        if (fibCache.containsKey(period)) {

            return fibCache.get(period);

        }

        memoizedCallCount++;

        double result;

        if (period <= 0) {

            result = baseValue;

        } else if (period == 1) {

            result = baseValue \* 1.1;

        } else if (period == 2) {

            result = baseValue \* 1.2;

        } else {

            result = fibonacciGrowthMemoized(baseValue, period - 1) \* 0.6 +

                    fibonacciGrowthMemoized(baseValue, period - 2) \* 0.4;

        }

        fibCache.put(period, result);

        return result;

    }

    public double predictFutureTrend(double[] historicalData, int currentIndex, double trendFactor, int periodsAhead) {

        recursiveCallCount++;

        // Base case: no more periods to predict

        if (periodsAhead == 0) {

            return historicalData[currentIndex];

        }

        // Calculate growth rate from historical data

        double growthRate = 0;

        if (currentIndex > 0) {

            growthRate = (historicalData[currentIndex] - historicalData[currentIndex - 1])

                        / historicalData[currentIndex - 1];

        }

        // Apply trend factor and recurse

        double nextValue = historicalData[currentIndex] \* (1 + growthRate \* trendFactor);

        // Create new array with predicted value

        double[] newData = Arrays.copyOf(historicalData, historicalData.length + 1);

        newData[newData.length - 1] = nextValue;

        return predictFutureTrend(newData, newData.length - 1, trendFactor, periodsAhead - 1);

    }

    public double calculatePortfolioGrowth(double currentValue, double monthlyInvestment ,double monthlyReturn, int monthsRemaining) {

        recursiveCallCount++;

        // Base case: no more months

        if (monthsRemaining == 0) {

            return currentValue;

        }

        // Add monthly investment and apply return

        double newValue = (currentValue + monthlyInvestment) \* (1 + monthlyReturn);

        // Recurse for remaining months

        return calculatePortfolioGrowth(newValue, monthlyInvestment, monthlyReturn, monthsRemaining - 1);

    }

    public double calculateLoanBalance(double currentBalance, double monthlyPayment,

                                     double monthlyInterestRate, int paymentsRemaining) {

        recursiveCallCount++;

        // Base case: no more payments or balance is zero

        if (paymentsRemaining == 0 || currentBalance <= 0) {

            return Math.max(0, currentBalance);

        }

        // Apply interest and subtract payment

        double newBalance = currentBalance \* (1 + monthlyInterestRate) - monthlyPayment;

        // Recurse for remaining payments

        return calculateLoanBalance(newBalance, monthlyPayment, monthlyInterestRate, paymentsRemaining - 1);

    }

    public double calculatePresentValue(double[] futureCashFlows, double discountRate, int index) {

        recursiveCallCount++;

        // Base case: processed all cash flows

        if (index >= futureCashFlows.length) {

            return 0;

        }

        // Calculate present value of current cash flow

        double presentValueCurrent = futureCashFlows[index] / Math.pow(1 + discountRate, index + 1);

        // Add present value of remaining cash flows

        return presentValueCurrent + calculatePresentValue(futureCashFlows, discountRate, index + 1);

    }

    public double calculateCompoundInterestTailRecursive(double principal, double rate,

                                                       int years, double accumulator) {

        recursiveCallCount++;

        // Base case

        if (years == 0) {

            return accumulator;

        }

        // Tail recursive call

        return calculateCompoundInterestTailRecursive(principal, rate, years - 1,

                                                    accumulator \* (1 + rate));

    }

    // Helper method for tail recursive compound interest

    public double calculateCompoundInterestTailRecursive(double principal, double rate, int years) {

        return calculateCompoundInterestTailRecursive(principal, rate, years, principal);

    }

    // Utility methods

    public void resetCounters() {

        recursiveCallCount = 0;

        memoizedCallCount = 0;

        memoizationCache.clear();

        fibCache.clear();

    }

    public int getRecursiveCallCount() { return recursiveCallCount; }

    public int getMemoizedCallCount() { return memoizedCallCount; }

    public void generateForecastReport(double principal, double rate, int years) {

        System.out.println("\n=== FINANCIAL FORECAST REPORT ===");

        System.out.println("Principal Amount: $" + df.format(principal));

        System.out.println("Annual Interest Rate: " + (rate \* 100) + "%");

        System.out.println("Investment Period: " + years + " years");

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

        // Year-by-year breakdown

        System.out.println("\nYear-by-Year Growth:");

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

        double currentValue = principal;

        for (int year = 1; year <= years; year++) {

            currentValue \*= (1 + rate);

            System.out.printf("Year %2d: $%s\n", year, df.format(currentValue));

        }

        System.out.println("\nFinal Value: $" + df.format(currentValue));

        System.out.println("Total Growth: $" + df.format(currentValue - principal));

        System.out.println("Growth Percentage: " +

                          df.format(((currentValue - principal) / principal) \* 100) + "%");

    }

}

// Demo class to showcase the Financial Forecasting System

public class FinancialForecastingDemo {

    public static void main(String[] args) {

        FinancialForecastingSystem ffs = new FinancialForecastingSystem();

        DecimalFormat df = new DecimalFormat("#,##0.00");

        System.out.println("=== FINANCIAL FORECASTING SYSTEM ===");

        System.out.println("Demonstrating Recursive Financial Algorithms\n");

        // 1. Compound Interest Calculation

        System.out.println("1. COMPOUND INTEREST CALCULATION");

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

        double principal = 10000;

        double rate = 0.08; // 8% annual rate

        int years = 10;

        ffs.resetCounters();

        double futureValue = ffs.calculateCompoundInterestRecursive(principal, rate, years);

        System.out.println("Basic Recursive Approach:");

        System.out.println("Principal: $" + df.format(principal));

        System.out.println("Rate: " + (rate \* 100) + "% annually");

        System.out.println("Years: " + years);

        System.out.println("Future Value: $" + df.format(futureValue));

        System.out.println("Recursive Calls: " + ffs.getRecursiveCallCount());

        // Memoized version

        ffs.resetCounters();

        double memoizedValue = ffs.calculateCompoundInterestMemoized(principal, rate, years);

        System.out.println("\nMemoized Recursive Approach:");

        System.out.println("Future Value: $" + df.format(memoizedValue));

        System.out.println("Memoized Calls: " + ffs.getMemoizedCallCount());

        // Tail recursive version

        ffs.resetCounters();

        double tailRecursiveValue = ffs.calculateCompoundInterestTailRecursive(principal, rate, years);

        System.out.println("\nTail Recursive Approach:");

        System.out.println("Future Value: $" + df.format(tailRecursiveValue));

        System.out.println("Recursive Calls: " + ffs.getRecursiveCallCount());

        // 2. Fibonacci Growth Prediction

        System.out.println("\n\n2. FIBONACCI GROWTH PREDICTION");

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

        double baseValue = 1000;

        int periods = 8;

        ffs.resetCounters();

        double fibGrowth = ffs.fibonacciGrowthPrediction(baseValue, periods);

        System.out.println("Basic Fibonacci Growth (Period " + periods + "):");

        System.out.println("Base Value: $" + df.format(baseValue));

        System.out.println("Predicted Value: $" + df.format(fibGrowth));

        System.out.println("Recursive Calls: " + ffs.getRecursiveCallCount());

        ffs.resetCounters();

        double fibMemoized = ffs.fibonacciGrowthMemoized(baseValue, periods);

        System.out.println("\nMemoized Fibonacci Growth:");

        System.out.println("Predicted Value: $" + df.format(fibMemoized));

        System.out.println("Memoized Calls: " + ffs.getMemoizedCallCount());

        // 3. Trend Analysis

        System.out.println("\n\n3. TREND ANALYSIS PREDICTION");

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

        double[] historicalData = {1000, 1100, 1250, 1400, 1600, 1850};

        int periodsAhead = 3;

        double trendFactor = 0.8;

        ffs.resetCounters();

        double trendPrediction = ffs.predictFutureTrend(historicalData,

                                                       historicalData.length - 1,

                                                       trendFactor, periodsAhead);

        System.out.println("Historical Data: " + Arrays.toString(historicalData));

        System.out.println("Periods Ahead: " + periodsAhead);

        System.out.println("Trend Factor: " + trendFactor);

        System.out.println("Predicted Value: $" + df.format(trendPrediction));

        System.out.println("Recursive Calls: " + ffs.getRecursiveCallCount());

        // 4. Portfolio Growth

        System.out.println("\n\n4. PORTFOLIO GROWTH CALCULATION");

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

        double currentValue = 5000;

        double monthlyInvestment = 500;

        double monthlyReturn = 0.007; // 0.7% monthly

        int months = 24;

        ffs.resetCounters();

        double portfolioValue = ffs.calculatePortfolioGrowth(currentValue, monthlyInvestment,

                                                           monthlyReturn, months);

        System.out.println("Initial Portfolio Value: $" + df.format(currentValue));

        System.out.println("Monthly Investment: $" + df.format(monthlyInvestment));

        System.out.println("Monthly Return: " + (monthlyReturn \* 100) + "%");

        System.out.println("Investment Period: " + months + " months");

        System.out.println("Final Portfolio Value: $" + df.format(portfolioValue));

        System.out.println("Total Growth: $" + df.format(portfolioValue - currentValue - (monthlyInvestment \* months)));

        System.out.println("Recursive Calls: " + ffs.getRecursiveCallCount());

        // 5. Loan Balance Calculation

        System.out.println("\n\n5. LOAN BALANCE CALCULATION");

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

        double loanBalance = 50000;

        double monthlyPayment = 800;

        double monthlyInterestRate = 0.004; // 0.4% monthly (4.8% annually)

        int payments = 60;

        ffs.resetCounters();

        double remainingBalance = ffs.calculateLoanBalance(loanBalance, monthlyPayment,

                                                         monthlyInterestRate, payments);

        System.out.println("Initial Loan Balance: $" + df.format(loanBalance));

        System.out.println("Monthly Payment: $" + df.format(monthlyPayment));

        System.out.println("Monthly Interest Rate: " + (monthlyInterestRate \* 100) + "%");

        System.out.println("Number of Payments: " + payments);

        System.out.println("Remaining Balance: $" + df.format(remainingBalance));

        System.out.println("Recursive Calls: " + ffs.getRecursiveCallCount());

        // 6. Present Value Calculation

        System.out.println("\n\n6. PRESENT VALUE CALCULATION");

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

        double[] futureCashFlows = {1000, 1500, 2000, 2500, 3000};

        double discountRate = 0.10; // 10% discount rate

        ffs.resetCounters();

        double presentValue = ffs.calculatePresentValue(futureCashFlows, discountRate, 0);

        System.out.println("Future Cash Flows: " + Arrays.toString(futureCashFlows));

        System.out.println("Discount Rate: " + (discountRate \* 100) + "%");

        System.out.println("Present Value: $" + df.format(presentValue));

        System.out.println("Recursive Calls: " + ffs.getRecursiveCallCount());

        // 7. Performance Comparison

        System.out.println("\n\n7. PERFORMANCE COMPARISON");

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

        System.out.println("Fibonacci Growth Comparison (Period 10):");

        // Regular Fibonacci

        long startTime = System.nanoTime();

        ffs.resetCounters();

        ffs.fibonacciGrowthPrediction(1000, 10);

        long regularTime = System.nanoTime() - startTime;

        int regularCalls = ffs.getRecursiveCallCount();

        // Memoized Fibonacci

        startTime = System.nanoTime();

        ffs.resetCounters();

        ffs.fibonacciGrowthMemoized(1000, 10);

        long memoizedTime = System.nanoTime() - startTime;

        int memoizedCalls = ffs.getMemoizedCallCount();

        System.out.println("Regular Fibonacci:");

        System.out.println("  Time: " + regularTime / 1000 + " microseconds");

        System.out.println("  Calls: " + regularCalls);

        System.out.println("Memoized Fibonacci:");

        System.out.println("  Time: " + memoizedTime / 1000 + " microseconds");

        System.out.println("  Calls: " + memoizedCalls);

        System.out.println("Improvement: " + (regularTime / (double) memoizedTime) + "x faster");

        // 8. Generate detailed report

        ffs.generateForecastReport(10000, 0.08, 10);

        // 9. Algorithm Analysis

        System.out.println("\n\n=== ALGORITHM ANALYSIS ===");

        System.out.println("Time Complexity Analysis:");

        System.out.println("• Compound Interest (Basic): O(n) - Linear recursion");

        System.out.println("• Compound Interest (Memoized): O(n) - Each subproblem solved once");

        System.out.println("• Fibonacci Growth (Basic): O(2^n) - Exponential due to overlapping subproblems");

        System.out.println("• Fibonacci Growth (Memoized): O(n) - Dynamic programming optimization");

        System.out.println("• Trend Analysis: O(n) - Linear recursion");

        System.out.println("• Portfolio Growth: O(n) - Linear recursion");

        System.out.println("• Present Value: O(n) - Linear recursion");

        System.out.println("\nSpace Complexity Analysis:");

        System.out.println("• All recursive algorithms: O(n) - Due to call stack");

        System.out.println("• Memoized versions: O(n) - Additional cache storage");

        System.out.println("• Tail recursive: O(1) - Optimized by compiler");

        System.out.println("\nOptimization Techniques:");

        System.out.println("• Memoization: Cache results to avoid recalculation");

        System.out.println("• Tail Recursion: Enable compiler optimization");

        System.out.println("• Iterative Conversion: Replace recursion with loops");

        System.out.println("• Dynamic Programming: Bottom-up approach");

        System.out.println("\nWhen to Use Recursive Approaches:");

        System.out.println("✓ Problem has optimal substructure");

        System.out.println("✓ Natural recursive definition exists");

        System.out.println("✓ Overlapping subproblems (use memoization)");

        System.out.println("✓ Tree-like problem structure");

        System.out.println("✗ Simple iterative solution exists");

        System.out.println("✗ Deep recursion with stack overflow risk");

        System.out.println("✗ No overlapping subproblems");

    }

}

