**Exercise 1: Inventory Management System**

Q1. Explain why data structures and algorithms are essential in handling large inventories?

Ans.

* Efficiently managing large quantities of products requires data structures that can store and retrieve information quickly.
* Adding, updating, deleting, and searching for products involve algorithms that determine the system's performance.
* Inventory levels need to be accurate and up-to-date, demanding efficient data manipulation.

Q2. Discuss the types of data structures suitable for this problem?

Ans.

* **ArrayList**: Useful for maintaining a dynamic list of products where the order of insertion is maintained. It allows fast random access and is easy to implement.
* **HashMap**: Suitable for scenarios where fast lookup, insertion, and deletion are needed. It provides average-case constant time complexity for these operations.

**Implementation: JAVA Code**

**Product.java**

*public class* Product {  
 *private* String productId;  
 *private* String productName;  
 *private int* quantity;  
 *private double* price;  
  
 *public* Product(String productId, String productName, *int* quantity, *double* price) {  
 *this*.productId = productId;  
 *this*.productName = productName;  
 *this*.quantity = quantity;  
 *this*.price = price;  
 }  
  
 *// Getters and setters  
 public* String getProductId() {  
 *return* productId;  
 }  
 *public void* setProductId(String productId) {  
 *this*.productId = productId;  
 }  
  
 *public* String getProductName() {  
 *return* productName;  
 }  
 *public void* setProductName(String productName) {  
 *this*.productName = productName;  
 }  
  
 *public int* getQuantity() {  
 *return* quantity;  
 }  
 *public void* setQuantity(*int* quantity) {  
 *this*.quantity = quantity;  
 }  
  
 *public double* getPrice() {  
 *return* price;  
 }  
 *public void* setPrice(*double* price) {  
 *this*.price = price;  
 }  
}

**InventoryManagementSystem.java**

*import* java.util.HashMap;  
  
*public class* InventoryManagementSystem {  
 *private* HashMap<String, Product> inventory;  
  
 *public* InventoryManagementSystem() {  
 inventory = *new* HashMap<>();  
 }  
  
 *// 1.Method to add a product  
 public void* addProduct(Product product) {  
 inventory.put(product.getProductId(), product);  
 }  
  
 *// 2.Method to update a product  
 public void* updateProduct(String productId, Product updatedProduct) {  
 *if* (inventory.containsKey(productId)) {  
 inventory.put(productId, updatedProduct);  
 } *else* {  
 System.***out***.println("Product not found");  
 }  
 }  
  
 *// 3.Method to delete a product  
 public void* deleteProduct(String productId) {  
 *if* (inventory.containsKey(productId)) {  
 inventory.remove(productId);  
 } *else* {  
 System.***out***.println("Product not found");  
 }  
 }  
  
 *// 4.Method to get the inventory  
 public* HashMap<String, Product> getInventory() {  
 *return* inventory;  
 }  
}

**Main.java**

*public class* Main {  
 *public static void* main(String[] args) {  
 *// Create an instance of the InventoryManagementSystem* InventoryManagementSystem inventorySystem = *new* InventoryManagementSystem();  
  
 *// Create some products* Product product1 = *new* Product("P1", "DellGamingLaptop", 10, 200000);  
 Product product2 = *new* Product("P2", "Smartphone", 20, 99999);  
 Product product3 = *new* Product("P3", "Tablet", 15, 89999);  
  
 *// Add products to the inventory* inventorySystem.addProduct(product1);  
 inventorySystem.addProduct(product2);  
 inventorySystem.addProduct(product3);  
  
 *// Display the inventory after adding products* System.***out***.println("Inventory after adding products:");  
 *displayInventory*(inventorySystem);  
  
 *// Update a product in the inventory* Product updatedProduct = *new* Product("P2", "Smartphone", 25, 99999);  
 inventorySystem.updateProduct("P2", updatedProduct);  
  
 *// Display the inventory after updating a product* System.***out***.println("\nInventory after updating product P002:");  
 *displayInventory*(inventorySystem);  
  
 *// Delete a product from the inventory* inventorySystem.deleteProduct("P1");  
  
 *// Display the inventory after deleting a product* System.***out***.println("\nInventory after deleting product P1:");  
 *displayInventory*(inventorySystem);  
 }  
  
 *// Method to display the inventory  
 public static void* displayInventory(InventoryManagementSystem inventorySystem) {  
 *for* (Product product : inventorySystem.getInventory().values()) {  
 System.***out***.println("Product ID: " + product.getProductId() +  
 ", Name: " + product.getProductName() +  
 ", Quantity: " + product.getQuantity() +  
 ", Price: ₹" + product.getPrice());  
 }  
 }  
}

Q3. Analyze the time complexity of each operation (add, update, delete) in your chosen data structure?

Ans.

* **Add Operation**: The time complexity for adding a product in a HashMap is O(1) on average. This is because it involves calculating the hash code of the productId and placing it in the appropriate bucket.
* **Update Operation**: The time complexity for updating a product is also O(1) on average, as it involves a lookup followed by an update.
* **Delete Operation**: The time complexity for deleting a product is O(1) on average, as it involves a lookup followed by a removal.

Q4. Discuss how you can optimize these operations?

Ans.

* **Choose appropriate hash function:** For HashMap, a good hash function can minimize collisions and improve performance.
* **Consider data structure size:** If the inventory is extremely large, explore specialized data structures like Tries or Bloom filters for faster search operations.
* **Batch updates:** For frequent updates, consider batching them to improve efficiency.
* **Indexing:** Create additional indexes (e.g., by product name) for faster search based on specific criteria.

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

Q1. Explain Big O notation and how it helps in analyzing algorithms?

Ans.

* Big O notation is a mathematical notation used to describe the upper bound of an algorithm's running time. It provides an approximation of the algorithm's time complexity in terms of input size (n), focusing on the worst-case scenario.
* It helps in analyzing the efficiency of algorithms by allowing comparisons of their performance and scalability.

Q2. Describe the best, average, and worst-case scenarios for search operations?

Ans.

* **Best Case**: The scenario where the algorithm performs the minimum number of operations. For example, in a search operation, the best case is finding the target element in the first position.
* **Average Case**: The expected performance of the algorithm over all possible inputs. It provides a realistic measure of the algorithm's efficiency.
* **Worst Case**: The scenario where the algorithm performs the maximum number of operations. For example, in a search operation, the worst case is not finding the target element or finding it at the last position.

**Implementation: JAVA Code**

*import* java.util.Arrays;  
*import* java.util.*Comparator*;  
*class* Product {  
 *private* String productId;  
 *private* String productName;  
 *private* String category;  
  
 *public* Product(String productId, String productName, String category) {  
 *this*.productId = productId;  
 *this*.productName = productName;  
 *this*.category = category;  
 }  
  
 *public* String getProductId() {  
 *return* productId;  
 }  
  
 *public* String getProductName() {  
 *return* productName;  
 }  
  
 *public* String getCategory() {  
 *return* category;  
 }  
  
 @Override  
 *public* String toString() {  
 *return* "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;  
 }  
}  
  
*class* SearchAlgorithms {  
 *public static* Product linearSearch(Product[] products, String targetId) {  
 *for* (Product product : products) {  
 *if* (product.getProductId().equals(targetId)) {  
 *return* product;  
 }  
 }  
 *return null*;  
 }  
  
 *public static* Product binarySearch(Product[] products, String targetId) {  
 Arrays.*sort*(products, *Comparator*.*comparing*(Product::getProductId));  
 *int* left = 0;  
 *int* right = products.length - 1;  
  
 *while* (left <= right) {  
 *int* mid = left + (right - left) / 2;  
 *int* cmp = products[mid].getProductId().compareTo(targetId);  
 *if* (cmp == 0) {  
 *return* products[mid];  
 } *else if* (cmp < 0) {  
 left = mid + 1;  
 } *else* {  
 right = mid - 1;  
 }  
 }  
 *return null*;  
 }  
}  
*public class* Main {  
 *public static void* main(String[] args) {  
 *// Create an array of products* Product[] products = {  
 *new* Product("P001", "Laptop", "Electronics"),  
 *new* Product("P002", "Smartphone", "Electronics"),  
 *new* Product("P003", "Tablet", "Electronics"),  
 *new* Product("P004", "Headphones", "Accessories"),  
 *new* Product("P005", "Camera", "Electronics")  
 };  
  
 *// Linear search example* Product foundProduct = SearchAlgorithms.*linearSearch*(products, "P003");  
 System.***out***.println("Linear Search Result: " + (foundProduct != *null* ? foundProduct : "Product not found"));  
  
 *// Binary search example* foundProduct = SearchAlgorithms.*binarySearch*(products, "P003");  
 System.***out***.println("Binary Search Result: " + (foundProduct != *null* ? foundProduct : "Product not found"));  
 }  
}

Q3. Compare the time complexity of linear and binary search algorithms?

Ans.

* Linear Search
  + **Best Case**: O(1) (element found at the first position)
  + **Average Case**: O(n) (element found somewhere in the middle or not at all)
  + **Worst Case**: O(n) (element found at the last position or not at all)
* Binary Search
  + **Best Case**: O(1) (element found at the middle position)
  + **Average Case**: O(log n) (element found after a few iterations)
  + **Worst Case**: O(log n) (element not found after several iterations)

Q4. Discuss which algorithm is more suitable for your platform and why?

Ans. **Binary search** is more suitable if the product array is sorted.

**Exercise 3: Sorting Customer Orders**

Q1. Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort)?

Ans.

1. Bubble Sort:
   * **Description**: Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.
   * **Time Complexity**:
     + Best Case: O(n) (when the list is already sorted)
     + Average Case: O(n²)
     + Worst Case: O(n²)
   * **Space Complexity**: O(1) (in-place sort)
2. **Insertion Sort**:
   * **Description**: Insertion Sort builds the sorted array one item at a time by repeatedly picking the next item and inserting it into its correct position among the previously sorted items.
   * Time Complexity:
     + Best Case: O(n) (when the list is already sorted)
     + Average Case: O(n²)
     + Worst Case: O(n²)
   * **Space Complexity**: O(1) (in-place sort)
3. **Quick Sort**:
   * **Description**: Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into two sub-arrays, according to whether the elements are less than or greater than the pivot. The sub-arrays are then sorted recursively.
   * Time Complexity:
     + Best Case: O(n log n)
     + Average Case: O(n log n)
     + Worst Case: O(n²) (rare, occurs when the smallest or largest element is always chosen as the pivot)
   * **Space Complexity**: O(log n) (due to recursive stack space)
4. **Merge Sort**:
   * **Description**: Merge Sort is a divide-and-conquer algorithm that divides the array into two halves, sorts them recursively, and then merges the sorted halves.
   * **Time Complexity**:
     + Best Case: O(n log n)
     + Average Case: O(n log n)
     + Worst Case: O(n log n)
   * **Space Complexity**: O(n) (due to auxiliary space used for merging)

**Implementation: JAVA Code**

*public class* Main {  
 *public static void* main(String[] args) {  
 *// Create an array of orders* Order[] orders = {  
 *new* Order("O001", "Alice", 300.50),  
 *new* Order("O002", "Bob", 150.75),  
 *new* Order("O003", "Charlie", 500.00),  
 *new* Order("O004", "David", 200.00),  
 *new* Order("O005", "Eve", 450.25)  
 };  
  
 *// Bubble Sort example* SortAlgorithms.*bubbleSort*(orders);  
 System.***out***.println("Orders sorted by totalPrice using Bubble Sort:");  
 *for* (Order order : orders) {  
 System.***out***.println(order);  
 }  
  
 *// Reset orders array* orders = *new* Order[]{  
 *new* Order("O001", "Alice", 300.50),  
 *new* Order("O002", "Bob", 150.75),  
 *new* Order("O003", "Charlie", 500.00),  
 *new* Order("O004", "David", 200.00),  
 *new* Order("O005", "Eve", 450.25)  
 };  
  
 *// Quick Sort example* SortAlgorithms.*quickSort*(orders, 0, orders.length - 1);  
 System.***out***.println("\nOrders sorted by totalPrice using Quick Sort:");  
 *for* (Order order : orders) {  
 System.***out***.println(order);  
 }  
 }  
}  
  
*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;  
 }  
  
 *public* String getOrderId() {  
 *return* orderId;  
 }  
  
 *public* String getCustomerName() {  
 *return* customerName;  
 }  
  
 *public double* getTotalPrice() {  
 *return* totalPrice;  
 }  
  
 @Override  
 *public* String toString() {  
 *return* "Order ID: " + orderId + ", Customer Name: " + customerName + ", Total Price: $" + totalPrice;  
 }  
}  
  
*class* SortAlgorithms {  
 *public static void* bubbleSort(Order[] orders) {  
 *int* n = orders.length;  
 *boolean* swapped;  
 *for* (*int* i = 0; i < n - 1; i++) {  
 swapped = *false*;  
 *for* (*int* j = 0; j < n - i - 1; j++) {  
 *if* (orders[j].getTotalPrice() > orders[j + 1].getTotalPrice()) {  
 *// Swap orders[j] and orders[j + 1]* Order temp = orders[j];  
 orders[j] = orders[j + 1];  
 orders[j + 1] = temp;  
 swapped = *true*;  
 }  
 }  
 *if* (!swapped) {  
 *break*;  
 }  
 }  
 }  
  
 *public static void* quickSort(Order[] orders, *int* low, *int* high) {  
 *if* (low < high) {  
 *int* pi = *partition*(orders, low, high);  
 *quickSort*(orders, low, pi - 1);  
 *quickSort*(orders, pi + 1, high);  
 }  
 }  
  
 *private static int* partition(Order[] orders, *int* low, *int* high) {  
 *double* pivot = orders[high].getTotalPrice();  
 *int* i = (low - 1);  
 *for* (*int* j = low; j <= high - 1; j++) {  
 *if* (orders[j].getTotalPrice() <= pivot) {  
 i++;  
 Order temp = orders[i];  
 orders[i] = orders[j];  
 orders[j] = temp;  
 }  
 }  
 Order temp = orders[i + 1];  
 orders[i + 1] = orders[high];  
 orders[high] = temp;  
 *return* (i + 1);  
 }  
}

Q2. Compare the performance (time complexity) of Bubble Sort and Quick Sort?

Ans.

* **Bubble Sort**:
  + **Time Complexity**: O(n²) in the average and worst-case scenarios. It is inefficient for large datasets due to the quadratic time complexity.
  + **Space Complexity**: O(1) (in-place sort).
* **Quick Sort**:
  + Time Complexity:
    - Best Case: O(n log n)
    - Average Case: O(n log n)
    - Worst Case: O(n²) (rare, occurs when the smallest or largest element is always chosen as the pivot).
  + **Space Complexity**: O(log n) (due to recursive stack space).
  + **Explanation**: Quick Sort is generally faster than Bubble Sort because it reduces the problem size more rapidly, leading to fewer comparisons and swaps overall.

Q3. Discuss why Quick Sort is generally preferred over Bubble Sort?

Ans.

* **Efficiency**: Quick Sort has a significantly better average-case time complexity (O(n log n)) compared to Bubble Sort (O(n²)).
* **Scalability**: Quick Sort is more scalable for larger datasets due to its divide-and-conquer approach, which reduces the overall number of comparisons and swaps needed to sort the array.
* **Practical Performance**: Even though the worst-case time complexity of Quick Sort is O(n²), this can be mitigated by using techniques such as randomizing the pivot or using the median-of-three method to choose the pivot, making it very efficient in practice.

**Exercise 4: Employee Management System**

Q1. Explain how arrays are represented in memory and their advantages?

Ans.

* **Array Representation in Memory**:
  + **Memory Layout**: Arrays are stored in contiguous memory locations. Each element of the array is located next to the previous one.
  + **Indexing**: Arrays provide constant-time (O(1)) access to elements using their index because the index is directly mapped to a memory address.
* **Advantages**:
  + **Fast Access**: Constant-time access to elements using their index.
  + **Memory Efficiency**: No additional memory overhead for storing pointers or links as in linked lists.

**Implementation: JAVA Code**

**Employee.java**

*public class* Employee {  
 *private* String employeeId;  
 *private* String name;  
 *private* String position;  
 *private double* salary;  
  
 *public* Employee(String employeeId, String name, String position, *double* salary) {  
 *this*.employeeId = employeeId;  
 *this*.name = name;  
 *this*.position = position;  
 *this*.salary = salary;  
 }  
  
 *public* String getEmployeeId() {  
 *return* employeeId;  
 }  
  
 *public* String getName() {  
 *return* name;  
 }  
  
 *public* String getPosition() {  
 *return* position;  
 }  
  
 *public double* getSalary() {  
 *return* salary;  
 }  
  
 @Override  
 *public* String toString() {  
 *return* "Employee ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: $" + salary;  
 }  
}

**EmployeeManagementSystem.java**

*import* java.util.Arrays;  
  
*public class* EmployeeManagementSystem {  
 *private* Employee[] employees;  
 *private int* size;  
  
 *public* EmployeeManagementSystem(*int* capacity) {  
 employees = *new* Employee[capacity];  
 size = 0;  
 }  
  
 *// Add an employee  
 public void* addEmployee(Employee employee) {  
 *if* (size == employees.length) {  
 employees = Arrays.*copyOf*(employees, employees.length \* 2);  
 }  
 employees[size++] = employee;  
 }  
  
 *// Search for an employee by ID  
 public* Employee searchEmployee(String employeeId) {  
 *for* (*int* i = 0; i < size; i++) {  
 *if* (employees[i].getEmployeeId().equals(employeeId)) {  
 *return* employees[i];  
 }  
 }  
 *return null*;  
 }  
  
 *// Traverse and display all employees  
 public void* traverseEmployees() {  
 *for* (*int* i = 0; i < size; i++) {  
 System.***out***.println(employees[i]);  
 }  
 }  
  
 *// Delete an employee by ID  
 public boolean* deleteEmployee(String employeeId) {  
 *for* (*int* i = 0; i < size; i++) {  
 *if* (employees[i].getEmployeeId().equals(employeeId)) {  
 *// Shift remaining elements to the left  
 for* (*int* j = i; j < size - 1; j++) {  
 employees[j] = employees[j + 1];  
 }  
 employees[--size] = *null*; *// Nullify the last element  
 return true*;  
 }  
 }  
 *return false*;  
 }  
}

**Main.java**

*public class* Main {  
 *public static void* main(String[] args) {  
 EmployeeManagementSystem ems = *new* EmployeeManagementSystem(5);  
  
 *// Adding employees* ems.addEmployee(*new* Employee("E001", "Alice", "Manager", 80000));  
 ems.addEmployee(*new* Employee("E002", "Bob", "Developer", 60000));  
 ems.addEmployee(*new* Employee("E003", "Charlie", "Designer", 55000));  
 ems.addEmployee(*new* Employee("E004", "David", "Developer", 60000));  
 ems.addEmployee(*new* Employee("E005", "Eve", "HR", 50000));  
  
 *// Traversing employees* System.***out***.println("All Employees:");  
 ems.traverseEmployees();  
  
 *// Searching for an employee* System.***out***.println("\nSearch for Employee ID E003:");  
 Employee foundEmployee = ems.searchEmployee("E003");  
 System.***out***.println(foundEmployee != *null* ? foundEmployee : "Employee not found");  
  
 *// Deleting an employee* System.***out***.println("\nDeleting Employee ID E002:");  
 *boolean* isDeleted = ems.deleteEmployee("E002");  
 System.***out***.println(isDeleted ? "Employee deleted" : "Employee not found");  
  
 *// Traversing employees after deletion* System.***out***.println("\nAll Employees after deletion:");  
 ems.traverseEmployees();  
 }  
}

Q2. Analyze the time complexity of each operation (add, search, traverse, delete)?

Ans.

* **Time Complexity**:
  + **Add**:
    - Best Case: O(1) (if there is space available in the array).
    - Worst Case: O(n) (if the array needs to be resized).
  + **Search**: O(n) (linear search through the array).
  + **Traverse**: O(n) (iterate through all elements).
  + **Delete**: O(n) (find the element and shift remaining elements).

Q3. Discuss the limitations of arrays and when to use them?

Ans.

* **Limitations of Arrays**:
  + **Fixed Size**: Initial size must be defined, and resizing can be costly.
  + **Inefficient Deletion and Insertion**: Requires shifting elements, leading to O(n) time complexity.
  + **Static Memory Allocation**: Memory is allocated at compile time, which might be inefficient if the number of elements varies significantly.
* **When to Use Arrays**:
  + When you need constant-time access to elements using their index.
  + When the number of elements is known in advance and is relatively fixed.
  + When memory overhead needs to be minimized.

**Exercise 5: Task Management System**

Q1. Explain the different types of linked lists (Singly Linked List, Doubly Linked List)?

Ans.

1. **Singly Linked List**:
   * **Description**: Each node contains data and a reference (or link) to the next node in the sequence.
   * **Structure**: Node -> Node -> Node -> ... -> null
   * **Operations**: Easy to traverse forward, adding or removing nodes can be done in constant time if the position is known.
2. **Doubly Linked List**:
   * **Description**: Each node contains data, a reference to the next node, and a reference to the previous node.
   * **Structure**: null <- Node <-> Node <-> Node -> null
   * **Operations**: Can be traversed both forward and backward. Easier to delete nodes without having a reference to the previous node.

Q2. Analyze the time complexity of each operation?

Ans.

* **Time Complexity**:
  + **Add**: O(n)
  + **Search**: O(n)
  + **Traverse**: O(n)
  + **Delete**: O(n)

Q3. Discuss the advantages of linked lists over arrays for dynamic data?

Ans.

* **Advantages of Linked Lists over Arrays for Dynamic Data**:
  + **Dynamic Size**: Linked lists can grow and shrink dynamically, unlike arrays which have a fixed size.
  + **Efficient Insertions/Deletions**: Insertions and deletions can be done in constant time (O(1)) if the position is known, without needing to shift elements.
  + **Memory Utilization**: Linked lists allocate memory as needed, which can be more efficient if the number of elements changes frequently.

**Exercise 6: Library Management System**

Q1. Explain linear search and binary search algorithms?

Ans.

* **Linear Search**:
  + **Description**: A straightforward algorithm that checks each element in the list sequentially until the desired element is found or the list ends.
  + **Time Complexity**: O(n) for best, average, and worst cases, where n is the number of elements in the list.
* **Binary Search**:
  + **Description**: An efficient algorithm that repeatedly divides a sorted list in half to locate the desired element.
  + **Time Complexity**: O(log n) for best, average, and worst cases, where n is the number of elements in the list.
  + **Prerequisite**: The list must be sorted.

**Implementation: JAVA Code**

**Book.java**

*public class* Book {  
 *private* String bookId;  
 *private* String title;  
 *private* String author;  
  
 *public* Book(String bookId, String title, String author) {  
 *this*.bookId = bookId;  
 *this*.title = title;  
 *this*.author = author;  
 }  
  
 *public* String getBookId() {  
 *return* bookId;  
 }  
  
 *public* String getTitle() {  
 *return* title;  
 }  
  
 *public* String getAuthor() {  
 *return* author;  
 }  
  
 @Override  
 *public* String toString() {  
 *return* "Book ID: " + bookId + ", Title: " + title + ", Author: " + author;  
 }  
}

**LibraryManagementSystem.java**

*import* java.util.Arrays;  
  
*public class* LibraryManagementSystem {  
 *private* Book[] books;  
 *private int* size;  
  
 *public* LibraryManagementSystem(*int* capacity) {  
 books = *new* Book[capacity];  
 size = 0;  
 }  
  
 *// Add a book  
 public void* addBook(Book book) {  
 *if* (size == books.length) {  
 books = Arrays.*copyOf*(books, books.length \* 2);  
 }  
 books[size++] = book;  
 }  
  
 *// Linear search to find a book by title  
 public* Book linearSearchByTitle(String title) {  
 *for* (*int* i = 0; i < size; i++) {  
 *if* (books[i].getTitle().equalsIgnoreCase(title)) {  
 *return* books[i];  
 }  
 }  
 *return null*;  
 }  
  
 *// Binary search to find a book by title (assuming the list is sorted)  
 public* Book binarySearchByTitle(String title) {  
 *int* left = 0;  
 *int* right = size - 1;  
  
 *while* (left <= right) {  
 *int* mid = left + (right - left) / 2;  
 *int* comparison = books[mid].getTitle().compareToIgnoreCase(title);  
 *if* (comparison == 0) {  
 *return* books[mid];  
 } *else if* (comparison < 0) {  
 left = mid + 1;  
 } *else* {  
 right = mid - 1;  
 }  
 }  
 *return null*;  
 }  
  
 *// Sort books by title (for binary search)  
 public void* sortBooksByTitle() {  
 Arrays.*sort*(books, 0, size, (b1, b2) -> b1.getTitle().compareToIgnoreCase(b2.getTitle()));  
 }  
}

**Main.java**

*public class* Main {  
 *public static void* main(String[] args) {  
 LibraryManagementSystem lms = *new* LibraryManagementSystem(5);  
  
 *// Adding books* lms.addBook(*new* Book("B001", "The Catcher in the Rye", "J.D. Salinger"));  
 lms.addBook(*new* Book("B002", "To Kill a Mockingbird", "Harper Lee"));  
 lms.addBook(*new* Book("B003", "1984", "George Orwell"));  
 lms.addBook(*new* Book("B004", "Pride and Prejudice", "Jane Austen"));  
 lms.addBook(*new* Book("B005", "The Great Gatsby", "F. Scott Fitzgerald"));  
  
 *// Linear search for a book by title* System.***out***.println("Linear Search for '1984':");  
 Book foundBook = lms.linearSearchByTitle("1984");  
 System.***out***.println(foundBook != *null* ? foundBook : "Book not found");  
  
 *// Sort books by title for binary search* lms.sortBooksByTitle();  
  
 *// Binary search for a book by title* System.***out***.println("\nBinary Search for 'Pride and Prejudice':");  
 foundBook = lms.binarySearchByTitle("Pride and Prejudice");  
 System.***out***.println(foundBook != *null* ? foundBook : "Book not found");  
 }  
}

Q2. Compare the time complexity of linear and binary search?

Ans.

* **Linear Search**: O(n)
  + Best Case: O(1)
  + Average Case: O(n/2) -> O(n).
  + Worst Case: O(n) (if the target element is the last element or not present).
* **Binary Search**: O(log n)
  + Best Case: O(1) (if the target element is the middle element).
  + Average Case: O(log n).
  + Worst Case: O(log n) (if the target element is not present).

Q3. Discuss when to use each algorithm based on the data set size and order?

Ans.

* **Linear Search**:
  + Use when the dataset is small or unsorted.
  + Simple and requires no preprocessing.
* **Binary Search**:
  + Use when the dataset is large and sorted.
  + More efficient than linear search due to logarithmic time complexity.
  + Requires sorting the dataset initially, which adds an O(n log n) overhead for the sorting step.

**Exercise 7: Financial Forecasting**

Q1. Explain the concept of recursion and how it can simplify certain problems?

Ans.

* **Concept of Recursion**:
  + **Definition**: Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem.
  + **Structure**: A recursive function calls itself with a smaller subset of the original problem until it reaches a base case, which is a simple instance of the problem that can be solved directly.
  + **Simplification**: Recursion can simplify complex problems by breaking them down into smaller, more manageable sub-problems.

**Implementation: JAVA Code**

**FinancialForecasting.java**

*public class* FinancialForecasting {  
  
 *// Method to calculate future value recursively  
 public static double* calculateFutureValue(*double* presentValue, *double* growthRate, *int* periods) {  
 *// Base case  
 if* (periods == 0) {  
 *return* presentValue;  
 }  
 *// Recursive case  
 return* (1 + growthRate) \* *calculateFutureValue*(presentValue, growthRate, periods - 1);  
 }  
  
 *public static void* main(String[] args) {  
 *double* presentValue = 1000.0; *// Present value  
 double* growthRate = 0.05; *// Growth rate (5%)  
 int* periods = 10; *// Number of periods  
  
 double* futureValue = *calculateFutureValue*(presentValue, growthRate, periods);  
 System.***out***.println("Future Value: $" + futureValue);  
 }  
}

Q2. Discuss the time complexity of your recursive algorithm?

Ans.

* The time complexity of the recursive algorithm is O(n), where n is the number of periods. This is because the algorithm makes a single recursive call for each period until it reaches the base case.

Q3. Explain how to optimize the recursive solution to avoid excessive computation?

Ans.

* **Memoization**: To avoid excessive computation, store the results of already calculated values and reuse them. This technique is known as memoization.
* **Iterative Approach**: Convert the recursive solution to an iterative one to avoid the overhead of recursive calls and reduce the risk of stack overflow for large input values.