Exercise 1: Inventory Management System

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

- **Efficiency**: Efficient data structures and algorithms ensure that operations like searching, adding, updating, and deleting products are performed quickly, which is crucial for large inventories.
- **Scalability**: Proper data structures help manage and scale the system as the number of products grows.
- **Memory Management**: Appropriate data structures ensure optimal use of memory, avoiding wastage and ensuring that the system can handle large volumes of data.

Types of Data Structures Suitable for This Problem:

- **ArrayList**: Good for dynamic arrays where quick access to elements is needed. However, it has a slower insert and delete time as elements need to be shifted.
- **HashMap**: Provides constant-time performance for basic operations (add, update, delete), making it suitable for scenarios where quick lookups are essential.

Product.java

```
public class Product {
    private int productId;
    private String productName;
    private int quantity;
    private double price;

    public Product(int productId, String productName, int quantity, double price) {
        this.productId = productId;
        this.productName = productName;
        this.quantity = quantity;
        this.price = price;
    }

    // Getters and Setters
    public int getProductId() {
        return productId;
    }

    public void setProductId(int 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;
    }

    @Override
    public String toString() {
        return "Product [productId=" + productId + ", productName=" + productName + ", quantity="
+ quantity + ", price=" + price + "]";
    }
}
```

Inventory.java

```
import java.util.HashMap;
import java.util.Map;

public class Inventory {
    private Map<Integer, Product> products;

    public Inventory() {
        products = new HashMap<>();
    }
```

```
    // Add a new product
    public void addProduct(Product product) {
        products.put(product.getProductId(), product);
    }

    // Update an existing product
    public void updateProduct(Product product) {
        if (products.containsKey(product.getProductId())) {
            products.put(product.getProductId(), product);
        } else {
            System.out.println("Product not found.");
        }
    }

    // Delete a product
    public void deleteProduct(int productId) {
        if (products.containsKey(productId)) {
            products.remove(productId);
        } else {
            System.out.println("Product not found.");
        }
    }

    // Display all products
    public void displayProducts() {
        for (Product product : products.values()) {
            System.out.println(product);
        }
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        Inventory inventory = new Inventory();

        // Adding products
        Product product1 = new Product(1, "Product1", 10, 100.0);
        Product product2 = new Product(2, "Product2", 20, 200.0);
        inventory.addProduct(product1);
        inventory.addProduct(product2);

        // Display products
```

```
        System.out.println("Products after adding:");
        inventory.displayProducts();

        // Update a product
        Product updatedProduct1 = new Product(1, "UpdatedProduct1", 15, 150.0);
        inventory.updateProduct(updatedProduct1);

        // Display products after update
        System.out.println("Products after updating:");
        inventory.displayProducts();

        // Delete a product
        inventory.deleteProduct(2);

        // Display products after deletion
        System.out.println("Products after deleting:");
        inventory.displayProducts();
    }
}
```

Analysis

Time Complexity:

- Add Operation: O(1) - HashMap provides constant-time performance for inserting a new element.
- Update Operation: O(1) - HashMap allows constant-time performance for updating an existing element.
- Delete Operation: O(1) - HashMap provides constant-time performance for removing an element.

Optimization:

- Indexing: Ensure that productId is a unique identifier and used as a key in the HashMap for O(1) access.
- Concurrency Handling: For a multi-threaded environment, consider using ConcurrentHashMap to handle concurrent modifications.
- Memory Management: Periodically check and remove unused or obsolete products to free up memory.

Exercise 2: E-commerce Platform Search Function

Big O Notation:

- **Big O Notation** is a mathematical representation used to describe the upper bound of an algorithm's running time. It characterizes functions according to their growth rates and helps in comparing the efficiency of different algorithms.

- **Best-case, Average-case, and Worst-case Scenarios**:
    - **Best-case**: The minimum time an algorithm takes to complete. For search operations, this happens when the element is found in the first position.
    - **Average-case**: The expected time an algorithm takes to complete over a set of possible inputs. For search operations, this is typically halfway through the data set.
    - **Worst-case**: The maximum time an algorithm takes to complete. For search operations, this happens when the element is at the last position or not present at all.

Product.java

```
public class Product {
    private int productId;
    private String productName;
    private String category;

    public Product(int productId, String productName, String category) {
        this.productId = productId;
        this.productName = productName;
        this.category = category;
    }

    // Getters
    public int getProductId() {
        return productId;
    }

    public String getProductName() {
        return productName;
    }

    public String getCategory() {
        return category;
    }

    @Override
    public String toString() {
            return "Product [productId=" + productId + ", productName=" + productName + ", category=" + category + "]";
    }
}
```

Linear Search Algorithm: SearchAlgorithms.java

```
public class SearchAlgorithms {

    public static Product linearSearch(Product[] products, int productId) {
        for (Product product : products) {
            if (product.getProductId() == productId) {
                return product;
            }
        }
        return null;
    }
}
```

Binary Search Algorithm: SearchAlgorithms.java

```
import java.util.Arrays;
import java.util.Comparator;

public class SearchAlgorithms {

    public static Product binarySearch(Product[] products, int productId) {
        Arrays.sort(products, Comparator.comparingInt(Product::getProductId)); // Ensure the array
is sorted

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

        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (products[mid].getProductId() == productId) {
                return products[mid];
            }
            if (products[mid].getProductId() < productId) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        Product[] products = {
            new Product(1, "Laptop", "Electronics"),
            new Product(2, "Phone", "Electronics"),
            new Product(3, "Shoes", "Apparel"),
            new Product(4, "Watch", "Accessories"),
            new Product(5, "Bag", "Accessories")
        };

        // Linear search
        Product foundProduct = SearchAlgorithms.linearSearch(products, 3);
          System.out.println("Linear Search: " + (foundProduct != null ? foundProduct : "Product not found"));

        // Binary search
        foundProduct = SearchAlgorithms.binarySearch(products, 3);
          System.out.println("Binary Search: " + (foundProduct != null ? foundProduct : "Product not found"));
    }
}
```

Analysis

Time Complexity:

- Linear Search:
    - Best-case: O(1) - The product is found at the first position.
    - Average-case: O(n/2) - The product is found halfway through the array.
    - Worst-case: O(n) - The product is found at the last position or not present.
- Binary Search:
    - Best-case: O(1) - The product is found at the middle position.
    - Average-case: O(log n) - The search space is halved with each iteration.
    - Worst-case: O(log n) - The product is not present in the array.

Which Algorithm is More Suitable and Why:

- Binary Search is more suitable for the platform if the products are frequently searched and the list can be kept sorted. It offers much better performance (O(log n)) compared to linear search (O(n)), especially for large datasets.
- Linear Search might be used if the dataset is small or if maintaining a sorted array is not feasible. However, for an e-commerce platform with potentially large inventories, binary search would typically be the preferred method due to its efficiency.

Exercise 3: Sorting Customer Orders

Sorting Algorithms:

- **Bubble Sort**: A simple comparison-based sorting algorithm where each pair of adjacent elements is compared, and the elements are swapped if they are in the wrong order. This process is repeated until the list is sorted.
    - **Time Complexity**: O(n^2) for best, average, and worst cases.
- **Insertion Sort**: Builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.
    - **Time Complexity**: O(n) for best case (nearly sorted array), O(n^2) for average and worst cases.
- **Quick Sort**: A highly efficient sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
    - **Time Complexity**: O(n log n) on average, O(n^2) in the worst case (when the smallest or largest element is always chosen as the pivot).
- **Merge Sort**: A divide-and-conquer algorithm that divides the unsorted list into n sublists, each containing one element, and then repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining.
    - **Time Complexity**: O(n log n) for best, average, and worst cases.

Order.java

```
public class Order {
    private int orderId;
    private String customerName;
    private double totalPrice;

    public Order(int orderId, String customerName, double totalPrice) {
        this.orderId = orderId;
        this.customerName = customerName;
        this.totalPrice = totalPrice;
    }

    // Getters
    public int getOrderId() {
        return orderId;
    }

    public String getCustomerName() {
        return customerName;
```

```
    }

    public double getTotalPrice() {
        return totalPrice;
    }

    @Override
    public String toString() {
         return "Order [orderId=" + orderId + ", customerName=" + customerName + ", totalPrice="
+ totalPrice + "]";
    }
}
```

Bubble Sort Algorithm: SortingAlgorithms.java

```
public class SortingAlgorithms {

    public static void bubbleSort(Order[] orders) {
        int n = orders.length;
        for (int i = 0; i < n - 1; i++) {
            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;
                }
            }
        }
    }
}
```

Quick Sort Algorithm: SortingAlgorithms.java

```
public class SortingAlgorithms {

    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); // Index of smaller element
        for (int j = low; j < high; j++) {
            if (orders[j].getTotalPrice() < pivot) {
                i++;
                // Swap orders[i] and orders[j]
                Order temp = orders[i];
                orders[i] = orders[j];
                orders[j] = temp;
            }
        }
        // Swap orders[i + 1] and orders[high] (or pivot)
        Order temp = orders[i + 1];
        orders[i + 1] = orders[high];
        orders[high] = temp;
        return i + 1;
    }
}
```

Main.java

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Order[] orders = {
            new Order(1, "Alice", 250.0),
            new Order(2, "Bob", 100.0),
            new Order(3, "Charlie", 150.0),
            new Order(4, "David", 200.0),
            new Order(5, "Eve", 300.0)
        };

        // Bubble Sort
        Order[] bubbleSortedOrders = Arrays.copyOf(orders, orders.length);
        SortingAlgorithms.bubbleSort(bubbleSortedOrders);
        System.out.println("Bubble Sorted Orders:");
        for (Order order : bubbleSortedOrders) {
            System.out.println(order);
        }

        // Quick Sort
        Order[] quickSortedOrders = Arrays.copyOf(orders, orders.length);
        SortingAlgorithms.quickSort(quickSortedOrders, 0, quickSortedOrders.length - 1);
        System.out.println("Quick Sorted Orders:");
```

```
        for (Order order : quickSortedOrders) {
            System.out.println(order);
        }
    }
}
```

Analysis

Time Complexity:

- **Bubble Sort**:
    - Best-case: O(n) - When the array is already sorted.
    - Average-case: O(n^2).
    - Worst-case: O(n^2).
- **Quick Sort**:
    - Best-case: O(n log n).
    - Average-case: O(n log n).
    - Worst-case: O(n^2) - When the smallest or largest element is always chosen as the pivot.

Why Quick Sort is Generally Preferred Over Bubble Sort:

- **Efficiency**: Quick Sort is generally faster than Bubble Sort, especially for large datasets, due to its average-case time complexity of O(n log n) compared to Bubble Sort's O(n^2).
- **Performance**: Quick Sort efficiently divides the array into smaller sub-arrays and sorts them independently, making it suitable for large datasets. Bubble Sort, on the other hand, performs poorly on large datasets due to its repetitive comparisons and swaps.
- **Optimization**: While Quick Sort's worst-case scenario is O(n^2), this can be mitigated by using randomized pivot selection or by choosing the median as the pivot. Bubble Sort lacks such optimizations and remains inefficient for large datasets.

Exercise 4: Employee Management System

How Arrays are Represented in Memory:

- **Contiguous Memory Allocation**: Arrays are stored in contiguous blocks of memory. This means that elements are stored one after another in a single block of memory.
- **Indexing**: Each element in the array can be accessed using its index, which provides constant-time access (O(1)) to any element.
- **Fixed Size**: The size of an array is fixed at the time of creation. This means the array cannot grow or shrink dynamically, which can be a limitation if the number of elements is not known in advance.

Advantages of Arrays:

- **Fast Access**: Arrays provide constant-time access to any element using its index.
- **Memory Efficiency**: Arrays use contiguous memory, which can be more efficient in terms of memory usage compared to other data structures that use pointers.
- **Cache Friendliness**: Due to contiguous memory allocation, arrays tend to be more cache-friendly, which can lead to better performance.

Employee.java

```
public class Employee {
    private int employeeId;
    private String name;
    private String position;
    private double salary;

    public Employee(int employeeId, String name, String position, double salary) {
        this.employeeId = employeeId;
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    // Getters and Setters
    public int getEmployeeId() {
        return employeeId;
    }

    public void setEmployeeId(int employeeId) {
        this.employeeId = employeeId;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }
```

```
    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public String toString() {
            return "Employee [employeeId=" + employeeId + ", name=" + name + ", position=" +
position + ", salary=" + salary + "]";
    }
}
```

EmployeeManagementSystem.java

```
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[size++] = employee;
        } else {
            System.out.println("Employee array is full. Cannot add more employees.");
        }
    }

    // Search for an employee by ID
    public Employee searchEmployee(int employeeId) {
        for (int i = 0; i < size; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                return employees[i];
            }
        }
        return null;
```

```
    }

    // Traverse all employees
    public void traverseEmployees() {
        for (int i = 0; i < size; i++) {
            System.out.println(employees[i]);
        }
    }

    // Delete an employee by ID
    public void deleteEmployee(int employeeId) {
        for (int i = 0; i < size; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                // Shift all subsequent employees left by one position
                for (int j = i; j < size - 1; j++) {
                    employees[j] = employees[j + 1];
                }
                employees[--size] = null; // Decrement size and set last element to null
                return;
            }
        }
        System.out.println("Employee not found.");
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        EmployeeManagementSystem ems = new EmployeeManagementSystem(10);

        // Add employees
        ems.addEmployee(new Employee(1, "Alice", "Manager", 60000));
        ems.addEmployee(new Employee(2, "Bob", "Developer", 50000));
        ems.addEmployee(new Employee(3, "Charlie", "Analyst", 45000));

        // Traverse employees
        System.out.println("All Employees:");
        ems.traverseEmployees();

        // Search for an employee
        System.out.println("Search for employee with ID 2:");
        Employee foundEmployee = ems.searchEmployee(2);
        System.out.println(foundEmployee != null ? foundEmployee : "Employee not found");
```

```
        // Delete an employee
        System.out.println("Deleting employee with ID 2:");
        ems.deleteEmployee(2);
        System.out.println("All Employees after deletion:");
        ems.traverseEmployees();
    }
}
```

Analysis

Time Complexity:

- **Add Operation**: O(1) - Adding an element to the array is a constant-time operation as long as there is space available.
- **Search Operation**: O(n) - In the worst case, the algorithm needs to search through all n elements.
- **Traverse Operation**: O(n) - Traversing all elements in the array requires visiting each element once.
- **Delete Operation**: O(n) - In the worst case, the algorithm needs to shift n-1 elements to fill the gap left by the deleted element.

Limitations of Arrays:

- **Fixed Size**: Arrays have a fixed size, which means you need to know the number of elements in advance. If the array is full, you cannot add more elements without creating a new larger array.
- **Inefficient Deletion and Insertion**: Deleting or inserting an element requires shifting elements, which can be inefficient for large arrays.
- **Memory Allocation**: Contiguous memory allocation can lead to fragmentation and inefficient use of memory if the array size is large.

When to Use Arrays:

- **Small, Fixed-size Collections**: Arrays are suitable for small collections where the size is known in advance and does not change.
- **Fast Access Requirements**: When constant-time access to elements is needed, arrays are a good choice.
- **Low Overhead**: Arrays have low overhead compared to other dynamic data structures like linked lists or dynamic arrays (ArrayList).

Exercise 5: Task Management System

Types of Linked Lists:

- **Singly Linked List**: A type of linked list where each node contains data and a reference (or link) to the next node in the sequence. It allows traversal in one direction only (forward).
    - **Structure**: `Node -> Node -> Node -> null`
- **Doubly Linked List**: A type of linked list where each node contains data, a reference to the next node, and a reference to the previous node. This allows traversal in both directions (forward and backward).
    - **Structure**: `null <- Node <-> Node <-> Node -> null`

Advantages of Linked Lists:

- **Dynamic Size**: Linked lists can grow or shrink in size dynamically, as opposed to arrays which have a fixed size.
- **Efficient Insertions/Deletions**: Inserting or deleting elements in a linked list is more efficient than in an array, especially when the elements are added or removed from the beginning or middle of the list.

Task.java

```
public class Task {
    private int taskId;
    private String taskName;
    private String status;

    public Task(int taskId, String taskName, String status) {
        this.taskId = taskId;
        this.taskName = taskName;
        this.status = status;
    }

    // Getters and Setters
    public int getTaskId() {
        return taskId;
    }

    public void setTaskId(int taskId) {
        this.taskId = taskId;
    }

    public String getTaskName() {
        return taskName;
    }

    public void setTaskName(String taskName) {
        this.taskName = taskName;
```

```
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    @Override
    public String toString() {
        return "Task [taskId=" + taskId + ", taskName=" + taskName + ", status=" + status + "]";
    }
}
```

Node.java

```
class Node {
    Task task;
    Node next;

    public Node(Task task) {
        this.task = task;
        this.next = null;
    }
}
```

TaskLinkedList.java

```
public class TaskLinkedList {
    private Node head;

    public TaskLinkedList() {
        this.head = null;
    }

    // Add a task to the list
    public void addTask(Task task) {
        Node newNode = new Node(task);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
```

```
            current = current.next;
        }
        current.next = newNode;
    }
}

// Search for a task by ID
public Task searchTask(int taskId) {
    Node current = head;
    while (current != null) {
        if (current.task.getTaskId() == taskId) {
            return current.task;
        }
        current = current.next;
    }
    return null;
}

// Traverse all tasks
public void traverseTasks() {
    Node current = head;
    while (current != null) {
        System.out.println(current.task);
        current = current.next;
    }
}

// Delete a task by ID
public void deleteTask(int taskId) {
    if (head == null) {
        System.out.println("Task list is empty.");
        return;
    }
    if (head.task.getTaskId() == taskId) {
        head = head.next;
        return;
    }
    Node current = head;
    while (current.next != null && current.next.task.getTaskId() != taskId) {
        current = current.next;
    }
    if (current.next == null) {
        System.out.println("Task not found.");
    } else {
```

```
            current.next = current.next.next;
        }
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        TaskLinkedList taskList = new TaskLinkedList();

        // Add tasks
        taskList.addTask(new Task(1, "Design system architecture", "Pending"));
        taskList.addTask(new Task(2, "Implement user authentication", "In Progress"));
        taskList.addTask(new Task(3, "Develop REST API", "Pending"));
        taskList.addTask(new Task(4, "Write unit tests", "Completed"));

        // Traverse tasks
        System.out.println("All Tasks:");
        taskList.traverseTasks();

        // Search for a task
        System.out.println("\nSearch for task with ID 2:");
        Task foundTask = taskList.searchTask(2);
        System.out.println(foundTask != null ? foundTask : "Task not found");

        // Delete a task
        System.out.println("\nDeleting task with ID 2:");
        taskList.deleteTask(2);
        System.out.println("All Tasks after deletion:");
        taskList.traverseTasks();
    }
}
```

Analysis

Time Complexity:

- **Add Operation**: O(n) - In the worst case, the algorithm needs to traverse the entire list to add a new task at the end.
- **Search Operation**: O(n) - In the worst case, the algorithm needs to traverse the entire list to find the task.
- **Traverse Operation**: O(n) - The algorithm needs to visit each node once to traverse the list.

- **Delete Operation**: O(n) - In the worst case, the algorithm needs to traverse the entire list to find and delete the task.

Advantages of Linked Lists Over Arrays for Dynamic Data:

- **Dynamic Size**: Linked lists can grow and shrink dynamically as tasks are added or removed, without the need for resizing or allocating a new array.
- **Efficient Insertions/Deletions**: Inserting or deleting elements, especially at the beginning or middle of the list, is more efficient in linked lists compared to arrays, as it only requires changing the references of a few nodes rather than shifting elements.
- **Memory Usage**: Linked lists can be more memory-efficient for dynamic data where the number of elements changes frequently. Arrays may waste memory if they are initialized with a large capacity but contain few elements.

Limitations of Linked Lists:

- **Access Time**: Linked lists have slower access times compared to arrays since elements must be accessed sequentially.
- **Memory Overhead**: Each node in a linked list requires additional memory for storing the reference to the next (and possibly previous) node, which can increase memory usage compared to arrays.

Exercise 6: Library Management System

Linear Search Algorithm:
Description: Linear search involves checking each element of the array or list sequentially until the desired element is found or the end of the list is reached.
Time Complexity: O(n) in the worst case, where n is the number of elements in the list. This is because in the worst-case scenario, the algorithm might need to check every element.

Binary Search Algorithm:
Description: Binary search works on sorted arrays or lists. It repeatedly divides the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. The process continues until the search key is found or the interval is empty.
Time Complexity: O(log n) in the worst case, where n is the number of elements in the list. This logarithmic time complexity is due to the halving of the search space with each step.

Book.java

```
public class Book {
    private int bookId;
    private String title;
    private String author;
```

```
    public Book(int bookId, String title, String author) {
        this.bookId = bookId;
        this.title = title;
        this.author = author;
    }

    // Getters and Setters
    public int getBookId() {
        return bookId;
    }

    public void setBookId(int bookId) {
        this.bookId = bookId;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    @Override
    public String toString() {
        return "Book [bookId=" + bookId + ", title=" + title + ", author=" + author + "]";
    }
}
```

Linear Search to Find Books by Title: LibraryManagementSystem.java

```
public class LibraryManagementSystem {
    private Book[] books;
    private int size;

    public LibraryManagementSystem(int capacity) {
        books = new Book[capacity];
```

```
        size = 0;
    }

    // Add a book to the library
    public void addBook(Book book) {
        if (size < books.length) {
            books[size++] = book;
        } else {
            System.out.println("Library is full. Cannot add more books.");
        }
    }

    // Linear search for 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 Books by Title (Assuming the List is Sorted): LibraryManagementSystem.java

```
import java.util.Arrays;
import java.util.Comparator;

public class LibraryManagementSystem {

    // Assuming other methods (addBook, etc.) are already implemented

    // Binary search for a book by title
    public Book binarySearchByTitle(String title) {
        Arrays.sort(books, 0, size, Comparator.comparing(Book::getTitle));
        int left = 0;
        int right = size - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int compareResult = books[mid].getTitle().compareToIgnoreCase(title);
            if (compareResult == 0) {
                return books[mid];
            } else if (compareResult < 0) {
```

```
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return null;
  }
}
```

Main.java

```
public class Main {
  public static void main(String[] args) {
    LibraryManagementSystem library = new LibraryManagementSystem(10);

    // Add books
    library.addBook(new Book(1, "Effective Java", "Joshua Bloch"));
    library.addBook(new Book(2, "Clean Code", "Robert C. Martin"));
    library.addBook(new Book(3, "Java Concurrency in Practice", "Brian Goetz"));

    // Linear search for a book
    System.out.println("Linear Search for 'Clean Code':");
    Book foundBookLinear = library.linearSearchByTitle("Clean Code");
    System.out.println(foundBookLinear != null ? foundBookLinear : "Book not found");

    // Binary search for a book (assuming the list is sorted)
    System.out.println("Binary Search for 'Effective Java':");
    Book foundBookBinary = library.binarySearchByTitle("Effective Java");
    System.out.println(foundBookBinary != null ? foundBookBinary : "Book not found");
  }
}
```

Analysis

Time Complexity:

- **Linear Search**:
  - **Best-case**: O(1) - The book is the first element in the list.
  - **Average-case**: O(n/2) - On average, the algorithm will check half of the elements.
  - **Worst-case**: O(n) - The book is the last element or not in the list at all.
- **Binary Search**:
  - **Best-case**: O(1) - The book is the middle element of the list.
  - **Average-case**: O(log n) - The algorithm repeatedly divides the list in half.

    - **Worst-case**: O(log n) - The book is the last element checked after multiple divisions.

When to Use Each Algorithm:

- **Linear Search**:
    - Use linear search when the list is small or unsorted.
    - It is simple and does not require the list to be sorted.
    - Ideal for scenarios where data is dynamically changing, and sorting the list frequently is not practical.
- **Binary Search**:
    - Use binary search when the list is large and sorted.
    - It is more efficient for larger datasets due to its logarithmic time complexity.
    - Ideal for scenarios where search operations are frequent, and the list can be maintained in a sorted order.

Exercise 7: Financial Forecasting

Concept of Recursion:

- **Definition**: Recursion is a programming technique where a function calls itself in order to solve a problem. The function typically has a base case that stops the recursion and a recursive case that divides the problem into smaller instances of the same problem.
- **Simplification**: Recursion can simplify problems by breaking them down into smaller, more manageable sub-problems. It is particularly useful for problems that have a natural hierarchical structure, such as tree traversals, combinatorial problems, and divide-and-conquer algorithms.

Example: Calculating the factorial of a number n can be naturally expressed using recursion:

factorial(n) = n * factorial(n-1)

With the base case:

factorial(0) = 1

Setup

1. **Define the Problem**:
    - Given past data points representing growth rates, predict future values.
    - For simplicity, assume a fixed annual growth rate.
2. **Formula for Future Value**:

    Future Value = Present Value * $(1+\text{growth rate})^{n}$

where n is the number of years into the future.

FinancialForecasting.java

```
public class FinancialForecasting {

    // Method to calculate future value using recursion
     public static double calculateFutureValue(double presentValue, double growthRate, int years)
{
        // Base case: no more years to forecast
        if (years == 0) {
            return presentValue;
        }
        // Recursive case: calculate the future value for the next year
        return calculateFutureValue(presentValue * (1 + growthRate), growthRate, years - 1);
    }

    public static void main(String[] args) {
        double presentValue = 1000.0; // Example present value
        double growthRate = 0.05; // Example annual growth rate (5%)
        int years = 10; // Number of years into the future

        double futureValue = calculateFutureValue(presentValue, growthRate, years);
        System.out.println("Future Value after " + years + " years: $" + futureValue);
    }
}
```

2. **Explanation of the Code**:
    - The `calculateFutureValue` method takes the current value, growth rate, and number of years as parameters.
    - The base case checks if the number of years is 0, in which case it returns the present value.
    - The recursive case multiplies the present value by (1 + growth rate) and decreases the number of years by 1, calling the method again with these new values.

Analysis

Time Complexity:

- The time complexity of this recursive algorithm is O(n), where n is the number of years. This is because each recursive call reduces the number of years by 1, leading to a linear number of calls.

Optimizing the Recursive Solution:

- **Memoization**: One way to optimize the recursive solution is to use memoization, where we store the results of already computed values in a map or array to avoid redundant calculations.

Example with Memoization:

```
import java.util.HashMap;
import java.util.Map;

public class FinancialForecasting {

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

    // Method to calculate future value using recursion with memoization
     public static double calculateFutureValue(double presentValue, double growthRate, int years)
{
        // Base case: no more years to forecast
        if (years == 0) {
            return presentValue;
        }
        // Check if the result is already computed
        if (memo.containsKey(years)) {
            return memo.get(years);
        }
        // Recursive case: calculate the future value for the next year
          double futureValue = calculateFutureValue(presentValue * (1 + growthRate), growthRate,
years - 1);
        memo.put(years, futureValue);
        return futureValue;
    }

    public static void main(String[] args) {
        double presentValue = 1000.0; // Example present value
        double growthRate = 0.05; // Example annual growth rate (5%)
        int years = 10; // Number of years into the future

        double futureValue = calculateFutureValue(presentValue, growthRate, years);
        System.out.println("Future Value after " + years + " years: $" + futureValue);
    }
```

```
}
```

Explanation:

- **Memoization**: A `Map` is used to store the results of already computed future values. Before making a recursive call, the algorithm checks if the value for the given number of years is already computed and stored in the map. If so, it returns the stored value; otherwise, it computes the value, stores it in the map, and then returns it.
- **Optimization**: This reduces the number of redundant calculations, especially for large numbers of years, and can significantly improve performance by avoiding repeated work.