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.

      The reason why data structures and algorithms are important to manage big inventories efficiently, because:

      1. Fast storage and retrieval: The inventory data can be stored effectively with less storage and information will be easy to search if the right datatype is used. For example, hash tables can perform lookups of individual items in near-constant time.

      2. Fast searching and sorting: When working with large datasets, searching and sorting the dataset are crucial to have optimized algorithms. There are faster ways to search, such as binary search trees or balanced-search trees like the red black tree.

      3. Scalability: The main aim of the optimizations is to ensure that with increasing sizes on your inventory, a well-designed data structures and algorithms must be able handle them without needing too much time or resource.

      4. Memory management: Efficient memory allocation de-allocation: System with large amount of data require proper management on how the data is being stored in the system and for this type of scenario we should use most optimal Data structures.

      5. Advanced queries and analytics: By having these advanced data structures and algorithm), latency of executing complex queries, analytics on inventory is very low.

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

      ArrayList: Useful in those areas, where you have elements to be accessed randomly and also order of element is significant. Insertion and deletion are expensive because of moving elements.

      HashMap: Fast access, insertion and deletion. For mapping product IDs to details, it uses key-value pairs.

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**.

```
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;
    }
    public int getProductId() {
        return productId;
    }
    public String getProductName() {
        return productName;
    }
    public int getQuantity() {
        return quantity;
    }
    public double getPrice() {
        return price;
    }
    public void setProductId(int productId) {
        this.productId = productId;
    }
    public void setProductName(String productName) {
        this.productName = productName;
```

```
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public String toString() {
        return "Product{" +
                "productId=" + productId +
                ", productName='" + productName + '\'' +
                ", quantity=" + quantity +
                ", price=" + 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.

```
import java.util.HashMap;
import java.util.Map;
public class InventoryManagementSystem {
    private Map<Integer, Product> inventory;
    public InventoryManagementSystem() {
        inventory = new HashMap<>();
    }
    public void addProduct(Product product) {
        if (inventory.containsKey(product.getProductId())) {
```

```
            System.out.println("Product with ID " + product.getProductId() + " already
exists.");
        } else {
            inventory.put(product.getProductId(), product);
            System.out.println("Product added: " + product);
        }
    }
    public void updateProduct(Product product) {
        if (inventory.containsKey(product.getProductId())) {
            inventory.put(product.getProductId(), product);
            System.out.println("Product updated: " + product);
        } else {
            System.out.println("Product with ID " + product.getProductId() + " not found.");
        }
    }

    public void deleteProduct(int productId) {
        Product removedProduct = inventory.remove(productId);
        if (removedProduct != null) {
            System.out.println("Product removed: " + removedProduct);
        } else {
            System.out.println("Product with ID " + productId + " not found.");
        }
    }
    public Product getProduct(int productId) {
        return inventory.get(productId);
    }
    public static void main(String[] args) {
        InventoryManagementSystem ims = new InventoryManagementSystem();
```

```
        ims.addProduct(new Product(1, "Laptop", 10, 1000.00));
        ims.addProduct(new Product(2, "Smartphone", 20, 500.00));
        ims.updateProduct(new Product(1, "Gaming Laptop", 5, 1500.00));
        Product product = ims.getProduct(1);
        System.out.println("Retrieved Product: " + product);
        ims.deleteProduct(2);
    }
}
```

4. **Analysis:**
    - Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.

      Add Operation: The time complexity is O(1) because HashMap allows constant-time performance for insertion.

      Update Operation: The time complexity is O(1) because updating a value in HashMap is done in constant time.

      Delete Operation: The time complexity is O(1) because removal from HashMap is also constant-time.

    - Discuss how you can optimize these operations.

      Use of HashMap: Using a HashMap is optimal for scenarios where fast lookup, insertion, and deletion are necessary.

      Load Factor and Capacity: Adjust the initial capacity and load factor of the HashMap to reduce the frequency of resizing operations, which can improve performance for very large inventories.

      Indexing: Consider implementing indexing strategies if the search operations become more complex or if additional attributes need to be searched frequently.

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.
      Big O Notation is a mathematical notation used to describe the upper bound of an algorithm's running time. It gives an estimate of the worst-case scenario in terms of input size. It helps in comparing the efficiency of different algorithms by providing a high-level understanding of their performance as the input size grows.
    - Describe the best, average, and worst-case scenarios for search operations.
      1. **Linear Search:**
         - **Best Case:** O(1)
         - **Average Case:** O(n)
         - **Worst Case:** O(n).
      2. **Binary Search:**
         - **Best Case:** O(1)
         - **Average Case:** O(log n)
         - **Worst Case:** O(log n)
2. **Setup:**
    - Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

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

```
    return productId;    }
  public String getProductName() {
    return productName;
  }
  public String getCategory() {
    return category;
  }
  public void setProductId(int productId) {
    this.productId = productId;
  }
  public void setProductName(String productName) {
    this.productName = productName;
  }
  public void setCategory(String category) {
    this.category = category;
  }
  @Override
  public String toString() {
    return "Product{" +
        "productId=" + productId +
        ", productName='" + productName + '\'' +
        ", category='" + 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.

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

    public static void main(String[] args) {
        Product[] products = {
            new Product(1, "Laptop", "Electronics"),
            new Product(2, "Smartphone", "Electronics"),
            new Product(3, "Tablet", "Electronics"),
            new Product(4, "Monitor", "Electronics")
        };
        System.out.println("Linear Search:");
        Product foundProduct = linearSearch(products, 3);
        if (foundProduct != null) {
            System.out.println("Product found: " + foundProduct);
        } else {
            System.out.println("Product not found.");
        }
```

```
        Arrays.sort(products, (p1, p2) -> Integer.compare(p1.getProductId(),
    p2.getProductId()));
        System.out.println("Binary Search:");
        foundProduct = binarySearch(products, 3);
        if (foundProduct != null) {
            System.out.println("Product found: " + foundProduct);
        } else {
            System.out.println("Product not found.");
        }
    }
}
```

4. **Analysis:**

    - Compare the time complexity of linear and binary search algorithms.
      Linear Search:
      - **Best Case:** O(1)
      - **Average Case:** O(n)
      - **Worst Case:** O(n)

      Binary Search:
      - **Best Case:** O(1)
      - **Average Case:** O(log n)
      - **Worst Case:** O(log n)

    - Discuss which algorithm is more suitable for your platform and why.
      Linear Search: Suitable for small datasets or unsorted data where sorting is not feasible.
      Binary Search: More suitable for large, sorted datasets due to its logarithmic time complexity, which significantly reduces the number of comparisons needed to find an element.

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).

      Bubble Sort: A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
      Insertion Sort: Builds the final sorted array one item at a time, picking the next item and placing it in the correct position within the sorted part of the array.
      Quick Sort: An efficient sorting algorithm that uses a divide-and-conquer strategy to partition the array into sub-arrays and then recursively sort them.
      Merge Sort: A stable, comparison-based sorting algorithm that divides the array into halves, sorts them, and then merges them back together.
2. **Setup:**
    - Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.

      ```
      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;
          }
          public int getOrderId() {
              return orderId;
          }

          public String getCustomerName() {
      ```

```
    return customerName;
  }
  public double getTotalPrice() {
    return totalPrice;
  }
  public void setOrderId(int 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 "Order{" +
        "orderId=" + orderId +
        ", customerName='" + customerName + '\'' +
        ", totalPrice=" + totalPrice +
        '}';
  }
}
```

3. **Implementation:**
   - Implement **Bubble Sort** to sort orders by **totalPrice**.
   - Implement **Quick Sort** to sort orders by **totalPrice**.

```
public class SortingCustomerOrders {
  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()) {
                Order temp = orders[j];
                orders[j] = orders[j + 1];
                orders[j + 1] = temp;
            }
        }
    }
}

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; 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;
}
public static void main(String[] args) {
    Order[] orders = {
        new Order(1, "John Doe", 250.0),
        new Order(2, "Jane Smith", 150.0),
        new Order(3, "Mike Brown", 300.0),
        new Order(4, "Emily White", 200.0)
    };
    System.out.println("Orders before Bubble Sort:");
    for (Order order : orders) {
        System.out.println(order);
    }
    bubbleSort(orders);
    System.out.println("\nOrders after Bubble Sort:");
    for (Order order : orders) {
        System.out.println(order);
    }
    orders = new Order[]{
        new Order(1, "John Doe", 250.0),
        new Order(2, "Jane Smith", 150.0),
        new Order(3, "Mike Brown", 300.0),
        new Order(4, "Emily White", 200.0)
    };
    System.out.println("\nOrders before Quick Sort:");
    for (Order order : orders) {
```

```
            System.out.println(order);
        }
        quickSort(orders, 0, orders.length - 1);
        System.out.println("\nOrders after Quick Sort:");
        for (Order order : orders) {
            System.out.println(order);
        }
    }
}
```

4. **Analysis:**
    - Compare the performance (time complexity) of Bubble Sort and Quick Sort.
      Bubble Sort: Time complexity is O(n^2).
      Quick Sort: Time complexity is O(n log n).
    - Discuss why Quick Sort is generally preferred over Bubble Sort.
      Quick Sort is generally preferred over Bubble Sort because it is much more efficient for large datasets. It reduces the number of comparisons and swaps needed to sort the array, making it faster and more suitable for practical applications like sorting customer orders on an e-commerce platform.

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.

      Arrays are stored in contiguous memory locations. This means that each element in the array is located next to its neighbour in memory.
      Advantages:
      - **Fast Access:** Elements can be accessed in constant time, O(1), using their index.
      - **Memory Efficiency:** Arrays use memory efficiently as there is no overhead for storing pointers or additional metadata.
      - **Predictable Iteration:** Iterating through an array is straightforward and predictable due to contiguous memory.
2. **Setup:**
    - Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.

```
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;
    }
    public int getEmployeeId() {
        return employeeId;
```

```
}
public String getName() {
   return name;
}
public String getPosition() {
   return position;
}
public double getSalary() {
   return salary;
}
public void setEmployeeId(int employeeId) {
   this.employeeId = employeeId;
}
public void setName(String name) {
   this.name = name;
}
public void setPosition(String position) {
   this.position = position;
}
public void setSalary(double salary) {
   this.salary = salary;
}
@Override
public String toString() {
   return "Employee{" +
         "employeeId=" + employeeId +
         ", name='" + name + '\'' +
         ", position='" + position + '\'' +
         ", salary=" + salary +
```

```
            '}';
    }
}
```

3. **Implementation:**
    - Use an array to store employee records.
    - Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.

```
import java.util.Arrays;
public class EmployeeManagementSystem {
    private Employee[] employees;
    private int size;
    public EmployeeManagementSystem(int capacity) {
        employees = new Employee[capacity];
        size = 0;
    }
    public void addEmployee(Employee employee) {
        if (size < employees.length) {
            employees[size++] = employee;
            System.out.println("Employee added: " + employee);
        } else {
            System.out.println("No more capacity to add new employees.");
        }
    }
    public Employee searchEmployee(int employeeId) {
        for (int i = 0; i < size; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                return employees[i];
            }
        }
        return null;
    }
    public void deleteEmployee(int employeeId) {
        for (int i = 0; i < size; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                employees[i] = employees[--size];
                employees[size] = null;
                System.out.println("Employee removed with ID: " + employeeId);
                return;
            }
        }
        System.out.println("Employee with ID " + employeeId + " not found.");
```

```
    }
    public void traverseEmployees() {
        System.out.println("Employee List:");
        for (int i = 0; i < size; i++) {
            System.out.println(employees[i]);
        }
    }
    public static void main(String[] args) {
        EmployeeManagementSystem ems = new EmployeeManagementSystem(10);
        ems.addEmployee(new Employee(1, "John Doe", "Manager", 50000.00));
        ems.addEmployee(new Employee(2, "Jane Smith", "Developer", 60000.00));
        ems.addEmployee(new Employee(3, "Emily White", "Designer", 55000.00));
        ems.traverseEmployees();
        Employee employee = ems.searchEmployee(2);
        System.out.println("Searched Employee: " + employee);
        ems.deleteEmployee(2);
        ems.traverseEmployees();
    }
}
```

4. **Analysis:**
    - Analyze the time complexity of each operation (add, search, traverse, delete).
        - **Add Operation:** $O(1)$
        - **Search Operation:** $O(n)$
        - **Traverse Operation:** $O(n)$
        - **Delete Operation:** $O(n)$
    - Discuss the limitations of arrays and when to use them.
        - **Fixed Size:** Once initialized, the size of the array cannot be changed. This can lead to wasted space or the need to resize and copy the array, which is costly.
        - **Inefficient Deletion and Insertion:** Deleting or inserting elements in the middle of the array requires shifting other elements, leading to O(n) time complexity for these operations.
        - **When to Use:** Arrays are suitable when the size of the dataset is known in advance and does not change frequently, and when fast random access to elements is required.

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).

      Singly Linked List: Each node contains data and a reference (or link) to the next node in the sequence.

      Doubly Linked List: Each node contains data and two references: one to the next node and one to the previous node.
2. **Setup:**
    - Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.

```
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;
   }
   public int getTaskId() {
      return taskId;
   }
   public String getTaskName() {
      return taskName;
   }
   public String getStatus() {
```

```
    return status;
  }
  public void setTaskId(int taskId) {
    this.taskId = taskId;
  }
  public void setTaskName(String taskName) {
    this.taskName = taskName;
  }
  public void setStatus(String status) {
    this.status = status;
  }
  @Override
  public String toString() {
    return "Task{" +
        "taskId=" + taskId +
        ", taskName='" + taskName + '\'' +
        ", status='" + status + '\'' +
        '}';
  }
}
```

3. **Implementation:**
   - Implement a singly linked list to manage tasks.
   - Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.

```
public class TaskManagementSystem {
  private class Node {
    Task task;
    Node next
    public Node(Task task) {
      this.task = task;
    }
  }
  private Node head;
```

```
public void addTask(Task task) {
    Node newNode = new Node(task);
    newNode.next = head;
    head = newNode;
    System.out.println("Task added: " + task);
}
public Task searchTask(int taskId) {
    Node current = head;
    while (current != null) {
        if (current.task.getTaskId() == taskId) {
            return current.task;
        }
        current = current.next;
    }
    return null;
}

public void deleteTask(int taskId) {
    Node current = head;
    Node previous = null;
    while (current != null && current.task.getTaskId() != taskId) {
        previous = current;
        current = current.next;
    }
    if (current != null) {
        if (previous == null) {
            head = current.next;
        } else {
            previous.next = current.next;
        }
        System.out.println("Task removed: " + current.task);
    } else {
        System.out.println("Task with ID " + taskId + " not found.");
    }
}
public void traverseTasks() {
    Node current = head;
    System.out.println("Task List:");
    while (current != null) {
        System.out.println(current.task);
        current = current.next;
    }
```

```
    }
    public static void main(String[] args) {
        TaskManagementSystem tms = new TaskManagementSystem();
        tms.addTask(new Task(1, "Design Interface", "Pending"));
        tms.addTask(new Task(2, "Implement Backend", "In Progress"));
        tms.addTask(new Task(3, "Test Application", "Pending"));
        tms.traverseTasks();
        Task task = tms.searchTask(2);
        System.out.println("Searched Task: " + task);
        tms.deleteTask(2);
        tms.traverseTasks();
    }
}
```

4. **Analysis:**
    - Analyze the time complexity of each operation.
        - **Add Operation:** O(1)
        - **Search Operation:** O(n)
        - **Traverse Operation:** O(n)
        - **Delete Operation:** O(n)
    - Discuss the advantages of linked lists over arrays for dynamic data.
        - **Dynamic Size:** Linked lists can grow and shrink in size dynamically, making them more flexible than arrays.
        - **Efficient Insertions/Deletions:** Insertions and deletions are more efficient in linked lists as they do not require shifting elements like in arrays.
        - **Memory Usage:** Linked lists can be more memory-efficient for sparse data as they do not need to allocate memory for unused elements like arrays.

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.

      Linear Search: Linear search is a straightforward algorithm that checks each element in the list sequentially until the desired element is found or the list is exhausted.

      Binary Search: Binary search is a more efficient algorithm that repeatedly divides a sorted list in half to locate the desired element.
2. **Setup:**
    - Create a class **Book** with attributes like **bookId**, **title**, and **author**.

      ```
      class Book {
          private int bookId;
          private String title;
          private String author;
          public Book(int bookId, String title, String author) {
              this.bookId = bookId;
              this.title = title;
              this.author = author;
          }
          public int getBookId() {
              return bookId;
          }
          public String getTitle() {
              return title;
          }
          public String getAuthor() {
              return author;
      ```

```
    }
    public void setBookId(int bookId) {
        this.bookId = bookId;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    @Override
    public String toString() {
        return "Book{" +
                "bookId=" + bookId +
                ", title='" + title + '\'' +
                ", author='" + author + '\'' +
                '}';
    }
}
```

3. **Implementation:**
    - Implement linear search to find books by title.
    - Implement binary search to find books by title (assuming the list is sorted).

```
import java.util.Arrays;
public class LibraryManagementSystem {
    public static Book linearSearch(Book[] books, String title) {
        for (Book book : books) {
            if (book.getTitle().equalsIgnoreCase(title)) {
                return book;
            }
        }
        return null;
    }
```

```
    public static Book binarySearch(Book[] books, String title) {
        int left = 0;
        int right = books.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int comparison = books[mid].getTitle().compareToIgnoreCase(title);
            if (comparison == 0) {
                return books[mid];
            }
            if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
    public static void main(String[] args) {
        Book[] books = {
            new Book(1, "The Great Gatsby", "F. Scott Fitzgerald"),
            new Book(2, "1984", "George Orwell"),
            new Book(3, "To Kill a Mockingbird", "Harper Lee"),
            new Book(4, "Moby-Dick", "Herman Melville")
        };
        System.out.println("Linear Search:");
        Book foundBook = linearSearch(books, "1984");
        if (foundBook != null) {
            System.out.println("Book found: " + foundBook);
        } else {
            System.out.println("Book not found.");
        }
        Arrays.sort(books, (b1, b2) -> b1.getTitle().compareToIgnoreCase(b2.getTitle()));
        System.out.println("Binary Search:");
        foundBook = binarySearch(books, "1984");
        if (foundBook != null) {
            System.out.println("Book found: " + foundBook);
        } else {
            System.out.println("Book not found.");
        }
    }
}
```

4. **Analysis:**

   - Compare the time complexity of linear and binary search.

     1. **Linear Search:**
        - **Time Complexity:** O(n)
     2. **Binary Search:**
        - **Time Complexity:** O(log n)

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

     - **Linear Search:** Use when the dataset is small or unsorted, as it doesn't require any pre-processing (like sorting) and is straightforward to implement.
     - **Binary Search:** Use when the dataset is large and sorted. The overhead of sorting the data initially is offset by the significantly faster search times for repeated queries.

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.

      Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem. It involves a function calling itself with a subset of the original problem. Recursion can simplify complex problems by breaking them down into more manageable sub-problems. For example, calculating the factorial of a number or traversing a tree structure can be more naturally expressed with recursion.
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.

```
public class FinancialForecasting {
   public static double predictFutureValue(double currentValue, double growthRate, int years) {
      if (years == 0) {
         return currentValue;
      }
      return predictFutureValue(currentValue * (1 + growthRate / 100), growthRate, years - 1);
   }
   public static void main(String[] args) {
      double currentValue = 1000.0;
      double growthRate = 5.0;
      int years = 10;
      double futureValue = predictFutureValue(currentValue, growthRate, years);
```

```
        System.out.println("The predicted future value after " + years + " years is: " +
futureValue);
    }
}
```

4. **Analysis:**
    - Discuss the time complexity of your recursive algorithm.
      Time Complexity: O(n), where n is the number of years
    - Explain how to optimize the recursive solution to avoid excessive computation.
      Memorization: Store previously calculated values to avoid redundant calculations. This technique can transform the recursive solution into a more efficient one by saving intermediate results.