## Q1.1. Understand the Problem

**Importance of Data Structures and Algorithms**

Efficient data structures and algorithms are essential in handling large inventories because they enable:

- **Fast Access:** Quick retrieval of product information is crucial for operations like sales, restocking, and inventory audits.
- **Efficient Storage:** Optimizing how data is stored can reduce memory usage and improve performance.
- **Scalability:** As the inventory grows, the system should maintain performance and responsiveness.
- **Data Integrity:** Ensuring data is stored and manipulated accurately to prevent inconsistencies.

**Suitable Data Structures**

For an inventory management system, the following data structures are suitable:

- **ArrayList:** Good for storing products in a dynamic array, providing fast access by index. However, insertion and deletion can be slow due to the need to shift elements.
- **HashMap:** Ideal for storing products with unique product IDs as keys, allowing for fast retrieval, insertion, and deletion by key.
- **TreeMap:** Maintains products in a sorted order based on keys, offering fast retrieval and ordered traversal. However, it can be slower than a HashMap for insertion and deletion.

Given the need for efficient lookups, insertions, and deletions, a **HashMap** is a suitable choice.

## 2. Setup

**Create a New Project**

### Project Structure:

- InventoryManagement/
  - src/
    - main/
      - java/
        - com/inventory/
          - InventoryManagementSystem.java
          - Product.java
    - test/

- java/
  - com/inventory/
    - InventoryManagementSystemTest.java
- build.gradle (or pom.xml for Maven)

### Initialize the Project:

- Use a build tool like Gradle or Maven to manage dependencies and build configurations.

## 2. Implementation

```
package com.inventory;

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;
   }

   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;
    }
}
```

**Choose a Data Structure and Implement the Inventory Management System**

```
package com.inventory;

import java.util.HashMap;

public class InventoryManagementSystem {
    private HashMap<String, Product> inventory;

    public InventoryManagementSystem() {
        this.inventory = new HashMap<>();
    }

    public void addProduct(Product product) {
        inventory.put(product.getProductId(), product);
    }

    public void updateProduct(Product product) {
        if (inventory.containsKey(product.getProductId())) {
            inventory.put(product.getProductId(), product);
        } else {
            throw new IllegalArgumentException("Product not found");
        }
    }

    public void deleteProduct(String productId) {
        if (inventory.containsKey(productId)) {
            inventory.remove(productId);
        } else {
            throw new IllegalArgumentException("Product not found");
        }
    }

    public Product getProduct(String productId) {
        return inventory.get(productId);
    }
```

**}**

**4. Analysis**

**Time Complexity Analysis**

- **Add Product:**
  - **Time Complexity: O(1)**
  - **Explanation: Adding a product to a HashMap involves calculating the hash code of the product ID and placing the product in the corresponding bucket. This is generally a constant-time operation.**
- **Update Product:**
  - **Time Complexity: O(1)**
  - **Explanation: Updating a product in a HashMap is essentially the same as adding it if the key already exists. The hash code is computed, and the product is replaced in the corresponding bucket.**
- **Delete Product:**
  - **Time Complexity: O(1)**
  - **Explanation: Deleting a product from a HashMap involves computing the hash code of the product ID and removing the entry from the bucket. This is also generally a constant-time operation.**

**Optimization Strategies**

- **Rehashing: Ensure efficient handling of hash collisions by implementing a good hash function and periodically rehashing the HashMap when it grows beyond a certain load factor.**
- **Load Factor Management: Adjust the load factor of the HashMap to balance between time and space efficiency.**
- **Concurrency: Use concurrent data structures like ConcurrentHashMap if the system needs to support concurrent access by multiple threads.**
- **Batch Operations: For bulk updates or deletions, consider batch operations to minimize the overhead of repeated hash computations and resizing operations.**

## Q2.1. Understand Asymptotic Notation

**Big O Notation**

**Big O notation is a mathematical notation used to describe the upper bound of an algorithm's running time. It provides a way to classify algorithms according to how their running time or space requirements grow as the input size grows. Big O notation helps in analyzing the performance and efficiency of algorithms by focusing on the growth rate of the running time rather than the exact running time.**

- **O(1): Constant time - the running time does not change with the size of the input.**
- **O(log n): Logarithmic time - the running time grows logarithmically with the size of the input.**
- **O(n): Linear time - the running time grows linearly with the size of the input.**
- **O(n log n): Log-linear time - the running time grows in a log-linear fashion.**
- **O(n^2): Quadratic time - the running time grows quadratically with the size of the input.**

**Best, Average, and Worst-Case Scenarios for Search Operations**

- **Best Case: The scenario where the algorithm performs the fewest number of steps. For example, in a linear search, the best case is when the element being searched for is the first element.**
- **Average Case: The scenario that represents the expected number of steps taken by the algorithm. It is a measure of the algorithm's performance over a typical set of inputs.**
- **Worst Case: The scenario where the algorithm performs the maximum number of steps. For example, in a linear search, the worst case is when the element is not present in the array.**

## 2. Setup

**Create the Product Class**
**java**
**Copy code**

```
public 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;
    }
}
```

## 3. Implementation

### Linear Search Algorithm

```
public class LinearSearch {
    public Product linearSearch(Product[] products, String productName) {
        for (Product product : products) {
            if (product.getProductName().equals(productName)) {
                return product;
            }
```

```
        }
        return null;
    }
}
```

### Binary Search Algorithm

**Binary search requires the array to be sorted. Here is an implementation assuming the array is sorted by productName.**

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

public class BinarySearch {
    public Product binarySearch(Product[] products, String productName) {
        int left = 0;
        int right = products.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int result = productName.compareTo(products[mid].getProductName());

            if (result == 0) {
                return products[mid];
            } else if (result > 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }

    // Helper method to sort products by productName
    public void sortProducts(Product[] products) {
        Arrays.sort(products, Comparator.comparing(Product::getProductName));
    }
}
```

## 4. Analysis

### Time Complexity

- **Linear Search:**
    - **Best Case: O(1) - When the product is the first element in the array.**
    - **Average Case: O(n) - When the product is somewhere in the middle.**
    - **Worst Case: O(n) - When the product is the last element or not present at all.**
- **Binary Search:**
    - **Best Case: O(1) - When the product is the middle element of the array.**

- **Average Case: O(log n) - Each step reduces the search space by half.**
- **Worst Case: O(log n) - The product is not present, and the search space is halved until the end.**
-

**Suitability for the E-commerce Platform**
**For an e-commerce platform where fast search performance is crucial, binary search is more suitable due to its logarithmic time complexity. However, it requires the array to be sorted, which may involve an initial sorting overhead (O(n log n)) but pays off with faster search times for subsequent queries.**
**Linear search can be simpler and does not require sorting but can be inefficient for large datasets due to its linear time complexity.**
**Given the need for high performance and frequent searches on an e-commerce platform, binary search is generally the better choice. Additionally, leveraging more advanced data structures like hash tables or search trees could further optimize search operations.**

## Q3. 1. Understand Sorting Algorithms

**Bubble Sort**

- **Description:** A simple comparison-based 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 array is sorted.
- **Time Complexity:**
  - **Best Case:** O(n) - When the array is already sorted.
  - **Average Case:** O(n^2) - Nested loops compare and swap elements.
  - **Worst Case:** O(n^2) - When the array is sorted in reverse order.

**Insertion Sort**

- **Description:** Builds the final sorted array one element at a time. It takes each element and inserts it into its correct position in a sorted portion of the array.
- **Time Complexity:**
  - **Best Case:** O(n) - When the array is already sorted.
  - **Average Case:** O(n^2) - Nested loops insert each element into the sorted portion.
  - **Worst Case:** O(n^2) - When the array is sorted in reverse order.

**Quick Sort**

- **Description:** A divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into two sub-arrays (elements less than the pivot and elements greater than the pivot). The sub-arrays are then recursively sorted.
- **Time Complexity:**
  - **Best Case:** O(n log n) - The pivot divides the array into two equal halves.

- **Average Case:** O(n log n) - The pivot divides the array into reasonably balanced halves.
- **Worst Case:** O(n^2) - The pivot results in highly unbalanced partitions (e.g., when the array is already sorted or all elements are the same).

**Merge Sort**

- **Description:** A divide-and-conquer algorithm that divides the array into two halves, sorts them, and then merges the sorted halves.
- **Time Complexity:**
  - **Best Case:** O(n log n) - The array is divided and merged in log(n) levels.
  - **Average Case:** O(n log n) - The same as the best case since the algorithm always divides and merges.
  - **Worst Case:** O(n log n) - The same as the best case since the algorithm always divides and merges.

## 2. Setup

```
public 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;
    }
}
```

# 3. Implementation

### Bubble Sort Algorithm

```
public class BubbleSort {
    public 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

```
public class QuickSort {
    public 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 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++;
                // 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;
    }
}
```

## 4. Analysis

**Time Complexity Comparison**

- **Bubble Sort:**
  - **Best Case:** O(n) - When the array is already sorted.
  - **Average Case:** O(n^2) - Nested loops compare and swap elements.
  - **Worst Case:** O(n^2) - When the array is sorted in reverse order.
- **Quick Sort:**
  - **Best Case:** O(n log n) - The pivot divides the array into two equal halves.
  - **Average Case:** O(n log n) - The pivot divides the array into reasonably balanced halves.
  - **Worst Case:** O(n^2) - The pivot results in highly unbalanced partitions (e.g., when the array is already sorted or all elements are the same).

**Preference of Quick Sort over Bubble Sort**

- **Efficiency:** Quick Sort is generally much faster than Bubble Sort, especially for large datasets. Its average time complexity of O(n log n) makes it suitable for high-performance applications.
- **Practical Performance:** Quick Sort performs well in practice with good cache performance and low overhead, despite its worst-case scenario.
- **Suitability for Large Datasets:** Due to its divide-and-conquer approach, Quick Sort handles large datasets more efficiently than Bubble Sort, which becomes impractical for large arrays due to its O(n^2) complexity.

  In summary, Quick Sort is generally preferred over Bubble Sort for sorting operations due to its significantly better performance and efficiency, making it a more suitable choice for an e-commerce platform handling large volumes of customer orders.

## Q4.1. Understand Array Representation

Array Representation in Memory

Arrays are contiguous blocks of memory where each element is stored at a fixed offset from the beginning of the block. This fixed size allows constant-time access to any element by its index.

**Advantages of Arrays:**

1. **Direct Access:** Arrays provide O(1) time complexity for accessing elements by index.
2. **Memory Efficiency:** Arrays have low memory overhead since they don't need additional structures like pointers or references.
3. **Simplicity:** Arrays are simple to use and understand, making them suitable for straightforward tasks and small datasets.

## 2. Setup

### Create the Employee Class

```
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;
}

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

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

```
    }

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

## 3. Implementation

### Use an Array to Store Employee Records

```
public class EmployeeManagementSystem {
    private Employee[] employees;
    private int count;

    public EmployeeManagementSystem(int capacity) {
        employees = new Employee[capacity];
        count = 0;
    }

    // Add an employee
    public void addEmployee(Employee employee) {
        if (count < employees.length) {
            employees[count++] = employee;
        } else {
            System.out.println("Array is full, cannot add more employees.");
        }
    }

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

```
        }
    }
    return null;
}

// Traverse all employees
public void traverseEmployees() {
    for (int i = 0; i < count; i++) {
        System.out.println("Employee ID: " + employees[i].getEmployeeId() + ", Name: " +
employees[i].getName() + ", Position: " + employees[i].getPosition() + ", Salary: " +
employees[i].getSalary());
    }
}

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

## 4. Analysis

**Time Complexity Analysis**

- **Add Employee:**

    - **Time Complexity:** O(1) - Adding an employee to the end of the array takes constant time, assuming there is space in the array.

- **Search Employee:**
    - **Time Complexity:** O(n) - In the worst case, you may have to search through the entire array to find the employee.
- **Traverse Employees:**
    - **Time Complexity:** O(n) - Traversing the array requires visiting each element once.
- **Delete Employee:**
    - **Time Complexity:** O(n) - In the worst case, you may have to shift all elements after the deleted element one position to the left.

**Limitations of Arrays**

1. **Fixed Size:** Arrays have a fixed size, making it challenging to handle dynamic or growing datasets without allocating new arrays and copying elements.
2. **Inefficient Insertions and Deletions:** Inserting or deleting elements from the middle of the array requires shifting elements, leading to O(n) time complexity.
3. **Memory Allocation:** Arrays require contiguous memory blocks, which can be an issue for large arrays.

**When to Use Arrays**

- **Small and Fixed Size Data:** Arrays are suitable for small, fixed-size collections where the size does not change frequently.
- **Direct Access Requirements:** Arrays are ideal when you need fast, direct access to elements by index.
- **Memory Constraints:** Arrays have lower memory overhead compared to linked structures like linked lists.

For dynamic datasets or scenarios requiring frequent insertions and deletions, other data structures like linked lists, hash maps, or dynamic arrays (e.g., ArrayList in Java) may be more suitable.

## Q5. 1: Understand Linked Lists

### Singly Linked List

A singly linked list is a data structure that consists of nodes. Each node contains two parts:

1. Data (in this case, the task details).

2. A reference to the next node in the sequence.

Operations typically supported:

- Add (to the front, end, or middle).

- Delete (specific node or by position).

- Traverse (iterate through the list).

- Search (find a node based on some criteria).

### Doubly Linked List

A doubly linked list is similar to a singly linked list, but each node contains an additional reference to the previous node. This allows for traversal in both directions (forward and backward).

Operations typically supported:

- Add (to the front, end, or middle).

- Delete (specific node or by position).

- Traverse (iterate through the list in both directions).

- Search (find a node based on some criteria).

## 2: Setup

creating a `Task` class.

```
public class Task {
    int taskId;
    String taskName;
    String status;
    public Task(int taskId, String taskName, String status) {
        this.taskId = taskId;
        this.taskName = taskName;
        this.status = status;
    }
    @Override
    public String toString() {
        return "Task ID: " + taskId + ", Task Name: " + taskName + ", Status: " + status;
```

```
    }
}
```

## 3: Implementation

Now, let's implement a singly linked list to manage tasks.

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

    public Node(Task task) {
        this.task = task;
        this.next = null;
    }
}
public class TaskLinkedList {
    private Node head;
    public TaskLinkedList() {
        this.head = null;
    }
    // Add a task to the end of 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 taskId
public Task searchTask(int taskId) {
    Node current = head;
    while (current != null) {
        if (current.task.taskId == taskId) {
            return current.task;
        }
        current = current.next;
    }
    return null;
}

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

// Delete a task by taskId
public boolean deleteTask(int taskId) {
    if (head == null) {
        return false;
    }
    if (head.task.taskId == taskId) {
        head = head.next;
        return true;
```

```
    }
    Node current = head;
    while (current.next != null) {
        if (current.next.task.taskId == taskId) {
            current.next = current.next.next;
            return true;
        }
        current = current.next;
    }
    return false;
}

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

    taskList.addTask(new Task(1, "Task 1", "Pending"));
    taskList.addTask(new Task(2, "Task 2", "In Progress"));
    taskList.addTask(new Task(3, "Task 3", "Completed"));

    System.out.println("Traversing tasks:");
    taskList.traverseTasks();

    System.out.println("\nSearching for Task with ID 2:");
    Task task = taskList.searchTask(2);
    if (task != null) {
        System.out.println(task);
    } else {
        System.out.println("Task not found.");
    }

    System.out.println("\nDeleting Task with ID 1:");
```

```
        if (taskList.deleteTask(1)) {
            System.out.println("Task deleted.");
        } else {
            System.out.println("Task not found.");
        }

        System.out.println("\nTraversing tasks after deletion:");
        taskList.traverseTasks();
    }
}
```

# 4: Analysis

## Time Complexity

Add Task: O(n) - We have to traverse to the end of the list to add a new task.

Search Task: O(n) - In the worst case, we might have to search through the entire list.

Traverse Tasks: O(n) - We need to visit each node.

Delete Task: O(n) - We might need to traverse the entire list to find the task to delete.

## Advantages of Linked Lists over Arrays for Dynamic Data

Dynamic Size: Linked lists can easily grow and shrink in size by adding or removing nodes. Arrays have a fixed size, and resizing an array requires creating a new array and copying the data.

Efficient Insertions/Deletions: Linked lists allow for efficient insertions and deletions, especially when these operations are done at the beginning of the list. In an array, insertions and deletions may require shifting elements, which can be costly in terms of time complexity.

## Q6.1: Understand Search Algorithms

Linear search is the simplest search algorithm. It checks each element of the list sequentially until the desired element is found or the list ends.

Time Complexity**: O(n), where n is the number of elements in the list.

Binary search is a more efficient algorithm but requires the list to be sorted. 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, the algorithm narrows the interval to the lower half. Otherwise, it narrows it to the upper half. This process continues until the value is found or the interval is empty.

Time Complexity: O(log n), where n is the number of elements in the list.

## 2: Setup

We'll start by creating a `Book` class.

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

    public Book(int bookId, String title, String author) {
        this.bookId = bookId;
        this.title = title;
        this.author = author;
    }
    @Override
    public String toString() {
        return "Book ID: " + bookId + ", Title: " + title + ", Author: " + author;
    }
}
```

## 3: Implementation

Now, let's implement linear search and binary search to find books by title.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Library {
    private ArrayList<Book> books;

    public Library() {
        books = new ArrayList<>();
    }

    // Add a book to the library
    public void addBook(Book book) {
        books.add(book);
    }

    // Implement linear search to find books by title
    public Book linearSearchByTitle(String title) {
        for (Book book : books) {
            if (book.title.equalsIgnoreCase(title)) {
                return book;
            }
        }
        return null;
    }

    // Implement binary search to find books by title (assuming the list is sorted)
    public Book binarySearchByTitle(String title) {
```

```
    Collections.sort(books, Comparator.comparing(b -> b.title));
    int left = 0;
    int right = books.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        Book midBook = books.get(mid);
        int cmp = midBook.title.compareToIgnoreCase(title);
        if (cmp == 0) {
            return midBook;
        } else if (cmp < 0) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return null;
}

public static void main(String[] args) {
    Library library = new Library();

    library.addBook(new Book(1, "The Catcher in the Rye", "J.D. Salinger"));
    library.addBook(new Book(2, "To Kill a Mockingbird", "Harper Lee"));
    library.addBook(new Book(3, "1984", "George Orwell"));
    library.addBook(new Book(4, "The Great Gatsby", "F. Scott Fitzgerald"));

    System.out.println("Linear Search for '1984':");
    Book book = library.linearSearchByTitle("1984");
    if (book != null) {
        System.out.println(book);
```

```
        } else {
            System.out.println("Book not found.");
        }

        System.out.println("\nBinary Search for 'The Great Gatsby':");
        book = library.binarySearchByTitle("The Great Gatsby");
        if (book != null) {
            System.out.println(book);
        } else {
            System.out.println("Book not found.");
        }
    }
}
```

## 4: Analysis

Time Complexity

- **Linear Search**: O(n) - It requires scanning through the entire list.

- **Binary Search**: O(log n) - It significantly reduces the number of comparisons needed by halving the search space each time. However, this requires the list to be sorted.

When to Use Each Algorithm

Linear Search: Use when:

- The list is small.
- The list is unsorted.
- The overhead of sorting the list is not justified.

Binary Search: Use when:

- The list is large.
- The list is sorted or can be sorted without significant overhead.
- Fast search times are required and the cost of sorting the list initially is acceptable.

## Q7.1: Understand Recursive Algorithms

**Concept of Recursion**

Recursion is a method of solving a problem where the solution depends on solving smaller instances of the same problem. A recursive function calls itself with a simpler input, and this process repeats until a base condition is met, which stops the recursion.

Recursion can simplify problems by breaking them down into smaller, more manageable sub-problems. However, it can also lead to excessive computation if not optimized properly, especially if the problem has overlapping sub-problems.

## 2: Setup

We'll create a method to calculate the future value using a recursive approach. Let's assume we want to predict the future value of an investment based on a constant annual growth rate.

## 3: Implementation

```
public class FinancialForecast {


    // Recursive method to calculate future value
    public static double calculateFutureValue(double presentValue, double annualGrowthRate, int years) {
        // Base case: if years is 0, return the present value
        if (years == 0) {
            return presentValue;
        }
        // Recursive case: calculate the future value for the remaining years
        return calculateFutureValue(presentValue * (1 + annualGrowthRate), annualGrowthRate, years - 1);
    }


    public static void main(String[] args) {
        double presentValue = 1000.0; // Initial investment
        double annualGrowthRate = 0.05; // 5% annual growth rate
        int years = 10; // Number of years


        double futureValue = calculateFutureValue(presentValue, annualGrowthRate, years);
```

```
        System.out.println("Future Value after " + years + " years: " + futureValue);
    }
}
```

# 4: Analysis

## Time Complexity

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

**Optimization to Avoid Excessive Computation**

To optimize the recursive solution and avoid excessive computation, we can use memoization. Memoization stores the results of expensive function calls and returns the cached result when the same inputs occur again. However, in this simple case of future value calculation with a constant growth rate, memoization is not needed as each call is independent and has a simple calculation.

```
public class FinancialForecast {

    // Iterative method to calculate future value
    public static double calculateFutureValue(double presentValue, double annualGrowthRate, int years) {
        double futureValue = presentValue;
        for (int i = 0; i < years; i++) {
            futureValue *= (1 + annualGrowthRate);
        }
        return futureValue;
    }

    public static void main(String[] args) {
        double presentValue = 1000.0; // Initial investment
        double annualGrowthRate = 0.05; // 5% annual growth rate
        int years = 10; // Number of years

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

}

## Summary

- **Recursion** simplifies certain problems by breaking them down into smaller sub-problems but can lead to excessive computation if not optimized.
- **Time Complexity**: The recursive approach has a time complexity of O(n).
- **Optimization**: For this problem, an iterative approach is more efficient and avoids the overhead of recursive calls. Memoization is another optimization technique but is not necessary for this simple problem.