ALGORITHMS_DATA STRUCTURE

Exercise 1: Inventory Management System

Steps:

1. **Understand the Problem:**

   Why Data Structures and Algorithms are Essential in Handling Large Inventories

   Data structures and algorithms are crucial for efficiently managing large inventories because they help:

- Optimize Storage: Efficient data structures use memory effectively, allowing you to store large amounts of data without wasting space.
- Speed Up Retrieval: Appropriate data structures enable fast retrieval of information, which is essential for operations like searching for a product.
- Improve Performance: Algorithms determine the speed of operations like adding, updating, and deleting products, which can significantly impact the system's performance.

Types of Data Structures Suitable for This Problem

- **ArrayList:** Good for storing a list of products where the order matters and quick access by index is needed. However, adding/removing elements in the middle can be slow.
- **HashMap:** Excellent for fast access, addition, and deletion based on a key (e.g., productId). Provides average-case constant time complexity for these operations.

2. **Setup:**
3. **Implementation:**

```
package com.example.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;
    }

    // Getters and setters
    public String getProductId() { return productId; }
```

```java
    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; }
}
```

```java
package com.example.inventory;

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

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

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

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

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

    public void deleteProduct(String productId) {
        products.remove(productId);
    }

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

```java
import com.example.inventory.Inventory;
import com.example.inventory.Product;

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

        // Add products
        Product product1 = new Product("1", "Laptop", 10, 999.99);
        Product product2 = new Product("2", "Smartphone", 50, 499.99);
        inventory.addProduct(product1);
        inventory.addProduct(product2);

        // Update product
```

```
        product1.setPrice(949.99);
        inventory.updateProduct(product1);

        // Get product
        Product retrievedProduct = inventory.getProduct("1");
        System.out.println("Retrieved Product: " +
retrievedProduct.getProductName() + ", Price: " +
retrievedProduct.getPrice());

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

        // Attempt to get deleted product
        Product deletedProduct = inventory.getProduct("2");
        if (deletedProduct == null) {
            System.out.println("Product with ID 2 has been deleted");
        }
    }
}
```

4. **Analysis:**

Time Complexity of Each Operation

1. **Add Product:**
    - **Operation:** addProduct
    - **Time Complexity:** O(1) (Average case)

2.Update Product**:**

Time Complexity: O(1) (Average case)

3. Delete Product: **Time Complexity:** O(1) (Average case)

5. Get Product: **Time Complexity:** O(1) (Average case).
   Optimization Suggestions

- **Load Factor Management:** Ensure the HashMap load factor is well-managed to avoid excessive collisions, which can degrade performance.
- **Concurrency:** For a multi-threaded environment, consider using ConcurrentHashMap to handle concurrent access to the inventory.
- **Batch Operations:** For large-scale operations, consider batch processing to reduce the overhead of multiple single operations.

Exercise 2: E-commerce Platform Search Function

1: Understand Asymptotic Notation

Big O Notation

- Definition: Big O notation is a mathematical representation used to describe the upper bound of an algorithm's running time or space requirements in terms of the input size. It provides a high-level understanding of the algorithm's efficiency.
- Purpose: It helps in analyzing the performance of algorithms, especially for large inputs, by focusing on the most significant factors that affect scalability.

Best, Average, and Worst-Case Scenarios

- Best Case: The scenario where the algorithm performs the minimum number of operations. For search operations, it's typically when the target element is at the beginning of the array.
- Average Case: The scenario that represents the expected performance of the algorithm across all possible inputs. For search operations, it's the average number of operations required to find an element.
- Worst Case: The scenario where the algorithm performs the maximum number of operations. For search operations, it's typically when the target element is at the end of the array or not present.

2. **Setup:**
3. **Implementation:**

```
package com.example.ecommerce;

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;
    }

    // Getters
    public String getProductId() { return productId; }
    public String getProductName() { return productName; }
    public String getCategory() { return category; }
}
```

```
package com.example.ecommerce;

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

```
package com.example.ecommerce;
```

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

public class BinarySearch {
    public Product binarySearch(Product[] products, String productId) {
        Arrays.sort(products, Comparator.comparing(Product::getProductId));
        int left = 0;
        int right = products.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int cmp = products[mid].getProductId().compareTo(productId);

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

```java
import com.example.ecommerce.*;

public class Main {
    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"),
                new Product("5", "Keyboard", "Accessories")
        };

        LinearSearch linearSearch = new LinearSearch();
        BinarySearch binarySearch = new BinarySearch();

        // Test Linear Search
        Product foundProduct = linearSearch.linearSearch(products, "3");
        if (foundProduct != null) {
            System.out.println("Linear Search: Found " +
foundProduct.getProductName());
        } else {
            System.out.println("Linear Search: Product not found");
        }

        // Test Binary Search
        foundProduct = binarySearch.binarySearch(products, "3");
        if (foundProduct != null) {
            System.out.println("Binary Search: Found " +
foundProduct.getProductName());
        } else {
            System.out.println("Binary Search: Product not found");
        }
```

```
    }
}
```

4: Analysis

1. **Compare Time Complexity:**
    - **Linear Search:**
        - **Best Case:** O(1) (Element found at the beginning)
        - **Average Case:** O(n)
        - **Worst Case:** O(n) (Element found at the end or not present)
    - **Binary Search:**
        - **Best Case:** O(1) (Element found at the middle initially)
        - **Average Case:** O(log n)
        - **Worst Case:** O(log n) (Element not present or at the end of the divided intervals)
2. **Suitability Analysis:**
    - **Linear Search:**
        - Suitable for small arrays or unsorted data.
        - Simpler to implement and requires no additional preprocessing.
    - **Binary Search:**
        - More efficient for large arrays due to logarithmic time complexity.
        - Requires the array to be sorted, which adds an initial overhead but significantly improves search performance for frequent queries.

Exercise 3: Sorting Customer Orders

Steps:

1. Understand Sorting Algorithms:

    Bubble Sort

- Description: Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process repeats until the list is sorted.
- Time Complexity: O(n^2) for the worst and average case, O(n) for the best case (when the list is already sorted).

    Insertion Sort

- Description: Insertion Sort builds the sorted array one item at a time by repeatedly picking the next item and inserting it into its correct position.

- Time Complexity: O(n^2) for the worst and average case, O(n) for the best case (when the list is already sorted).

Quick Sort

- Description: Quick Sort selects a 'pivot' element and partitions the array into elements less than the pivot and elements greater than the pivot. It recursively sorts the partitions.
- Time Complexity: O(n log n) on average, O(n^2) in the worst case (when the pivot selection is poor, such as always selecting the smallest or largest element).

Merge Sort

- Description: Merge Sort divides the list into halves, recursively sorts each half, and then merges the sorted halves back together.
- Time Complexity: O(n log n) for all cases.

3. Implementation:

```
package com.example.orders;

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; }

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

```
package com.example.orders;

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()) {
```

```java
                    // Swap orders[j] and orders[j + 1]
                    Order temp = orders[j];
                    orders[j] = orders[j + 1];
                    orders[j + 1] = temp;
                }
            }
        }
    }
}
```

```java
package com.example.orders;

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++;
                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;
    }
}
```

```java
import com.example.orders.*;

import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Order[] orders = {
                new Order("1", "Arya", 250.75),
                new Order("2", "Mahi", 100.50),
                new Order("3", "Harsh", 320.40),
                new Order("4", "Rahul", 150.30),
                new Order("5", "Anu", 210.20)
        };

        BubbleSort bubbleSort = new BubbleSort();
        QuickSort quickSort = new QuickSort();

        // Test Bubble Sort
        Order[] bubbleSortedOrders = Arrays.copyOf(orders, orders.length);
```

```
        bubbleSort.bubbleSort(bubbleSortedOrders);
        System.out.println("Bubble Sorted Orders: " +
Arrays.toString(bubbleSortedOrders));

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

4. Analysis:

Compare Time Complexity:

- **Bubble Sort:**
  - **Best Case:** O(n) (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) (poor pivot selection)

Why Quick Sort is Generally Preferred:

- **Efficiency:** Quick Sort is generally faster than Bubble Sort for large datasets due to its O(n log n) average time complexity.
- **Practical Performance:** Despite its O(n^2) worst-case complexity, Quick Sort's performance can be improved with good pivot selection strategies (like choosing the median or using randomized pivots).
- **Memory Usage:** Quick Sort typically has better memory usage compared to other O(n log n) algorithms like Merge Sort, as it is an in-place sorting algorithm.

Exercise 4: Employee Management System

Steps:

1. **Understand Array Representation:**

   Array Representation in Memory

- Contiguous Memory Allocation: Arrays are stored in contiguous memory locations. This means that elements are stored sequentially in adjacent memory addresses.
- Fixed Size: The size of an array is fixed at the time of creation. This means that the number of elements it can hold is determined when the array is instantiated.
- Indexing: Arrays allow direct access to any element using its index, making retrieval operations very efficient (O(1) time complexity).

   Advantages of Arrays

- Fast Access: Constant time access to elements using indices.
- Memory Efficiency: Minimal overhead compared to some other data structures.
- Predictability: Memory allocation is straightforward and predictable.

2. Setup:

3. Implementation:

```
package com.example.employees;

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;
    }

    public int getEmployeeId() { return employeeId; }
    public String getName() { return name; }
    public String getPosition() { return position; }
    public double getSalary() { return salary; }

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

```
package com.example.employees;

import java.util.Arrays;

public class EmployeeManager {
    private Employee[] employees;
    private int size;

    public EmployeeManager(int capacity) {
        employees = new Employee[capacity];
        size = 0;
    }

    // Add an employee
    public void addEmployee(Employee employee) {
        if (size >= employees.length) {
```

```java
            System.out.println("Array is full, cannot add more 
employees.");
            return;
        }
        employees[size++] = employee;
    }

    // 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 and print 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) {
        int index = -1;
        for (int i = 0; i < size; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                index = i;
                break;
            }
        }

        if (index == -1) {
            System.out.println("Employee not found.");
            return;
        }

        for (int i = index; i < size - 1; i++) {
            employees[i] = employees[i + 1];
        }

        employees[--size] = null;
    }
}
```

```java
import com.example.employees.*;

public class Main {
    public static void main(String[] args) {
        EmployeeManager manager = new EmployeeManager(10);

        Employee emp1 = new Employee(1, "Alice", "Manager", 75000);
        Employee emp2 = new Employee(2, "Bob", "Developer", 50000);
        Employee emp3 = new Employee(3, "Charlie", "Analyst", 60000);

        manager.addEmployee(emp1);
```

```
        manager.addEmployee(emp2);
        manager.addEmployee(emp3);

        System.out.println("All Employees:");
        manager.traverseEmployees();

        System.out.println("\nSearching for Employee with ID 2:");
        Employee searchedEmployee = manager.searchEmployee(2);
        System.out.println(searchedEmployee);

        System.out.println("\nDeleting Employee with ID 2:");
        manager.deleteEmployee(2);
        manager.traverseEmployees();
    }
}
```

4. **Analysis:**

   Analyze Time Complexity:

- Add Operation:
  - Time Complexity: O(1) (inserting at the end of the array, assuming there is space).
- Search Operation:
  - Time Complexity: O(n) (linear search through the array).
- Traverse Operation:
  - Time Complexity: O(n) (printing all elements).
- Delete Operation:
  - Time Complexity: O(n) (linear search to find the element, then shifting elements).

Limitations of Arrays:

- Fixed Size: Arrays have a fixed size, which can be a limitation if the number of elements changes frequently.
- Inefficient Deletions/Insertions: Deleting or inserting elements (other than at the end) requires shifting elements, which can be inefficient (O(n) time complexity).
- Memory Allocation: Allocating large arrays may lead to memory wastage if not fully utilized

Exercise 5: Task Management System.

Steps:

1. **Understand Linked Lists:**

   Types of Linked Lists

1. **Singly Linked List:**
   - Each node contains data and a reference to the next node.

- The last node points to null, indicating the end of the list.
- Operations: Add, delete, search, and traverse.

2. **Doubly Linked List:**
   - Each node contains data, a reference to the next node, and a reference to the previous node.
   - Allows traversal in both directions.
   - More complex and requires more memory compared to singly linked lists.

2. **Setup:.**

3. **Implementation:**

```
package com.example.tasks;

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;
    }

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

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

```
package com.example.tasks;

public class TaskManager {
    private Node head;

    private class Node {
        Task task;
        Node next;

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

```java
    // Add a task to the linked 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 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 ID
    public void deleteTask(int taskId) {
        if (head == null) {
            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) {
            current.next = current.next.next;
        }
    }
}
```

```java
import com.example.tasks.*;

public class Main {
    public static void main(String[] args) {
        TaskManager manager = new TaskManager();

        Task task1 = new Task(1, "Design Database", "Pending");
        Task task2 = new Task(2, "Develop API", "In Progress");
        Task task3 = new Task(3, "Test Application", "Pending");

        manager.addTask(task1);
        manager.addTask(task2);
        manager.addTask(task3);

        System.out.println("All Tasks:");
        manager.traverseTasks();

        System.out.println("\nSearching for Task with ID 2:");
        Task searchedTask = manager.searchTask(2);
        System.out.println(searchedTask);

        System.out.println("\nDeleting Task with ID 2:");
        manager.deleteTask(2);
        manager.traverseTasks();
    }
}
```

4. **Analysis:**

Analyze Time Complexity:

- Add Operation:
    - Time Complexity: O(n) (inserting at the end of the list requires traversing the list).
- Search Operation:
    - Time Complexity: O(n) (linear search through the list).
- Traverse Operation:
    - Time Complexity: O(n) (printing all elements).
- Delete Operation:
    - Time Complexity: O(n) (linear search to find the element, then adjusting pointers).

Advantages of Linked Lists Over Arrays:

- Dynamic Size: Linked lists can grow and shrink dynamically without needing to allocate or deallocate memory for the entire list.
- Efficient Insertions/Deletions: Insertions and deletions at the beginning or middle of the list are more efficient (O(1) for beginning insertions/deletions) compared to arrays, which require shifting elements (O(n) time complexity).

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:**
2. **Setup:**.
3. **Implementation:**

```java
package com.example.library;

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;
    }

    public int getBookId() { return bookId; }
    public String getTitle() { return title; }
    public String getAuthor() { return author; }

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

```java
package com.example.library;

import java.util.List;

public class Library {
    private List<Book> books;

    public Library(List<Book> books) {
        this.books = books;
    }

    public Book linearSearchByTitle(String title) {
        for (Book book : books) {
            if (book.getTitle().equalsIgnoreCase(title)) {
                return book;
            }
        }
```

```java
        return null;
    }

    public Book binarySearchByTitle(String 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.getTitle().compareToIgnoreCase(title);

            if (cmp == 0) {
                return midBook;
            } else if (cmp < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

```java
import com.example.library.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<Book> books = Arrays.asList(
                new Book(1, "The Great Gatsby", "F. Scott Fitzgerald"),
                new Book(2, "To Kill a Mockingbird", "Harper Lee"),
                new Book(3, "1984", "George Orwell"),
                new Book(4, "Pride and Prejudice", "Jane Austen"),
                new Book(5, "The Catcher in the Rye", "J.D. Salinger")
        );

        // Assume books list is sorted by title for binary search
        Collections.sort(books, Comparator.comparing(Book::getTitle));

        Library library = new Library(books);

        System.out.println("Linear Search:");
        Book linearSearchResult = library.linearSearchByTitle("1984");
        System.out.println(linearSearchResult);

        System.out.println("\nBinary Search:");
        Book binarySearchResult = library.binarySearchByTitle("1984");
        System.out.println(binarySearchResult);
    }
}
```

4. **Analysis:**

Compare Time Complexity:

- Linear Search:
    - Time Complexity: O(n)
    - Suitable for: Unsorted or small datasets.
- Binary Search:
    - Time Complexity: O(log n)
    - Suitable for: Large, sorted datasets.

When to Use Each Algorithm:

- Linear Search: Use when the dataset is unsorted or the list size is small, as it does not require sorting and is straightforward to implement.
- Binary Search: Use when the dataset is large and sorted, as it significantly reduces the number of comparisons needed to find the desired element.

Exercise 7: Financial Forecasting

Steps:

1. **Understand Recursive Algorithms:**

   Recursion is a process where a function calls itself directly or indirectly to solve a problem. It can simplify certain problems by breaking them down into smaller sub-problems of the same type.

- Base Case: The condition under which the recursion stops.
- Recursive Case: The part of the function that calls itself with a modified argument.

2. **Setup:**
3. **Implementation:**

```
package com.example.financialforecasting;

public class FinancialForecasting {

    // Recursive method to predict future value
    public static double predictFutureValue(double initialValue,
double growthRate, int years) {
        // Base case: if years is 0, return the initial value
        if (years == 0) {
            return initialValue;
        }

        // Recursive case: calculate future value for the next year
        double previousYearValue = predictFutureValue(initialValue,
growthRate, years - 1);
        return previousYearValue * (1 + growthRate);
    }
```

```java
    public static void main(String[] args) {
        double initialValue = 1000.0;
        double growthRate = 0.05; // 5% annual growth rate
        int years = 10;

        double futureValue = predictFutureValue(initialValue,
growthRate, years);
        System.out.println("Future value after " + years + " years: "
+ futureValue);
    }
}
```

```java
package com.example.financialforecasting;

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

public class FinancialForecastingOptimized {

    // Recursive method with memoization to predict future value
    public static double predictFutureValue(double initialValue, double
growthRate, int years, Map<Integer, Double> memo) {
        // Base case: if years is 0, return the initial value
        if (years == 0) {
            return initialValue;
        }

        // Check if the result for the current number of years is already
computed
        if (memo.containsKey(years)) {
            return memo.get(years);
        }

        // Recursive case: calculate future value for the next year
        double previousYearValue = predictFutureValue(initialValue,
growthRate, years - 1, memo);
        double currentValue = previousYearValue * (1 + growthRate);

        // Store the result in the memoization map
        memo.put(years, currentValue);

        return currentValue;
    }

    public static void main(String[] args) {
        double initialValue = 1000.0;
        double growthRate = 0.05; // 5% annual growth rate
        int years = 10;

        // Create a memoization map to store intermediate results
        Map<Integer, Double> memo = new HashMap<>();

        double futureValue = predictFutureValue(initialValue, growthRate,
years, memo);
        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 method is called recursively n times.

Optimizing the Recursive Solution:

- Memoization: Store the results of previous calculations to avoid redundant computations. This technique can transform a simple recursive solution into a more efficient one by reducing the time complexity**.**

.

Submitted By:-

ARYA KUMAR BHANJA

Superset Id - 5014081