**Exercise 1: Inventory Management System**

**1. Understand the Problem:**

1. **Explain why data structures and algorithms are essential in handling large inventories.**

In an inventory management system, efficient data storage and retrieval are crucial for handling large volumes of data so we require data structure and algorithms as :

* Performance: We need to deal with real-world complex problems and are expected to complete a particular task as soon as possible with efficiency and by using fewer resources. Tasks just as adding, searching, updating, and deleting items can be performed easily and quickly irrespective of the growing size of the inventory.
* Scalability: With the growth of the inventory, the problems may become larger and harder but with the right tools we can handle the increasing load without dropping our performance.
* To handle memory: One of the main issues with increasing inventory size is memory management, with efficient data structure we can prevent excessive use of memory and system crashes.

**b. Discuss the types of data structures suitable for this problem.**

For an inventory management system, some of the data structures that are suitable :

* Array List: It is efficient in maintaining a list of products, accessed by index but not good for frequent insertion and deletion.
* HashMap: It can be said as an excellent choice for an inventory system. A hash table stores key-value pairs, where the key is the product ID and the value is the product object, it can be used for fast lookups, insertion, and deletion based on a unique key which makes it suitable for searching and updating.

**2. Setup**

1. **Create a New Project**:

We can create it by using Integrated Development Environment (IDE) (e.g., IntelliJ IDEA, Eclipse, VS Code). Now we create a new project named "InventoryManagementSystem".

**3. Implementation**

**a. Define a class Product with attributes like productId, productName, quantity, and price.**

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;

}

public String toString() {

return "Product {" +

“productId=’” + productId + '\'' +

", productName='" + productName + '\'' +

", quantity=" + quantity +

", price=" + price +

'}';

}

}

**b.Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).**

For implementation, we will use a HashMap to store products. The key will be the productId and the value will be the Product object.

**c.Implement methods to add, update, and delete products from the inventory.**

import java.util.HashMap;

public class InventoryManager {

private HashMap<String, Product> inventory;

public InventoryManager() {

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 {

System.out.println("Product not found in the inventory.");

}

}

public void deleteProduct(String productId) {

if (inventory.containsKey(productId)) {

inventory.remove(productId);

} else {

System.out.println("Product not found in the inventory.");

}

}

public void displayInventory() {

for (Product product : inventory.values()) {

System.out.println(product);

}

}

}

**4.Analysis:**

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

* Add Product:

Time Complexity: O(1)

Explanation: Inserting an element into a HashMap takes constant time on average.

* Update Product:

Time Complexity: O(1)

Explanation: Updating an element in a HashMap also takes constant time on average. It first checks if the key exists, which is an O(1) operation, and then updates the value.

* Delete Product:

Time Complexity: O(1)

Explanation: Deleting an element from a HashMap takes constant time on average. It checks if the key exists and then removes it.

**b.Discuss how you can optimize these operations.**

1. Since all basic operations (add, update, delete) are already O(1) due to the use of a HashMap, optimization is inherently achieved.
2. Ensuring that the hashCode and equals methods of the Product class (if we ever override them) are implemented efficiently will maintain these time complexities.
3. Regularly cleaning up and resizing the HashMap (which Java's internal implementation handles automatically) can ensure it operates efficiently.

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

**1. Understand Asymptotic Notation:**

1. **Explain Big O notation and how it helps in analyzing algorithms.**

First of all Big-O notation is a mathematical concept that is used to describe the upper bound running time or space, the time requirement of an algorithm, performance or complexity of an algorithm. Specifically, it describes the worst-case scenario in terms of time or space complexity.

 Big O notation only describes the asymptotic behaviour(describes how a function behaves as the input approaches infinity) of a function, not its exact value. The Big O notation can be used to compare the efficiency of different algorithms or data structures.

Some of the Big O Notations

1. **O(1)**: Constant time complexity
2. **O(log n)**: Logarithmic time complexity
3. **O(n)**: Linear time complexity
4. **O(n log n)**: Line arithmetic time complexity
5. **O(n^2)**: Quadratic time complexity
6. **O(2^n)**: Exponential time complexity
7. **O(n!)**: Factorial time complexity

Big O notation helps to analyze in  several ways:

Comparing Algorithms: By expressing the time complexity of different algorithms using Big O notation, we can compare their efficiency and choose the best one. By focusing on the growth rates, we can determine which algorithm will perform better as the input size increases. Which can help us optimize a problem .

Scalability: Big O notation is important for developing a scalable system as it helps you understand how an algorithm's performance will change by understanding the growth rate as the input size scales up.

Optimization and Bottleneck: By identifying the time complexity of an algorithm, we can focus on optimizing the parts that have the greatest impact on performance.

Also by analyzing the Big O complexity of different parts of an algorithm helps in identifying the bottlenecks.

1. **Describe the best, average, and worst-case scenarios for search operations.**

Best, Average and Worst-Case Scenarios for Search Operations are as follows :

1. Best Case

The best-case scenario happens where the search operation completes in the minimum time possible. This happens when the desired element is found at the earliest possible position.

* Linear Search

O(1) - The target element is the first element in the list.

* Binary Search

O(1) -The target element is found in the middle of the array on the first check.

2. Average-Case

 The average-case scenario happens when the performance of the search operation considers all possible inputs.

* Linear Search

O(n) - The target element is somewhere in the middle of the list and half of the elements need to be checked.

* Binary Search

 O(log n) - The target element is equally likely to be anywhere in the array, resulting in a logarithmic number of comparisons on average.

3. Worst-Case

The worst-case scenario happens when the search operation takes the maximum time. This usually happens when the wanted element is located at the last position.

* Linear Search

 O(n) - The target element is the last element in the list or not present at all.

* Binary Search

O(log n) - We search half of the space at each step, but the desired element is not present, requiring the maximum number of comparisons.

1. **Setup:**

**Create a class Product with attributes for searching, such as productId, productName, and category**.

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 and setters

public String getProductId() {

return productId;

}

public void setProductId(String productId) {

this.productId = productId;

}

public String getProductName() {

return productName;

}

public void setProductName(String productName) {

this.productName = productName;

}

public String getCategory() {

return category;

}

public void setCategory(String category) {

this.category = category;

}

}

1. **Implementation:**
   1. **Implement linear search and binary search algorithms.**
   2. **Store products in an array for linear search and a sorted array for binary search.**

**Linear Search**

public class LinearSearch {

public static Product linearSearch(Product[] products, String productId) {

for (Product product : products) {

if (product.getProductId().equals(productId)) {

return product;

}

}

return null;

}

}

**Binary Search**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static Product binarySearch(Product[] products, String productId) {

Arrays.sort(products, Comparator.comparing(Product::getProductId));

int left = 0, 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;

}

}

1. **Analysis:**

**a.Compare the time complexity of linear and binary search algorithms.**

* Linear Search:
  + Best-case: O(1)O(1)O(1) (when the target element is the first element in the array).
  + Average-case: O(n)O(n)O(n) (when the target element is in the middle).
  + Worst-case: O(n)O(n)O(n) (when the target element is the last element or not present).
* Binary Search:
  + Best-case: O(1)O(1)O(1) (when the target element is the middle element in the sorted array).
  + Average-case: O(log⁡n)O(\log n)O(logn) (involves dividing the array in half each time).
  + Worst-case: O(log⁡n)O(\log n)O(logn) (when the target element is not present).

**b.Discuss which algorithm is more suitable for your platform and why.**

Linear Search:

* Suitable for smaller data sets or unsorted arrays.
* Simple to implement and requires no additional sorting.

Binary Search:

* Suitable for larger data sets or when the array is already sorted.
* More efficient with a time complexity of O(log⁡n)O(\log n)O(logn).
* Requires the array to be sorted, adding an overhead of O(nlog⁡n)O(n \log n)O(nlogn) for sorting if the array isn't sorted already.

**Exercise 3: Sorting Customer Orders**

1. **Understand Sorting Algorithms:**

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

Sorting algorithms are fundamental for organizing data in a specific order. The different sorting algorithms start with :

**a. Bubble Sort**

Bubble Sort is a comparison-based sorting algorithm. It checks each element in the list provided step by step compares, adjacent, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Algorithm:

Step 1- Start from the beginning of the array provided.

Step 2-Compare and check the first two elements.

Step 3- If the first element is greater than the second, swap them. Moving to the next pair repeat the process until the end.

Step 4-Repeat the previous step again and again until no more swapping is required.

Time Complexity:

* Best Case: O(n) (when the array is already sorted)
* Average Case: O(n²)
* Worst Case: O(n²)

**b.Insertion Sort**

Insertion Sort builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as Quick Sort or Merge Sort but is efficient for small data sets and nearly sorted data.

Algorithm:

1. Start with the second element (the first element is considered sorted).
2. Compare it to the elements before it and insert it into its correct position.
3. Repeat this process for all elements in the array.

Time Complexity:

* Best Case: O(n) (when the array is already sorted)
* Average Case: O(n²)
* Worst Case: O(n²)

**c. Quick Sort**

Quick Sort uses a divide-and-conquer approach. It recursively sorts the sub-array by selecting a 'pivot' element from the array and then, partitions the other elements into two sub-arrays. According to whether they are less than or greater than the pivot.

Algorithm:

1. Pick a pivot element.
2. Partition the array such that elements less than the pivot are on the left, and elements greater than the pivot are on the right.
3. Recursively apply the above steps to the sub-arrays.

Time Complexity:

* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n²)

**d. Merge Sort**

Merge Sort is an efficient, stable, comparison-based, divide-and-conquer sorting algorithm. It divides the array into two halves, sorts each half, and then merges the sorted halves to produce the final sorted array.

Algorithm:

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two halves to form the sorted array.

Time Complexity:

* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n log n)

1. **Setup:**

**Create a class Order with attributes like orderId, customerName, and totalPrice.**

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;

}

public String toString() {

return "Order{" +

"orderId='" + orderId + '\'' +

", customerName='" + customerName + '\'' +

", totalPrice=" + totalPrice +

'}';

}

}

**3. Implementation**

**Bubble Sort**

import java.util.List;

public class BubbleSort {

public static void bubbleSortOrders(List<Order> orders) {

int n = orders.size();

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (orders.get(j).getTotalPrice() > orders.get(j + 1).getTotalPrice()) {

Order temp = orders.get(j);

orders.set(j, orders.get(j + 1));

orders.set(j + 1, temp);

}

}

}

}

}

**Quick Sort**

import java.util.List;

public class QuickSort {

public static void quickSortOrders(List<Order> orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

quickSortOrders(orders, low, pi - 1);

quickSortOrders(orders, pi + 1, high);

}

}

private static int partition(List<Order> orders, int low, int high) {

Order pivot = orders.get(high);

int i = (low - 1);

for (int j = low; j < high; j++) {

if (orders.get(j).getTotalPrice() <= pivot.getTotalPrice()) {

i++;

Order temp = orders.get(i);

orders.set(i, orders.get(j));

orders.set(j, temp);

}

}

Order temp = orders.get(i + 1);

orders.set(i + 1, orders.get(high));

orders.set(high, temp);

return i + 1;

}

}

**4. Analysis**

**Time Complexity Comparison**

* **Bubble Sort**: O(n²)
* **Quick Sort**: O(n log n) on average

**Discuss why Quick Sort is generally preferred over Bubble Sort**

* **Efficiency**: Quick Sort is significantly faster on average compared to Bubble Sort, especially for larger datasets.
* **Scalability**: Quick Sort handles large arrays more efficiently due to its O(n log n) average time complexity.
* **Optimization**: Quick Sort is optimized for systems that manage memory effectively, and it can be implemented in-place with O(log n) extra space.

**Exercise 4: Employee Management System**

1. **Understand Array Representation:**

**Explain how arrays are represented in memory and their advantages**

Arrays are a collection of elements stored in contiguous memory locations. Each element in the array is identified by its index, and the index starts from 0. The memory address of each element is calculated as:

Address of element=Base address+(Index×Size of each element)

Advantages of Arrays

* Contiguous Memory Allocation: Allows for efficient access to elements using their index.
* Fixed Size: Memory is allocated at the time of array creation, leading to predictable memory usage.
* Efficient Indexing: Direct access to any element using its index in constant time O(1).

1. **Setup**

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;

}

public String toString() {

return "Employee{" +

"employeeId=" + employeeId +

", name='" + name + '\'' +

", position='" + position + '\'' +

", salary=" + salary +

'}';

}

}

1. **Implementation**

public class EmployeeManagementSystem {

private Employee[] employees;

private int count;

public EmployeeManagementSystem(int capacity) {

employees = new Employee[capacity];

count = 0;

}

public void addEmployee(Employee employee) {

if (count < employees.length) {

employees[count++] = employee;

} else {

System.out.println("Array is full, cannot add more employees.");

}

}

public Employee searchEmployee(int employeeId) {

for (int i = 0; i < count; i++) {

if (employees[i].getEmployeeId() == employeeId) {

return employees[i];

}

}

return null;

}

public void traverseEmployees() {

for (int i = 0; i < count; i++) {

System.out.println(employees[i]);

}

}

public void deleteEmployee(int employeeId) {

for (int i = 0; i < count; i++) {

if (employees[i].getEmployeeId() == employeeId) {

for (int j = i; j < count - 1; j++) {

employees[j] = employees[j + 1];

}

employees[--count] = null;

return;

}

}

System.out.println("Employee not found.");

}

}

**4. Analysis**

**a.Time Complexity of Each Operation**

* **Add Employee**: O(1) if there is space available.
* **Search Employee**: O(n) in the worst case (linear search).
* **Traverse Employees**: O(n).
* **Delete Employee**: O(n) in the worst case (need to shift elements).

**b.Limitations of Arrays**

* **Fixed Size**: The size of an array is fixed at the time of creation and cannot be changed. This can lead to wastage of memory if the array is not fully utilized, or insufficient memory if more elements need to be added.
* **Inefficient Deletions and Insertions**: Deletions and insertions can be inefficient because they may require shifting elements.
* **Sequential Memory Allocation**: Arrays require contiguous memory allocation, which might not be available for large arrays, leading to memory allocation issues.

**When to Use Arrays**

* When the number of elements is known and fixed.
* When fast access to elements is required using indices.
* When memory overhead of dynamic data structures like linked lists is not acceptable.

**Exercise 5: Task Management System**

1. **Understand Linked Lists:**

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

Singly Linked List

A Singly Linked List is a linear collection of data elements, called nodes, where each node points to the next node by means of a pointer. The list terminates at a node whose next pointer points to null.

* Advantages:
  + Dynamic size.
  + Ease of insertion/deletion.
* Disadvantages:
  + No backward traversal.
  + Extra memory for storing pointers.

Doubly Linked List

A Doubly Linked List is similar to a Singly Linked List but each node contains an additional pointer that points to the previous node.

* Advantages:
  + Can be traversed in both directions.
  + More complex operations like deletion of a given node can be done more efficiently.
* Disadvantages:
  + Extra memory for storing two pointers.
  + More complex implementation.

1. **Setup:**

**Create a class Task with attributes like taskId, taskName, and status.**

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;

}

public String toString() {

return "Task{" +

"taskId=" + taskId +

", taskName='" + taskName + '\'' +

", status='" + status + '\'' +

'}';

}

}

1. **Implementation:**

**a.Implement a singly linked list to manage tasks.**

class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

**b.Implement methods to add, search, traverse, and delete tasks in the linked list.**

public class TaskManagementSystem {

private Node head;

public TaskManagementSystem() {

this.head = null;

}

public void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public Task searchTask(int taskId) {

Node temp = head;

while (temp != null) {

if (temp.task.getTaskId() == taskId) {

return temp.task;

}

temp = temp.next;

}

return null;

}

public void traverseTasks() {

Node temp = head;

while (temp != null) {

System.out.println(temp.task);

temp = temp.next;

}

}

public void deleteTask(int taskId) {

if (head == null) {

return;

}

if (head.task.getTaskId() == taskId) {

head = head.next;

return;

}

Node temp = head;

while (temp.next != null && temp.next.task.getTaskId() != taskId) {

temp = temp.next;

}

if (temp.next != null) {

temp.next = temp.next.next;

}

}

}

1. **Analysis:**
   1. **Analyze the time complexity of each operation.**

Time Complexity of Each Operation

* Add Task: O(n) (since it involves traversing to the end of the list).
* Search Task: O(n) (involves traversing the list).
* Traverse Tasks: O(n) (involves traversing the list).
* Delete Task: O(n) (involves searching and then removing the task).
  1. **Discuss the advantages of linked lists over arrays for dynamic data.**
* Dynamic Size: Linked lists can grow and shrink in size dynamically, which is not possible with arrays.
* Efficient Insertions/Deletions: Insertions and deletions are more efficient in linked lists compared to arrays, as they do not require shifting elements.
* Memory Utilization: Linked lists use memory more efficiently for dynamic data, as they do not require a predefined size and avoid memory wastage.

**Exercise 6: Library Management System**

**1.Understand Search Algorithms:**

**Explain linear search and binary search algorithms.**

Linear Search

Linear search is a simple search algorithm that checks each element in the list sequentially until the target element is found or the list ends.

* Time Complexity: O(n) in the worst and average cases.
* Space Complexity: O(1)

Binary Search

Binary search is a more efficient search algorithm that works on sorted lists. It repeatedly divides the search interval in half. If the value of the target element is less than the middle element, it searches the left half. Otherwise, it searches the right half.

* Time Complexity: O(log n) in the worst and average cases.
* Space Complexity: O(1) (iterative implementation) or O(log n) (recursive implementation**)**

1. **Setup:**

**Create a class Book with attributes like bookId, title, and author.**

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;

}

public String toString() {

return "Book{" +

"bookId=" + bookId +

", title='" + title + '\'' +

", author='" + author + '\'' +

'}';

}

}

1. **Implementation:**
   1. Implement linear search to find books by title.

import java.util.List;

public class LibraryManagementSystem {

private List<Book> books;

public LibraryManagementSystem(List<Book> books) {

this.books = books;

}

public Book linearSearchByTitle(String title) {

for (Book book : books) {

if (book.getTitle().equalsIgnoreCase(title)) {

return book;

}

}

return null;

}

}

* 1. Implement binary search to find books by title (assuming the list is sorted).

import java.util.Collections;

import java.util.Comparator;

public class LibraryManagementSystem {

private List<Book> books;

public LibraryManagementSystem(List<Book> books) {

this.books = books;

Collections.sort(this.books, Comparator.comparing(Book::getTitle));

}

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;

}

}

1. **Analysis:**

a.Compare the time complexity of linear and binary search.

Time Complexity Comparison

* Linear Search: O(n)
* Binary Search: O(log n)

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

**Linear Search**:

* Suitable for unsorted or small data sets.
* Simple to implement and understand.
* No preconditions on the data set order.

**Binary Search**:

* Suitable for sorted data sets.
* More efficient for larger data sets due to logarithmic time complexity.
* Requires the data set to be sorted, which may involve additional preprocessing time.

**Exercise 7: Financial Forecasting**

1. **Understand Recursive Algorithms:**

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

Recursion is a method of solving problems where a function calls itself as a subroutine. This allows the function to be repeated several times as it processes each subset of the problem. The key components of a recursive function are:

* **Base Case**: The condition under which the recursion ends.
* **Recursive Case**: The part where the function calls itself.

**Benefits of Recursion**

* Simplifies the code for problems that can be broken down into similar subproblems.
* Makes the code more readable and easier to understand for some complex problems.

**Drawbacks of Recursion**

* Can lead to excessive use of memory and stack overflow if not implemented carefully.
* May be less efficient than iterative solutions for certain problems.

1. **Setup:**

Create a method to calculate the future value using a recursive approach.

public class FinancialForecasting {

public static double calculateFutureValue(double presentValue, double growthRate, int periods) {

if (periods == 0) {

return presentValue;

}

return calculateFutureValue(presentValue \* (1 + growthRate), growthRate, periods - 1);

}

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05;

int periods = 10;

double futureValue = calculateFutureValue(presentValue, growthRate, periods);

System.out.println("Future Value: " + futureValue);

}

}

1. **Implementation:**

Implement a recursive algorithm to predict future values based on past growth rates**.**

The above implementation demonstrates the recursive calculation of future values based on a constant growth rate.

public class FinancialForecasting {

public static double calculateFutureValue(double presentValue, double growthRate, int periods) {

if (periods == 0) {

return presentValue;

}

return calculateFutureValue(presentValue \* (1 + growthRate), growthRate, periods - 1);

}

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05;

int periods = 10;

double futureValue = calculateFutureValue(presentValue, growthRate, periods);

System.out.println("Future Value: " + futureValue);

}

}

1. **Analysis:**
   1. Discuss the time complexity of your recursive algorithm.

* **Time Complexity**: O(n)

The recursive algorithm will make n calls before reaching the base case.

* **Space Complexity**: O(n)
  1. Explain how to optimize the recursive solution to avoid excessive computation.

To avoid excessive computation and stack overflow, we can use Memoization or convert the recursive approach to an Iterative Approach.

**Memoization**

Memoization involves storing the results of expensive function calls and reusing the cached result when the same inputs occur again.