**WEEK-1**

**Data Structures and Algorithms**

**Exercise 1: Inventory Management System**

**Scenario:**

**1.Understand the Problem:**

**Importance of Data Structures and Algorithms:**

* **Efficiency**: For managing a large inventory, operations like adding, updating, and deleting products need to be efficient. Choosing the right data structure ensures that these operations are performed quickly.
* **Scalability**: As the inventory grows, operations should remain manageable in terms of time and space complexity.
* **Data Retrieval**: Efficient data retrieval mechanisms are crucial for quickly accessing and displaying product information.

**Suitable Data Structures:**

* **ArrayList:**
  + **Pros:** Dynamic resizing and easy indexing.
  + **Cons:** Slower insertions and deletions, as these may require shifting elements.
* **HashMap:**
  + **Pros:** Average O(1) time complexity for insertions, deletions, and lookups due to hashing.
  + **Cons:** Requires handling hash collisions.
* **Binary Search Tree (BST) or Balanced Trees (e.g., AVL Tree):**
  + **Pros:** O(log n) time complexity for search, insertion, and deletion operations, while maintaining sorted order.
  + **Cons:** More complex to implement and manage compared to HashMap.

**2.Setup**

**Create a New Project:**

* Use an IDE like IntelliJ IDEA, Eclipse, or Visual Studio Code.
* Create a new project directory and set up version control (e.g., Git).

**3.Implementation:**

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

class Product {

private int productId;

private String productName;

private int quantity;

private double price;

public Product(int productId, String productName, int quantity, double price) {

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public int getQuantity() {

return quantity;

}

public double getPrice() {

return price;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public void setPrice(double price) {

this.price = price;

}

@Override

public String toString() {

return "Product ID: " + productId + ", Name: " + productName + ", Quantity: " + quantity + ", Price: $" + price;

}

}

class InventorySystem {

private Map<Integer, Product> inventory;

public InventorySystem() {

this.inventory = new HashMap<>();

}

public void addProduct(Product product) {

inventory.put(product.getProductId(), product);

}

public void updateProduct(int productId, int quantity, double price) {

Product product = inventory.get(productId);

if (product != null) {

product.setQuantity(quantity);

product.setPrice(price);

} else {

System.out.println("Product with ID " + productId + " not found.");

}

}

public void deleteProduct(int productId) {

inventory.remove(productId);

}

public Product getProduct(int productId) {

return inventory.get(productId);

}

public void displayInventory() {

System.out.println("Product Inventory:");

System.out.printf("%-10s %-20s %-10s %-10s\n", "Product ID", "Product Name", "Quantity", "Price");

System.out.println("-------------------------------------------------------------");

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

System.out.printf("%-10d %-20s %-10d %-10.2f\n", product.getProductId(), product.getProductName(), product.getQuantity(), product.getPrice());

}

}

}

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

InventorySystem inventorySystem = new InventorySystem();

inventorySystem.addProduct(new Product(1, "Laptop", 10, 999.99));

inventorySystem.addProduct(new Product(2, "Smartphone", 20, 499.99));

inventorySystem.addProduct(new Product(3, "Tablet", 15, 299.99));

inventorySystem.displayInventory();

System.out.print("Enter the product ID to update: ");

int productId = scanner.nextInt();

System.out.print("Enter new quantity: ");

int quantity = scanner.nextInt();

System.out.print("Enter new price: ");

double price = scanner.nextDouble();

inventorySystem.updateProduct(productId, quantity, price);

inventorySystem.displayInventory();

System.out.print("Enter the product ID to delete: ");

productId = scanner.nextInt();

inventorySystem.deleteProduct(productId);

inventorySystem.displayInventory();

scanner.close();

}

}

**4. Analysis:  
Time Complexity:**

* **Add Operation:**
  + **HashMap:** O(1) on average. Adding a new product involves hashing the key and storing it in the map.
* **Update Operation:**
  + **HashMap:** O(1) on average. Updating involves replacing the value associated with a key.
* **Delete Operation:**
  + **HashMap:** O(1) on average. Removing a product involves hashing and deleting the key-value pair.

**Optimizations:**

* **HashMap:** Ensure a good hash function and handle collisions to maintain O(1) performance.
* **Memory Management:** Consider using more space-efficient data structures or compressing data if memory usage becomes a concern.
* **Concurrency:** Use concurrent data structures or synchronize access if multiple threads are involved in accessing or modifying the inventory.

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

* + 1. **Understand Asymptotic Notation:**

**Big O Notation:** Big O notation is a mathematical representation that describes the upper bound of an algorithm's runtime or space requirements in terms of the size of the input. It provides a way to compare the efficiency of different algorithms, especially as the input size grows.

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

* **Best Case:** The scenario where the search operation performs the minimum number of steps.
* **Average Case:** The expected scenario where the search operation performs a typical number of steps.
* **Worst Case:** The scenario where the search operation performs the maximum number of steps.
  + 1. **Setup:**

class Product {

private int productId;

private String productName;

private String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

@Override

public String toString() {

return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

* + 1. **Implementation:**

import java.util.Arrays;

import java.util.Scanner;

public class ECommercePlatformSearch {

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, "Shirt", "Clothing"),

new Product(5, "Jeans", "Clothing"),

new Product(6, "Sneakers", "Footwear")

};

// Sort products array for binary search

Arrays.sort(products, (a, b) -> a.getProductId() - b.getProductId());

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the product ID to search: ");

int productId = scanner.nextInt();

System.out.println("Performing linear search...");

Product linearSearchResult = linearSearch(products, productId);

if (linearSearchResult != null) {

System.out.println("Product found: " + linearSearchResult);

} else {

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

}

System.out.println("Performing binary search...");

Product binarySearchResult = binarySearch(products, productId);

if (binarySearchResult != null) {

System.out.println("Product found: " + binarySearchResult);

} else {

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

}

scanner.close();

}

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (products[mid].getProductId() == productId) {

return products[mid];

} else if (products[mid].getProductId() < productId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

* + 1. **Analysis:**
* **Linear Search:**
  + **Best Case:** O(1) (product found at the first position)
  + **Average Case:** O(n) (product found in the middle)
  + **Worst Case:** O(n) (product found at the last position or not at all)
* **Binary Search:**
  + **Best Case:** O(1) (product found at the middle)
  + **Average Case:** O(log n)
  + **Worst Case:** O(log n)

**Suitability for the Platform:**

* **Linear Search:** Suitable for small datasets due to its simplicity, but becomes inefficient for large datasets.
* **Binary Search:** More suitable for large datasets due to its logarithmic time complexity, but requires the array to be sorted.

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

1. **Understand Array Representation:**

**Bubble Sort:** Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

* **Best Case:** O(n) (already sorted)
* **Average Case:** O(n^2)
* **Worst Case:** O(n^2)

**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 quicksort, heapsort, or merge sort.

* **Best Case:** O(n) (already sorted)
* **Average Case:** O(n^2)
* **Worst Case:** O(n^2)

**Quick Sort:** Quick Sort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

* **Best Case:** O(n log n)
* **Average Case:** O(n log n)
* **Worst Case:** O(n^2) (rare, occurs when the pivot is always the smallest or largest element)

**Merge Sort:** Merge Sort is also a divide-and-conquer algorithm. It works by dividing the array into two halves, sorting them, and then merging the sorted halves.

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

1. **Setup:**

class Order {

private int orderId;

private String customerName;

private double totalPrice;

public Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public int getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

@Override

public String toString() {

return "Order ID: " + orderId + ", Customer Name: " + customerName + ", Total Price: $" + totalPrice;

}

}

1. **Implementation:**

import java.util.Arrays;

public class OrderSorting {

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 250.00),

new Order(2, "Bob", 150.00),

new Order(3, "Charlie", 350.00),

new Order(4, "David", 100.00),

new Order(5, "Eve", 200.00)

};

System.out.println("Original Orders:");

displayOrders(orders);

System.out.println("\nOrders Sorted by Bubble Sort:");

bubbleSort(orders.clone());

displayOrders(orders);

System.out.println("\nOrders Sorted by Quick Sort:");

quickSort(orders.clone(), 0, orders.length - 1);

displayOrders(orders);

}

public static void bubbleSort(Order[] orders) {

int n = orders.length;

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

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

if (orders[j].getTotalPrice() > orders[j + 1].getTotalPrice()) {

// Swap orders[j] and orders[j + 1]

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

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

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].getTotalPrice();

int i = (low - 1);

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

if (orders[j].getTotalPrice() <= pivot) {

i++;

// 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;

}

public static void displayOrders(Order[] orders) {

for (Order order : orders) {

System.out.println(order);

}

}

}

1. **Analysis:**

**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) (rare)

**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**Steps:**

1. **Understand Array Representation:**

* **Contiguous Memory Allocation:** Arrays are stored in contiguous blocks of memory. Each element is placed one after the other, which makes it easy to compute the address of any element using its index.
* **Index-Based Access:** Array elements are accessed via indices, allowing constant-time (O(1)) access to any element. The index provides an offset from the start of the array.
* **Advantages:**
* **Fast Access:** Direct access to elements via indices is very efficient.
* **Simple Structure:** Arrays are straightforward to implement and use.
* **Predictable Memory Use:** Memory consumption is predictable because the size is fixed.

1. **Setup:**

class Employee {

constructor(employeeId, name, position, salary) {

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

}

1. **Implementation:**

class EmployeeManagementSystem {

constructor() {

this.employees = []; // Array to store Employee objects

}

// Method to add an employee

addEmployee(employee) {

this.employees.push(employee);

}

// Method to search for an employee by ID

searchEmployee(employeeId) {

return this.employees.find(emp => emp.employeeId === employeeId);

}

// Method to traverse and list all employees

listEmployees() {

return this.employees;

}

// Method to delete an employee by ID

deleteEmployee(employeeId) {

const index = this.employees.findIndex(emp => emp.employeeId === employeeId);

if (index !== -1) {

this.employees.splice(index, 1); // Remove employee from array

}

}

}

1. **Analysis:**

* **Add Operation:** addEmployee – Adding an employee to the end of the array (using push) is O(1) on average.
* **Search Operation:** searchEmployee – Searching for an employee by ID (using find) is O(n) because it requires scanning through each element until the match is found.
* **Traverse Operation:** listEmployees – Listing all employees (simply returning the array) is O(n) because it involves iterating over all elements.
* **Delete Operation:** deleteEmployee – Finding the employee (using findIndex) is O(n), and removing the element (using splice) is O(n) in the worst case due to the need to shift elements.

**Limitations of Arrays:**

* **Fixed Size:** Arrays have a fixed size once created. If the array is too small, resizing it is inefficient. JavaScript arrays handle resizing automatically, but this can still be inefficient if frequent resizing is needed.
* **Inefficient Insertions/Deletions:** Insertions and deletions (especially in the middle) can be slow because they require shifting elements.
* **Linear Time Operations:** Operations like search and delete can be slow for large datasets because they require linear time (O(n)).

**When to Use Arrays:**

* **Use Arrays When:**
  + You need fast, index-based access to elements.
  + The number of elements is relatively small or manageable.
  + The size of the data structure is fixed or changes infrequently.
* **Consider Alternatives When:**
  + You need frequent insertions and deletions.
  + You require efficient search operations with large datasets. In such cases, data structures like linked lists, hash tables, or balanced trees may be more suitable.

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**Steps:**

1. **Understand Linked Lists:**

* **Singly Linked List:** A singly linked list is a data structure that consists of a sequence of elements called nodes. Each node contains data and a reference (or link) to the next node in the sequence. The first node is called the head, and the last node points to null.
* **Doubly Linked List:** A doubly linked list is similar to a singly linked list, but each node contains two references: one to the next node and one to the previous node. This allows for traversal in both directions.

1. **Setup:**

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 + ", Name: " + taskName + ", Status: " + status;

}

}

1. **Implementation:**

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

class TaskLinkedList {

private TaskNode head;

public TaskLinkedList() {

this.head = null;

}

// Add task to the list

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

// Search task by ID

public Task searchTask(int taskId) {

TaskNode current = head;

while (current != null) {

if (current.task.taskId == taskId) {

return current.task;

}

current = current.next;

}

return null;

}

// Delete task by ID

public boolean deleteTask(int taskId) {

if (head == null) {

return false;

}

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

head = head.next;

return true;

}

TaskNode current = head;

while (current.next != null) {

if (current.next.task.taskId == taskId) {

current.next = current.next.next;

return true;

}

current = current.next;

}

return false;

}

// Traverse tasks

public void traverseTasks() {

TaskNode current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

}

public class TaskManagementSystem {

public static void main(String[] args) {

TaskLinkedList taskList = new TaskLinkedList();

// Adding tasks

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"));

// Traversing tasks

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

taskList.traverseTasks();

// Searching for a task

int searchId = 2;

Task task = taskList.searchTask(searchId);

if (task != null) {

System.out.println("Task Found: " + task);

} else {

System.out.println("Task with ID " + searchId + " not found.");

}

// Deleting a task

int deleteId = 1;

boolean isDeleted = taskList.deleteTask(deleteId);

if (isDeleted) {

System.out.println("Task with ID " + deleteId + " deleted successfully.");

} else {

System.out.println("Task with ID " + deleteId + " not found.");

}

// Traversing tasks after deletion

System.out.println("Tasks after deletion:");

taskList.traverseTasks();

}

}

1. **Analysis:**

* **Add Task:** O(n) (traverse to the end of the list)
* **Search Task:** O(n) (traverse the list to find the task)
* **Delete Task:** O(n) (traverse the list to find and remove the task)
* **Traverse Tasks:** O(n) (visit each node)

**Advantages of Linked Lists Over Arrays for Dynamic Data:**

* **Dynamic Size:** Linked lists can grow and shrink in size dynamically, while arrays have a fixed size once allocated.
* **Ease of Insertion/Deletion:** Inserting or deleting an element in a linked list is generally easier and more efficient than in an array, as it does not require shifting elements.
* **Memory Utilization:** Linked lists use memory more efficiently for dynamic data since they do not allocate memory for unused elements.

**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:**

* **Linear Search:** Linear search is a simple search algorithm that checks every element in the list sequentially until it finds the target element or reaches the end of the list.
* **Time Complexity:** O(n) in the worst case, where n is the number of elements in the list.
* **Best Case:** O(1) if the target element is at the first position.
* **Worst Case:** O(n) if the target element is at the last position or not present at all.
* **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 target value 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 case, where n is the number of elements in the list.
* **Best Case:** O(1) if the target element is the middle element.
* **Worst Case:** O(log n) if the target element is not present.

1. **Setup:**

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;

}

}

1. **Implementation:**

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

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;

}

}

class LibraryManagementSystem {

private List<Book> books;

public LibraryManagementSystem() {

books = new ArrayList<>();

}

// Add book to the library

public void addBook(Book book) {

books.add(book);

}

// 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;

}

// Binary search to find books by title (assumes the list is sorted)

public Book binarySearchByTitle(String title) {

Collections.sort(books, Comparator.comparing(book -> book.title));

int left = 0;

int right = books.size() - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

Book midBook = books.get(mid);

if (midBook.title.equalsIgnoreCase(title)) {

return midBook;

} else if (midBook.title.compareToIgnoreCase(title) < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// Display all books

public void displayBooks() {

for (Book book : books) {

System.out.println(book);

}

}

public static void main(String[] args) {

LibraryManagementSystem library = new LibraryManagementSystem();

// Adding books

library.addBook(new Book(1, "The Great Gatsby", "F. Scott Fitzgerald"));

library.addBook(new Book(2, "To Kill a Mockingbird", "Harper Lee"));

library.addBook(new Book(3, "1984", "George Orwell"));

library.addBook(new Book(4, "Pride and Prejudice", "Jane Austen"));

// Displaying all books

System.out.println("Library Books:");

library.displayBooks();

// Searching for a book by title using linear search

String searchTitle = "1984";

Book foundBook = library.linearSearchByTitle(searchTitle);

if (foundBook != null) {

System.out.println("Book found using linear search: " + foundBook);

} else {

System.out.println("Book with title '" + searchTitle + "' not found using linear search.");

}

// Searching for a book by title using binary search

foundBook = library.binarySearchByTitle(searchTitle);

if (foundBook != null) {

System.out.println("Book found using binary search: " + foundBook);

} else {

System.out.println("Book with title '" + searchTitle + "' not found using binary search.");

}

}

}

1. **Analysis:**

* **Linear Search:** O(n) in the worst case. This is because each element must be checked until the target is found or the list ends.
* **Binary Search:** O(log n) in the worst case. This is due to repeatedly dividing the search interval in half.

**When to Use Each Algorithm:**

* **Linear Search:** Use linear search for small or unsorted data sets. It is simple to implement and doesn't require the list to be sorted.
* **Binary Search:** Use binary search for large, sorted data sets. It is much more efficient than linear search for large lists due to its logarithmic time complexity.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It simplifies complex problems by breaking them down into more manageable subproblems. A recursive algorithm typically has two components:

* **Base Case:** The simplest instance of the problem which can be solved directly.
* **Recursive Case:** The function calls itself with a modified argument to solve the smaller subproblem.

**Example:** To compute the factorial of a number n:

* **Base Case:** If n is 0 or 1, return 1.
* **Recursive Case:** Return n multiplied by the factorial of n-1

1. **Setup:**

For financial forecasting, you want to predict future values based on past growth rates. Here, you can use recursion to compute the future value given a growth rate and a number of periods.

1. **Implementation:**

public class FinancialForecasting {

// Method to calculate future value using recursion

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

// Base case: if no more periods left, return the initial value

if (periods == 0) {

return initialValue;

}

// Recursive case: calculate the value for one period and recurse

double newValue = initialValue \* (1 + growthRate);

return calculateFutureValue(newValue, growthRate, periods - 1);

}

public static void main(String[] args) {

double initialValue = 1000; // Initial investment

double growthRate = 0.05; // Growth rate (5%)

int periods = 10; // Number of periods

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

System.out.println("Future Value: $" + String.format("%.2f", futureValue));

}

}

1. **Analysis:**

* **Time Complexity:** The time complexity of this recursive algorithm is O(n), where n is the number of periods. This is because each recursive call reduces the number of periods by 1, leading to n recursive calls. Each call performs a constant amount of work (multiplication and addition).
* **Space Complexity:** The space complexity is also O(n) due to the call stack used for recursion. Each recursive call adds a new frame to the stack, so with n calls, the stack grows linearly with the number of periods.