WEEK 1: DATA STRUCTURES AND ALGORITHMS

# Exercise 1: Inventory Management System

**Scenario:** To develop an inventory management system for a warehouse with efficient data storage and retrieval.

**Understand the Problem:**

**Why Data Structures and Algorithms are Essential in Handling Large Inventories**

Data structures and algorithms are essential for managing large inventories due to several key factors:

* Efficiency:Appropriate data structures facilitate efficient storage, retrieval, and manipulation of data. Algorithms ensure that operations like searching, adding, updating, and deleting items are performed optimally.
* Scalability: Efficient data structures and algorithms help maintain performance as the inventory size grows, preventing slowdowns and ensuring smooth operations.
* Memory Management:Optimized data structures make effective use of memory, preventing wastage and enabling the system to handle large datasets efficiently.
* Complexity Management: They simplify the inherent complexity of managing large datasets, making the system easier to implement and maintain.**Discuss the types of data structures suitable for this problem.**

The suitable types of data structures for an Inventory Management System are as follows:

* \*\*ArrayList:\*\* Provides dynamic arrays that can grow as needed. It is good for scenarios where the number of items is variable, and accessing and iterating through the list is frequent.
* \*\*Binary Search Tree (BST):\*\* Allows for sorted storage and efficient in-order traversal. It is suitable for scenarios requiring ordered data.
* \*\*HashMap:\*\* Also known as a Hash Table, it provides efficient key-value pair storage. It is excellent for quick lookups, additions, and deletions based on unique identifiers like product IDs.
* \*\*Linked List:\*\* Useful for constant-time insertions and deletions. However, it provides linear-time access, which might be a drawback for large datasets.

Among these, choosing the HashMap would be more appropriate in terms of time complexity for insertion, deletion, and updating operations.

**Setup:**

import java.util.HashMap;

import java.util.Map;

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 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 +

'}';

}

}

class InventoryManagementSystem {

private Map<Integer, Product> inventory;

public InventoryManagementSystem() {

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);

}

}

public void deleteProduct(int productId) {

inventory.remove(productId);

}

public void displayInventory() {

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

System.out.println(product);

}

}

}

public class Main {

public static void main(String[] args) {

InventoryManagementSystem sys = new InventoryManagementSystem();

// Adding products

sys.addProduct(new Product(1, "Smart TV", 7, 19999.99));

sys.addProduct(new Product(2, "Video Games", 15, 1699.99));

sys.addProduct(new Product(3, "Camera", 10, 5000.0));

// Displaying inventory

System.out.println("Inventory after adding products:");

sys.displayInventory();

// Updating a product

sys.updateProduct(2, 10, 649.99);

// Displaying inventory after update

System.out.println("Inventory after updating product 2:");

sys.displayInventory();

// Deleting a product

sys.deleteProduct(2);

// Displaying inventory after deletion

System.out.println("Inventory after deleting product 2:");

sys.displayInventory();

}

}**Time Complexity Analysis:**

1. **Add Product:**

* Time Complexity: O(1)
  + Inserting a product into a HashMap is O(1) due to the constant time complexity of hash-based data structures.

1. **Update Product:**
   * Time Complexity: O(1)
   * Updating a product in a HashMap is also O(1) since it involves accessing the element by key and replacing the value.
2. **Delete Product:**
   * Time Complexity: O(1)
   * Removing a product from a HashMap is O(1) as it involves finding the element by key and deleting it.

**Optimization:**

To optimize the inventory management system, we should ensure that the `HashMap` is sized properly to avoid frequent rehashing and adjust the load factor for a balance between performance and memory usage. For concurrent access, using `ConcurrentHashMap` can help prevent synchronization issues and ensure thread-safe operations.

# Exercise 2: E-commerce Platform Search Function

**Scenario:** To work on the search functionality of an e-commerce platform with optimized performance.

**Understand Asymptotic Notation:**

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

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 size of the input data. It also helps in analyzing the efficiency of algorithms by providing an approximation of the worst-case scenario in terms scales as the input size increases. This allows developers to predict performance and make informed decisions about which algorithms to use.

Big O Notation helps in analyzing the Algorithms

* **Algorithm Comparison**: Standardizes efficiency comparison, e.g., O(n log n) vs. O(n^2).
* **Scalability**: Assesses how well an algorithm handles larger inputs.
* **Performance Prediction**: Predicts how an algorithm scales with input size, guiding suitability for large datasets.
* **Worst-Case Analysis**: Ensures the system can handle the algorithm's maximum resource needs.
* **Optimization**: Guides code optimization by highlighting less efficient algorithms.

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

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

1. **Best Case**: This is the best-case scenario where the search operation completes in the shortest possible time, usually when the desired element is at the beginning of the collection.
2. **Average Case**: The scenario of average case, represents a typical run where the position of the desired element is uniformly distributed in the list of elements.
3. **Worst Case**: The scenario of worst case is where the search operation takes the longest time, this is usually considered when the desired element is at the end of the collection or not present in the list at all.

**Setup:**

import java.util.Arrays;

class Item {

    private String id;

    private String name;

    private String category;

    public Item(String id, String name, String category) {

        this.id = id;

        this.name = name;

        this.category = category;

    }

    public String getId() {

        return id;

    }

    public void setId(String id) {

        this.id = id;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public String getCategory() {

        return category;

    }

    public void setCategory(String category) {

        this.category = category;

    }

}

class ECommerceSearchUtil {

    // Linear Search

    public static Item linearSearch(Item[] items, String searchTerm) {

        for (Item item : items) {

            if (item.getName().equalsIgnoreCase(searchTerm)) {

                return item;

            }

        }

        return null; // Item not found

    }

    // Binary Search

    public static Item binarySearch(Item[] items, String searchTerm) {

        int left = 0;

        int right = items.length - 1;

        while (left <= right) {

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

            int cmp = items[mid].getName().compareToIgnoreCase(searchTerm);

            if (cmp == 0) {

                return items[mid]; // Item found

            } else if (cmp < 0) {

                left = mid + 1; // Search in the right half

            } else {

                right = mid - 1; // Search in the left half

            }

        }

        return null; // Item not found

    }

    // Utility method to sort the array before binary search

    public static void sortItemsByName(Item[] items) {

        Arrays.sort(items, (a, b) -> a.getName().compareToIgnoreCase(b.getName()));

    }

}

**Time Complexity:**

**Linear Search:**

* Best Case: O(1) (when the element is at the beginning)
* Average Case: O(n)
* Worst Case: O(n)

**Binary Search:**

* Best Case: O(1) (when the element is at the middle)
* Average Case: O(log n)
* Worst Case: O(log n)

**Suitable Algorithm for this Platform:**

Binary search is more suitable for the e-commerce platform because it has a lower time complexity of O(log n) compared to linear search O(n) for large datasets. However, it requires the dataset to be sorted. If the dataset is not sorted or frequently updated, linear search might be simpler to implement initially but less efficient for larger datasets.

# Exercise 3: Sorting Customer Orders

**Scenario:** To sort customer orders by their total price on an e-commerce platform which helps in prioritizing high-value orders.

**Understand Sorting Algorithms:**

**Bubble Sort**

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

* **Time Complexity**: O(n^2) in the average and worst case, O(n) in the best case
* **Space Complexity**: O(1)
* **Stability**: Stable

**Insertion Sort**

Insertion Sort builds the final sorted array one item at a time. It takes each element from the input and inserts it into the correct position within the already sorted part of the array.

* **Time Complexity**: O(n^2) in the average and worst case, O(n) in the best case
* **Space Complexity**: O(1)
* **Stability**: Stable

**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.

* **Time Complexity**: O(n log n) on average, O(n^2) in the worst case.
* **Space Complexity**: O(log n)
* **Stability**: Not stable

**Merge Sort**

Merge Sort is also a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts them, and then merges the two sorted halves.

* **Time Complexity**: O(n log n)
* **Space Complexity**: O(n)
* **Stability**: Stable

**Setup:**

**Create a class Order**

class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

public String toString() {

return "Order{" +

"orderId=" + orderId +

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

", totalPrice=" + totalPrice +

'}';

}

}

public class BubbleSort {

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].totalPrice > orders[j + 1].totalPrice) {

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

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

}

public class QuickSort {

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

if (low < high) {

int pi = partition(orders, low, high)

// Recursively sort elements before

// partition and after partition

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

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

double pivot = orders[high].totalPrice;

int i = (low - 1); // index of smaller element

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

// If current element is smaller than or

// equal to pivot

if (orders[j].totalPrice <= 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 class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Kumar", 350.0),

new Order(2, "Raju", 950.0),

new Order(3, "Joe", 400.0),

new Order(4, "Pankaj", 600.0),

new Order(5, "Emma", 300.0)

};

// Bubble Sort

System.out.println("Orders before Bubble Sort:");

for (Order order : orders) {

System.out.println(order);

}

BubbleSort.bubbleSort(orders)

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

for (Order order : orders) {

System.out.println(order);

}

// Reset the orders array for Quick Sort

orders = new Order[]{

new Order(1, "Kumar", 350.0),

new Order(2, "Raju", 950.0),

new Order(3, "Joe", 400.0),

new Order(4, "Pankaj", 600.0),

new Order(5, "Emma", 300.0)

};

// Quick Sort

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

for (Order order : orders) {

System.out.println(order);

}

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

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

for (Order order : orders) {

System.out.println(order);

}

}

}

**Analysis**

**Time Complexity Comparison**

* Bubble Sort:
* Best Case: O(n)
* 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)

**Quick Sort is Preferred Over Bubble Sort**

Quick Sort is generally preferred over Bubble Sort because it has a much better average-case time complexity of O(n log n) compared to Bubble Sort's O(n^2).

Even though Quick Sort can degrade to O(n^2) in the worst case, this can be mitigated with good pivot selection strategies, such as choosing the median or using randomization.

Quick Sort also tends to have better cache performance and is more efficient in practice, making it more suitable for sorting large datasets on an e-commerce platform.

# Exercise 4: Employee Management System

**Scenario:** To develop an employee management system for a company and efficiently manage employee records.

**Understand Array Representation:**

**Arrays are Represented in Memory**

Arrays are a fundamental data structure in Java, where elements are stored in contiguous memory locations. This arrangement provides efficient indexing and quick access to elements.

* Contiguous Memory Allocation: Elements are stored in adjacent memory blocks, allowing direct access via an index.
* Fixed Size: The size of an array is defined at the time of its creation and cannot be changed.

**Advantages:**

* Fast Access: O(1) time complexity for accessing elements by index.
* Simplicity: Easy to use and understand, with a straightforward syntax.
* Memory Efficiency: Efficient memory usage due to contiguous allocation.

**Setup:**

**Create a class Employee**

class Employee {

int employeeId;

String name;

String position;

double salary;

public Employee(int employeeId, String name, String position, double salary) {

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

@Override

public String toString() {

return "Employee{" +

"employeeId=" + employeeId +

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

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

", salary=" + salary +

'}';

}

}

public class EmployeeManagementSystem {

private Employee[] employees;

private int count;

public EmployeeManagementSystem(int size) {

employees = new Employee[size];

count = 0;

}

public void addEmployee(Employee employee) {

if (count < employees.length) {

employees[count++] = employee;

System.out.println("Employee added successfully.");

} 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].employeeId == 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) {

int index = -1;

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

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

index = i;

break;

}

}

if (index != -1) {

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

employees[i] = employees[i + 1];

}

employees[--count] = null;

System.out.println("Employee deleted successfully.");

} else {

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

}

}

public static void main(String[] args) {

EmployeeManagementSystem system = new EmployeeManagementSystem(10);

// Add employees

system.addEmployee(new Employee(1, "Ankitha", "Manager", 75000));

system.addEmployee(new Employee(2, "Manoj", "Developer", 60000));

system.addEmployee(new Employee(3, "Charan", "Analyst", 55000));

// Traverse employees

System.out.println("\nEmployees:");

system.traverseEmployees();

// Search for an employee

Employee employee = system.searchEmployee(2);

if (employee != null) {

System.out.println("\nEmployee found: " + employee);

} else {

System.out.println("\nEmployee not found.");

}

// Delete an employee

system.deleteEmployee(2);

// Traverse employees again

System.out.println("\nEmployees after deletion:");

system.traverseEmployees();

}

}

**Analysis:**

**Time Complexity of Operations**

* Add Employee: Time Complexity: O(1)
* Search Employee by ID: Time Complexity: O(n)
* Traverse Employees: Time Complexity: O(n)
* Delete Employee by ID: Time Complexity: O(n)

**Limitations of Arrays and When to Use Them**

* **Fixed Size**: Arrays have a fixed size, making them unsuitable when the number of elements is unknown or changes frequently.
* **Inefficient for Frequent Insertions/Deletions:** Operations like insertion and deletion are costly (O(n)) compared to dynamic data structures like ArrayList or LinkedList.
* **When to Use Arrays:**
* When the number of elements is known and fixed.
* For applications requiring fast access to elements by index.
* When memory efficiency and performance of access are critical.

# Exercise 5: Task Management System

**Scenario:** To develop a task management system where tasks need to be added, deleted, and traversed efficiently.

**Understand Linked Lists**

Linked lists offer better management of dynamic data due to their flexible size and efficient insertions/deletions, despite having a higher time complexity for search operations compared to arrays.

**Types of Linked Lists;**

* **Singly Linked List:**
* **Structure**: Each node contains data and a reference to the next node.
* **Traversal:** Can only traverse in one direction (forward).
* **Advantages:** Simple implementation, uses less memory compared to doubly linked lists.
* **Doubly Linked List:**
* **Structure:** Each node contains data, a reference to the next node, and a reference to the previous node.
* **Traversal:** Can traverse in both directions (forward and backward).
* **Advantages:** Easier to implement certain operations (like deletion) and more flexible traversal.

**Setup**

**Create a class Task**

class Assignment {

private int id;

private String name;

private String status;

public Assignment(int id, String name, String status) {

this.id = id;

this.name = name;

this.status = status;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public String getStatus() {

return status;

}

@Override

public String toString() {

return "Assignment ID: " + id + ", Name: " + name + ", Status: " + status;

}

}

class Node {

Assignment assignment;

Node next;

public Node(Assignment assignment) {

this.assignment = assignment;

this.next = null;

}

}

class AssignmentLinkedList {

private Node head;

public AssignmentLinkedList() {

this.head = null;

}

// Add an assignment to the end of the list

public void addAssignment(Assignment assignment) {

Node newNode = new Node(assignment);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

// Search for an assignment by ID

public Assignment searchAssignmentById(int id) {

Node current = head;

while (current != null) {

if (current.assignment.getId() == id) {

return current.assignment;

}

current = current.next;

}

return null;

}

// Traverse and print all assignments

public void traverseAssignments() {

Node current = head;

while (current != null) {

System.out.println(current.assignment);

current = current.next;

}

}

// Delete an assignment by ID

public boolean deleteAssignmentById(int id) {

if (head == null) return false;

if (head.assignment.getId() == id) {

head = head.next;

return true;

}

Node current = head;

while (current.next != null && current.next.assignment.getId() != id) {

current = current.next;

}

if (current.next == null) return false;

current.next = current.next.next;

return true;

}

}

class TaskManagementSystem {

public static void main(String[] args) {

AssignmentLinkedList assignmentList = new AssignmentLinkedList();

// Add new assignments

assignmentList.addAssignment(new Assignment(1, "Design UI", "In Progress"));

assignmentList.addAssignment(new Assignment(2, "Develop Backend", "Not Started"));

assignmentList.addAssignment(new Assignment(3, "Write Tests", "Not Started"));

assignmentList.addAssignment(new Assignment(4, "Deploy Application", "Completed"));

// Traversing and printing assignments

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

assignmentList.traverseAssignments();

// Searching for an assignment by ID

System.out.println("\nSearching for assignment with ID 3:");

Assignment assignment = assignmentList.searchAssignmentById(3);

if (assignment != null) {

System.out.println("Found: " + assignment);

} else {

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

}

// Deleting an assignment by ID

System.out.println("\nDeleting assignment with ID 2:");

boolean isDeleted = assignmentList.deleteAssignmentById(2);

System.out.println("Deleted: " + isDeleted);

// Traversing and printing assignments after deletion

System.out.println("\nAll assignments after deletion:");

assignmentList.traverseAssignments();

}

}**Analysis:**

**Time Complexity of Operations:**

* + **Add Task:**
  + Time Complexity: O(n)
  + **Search Task by ID:**
    - Time Complexity: O(n)
  + **Traverse Tasks:**
    - Time Complexity: O(n)
  + **Delete Task by ID:**
* Time Complexity: O(n)

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

* Dynamic Size: Linked lists can grow and shrink dynamically, unlike arrays that have a fixed size.
* Efficient Insertions/Deletions: Insertions and deletions can be more efficient (O(1)) if the position is known, as there's no need to shift elements.
* Memory Usage: Linked lists use memory more efficiently for dynamic data as they allocate memory as needed, whereas arrays may allocate more memory than necessary.
* Flexibility: Linked lists provide more flexibility with dynamic data structures, making them more suitable for tasks where the size of the dataset changes frequently.

# Exercise 6: Library Management System

**Scenario:** To develop a library management system where users can search for books by title or author.

**Understand Search Algorithms**

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

**Linear Search:**

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

* Time Complexity: O(n)
* Space Complexity: O(1)
* Best Case: O(1) (if the element is at the beginning)
* Worst Case: O(n) (if the element is at the end or not present)
* Use Case: Suitable for unsorted or small lists.

**Binary Search:**

Binary search is a more efficient search algorithm for sorted lists. It repeatedly divides the search interval in half, comparing the middle element with the target value.

* Time Complexity: O(log n)
* Space Complexity: O(1)
* Best Case: O(1) (if the middle element is the target)
* Worst Case: O(log n) (if the element is not present)
* Use Case: Suitable for large, sorted lists.

Linear search is straightforward and suitable for small or unsorted datasets, while binary search is more efficient for larger, sorted datasets due to its significantly lower time complexity.

**Setup & Implementation:**

**Create a class Book**

import java.util.Arrays;

import java.util.Comparator;

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{" +

"bookId=" + bookId +

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

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

'}';

}

}

public class LibraryManagementSystem {

private Book[] books;

private int count;

public LibraryManagementSystem(int size) {

books = new Book[size];

count = 0;

}

public void addBook(Book book) {

if (count < books.length) {

books[count++] = book;

System.out.println("Book added successfully.");

} else {

System.out.println("Library is full. Cannot add more books.");

}

}

// Linear search to find books by title

public Book linearSearchByTitle(String title) {

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

if (books[i].title.equalsIgnoreCase(title)) {

return books[i];

}

}

return null;

}

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

public Book binarySearchByTitle(String title) {

int left = 0;

int right = count - 1;

while (left <= right) {

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

int comparison = books[mid].title.compareToIgnoreCase(title);

if (comparison == 0) {

return books[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// Method to sort books by title

public void sortBooksByTitle() {

Arrays.sort(books, 0, count, Comparator.comparing(book -> book.title));

}

public static void main(String[] args) {

LibraryManagementSystem system = new LibraryManagementSystem(10);

// Add books

system.addBook(new Book(1, "The Wings of Fire", "Dr. A.P.J. Abdul Kalam"));

system.addBook(new Book(2, "1984", "George Orwell"));

system.addBook(new Book(3, "The god of Small Things", "Arundathi Roy"));

system.addBook(new Book(4, "The White Tiger", "Aravind Adiga"));

system.addBook(new Book(5, "Pride and Prejudice", "Jane Austen"));

// Sort books by title for binary search

system.sortBooksByTitle();

// Linear search for a book by title

String searchTitle = "1984";

Book foundBook = system.linearSearchByTitle(searchTitle);

if (foundBook != null) {

System.out.println("\nLinear search: Book found: " + foundBook);

} else {

System.out.println("\nLinear search: Book not found.");

}

// Binary search for a book by title

foundBook = system.binarySearchByTitle(searchTitle);

if (foundBook != null) {

System.out.println("\nBinary search: Book found: " + foundBook);

} else {

System.out.println("\nBinary search: Book not found.");

}

}

}**Analysis**

**Time Complexity of Search Algorithms**

**Linear Search:**

* Best Case: O(1)
* Average Case: O(n)
* Worst Case: O(n)
* Space Complexity: O(1)

**Binary Search:**

* Best Case: O(1)
* Average Case: O(log n)
* Worst Case: O(log n)
* Space Complexity: O(1)

**When to Use Each Algorithm**

**Linear Search:**

* Use for unsorted or small datasets.
* Simple to implement and does not require sorting.
* Efficient for cases where the dataset size is small or the target element is frequently near the beginning.

**Binary Search:**

* Use for large, sorted datasets.
* Much more efficient for large datasets due to its O(log n) time complexity.
* Requires the list to be sorted, adding an additional step if the data is not already sorted.

# Exercise 7: Financial Forecasting

**Scenario:** To develop a financial forecasting tool that predicts future values based on past data.

**Understand Recursive Algorithms**

**Concept of Recursion**

Recursion is a technique where a function calls itself to solve smaller instances of the same problem. It can simplify complex problems by breaking them down into more manageable subproblems.

* **Base Case**: The condition under which the recursion stops.
* **Recursive Case**: The part of the function where it calls itself with a smaller or simpler input.
* **Advantages**:
  + Simplifies code for problems that have repetitive structures.
  + Often more intuitive for problems like tree traversal, factorial calculation, etc.
* **Disadvantages**:
  + Can lead to excessive memory use due to function call stack.
  + Potential for stack overflow if not properly controlled.

**Setup**

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

import java.util.Scanner;

public class FinancialForecasting {

// Recursive method to calculate future value

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

// Base case: If no years left to predict, return the present value

if (years == 0) {

return presentValue;

}

// Recursive case: Calculate future value

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

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Input present value, growth rate, and number of years

System.out.print("Enter the present value: ");

double presentValue = scanner.nextDouble();

System.out.print("Enter the growth rate (as a decimal, e.g., 0.05 for 5%): ");

double growthRate = scanner.nextDouble();

System.out.print("Enter the number of years: ");

int years = scanner.nextInt();

// Calculate future value

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

// Output the result

System.out.printf("The future value after %d years is: %.2f\n", years, futureValue);

}

}**Analysis**

**Time Complexity of Recursive Algorithm**

* **Time Complexity**: O(n), where n is the number of periods.
* **Space Complexity**: O(n), due to the function call stack. Each recursive call adds a new frame to the stack.

**Optimizing the Recursive Solution**

To avoid excessive computation and potential stack overflow, memoization can be used to store previously computed results. Memoization helps in cases where the same subproblems are solved multiple times, reducing redundant calculations. However, in this simple growth rate model, memoization is not necessary because each computation only depends on the immediate previous step and does not involve overlapping subproblems. Each recursive call in this model is unique to its period and does not repeat previous calculations, so the benefits of memoization are minimal in this specific scenario.