# 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 crucial in managing large inventories for several reasons:

1. **Efficiency**: Proper data structures enable the efficient data storage, retrieval, and manipulation of data. Algorithms ensure that operations like searching, adding, updating, and deleting items are performed optimally.
2. **Scalability**: As inventory size grows, the efficient data structures and algorithms help us maintain performance and prevent slow process progress.
3. **Memory Management**: Optimized data structures help utilize memory effectively, avoiding wastage and ensuring that the system can handle large datasets.
4. **Complexity Management**: They help manage the inherent complexity of large datasets, making it easier to implement and maintain the system.

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

The suitable types of Data Structures for Inventory Management System are as follows:

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

   Here, choosing the data structure HashMap would be more appropriate in case of time complexity and for also insertion, deletion, and updating.

## Setup:

**Creation of product class**

```
public class Product {
    private String productId;
    private String productName;
    private int quantity;
    private double price;
    //constructor
    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 for the product class attributes
    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;
    }
}
```

## Implementation:

**Choosing appropriate Data Structures and implementing methods in the class product:**

```
import java.util.HashMap;


public class InventoryManager {
    private HashMap<String, Product> inventory;
    public InventoryManager() {
        inventory = new HashMap<>();
    }
    //To Add a product to the inventory
    public void addProduct(Product product) {
        inventory.put(product.getProductId(), product);
    }
    //To Update a product in the inventory
    public void updateProduct(Product product) {
        if (inventory.containsKey(product.getProductId())) {
            inventory.put(product.getProductId(), product);
        }
        else {
            System.out.println("Product not found.");
        }
    }
    //To Delete a product from the inventory
```

```
public void deleteProduct(String productId) {
    if (inventory.containsKey(productId)) {
        inventory.remove(productId);
    }
    else {
        System.out.println("Product not found.");
    }
}
//To Display all products in the inventory
public void displayProducts() {
    for (Product product : inventory.values()) {
        System.out.println("Product ID: " + product.getProductId());
        System.out.println("Product Name: " + product.getProductName());
        System.out.println("Quantity: " + product.getQuantity());
        System.out.println("Price: " + product.getPrice());
        System.out.println();
    }
}
public static void main(String[] args) {
    InventoryManager manager = new InventoryManager();
    // Adding products
    Product product1 = new Product("P001", "Product_no_1", 50, 999.99);
    Product product2 = new Product("P002", "Product_no_2", 30, 899.99);
    Product product3 = new Product("P003", "Product_no_3", 100, 349.99);
    Product product4 = new Product("P004", "Product_no_4", 25, 1249.99);
    Product product5 = new Product("P005", "Product_no_5", 15, 2399.99);
    manager.addProduct(product1);
    manager.addProduct(product2);
    manager.addProduct(product3);
    manager.addProduct(product4);
    manager.addProduct(product5);
    // To Display products
    manager.displayProducts();
```

```
        // To Update a product
        Product updatedProduct1 = new Product("P001", "Product_no_1", 60, 949.99);
        manager.updateProduct(updatedProduct1);
        // To Display products after update
        manager.displayProducts();
        // For Deleting a product
        manager.deleteProduct("P002");
        // To Display products after deletion
        manager.displayProducts();
    }
}
```

## 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.
2. **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.
3. **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 above inventory management system, we should ensure that the `HashMap` is sized properly to avoid frequent rehashing, and also adjust the load factor for a balance between performance and memory usage.

For concurrent access, we can also consider using `ConcurrentHashMap` to prevent the chances of occurrence of synchronization issues.

# 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

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

**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:

**Create a product class**

```
public class Product {
    private String productId;
    private String productName;
    private String category;
    //constructor
    public Product(String productId, String productName, String category) {
        this.productId = productId;
        this.productName = productName;
        this.category = category;
    }
   //Setter and Getter methods
    public String getProductId() {
        return productId;
    }
    public String getProductName() {
        return productName;
    }
    public String getCategory() {
        return category;
    }
    public override String toString() {
        return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;
    }
}
```

## Implementation:

```
import java.util.Arrays;
import java.util.Comparator;
public class SearchAlgorithms {
    // Linear Search
```

```
    public static Product linearSearch(Product[] products, String productName) {
        for (Product product : products) {
            if (product.getProductName().equalsIgnoreCase(productName)) {
                return product;
            }
        }
        return null;
    }
    // Binary Search
    public static Product binarySearch(Product[] products, String productName) {
        Arrays.sort(products, Comparator.comparing(Product::getProductName));
        int left = 0;
        int right = products.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int comparison = products[mid].getProductName().compareToIgnoreCase(productName);
            if (comparison == 0) {
                return products[mid];
            } else if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

**Main.java**

```
public class Main {
    public static void main(String[] args) {
        Product[] products = {
```

```
            new Product("P001", "Laptop", "Electronics"),
            new Product("P002", "Smartphone", "Electronics"),
            new Product("P003", "Tablet", "Electronics"),
            new Product("P004", "Monitor", "Electronics"),
            new Product("P005", "Keyboard", "Accessories")
        };
        // Linear search demonstration
        System.out.println("Linear Search:");
        String searchName = "Tablet";
        Product foundProduct = SearchAlgorithms.linearSearch(products, searchName);
        if (foundProduct != null) {
            System.out.println("Found: " + foundProduct);
        } else {
            System.out.println("Product not found.");
        }
        // Binary search demonstration
        System.out.println("\nBinary Search:");
        searchName = "Monitor";
        foundProduct = SearchAlgorithms.binarySearch(products, searchName);
        if (foundProduct != null) {
            System.out.println("Found: " + foundProduct);
        } else {
            System.out.println("Product not found.");
        }
    }
}
```

## 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**

```
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 ID: " + orderId + ", Customer Name: " + customerName + ", Total Price: $" +
totalPrice;
    }
}
```

## Implementation

```
public class SortingAlgorithms {    // Bubble Sort Implementation
    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) {    // Quick Sort Implementation
    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;
        }
    }
    Order temp = orders[i + 1];
    orders[i + 1] = orders[high];
    orders[high] = temp;
```

```
        return i + 1;
    }
}
```

**Main.java**

```
public class Main {
    public static void main(String[] args) {
        Order[] orders = {
            new Order("O001", "Alice", 250.50),
            new Order("O002", "Bob", 150.75),
            new Order("O003", "Charlie", 300.10),
            new Order("O004", "David", 175.20),
            new Order("O005", "Eve", 210.80)
        };
        System.out.println("Before Bubble Sort:");
        printOrders(orders);
        SortingAlgorithms.bubbleSort(orders);         // Bubble Sort demonstration
        System.out.println("\nAfter Bubble Sort:");
        printOrders(orders);
        orders = new Order[]{     // Reset orders
            new Order("O001", "Alice", 250.50),
            new Order("O002", "Bob", 150.75),
            new Order("O003", "Charlie", 300.10),
            new Order("O004", "David", 175.20),
            new Order("O005", "Eve", 210.80)
        };
        System.out.println("\nBefore Quick Sort:");
        printOrders(orders);        // Quick Sort demonstration
        SortingAlgorithms.quickSort(orders, 0, orders.length - 1);
        System.out.println("\nAfter Quick Sort:");
        printOrders(orders);
    }
```

```
    public static void printOrders(Order[] orders) {
        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**

```
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 ID: " + employeeId + ", Name: " + name + ", Position: " + position + ",
Salary: $" + salary;
    }
}
```

## Implementation

### Using an Array to Store Employee Records

```
public class EmployeeManager {
    private Employee[] employees;
    private int size;
    public EmployeeManager(int capacity) {
        employees = new Employee[capacity];
        size = 0;
    }
    // Add an employee to the list
    public void addEmployee(Employee employee) {
        if (size < employees.length) {
            employees[size++] = employee;
        } else {
            System.out.println("Array is full. Cannot add more employees.");
        }
    }
    // Search for an employee by ID
    public Employee searchEmployeeById(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 boolean deleteEmployeeById(int employeeId) {
    for (int i = 0; i < size; i++) {
        if (employees[i].getEmployeeId() == employeeId) {
            for (int j = i; j < size - 1; j++) {
                employees[j] = employees[j + 1];
            }
            employees[--size] = null;
            return true;
        }
    }
    return false;
}
public static void main(String[] args) {
    EmployeeManager manager = new EmployeeManager(10);
    // Adding sample employees
    manager.addEmployee(new Employee(1, "Alice", "Developer", 70000));
    manager.addEmployee(new Employee(2, "Bob", "Manager", 85000));
    manager.addEmployee(new Employee(3, "Charlie", "Analyst", 60000));
    manager.addEmployee(new Employee(4, "David", "Designer", 65000));
    System.out.println("All employees:");
    manager.traverseEmployees(); // Traversing and printing employees
    System.out.println("\nSearching for employee with ID 3:");
    Employee employee = manager.searchEmployeeById(3); // Searching for an employee by ID
    if (employee != null) {
        System.out.println("Found: " + employee);
    } else {
```

```
            System.out.println("Employee not found.");
        }
        System.out.println("\nDeleting employee with ID 2:");
        boolean isDeleted = manager.deleteEmployeeById(2);        // Deleting an employee by ID
        System.out.println("Deleted: " + isDeleted);
        // Traversing and printing employees after deletion
        System.out.println("\nAll employees after deletion:");
        manager.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**

```
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 ID: " + taskId + ", Task Name: " + taskName + ", Status: " + status;
    }
}
```

## Implementation

### Linked List to Manage Tasks

- **Node Class:**

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

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

- **LinkedList Class:**

```
public class TaskLinkedList {
    private Node head;

    public TaskLinkedList() {
        this.head = null;
    }
    // Add a task to the end of the list
    public void addTask(Task task) {
        Node newNode = new Node(task);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
```

```
        }
    }
    // Search for a task by ID
    public Task searchTaskById(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 boolean deleteTaskById(int taskId) {
        if (head == null) return false;
        if (head.task.getTaskId() == taskId) {
            head = head.next;
            return true;
        }
        Node current = head;
        while (current.next != null && current.next.task.getTaskId() != taskId) {
            current = current.next;
        }
        if (current.next == null) return false;
        current.next = current.next.next;
        return true;
    }
}
```

**Main.java**

```
public class Main {
    public static void main(String[] args) {
        TaskLinkedList taskList = new TaskLinkedList();
        //add new tasks
        taskList.addTask(new Task(1, "Design UI", "In Progress"));
        taskList.addTask(new Task(2, "Develop Backend", "Not Started"));
        taskList.addTask(new Task(3, "Write Tests", "Not Started"));
        taskList.addTask(new Task(4, "Deploy Application", "Completed"));
        // Traversing and printing tasks
```

```
        System.out.println("All tasks:");
        taskList.traverseTasks();
        // Searching for a task by ID
        System.out.println("\nSearching for task with ID 3:");
        Task task = taskList.searchTaskById(3);
        if (task != null) {
            System.out.println("Found: " + task);
        } else {
            System.out.println("Task not found.");
        }
        // Deleting a task by ID
        System.out.println("\nDeleting task with ID 2:");
        boolean isDeleted = taskList.deleteTaskById(2);
        System.out.println("Deleted: " + isDeleted);

        // Traversing and printing tasks after deletion
        System.out.println("\nAll tasks after deletion:");
        taskList.traverseTasks();
    }
}
```

## 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**

```
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 override String toString() {
        return "Book ID: " + bookId + ", Title: " + title + ", Author: " + author;
    }
}
```

**//LibraryManager.java**

**//Linear Search to Find Books by Title**

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

public class LibraryManager {
    private Book[] books;
    private int size;
    public LibraryManager(int capacity) {
        books = new Book[capacity];
        size = 0;
    }
    public void addBook(Book book) {    // Add a book to the list
        if (size < books.length) {
            books[size++] = book;
        } else {
            System.out.println("Array is full. Cannot add more books.");
```

```
        }
    }
    // Linear search for books by title
    public Book linearSearchByTitle(String title) {
        for (int i = 0; i < size; i++) {
            if (books[i].getTitle().equalsIgnoreCase(title)) {
                return books[i];
            }
        }
        return null;
    }
}
```

**//Binary Search to Find Books by Title (Assuming the List is Sorted)**

```
public class LibraryManager {
    private Book[] books;
    private int size;
    public LibraryManager(int capacity) {
        books = new Book[capacity];
        size = 0;
    }
    // Add a book to the list
    public void addBook(Book book) {
        if (size < books.length) {
            books[size++] = book;
        } else {
            System.out.println("Array is full. Cannot add more books.");
        }
    }
    // Sort books by title
    public void sortBooksByTitle() {
        Arrays.sort(books, 0, size, Comparator.comparing(Book::getTitle,
String.CASE_INSENSITIVE_ORDER));
```

```
    }
    // Binary search for books by title
    public Book binarySearchByTitle(String title) {
        int low = 0;
        int high = size - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            int comparison = books[mid].getTitle().compareToIgnoreCase(title);
            if (comparison == 0) {
                return books[mid];
            } else if (comparison < 0) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return null;
    }
}
```

## 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**

```
public class FinancialForecasting {
    // Recursive method to calculate future value
    public static double predictFutureValue(double presentValue, double growthRate, int periods) {
        // Base case: if no periods left, return present value
        if (periods == 0) {
            return presentValue;
        }
        // Recursive case: apply growth rate to present value and reduce the period
        return predictFutureValue(presentValue * (1 + growthRate), growthRate, periods - 1);
    }
```

## Implementation

### Recursive Algorithm to Predict Future Values

```
public class FinancialForecasting {
    public static double predictFutureValue(double presentValue, double growthRate, int periods) {
        if (periods == 0) {
            return presentValue;
        }
        return predictFutureValue(presentValue * (1 + growthRate), growthRate, periods - 1);
    }
    public static void main(String[] args) {
        double presentValue = 1000.0;
        double growthRate = 0.05; // 5% growth rate
        int periods = 10;
        double futureValue = predictFutureValue(presentValue, growthRate, periods);
        System.out.println("Predicted Future Value: $" + futureValue);
    }
}
```

## Analysis

### Time Complexity of Recursive Algorithm

- **Time Complexity**: O(n), where n is the number of periods.
    - Each call to the function handles one period, leading to n recursive calls.
- **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, we can use memoization to store previously computed results. However, in this simple growth rate model, memoization is not necessary as each step only depends on the previous step and does not repeat subproblems.