Algorithms_Data Structures

Exercise 1: Inventory Management System

1. Understand the Problem

a. Importance of Data Structures and Algorithms:

Data structures and algorithms are essential for:

- Efficient Searching: Quickly locating a product by ID or name.
- Fast Updates: Quickly updating stock quantities or prices.
- Scalability: Handling thousands of products without performance issues.
- Memory Efficiency: Organizing data compactly for better storage use.

b. Suitable Data Structures:

- ArrayList: Good for maintaining insertion order, but searching is O(n).
- HashMap: Best for quick access using keys (O(1) average time complexity).
    - Key: productId
    - Value: Product object

2.**Implementation**

```
import java.util.HashMap;
import java.util.Scanner;
class Product {
   int productId;
   String productName;
   int quantity;
   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 String toString() {
      return "[" + productId + "] " + productName + " | Qty: " + quantity + " | Price: ₹" + price;
   }
}
```

```
class Inventory {
  HashMap<Integer, Product> products = new HashMap<>();
  public void addProduct(Product p) {
    if (products.containsKey(p.productId)) {
      System.out.println("Product already exists. Use updateProduct to modify.");
    } else {
      products.put(p.productId, p);
      System.out.println("Product added.");
    }
  }
  public void updateProduct(int id, int quantity, double price) {
    if (products.containsKey(id)) {
      Product p = products.get(id);
      p.quantity = quantity;
      p.price = price;
      System.out.println("Product updated.");
    } else {
      System.out.println("Product not found.");
    }
  }
  public void deleteProduct(int id) {
    if (products.remove(id) != null) {
      System.out.println("Product deleted.");
    } else {
      System.out.println("Product not found.");
    }
  }
  public void listProducts() {
    if (products.isEmpty()) {
      System.out.println("Inventory is empty.");
    } else {
      products.values().forEach(System.out::println);
    }
  }
}
public class InventoryManagementSystem {
  public static void main(String[] args) {
    Inventory inventory = new Inventory();
    Scanner scanner = new Scanner(System.in);
//sample
    inventory.addProduct(new Product(101, "Mouse", 50, 299.99));
    inventory.addProduct(new Product(102, "Keyboard", 20, 499.00));
    inventory.updateProduct(101, 45, 279.99);
    inventory.deleteProduct(102);
    inventory.listProducts();
  }
}
```

3.Analysis

Operation	Time Complexity
Add Product	O(1)
Update Product	O(1)
Delete Product	O(1)
List Product	O(n)

Optimization Ideas:

- Use a **TreeMap** if sorted order of products is needed.
- Use **caching** strategies for frequently accessed items.
- If search by productName is needed often, maintain a second HashMap keyed by productName.

Exercise 2: E-commerce Platform Search Function

1.Asymptotic Notation

(a).Big O Notation:

Big O notation describes the **upper bound** of an algorithm's running time based on input size n. It helps in analyzing **scalability** and **performance**.

(b).Best, Average, Worst Cases

Algorithm	Best cases	Average cases	Worst cases
Linear Search	O(1)	O(n/2) ≈ O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

2. Setup

```
class Product {
   int productId;
   String productName;
   String category;
   public Product(int productId, String productName, String category) {
      this.productId = productId;
      this.productName = productName;
      this.category = category;
   }
   public String toString() {
      return "[" + productId + "] " + productName + " - " + category;
   }
}
```

3.Implementation

(a).Linear Search

```
public static Product linearSearch(Product[] products, String name) {
   for (Product p : products) {
      if (p.productName.equalsIgnoreCase(name)) {
         return p;
      }
   }
   return null;
}
```

(b).Binary Search

```
import java.util.Arrays;
import java.util.Comparator;
public static Product binarySearch(Product[] products, String name) {
   Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));
   int low = 0, high = products.length - 1;
   while (low <= high) {
      int mid = (low + high) / 2;
      int comparison = name.compareToIgnoreCase(products[mid].productName);
      if (comparison == 0)
         return products[mid];
      else if (comparison < 0)
         high = mid - 1;
      else
         low = mid + 1;
   }
   return null;
}
```

4.Analysis

Algorithm	Time Complexity
Linear Search	O(n)
Binary Search	O(log n)

(a).Which is More Suitable?

- Linear Search is useful for small datasets or when the data is not sorted.
- Binary Search is much faster for large datasets if the array is sorted by the search key.

For an e-commerce platform, Binary Search is preferred when:

- Data is pre-sorted or stored in structures like TreeMap.
- High-frequency searches require fast response.

Exercise 3: Sorting Customer Orders

1.Sorting Algorithms

Algorithm	Best Cases	Average Cases	Worst Cases	Space
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)

2.Setup

```
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 "[" + orderId + "] " + customerName + " | ₹" + totalPrice;
   }
}
```

3.Implementation

(a).Bubble Sort

```
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) {
        Order temp = orders[j];
        orders[j] = orders[j + 1];
        orders[j + 1] = temp;
      }
    }
  }
}
```

(b). Quick Sort

```
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].totalPrice;
  int i = low - 1;
  for (int j = low; j < high; j++) {
    if (orders[j].totalPrice < pivot) {
      i++;
      Order temp = orders[i];
      orders[i] = orders[j];
      orders[j] = temp;
    }
  }
  Order temp = orders[i + 1];
  orders[i + 1] = orders[high];
  orders[high] = temp;
  return i + 1;
}
```

4.Analysis

Algorithm	Time Complexity
Bubble Sort	$O(n^2)$
Quick Sort	$O(n \log n)$ avg, $O(n^2)$ worst

Why Quick Sort is Preferred:

- Faster average-case performance (O(n log n)).
- More efficient for large datasets.
- Can be optimized with random pivots or tail call elimination.

Bubble Sort is simple but inefficient, making it useful only for educational or very small input sizes.

Exercise 4: Employee Management System

1.Array Representation

How Arrays Are Represented in Memory:

- Arrays are contiguous blocks of memory.
- The index of each element can be calculated as:
  address = base_address + index * element_size
- This allows O(1) access time for reading or writing an element at a known index.

Advantages of Arrays:

- Fast random access using index.
- Easy to traverse and manipulate sequentially.
- Memory-efficient for fixed-size datasets.

2.Setup

```
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;
  }
  public String toString() {
    return "[" + employeeId + "] " + name + " - " + position + " - ₹" + salary;
  }
}
```

3.Implementation

```
public class EmployeeManagementSystem {
   private Employee[] employees;
   private int size;

   public EmployeeManagementSystem(int capacity) {
      employees = new Employee[capacity];
      size = 0;
   }
   public void addEmployee(Employee e) {
      if (size < employees.length) {
         employees[size++] = e;
         System.out.println("Employee added.");
      } else {
         System.out.println("Employee array is full.");
      }
   }
   public Employee searchEmployee(int empId) {
      for (int i = 0; i < size; i++) {
         if (employees[i].employeeId == empId)
            return employees[i];
      }
      return null;
   }
   public void traverseEmployees() {
      if (size == 0) {
         System.out.println("No employees.");
         return;
      }
      for (int i = 0; i < size; i++) {
         System.out.println(employees[i]);
      }
   }
   public void deleteEmployee(int empId) {
      int index = -1;
      for (int i = 0; i < size; i++) {
         if (employees[i].employeeId == empId) {
            index = i;
            break;
         }
      }
      if (index == -1) {
         System.out.println("Employee not found.");
         return;
      }
```

```
        // Shift elements left
        for (int i = index; i < size - 1; i++) {
            employees[i] = employees[i + 1];
        }
        employees[--size] = null;
        System.out.println("Employee deleted.");
    }
}
```

4.Analysis

Operation	Time Complexity
Add	O(1)
Search	O(n)
Traverse	O(n)
Delete	O(n)

Limitations of Arrays

- **Fixed Size**: Cannot grow dynamically — leads to wasted space or overflow.
- **Insertion/Deletion**: Costly as elements may need shifting.
- **No Index-based Sorting**: Requires external logic.

When to Use Arrays:

- When the number of employees is **known/fixed**.
- When **fast access** by index is required.
- For **simple, static lists**.

Exercise 5: Task Management System

1.Linked Lists

Type	Description
Singly Linked List	Each node has a value and a pointer to the **next** node only.
Doubly Linked List	Each node has a pointer to the **next** and **previous** node. Allows backward traversal.

2.Setup

```
class Task {
  int taskId;
  String taskName;
  String status; // e.g., "Pending", "Completed"
  public Task(int taskId, String taskName, String status) {
    this.taskId = taskId;
    this.taskName = taskName;
    this.status = status;
  }
  public String toString() {
    return "[" + taskId + "] " + taskName + " - " + status;
  }
}
```

3.Implementation

```
class Node {
  Task task;
  Node next;
  public Node(Task task) {
    this.task = task;
    this.next = null;
  }
}
class TaskLinkedList {
  private Node head;
  // Add task at the end
  public void addTask(Task task) {
    Node newNode = new Node(task);
    if (head == null) {
      head = newNode;
    } else {
      Node curr = head;
      while (curr.next != null) {
        curr = curr.next;
      }
      curr.next = newNode;
    }
    System.out.println("Task added.");
  }
  // Search task by ID
  public Task searchTask(int taskId) {
    Node curr = head;
    while (curr != null) {
      if (curr.task.taskId == taskId)
```

```java
            return curr.task;
        curr = curr.next;
      }
      return null;
   }
   // Traverse all tasks
   public void traverseTasks() {
      if (head == null) {
         System.out.println("No tasks available.");
         return;
      }
      Node curr = head;
      while (curr != null) {
         System.out.println(curr.task);
         curr = curr.next;
      }
   }
   // Delete task by ID
   public void deleteTask(int taskId) {
      if (head == null) {
         System.out.println("Task list is empty.");
         return;
      }
      if (head.task.taskId == taskId) {
         head = head.next;
         System.out.println("Task deleted.");
         return;
      }
      Node curr = head;
      while (curr.next != null && curr.next.task.taskId != taskId) {
         curr = curr.next;
      }
      if (curr.next == null) {
         System.out.println("Task not found.");
      } else {
         curr.next = curr.next.next;
         System.out.println("Task deleted.");
      }
   }
}
```

4.Analysis

Operation	Time Complexity
Add	O(n)
Search	O(n)
Delete	O(n)
Traverse	O(n)

Advantages of Linked Lists over Arrays

1. **Dynamic Size**:
    - Linked lists can grow or shrink at runtime without reallocation.
2. **Efficient Insertions/Deletions**:
    - Insertion or deletion is faster (O(1)) when position is known.
3. **No Need for Contiguous Memory**:
    - Nodes can be stored anywhere in memory, not necessarily in one block.
4. **No Predefined Size Needed**:
    - You don't have to know the number of elements in advance.
5. **No Wasted Memory**:
    - Memory is allocated only when needed — no over-allocation.
6. **Better for Frequent Add/Remove Operations**:
    - Ideal when the number of elements changes frequently.
7. **Easier to Implement Certain Data Structures**:
    - Useful for implementing stacks, queues, and graphs.

Exercise 6: Library Management System

1.Search Algorithms

◆ **Linear Search:**

- Checks each element in the list one by one.
- No need for sorting.
- Time Complexity:
    - Best: O(1) (match at start)
    - Worst: O(n) (match at end or not found)

◆ **Binary Search:**

- Divides the list in half repeatedly to search.
- Requires the list to be sorted by the key (e.g., title).
- Time Complexity:
    - Best: O(1) (match in middle)
    - Worst: O(log n)

2. 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;
   }
   public String toString() {
      return "[" + bookId + "] " + title + " by " + author;
   }
}
```

3.Implementation

- **Linear Search by Title**

```
    public static Book linearSearch(Book[] books, String title) {
   for (Book book : books) {
      if (book.title.equalsIgnoreCase(title)) {
         return book;
      }
   }
   return null;
}
```

- **Binary Search by Title**

```
import java.util.Arrays;
import java.util.Comparator;
public static Book binarySearch(Book[] books, String title) {
   Arrays.sort(books, Comparator.comparing(b -> b.title.toLowerCase())); // Ensure
sorted list
   int low = 0, high = books.length - 1;
   while (low <= high) {
      int mid = (low + high) / 2;
      int compare = title.compareToIgnoreCase(books[mid].title);

      if (compare == 0) return books[mid];
      else if (compare < 0) high = mid - 1;
      else low = mid + 1;
   }
   return null;
}
```

4.Analysis

Searchs	Time Complexity
Linear Search	O(n)
Binary Search	O(log n)

Exercise 7: Financial Forecasting

1.Recursive Algorithms

- **What is Recursion?**

Recursion is when a method calls itself to solve a smaller instance of the problem until it reaches a base case.

- **Why Use Recursion?**
  - Simplifies problems that have repetitive or self-similar structure.
  - Ideal for problems like factorial, Fibonacci series, tree traversals, and financial projections.

2. **Setup**

Future Value = Present Value × (1 + growthRate)^n

Where:

- growthRate is a decimal (e.g., 5% = 0.05)
- n is the number of years
- We'll implement this using **recursion**.

3.Implementation

```
public class FinancialForecasting {
  public static double futureValue(double presentValue, double growthRate, int years) {
    if (years == 0) {
      return presentValue;  // Base case
    }
    return (1 + growthRate) * futureValue(presentValue, growthRate, years - 1);
  }
  public static void main(String[] args) {
    double presentValue = 10000;     // ₹10,000
    double growthRate = 0.08;       // 8% annual growth
    int years = 5;
```

```java
        double result = futureValue(presentValue, growthRate, years);
        System.out.printf("Future Value after %d years: ₹%.2f\n", years, result);
    }
}
```

Output:

Future Value after 5 years: ₹14693.28

4. **Analysis**

- **Time Complexity:**
  - Each recursive call reduces years by 1.
  - So there are **n calls** ⇒ **Time Complexity: O(n)**
- **Space Complexity:**
  - Each call adds a frame to the stack.
  - So **Space Complexity: O(n)** due to recursion stack.

Optimization Techniques

1. Use Memoization:
If values are reused (e.g., overlapping subproblems), memoize results to avoid repeated computation.
2. Use Iteration Instead (Tail Recursion or Loop):

```java
public static double futureValueIterative(double presentValue, double growthRate, int years) {
    double result = presentValue;
    for (int i = 0; i < years; i++) {
        result *= (1 + growthRate);
    }
    return result;
}
```

- **Time Complexity**: O(n)
- **Space Complexity**: O(1)