**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

**Code:**

import java.util.HashMap;

import java.util.Map;

public class InventorySystem {

static class Product {

int id;

String name;

int quantity;

double price;

Product(int id, String name, int qty, double price) {

this.id = id;

this.name = name;

this.quantity = qty;

this.price = price;

}

public String toString() {

return id + " | " + name + " | Qty: " + quantity + " | ₹" + price;

}

}

static class Inventory {

Map<Integer, Product> products = new HashMap<>();

void add(Product p) {

products.put(p.id, p);

System.out.println("Added: " + p.name);

}

void update(int id, int qty, double price) {

Product p = products.get(id);

if (p != null) {

p.quantity = qty;

p.price = price;

System.out.println("Updated: " + p.name);

}

}

void delete(int id) {

products.remove(id);

System.out.println("Deleted product ID: " + id);

}

void viewAll() {

for (Product p : products.values()) {

System.out.println(p);

}

}

}

public static void main(String[] args) {

Inventory inv = new Inventory();

inv.add(new Product(1, "Laptop", 5, 75000));

inv.add(new Product(2, "Mouse", 20, 500));

inv.viewAll();

inv.update(1, 3, 72000);

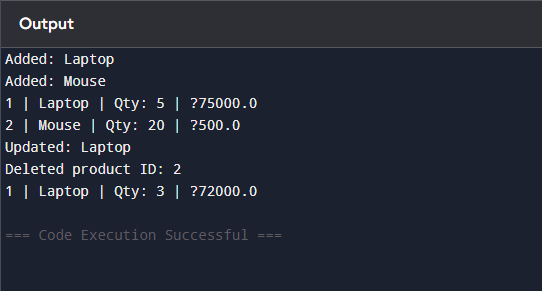
inv.delete(2);

inv.viewAll();

}

}

**Output Screenshot:**



**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:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

Code:

import java.util.Arrays;

public class SearchExample {

static class Product {

int id;

String name;

Product(int id, String name) {

this.id = id;

this.name = name;

}

}

static Product linearSearch(Product[] arr, String name) {

for (Product p : arr)

if (p.name.equalsIgnoreCase(name))

return p;

return null;

}

static Product binarySearch(Product[] arr, String name) {

int left = 0, right = arr.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = name.compareToIgnoreCase(arr[mid].name);

if (cmp == 0) return arr[mid];

else if (cmp < 0) right = mid - 1;

else left = mid + 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Phone"),

new Product(102, "Tablet"),

new Product(103, "Laptop")

};

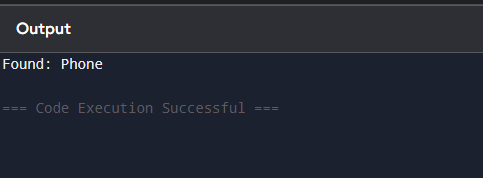
Arrays.sort(products, (a, b) -> a.name.compareToIgnoreCase(b.name));

Product found = binarySearch(products, "Phone");

if (found != null) System.out.println("Found: " + found.name);

}

Output Screenshot:

****

**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 Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**Code:**

public class OrderSorting {

static class Order {

int id;

String customer;

double total;

Order(int id, String customer, double total) {

this.id = id;

this.customer = customer;

this.total = total;

}

public String toString() {

return id + " | " + customer + " | ₹" + total;

}

}

static void bubbleSort(Order[] orders) {

for (int i = 0; i < orders.length - 1; i++) {

for (int j = 0; j < orders.length - 1 - i; j++) {

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

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Arjun", 2000),

new Order(2, "Bhavya", 800),

new Order(3, "Charan", 3000)

};

bubbleSort(orders);

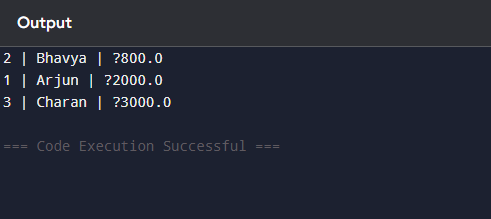
for (Order o : orders)

System.out.println(o);

}

}

**Output Screenshot:**

****

**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:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

Code:

public class EmployeeManager {

static class Employee {

int id;

String name;

String role;

Employee(int id, String name, String role) {

this.id = id;

this.name = name;

this.role = role;

}

public String toString() {

return id + " | " + name + " | " + role;

}

}

public static void main(String[] args) {

Employee[] employees = new Employee[5];

int count = 0;

employees[count++] = new Employee(1, "Aarav", "Developer");

employees[count++] = new Employee(2, "Riya", "Manager");

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

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

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

}

// Search

int searchId = 2;

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

if (employees[i].id == searchId)

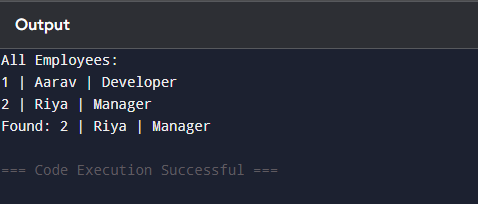
System.out.println("Found: " + employees[i]);

}

}

}

Output Screenshot:

****

**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:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

Code:

public class TaskManagement {

static class Task {

int taskId;

String taskName;

String status;

Task next; // points to next Task in the list

Task(int id, String name, String status) {

this.taskId = id;

this.taskName = name;

this.status = status;

}

}

static class TaskList {

Task head = null;

void add(Task newTask) {

if (head == null) head = newTask;

else {

Task current = head;

while (current.next != null) current = current.next;

current.next = newTask;

}

}

void search(int id) {

Task current = head;

while (current != null) {

if (current.taskId == id) {

System.out.println("Found: " + current.taskName);

return;

}

current = current.next;

}

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

}

void delete(int id) {

if (head == null) return;

if (head.taskId == id) {

head = head.next;

return;

}

Task current = head;

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

current = current.next;

}

if (current.next != null)

current.next = current.next.next;

}

void showAll() {

Task current = head;

while (current != null) {

System.out.println(current.taskId + " | " + current.taskName + " | " + current.status);

current = current.next;

}

}

}

public static void main(String[] args) {

TaskList tasks = new TaskList();

tasks.add(new Task(1, "Design UI", "Pending"));

tasks.add(new Task(2, "Write Code", "In Progress"));

tasks.showAll();

tasks.search(2);

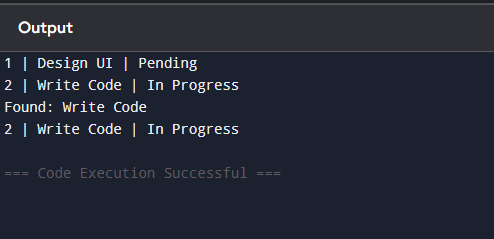
tasks.delete(1);

tasks.showAll();

}

}

Output Screenshot:



**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:**
   * Explain linear search and binary search algorithms.
2. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

Code:

public class LibrarySearch {

static class Book {

int bookId;

String title;

String author;

Book(int id, String title, String author) {

this.bookId = id;

this.title = title;

this.author = author;

}

}

static Book linearSearch(Book[] books, String title) {

for (Book b : books)

if (b.title.equalsIgnoreCase(title)) return b;

return null;

}

static Book binarySearch(Book[] books, String title) {

int left = 0, right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) return books[mid];

if (cmp < 0) right = mid - 1;

else left = mid + 1;

}

return null;

}

public static void main(String[] args) {

Book[] books = {

new Book(1, "Atomic Habits", "James Clear"),

new Book(2, "Clean Code", "Robert C. Martin"),

new Book(3, "Deep Work", "Cal Newport")

};

java.util.Arrays.sort(books, (a, b) -> a.title.compareToIgnoreCase(b.title));

Book b1 = linearSearch(books, "Clean Code");

Book b2 = binarySearch(books, "Deep Work");

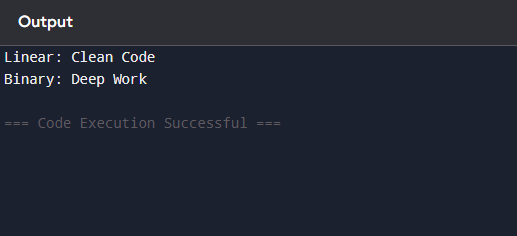
System.out.println("Linear: " + (b1 != null ? b1.title : "Not found"));

System.out.println("Binary: " + (b2 != null ? b2.title : "Not found"));

}

}

Output Screenshot:



**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:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Code:**

public class FinancialForecast {

// Recursive method: futureValue = amount \* (1 + rate)^years

static double predictValue(double amount, double rate, int years) {

if (years == 0) return amount;

return predictValue(amount \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

double startingAmount = 10000;

double growthRate = 0.10; // 10% annual growth

int forecastYears = 5;

double futureValue = predictValue(startingAmount, growthRate, forecastYears);

System.out.printf("Estimated value after %d years: ₹%.2f%n", forecastYears, futureValue);

}

}

**Output Screenshot:**

