ALGORITHM SOLUTION

**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:-

package week1.algorithm.q1;

import java.util.HashMap;

class Product {

private String productId;

private String productName;

private int quantity;

private double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String getProductId() {

return productId;

}

public void setProductName(String productName) {

this.productName = productName;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public void setPrice(double price) {

this.price = price;

}

public String toString() {

return productId + " | " + productName + " | " + quantity + " | " + price;

}

}

class Inventory {

private HashMap<String, Product> products;

public Inventory() {

products = new HashMap<>();

}

public void addProduct(Product product) {

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

}

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

if (products.containsKey(productId)) {

Product p = products.get(productId);

p.setProductName(name);

p.setQuantity(quantity);

p.setPrice(price);

}

}

public void deleteProduct(String productId) {

products.remove(productId);

}

public void displayInventory() {

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

System.out.println(p);

}

}

}

public class Main {

public static void main(String[] args) {

Inventory inventory = new Inventory();

Product p1 = new Product("P101", "Laptop", 10, 55000.0);

Product p2 = new Product("P102", "Keyboard", 50, 700.0);

Product p3 = new Product("P103", "Mouse", 100, 400.0);

inventory.addProduct(p1);

inventory.addProduct(p2);

inventory.addProduct(p3);

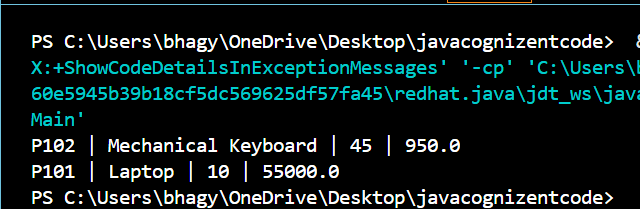
inventory.updateProduct("P102", "Mechanical Keyboard", 45, 950.0);

inventory.deleteProduct("P103");

inventory.displayInventory();

}

}



**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;

import java.util.Comparator;

class Product {

String productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String toString() {

return productId + " | " + productName + " | " + category;

}

}

class SearchEngine {

public static Product linearSearch(Product[] products, String targetName) {

for (Product p : products) {

if (p.productName.equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

public static Product binarySearch(Product[] products, String targetName) {

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

while (left <= right) {

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

int cmp = products[mid].productName.compareToIgnoreCase(targetName);

if (cmp == 0)

return products[mid];

else if (cmp < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product("P101", "Laptop", "Electronics"),

new Product("P102", "Mouse", "Accessories"),

new Product("P103", "Keyboard", "Accessories"),

new Product("P104", "Monitor", "Electronics"),

new Product("P105", "Headphones", "Audio")

};

System.out.println("Linear Search Result:");

Product result1 = SearchEngine.linearSearch(products, "Monitor");

System.out.println(result1 != null ? result1 : "Product Not Found");

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

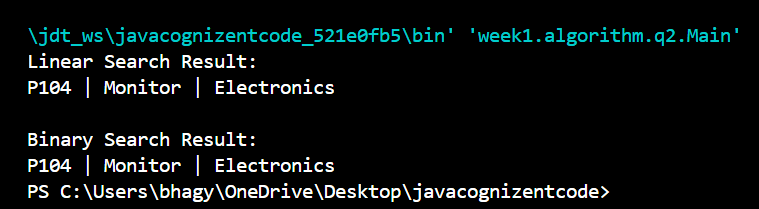
System.out.println("\nBinary Search Result:");

Product result2 = SearchEngine.binarySearch(products, "Monitor");

System.out.println(result2 != null ? result2 : "Product Not Found");

}

}



**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:-

class Order {

String orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String toString() {

return orderId + " | " + customerName + " | ₹" + totalPrice;

}

}

class Sorter {

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;

}

}

}

}

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;

}

public static void printOrders(Order[] orders) {

for (Order o : orders) {

System.out.println(o);

}

}

}

public class SortCustomerOrders {

public static void main(String[] args) {

Order[] orders = {

new Order("O101", "Ravi", 1500.50),

new Order("O102", "Anita", 2599.00),

new Order("O103", "Suman", 799.99),

new Order("O104", "Rahul", 1250.75),

new Order("O105", "Neha", 5999.90)

};

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

Sorter.printOrders(orders);

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

Sorter.bubbleSort(orders);

Sorter.printOrders(orders);

orders = new Order[] {

new Order("O101", "Ravi", 1500.50),

new Order("O102", "Anita", 2599.00),

new Order("O103", "Suman", 799.99),

new Order("O104", "Rahul", 1250.75),

new Order("O105", "Neha", 5999.90)

};

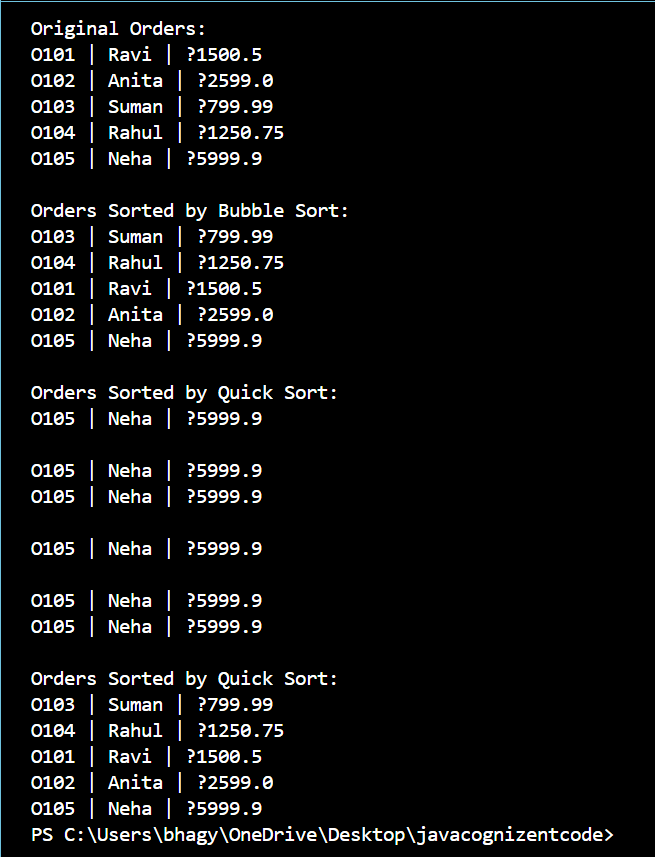
System.out.println("\nOrders Sorted by Quick Sort:");

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

Sorter.printOrders(orders);

}

}



**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:-

class Employee {

String employeeId;

String name;

String position;

double salary;

public Employee(String 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;

}

}

class EmployeeSystem {

private Employee[] employees;

private int size;

public EmployeeSystem(int capacity) {

employees = new Employee[capacity];

size = 0;

}

public void addEmployee(Employee emp) {

if (size < employees.length) {

employees[size++] = emp;

}

}

public Employee searchEmployee(String empId) {

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

if (employees[i].employeeId.equals(empId)) {

return employees[i];

}

}

return null;

}

public void deleteEmployee(String empId) {

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

if (employees[i].employeeId.equals(empId)) {

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

employees[j] = employees[j + 1];

}

employees[--size] = null;

return;

}

}

}

public void listEmployees() {

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

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

}

}

}

public class EmployeeManagementSystem {

public static void main(String[] args) {

EmployeeSystem system = new EmployeeSystem(10);

system.addEmployee(new Employee("E101", "Riya", "Manager", 75000));

system.addEmployee(new Employee("E102", "Arjun", "Developer", 50000));

system.addEmployee(new Employee("E103", "Sneha", "Tester", 40000));

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

system.listEmployees();

System.out.println("\nSearching for E102:");

Employee e = system.searchEmployee("E102");

System.out.println(e != null ? e : "Not Found");

System.out.println("\nDeleting E102...");

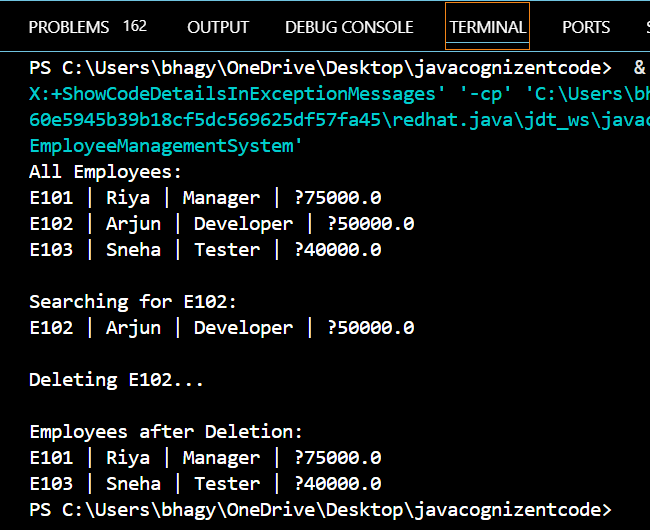
system.deleteEmployee("E102");

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

system.listEmployees();

}

}



**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:-

class Task {

String taskId;

String taskName;

String status;

Task next;

public Task(String taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

this.next = null;

}

public String toString() {

return taskId + " | " + taskName + " | " + status;

}

}

class TaskList {

private Task head;

public void addTask(String taskId, String taskName, String status) {

Task newTask = new Task(taskId, taskName, status);

if (head == null) {

head = newTask;

} else {

Task temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newTask;

}

}

public Task searchTask(String taskId) {

Task temp = head;

while (temp != null) {

if (temp.taskId.equals(taskId)) {

return temp;

}

temp = temp.next;

}

return null;

}

public void deleteTask(String taskId) {

if (head == null) return;

if (head.taskId.equals(taskId)) {

head = head.next;

return;

}

Task prev = null, curr = head;

while (curr != null && !curr.taskId.equals(taskId)) {

prev = curr;

curr = curr.next;

}

if (curr != null) {

prev.next = curr.next;

}

}

public void displayTasks() {

Task temp = head;

while (temp != null) {

System.out.println(temp);

temp = temp.next;

}

}

}

public class TaskManagementSystem {

public static void main(String[] args) {

TaskList taskList = new TaskList();

taskList.addTask("T101", "Fix Bug", "Pending");

taskList.addTask("T102", "Write Tests", "In Progress");

taskList.addTask("T103", "Code Review", "Pending");

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

taskList.displayTasks();

System.out.println("\nSearching for T102:");

Task result = taskList.searchTask("T102");

System.out.println(result != null ? result : "Task Not Found");

System.out.println("\nDeleting T102...");

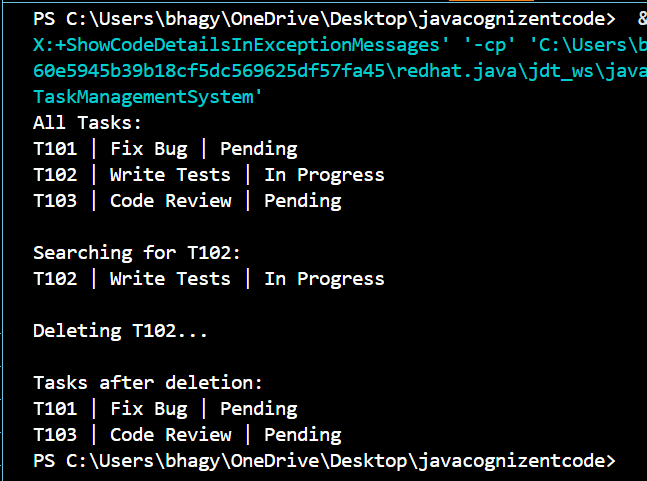
taskList.deleteTask("T102");

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

taskList.displayTasks();

}

}



**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:-

import java.util.Arrays;

import java.util.Comparator;

class Book {

String bookId;

String title;

String author;

public Book(String bookId, String title, String author) {

this.bookId = bookId;

this.title = title;

this.author = author;

}

public String toString() {

return bookId + " | " + title + " | " + author;

}

}

class Library {

public static Book linearSearch(Book[] books, String targetTitle) {

for (Book b : books) {

if (b.title.equalsIgnoreCase(targetTitle)) {

return b;

}

}

return null;

}

public static Book binarySearch(Book[] books, String targetTitle) {

int low = 0, high = books.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

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

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

else if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void printBooks(Book[] books) {

for (Book b : books) {

System.out.println(b);

}

}

}

public class LibraryManagementSystem {

public static void main(String[] args) {

Book[] books = {

new Book("B101", "The Alchemist", "Paulo Coelho"),

new Book("B102", "Wings of Fire", "A.P.J. Abdul Kalam"),

new Book("B103", "The Guide", "R. K. Narayan"),

new Book("B104", "Ignited Minds", "A.P.J. Abdul Kalam"),

new Book("B105", "Half Girlfriend", "Chetan Bhagat")

};

System.out.println("Linear Search:");

Book result1 = Library.linearSearch(books, "The Guide");

System.out.println(result1 != null ? result1 : "Book Not Found");

Arrays.sort(books, Comparator.comparing(b -> b.title.toLowerCase()));

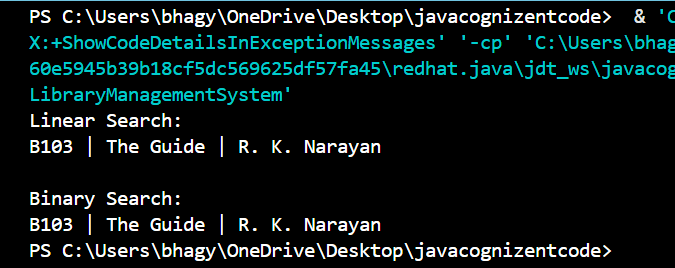
System.out.println("\nBinary Search:");

Book result2 = Library.binarySearch(books, "The Guide");

System.out.println(result2 != null ? result2 : "Book Not Found");

}

}



**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.

class Forecast {

public static double predictValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

return predictValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

}

public class FinancialForecasting {

public static void main(String[] args) {

double currentValue = 10000;

double annualGrowthRate = 0.08;

int years = 5;

double futureValue = Forecast.predictValue(currentValue, annualGrowthRate, years);

System.out.println("Future Value after " + years + " years: ₹" + futureValue);

}

}

