Data structures and Algorithms

Mandatory Hands-On

**Exercise 2: E-commerce Platform Search Function**

**Step 1: Understanding Asymptotic Notation**

**Big O Notation is used to describe the performance or complexity of an algorithm as the input size grows. It helps in analyzing and selecting the most efficient algorithm.**

| **Scenario** | **Meaning** |
| --- | --- |
| **Best Case** | **Minimum time the algorithm takes for input** |
| **Average Case** | **Expected time across random input sets** |
| **Worst Case** | **Maximum time taken for the most complex input** |

**Step 2: Project Setup**

* **Java Project Name: EcommerceSearchExample**
* **Package Structure: com.ecommerce.search**
* **Files Created:**
  + **Product.java**
  + **SearchEngine.java**
  + **SearchTest.java**

**Step 3: Product Class**

**Product.java**

**package com.ecommerce.search;**

**public class Product {**

**private int productId;**

**private String productName;**

**private String category;**

**public Product(int id, String name, String category) {**

**this.productId = id;**

**this.productName = name;**

**this.category = category;**

**}**

**public int getProductId() {**

**return productId;**

**}**

**public String getProductName() {**

**return productName;**

**}**

**public String getCategory() {**

**return category;**

**}**

**}**

**Step 4: Implementing Search Algorithms**

**SearchEngine.java**

**package com.ecommerce.search;**

**import java.util.Arrays;**

**import java.util.Comparator;**

**public class SearchEngine {**

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

**for (Product product : products) {**

**if (product.getProductName().equalsIgnoreCase(name)) {**

**return product;**

**}**

**}**

**return null;**

**}**

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

**Arrays.sort(products, Comparator.comparing(Product::getProductName));**

**int low = 0, high = products.length - 1;**

**while (low <= high) {**

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

**int result = name.compareToIgnoreCase(products[mid].getProductName());**

**if (result == 0) return products[mid];**

**else if (result < 0) high = mid - 1;**

**else low = mid + 1;**

**}**

**return null;**

**}**

**}**

**Step 5: Testing the Search**

**SearchTest.java**

**package com.ecommerce.search;**

**public class SearchTest {**

**public static void main(String[] args) {**

**Product[] products = {**

**new Product(101, "Laptop", "Electronics"),**

**new Product(102, "Shoes", "Fashion"),**

**new Product(103, "Watch", "Accessories"),**

**new Product(104, "Phone", "Electronics"),**

**new Product(105, "Bag", "Fashion")**

**};**

**String searchKey = "Phone";**

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

**Product result1 = SearchEngine.linearSearch(products, searchKey);**

**display(result1);**

**System.out.println("Binary Search Result:");**

**Product result2 = SearchEngine.binarySearch(products, searchKey);**

**display(result2);**

**}**

**private static void display(Product p) {**

**if (p != null) {**

**System.out.println("Found: " + p.getProductName() + " [" + p.getCategory() + "]");**

**} else {**

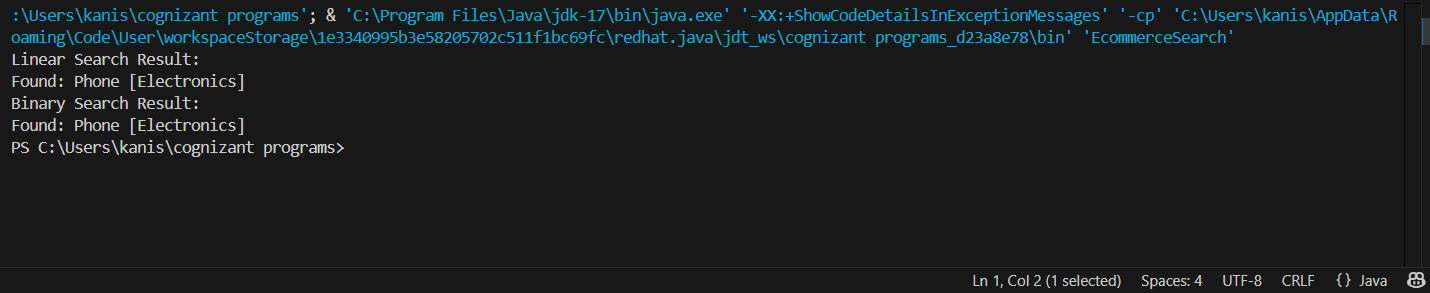
**System.out.println("Product not found.");**

**}**

**}**

**}**

**Step 6: Output**

****

**Step 7: Complexity Analysis**

| **Algorithm** | **Time Complexity** | **Best Case** | **Worst Case** | **Sorted Required?** |
| --- | --- | --- | --- | --- |
| **Linear Search** | **O(n)** | **O(1)** | **O(n)** | **No** |
| **Binary Search** | **O(log n)** | **O(1)** | **O(log n)** | **Yes** |

Exercise 7: Financial Forecasting

**Step 1: Understanding Recursive Algorithms**

**Recursion** is a technique where a method calls itself to solve a smaller part of the original problem. It simplifies problems that follow a **repeatable pattern** or can be broken into **sub-problems** of the same type.

In financial forecasting, we can use recursion to predict future values based on a fixed growth rate and a base value.

**Step 2: Setup – Forecasting Problem**

Let’s assume:

* **Base amount** = Present value
* **Growth rate** = % increase each year
* **Years (n)** = Number of future years to predict

We’ll create a recursive method:

Copy code

futureValue(years) = futureValue(years - 1) \* (1 + growthRate)

**Step 3: Java Implementation**

**FinancialForecast.java**

public class FinancialForecast {

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

if (years == 0) {

return currentValue;

}

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

}

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

double[] dp = new double[years + 1];

dp[0] = currentValue;

for (int i = 1; i <= years; i++) {

dp[i] = dp[i - 1] \* (1 + growthRate);

}

return dp[years];

}

}

**ForecastTest.java**

public class ForecastTest {

public static void main(String[] args) {

double currentValue = 10000.0;

double growthRate = 0.08; // 8% annual growth

int years = 5;

double resultRecursive = FinancialForecast.predictValueRecursive(currentValue, growthRate, years);

System.out.printf("Recursive Forecast (after %d years): ₹%.2f%n", years, resultRecursive);

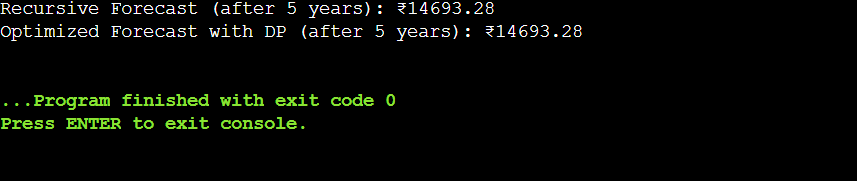
double resultDP = FinancialForecast.predictValueDP(currentValue, growthRate, years);

System.out.printf("Optimized Forecast with DP (after %d years): ₹%.2f%n", years, resultDP);

}

}

**Step 4: Output**

****

**Step 5: Analysis**

| **Version** | **Time Complexity** | **Space Complexity** | **Description** |
| --- | --- | --- | --- |
| Recursive | O(n) | O(n) (stack calls) | Elegant but uses stack memory |
| Optimized (DP) | O(n) | O(n) | Uses loop and array for storage |

**Step 6: About Optimization**

* **Recursive solutions** can lead to **stack overflow** or excessive computation for large inputs.
* **Dynamic Programming (DP)** avoids recomputation by storing past results (also called **memoization** if done recursively).
* For this problem, **iterative DP** is memory-safe and efficient.

Additional Hand-On

**Exercise 3: Sorting Customer Orders**

**Step 1: Understanding Sorting Algorithms**

Sorting is a fundamental operation in software applications. In this task, we focus on sorting customer orders by total price to prioritize high-value transactions.

| **Algorithm** | **Description** |
| --- | --- |
| **Bubble Sort** | Compares adjacent elements and swaps if out of order. Easy to implement but slow. |
| **Insertion Sort** | Builds the final sorted array one element at a time. Good for small datasets. |
| **Quick Sort** | Uses a divide-and-conquer strategy to sort faster than most simple sorts. |
| **Merge Sort** | Also divide-and-conquer, but uses more memory. Stable and consistent. |

**Step 2: Java Implementation**

**Order.java**

public class Order {

private int orderId;

private String customerName;

private double totalPrice;

public Order(int id, String name, double price) {

this.orderId = id;

this.customerName = name;

this.totalPrice = price;

}

public double getTotalPrice() {

return totalPrice;

}

public String toString() {

return "OrderID: " + orderId + ", Name: " + customerName + ", Total: ₹" + totalPrice;

}

}

**Sorter.java**

public 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].getTotalPrice() > orders[j + 1].getTotalPrice()) {

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].getTotalPrice();

int i = low - 1;

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

if (orders[j].getTotalPrice() < 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;

}

}

**OrderTest.java**

public class OrderTest {

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Arun", 5500.00),

new Order(2, "Divya", 1200.50),

new Order(3, "Meera", 9999.99),

new Order(4, "Ravi", 4200.75),

new Order(5, "Kiran", 3150.00)

};

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

Sorter.bubbleSort(orders);

for (Order o : orders) {

System.out.println(o);

}

Order[] orders2 = {

new Order(1, "Arun", 5500.00),

new Order(2, "Divya", 1200.50),

new Order(3, "Meera", 9999.99),

new Order(4, "Ravi", 4200.75),

new Order(5, "Kiran", 3150.00)

};

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

Sorter.quickSort(orders2, 0, orders2.length - 1);

for (Order o : orders2) {

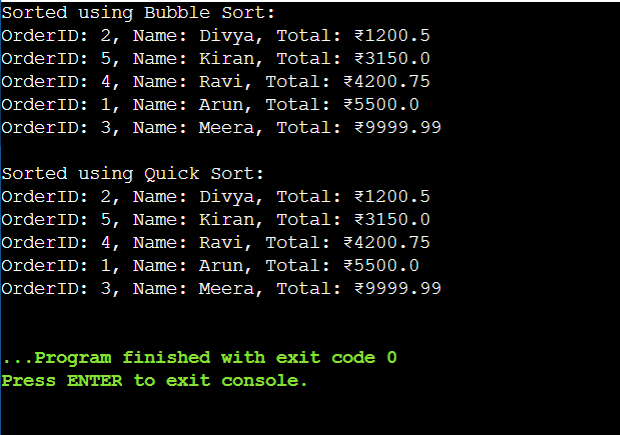
System.out.println(o);

}

}

}

**Step 3: Sample Output**

****

**Step 4: Time Complexity Analysis**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) |

**Step 5: Why Quick Sort is Preferred**

* **Bubble Sort** is simple but inefficient for large datasets.
* **Quick Sort** is faster due to divide-and-conquer strategy.
* Though worst-case is O(n²), with good pivot choice, it performs very efficiently.

**Exercise 5: Task Management System**

**Step 1: Understanding Linked Lists**

A **linked list** is a linear data structure where elements (nodes) point to the next node in the sequence.

**Types of Linked Lists:**

| **Type** | **Description** |
| --- | --- |
| **Singly Linked List** | Each node points to the next; traversal is one-way. |
| **Doubly Linked List** | Each node points to both previous and next nodes; allows bidirectional traversal. |

**Why Linked Lists?**

* Efficient insertion and deletion without shifting elements
* Great for dynamic memory allocation

**Step 2: Java Implementation**

**Task.java**

public class Task {

private int taskId;

private String taskName;

private String status;

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

this.taskId = id;

this.taskName = name;

this.status = status;

}

public int getTaskId() {

return taskId;

}

public String toString() {

return "TaskID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

**TaskNode.java**

public class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

**TaskLinkedList.java**

public class TaskLinkedList {

private TaskNode head;

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public void deleteTask(int taskId) {

if (head == null) return;

if (head.task.getTaskId() == taskId) {

head = head.next;

return;

}

TaskNode current = head;

while (current.next != null) {

if (current.next.task.getTaskId() == taskId) {

current.next = current.next.next;

return;

}

current = current.next;

}

}

public Task searchTask(int taskId) {

TaskNode current = head;

while (current != null) {

if (current.task.getTaskId() == taskId) {

return current.task;

}

current = current.next;

}

return null;

}

public void displayTasks() {

TaskNode current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

}

**TaskManager.java**

public class TaskManager {

public static void main(String[] args) {

TaskLinkedList list = new TaskLinkedList();

list.addTask(new Task(1, "Design Module", "Pending"));

list.addTask(new Task(2, "Code Review", "In Progress"));

list.addTask(new Task(3, "Testing", "Pending"));

list.addTask(new Task(4, "Deployment", "Completed"));

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

list.displayTasks();

System.out.println("\nSearching Task ID 3:");

Task found = list.searchTask(3);

System.out.println(found != null ? found : "Task not found");

System.out.println("\nDeleting Task ID 2");

list.deleteTask(2);

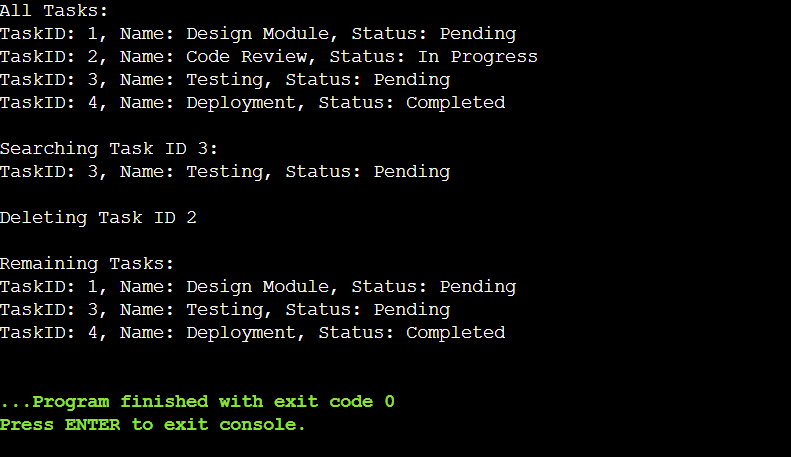
System.out.println("\nTasks After Deletion:");

list.displayTasks();

}

}

**Step 3: Output**

****

**Step 4: Time Complexity Analysis**

| **Operation** | **Time Complexity** |
| --- | --- |
| Add Task | O(n) |
| Search Task | O(n) |
| Delete Task | O(n) |
| Display Tasks | O(n) |

**Step 5: Advantages of Linked Lists Over Arrays**

| **Feature** | **Arrays** | **Linked Lists** |
| --- | --- | --- |
| Fixed Size | Yes | No |
| Insert/Delete Middle | O(n) (with shifting) | O(1) (with pointer change) |
| Memory Use | May waste memory slots | Allocated as needed |
| Random Access | O(1) | O(n) |