**Exercise 1: Inventory Management System**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.

Data structures is a way of organizing the data or structuring the data in the computer memory and algorithm is a step by step procedure to solve a problem. Together, data structure and algorithm is used to solve a problem efficiently. So, it is also very essential in handling large inventories because it makes accessing, storing, and modifying very easy and efficient.

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

ArrayList : Useful when order matters and operations like sorting or iteration are frequent.

HashMap : Ideal for quick lookups based on a unique key like productId.

LinkedList : Good for dynamic data where insertion and deletion are frequent.

TreeMap : Maintains sorted order and allows efficient range queries.

1. **Implementation:**

import java.util.HashMap;

class Product {

int productId;

String productName;

int quantity;

double price;

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

this.productId = id;

this.productName = name;

this.quantity = qty;

this.price = price;

}

}

class Inventory {

HashMap<Integer, Product> inventory = new HashMap<>();

void addProduct(Product product) {

inventory.put(product.productId, product);

}

void updateProduct(int productId, int quantity, double price) {

if (inventory.containsKey(productId)) {

Product p = inventory.get(productId);

p.quantity = quantity;

p.price = price;

}

}

void deleteProduct(int productId) {

inventory.remove(productId);

}

}

1. **Analysis:**

Time complexity : O(1)

For optimizing, Use HashMap for constant-time access. If sorted access is required, use TreeMap instead (O(log n) operations). For large systems, consider caching and indexing for faster access.

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

**Steps:**

1. **Understand Asymptotic Notation:**

Big O Notation:

Big O notation is used to describe the efficiency of an algorithm in terms of time and space. It gives the upper bound of an algorithm's running time as the input size grows, helping developers understand how the algorithm performs at scale.

Search Scenarios:

Best Case: The element is found at the first index.

Average Case: The element is found somewhere in the middle.

Worst Case: The element is not present, or it's at the last index (requiring full traversal).

1. **Implementation:**

class Product {

int productId;

String productName;

String category;

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

this.productId = id;

this.productName = name;

this.category = category;

}

}

class Search {

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

for (int i = 0; i < products.length; i++) {

if (products[i].productName.equalsIgnoreCase(targetName)) {

return i;

}

}

return -1;

}

static int 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 mid;

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

else right = mid - 1;

}

return -1;

}

}

1. **Analysis:**

Time Complexity of linear search is O(n) and binary search is O(log n).

Linear search can be used for unsorted and small cases, whereas binary search can be suitable for large, organized and sorted arrays.

**Exercise 3: Sorting Customer Orders**

**Steps:**

1. **Understand Sorting Algorithms:**

Bubble Sort: Compares adjacent elements and swaps them if out of order. Simple but inefficient (O(n²)).

Insertion Sort: Builds a sorted list one element at a time. Good for small datasets (O(n²)).

Quick Sort: Divides the array using a pivot, then recursively sorts subarrays. Efficient for large datasets (Average: O(n log n)).

Merge Sort: Divides the array into halves, recursively sorts, and merges them (O(n log n), stable).

1. **Implementation:**

class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

static void bubbleSort(Order[] orders) {

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - 1 - i; j++) {

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

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

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);

}

}

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;

}

1. **Analysis:**

Time complexity of bubble sort is O(n), whereas quick sort is O(n logn).

Quick sort is generally preferred as it is generally faster and more memory-efficient than Bubble Sort for large datasets. It's a divide-and-conquer algorithm that efficiently handles large input sizes and is widely used in practice.

**Exercise 4: Employee Management System**

**Steps:**

1. **Understand Array Representation:**

Arrays are stored in contiguous memory locations, meaning each element is placed one after another in memory. This allows:

Fast access via indexing (O(1) time).

Efficient iteration and traversal.

Advantages:

Simple to use.

Quick access to elements via index.

Useful when the number of elements is fixed or known.

1. **Implementation:**

class Employee {

int employeeId;

String name;

String position;

double salary;

Employee(int id, String name, String position, double salary) {

this.employeeId = id;

this.name = name;

this.position = position;

this.salary = salary;

}

}

class EmployeeManagement {

Employee[] employees = new Employee[100]; // Static array

int count = 0;

void addEmployee(Employee e) {

if (count < employees.length) {

employees[count++] = e;

} else {

System.out.println("Employee list is full.");

}

}

Employee searchEmployee(int id) {

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

if (employees[i].employeeId == id) {

return employees[i];

}

}

return null;

}

void displayAll() {

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

System.out.println("ID: " + employees[i].employeeId + ", Name: " + employees[i].name);

}

}

void deleteEmployee(int id) {

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

if (employees[i].employeeId == id) {

// Shift elements

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

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

}

employees[--count] = null;

return;

}

}

}

}

1. **Analysis:**

Time Complexity:

Add: O(1) (at the end)

Search: O(n)

Traverse: O(n)

Delete: O(n) (due to shifting)

Limitations of Arrays:

Fixed size: Cannot grow or shrink dynamically.

Insertion/deletion is costly if not at the end.

Inefficient memory use if most slots are unused.

When to Use Arrays:

When the number of elements is known in advance.

When fast access via indexing is required.

For simple, memory-efficient applications with low insertion/deletion needs.

**Exercise 5: Task Management System**

**Steps:**

1. **Understand Linked Lists:**

Types of Linked Lists:

Singly Linked List: Each node points to the next node. Traversal is only forward.

Doubly Linked List: Each node has pointers to both the next and previous nodes. Allows traversal in both directions.

Circular Linked List: Last node points back to the first node, forming a loop.

They allow dynamic memory allocation, easy insertion and deletion without shifting elements, and are ideal for applications where data changes frequently.

1. **Implementation:**

class Task {

int taskId;

String taskName;

String status;

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

this.taskId = id;

this.taskName = name;

this.status = status;

}

}

class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.next = null;

}

}

class TaskManager {

Node head = null;

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;

}

}

Task searchTask(int id) {

Node current = head;

while (current != null) {

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

return current.task;

}

current = current.next;

}

return null;

}

void displayTasks() {

Node current = head;

while (current != null) {

System.out.println("ID: " + current.task.taskId + ", Name: " + current.task.taskName + ", Status: " + current.task.status);

current = current.next;

}

}

void deleteTask(int id) {

if (head == null) return;

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

head = head.next;

return;

}

Node current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

}

}

}

1. **Analysis:**

Time Complexity:

Add: O(n) (at end)

Search: O(n)

Traverse: O(n)

Delete: O(n)

Advantages of Linked Lists Over Arrays:

Dynamic size — no need to predefine capacity.

Easier insertions and deletions (no shifting).

Efficient for applications where frequent add/delete operations are required.

**Exercise 6: Library Management System**

**Steps:**

1. **Understand Search Algorithms:**

Linear Search: Scans each element one by one. No need for the list to be sorted. Time complexity is O(n).

Binary Search: Works only on sorted arrays. Repeatedly divides the array in half to locate the element. Time complexity is O(log n).

1. **Implementation:**

class Book {

int bookId;

String title;

String author;

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

this.bookId = id;

this.title = title;

this.author = author;

}

}

class Library {

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

for (int i = 0; i < books.length; i++) {

if (books[i].title.equalsIgnoreCase(title)) {

return i;

}

}

return -1;

}

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

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

while (low <= high) {

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

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

if (cmp == 0) return mid;

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

else high = mid - 1;

}

return -1;

}

}

1. **Analysis:**

Time Complexity Comparison:

Linear Search: O(n)

Binary Search: O(log n)

When to Use Each:

Linear Search:

Use when the dataset is small or unsorted.

Easy to implement and flexible.

Binary Search:

Use when the dataset is large and sorted.

Much faster due to halving the search range each time**.**

**Exercise 7: Financial Forecasting**

**Steps:**

1. **Understand Recursive Algorithms:**

What is Recursion?

Recursion is a technique where a function calls itself to solve a smaller version of the original problem. It simplifies code for problems that can be broken down into similar sub-problems.

Why use recursion?

Simplifies logic for problems like factorial, Fibonacci, or tree traversals.

Helps model problems that are naturally recursive (e.g., forecasting based on previous years).

1. **Implementation:**

class Forecast {

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

if (years == 0) return currentValue;

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

}

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 0.05; // 5% growth

int futureYears = 5;

double futureValue = predictValue(initialValue, growthRate, futureYears);

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

}

}

1. **Analysis:**

Time Complexity:

Recursive Approach: O(n), where n is the number of years. Each recursive call handles one year.

Space Complexity: O(n) due to the call stack.

Optimization Tips:

For larger inputs, recursion can lead to stack overflow. To avoid this:

Use an iterative approach for better efficiency.

Or apply memoization if overlapping subproblems exist (not needed in this simple forecast case).