**Exercise 1: Inventory Management System**

**1.Understanding the Problem**

Efficiently managing a large inventory in a warehouse requires the use of appropriate data structures and algorithms to ensure fast data storage, retrieval, and updates.

**Why DS & Algorithms?**  
They ensure fast access, update, and management of large product data, preventing slow performance as inventory grows.

**Suitable DS:**  
Use HashMap for quick lookup by ID (O(1)), ArrayList for simple lists, and TreeMap for sorted product access.

import java.util.\*;

class Product {

int productId;

String productName;

int quantity;

double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public static void details(Product p) {

System.out.println("ID: " + p.productId + "\tName: " + p.productName +

"\tQuantity: " + p.quantity + "kg\tPrice: ₹" + p.price);

}

}

class Inventory {

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

public void add(Product p) {

inventory.put(p.productId, p);

}

public void delete(int id) {

Product removed = inventory.remove(id);

if (removed == null) {

System.out.println("Product ID " + id + " not found for deletion.");

}

}

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

Product p = inventory.get(productId);

if (p != null) {

p.productName = productName;

p.quantity = quantity;

p.price = price;

} else {

System.out.println("Product ID " + productId + " not found for update.");

}

}

public void display() {

if (inventory.isEmpty()) {

System.out.println("Inventory is empty.");

} else {

for (int id : inventory.keySet()) {

Product.details(inventory.get(id));

}

}

}

}

class Main {

public static void main(String[] args) {

Inventory invent = new Inventory();

invent.add(new Product(1, "Jaggery", 2, 200));

invent.add(new Product(2, "Rice", 1, 100));

invent.update(1, "Jaggery", 2, 220); // updated price

invent.display();

}

}

**Time Complexity:**  
Using a HashMap, the time complexity for **add**, **update**, and **delete** operations is **O(1)** on average, due to direct access via keys.

**Optimization:**  
To optimize further, ensure good hash function usage to reduce collisions, and use ConcurrentHashMap in multi-threaded environments for thread safety and performance.

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

**Understanding Asymptotic Notation**:  
**Big O notation** is used to describe the performance or complexity of an algorithm in terms of input size (n). It helps evaluate how the runtime grows as the input grows, ignoring constants and lower-order terms. For search operations, the **best case** occurs when the item is found early (e.g., first index), the **average case** assumes the item is somewhere in the middle, and the **worst case** happens when the item is not found or is at the end of the list.

**Implementation**:  
Store products in a normal array for linear search and a **sorted array** for binary search.

**Linear Search** goes through each element one by one and works on **unsorted arrays**.

**Binary Search** repeatedly divides the search space in half and works only on **sorted arrays**.

**Code:**

class Product {

int productId;

String productName;

String category;

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

productId = id;

productName = name;

category = cat;

}

}

int linearSearch(Product[] products, String name) {

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

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

return i;

}

}

return -1;

}

int binarySearch(Product[] products, String name) {

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

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) return mid;

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

else high = mid - 1;

}

return -1;

}

**Analysis**:

**Linear Search** has a time complexity of **O(n)** in all cases, as it checks each element one by one.

**Binary Search** has a time complexity of **O(log n)** in the best, average, and worst cases, making it much faster for large datasets.  
Thus, **binary search** is more suitable for an e-commerce platform with many products, provided the data is sorted. It significantly improves performance, especially as the product list grows.

**Exercise 3: Sorting Customer Orders**

**Bubble Sort**: Compares adjacent elements and swaps them if needed. Repeats until the list is sorted. Easy to understand but inefficient for large data with time complexity **O(n²)**.

**Insertion Sort**: Builds the sorted list one item at a time. Efficient for small or nearly sorted data. Time complexity: **O(n²)**.

**Quick Sort**: A divide-and-conquer algorithm that selects a pivot, partitions the array, and recursively sorts the parts. Fast in practice with average time complexity **O(n log n)**.

**Merge Sort**: Also divide-and-conquer, divides array into halves, sorts, then merges. Stable and predictable performance, but uses extra space. Time complexity: **O(n log n)**.

**Code:**

class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

void print() {

System.out.println(orderId + " - " + customerName + " - ₹" + totalPrice);

}

}

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(101, "Alice", 1500.0),

new Order(102, "Bob", 2300.0),

new Order(103, "Charlie", 900.0),

new Order(104, "David", 1200.0)

};

Main sorter = new Main();

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

for (Order o : orders) {

o.print();

}

}

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;

}

}

}

}

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);

}

}

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;

}

}

**Bubble Sort** is simple but inefficient and becomes very slow with large datasets.

**Quick Sort** is generally **preferred** because it's much faster for medium and large data sets due to its average-case performance of **O(n log n)**.

**Exercise 4: Employee Management System**

Arrays are a contiguous block of memory where elements of the same type are stored sequentially. This allows for constant-time access (O(1)) to any element by index since the memory address of an element can be calculated directly. Arrays are simple and efficient for fixed-size data, but resizing or inserting/deleting elements in the middle requires shifting elements, which is costly.

**Code**:

class Employee {

int employeeId;

String name;

String position;

double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

void print() {

System.out.println(employeeId + " - " + name + " - " + position + " - ₹" + salary);

}

}

class EmployeeManagement {

Employee[] employees;

int size;

int capacity;

EmployeeManagement(int capacity) {

this.capacity = capacity;

employees = new Employee[capacity];

size = 0;

}

public void add(Employee emp) {

if (size == capacity) {

System.out.println("Array is full. Cannot add more employees.");

return;

}

employees[size++] = emp;

}

public Employee search(int employeeId) {

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

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

return employees[i];

}

}

return null;

}

public void traverse() {

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

employees[i].print();

}

}

public void delete(int employeeId) {

int index = -1;

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

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

index = i;

break;

}

}

if (index == -1) {

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

return;

}

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

employees[i] = employees[i + 1];

}

employees[size - 1] = null;

size--;

}

}

**Analysis:**

**Add:** O(1) if space is available (adding at the end).

**Search:** O(n) linear search by employeeId.

**Traverse:** O(n) to iterate through all employees.

**Delete:** O(n) to find and shift elements after deletion.

**Limitations of Arrays:**

* Fixed size; resizing is expensive as it requires creating a new array and copying.
* Insertion and deletion (except at the end) require shifting elements.
* Better to use dynamic data structures like ArrayList or linked lists when frequent additions/removals are expected.

**Exercise 5: Task Management System**

Understanding Linked Lists

* Singly Linked List: Each node contains data and a pointer/reference to the next node only. Traversal is one-directional from head to tail.
* Doubly Linked List: Each node contains data and two pointers/references — one to the next node and one to the previous node, allowing traversal in both directions.

**Code:**

class Task {

int taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

void print() {

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

}

}

class TaskNode {

Task task;

TaskNode next;

TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

class TaskManagement {

TaskNode head;

public void add(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

public Task search(int taskId) {

TaskNode current = head;

while (current != null) {

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

return current.task;

}

current = current.next;

}

return null;

}

public void traverse() {

TaskNode current = head;

while (current != null) {

current.task.print();

current = current.next;

}

}

public void delete(int taskId) {

if (head == null) return;

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

head = head.next;

return;

}

TaskNode current = head;

TaskNode prev = null;

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

prev = current;

current = current.next;

}

if (current == null) return;

prev.next = current.next;

}

}

**Analysis**

* **Add**: O(n) — need to traverse to the end to add a new node (can be optimized with a tail pointer).
* **Search**: O(n) — must traverse nodes until the task is found.
* **Traverse**: O(n) — visits each node.
* **Delete**: O(n) — search node to delete and update links.

**Advantages of Linked Lists over Arrays**

* Dynamic size: Linked lists can grow and shrink easily without resizing overhead.
* Efficient insertion/deletion: Adding or removing nodes (especially at head or middle) is faster because no shifting of elements is needed.
* Better memory utilization for unpredictable data sizes.

**Exercise 6: Library Management System**

**Understanding Search Algorithms**

* Linear Search: Checks each element one by one until the target is found or the list ends. Works on both sorted and unsorted lists. Time complexity: O(n).
* Binary Search: Efficiently searches in a sorted list by repeatedly dividing the search interval in half. Time complexity: O(log n).

**Code:**

class Book {

int bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

void print() {

System.out.println(bookId + " - " + title + " by " + author);

}

}

class Library {

Book[] books;

Library(Book[] books) {

this.books = books;

}

public Book linearSearchByTitle(String title) {

for (Book book : books) {

if (book.title.equalsIgnoreCase(title)) {

return book;

}

}

return null;

}

public Book binarySearchByTitle(String title) {

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

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) {

return books[mid];

} else if (cmp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

}

**Analysis:**

Linear Search Time Complexity:

* Best, average, and worst case: **O(n)**
* Checks each element one by one until the target is found or list ends.

Binary Search Time Complexity:

* Best, average, and worst case: **O(log n)**
* Requires sorted list and repeatedly halves the search space.

When to Use Linear Search:

* Suitable for small or unsorted datasets.
* Simple to implement, no need for sorting.

When to Use Binary Search:

* Ideal for large, sorted datasets.
* Much faster search compared to linear search.

**Exercise 7: Financial Forecasting**

Understanding Recursive Algorithms

* Recursion is a programming technique where a method calls itself to solve smaller instances of the same problem until it reaches a base case.
* It simplifies problems that have a natural repetitive structure, like calculating factorials or Fibonacci numbers, by breaking them down into smaller subproblems.

**Code:**

public class FinancialForecast {

public static double futureValue(double initialValue, double growthRate, int periods) {

if (periods == 0) {

return initialValue;

}

return futureValue(initialValue, growthRate, periods - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

double initialInvestment = 1000;

double growthRate = 0.05; // 5%

int years = 3;

double fv = futureValue(initialInvestment, growthRate, years);

System.out.println("Future value after " + years + " years: ₹" + fv);

}

}

**Analysis**

* Time Complexity:  
  The recursive method calls itself once per period, so time complexity is O(n) for n periods.
* Optimization to Avoid Excessive Computation:  
  Since this recursion is simple and linear, it’s efficient enough. But for more complex recursive problems (like Fibonacci), memoization or dynamic programming can be used to store and reuse results of subproblems, avoiding repeated calculations.  
  For this financial forecasting, an iterative solution might be simpler and more efficient in practice.