**MODEL 2 - ALGORITHMS DATA STRUCTURES**

**Exercise 1: Inventory Management System**

1. **Understand the Problem**

**Why DS & Algos Matter**:

Efficient retrieval, updates, and deletions are *critical* when handling thousands of products.

* + Poor design can lead to slow searches, data inconsistencies, or even app crashes under scale.
* **Recommended Data Structures**:
* **ArrayList**: Good for small, ordered data but inefficient for key-based lookups.
* **HashMap / Dictionary**: Best for key-based access (e.g., by productId); constant-time performance on average.
* **Balanced BSTs or Tries** (for more advanced use cases): Support sorted inventory or prefix searches.

2. **Setup: Project Skeleton**

import java.util.\*;

class Product {

int productId;

String productName;

int quantity;

double price;

public Product(int id, String name, int qty, double pr) {

productId = id;

productName = name;

quantity = qty;

price = pr;

}

}

class Inventory {

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

void addProduct(Product p) {

inventory.put(p.productId, p);

}

void updateProduct(Product p) {

inventory.put(p.productId, p);

}

void deleteProduct(int productId) {

inventory.remove(productId);

}

}

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

1.

i)Explain Big O notation and how it helps in analyzing algorithms.

Answer:- Big O notation is the asymptotic notation which works or describes the Upper bound of an algorithm time or space complexity for the input size.

It gives us the worst scenario for which we can know how our algorithm will work for the desired input.

ii) Describe the best, average, and worst-case scenarios for search operations.

**Best Case**- if we got the desired output in no time or very minimal time then it is best case. Ex. Searching an element and when we search the first element is what we are searching for.

**Average Case**- An calculated time taken for all inputs i.e. what we have expected till which time it will be taking. It works for all possible inputs.

**Worst Case-** The maximum time taken for a desired output . Ex. When we search for an element if it is present at last of and array/list or it is not present .

**2. code Implementation:**

**class SearchProduct {**

**int productId;**

**String productName;**

**String category;**

**public SearchProduct(int id, String name, String cat) {**

**productId = id;**

**productName = name;**

**category = cat;**

**}**

**}**

**class SearchFunctions {**

**public static int linearSearch(SearchProduct[] arr, String name) {**

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

**if (arr[i].productName.equals(name)) return i;**

**}**

**return -1;**

**}**

**public static int binarySearch(SearchProduct[] arr, String name) {**

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

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

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

**int comp = arr[mid].productName.compareTo(name);**

**if (comp == 0) return mid;**

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

**else high = mid - 1;**

**}**

**return -1;**

**}**

**}**

**Exercise 3: Sorting Customer Orders**

**1. Bubble Sort**

Think of this one as the “compare and swap” dance.

* **How it works**: It repeatedly steps through the list, compares adjacent elements, and swaps them if they’re in the wrong order.
* **Time Complexity**: O(n²) — not ideal for large datasets.
* **Best for**: Simple educational demos, or tiny lists where performance doesn’t matter.

**2. Insertion Sort**

Like sorting playing cards in your hand.

* **How it works**: Builds the sorted list one item at a time, inserting each new element into its proper place.
* **Time Complexity**: O(n²), but performs well on nearly sorted or small lists.
* **Best for**: Small datasets or nearly sorted arrays.

**3. Quick Sort**

A divide-and-conquer wizard.

* **How it works**: Picks a “pivot” element, partitions the array into two subarrays (less than and greater than the pivot), then sorts the subarrays recursively.
* **Time Complexity**: Average case is O(n log n); worst case (already sorted input with bad pivot choice) is O(n²).
* **Best for**: General-purpose efficient sorting—often the go-to choice in practice.

**4. Merge Sort**

Precision and reliability wrapped in recursion.

* **How it works**: Divides the array into halves, recursively sorts them, and merges the sorted halves.
* **Time Complexity**: Always O(n log n), but uses extra space for merging.
* **Best for**: Sorting linked lists, or when stable sort is required.

**Implementation**

class Order {

int orderId;

String customerName;

double totalPrice;

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

orderId = id;

customerName = name;

totalPrice = price;

}

}

class OrderSorting {

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;

}

}

**Exercise 4: Employee Management System**

1. **Array Representation in Memory**

Arrays are stored in **contiguous memory locations**, meaning elements are placed right next to each other in memory. This enables:

* **Direct access** via indexing (O(1) time complexity)
* **Efficient memory use**, especially for homogeneous data types
* **Cache friendliness** due to sequential memory layout

class Employee {

int employeeId;

String name;

String position;

double salary;

public Employee(int id, String n, String pos, double sal) {

employeeId = id;

name = n;

position = pos;

salary = sal;

}

}

class EmployeeArray {

Employee[] employees;

int count = 0;

public EmployeeArray(int size) {

employees = new Employee[size];

}

void addEmployee(Employee emp) {

employees[count++] = emp;

}

Employee searchEmployee(int id) {

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

if (employees[i].employeeId == id) return employees[i];

}

return null;

}

void traverse() {

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

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

}

}

void deleteEmployee(int id) {

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

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

employees[i] = employees[--count];

break;

}

}

}

}

**5. Limitations of Arrays**

* **Fixed size**: Static allocation requires knowing capacity in advance
* **Costly insertions/deletions**: Need element shifting for mid-array operations
* **Wasted space or overflow** if actual data doesn't match estimated capacity

When to Use Arrays

* When data size is known and fixed
* When constant-time access is a priority
* For tasks involving indexed lookup (e.g., CPU registers, look-up tables)

**Exercise 5: Task Management System**

**1. Understanding Linked Lists**

* **Singly Linked List:**

**Each node holds data and a pointer to the *next* node.**

**Simple to implement and uses less memory than doubly linked lists.**

**Limitation: Can only traverse forward.**

* **Doubly Linked List:**

**Each node has data, a pointer to the *next*, and a pointer to the *previous* node.**

**Allows bidirectional traversal.**

**More flexible but consumes extra memory for the backward pointer**

**class Task {**

**int taskId;**

**String taskName;**

**String status;**

**Task next;**

**public Task(int id, String name, String stat) {**

**taskId = id;**

**taskName = name;**

**status = stat;**

**next = null;**

**}**

**}**

**class TaskLinkedList {**

**Task head;**

**void addTask(Task t) {**

**t.next = head;**

**head = t;**

**}**

**Task searchTask(int id) {**

**Task current = head;**

**while (current != null) {**

**if (current.taskId == id) return current;**

**current = current.next;**

**}**

**return null;**

**}**

**void traverse() {**

**Task current = head;**

**while (current != null) {**

**System.out.println(current.taskName);**

**current = current.next;**

**}**

**}**

**void deleteTask(int id) {**

**if (head == null) return;**

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

**head = head.next;**

**return;**

**}**

**Task current = head;**

**while (current.next != null) {**

**if (current.next.taskId == id) {**

**current.next = current.next.next;**

**return;**

**}**

**current = current.next;**

**}**

**}**

**}**

**Advantages Over Arrays**

* **Dynamic sizing: Easily grow or shrink without reallocation.**
* **Efficient insertions/deletions: No shifting needed, especially at the start.**
* **No pre allocated memory: Saves space if exact size is unpredictable.**

**Exercise 6: Library Management System**

1. **Understanding Search Algorithms**

* **Linear Search**  
  Scans each element in the list one by one until it finds the target or reaches the end.
  + Works on *unsorted* data
  + Simple, but slow for large datasets
* **Binary Search**  
  Efficiently narrows down the search by halving the list each time.
* Requires the list to be *sorted*
* Much faster for large, sorted datasets

class Book {

int bookId;

String title;

String author;

public Book(int id, String t, String a) {

bookId = id;

title = t;

author = a;

}

}

class BookSearch {

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

for (Book b : books) {

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

}

return null;

}

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

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

while (low <= high) {

int mid = (low + high) / 2;

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

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

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

else high = mid - 1;

}

return null;

}

}

* **Use Linear Search**: When data isn't sorted and the list is small or rarely searched.
* **Use Binary Search**: When you’re dealing with frequent searches on sorted or large datasets.

**Exercise 7: Financial Forecasting**

1. **Understanding Recursive Algorithms**

Recursion is when a function calls itself to solve a problem by breaking it down into smaller subproblems. It’s especially useful when the problem has a **self-similar structure**, like computing factorials, tree traversals, or—our case—compounding values over time.

*Key components of recursion*:

* **Base Case**: The stopping condition
* **Recursive Case**: Reduces the problem and calls itself

class Forecast {

public static double predictFuture(double current, double rate, int years) {

if (years == 0) return current;

return predictFuture(current \* (1 + rate), rate, years - 1);

}

}

**Optimization Options**

* **Tail Recursion** (not natively optimized in Python)
* **Memoization**: Cache previously calculated results
* **Iterative Approach**: Convert to a loop for better space efficiency

def future\_value\_iter(pv, rate, years): for \_ in range(years): pv \*= (1 + rate) return pv