**WEEK-1 ( ALGORITHMS\_DATA STRUCTURES HANDS-ON)**

**(MANDATORY HANDS-ON)**

**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.

**SOLUTION :**

**(i)Big O Notation:**

Big O notation is used to describe the **time complexity** or **space complexity** of an algorithm in the **worst-case scenario**. It provides an upper bound on the growth rate of an algorithm as the input size increases. It helps in analyzing the scalability and efficiency of algorithms independent of machine or compiler variations.

* For example, if an algorithm has a time complexity of **O(n)**, it means the time taken grows linearly with the size of input n.
* If it’s **O(log n)**, the time grows logarithmically, which is faster.

**Best, Average, and Worst Case:**

* **Best Case**: Minimum time taken by the algorithm for the smallest input scenario. E.g., in linear search, if the element is the first, time is **O(1)**.
* **Average Case**: Expected time taken, assuming the input is random. For linear search, this would be **O(n/2)** → simplified to **O(n)**.
* **Worst Case**: Maximum time taken by the algorithm. In linear search, if the element is not present or is the last, time is **O(n)**.

**(ii) Product class SetUp :**

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return productId + ": " + productName + " (" + category + ")";

}

}

**(iii) Implementation :**

* 1. **Linear Search**

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

for (Product product : products) {

if (product.productName.equalsIgnoreCase(key)) {

return product;

}

}

return null;

}

* 1. **Binary Search**

import java.util.Arrays;

import java.util.Comparator;

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

int comparison = products[mid].productName.compareToIgnoreCase(key);

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

**(iv)** **Analysis :**

Time Complexity Comparison:

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) | O(1) |
| **Binary Search** | O(1) | O(log n) | O(log n) | O(1) |

**Which Algorithm is More Suitable?**

* **Linear Search** is simple and doesn’t require sorted data. It is ideal for small datasets or unsorted data. However, it's inefficient for large lists.
* **Binary Search** is **much faster** (logarithmic time) for large datasets but requires data to be sorted. Sorting takes O(n log n) time, but if we’re frequently searching and infrequently updating the list, sorting once is worth it.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

1. **Recursion**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It is especially useful for problems that can be broken down into repetitive sub-problems, like factorials, Fibonacci series etc.

Ex: Finding factorial of a number

int factorial(int n) {

if (n == 0 || n == 1) return 1;

return n \* factorial(n - 1);

}

**(ii) SetUp:**

We are forecasting future financial values based on a constant growth rate.

The formula we'll use recursively is:

FV(n) = FV(n-1) \* (1 + growthRate)

Where:

* FV(n) = future value in year n
* growthRate = annual growth rate (e.g., 0.1 for 10%)
* FV(0) = initial investment or base value

**(iii) Implementation :**

public class FinancialForecasting {

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

if (years == 0) {

return currentValue;

}

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

}

public static void main(String[] args) {

double initialInvestment = 10000.0;

double annualGrowthRate = 0.08;

int forecastYears = 5;

double futureValue = calculateFutureValue(forecastYears, initialInvestment, annualGrowthRate);

System.out.printf("Future value after %d years: %.2f\n", forecastYears, futureValue);

}

}

**(iv) Analysis**

(a) Time Complexity:

The time complexity of this recursive algorithm is O(n), where n is the number of years being forecasted.

* This is because the function performs one recursive call per year.
* There is no overlapping subproblem, so memoization is not needed here.

(b) Space Complexity:

The space complexity is also O(n) because each recursive call adds a new frame to the call stack.

To optimize a recursive solution and avoid excessive computation:

1. **Use memoization** – Store the results of already computed subproblems in a map or array so they don't get calculated again.
2. **Convert to iteration** – Replace the recursive approach with a loop (iterative version), which is faster and avoids stack overflow.
3. **Avoid deep recursion** – Limit the depth of recursion if possible, or use tail recursion (if the language supports optimization).

**( ADDITIONAL HANDS-ON )**

**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

1. **Understanding Data Structures**
2. we need data structures and algorithms to store and manage a large number of products efficiently. They help in fast searching, adding, updating, and deleting items. Without proper DS, system will be slow and messy.
3. Suitable Data Structures:

* ArrayList : Good for small data, but slow for searching/updating by ID.
* HashMap : Fast in lookup, update, and delete.

So I’ll go with HashMap.

**(ii) & (iii) SetUP and Implementation**

**Product Class**

public class Product {

int productId;

String productName;

int quantity;

double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

}

**Inventoty class**

import java.util.HashMap;

public class Inventory {

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

public void addProduct(Product p) {

products.put(p.productId, p);

}

public void updateProduct(int id, String name, int qty, double price) {

if (products.containsKey(id)) {

Product p = products.get(id);

p.productName = name;

p.quantity = qty;

p.price = price;

}

}

public void deleteProduct(int id) {

products.remove(id);

}

}

**(iv) Analysis**

* Time Complexities :
  + Add: O(1)
  + Update: O(1)
  + Delete: O(1)

**Optimizations:**

1)Avoid Duplicate IDs : Before adding, check if the productId already exists. This avoids data issues.

2)If implemented in ArrayList recommended to use HashMap since it is much faster than using ArrayList where you search linearly (O(n)).

**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.

**(i)Understanding Sorting Techniques :**

* **Bubble Sort:**
  + Compares two elements and swaps them if needed.
  + Repeats till everything is sorted.
  + **Slow** for big data (O(n²))
  + Easy to write.
* **Insertion Sort:**
  + Picks elements one by one and puts them in the right place.
  + Good for small datasets.
  + Also O(n²).
* **Quick Sort:**
  + Picks a pivot, splits data into smaller and bigger parts, and sorts recursively.
  + Very fast for big data.
  + Avg time = **O(n log n)**
* **Merge Sort:**
  + Splits data in half, sorts both halves, and merges them.
  + Time = O(n log n), but uses more memory.

**(ii) SetUp :**

public class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

**(iii) Implementation :**

1. Bubble Sort

public 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) {

// Swap

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

1. Quick Sort

public 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);

}

}

public 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;

}

**(iv) Analysis**

* **Bubble Sort:**
  + Time Complexity: **O(n²)**
  + Bad for large data. Too many comparisons.
* **Quick Sort:**
  + Avg Time: **O(n log n)**
  + Fast and efficient.

**Why Quick Sort is better than Bubble Sort?**

1. Bubble Sort is simple, but **too slow**.
2. Quick Sort is much **faster**, especially with large orders.
3. That’s why we prefer Quick Sort in real-world problems.

**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**(i) Understanding Array Representation**

* **How arrays are stored:**  
  Arrays are stored in **continuous memory** blocks. Each element is placed next to the other.
* **Advantages:**
  + Fast access using index (O(1)).
  + Simple to use.
  + Good for fixed-size data.

**(ii) SetUp:**

public class Employee {

int employeeId;

String name;

String position;

double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

}

**(iii) Implementation :**

public class EmployeeSystem {

Employee[] employees = new Employee[100];

int count = 0;

public void addEmployee(Employee e) {

if (count < employees.length) {

employees[count] = e;

count++;

}

}

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

public void displayEmployees() {

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

System.out.println(employees[i].name + " - " + employees[i].position);

}

}

public void deleteEmployee(int id) {

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

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

// shift elements left

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

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

}

employees[count - 1] = null;

count--;

break;

}

}

}

}

**(iv) Analysis**

* **Add:** O(1) → adding at end
* **Search:** O(n) → need to check one by one
* **Traverse:** O(n)
* **Delete:** O(n) → shifting needed

**Limitations of Arrays**

* Fixed size (can't grow automatically).
* Insertion/deletion in middle is slow.
* Not good for big dynamic data.

**When to Use Arrays**

* When size is known and fixed.
* When you need fast index access.
* For small/medium size datasets.

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**(i) Understanding Linked Lists**

* **Singly Linked List:**  
  Each node has data + next pointer.  
  Can go forward only.
* **Doubly Linked List:**  
  Each node has data + next + prev pointer.  
  Can go forward and backward.  
  Takes more memory but more flexible.

**(ii) SetUp**

public class Task {

int taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

}

**(ii) Implementation**

class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

public class TaskManager {

Node head = null;

public void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public Task searchTask(int id) {

Node temp = head;

while (temp != null) {

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

return temp.task;

}

temp = temp.next;

}

return null;

}

public void displayTasks() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

}

public void deleteTask(int id) {

if (head == null) return;

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

head = head.next;

return;

}

Node temp = head;

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

temp = temp.next;

}

if (temp.next != null) {

temp.next = temp.next.next;

}

}

}

**(iv) Analysis**

* **Add:** O(n) – adding at end
* **Search:** O(n) – need to traverse
* **Traverse:** O(n)
* **Delete:** O(n) – find node, then unlink

**Advantages of Linked List over Array**

* Dynamic size – can grow/shrink as needed
* Easy insertion/deletion – no shifting needed
* Memory efficient, if size keeps changing
* No fixed limit like arrays

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**(i) Understanding Search Algorithms**

* **Linear Search:**
  + Go through the list one by one.
  + Works on unsorted data.
  + Simple, but slow.
  + Time: O(n)
* **Binary Search:**
  + Works on sorted list only.
  + Divide and search (like cutting in half).
  + Much faster.
  + Time: O(log n)

**(ii) SetUp**

public class Book {

int bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

}

**(iii) Implementation**

* 1. Linear Search

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

for (Book book : books) {

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

return book;

}

}

return null;

}

* 1. Binary Search

import java.util.Arrays;

import java.util.Comparator;

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

int left = 0;

int right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) {

return books[mid];

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// Sorting the array before binary search

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

**(iv) Analysis**

* **Linear Search:**
  + Time: O(n)
  + Works on any list (sorted or not).
  + Good for small data.
* **Binary Search:**
  + Time: O(log n)
  + Needs sorted data.
  + Super fast for large lists.

**When to Use What?**

* Use **linear search** if:
  + List is small
  + Or unsorted
  + Or only occasional search
* Use **binary search** if:
  + List is big and sorted
  + You search a lot
  + Want better speed