**HANDSON EXERCISES - WEEK 1**

**Skill : Data Structures and Algorithms**

**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.

#### Big O Notation:

Big O notation describes how the **time or space complexity** of an algorithm grows as the input size increases. It helps us analyze how efficient an algorithm is.

Examples:

**O(1)** → constant time (best)

**O(n)** → linear time (slower as data grows)

**O(log n)** → logarithmic time (very fast even for big data)

### Best, Average, and Worst-Case Time Complexities for Searches:

**Linear Search:**  
Best Case: O(1)  
Average Case: O(n)  
Worst Case: O(n)

**Binary Search:**  
Best Case: O(1)  
Average Case: O(log n)  
Worst Case: O(log n)

**CODE :**

**import java.util.Arrays;**

**import java.util.Comparator;**

**public class ECommerceSearch {**

**static class Product {**

**String productId;**

**String productName;**

**String category;**

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

**this.productId = productId;**

**this.productName = productName;**

**this.category = category;**

**}**

**public void display() {**

**System.out.println("ID: " + productId + ", Name: " + productName + ", Category: " + category);**

**}**

**}**

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

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

**if (p.productName.equalsIgnoreCase(targetName)) {**

**return p;**

**}**

**}**

**return null;**

**}**

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

**int low = 0;**

**int high = products.length - 1;**

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

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

**int compare = products[mid].productName.compareToIgnoreCase(targetName);**

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

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

**else high = mid - 1;**

**}**

**return null;**

**}**

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

**Product[] products = {**

**new Product("P101", "Shoes", "Footwear"),**

**new Product("P102", "Shirt", "Clothing"),**

**new Product("P103", "Watch", "Accessories"),**

**new Product("P104", "Laptop", "Electronics"),**

**new Product("P105", "Bag", "Bags")**

**};**

**System.out.println("=== Linear Search ===");**

**Product result1 = linearSearch(products, "Watch");**

**if (result1 != null) result1.display();**

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

**Arrays.sort(products, Comparator.comparing(p -> p.productName));**

**System.out.println("=== Binary Search ===");**

**Product result2 = binarySearch(products, "Watch");**

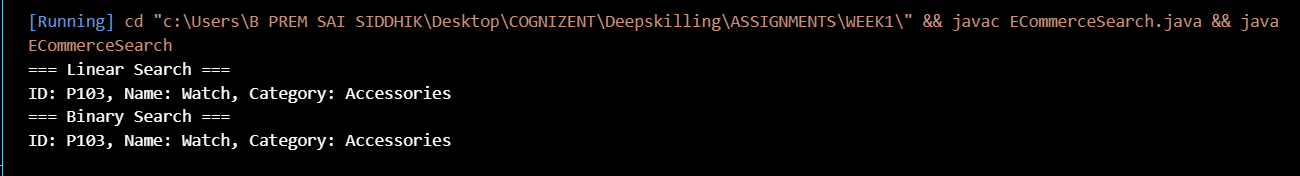
**if (result2 != null) result2.display();**

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

**}**

**}**

**Output :**

****

Used a HashMap to store products by product ID,so time complexities for different operations are:

**Add:** O(1) – uses put(), adds directly.

**Update:** O(1) – accesses product by ID and updates fields.

**Delete:** O(1) – removes entry by ID.

The main difference is that binary search requires sorting , where as linear search searches through all linearly.

**Best Approach for our problem :**

· For **small datasets**, linear search is fine.

· For **large datasets**, **binary search is much faster**, but the data must be sorted.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Understanding Recursive Algorithms:**

* Recursion is when a method calls itself to solve a smaller version of the problem.
* It can be useful in problems that have a repeated pattern or where the solution builds upon smaller results.
* In forecasting, we can use recursion to repeatedly apply a growth rate to calculate future values.

**CODE**

**public class FinancialForecast {**

**static double forecast(double currentValue, double growthRate, int years) {**

**if (years == 0) {**

**return currentValue;**

**}**

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

**}**

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

**double startingValue = 10000;**

**double growthRate = 0.08;**

**int years = 5;**

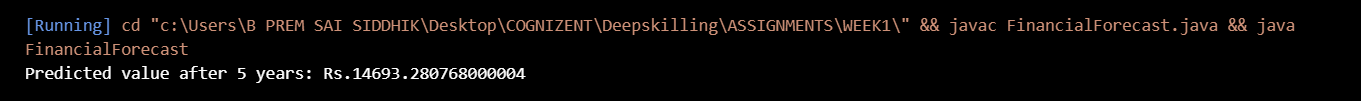
**double futureValue = forecast(startingValue, growthRate, years);**

**System.out.println("Predicted value after " + years + " years: Rs." + futureValue);**

**}**

**}**

**Output :**

****

**Time Complexity Analysis:**

The recursive method runs once for every year, so its time complexity is **O(n)**, where n is the number of years.Linear time , but uses call stack memory

**Optimizing :**

Since this problem is simple and doesn’t repeat subproblems, memoization is not needed. But if recursion gets too deep or performance becomes an issue, it can be replaced with an iterative approach which is usually faster and more memory-efficient.

**Exercise 1: Inventory Management System**

**Scenario:**

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

**Why are Data Structures and Algorithms important ?**

· To store and access large product data efficiently.

· To reduce the time it takes to search, update, or delete a product.

· To keep the system fast as the inventory grows.

**Suitable Data Structures for this problem are :**

* **ArrayList** – A good choice when the focus is mostly on adding and showing products. It maintains the order in which items are added and is simple to use.
* **HashMap** – Ideal when we need to quickly find, update, or remove a product using its ID. It gives constant-time performance for these operations and is the most efficient option for managing products by their ID.

**CODE:**

**import java.util.HashMap;**

**public class InventoryManagementSystem {**

**class Product {**

**String productId;**

**String productName;**

**int quantity;**

**double price;**

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

**this.productId = productId;**

**this.productName = productName;**

**this.quantity = quantity;**

**this.price = price;**

**}**

**public void display() {**

**System.out.println("ID: " + productId + ", Name: " + productName + ", Qty: " + quantity + ", Price: Rs." + price);**

**}**

**}**

**class Inventory {**

**private HashMap<String, Product> products = new HashMap<>();**

**public void addProduct(Product p) {**

**if (products.containsKey(p.productId)) {**

**System.out.println("Product with ID " + p.productId + " already exists.");**

**} else {**

**products.put(p.productId, p);**

**System.out.println("Product added: " + p.productName);**

**}**

**}**

**public void updateProduct(String 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;**

**System.out.println("Product updated: " + id);**

**} else {**

**System.out.println("No product found with ID " + id);**

**}**

**}**

**public void deleteProduct(String id) {**

**if (products.remove(id) != null) {**

**System.out.println("Product removed: " + id);**

**} else {**

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

**}**

**}**

**public void listProducts() {**

**if (products.isEmpty()) {**

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

**return;**

**}**

**System.out.println("Product List:");**

**for (Product p : products.values()) {**

**p.display();**

**}**

**}**

**}**

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

**InventoryManagementSystem system = new InventoryManagementSystem();**

**Inventory inventory = system.new Inventory();**

**Product p1 = system.new Product("P101", "Wireless Mouse", 40, 550.0);**

**Product p2 = system.new Product("P102", "USB Keyboard", 25, 700.0);**

**inventory.addProduct(p1);**

**inventory.addProduct(p2);**

**inventory.listProducts();**

**inventory.updateProduct("P102", "Mechanical Keyboard", 20, 999.0);**

**inventory.deleteProduct("P101");**

**inventory.listProducts();**

**}**

**}**

**Output :**

****

**Time Complexity Analysis:**

* **Adding, updating, or deleting a product** using a HashMap takes constant time, i.e., **O(1)**, because we use the product ID as the key.
* **Viewing all products** takes **O(n)** time since we need to loop through the entire collection.
* Since HashMap is already efficient for ID-based access, if we want to **sort or display products by name or price**, we can first convert the values() of the map into a list and then sort it using Collections.sort() with a custom comparator.

**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.

**Understand Sorting Algorithms:**

Sorting is used to arrange data in a specific order. In this case, we’re sorting customer orders by their total price, so we can handle the highest-value orders first.

There are many sorting algorithms. Some basic ones are:

**Bubble Sort:** Compares two elements at a time and keeps swapping them if they’re in the wrong order. It’s simple but slow for large lists.

**Insertion Sort:** Builds the sorted list one item at a time. It’s better than bubble sort for small lists.

**Quick Sort:** Very fast. It picks a pivot element, and rearranges the list so that elements smaller than the pivot go to one side and larger go to the other. It then sorts the two halves recursively.

**Merge Sort:** Also fast and works by splitting the list into halves, sorting them, and then merging the sorted parts.

**CODE :**

**public class OrderSorting {**

**static class Order {**

**String orderId;**

**String customerName;**

**double totalPrice;**

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

**this.orderId = orderId;**

**this.customerName = customerName;**

**this.totalPrice = totalPrice;**

**}**

**public void display() {**

**System.out.println("Order ID: " + orderId + ", Name: " + customerName + ", Price: Rs." + totalPrice);**

**}**

**}**

**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);**

**}**

**}**

**public 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;**

**}**

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

**Order[] orders = {**

**new Order("O101", "Ravi", 1200.0),**

**new Order("O102", "Priya", 3400.0),**

**new Order("O103", "Amit", 800.0),**

**new Order("O104", "Neha", 2500.0)**

**};**

**System.out.println("Before Sorting:");**

**for (Order o : orders) o.display();**

**bubbleSort(orders);**

**System.out.println("\nAfter Bubble Sort (By Price):");**

**for (Order o : orders) o.display();**

**Order[] orders2 = {**

**new Order("O101", "Ravi", 1200.0),**

**new Order("O102", "Priya", 3400.0),**

**new Order("O103", "Amit", 800.0),**

**new Order("O104", "Neha", 2500.0)};**

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

**System.out.println("\nAfter Quick Sort (By Price):");**

**for (Order o : orders2) o.display();**

**}**

**}**

**Output :**

****

**Time Complexity Analysis:**

* Bubble Sort is slow (O(n²)) and not suitable for large data.
* Quick Sort is faster (O(n log n)) and better for big lists, so it's preferred in real-world systems like e-commerce platforms.

**Exercise 4: Employee Management System**

**Scenario:**

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

**Understanding Array Representation:**

* Arrays are stored in a continuous block of memory, which makes accessing elements fast using an index.
* They are useful when the number of elements is fixed or known in advance.
* Arrays are easy to use and provide fast access, but inserting and deleting elements can be slower if the size changes frequently.

**CODE :**

**import java.util.Scanner;**

**public class EmployeeManagement {**

**static 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 display() {**

**System.out.println("ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: Rs." + salary);**

**}**

**}**

**static Employee[] employees = new Employee[100];**

**static int count = 0;**

**static void addEmployee(int id, String name, String position, double salary) {**

**employees[count++] = new Employee(id, name, position, salary);**

**}**

**static void searchEmployee(int id) {**

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

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

**employees[i].display();**

**return;**

**}**

**}**

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

**}**

**static void traverseEmployees() {**

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

**employees[i].display();**

**}**

**}**

**static void deleteEmployee(int id) {**

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

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

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

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

**}**

**employees[--count] = null;**

**System.out.println("Employee deleted.");**

**return;**

**}**

**}**

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

**}**

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

**addEmployee(101, "Ravi", "Manager", 60000);**

**addEmployee(102, "Priya", "Developer", 45000);**

**addEmployee(103, "Amit", "Designer", 40000);**

**System.out.println("All Employees:");**

**traverseEmployees();**

**System.out.println("\nSearch Employee with ID 102:");**

**searchEmployee(102);**

**System.out.println("\nDeleting Employee with ID 101:");**

**deleteEmployee(101);**

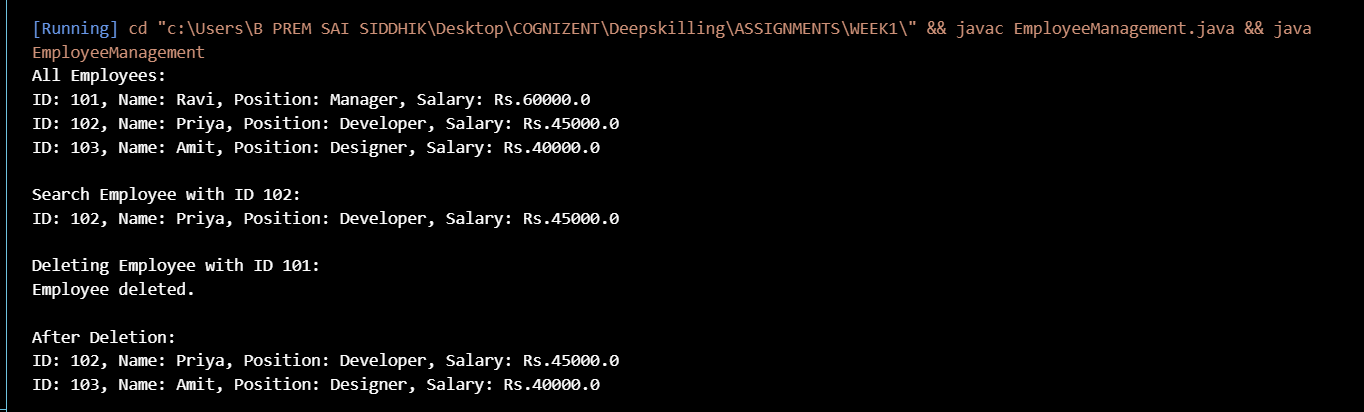
**System.out.println("\nAfter Deletion:");**

**traverseEmployees();**

**}**

**}**

**Output :**



**Time Complexity Analysis:**

* Adding an employee at the end takes O(1) time.
* Searching takes O(n) time since we have to check each element one by one.
* Traversing also takes O(n) because we need to go through every record.
* Deleting takes O(n) because after finding the employee, we shift the remaining elements.

**Limitations of using Arrays:**

Arrays are simple and fast for access, but they are not flexible when it comes to dynamic size changes. If the number of employees changes often or if we need faster insert/delete operations, it’s better to use a dynamic structure like ArrayList or LinkedList.

**Exercise 5: Task Management System**

**Scenario:**

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

**Understanding Linked Lists:**

Linked lists are used to store data in a chain-like structure using nodes. Each node contains data and a reference to the next node. There are two main types:

* **Singly Linked List:** Each node points only to the next node.
* **Doubly Linked List:** Each node has references to both the next and previous nodes.

**CODE :**

**public class TaskManagement {**

**static class Task {**

**int taskId;**

**String taskName;**

**String status;**

**Task next;**

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

**this.taskId = taskId;**

**this.taskName = taskName;**

**this.status = status;**

**this.next = null;**

**}**

**void display() {**

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

**}**

**}**

**static Task head = null;**

**static void addTask(int id, String name, String status) {**

**Task newTask = new Task(id, name, status);**

**if (head == null) {**

**head = newTask;**

**} else {**

**Task temp = head;**

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

**temp = temp.next;**

**}**

**temp.next = newTask;**

**}**

**}**

**static void traverseTasks() {**

**Task temp = head;**

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

**temp.display();**

**temp = temp.next;**

**}**

**}**

**static void searchTask(int id) {**

**Task temp = head;**

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

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

**temp.display();**

**return;**

**}**

**temp = temp.next;**

**}**

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

**}**

**static void deleteTask(int id) {**

**if (head == null) {**

**System.out.println("No tasks to delete.");**

**return;**

**}**

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

**head = head.next;**

**System.out.println("Task deleted.");**

**return;**

**}**

**Task prev = head;**

**Task curr = head.next;**

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

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

**prev.next = curr.next;**

**System.out.println("Task deleted.");**

**return;**

**}**

**prev = curr;**

**curr = curr.next;**

**}**

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

**}**

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

**addTask(1, "Design UI", "Pending");**

**addTask(2, "Implement Backend", "In Progress");**

**addTask(3, "Testing", "Pending");**

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

**traverseTasks();**

**System.out.println("\nSearch Task with ID 2:");**

**searchTask(2);**

**System.out.println("\nDelete Task with ID 1:");**

**deleteTask(1);**

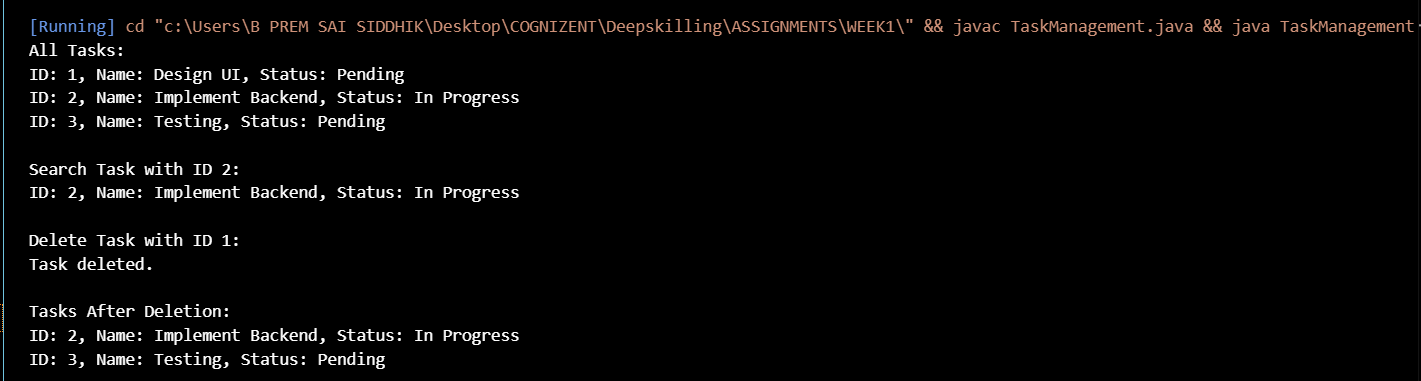
**System.out.println("\nTasks After Deletion:");**

**traverseTasks();**

**}**

**}**

**Output :**



**Time Complexity Analysis:**

* **Add:** O(n) – we add at the end, so we need to traverse the list.
* **Search:** O(n) – we have to go through each node until we find the task.
* **Traverse:** O(n) – we visit each task one by one.
* **Delete:** O(n) – we need to find the task and adjust the links.

**Advantages if Linked List over Array :**

* They are flexible for dynamic data where size keeps changing.
* Inserting or deleting in the middle is easier because we just need to adjust the links.

**Exercise 6: Library Management System**

**Scenario:**

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

**Understanding Search Algorithms:**

* **Linear Search** goes through each element one by one until it finds the target. It works for unsorted data.
* **Binary Search** is faster but only works if the data is sorted. It keeps dividing the search space in half to find the target.

**CODE :**

**import java.util.Arrays;**

**public class LibrarySearch {**

**static 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 display() {**

**System.out.println("ID: " + bookId + ", Title: " + title + ", Author: " + author);**

**}**

**}**

**static void linearSearch(Book[] books, String searchTitle) {**

**boolean found = false;**

**for (Book book : books) {**

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

**book.display();**

**found = true;**

**}**

**}**

**if (!found) {**

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

**}**

**}**

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

**int low = 0;**

**int high = books.length - 1;**

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

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

**int result = books[mid].title.compareToIgnoreCase(searchTitle);**

**if (result == 0) {**

**return mid;**

**} else if (result < 0) {**

**low = mid + 1;**

**} else {**

**high = mid - 1;**

**}**

**}**

**return -1;**

**}**

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

**Book[] books = {**

**new Book(101, "Java Programming", "James Gosling"),**

**new Book(102, "Data Structures", "Mark Allen"),**

**new Book(103, "Operating Systems", "Silberschatz"),**

**new Book(104, "Computer Networks", "Andrew Tanenbaum")**

**};**

**System.out.println("Linear Search for 'Data Structures':");**

**linearSearch(books, "Data Structures");**

**Arrays.sort(books, (a, b) -> a.title.compareToIgnoreCase(b.title));**

**System.out.println("\nBinary Search for 'Operating Systems':");**

**int index = binarySearch(books, "Operating Systems");**

**if (index != -1) {**

**books[index].display();**

**} else {**

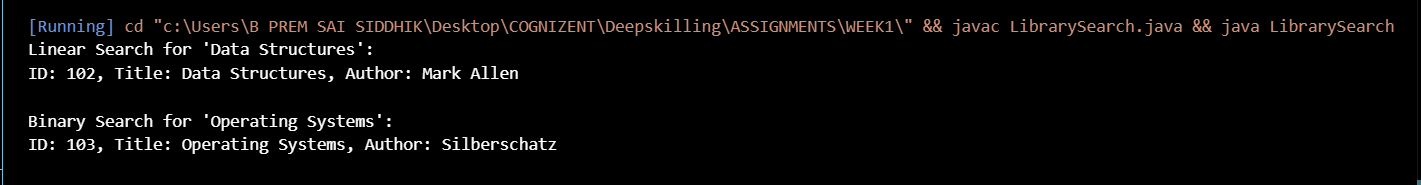
**System.out.println("Book not found.");**

**}**

**}**

**}**

**Output :**

****

**Time Complexity Analysis:**

* **Linear Search** has a time complexity of O(n). It’s useful when the list is small or unsorted.
* **Binary Search** has a time complexity of O(log n), but it only works if the list is sorted. It’s better for large datasets.

**When to use which one ?**

Linear search is suitable for small or dynamic lists, and binary search for large, sorted data when speed is important.