## 

## **WEEK-1**

**Module 1 - Data Structures and Algorithms HandsOn**

**NAME : SIDHARTH K**

**SUPERSET : 6430194**

**Exercise 1: Inventory Management System**

Thousands of products are handled in warehouses, where effective search, update, and deletion are essential. Slow performance and bottlenecks can result from poor data structures, particularly as the inventory grows.

**Suitable Data Structures:**

* Array List: Good for sequential storage but search and update are slow.

Time Complexity: **O(n)**

* HashMap: Used for fast access by key, ideal for lookups, updates, and deletions.

Time Complexity: **O(n)**

* TreeMap: Maintains sorted order by key. Useful when sorted traversal or range-based search is required.
* Time Complexity: **O (log n)**

**Code:**

**InventoryManagementSystem.java:**

import java.util.HashMap;

import java.util.Map;

class Item {

    int itemCode;

    String itemName;

    int stockCount;

    double unitCost;

    public Item(int itemCode, String itemName, int stockCount, double unitCost) {

        this.itemCode = itemCode;

        this.itemName = itemName;

        this.stockCount = stockCount;

        this.unitCost = unitCost;

    }

    public void printItemDetails() {

        System.out.println("Item Code: " + itemCode);

        System.out.println("Item Name: " + itemName);

        System.out.println("Stock Count: " + stockCount);

        System.out.println("Unit Cost: ₹" + unitCost);

        System.out.println();

    }

}

class StockManager {

    private Map<Integer, Item> itemRecords = new HashMap<>();

    public void insertItem(Item item) {

        itemRecords.put(item.itemCode, item);

        System.out.println("Item successfully inserted.");

    }

    public void modifyItem(int code, String name, int stock, double cost) {

        if (itemRecords.containsKey(code)) {

            Item i = itemRecords.get(code);

            i.itemName = name;

            i.stockCount = stock;

            i.unitCost = cost;

            System.out.println("Item updated successfully.");

        } else {

            System.out.println("Item with given code not found.");

        }

    }

    public void removeItem(int code) {

        if (itemRecords.remove(code) != null) {

            System.out.println("Item removed from stock.");

        } else {

            System.out.println("Item not found in records.");

        }

    }

    public void showAllItems() {

        for (Item i : itemRecords.values()) {

            i.printItemDetails();

        }

    }

}

class InventoryManagementSystem {

    public static void main(String[] args) {

        StockManager manager = new StockManager();

        manager.insertItem(new Item(101, "Smartphone", 25, 19999.99));

        manager.insertItem(new Item(102, "Wireless Charger", 100, 1499.50));

        manager.showAllItems();

        manager.modifyItem(101, "5G Smartphone", 20, 24999.75);

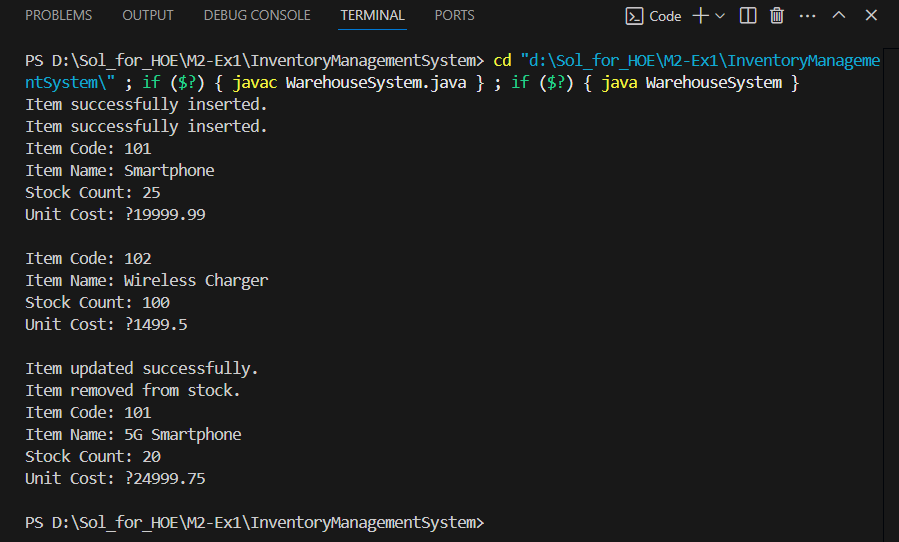
        manager.removeItem(102);

        manager.showAllItems();

    }

}

**Output:**

****

**Time Complexity Analysis:**

Add Product – O(1) Insert into HashMap by key.

Update Product – O(1) Direct access and update using key.

Delete Product – O(1) Remove by key from HashMap.

Display All – O(n) Iterates through all product values

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

**Big O Notation:**

Mathematical representation of an algorithm's complexity or performance, particularly how its memory or runtime grows with increasing input size.

**Search operations:**

There are two types of searching methods:  
     i) Linear Search  
     ii) Binary Search

**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 Item {

        int itemId;

        String itemTitle;

        String itemGroup;

        public Item(int itemId, String itemTitle, String itemGroup) {

            this.itemId = itemId;

            this.itemTitle = itemTitle;

            this.itemGroup = itemGroup;

        }

        public String showDetails() {

            return "ID : " + itemId + " Title : " + itemTitle + " Group : " + itemGroup;

        }

    }

    public static Item linearLookup(Item[] items, String keyword) {

        for (Item i : items) {

            if (i.itemTitle.equalsIgnoreCase(keyword)) {

                return i;

            }

        }

        return null;

    }

    public static Item binaryLookup(Item[] items, String keyword) {

        int start = 0;

        int end = items.length - 1;

        while (start <= end) {

            int mid = (start + end) / 2;

            int compare = items[mid].itemTitle.compareToIgnoreCase(keyword);

            if (compare == 0) {

                return items[mid];

            } else if (compare < 0) {

                start = mid + 1;

            } else {

                end = mid - 1;

            }

        }

        return null;

    }

    public static void arrangeItems(Item[] items) {

        Arrays.sort(items, Comparator.comparing(i -> i.itemTitle.toLowerCase()));

    }

    public static void main(String[] args) {

        Item[] itemArray = {

            new Item(101, "Tablet", "Gadgets"),

            new Item(102, "Notebook", "Computing"),

            new Item(103, "Sunglasses", "Wearables"),

            new Item(104, "Jacket", "Apparel")

        };

        String searchKeyword = "Jacket";

        Item result1 = linearLookup(itemArray, searchKeyword);

        System.out.println("Linear Search : " + (result1 != null ? result1.showDetails() : "Item Not Found"));

        arrangeItems(itemArray);

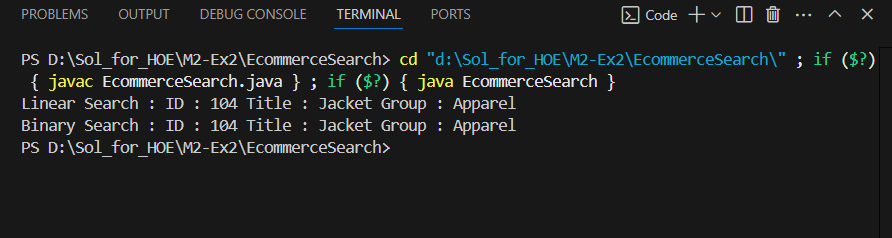
        Item result2 = binaryLookup(itemArray, searchKeyword);

        System.out.println("Binary Search : " + (result2 != null ? result2.showDetails() : "Item Not Found"));

    }

}

**Output:**



**Linear Search:**

Time Complexity – O(n)

Space Complexity – O(1)

**Binary Search:**

Time Complexity – O(log n)

Space Complexity – O(1)

Linear Search is simple and works on unsorted data. Whereas Binary Search is more efficient for large datasets but requires sorted data.

For e-commerce platforms with large datasets, Binary Search is preferred for fast performance. Binary Search is a good algorithm for the searching approach in e-commerce platforms.

**Exercise 3: Sorting Customer Orders**

**Sorting Algorithms:**

* **Bubble Sort**:  
  Repeatedly compares adjacent elements and swaps them if they are out of order.
* **Insertion Sort**:  
  Builds the sorted array by inserting each element into its correct position in the sorted part.
* **Quick Sort**:  
  A divide-and-conquer algorithm that picks a pivot, partitions the array, and recursively sorts the subarrays.
* **Merge Sort**:  
  Divides the array into halves, sorts them recursively, and merges the sorted halves.

**Code:**

class Sorting {

    static class Invoice {

        String invoiceId;

        String clientName;

        double amountDue;

        public Invoice(String invoiceId, String clientName, double amountDue) {

            this.invoiceId = invoiceId;

            this.clientName = clientName;

            this.amountDue = amountDue;

        }

        public void show() {

            System.out.println(invoiceId + "  " + clientName + "  ₹" + amountDue);

        }

    }

    public static void bubbleArrange(Invoice[] records) {

        int len = records.length;

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

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

                if (records[j].amountDue > records[j + 1].amountDue) {

                    Invoice temp = records[j];

                    records[j] = records[j + 1];

                    records[j + 1] = temp;

                }

            }

        }

    }

    public static void quickArrange(Invoice[] records, int start, int end) {

        if (start < end) {

            int pivot = separate(records, start, end);

            quickArrange(records, start, pivot - 1);

            quickArrange(records, pivot + 1, end);

        }

    }

    private static int separate(Invoice[] records, int start, int end) {

        double pivotValue = records[end].amountDue;

        int index = start - 1;

        for (int k = start; k < end; k++) {

            if (records[k].amountDue <= pivotValue) {

                index++;

                Invoice temp = records[index];

                records[index] = records[k];

                records[k] = temp;

            }

        }

        Invoice temp = records[index + 1];

        records[index + 1] = records[end];

        records[end] = temp;

        return index + 1;

    }

    public static void showAllInvoices(Invoice[] records) {

        for (Invoice inv : records) {

            inv.show();

        }

    }

    public static void main(String[] args) {

        Invoice[] invoices = {

            new Invoice("INV101", "Anya", 875.0),

            new Invoice("INV102", "Bharat", 1200.5),

            new Invoice("INV103", "Chitra", 560.75),

            new Invoice("INV104", "Dev", 2300.0),

            new Invoice("INV105", "Esha", 999.99)

        };

        System.out.println("Original Invoice List:");

        showAllInvoices(invoices);

        bubbleArrange(invoices);

        System.out.println("After Bubble Sort (by amountDue):");

        showAllInvoices(invoices);

        invoices = new Invoice[] {

            new Invoice("INV101", "Anya", 875.0),

            new Invoice("INV102", "Bharat", 1200.5),

            new Invoice("INV103", "Chitra", 560.75),

            new Invoice("INV104", "Dev", 2300.0),

            new Invoice("INV105", "Esha", 999.99)

        };

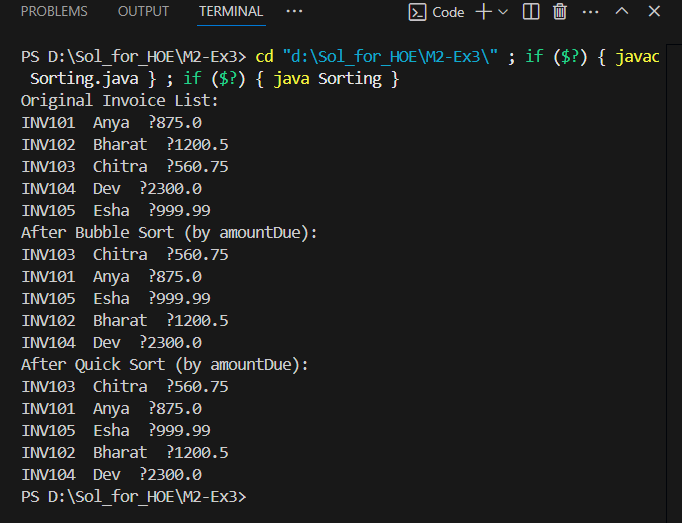
        quickArrange(invoices, 0, invoices.length - 1);

        System.out.println("After Quick Sort (by amountDue):");

        showAllInvoices(invoices);

    }

}

**Output:** 

**Time Complexity and Performance:**

**Bubble Sort:**

Time Complexity:

i)Best Case – O(n)

ii)Average Case – O(n^2)

iii)Worst Case – O(n^2)

Space Complexity – O(1)

**Quick Sort:**

Time Complexity:

i)Best Case – O(n log n)

ii)Average Case – O(n log n)

iii)Worst Case – O(n^2)

Space Complexity – O(log n)

Bubble Sort is simple but inefficient for large datasets due to its O(n²) complexity. Quick Sort is faster and more scalable because it uses a divide-and-conquer strategy. For performance-critical applications like order sorting, Quick Sort is preferred. Choosing the right sorting algorithm improves system efficiency and responsiveness.

**Exercise 4: Employee Management System**

Arrays are contiguous blocks of memory where elements are stored sequentially. Accessing an element by index is efficient and takes **O(1)** time using the formula:  
base\_address + index \* element\_size

**Advantages of Array:**

* Constant-time random access using indices (**O(1)**)
* Straightforward to implement for fixed-size data
* Improved cache locality for better performance

**Code:**

class EmployeeManagementSystem {

    static class Staff {

        int staffId;

        String fullName;

        String designation;

        double monthlyPay;

        public Staff(int staffId, String fullName, String designation, double monthlyPay) {

            this.staffId = staffId;

            this.fullName = fullName;

            this.designation = designation;

            this.monthlyPay = monthlyPay;

        }public void printDetails() {

            System.out.println(staffId + "  " + fullName + "  " + designation + "  ₹" + monthlyPay);

        }

    }static final int MAX\_STAFF = 100;

    static Staff[] staffRecords = new Staff[MAX\_STAFF];

    static int currentCount = 0;

    public static void registerStaff(Staff member) {

        if (currentCount < MAX\_STAFF) {

            staffRecords[currentCount++] = member;

            System.out.println("Staff member added: " + member.fullName);

        } else {

            System.out.println("Cannot add more staff. Maximum capacity reached.");

        }

    }

    public static void findStaffById(int id) {

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

            if (staffRecords[i].staffId == id) {

                System.out.println("Staff found:");

                staffRecords[i].printDetails();

                return;

            }

        }

        System.out.println("Staff with ID " + id + " not found.");

    }

    public static void showAllStaff() {

        if (currentCount == 0) {

            System.out.println("No staff records available.");

            return;

        }

        System.out.println("Staff Directory:");

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

            staffRecords[i].printDetails();

        }

    }

    public static void removeStaff(int id) {

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

            if (staffRecords[i].staffId == id) {

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

                    staffRecords[j] = staffRecords[j + 1];

                }

                staffRecords[--currentCount] = null;

                System.out.println("Staff with ID " + id + " has been deleted.");

                return;

            }

        }

        System.out.println("Staff with ID " + id + " not found.");

    }

    public static void main(String[] args) {

        registerStaff(new Staff(301, "Nina Verma", "Team Lead", 92000));

        registerStaff(new Staff(302, "Ravi Kumar", "Software Engineer", 68000));

        registerStaff(new Staff(303, "Leena Shah", "UI/UX Specialist", 61000));

        System.out.println();

        showAllStaff();

        System.out.println("Looking for staff with ID 302:");

        findStaffById(302);

        System.out.println();

        System.out.println("Deleting staff with ID 301:");

        removeStaff(301);

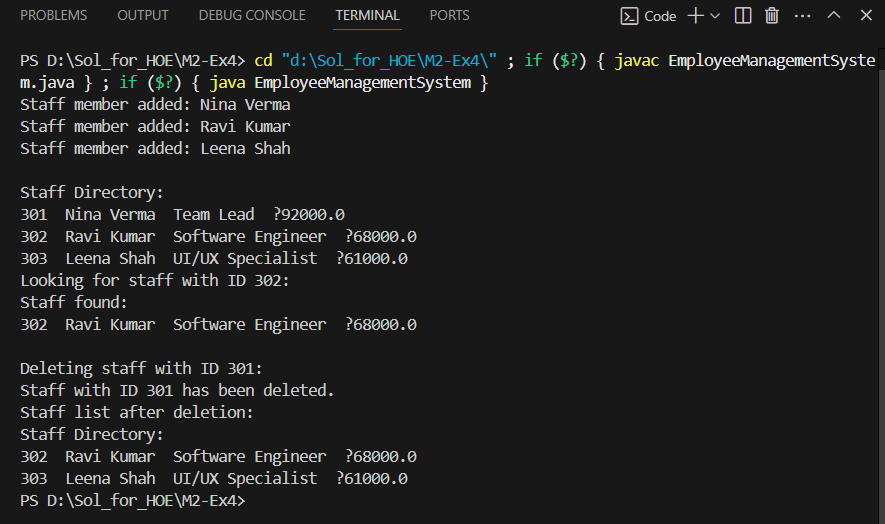
        System.out.println("Staff list after deletion:");

        showAllStaff();

    }

}

**Output:**



**Time Complexity of Operations:**

Add-O(1)

Search- O(n)

Traverse- O(n)

Delete- O(n)

**Limitations of Arrays:**

Fixed size: Cannot grow or shrink dynamically

Inefficient deletion/insertion: Requires shifting elements

Memory waste: Pre-allocated space may be unused

**When to Use Arrays:**

When the number of elements is fixed and known in advance

When fast index-based access is required

When memory consumption is predictable

**Exercise 5: Task Management System**

**Understand Linked Lists**

**Singly Linked List:**

Each node stores data and a pointer to the next node.

Allows traversal in a single direction only.

Insertion and deletion at the head are efficient.

Ideal when memory usage is a priority over bidirectional navigation.

**Doubly Linked List:**

Each node holds data, a pointer to the next node, and a pointer to the previous

node.

Enables forward and backward traversal.

More flexible but uses additional memory for the extra pointer.

**Code:**

class TaskManagementSystem {

    static class Task {

        int taskId;

        String taskName;

        String status;

        Task next;

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

            this.taskId = taskId;

            this.taskName = taskName;

            this.status = status;

            this.next = null;

        }

        public void showDetails() {

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

        }

    }

    static class TaskList {

        Task head = null;

        public void insertTask(int taskId, String taskName, String status) {

            Task newTask = new Task(taskId, taskName, status);

            if (head == null) {

                head = newTask;

            } else {

                Task current = head;

                while (current.next != null) {

                    current = current.next;

                }

                current.next = newTask;

            }

            System.out.println("New task inserted: " + taskName);

        }

        public void findTask(int taskId) {

            Task current = head;

            while (current != null) {

                if (current.taskId == taskId) {

                    System.out.println("Task found:");

                    current.showDetails();

                    return;

                }

                current = current.next;

            }

            System.out.println("No task found with ID: " + taskId);

        }

        public void listAllTasks() {

            if (head == null) {

                System.out.println("Task list is empty.");

                return;

            }

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

            Task current = head;

            while (current != null) {

                current.showDetails();

                current = current.next;

            }

        }

        public void removeTask(int taskId) {

            if (head == null) {

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

                return;

            }

            if (head.taskId == taskId) {

                head = head.next;

                System.out.println("Deleted task with ID: " + taskId);

                return;

            }

            Task previous = null;

            Task current = head;

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

                previous = current;

                current = current.next;

            }

            if (current == null) {

                System.out.println("No task found with ID: " + taskId);

            } else {

                previous.next = current.next;

                System.out.println("Deleted task with ID: " + taskId);

            }

        }

    }

    public static void main(String[] args) {

        TaskList taskManager = new TaskList();

        taskManager.insertTask(1, "UI Design", "Pending");

        taskManager.insertTask(2, "API Integration", "In Progress");

        taskManager.insertTask(3, "Database Setup", "Not Started");

        System.out.println();

        taskManager.listAllTasks();

        System.out.println();

        System.out.println("Looking up Task with ID 2:");

        taskManager.findTask(2);

        System.out.println();

        System.out.println("Removing Task with ID 1:");

        taskManager.removeTask(1);

        System.out.println();

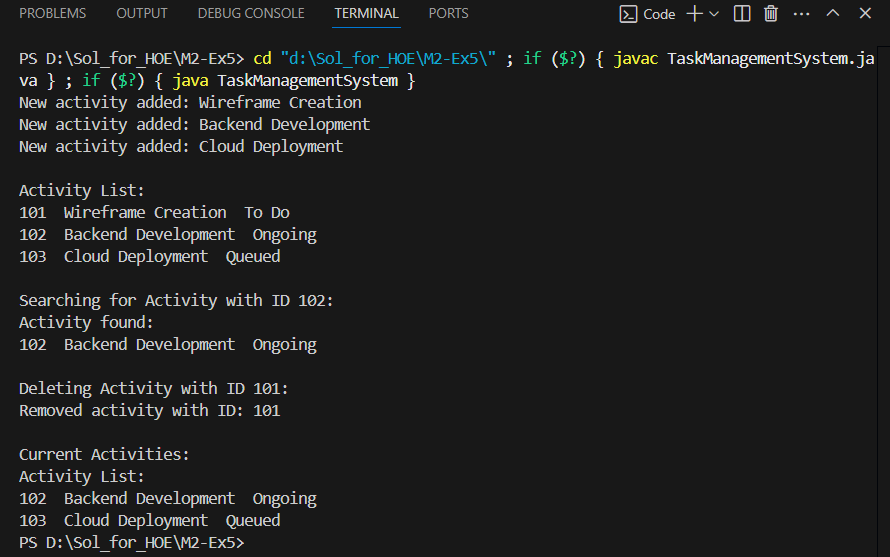
        System.out.println("Updated Task List:");

        taskManager.listAllTasks();

    }

}

**Output:**



**Exercise 6: Library Management System**

**Linear Search:**

Scans each element one by one until the target is found or the end of the list is reached.

**Binary Search:**

Works on sorted data. Repeatedly divides the list into halves to find the target more efficiently.

**Code:**

import java.util.Arrays;

import java.util.Comparator;

class LibraryManagementSystem {

    static class Volume {

        int volumeId;

        String volumeTitle;

        String writer;

        public Volume(int volumeId, String volumeTitle, String writer) {

            this.volumeId = volumeId;

            this.volumeTitle = volumeTitle;

            this.writer = writer;

        }

        public void printDetails() {

            System.out.println(volumeId + "  " + volumeTitle + "  " + writer);

        }

    }

    static Volume[] library;

    public static void searchByTitleLinear(String searchTitle) {

        boolean matchFound = false;

        for (Volume v : library) {

            if (v.volumeTitle.equalsIgnoreCase(searchTitle)) {

                System.out.println("Volume found using Linear Search:");

                v.printDetails();

                matchFound = true;

                break;

            }

        }

        if (!matchFound) {

            System.out.println("Volume not found using Linear Search.");

        }

    }

    public static void searchByTitleBinary(String searchTitle) {

        int left = 0;

        int right = library.length - 1;

        while (left <= right) {

            int middle = (left + right) / 2;

            int comparison = searchTitle.compareToIgnoreCase(library[middle].volumeTitle);

            if (comparison == 0) {

                System.out.println("Volume found using Binary Search:");

                library[middle].printDetails();

                return;

            } else if (comparison < 0) {

                right = middle - 1;

            } else {

                left = middle + 1;

            }

        }

        System.out.println("Volume not found using Binary Search.");

    }

    public static void sortVolumesByTitle() {

        Arrays.sort(library, Comparator.comparing(v -> v.volumeTitle.toLowerCase()));

    }

    public static void main(String[] args) {

        library = new Volume[] {

            new Volume(201, "Intro to Java", "Nina Patel"),

            new Volume(202, "Data Structures", "Raj Mehra"),

            new Volume(203, "Operating Systems", "Kiran Shah"),

            new Volume(204, "Network Security", "Zoya Khan")

        };

        System.out.println("Linear Search:");

        searchByTitleLinear("Data Structures");

        System.out.println();

        System.out.println("Binary Search:");

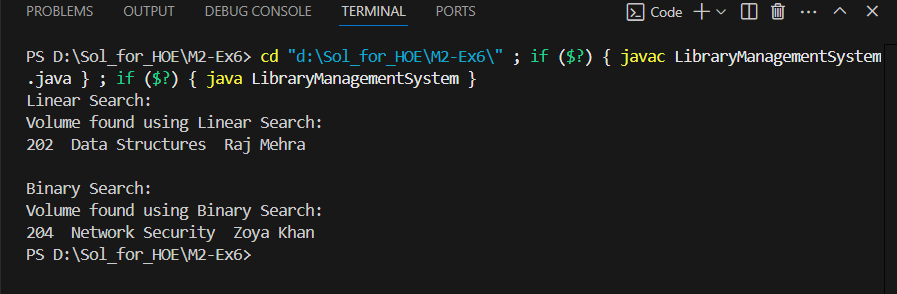
        sortVolumesByTitle();

        searchByTitleBinary("Network Security");

    }

}

**Output:**

  
**Time Complexity:**

**i)Linear Search:**

Best Case : O(1)

Average Case : O(n)

Worst Case : O(n)

**ii)Binary Search:**

Best Case : O(1)

Average Case : O(log n)

Worst Case : O(log n)

**Use Linear Search:**

When the list is unsorted or small.

When insertion/deletion happens frequently and sorting is costly.

**Use Binary Search:**

When the list is already sorted.

When you have a large static dataset.

**Exercise 7: Financial Forecasting**

**Recursion:**

Recursion is a programming method where a function calls itself to solve smaller subproblems. It is especially effective for problems with repetitive or self-similar structure, such as computing factorials, Fibonacci numbers, or forecasting compound interest over time.

**Code:**

class FinancialForecasting {

    public static double computeProjectedAmount(double baseAmount, double interestRate, int durationYears) {

        if (durationYears == 0) {

            return baseAmount;

        }

        return computeProjectedAmount(baseAmount, interestRate, durationYears - 1) \* (1 + interestRate);

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

        double initialInvestment = 15000;

        double yearlyReturnRate = 0.065;

        int timeHorizon = 6;

        double estimatedOutcome = computeProjectedAmount(initialInvestment, yearlyReturnRate, timeHorizon);

        System.out.printf("Estimated Future Value after %d years: %.2f", timeHorizon, estimatedOutcome);

    }

}

**Output:**

**A computer screen with text and numbers

AI-generated content may be incorrect.**

**Complexity:**

Time Complexity – O(n)

Space Complexity – O(n)

**Why is Optimization Needed?**

Although the recursion works here deeper or repeated recursion without caching may lead to

Redundant calculations.

Stack overflow for large n.

**Optimization Technique:**

Use memorization which store already computed values to avoid repeated calls.

For even better performance with low overhead, consider using iteration instead of recursion.