Algorithms_Data Structures

1. Inventory Management System

Understand the Problem

Why data structures and algorithms are essential:

- A **warehouse** may manage **thousands of products**, requiring fast **search, update, and retrieval**.
- Efficient **algorithms** reduce time complexity, while appropriate **data structures** ensure better **memory usage** and performance.

Suitable Data Structures:

Data Structure	Use Case	Pros	Cons
ArrayList	Simple list of products	Easy to implement	Slow lookup and delete
LinkedList	Dynamic size changes	Efficient insert/delete	Slow search
HashMap	Map productId to Product	**Fast search/update/delete** (O (1))	Slightly more memory

Best Choice: HashMap<Integer, Product> — key is productId, value is Product.

Setup

Create a Java project named: InventoryManagementSystem

Implementation

```
import java.util.HashMap;
import java.util.Scanner;

// Product class
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;
    }

    public String toString() {
        return "[" + productId + "] " + productName + " | Qty: " + quantity + " | Price: ₹" + price;
    }
}

// Inventory class
class Inventory {
    HashMap<Integer, Product> products = new HashMap<>();

    public void addProduct(Product p) {
        if (products.containsKey(p.productId)) {
            System.out.println("Product already exists.");
        } else {
            products.put(p.productId, p);
            System.out.println("Product added.");
        }
    }

    public void updateProduct(int productId, int newQty, double newPrice) {
        if (products.containsKey(productId)) {
            Product p = products.get(productId);
            p.quantity = newQty;
            p.price = newPrice;
```

```
            System.out.println("Product updated.");
        } else {
            System.out.println("Product not found.");
        }
    }

    public void deleteProduct(int productId) {
        if (products.containsKey(productId)) {
            products.remove(productId);
            System.out.println("Product deleted.");
        } else {
            System.out.println("Product not found.");
        }
    }

    public void displayInventory() {
        if (products.isEmpty()) {
            System.out.println("Inventory is empty.");
        } else {
            System.out.println("\nCurrent Inventory:");
            for (Product p : products.values()) {
                System.out.println(p);
            }
        }
    }
}

// Main class with entry point
public class InventoryManagementSystem {
```

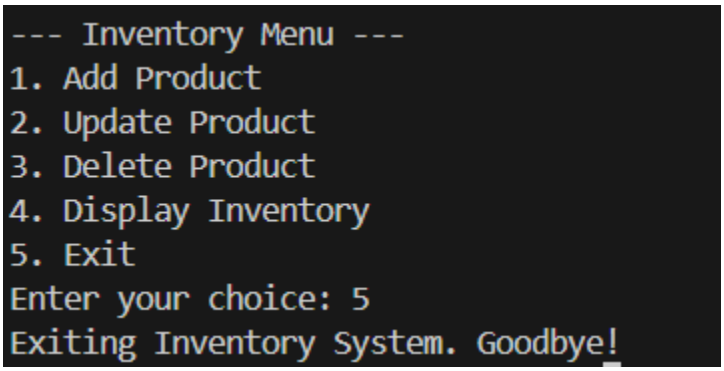
```
public static void main(String[] args) {
    Inventory inv = new Inventory();
    Scanner sc = new Scanner(System.in);
    int choice;

    do {
        System.out.println("\n--- Inventory Menu ---");
        System.out.println("1. Add Product");
        System.out.println("2. Update Product");
        System.out.println("3. Delete Product");
        System.out.println("4. Display Inventory");
        System.out.println("5. Exit");
        System.out.print("Enter your choice: ");
        choice = sc.nextInt();

        switch (choice) {
            case 1:
                System.out.print("Enter Product ID: ");
                int id = sc.nextInt();
                sc.nextLine(); // consume newline
                System.out.print("Enter Product Name: ");
                String name = sc.nextLine();
                System.out.print("Enter Quantity: ");
                int qty = sc.nextInt();
                System.out.print("Enter Price: ");
                double price = sc.nextDouble();
                inv.addProduct(new Product(id, name, qty, price));
                break;
            case 2:
```

```
                System.out.print("Enter Product ID to update: ");
                int uid = sc.nextInt();
                System.out.print("Enter New Quantity: ");
                int newQty = sc.nextInt();
                System.out.print("Enter New Price: ");
                double newPrice = sc.nextDouble();
                inv.updateProduct(uid, newQty, newPrice);
                break;
            case 3:
                System.out.print("Enter Product ID to delete: ");
                int did = sc.nextInt();
                inv.deleteProduct(did);
                break;
            case 4:
                inv.displayInventory();
                break;
            case 5:
                System.out.println("Exiting Inventory System. Goodbye!");
                break;
            default:
                System.out.println("Invalid choice.");
        }
    } while (choice != 5);

    sc.close();
  }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA> cd "c:\Codes\DIgital Nurture\DSA\" ; if ($?) { javac InventoryManagementSystem.java } ; if ($?) { java InventoryManagementSystem }

--- Inventory Menu ---
1. Add Product
2. Update Product
3. Delete Product
4. Display Inventory
5. Exit
Enter your choice: 1
Enter Product ID: 1
Enter Product Name: Mouse
Enter Quantity: 5
Enter Price: 600
Product added.

--- Inventory Menu ---
1. Add Product
2. Update Product
3. Delete Product
4. Display Inventory
5. Exit
Enter your choice: 1
Enter Product ID: 2
Enter Product Name: Keyboard
Enter Quantity: 2
Enter Price: 800
Product added.
```

```
--- Inventory Menu ---
1. Add Product
2. Update Product
3. Delete Product
4. Display Inventory
5. Exit
Enter your choice: 3
Enter Product ID to delete: 2
Product deleted.

--- Inventory Menu ---
1. Add Product
2. Update Product
3. Delete Product
4. Display Inventory
5. Exit
Enter your choice: 4

Current Inventory:
[1] Mouse | Qty: 5 | Price: Rs.600.0

--- Inventory Menu ---
1. Add Product
2. Update Product
3. Delete Product
4. Display Inventory
5. Exit
```



Analysis

Time Complexity:

Operation	**Data Structure**	**Time Complexity**
Add Product	HashMap	**O (1)** average
Update Product	HashMap	**O (1)** average
Delete Product	HashMap	**O (1)** average
Display Inventory	HashMap.values()	**O(n)**

Optimization Tips:

- Use **HashMap** for direct access via productId.
- If needing **sorted inventory**, use TreeMap instead of HashMap (but slower: O (log n)).
- Use **concurrent maps** (e.g., ConcurrentHashMap) if multithreaded access is needed.

2. E-commerce Platform Search Function

Understand Asymptotic Notation

Big O Notation:

- Big O notation describes the **upper bound** of an algorithm's runtime as input size grows.
- It helps compare **efficiency** and **scalability** of algorithms.

Complexity	**Meaning**	**Example**
O(1)	Constant Time	HashMap lookup
O(n)	Linear Time	Linear search
O(log n)	Logarithmic Time	Binary search

Complexity	Meaning	Example
$O(n^2)$	Quadratic Time	Nested loops

Search Operation Scenarios:

Case	Linear Search	Binary Search
Best Case	O(1) (first item)	O(1) (middle item)
Average Case	O(n/2) ≈ O(n)	O(log n)
Worst Case	O(n) (last/missing)	O(log n)

Setup

Create a Product class for search functionality.

```
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;
    }

    public String toString() {
        return "[" + productId + "] " + productName + " - " + category;
    }
}
```

Implementation

```
import java.util.Arrays;
import java.util.Comparator;
```

```
public class EcommerceSearch {

    // Linear Search by product name
    public static int linearSearch(Product[] products, String targetName) {
        for (int i = 0; i < products.length; i++) {
            if (products[i].productName.equalsIgnoreCase(targetName)) {
                return i;
            }
        }
        return -1;
    }

    // Binary Search by product name (must be sorted)
    public static int binarySearch(Product[] products, String targetName) {
        int low = 0, high = products.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            int compare = products[mid].productName.compareToIgnoreCase(targetName);
            if (compare == 0)
                return mid;
            else if (compare < 0)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return -1;
    }
```

```
    public static void main(String[] args) {
        Product[] products = {
            new Product(101, "Laptop", "Electronics"),
            new Product(102, "Shoes", "Fashion"),
            new Product(103, "Watch", "Accessories"),
            new Product(104, "Mobile", "Electronics"),
            new Product(105, "Bag", "Fashion")
        };

        // ➤ Linear Search (unsorted array)
        System.out.println("\n--- Linear Search ---");
        int index1 = linearSearch(products, "Watch");
        System.out.println(index1 != -1 ? "Found: " + products[index1] : "Product not found");

        // ➤ Sort array for Binary Search
        Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

        // ➤ Binary Search (sorted array)
        System.out.println("\n--- Binary Search ---");
        int index2 = binarySearch(products, "Watch");
        System.out.println(index2 != -1 ? "Found: " + products[index2] : "Product not found");
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA> cd "c:\Codes\DIgital Nurture\DSA\" ; if ($?) { javac EcommerceSearch.java } ; if ($?) { java EcommerceSearch }

--- Linear Search ---
Found: [103] Watch - Accessories

--- Binary Search ---
Found: [103] Watch - Accessories
```

Analysis

Feature	Linear Search	Binary Search
Time Complexity	O(n)	O(log n)
Requirement	Unsorted array	Sorted array
Performance	Slower with large data	Fast for large data
Use Case	Small lists, simple searches	Large sorted data sets

Which is Better for E-commerce?

- **Binary Search is more suitable for an e-commerce platform where:**
  - **Products are indexed/sorted by name or ID.**
  - **Frequent searches are expected.**
  - **Speed is critical for user experience.**

3. Sorting Customer Orders

Understand Sorting Algorithms

- **Bubble Sort**
  - **Repeatedly compares adjacent elements and swaps them if out of order.**
  - **Time Complexity:**
    - **Best: O(n) (already sorted)**
    - **Average/Worst: $O(n^2)$**
  - **Simple but inefficient for large data.**

- **Quick Sort**
  - **Divide-and-conquer strategy:**
    - **Choose a pivot, partition array into subarrays, and recursively sort.**
  - **Time Complexity:**
    - **Best/Average: O(n log n)**
    - **Worst: $O(n^2)$ (rare, with bad pivot choice)**

Setup

Create a file named:

CustomerOrderSorting.java

```
import java.util.Arrays;

// Order class
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;
    }

    public String toString() {
        return "[" + orderId + "] " + customerName + " | ₹" + totalPrice;
    }
}

public class CustomerOrderSorting {

    // Bubble Sort
    public static void bubbleSort(Order[] orders) {
        int n = orders.length;
        for (int i = 0; i < n - 1; i++) {
            boolean swapped = false;
            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;
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}

// Quick Sort
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;
}

// Display helper
public static void displayOrders(Order[] orders) {
    for (Order o : orders) {
        System.out.println(o);
    }
}

public static void main(String[] args) {
    Order[] orders1 = {
        new Order(201, "Alice", 3499.50),
        new Order(202, "Bob", 1200.00),
        new Order(203, "Charlie", 5799.00),
        new Order(204, "Daisy", 2200.75),
        new Order(205, "Eve", 999.99)
    };

    // Copy array for different sorting
    Order[] orders2 = Arrays.copyOf(orders1, orders1.length);
```

```
        System.out.println("--- Before Sorting ---");
        displayOrders(orders1);

        // Bubble Sort
        System.out.println("\n--- Bubble Sort (by Price) ---");
        bubbleSort(orders1);
        displayOrders(orders1);

        // Quick Sort
        System.out.println("\n--- Quick Sort (by Price) ---");
        quickSort(orders2, 0, orders2.length - 1);
        displayOrders(orders2);
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA> cd "c:\Codes\DIgital Nurture\DSA\" ; if ($?) { javac CustomerOrderSetting.java } ; if ($?) { java CustomerOrderSetting }
--- Before Sorting ---
[201] Alice | Rs.3499.5
[202] Bob | Rs.1200.0
[203] Charlie | Rs.5799.0
[204] Daisy | Rs.2200.75
[205] Eve | Rs.999.99

--- Bubble Sort (by Price) ---
[205] Eve | Rs.999.99
[202] Bob | Rs.1200.0
[204] Daisy | Rs.2200.75
[201] Alice | Rs.3499.5
[203] Charlie | Rs.5799.0

--- Quick Sort (by Price) ---
[205] Eve | Rs.999.99
[202] Bob | Rs.1200.0
[204] Daisy | Rs.2200.75
[201] Alice | Rs.3499.5
[203] Charlie | Rs.5799.0
```

Analysis

- **Bubble Sort**
  - **Time Complexity**: O(n^2)
  - **Space Complexity**: O(1) (in-place)
  - Inefficient for large datasets.

- **Quick Sort**
  - **Time Complexity**:
    - Best/Average: O(n log n)
    - Worst: O(n^2) (unbalanced partition)
  - **Space Complexity**: O(log n) (recursive stack)
  - Very efficient for large datasets in practice.

Why Quick Sort is Preferred:

- Much **faster on average** for large datasets.
- Divide-and-conquer makes it scalable.
- Used internally in Java's Arrays.sort() (for primitive types).

4. **Employee Management System**

Understanding Arrays in Memory

- **Array Representation**:
  Arrays are stored in **contiguous memory blocks**. Each element can be accessed directly using an **index**.
- **Advantages**:
  - Fast access with index: O(1)
  - Memory-efficient for fixed-size data
  - Easy to iterate/traverse

---

2. Setup: Employee Class

Create the file as:

EmployeeManagementSystem.java

3.Implementation

```
import java.util.Scanner;


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;
    }

    public String toString() {
        return "[" + employeeId + "] " + name + " - " + position + " - ₹" + salary;
    }
}

public class EmployeeManagementSystem {
    static final int MAX = 100;
    static Employee[] employees = new Employee[MAX];
    static int count = 0;

    // Add employee
    public static void addEmployee(Employee e) {
        if (count < MAX) {
            employees[count++] = e;
            System.out.println("Employee added.");
        } else {
            System.out.println("Employee list is full.");
        }
    }
```

```
// Search employee by ID
public static void searchEmployee(int id) {
    for (int i = 0; i < count; i++) {
        if (employees[i].employeeId == id) {
            System.out.println("Found: " + employees[i]);
            return;
        }
    }
    System.out.println("Employee not found.");
}

// Traverse all employees
public static void displayAllEmployees() {
    if (count == 0) {
        System.out.println("No employees found.");
        return;
    }
    for (int i = 0; i < count; i++) {
        System.out.println(employees[i]);
    }
}

// Delete employee by ID
public static void deleteEmployee(int id) {
    for (int i = 0; i < count; i++) {
        if (employees[i].employeeId == id) {
            // Shift left to overwrite
            for (int j = i; j < count - 1; j++) {
```

```
            employees[j] = employees[j + 1];
        }
        employees[--count] = null; // clear last
        System.out.println("Employee deleted.");
        return;
      }
    }
    System.out.println("Employee not found.");
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (true) {
        System.out.println("\n1. Add  2. Search  3. Display  4. Delete  5. Exit");
        System.out.print("Choose: ");
        int choice = sc.nextInt();
        switch (choice) {
            case 1:
                System.out.print("ID: ");
                int id = sc.nextInt();
                sc.nextLine(); // consume newline
                System.out.print("Name: ");
                String name = sc.nextLine();
                System.out.print("Position: ");
                String pos = sc.nextLine();
                System.out.print("Salary: ");
                double salary = sc.nextDouble();
                addEmployee(new Employee(id, name, pos, salary));
                break;
```

```
            case 2:
                System.out.print("Enter ID to search: ");
                searchEmployee(sc.nextInt());
                break;
            case 3:
                displayAllEmployees();
                break;
            case 4:
                System.out.print("Enter ID to delete: ");
                deleteEmployee(sc.nextInt());
                break;
            case 5:
                System.out.println("Exiting...");
                return;
            default:
                System.out.println("Invalid choice!");
        }
    }
  }
}
```

Output

```
1. Add  2. Search  3. Display  4. Delete  5. Exit
Choose: 1
ID: 1
Name: Sukhpreet Jena
Position: SDE
Salary: 3000000
Employee added.

1. Add  2. Search  3. Display  4. Delete  5. Exit
Choose: 1
ID: 2
Name: Ekriti Jena
Position: SME
Salary: 100000
Employee added.

1. Add  2. Search  3. Display  4. Delete  5. Exit
Choose: 2
Enter ID to search: 2
Found: [2] Ekriti Jena - SME - Rs.100000.0

1. Add  2. Search  3. Display  4. Delete  5. Exit
Choose: 3
[1] Sukhpreet Jena - SDE - Rs.3000000.0
[2] Ekriti Jena - SME - Rs.100000.0
```

```
1. Add  2. Search  3. Display  4. Delete  5. Exit
Choose: 4
Enter ID to delete: 2
Employee deleted.

1. Add  2. Search  3. Display  4. Delete  5. Exit
Choose: 5
Exiting...
```

Analysis

- **Time Complexities:**

Operation	Time Complexity	Explanation
Add	**O(1) (amortized)**	**Insert at the end**
Search	**O(n)**	**Linear search by ID**
Traverse	**O(n)**	**Loop through array**

Operation Time Complexity Explanation

Delete O(n) Shift elements left after deletion

- **Limitations of Arrays:**
  - **Fixed size: Must declare max size (e.g., 100)**
  - **Inefficient deletion: Requires shifting**
  - **No dynamic resizing: Need to manually expand if exceeded**

When to Use Arrays:

- **When the number of elements is known and small**
- **When fast indexed access is needed**
- **When memory overhead should be minimized**

5. Task Management System

Understand Linked Lists

- **Singly Linked List:**
  - **Each node points to the next node**
  - **One-way traversal**
- **Doubly Linked List:**
  - **Each node has next and prev references**
  - **Two-way traversal**
  - **Easier to delete or insert in both directions but consumes more memory**

Setup: Create Task class

Create a file:

TaskManagementSystem.java

Implementation

import java.util.Scanner;

```
// Node class
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 String toString() {
        return "[" + taskId + "] " + taskName + " - " + status;
    }
}

// Linked List class
public class TaskManagementSystem {
    static Task head = null;

    // Add task at end
    public 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;
    }
    System.out.println("Task added.");
}

// Search task by ID
public static void searchTask(int id) {
    Task temp = head;
    while (temp != null) {
        if (temp.taskId == id) {
            System.out.println("Found: " + temp);
            return;
        }
        temp = temp.next;
    }
    System.out.println("Task not found.");
}

// Display all tasks
public static void displayTasks() {
    if (head == null) {
        System.out.println("No tasks available.");
        return;
    }
    Task temp = head;
```

```
    while (temp != null) {
        System.out.println(temp);
        temp = temp.next;
    }
}

// Delete task by ID
public 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 temp = head;
    while (temp.next != null && temp.next.taskId != id) {
        temp = temp.next;
    }

    if (temp.next == null) {
        System.out.println("Task not found.");
    } else {
        temp.next = temp.next.next;
        System.out.println("Task deleted.");
```

```
    }
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (true) {
        System.out.println("\n1. Add Task  2. Search Task  3. Display All  4. Delete Task  5. Exit");
        System.out.print("Choose: ");
        int ch = sc.nextInt();
        switch (ch) {
            case 1:
                System.out.print("Task ID: ");
                int id = sc.nextInt();
                sc.nextLine(); // consume newline
                System.out.print("Task Name: ");
                String name = sc.nextLine();
                System.out.print("Status: ");
                String status = sc.nextLine();
                addTask(id, name, status);
                break;
            case 2:
                System.out.print("Enter ID to search: ");
                searchTask(sc.nextInt());
                break;
            case 3:
                displayTasks();
                break;
            case 4:
                System.out.print("Enter ID to delete: ");
```

```
                deleteTask(sc.nextInt());
                break;
            case 5:
                System.out.println("Exiting...");
                return;
            default:
                System.out.println("Invalid choice.");
        }
    }
}
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA> cd "c:\Codes\DIgital Nurture\DSA\" ; if ($?) { javac TaskManagementSystem.java } ; if ($?) { java TaskManagementSystem }

1. Add Task  2. Search Task  3. Display All  4. Delete Task  5. Exit
Choose: 1
Task ID: 1
Task Name: Cooking
Status: Ongoing
Task added.

1. Add Task  2. Search Task  3. Display All  4. Delete Task  5. Exit
Choose: 1
Task ID: 2
Task Name: Washing
Status: Done
Task added.

1. Add Task  2. Search Task  3. Display All  4. Delete Task  5. Exit
Choose: 3
[1] Cooking - Ongoing
[2] Washing - Done

1. Add Task  2. Search Task  3. Display All  4. Delete Task  5. Exit
Choose: 4
Enter ID to delete: 2
Task deleted.
```

```
1. Add Task  2. Search Task  3. Display All  4. Delete Task  5. Exit
Choose: 5
Exiting...
```

Analysis

Operation	Time Complexity	Explanation
Add	O(n)	Traverse to end for insertion
Search	O(n)	Linear search through nodes
Traverse	O(n)	Visit each node

Operation	Time Complexity	Explanation
Delete	O(n)	Find and unlink node

Advantages of Linked Lists Over Arrays

Array	Linked Lists
Fixed size (static)	Dynamic size
Insert/Delete = O(n)	Insert/Delete = O(1) (at head)
Memory pre-allocation needed	Grows/shrinks at runtime
Fast index access O(1)	No random access O(n)

6. Library Management System

- **Linear Search:**
  - **Check each element one by one**
  - **Works on unsorted or sorted data**
  - **Time complexity:**
    - **Best: O(1)**
    - **Worst: O(n)**
- **Binary Search:**
  - **Repeatedly divides a sorted array into halves**
  - **Much faster for large sorted datasets**
  - **Time complexity:**
    - **Best: O(1)**
    - **Worst: O(log n)**

Setup: Create Book class

Create the file:

LibraryManagementSystem.java

Implementation

```
import java.util.Arrays;
import java.util.Scanner;

class Book implements Comparable<Book> {
   int bookId;
   String title;
   String author;

   public Book(int bookId, String title, String author) {
      this.bookId = bookId;
      this.title = title.toLowerCase(); // for case-insensitive search
      this.author = author;
   }

   public String toString() {
      return "[" + bookId + "] \"" + title + "\" by " + author;
   }

   @Override
   public int compareTo(Book other) {
      return this.title.compareTo(other.title);
   }
}
```

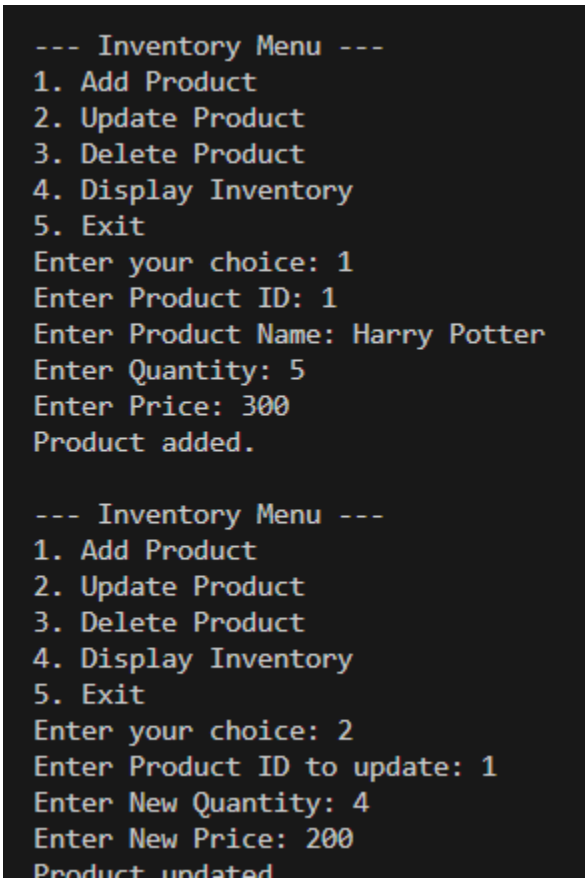
```
public class LibraryManagementSystem {
    static Book[] books = new Book[10];
    static int count = 0;

    public static void addBook(int id, String title, String author) {
        if (count < books.length) {
            books[count++] = new Book(id, title, author);
            System.out.println("Book added.");
        } else {
            System.out.println("Library is full.");
        }
    }

    public static void linearSearch(String searchTitle) {
        boolean found = false;
        for (int i = 0; i < count; i++) {
            if (books[i].title.equalsIgnoreCase(searchTitle)) {
                System.out.println("Found (Linear): " + books[i]);
                found = true;
            }
        }
        if (!found) System.out.println("Book not found (Linear Search).");
    }

    public static void binarySearch(String searchTitle) {
        Arrays.sort(books, 0, count); // sort only filled part
        int low = 0, high = count - 1;
```

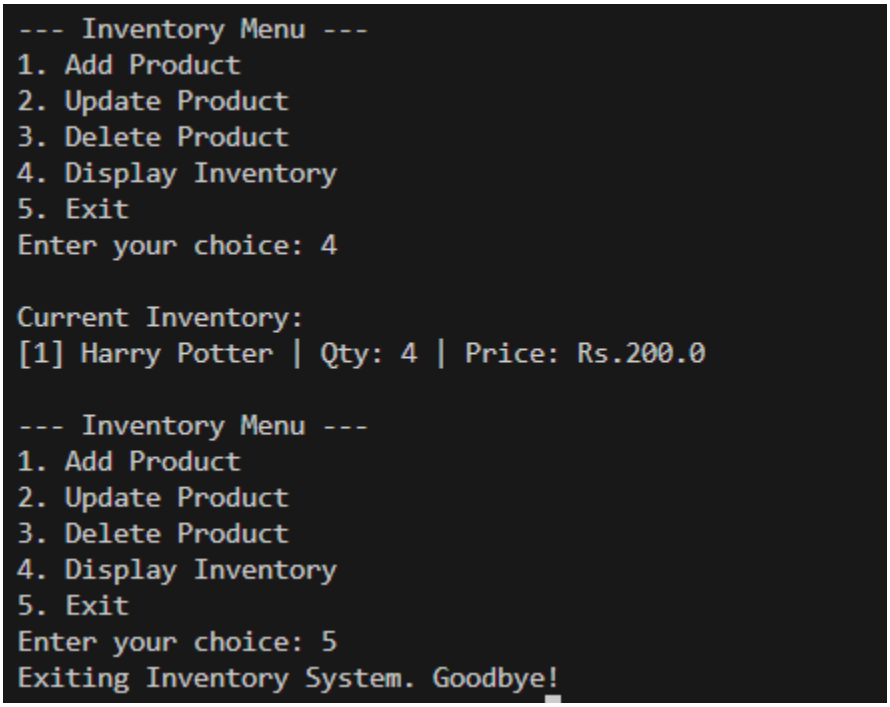
```
    searchTitle = searchTitle.toLowerCase();
    while (low <= high) {
        int mid = (low + high) / 2;
        int cmp = books[mid].title.compareTo(searchTitle);
        if (cmp == 0) {
            System.out.println("Found (Binary): " + books[mid]);
            return;
        } else if (cmp < 0) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    System.out.println("Book not found (Binary Search).");
}

public static void displayBooks() {
    if (count == 0) {
        System.out.println("No books available.");
        return;
    }
    for (int i = 0; i < count; i++) {
        System.out.println(books[i]);
    }
}

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

```
Scanner sc = new Scanner(System.in);
// Sample data
addBook(101, "Harry Potter", "J.K. Rowling");
addBook(102, "The Alchemist", "Paulo Coelho");
addBook(103, "Wings of Fire", "A.P.J. Abdul Kalam");
addBook(104, "Ikigai", "Francesc Miralles");
addBook(105, "Think and Grow Rich", "Napoleon Hill");

while (true) {
   System.out.println("\n1. Display 2. Linear Search 3. Binary Search 4. Exit");
   System.out.print("Choose: ");
   int ch = sc.nextInt();
   sc.nextLine(); // consume newline

   switch (ch) {
      case 1:
         displayBooks();
         break;
      case 2:
         System.out.print("Enter title to search (Linear): ");
         linearSearch(sc.nextLine());
         break;
      case 3:
         System.out.print("Enter title to search (Binary): ");
         binarySearch(sc.nextLine());
         break;
      case 4:
```

```
            System.out.println("Exiting...");
            return;
          default:
            System.out.println("Invalid option.");
        }
      }
    }
}
```

Output





Analysis

Operation	Linear Search	Binary Search
Best Case	O(1)	O(1)
Avg/Worst	O(n)	O(log n)
Precondition	None	Requires sorted data

When to Use What?

- **Linear Search:**
  - **For small or unsorted datasets**
  - **Easier to implement**
- **Binary Search:**
  - **For large datasets**
  - **Only when data is sorted**
  - **Much faster for search-heavy systems**

7. **Financial Forecasting**

Understand Recursive Algorithms

- **What is Recursion?**

Recursion is when a method calls **itself** to solve a smaller version of a problem.

- **Why Use Recursion?**
  - It can **simplify** code for problems like factorials, Fibonacci, and compound interest.
  - Recursive methods can be **elegant**, but may be inefficient if not optimized.

Setup

We will calculate the future value of an investment using this recursive formula:

FV=PV×(1+r)nFV = PV \times (1 + r)^nFV=PV×(1+r)n

Where:

- FV = Future Value
- PV = Present Value
- r = growth rate
- n = number of periods

Implementation

File name: FinancialForecasting.java

```
import java.util.Scanner;


public class FinancialForecasting {


   // Recursive method to calculate future value
   public static double calculateFutureValue(double presentValue, double rate, int years) {
      if (years == 0) {
```

```
        return presentValue;
    } else {
        return (1 + rate) * calculateFutureValue(presentValue, rate, years - 1);
    }
  }

  // Optimized using memoization
  public static double calculateFutureValueMemo(double presentValue, double rate, int
years, Double[] memo) {
    if (years == 0) return presentValue;
    if (memo[years] != null) return memo[years];
    memo[years] = (1 + rate) * calculateFutureValueMemo(presentValue, rate, years - 1,
memo);
    return memo[years];
  }

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter present value: ");
    double pv = sc.nextDouble();

    System.out.print("Enter annual growth rate (as decimal, e.g. 0.05 for 5%): ");
    double rate = sc.nextDouble();

    System.out.print("Enter number of years: ");
    int years = sc.nextInt();
```

```
        double futureVal = calculateFutureValue(pv, rate, years);
        System.out.printf("Predicted Future Value (Recursive): %.2f%n", futureVal);

        Double[] memo = new Double[years + 1];
        double memoVal = calculateFutureValueMemo(pv, rate, years, memo);
        System.out.printf("Predicted Future Value (Memoized): %.2f%n", memoVal);
    }
}
```

Output

```
Enter present value: 5000
Enter annual growth rate (as decimal, e.g. 0.05 for 5%): 0.11
Enter number of years: 4
Predicted Future Value (Recursive): 7590.35
Predicted Future Value (Memoized): 7590.35
```

Analysis

- **Time Complexity**

Version	Time Complexity
Basic Recursion	O(n)
Memoized	O(n) (with less work)

- **Space Complexity**
  - Both versions use **O(n)** stack space.
  - Memoized version uses additional **O(n)** for memo array.

- **When to Optimize?**
  - If n is large (e.g. 1000+), recursion may cause **stack overflow**.
  - Use **memoization** or switch to **iterative** for better performance.