**Exercise 1: Inventory Management System**

**1. Understand the Problem**

In a large warehouse, there can be thousands of products.

Efficient storage (using optimal data structures) ensures fast retrieval and updates.

Algorithms help maintain performance, especially when handling search, insert, update, or delete operations frequently.

Suitable Data Structures:

HashMap (Java):

Best for fast lookups using productId as key.

Ideal for add/update/delete/search in O(1) average time.

ArrayList / LinkedList:

Simpler but slower (especially for search/delete, which are O(n)).

**2. Setup**

**3. Implementation**

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;

    }

    @Override

    public String toString() {

        return productId + " | " + productName + " | " + quantity + " | Rs " + price;

    }

}

import java.util.HashMap;

class InventoryManager {

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

    // Add product

    public void addProduct(Product p) {

        inventory.put(p.productId, p);

    }

    // Update product

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

        if (inventory.containsKey(id)) {

            Product p = inventory.get(id);

            p.productName = name;

            p.quantity = quantity;

            p.price = price;

        }

    }

    // Delete product

    public void deleteProduct(int id) {

        inventory.remove(id);

    }

    // Display inventory

    public void displayInventory() {

        for (Product p : inventory.values()) {

            System.out.println(p);

        }

    }

}

public class Main {

    public static void main(String[] args) {

        InventoryManager manager = new InventoryManager();

        // Adding products

        manager.addProduct(new Product(101, "Laptop", 10, 75000.0));

        manager.addProduct(new Product(102, "Smartphone", 25, 30000.0));

        manager.addProduct(new Product(103, "Headphones", 50, 1500.0));

        System.out.println("Inventory after adding products:");

        manager.displayInventory();

        // Updating a product

        manager.updateProduct(102, "Smartphone", 30, 28000.0);

        System.out.println("\nInventory after updating product ID 102:");

        manager.displayInventory();

        // Deleting a product

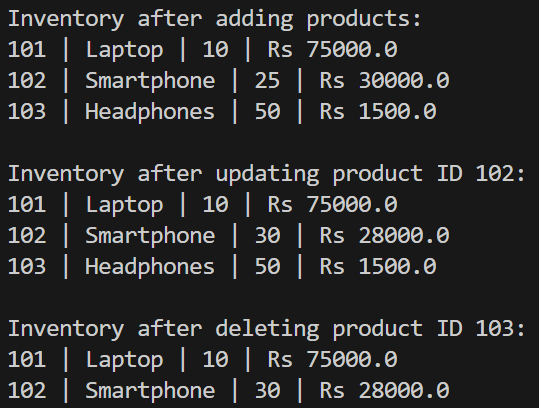
        manager.deleteProduct(103);

        System.out.println("\nInventory after deleting product ID 103:");

        manager.displayInventory();

    }

}

****

**3. Analysis**

| **Operation** | **Time Complexity (HashMap)** | **Reason** |
| --- | --- | --- |
| Add | O(1) | Direct key-based insertion |
| Update | O(1) | Direct key access/update |
| Delete | O(1) | Key-based removal |
| Search/Display | O(n) | Loop over all products |

**Optimization Ideas:**

If you require sorting (e.g., by price), consider using a **TreeMap** or sort the values manually.

Use a **database** in real applications to handle persistence and concurrent access.

Add validation and error handling in a production-level system.

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

**1. Understand Asymptotic Notation**

Big O notation is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. Big-O is a way to express the upper bound of an algorithm’s time or space complexity.

* Describes the asymptotic behavior (order of growth of time or space in terms of input size) of a function, not its exact value.
* Can be used to compare the efficiency of different algorithms or data structures.
* It provides an upper limit on the time taken by an algorithm in terms of the size of the input. We mainly consider the worst-case scenario of the algorithm to find its time complexity in terms of Big O
* It’s denoted as O(f(n)), where f(n) is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size n.

|  |  |  |
| --- | --- | --- |
| Case | Linear Search | Binary Search |
| Best | O(1) - first element is the key | O(1) - middle element is the key |
| Average | O(n/2) ≈ O(n) | O(log n) |
| Worst | O(n) - last element is the key/key absent | O(log n) |

**2. Setup**

public class Product{

    int productId;

    String productName;

    String category;

    public Product(int id, String name, String category){

        this.productId=id;

        this.productName=name;

        this.category=category;

    }

}

**3. Implementation**

public class Search{

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

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

            if (products[i].productName.equalsIgnoreCase(name)) {

                return products[i];

            }

        }

        return null;

    }

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

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

        while (low <= high) {

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

            int cmp = products[mid].productName.compareToIgnoreCase(name);

            if (cmp == 0) {

                return products[mid];

            } else if (cmp < 0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return null;

    }

}

public class ProductTest {

    public static void main(String[] args) {

        Product[] products = {

            new Product(1, "Laptop", "Electronics"),

            new Product(2, "Shoes", "Footwear"),

            new Product(3, "Keyboard", "Electronics"),

            new Product(4, "Book", "Stationery")

        };

        Product linearResult=Search.linearSearch(products,"keyboard");

        System.out.println("id: "+linearResult.productId+"\nCategory: "+linearResult.category);

        Product[] sorted\_products = {

            new Product(1, "Book", "Stationery"),

            new Product(2, "Keyboard", "Electronics"),

            new Product(3, "Laptop", "Electronics"),

            new Product(4, "Shoes", "Footwear")

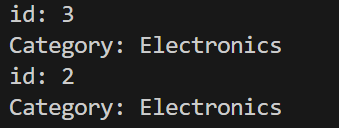
        };

        Product binaryResult=Search.binarySearch(sorted\_products,"keyboard");

        System.out.println("id: "+binaryResult.productId+"\nCategory: "+binaryResult.category);

    }

}



**4. Analysis**

Linear Search – O(n)

Binary Search – O(log n)

Binary search is better for performance, but only if the product list is sorted. For dynamic/unsorted data or very small datasets, linear search is simpler.

**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.

**Steps:**

**1. Understand Sorting Algorithms**

**Bubble Sort**

* Compares adjacent items and swaps them if they are in the wrong order.
* Repeats this process until the list is sorted.

**Time Complexity:**

* Best: O(n) → when the list is already sorted
* Average: O(n²)
* Worst: O(n²)

**Insertion Sort**

* Builds the sorted list one item at a time.
* Each new item is compared with already sorted items and placed in the correct position.

**Time Complexity:**

* Best: O(n)
* Average: O(n²)
* Worst: O(n²)

**Quick Sort**

* A **divide-and-conquer** algorithm.
* Picks a pivot and partitions the array into elements smaller and greater than the pivot.
* Recursively applies the same logic to the subarrays.

**Time Complexity:**

* Best: O(n log n)
* Average: O(n log n)
* Worst: O(n²) (rare; depends on pivot choice)

**Merge Sort**

* Also divide-and-conquer.
* Divides array into halves, sorts each recursively, and merges them.

**Time Complexity:**

* Best/Average/Worst: O(n log n)

**2. Setup**

**3. Implementation**

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;

    }

    @Override

    public String toString() {

        return orderId + " | " + customerName + " | Rs " + totalPrice;

    }

}

public class OrderSorter {

    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;

        }

    }

    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;

    }

}

public class Main {

    public static void main(String[] args) {

        Order[] orders = {

            new Order(101, "Alice", 150.75),

            new Order(102, "Bob", 99.50),

            new Order(103, "Charlie", 199.99),

            new Order(104, "Daisy", 120.00)

        };

        System.out.println("Original Orders:");

        for (Order order : orders) {

            System.out.println(order);

        }

        Order[] ordersBubble = orders.clone();

        OrderSorter.bubbleSort(ordersBubble);

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

        for (Order order : ordersBubble) {

            System.out.println(order);

        }

        Order[] ordersQuick = orders.clone();

        OrderSorter.quickSort(ordersQuick, 0, ordersQuick.length - 1);

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

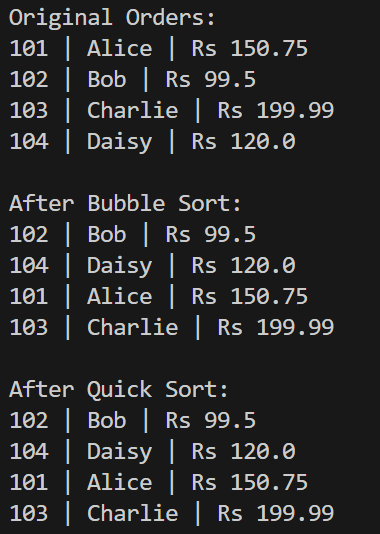
        for (Order order : ordersQuick) {

            System.out.println(order);

        }

    }

}

****

**4. Analysis**

Criteria Bubble Sort Quick Sort

Time Complexity O(n²) Avg: O(n log n), Worst: O(n²)

Space Complexity O(1) O(log n) (due to recursion)

Stability Yes No

Ease of Coding Very Easy Moderate

Real-world Use Rare Very Common

Why Quick Sort is Preferred

Bubble Sort becomes inefficient as the dataset grows.

Quick Sort, although recursive, performs very well on average and is widely used in production systems.

**Exercise 4: Employee Management System**

**1. Understand Array Representation**

* Arrays are contiguous blocks of memory.
* Each element is stored at a fixed offset from the base address:

Address of arr[i] = base + (i × size of each element)

* Arrays support random access — any element can be accessed directly using its index in O(1) time.

Advantages of Arrays

* Fast random access
* Simple and easy to use
* Memory-efficient for fixed-size datasets
* Cache-friendly due to contiguous memory layout

**2. Setup**

3. **Implementation**

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;

    }

    @Override

    public String toString() {

        return employeeId + " | " + name + " | " + position + " | $" + salary;

    }

}

public class EmployeeManager {

    private Employee[] employees;

    private int count;

    public EmployeeManager(int size) {

        employees = new Employee[size];

        count = 0;

    }

    // Add Employee

    public void addEmployee(Employee emp) {

        if (count < employees.length) {

            employees[count++] = emp;

        } else {

            System.out.println("Array full. Cannot add more employees.");

        }

    }

    // Search by employeeId

    public Employee searchEmployee(int empId) {

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

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

                return employees[i];

            }

        }

        return null;

    }

    // Traverse all employees

    public void displayAll() {

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

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

        }

    }

    // Delete employee by ID

    public void deleteEmployee(int empId) {

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

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

                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 class Main {

    public static void main(String[] args) {

        EmployeeManager manager = new EmployeeManager(5);

        manager.addEmployee(new Employee(1, "Alice", "Engineer", 70000));

        manager.addEmployee(new Employee(2, "Bob", "Manager", 85000));

        manager.addEmployee(new Employee(3, "Charlie", "HR", 60000));

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

        manager.displayAll();

        System.out.println("\nSearching for employee with ID 2:");

        Employee e = manager.searchEmployee(2);

        System.out.println(e != null ? e : "Not found");

        System.out.println("\nDeleting employee with ID 2:");

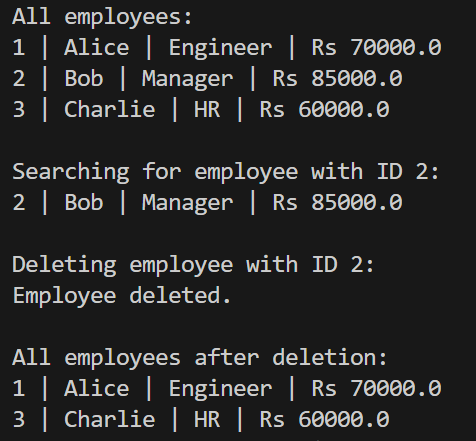
        manager.deleteEmployee(2);

        System.out.println("\nAll employees after deletion:");

        manager.displayAll();

    }

}



**4. Analysis**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Add** | **O(1)** | **Append at the end using index** |
| **Search** | **O(n)** | **Linear search through array** |
| **Traverse** | **O(n)** | **Visit each element** |
| **Delete** | **O(n)** | **Find and shift elements after deletion** |

**Limitations of Arrays**

**Drawbacks:**

* **Fixed size: Cannot grow/shrink dynamically**
* **Insertion/Deletion is costly (O(n)) due to shifting elements**
* **Wastes memory if the array is underutilized**

**When to Use Arrays:**

* **When the number of elements is known and fixed**
* **When fast random access is needed**
* **When working with small, static datasets**

**Exercise 5: Task Management System**

**1. Understand Linked Lists**

**Singly Linked List**

* **A linear data structure where each node points to the next node.**
* **Structure: Data -> Next**
* **Operations: traversal is one-way only (forward).**

**Doubly Linked List**

* **Each node points to both the next and the previous node.**
* **Structure: Prev <- Data -> Next**
* **Allows forward and backward traversal.**
* **Slightly more memory-intensive than singly linked lists.**

**2. Setup**

**3 Implementation**

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;

    }

    @Override

    public String toString() {

        return taskId + " | " + taskName + " | " + status;

    }

}

class Node {

    Task task;

    Node next;

    public Node(Task task) {

        this.task = task;

        this.next = null;

    }

}

public class TaskManager {

    private Node head;

    // Add task at the end

    public void addTask(Task task) {

        Node newNode = new Node(task);

        if (head == null) {

            head = newNode;

        } else {

            Node current = head;

            while (current.next != null)

                current = current.next;

            current.next = newNode;

        }

    }

    // Search by taskId

    public Task searchTask(int taskId) {

        Node current = head;

        while (current != null) {

            if (current.task.taskId == taskId) {

                return current.task;

            }

            current = current.next;

        }

        return null;

    }

    // Traverse and print all tasks

    public void displayTasks() {

        Node current = head;

        while (current != null) {

            System.out.println(current.task);

            current = current.next;

        }

    }

    // Delete task by taskId

    public void deleteTask(int taskId) {

        if (head == null) return;

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

            head = head.next;

            return;

        }

        Node current = head;

        while (current.next != null) {

            if (current.next.task.taskId == taskId) {

                current.next = current.next.next;

                return;

            }

            current = current.next;

        }

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

    }

}

public class Main {

    public static void main(String[] args) {

        TaskManager manager = new TaskManager();

        manager.addTask(new Task(101, "Design Database", "Pending"));

        manager.addTask(new Task(102, "Develop API", "In Progress"));

        manager.addTask(new Task(103, "Write Unit Tests", "Completed"));

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

        manager.displayTasks();

        System.out.println("\nSearch for Task ID 102:");

        Task t = manager.searchTask(102);

        System.out.println(t != null ? t : "Not Found");

        System.out.println("\nDeleting Task ID 102");

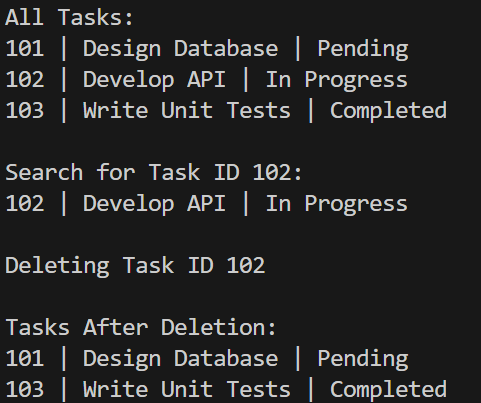
        manager.deleteTask(102);

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

        manager.displayTasks();

    }

}



**4. Analysis**

Operation Time Complexity Explanation

Add O(n) Traverse to the end

Search O(n) Linear search

Traverse O(n) One pass through the list

Delete O(n) Find and adjust pointers

| **Feature** | **Array** | **Linked List** |
| --- | --- | --- |
| Memory Allocation | Fixed, contiguous | Dynamic, scattered |
| Insertion/Deletion | Costly (shift elements) | Efficient (adjust pointers) |
| Access Time | O(1) random access | O(n) traversal required |
| Memory Efficiency | Less (no extra pointers) | More (each node has a pointer) |
| Ideal For | Static datasets | Dynamic data with frequent add/delete |

**Exercise 6: Library Management System**

**Scenario:**

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

**Steps:**

**1. Understand Search Algorithms**

**Linear Search**

* Checks each element one by one.
* **Works on unsorted data.**
* **Time Complexity**:
  + Best: O(1) (match at start)
  + Worst: O(n) (match at end or not found)

**Binary Search**

* Works only on **sorted** data.
* Divides the array in halves repeatedly.
* **Time Complexity**:
  + Best: O(1)
  + Average/Worst: O(log n)

1. **Setup**
2. **Implementation**

import java.util.List;

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;

    }

    @Override

    public String toString() {

        return bookId + " | " + title + " | " + author;

    }

    public static Book linearSearchByTitle(List<Book> books, String targetTitle) {

        for (Book book : books) {

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

            return book;

            }

        }

    return null;

    }

    public static Book binarySearchByTitle(List<Book> books, String targetTitle) {

        int left = 0, right = books.size() - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

            int cmp = books.get(mid).title.compareToIgnoreCase(targetTitle);

            if (cmp == 0)

                return books.get(mid);

            else if (cmp < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

}

import java.util.\*;

public class Main {

    public static void main(String[] args) {

        List<Book> books = new ArrayList<>();

        books.add(new Book(1, "The Alchemist", "Paulo Coelho"));

        books.add(new Book(2, "1984", "George Orwell"));

        books.add(new Book(3, "To Kill a Mockingbird", "Harper Lee"));

        books.add(new Book(4, "Pride and Prejudice", "Jane Austen"));

        books.add(new Book(5, "The Great Gatsby", "F. Scott Fitzgerald"));

        // For binary search, the list must be sorted

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

        String targetTitle = "1984";

        // Linear Search

        Book resultLinear = Book.linearSearchByTitle(books, targetTitle);

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

        System.out.println(resultLinear != null ? resultLinear : "Book not found.");

        // Binary Search

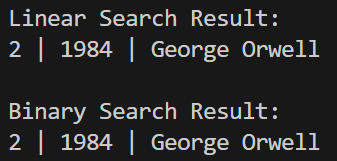
        Book resultBinary = Book.binarySearchByTitle(books, targetTitle);

        System.out.println("\nBinary Search Result:");

        System.out.println(resultBinary != null ? resultBinary : "Book not found.");

    }

}



1. **Analysis**

| **Algorithm** | **Time Complexity** |
| --- | --- |
| Linear Search | O(n) |
| Binary Search | O(log n) |

When to Use Linear Search When to Use Binary Search

Unsorted or small datasets Large datasets that are sorted

Quick implementation Performance-critical apps

Dynamic data with frequent insertions Static or rarely modified lists

**Exercise 7: Financial Forecasting**

**1. Understand Recursive Algorithms**

Recursion is a process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

**2. Setup**

**3. Implementation**

public class Forecast {

    public static double calculateFutureValue(double principal, double rate, int years){

        if(years==0)

            return principal;

        return calculateFutureValue(principal\*(1+rate), rate, years-1);

    }

    public static void main(String[] args) {

        double principal=10000.00;

        double rate=0.1;

        int years=5;

        double futureValue=calculateFutureValue(principal, rate, years);

        System.out.println("Future value = "+futureValue);

    }

}



**4. Analysis**

Time Complexity: O(n) - because for each year, we perform one recursive call.

Recursion can lead to stack overflow if n is large. Use tail recursion or iterative approach or memoization for optimized results.