## **ALGORITHMS\_DATA STRUCTURES**

## **Exercise 1: Inventory Management System**

### **Step 1: Understand the Problem**

#### **Why are Data Structures and Algorithms essential in handling large inventories?**

Efficient data structures and algorithms are vital because:

* **Scalability:** Warehouses can have thousands of products; naive storage like arrays can make searching/updating very slow.
* **Performance:** Operations like searching, updating, or deleting must be fast to ensure smooth functioning.
* **Memory Management:** Proper data structures help utilize memory efficiently, avoiding redundancy.
* **Real-Time Processing:** In a dynamic environment like a warehouse, real-time stock updates are crucial.

#### **Suitable Data Structures:**

| **Data Structure** | **Use Case** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **ArrayList** | Small inventories | Easy to implement | Slow for searching (O(n)) |
| **HashMap** (Map<productId, Product>) | Large inventories | Fast access, update, delete (O(1) avg) | More memory overhead |
| **TreeMap** (SortedMap) | When ordered data is needed | Maintains sorted keys | Slower (O(log n)) |
| **LinkedList** | When frequent insertions/deletions are needed in sequence | Dynamic size | Searching is slow (O(n)) |

**Best choice:** HashMap for productId-based access and manipulation.

### **Step 2: Setup**

You can create a Java project using an IDE like IntelliJ, Eclipse, or in VS Code.

**Folder Structure Example:**

css

InventorySystem/

│

├── src/

│ └── InventoryManager.java

│ └── Product.java

### **Step 3: Implementation**

#### **Define Product class:**

public class Product {

int productId;

String productName;

int quantity;

double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

@Override

public String toString() {

return productId + " - " + productName + " | Qty: " + quantity + " | Price: $" + price;

}

}

#### **Inventory Manager using HashMap:**

import java.util.HashMap;

public class InventoryManager {

private HashMap<Integer, Product> inventory;

public InventoryManager() {

inventory = new HashMap<>();

}

// Add product

public void addProduct(Product p) {

inventory.put(p.productId, p);

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

}

// Update product

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

if (inventory.containsKey(productId)) {

Product p = inventory.get(productId);

p.productName = name;

p.quantity = quantity;

p.price = price;

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

} else {

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

}

}

// Delete product

public void deleteProduct(int productId) {

if (inventory.containsKey(productId)) {

inventory.remove(productId);

System.out.println("Product deleted.");

} else {

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

}

}

// View all products

public void displayInventory() {

System.out.println("\nCurrent Inventory:");

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

System.out.println(p);

}

}

}

#### **Main Method Example:**

public class Main {

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

manager.addProduct(new Product(101, "Mouse", 50, 299.99));

manager.addProduct(new Product(102, "Keyboard", 30, 499.99));

manager.displayInventory();

manager.updateProduct(101, "Wireless Mouse", 40, 399.99);

manager.displayInventory();

manager.deleteProduct(102);

manager.displayInventory();

}

}

### **Step 4: Analysis**

#### **Time Complexity:**

| **Operation** | **Data Structure** | **Time Complexity** |
| --- | --- | --- |
| **Add** | HashMap | O(1) average |
| **Update** | HashMap | O(1) average |
| **Delete** | HashMap | O(1) average |
| **Search/View** | HashMap | O(1) average |

In worst case (very rare, with poor hash function), HashMap can degrade to O(n), but average case is O(1).

### **How to Optimize:**

1. **Use better hashing functions:** Reduces collision in HashMap.
2. **Validation checks:** Prevent duplicate product IDs during insert.
3. **Lazy updates:** Avoid full object replacement; update only changed fields.
4. **Thread Safety (for concurrent use):**
   * Use ConcurrentHashMap instead of HashMap.
5. **Indexing by multiple keys:** Maintain secondary structures if filtering by name/category is needed.

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

### **Step 1: Understand Asymptotic Notation**

#### **What is Big O Notation?**

**Big O Notation** describes the upper bound (worst-case) runtime of an algorithm in terms of input size n. It helps:

* Evaluate **efficiency** regardless of hardware.
* Understand **scalability** as data grows.
* Compare algorithms for **performance trade-offs**.

**Common Big O examples:**

* **O(1):** Constant time (e.g., accessing an array index)
* **O(log n):** Binary search
* **O(n):** Linear search
* **O(n log n):** Merge sort
* **O(n²):** Nested loops (e.g., bubble sort)

#### **Best, Average, and Worst-Case for Search Operations**

| **Case** | **Linear Search (Unsorted)** | **Binary Search (Sorted)** |
| --- | --- | --- |
| **Best** | O(1) (match at first) | O(1) (match at mid) |
| **Average** | O(n/2) ≈ O(n) | O(log n) |
| **Worst** | O(n) (last/no match) | O(log n) |

### **Step 2: Setup**

#### **Define Product Class**

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;

}

@Override

public String toString() {

return productId + " - " + productName + " [" + category + "]";

}

}

### **Step 3: Implementation**

#### **Linear Search (unsorted array)**

public class SearchUtil {

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

for (Product p : products) {

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

return p;

}

}

return null;

}

}

#### **Binary Search (sorted array by productName)**

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtil {

// Binary search assumes the array is sorted by productName

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

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) return products[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

// Optional: Sort products by name

public static void sortProductsByName(Product[] products) {

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

}

}

### **Example Usage**

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(101, "Shoes", "Fashion"),

new Product(102, "Laptop", "Electronics"),

new Product(103, "Camera", "Electronics"),

new Product(104, "T-shirt", "Fashion")

};

// Linear search

Product result1 = SearchUtil.linearSearch(products, "Laptop");

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

// Sort for binary search

SearchUtil.sortProductsByName(products);

Product result2 = SearchUtil.binarySearch(products, "Laptop");

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

}

}

**Step 4: Analysis**

#### **Time Complexity Comparison**

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

#### **Which is Better for E-Commerce?**

* **Binary Search** is more suitable for:  
  + Large product catalogs
  + Frequent search operations
  + When sorting once is acceptable
* **Linear Search** may be used:  
  + For very small datasets
  + When data is unsorted and sorting is too expensive for real-time

### **Final Recommendation:**

For an **e-commerce platform**, where:

* Product data is large,
* Searches are frequent,
* Speed is critical,

**Binary Search** is preferred (with sorted data).

To further improve, you can use:

* **Tries** for auto-suggestions,
* **HashMaps** for constant-time lookup,
* **Search indexing (e.g., Elasticsearch)** for large-scale enterprise-level systems.

## **Exercise 3: Sorting Customer Orders**

### **Step 1: Understand Sorting Algorithms**

#### **Sorting Algorithms Overview**

| **Algorithm** | **Description** | **Best Case** | **Average Case** | **Worst Case** | **Space** | **Stability** |
| --- | --- | --- | --- | --- | --- | --- |
| **Bubble Sort** | Repeatedly compares adjacent elements and swaps them | O(n) | O(n²) | O(n²) | O(1) | Stable |
| **Insertion Sort** | Builds the sorted array one item at a time | O(n) | O(n²) | O(n²) | O(1) | Stable |
| **Quick Sort** | Uses divide-and-conquer with a pivot to partition | O(n log n) | O(n log n) | O(n²)\* | O(log n) | Not stable |
| **Merge Sort** | Divides the array, sorts each half, and merges | O(n log n) | O(n log n) | O(n log n) | O(n) | Stable |

\*Worst case in Quick Sort occurs when pivot is poorly chosen (e.g., smallest or largest element repeatedly).

### **Step 2: Setup**

#### **Define the Order Class**

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 + " - ₹" + totalPrice;

}

}

### **Step 3: Implementation**

#### **Bubble Sort Implementation**

public class SortUtil {

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) { // Descending

Order temp = orders[j];

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

orders[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break; // Optimization: stop if no swaps

}

}

}

#### **Quick Sort Implementation**

public class SortUtil {

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pivotIndex = partition(orders, low, high);

quickSort(orders, low, pivotIndex - 1);

quickSort(orders, pivotIndex + 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) { // Descending

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;

}

}

#### **Example Usage**

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(201, "Aarav", 4500),

new Order(202, "Neha", 8200),

new Order(203, "Zoya", 3100),

new Order(204, "Raj", 6900)

};

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

for (Order o : orders) System.out.println(o);

// Bubble Sort

SortUtil.bubbleSort(orders); // or SortUtil.quickSort(orders, 0, orders.length - 1);

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

for (Order o : orders) System.out.println(o);

}

}

### **Step 4: Analysis**

#### **Time Complexity Comparison**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Bubble Sort** | O(n) | O(n²) | O(n²) |
| **Quick Sort** | O(n log n) | O(n log n) | O(n²) |

#### **Why is Quick Sort Preferred?**

* **Faster on average:** Due to divide-and-conquer approach.
* **Less swapping:** More efficient than repeatedly swapping adjacent elements.
* **Better suited for large datasets:** Bubble Sort becomes impractically slow as n grows.
* **Quick Sort’s O(n log n)** performance is ideal in most real-world scenarios.

### **Summary:**

| **Metric** | **Bubble Sort** | **Quick Sort** |
| --- | --- | --- |
| Speed | Slow (O(n²)) | Fast (O(n log n)) |
| Use case | Small arrays, easy to implement | Large datasets |
| Space | O(1) | O(log n) |
| Stability | Yes | No (unless modified) |

## **Exercise 4: Employee Management System**

### **Step 1: Understand Array Representation**

#### **How Are Arrays Represented in Memory?**

* **Contiguous Block:** Arrays are stored in **contiguous memory locations**, meaning each element is placed one after another in memory.
* **Indexing:** Access to any element is done using an **index** (arr[i]), and it is done in constant time (O(1)).
* **Fixed Size:** Arrays are **static in size**, which means the size must be known at compile time or initialized before use.

#### **Advantages of Arrays**

* **Fast access:** O(1) for accessing elements using an index.
* **Simple to implement:** Especially when number of elements is known.
* **Cache friendly:** Due to contiguous memory, improves performance in iteration.

### **Step 2: Setup**

#### **Define the Employee Class**

public class Employee {

int employeeId;

String name;

String position;

double salary;

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

this.employeeId = id;

this.name = name;

this.position = position;

this.salary = salary;

}

@Override

public String toString() {

return employeeId + " - " + name + " | " + position + " | ₹" + salary;

}

}

### **Step 3: Implementation**

#### **Employee Management Using Array**

public class EmployeeManager {

private Employee[] employees;

private int size;

public EmployeeManager(int capacity) {

employees = new Employee[capacity];

size = 0;

}

// Add an employee

public void addEmployee(Employee e) {

if (size < employees.length) {

employees[size++] = e;

System.out.println("Employee added: " + e);

} else {

System.out.println("Employee list is full!");

}

}

// Search an employee by ID

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

// Traverse all employees

public void displayAll() {

System.out.println("\nEmployee Records:");

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

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

}

}

// Delete an employee by ID

public void deleteEmployee(int id) {

boolean found = false;

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

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

found = true;

// Shift elements left

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

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

}

employees[--size] = null;

System.out.println("Employee with ID " + id + " deleted.");

break;

}

}

if (!found) System.out.println("Employee not found.");

}

}

### **Main Method Example**

public class Main {

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager(5);

manager.addEmployee(new Employee(101, "Aryan", "Manager", 50000));

manager.addEmployee(new Employee(102, "Sneha", "Developer", 45000));

manager.addEmployee(new Employee(103, "Rahul", "Tester", 30000));

manager.displayAll();

Employee found = manager.searchEmployee(102);

System.out.println("\nSearch Result: " + (found != null ? found : "Not Found"));

manager.deleteEmployee(102);

manager.displayAll();

}

}

### 

### **Step 4: Analysis**

#### **Time Complexity**

| **Operation** | **Time Complexity** |
| --- | --- |
| **Add** | O(1) (if space is available) |
| **Search** | O(n) (linear search) |
| **Traverse** | O(n) |
| **Delete** | O(n) (due to shifting) |

### **Limitations of Arrays**

| **Limitation** | **Explanation** |
| --- | --- |
| **Fixed Size** | You must declare the size in advance. Cannot grow dynamically. |
| **Costly Insertion/Deletion** | Requires shifting elements (O(n) time). |
| **Inefficient Memory Use** | Unused elements waste memory. |

### **When to Use Arrays**

Use arrays when:

* You **know the number of elements** in advance.
* You need **fast indexed access** (random access).
* You need to **traverse** data frequently.

### **Alternatives**

| **Data Structure** | **Why It's Better** |
| --- | --- |
| **ArrayList** | Dynamic resizing |
| **LinkedList** | Efficient insertion/deletion |
| **HashMap** | Fast search by key (like employee ID) |

## **Exercise 6: Library Management System**

### **Step 1: Understand Search Algorithms**

#### **Linear Search**

* **How it works:** Scans each element one by one until a match is found or the list ends.
* **Best Case:** O(1) — if the element is at the beginning.
* **Average/Worst Case:** O(n) — in the middle or not present.
* **Use case:** Unsorted or small lists.

#### **Binary Search**

* **How it works:** Repeatedly divides a sorted list into halves, comparing the middle element to the target.
* **Best Case:** O(1) — if the middle element is the target.
* **Average/Worst Case:** O(log n) — halves the search space each step.
* **Use case:** Sorted data; better for large datasets.

### **Step 2: Setup**

#### **Define the Book Class**

public class Book {

int bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

@Override

public String toString() {

return bookId + " - \"" + title + "\" by " + author;

}

}

### **Step 3: Implementation**

#### **Linear Search (by title)**

public class SearchUtil {

public static Book linearSearch(Book[] books, String targetTitle) {

for (Book b : books) {

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

return b;

}

}

return null;

}

}

#### 

#### **Binary Search (by title — sorted list required)**

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtil {

// Sort the array before binary search

public static void sortBooksByTitle(Book[] books) {

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

}

public static Book binarySearch(Book[] books, String targetTitle) {

int left = 0, right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) return books[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

}

### **Example Usage**

public class Main {

public static void main(String[] args) {

Book[] books = {

new Book(1, "The Alchemist", "Paulo Coelho"),

new Book(2, "Atomic Habits", "James Clear"),

new Book(3, "Ikigai", "Francesc Miralles"),

new Book(4, "1984", "George Orwell")

};

// Linear Search

Book result1 = SearchUtil.linearSearch(books, "Ikigai");

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

// Binary Search (requires sorted list)

SearchUtil.sortBooksByTitle(books);

Book result2 = SearchUtil.binarySearch(books, "Ikigai");

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

}

}

### **Step 4: Analysis**

#### **Time Complexity Comparison**

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

### **When to Use Which?**

| **Criteria** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Data is sorted?** | ❌ Not required | Required |
| **Data size is small?** | Suitable | Can be used, but overkill |
| **Data size is large?** | ❌ Slow | Very efficient |
| **Real-time sorting costly?** | Use Linear | ❌ Avoid unless sorted |

## **Exercise 7: Financial Forecasting**

### **Step 1: Understand Recursive Algorithms**

#### **What is Recursion?**

**Recursion** is a technique where a function calls itself to solve smaller instances of a problem.

#### **Why Use Recursion?**

* Breaks down **complex problems** into simpler sub-problems.
* Often leads to **cleaner and simpler code**.
* Ideal for **problems with repeating patterns** (e.g., Fibonacci, growth models, tree traversal).

### **Step 2: Setup**

Let’s assume:

* initialValue: the present amount (e.g., revenue, profit)
* rate: annual growth rate in percentage
* years: number of years to forecast

We’ll implement a recursive method to calculate **future value**:

### 

### **Step 3: Implementation**

#### **Recursive Function in Java**

public class Forecasting {

// Recursive function to calculate future value

public static double forecastValue(double initialValue, double rate, int years) {

if (years == 0) {

return initialValue;

}

return forecastValue(initialValue, rate, years - 1) \* (1 + rate / 100);

}

public static void main(String[] args) {

double initial = 10000; // Initial investment

double rate = 10; // 10% growth per year

int years = 5;

double futureValue = forecastValue(initial, rate, years);

System.out.printf("Future Value after %d years: ₹%.2f%n", years, futureValue);

}

}

### **Step 4: Analysis**

#### **Time Complexity**

| **Operation** | **Complexity** |
| --- | --- |
| **Recursive Forecast** | O(n) — n is the number of years |

Because each recursive call computes one year of growth, the total number of calls is proportional to n.

### 

### **How to Optimize Recursive Solution**

#### **Problem with Plain Recursion:**

* Recomputes values unnecessarily.
* For deep recursion (e.g., large n), it may hit the stack limit.

#### **Optimizations:**

1. **Memoization** (store results of subproblems):  
   * Useful if the function has overlapping subproblems (like Fibonacci).
   * Not necessary here as each call is unique, but useful for advanced forecasting.
2. **Convert to Iterative Solution** (preferred in real-world apps):  
   * More efficient, avoids stack overflow.

public static double forecastIterative(double initialValue, double rate, int years) {

double value = initialValue;

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

value \*= (1 + rate / 100);

}

return value;

}

### **Summary**

| **Feature** | **Recursive** | **Iterative** |
| --- | --- | --- |
| Readability | High (simpler logic) | Medium |
| Performance | Slower for large n | Faster |
| Stack usage | High (may cause overflow) | Low |
| Time Complexity | O(n) | O(n) |

### **Final Output Example (for 5 years at 10%):**

yaml

Future Value after 5 years: ₹16105.10