**Exercise 1: Inventory Management System**

**Scenario:**

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

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

**Ans: Understand the Problem**  
Handling thousands (or more) of products in a warehouse requires fast lookups, inserts, updates, and removals. Naïvely scanning an array for every operation becomes prohibitively slow as the inventory grows. Well-chosen data structures (and the algorithms built atop them) ensure we can maintain interactive performance even at large scale.

* **Why data structures & algorithms matter:**
  + Minimizing response times for frequent operations (e.g. processing a customer request for availability).
  + Controlling memory usage when storing product records.
  + Enabling scalability and maintainability.
* **Suitable structures:**
  + **ArrayList** (or simple arrays) give O(1) indexed access but O(n) search/update/delete.
  + **HashMap** uses a hash table keyed by productId, yielding (average) O(1) add/update/delete.
  + **TreeMap** orders entries by key in O(log n), useful if you need sorted traversal.

**Source Code:**

import java.util.HashMap;

import java.util.Map;

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 + "]";

}

}

class Inventory {

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

public void addProduct(Product product) {

if (productMap.containsKey(product.productId)) {

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

} else {

productMap.put(product.productId, product);

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

}

}

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

if (productMap.containsKey(productId)) {

Product product = productMap.get(productId);

product.productName = name;

product.quantity = quantity;

product.price = price;

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

} else {

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

}

}

public void deleteProduct(int productId) {

if (productMap.containsKey(productId)) {

productMap.remove(productId);

System.out.println("Product with ID " + productId + " deleted.");

} else {

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

}

}

public void displayInventory() {

System.out.println("\n--- Inventory ---");

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

System.out.println(p);

}

}

}

public class Main {

public static void main(String[] args) {

Inventory inventory = new Inventory();

// Adding Products

inventory.addProduct(new Product(101, "Laptop", 10, 75000));

inventory.addProduct(new Product(102, "Mouse", 50, 500));

inventory.addProduct(new Product(103, "Keyboard", 30, 1000));

// Updating Product

inventory.updateProduct(102, "Wireless Mouse", 60, 650);

// Deleting Product

inventory.deleteProduct(103);

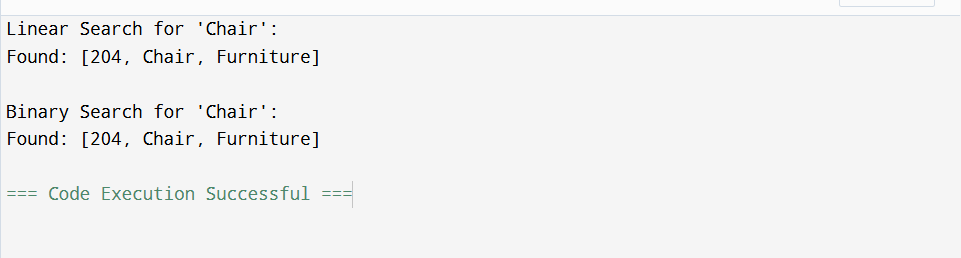
// Display Inventory

inventory.displayInventory();

}

}

**Output:**

****

**Time complexities (average):**

* Add: O(1)
* Update (lookup + modify): O(1)
* Delete: O(1)

**Optimization notes:**

* In cases of hash collisions, operations degrade to O(n) in the worst case—using a well-tuned hash function or switching to ConcurrentHashMap can help under concurrency.
* If you need sorted output, you might maintain a secondary TreeMap or sort on demand (O(n log n)).

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Ans: Understand Asymptotic Notation**  
Big-O notation characterizes how an algorithm’s running time grows with input size n:

* **O(1):** constant time (e.g. direct index lookup).
* **O(n):** linear time (e.g. scanning each element).
* **O(log n):** logarithmic time (e.g. binary search on a sorted array).

**Search scenarios:**

* **Linear Search:**
  + Best: O(1) (target at index 0)
  + Average/Worst: O(n) (on average half the list, worst at end/missing)
* **Binary Search (sorted):**
  + Best: O(1) (middle element match)
  + Average/Worst: O(log n)

**Source Code:**

import java.util.Arrays;

import java.util.Comparator;

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 + "]";

}

}

public class Main {

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

if (cmp == 0) return mid;

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

else high = mid - 1;

}

return -1;

}

public static void main(String[] args) {

Product[] products = {

new Product(201, "Laptop", "Electronics"),

new Product(202, "Shampoo", "Beauty"),

new Product(203, "Keyboard", "Electronics"),

new Product(204, "Chair", "Furniture"),

new Product(205, "T-Shirt", "Clothing")

};

// Linear Search (unsorted array)

System.out.println("Linear Search for 'Chair':");

int index1 = linearSearch(products, "Chair");

System.out.println(index1 != -1 ? "Found: " + products[index1] : "Not found");

// Sort for Binary Search

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

// Binary Search (sorted array)

System.out.println("\nBinary Search for 'Chair':");

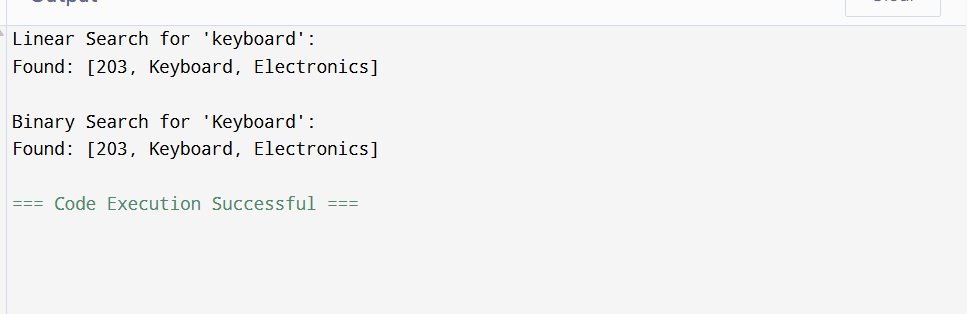
int index2 = binarySearch(products, "Chair");

System.out.println(index2 != -1 ? "Found: " + products[index2] : "Not found");

}

}

**Output:**

****

* Linear search: O(n)
* Binary search: O(log n)
* **Recommendation:** For small or unsorted lists, linear search suffices. For large, frequently-queried catalogs, maintain a sorted array (or better yet, a hash/index structure) to leverage O(log n) or even O(1) lookups.

**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:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**Ans: Understand Sorting Algorithms**

* **Bubble Sort:** compare & swap adjacent elements repeatedly → O(n²)
* **Insertion Sort:** insert each element into its correct position in a growing sorted portion → O(n²)
* **Quick Sort:** pick pivot, partition into less/greater, recurse → average O(n log n), worst O(n²)
* **Merge Sort:** divide array, sort halves, merge → O(n log n)

**Source Code:**

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 Main {

// Bubble Sort

public static void bubbleSort(Order[] orders) {

int n = orders.length;

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

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

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

// Swap

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

// 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;

}

public static void printOrders(Order[] orders) {

for (Order o : orders) {

System.out.println(o);

}

}

public static void main(String[] args) {

Order[] orders1 = {

new Order(301, "Alice", 2500.50),

new Order(302, "Bob", 1500.00),

new Order(303, "Charlie", 3500.75),

new Order(304, "David", 900.00)

};

Order[] orders2 = orders1.clone(); // for quick sort

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

printOrders(orders1);

bubbleSort(orders1);

System.out.println("\nSorted by Bubble Sort (Ascending by totalPrice):");

printOrders(orders1);

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

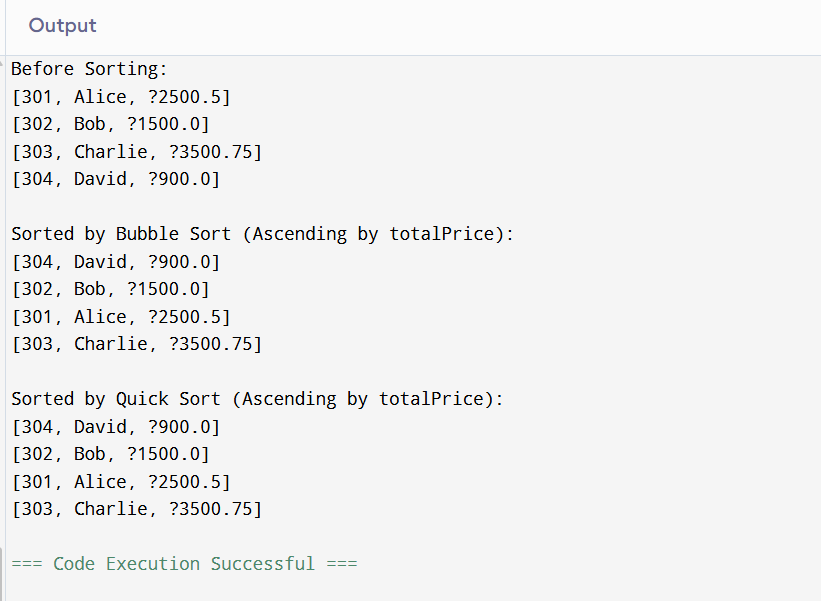
System.out.println("\nSorted by Quick Sort (Ascending by totalPrice):");

printOrders(orders2);

}

}

**Output:**

****

**Analysis**

* Bubble Sort: always O(n²), many swaps—poor for large n.
* Quick Sort: average O(n log n) with fewer swaps; worst-case O(n²) if pivot choice is poor (mitigate via randomized or median-of-three pivots).
* Conclusion: Quick Sort is preferred for large datasets due to much better average-case performance.

**Exercise 4: Employee Management System**

**Scenario:**

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

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**Ans: Understand Array Representation**  
Arrays occupy contiguous memory; each element’s address is computed via base + index × elementSize.

Advantages: constant-time indexed access (O(1)) and memory locality (cache-friendly).

**Source Code:**

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 Main {

static final int MAX = 100;

static Employee[] employees = new Employee[MAX];

static int count = 0;

// Add employee

public static void addEmployee(Employee emp) {

if (count < MAX) {

employees[count++] = emp;

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

} else {

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

}

}

// Search employee by ID

public static Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

// Traverse employee list

public static void listEmployees() {

System.out.println("\n--- Employee List ---");

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

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

}

}

// Delete employee by ID

public static void deleteEmployee(int id) {

int index = -1;

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

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

index = i;

break;

}

}

if (index != -1) {

// Shift remaining elements left

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

employees[i] = employees[i + 1];

}

employees[--count] = null;

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

} else {

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

}

}

public static void main(String[] args) {

addEmployee(new Employee(1, "Alice", "Manager", 80000));

addEmployee(new Employee(2, "Bob", "Developer", 60000));

addEmployee(new Employee(3, "Charlie", "Designer", 50000));

listEmployees();

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

Employee emp = searchEmployee(2);

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

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

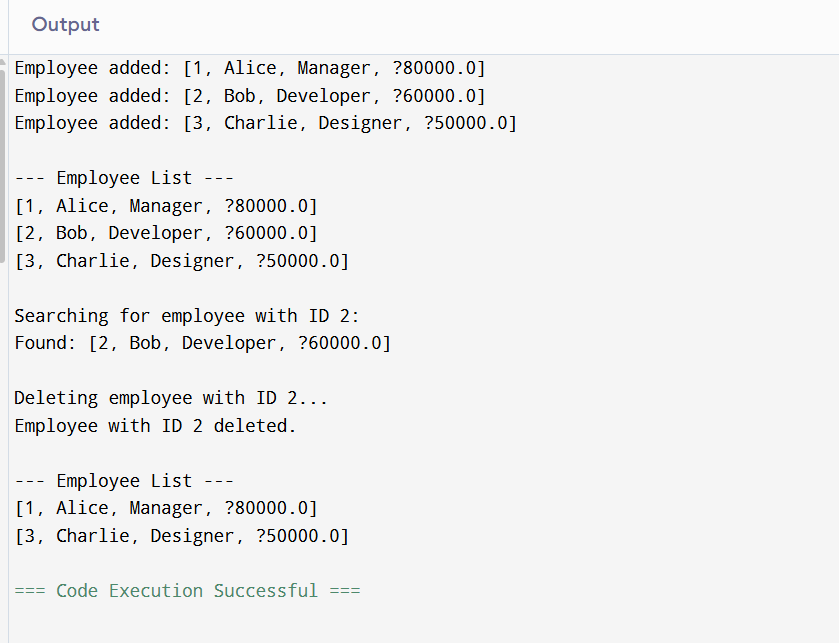
deleteEmployee(2);

listEmployees();

}

}

**Output:**

****

**Analysis**

* Add: O(1)
* Search: O(n)
* Traverse: O(n)
* Delete (find + shift): O(n)
* **Limitations:** fixed capacity, expensive deletions (shifting), no dynamic resizing. Use an ArrayList or linked structure when size varies unpredictably.

**Exercise 5: Task Management System**

**Scenario:**

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

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**Ans:** **Understand Linked Lists**

* **Singly Linked List:** each node holds data + pointer to next.
* **Doubly Linked List:** nodes have both next and prev pointers—for bidirectional traversal.

**Source Code:**

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;

}

public String toString() {

return "[" + taskId + ", " + taskName + ", " + status + "]";

}

}

// Node for singly linked list

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

class TaskManager {

private TaskNode head;

// Add task at end

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

System.out.println("Task added: " + task);

}

// Search task by ID

public Task searchTask(int id) {

TaskNode temp = head;

while (temp != null) {

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

return temp.task;

}

temp = temp.next;

}

return null;

}

// Traverse all tasks

public void listTasks() {

System.out.println("\n--- Task List ---");

TaskNode temp = head;

while (temp != null) {

System.out.println(temp.task);

temp = temp.next;

}

}

// Delete task by ID

public void deleteTask(int id) {

if (head == null) {

System.out.println("List is empty.");

return;

}

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

head = head.next;

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

return;

}

TaskNode current = head;

while (current.next != null && current.next.task.taskId != id) {

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

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

} else {

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

}

}

}

public class Main {

public static void main(String[] args) {

TaskManager tm = new TaskManager();

tm.addTask(new Task(1, "Design Module", "Pending"));

tm.addTask(new Task(2, "Write Code", "In Progress"));

tm.addTask(new Task(3, "Test Application", "Pending"));

tm.listTasks();

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

Task t = tm.searchTask(2);

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

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

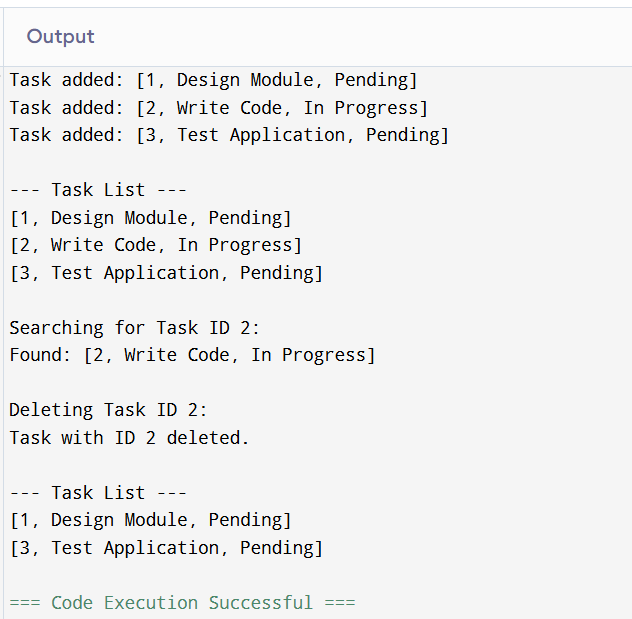
tm.deleteTask(2);

tm.listTasks();

}

}

**Output:**

****

**Analysis**

* Add (end): O(n) unless you maintain a tail pointer (then O(1)).
* Search: O(n)
* Traverse: O(n)
* Delete: O(n) to find and relink
* **Advantages vs. arrays:** dynamic size, O(1) insert/delete when pointer is known, no contiguous memory requirement.

**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:**
   * Explain linear search and binary search algorithms.
2. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

**Ans: Understand Asymptotic Notation**  
Big-O notation characterizes how an algorithm’s running time grows with input size n:

* **O(1):** constant time (e.g. direct index lookup).
* **O(n):** linear time (e.g. scanning each element).
* **O(log n):** logarithmic time (e.g. binary search on a sorted array).

**Search scenarios:**

* **Linear Search:**
  + Best: O(1) (target at index 0)
  + Average/Worst: O(n) (on average half the list, worst at end/missing)
* **Binary Search (sorted):**
  + Best: O(1) (middle element match)
  + Average/Worst: O(log n)

**Source Code:**

import java.util.Arrays;

import java.util.Comparator;

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;

}

public String toString() {

return "[" + bookId + ", " + title + ", " + author + "]";

}

}

public class Main {

// Linear Search by Title

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

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

if (books[i].title.equalsIgnoreCase(title)) {

return i;

}

}

return -1;

}

// Binary Search by Title (requires sorted array)

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

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

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) return mid;

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

else high = mid - 1;

}

return -1;

}

// Print all books

public static void listBooks(Book[] books) {

System.out.println("\n--- Book List ---");

for (Book b : books) {

System.out.println(b);

}

}

public static void main(String[] args) {

Book[] books = {

new Book(101, "The Alchemist", "Paulo Coelho"),

new Book(102, "To Kill a Mockingbird", "Harper Lee"),

new Book(103, "1984", "George Orwell"),

new Book(104, "Pride and Prejudice", "Jane Austen")

};

System.out.println("Linear Search for '1984':");

int linearIndex = linearSearch(books, "1984");

System.out.println(linearIndex != -1 ? "Found: " + books[linearIndex] : "Not found");

// Sort books for binary search

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

System.out.println("\nBinary Search for '1984':");

int binaryIndex = binarySearch(books, "1984");

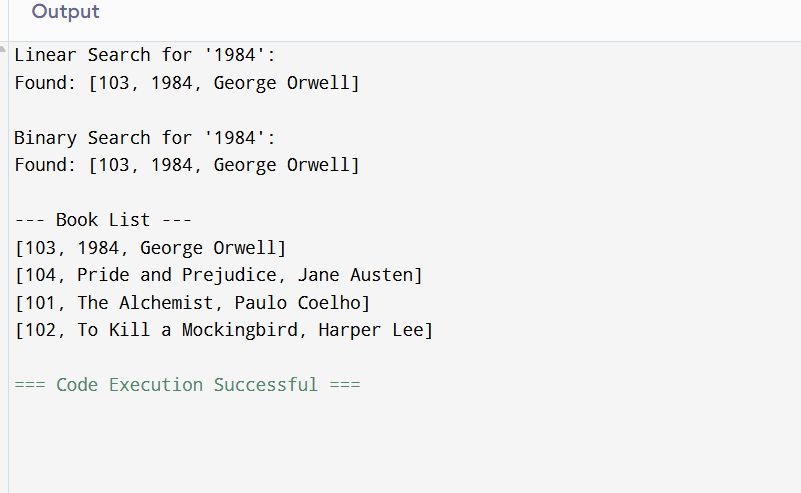
System.out.println(binaryIndex != -1 ? "Found: " + books[binaryIndex] : "Not found");

listBooks(books);

}

}

**Output:**



**Analysis**

* Linear: O(n)
* Binary: O(log n)
* **When to use which:** unsorted or small collections → linear; large, sorted lists → binary (or better, maintain a hash map keyed by title/ID for O(1) average lookups).

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Ans: Understand Recursive Algorithms**  
Recursion solves a problem by calling itself on smaller instances and combining results. A clear base case prevents infinite descent.

**Source Code:**

public class Main {

// Recursive function to calculate future value

public static double futureValueRecursive(double initialValue, double growthRate, int years) {

if (years == 0) {

return initialValue;

}

return futureValueRecursive(initialValue, growthRate, years - 1) \* (1 + growthRate);

}

// Optimized version using iteration (to avoid deep recursion)

public static double futureValueIterative(double initialValue, double growthRate, int years) {

double result = initialValue;

for (int i = 1; i <= years; i++) {

result \*= (1 + growthRate);

}

return result;

}

public static void main(String[] args) {

double initial = 10000; // Initial amount: Rs.10,000

double rate = 0.08; // 8% annual growth

int years = 5;

System.out.println("Recursive Forecast:");

double recursiveResult = futureValueRecursive(initial, rate, years);

System.out.printf("Future value after %d years: Rs.%.2f\n", years, recursiveResult);

System.out.println("\nIterative Forecast (Optimized):");

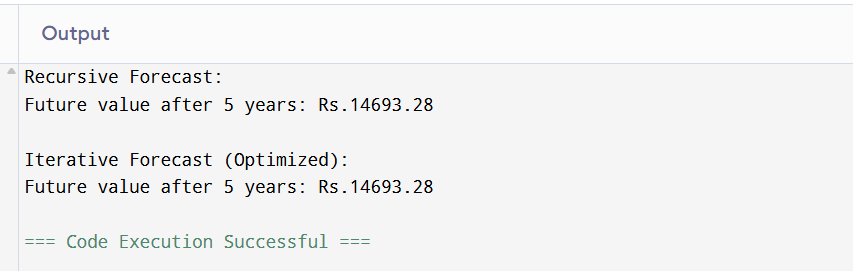
double iterativeResult = futureValueIterative(initial, rate, years);

System.out.printf("Future value after %d years: Rs.%.2f\n", years, iterativeResult);

}

}

**Output:**

****

**Analysis**

* Recursive time complexity: O(n) calls; space complexity O(n) due to call stack.
* Iterative time: O(n), space O(1).
* **Optimizations:** tail recursion (not optimized by Java JVM), memoization (not needed here since each year depends only on the prior), or direct exponentiation via P×(1+r)nP \times (1+r)^nP×(1+r)n using Math.pow, which reduces the algorithmic complexity to O(1) (though internally pow may cost more, it’s still constant relative to n).