**Exercise 1: Inventory Management System**

Importance of Data Structures and Algorithms:

* Efficient storage and fast retrieval are crucial in managing large inventories.
* Using proper data structures avoids redundancy and speeds up operations like search, insert, delete.

Suitable Data Structures:

* ArrayList: Maintains order but has slower search.
* HashMap: Offers constant time for search, insert, and delete by key (ideal for productId).

Code:

import java.util.\*;

class Product {

String id, name;

int quantity;

double price;

Product(String id, String name, int quantity, double price) {

this.id = id; this.name = name; this.quantity = quantity; this.price = price;

}

public String toString() {

return id + ": " + name + " | Qty: " + quantity + " | Rs." + price;

}

}

public class InventorySystem {

static Map<String, Product> inventory = new HashMap<>();

static Scanner sc = new Scanner(System.in);

public static void main(String[] args) {

while (true) {

System.out.println("\n1. Add 2. Update 3. Delete 4. View 5. Exit");

switch (sc.nextInt()) {

case 1 -> add();

case 2 -> update();

case 3 -> delete();

case 4 -> view();

case 5 -> { System.out.println("Exiting."); return; }

default -> System.out.println("Invalid choice.");

}

}

}

static void add() {

System.out.print("ID Name Qty Price: ");

String id = sc.next(); String name = sc.next(); int q = sc.nextInt(); double p = sc.nextDouble();

if (inventory.containsKey(id)) System.out.println("ID exists.");

else inventory.put(id, new Product(id, name, q, p));

}

static void update() {

System.out.print("ID Name Qty Price: ");

String id = sc.next(); String name = sc.next(); int q = sc.nextInt(); double p = sc.nextDouble();

Product pdt = inventory.get(id);

if (pdt != null) { pdt.name = name; pdt.quantity = q; pdt.price = p; }

else System.out.println("Not found.");

}

static void delete() {

System.out.print("Enter ID: ");

String id = sc.next();

if (inventory.remove(id) == null) System.out.println("Not found.");

}

static void view() {

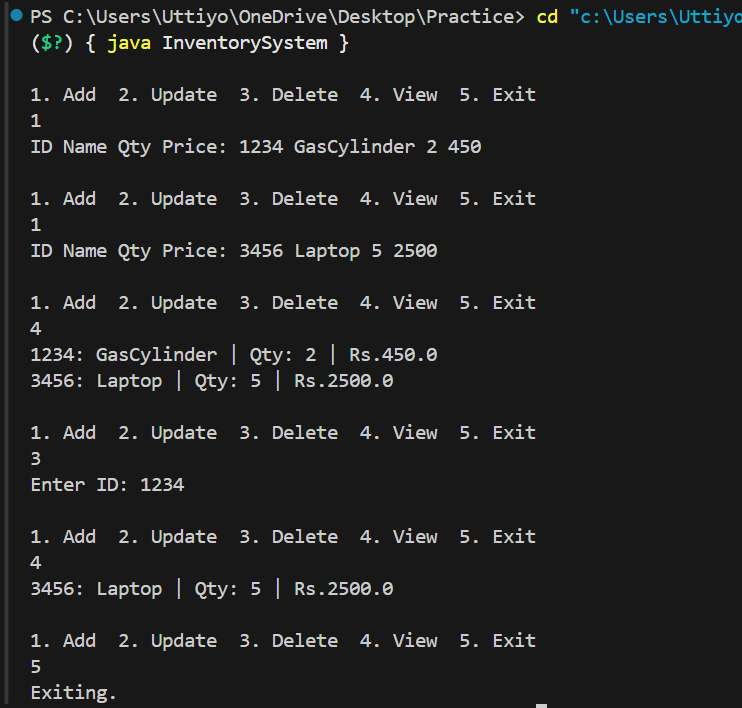
if (inventory.isEmpty()) System.out.println("Inventory empty.");

else inventory.values().forEach(System.out::println);

}

}

**Output:**



**Time Complexity**:

* Add: O(1)
* Update: O(1)
* Delete: O(1)

**Optimization**:

* Use concurrent maps for thread safety in multi-threaded systems.
* Use indexing on frequently searched attributes if backed by DB.

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

**1. Understand Asymptotic Notation**

**Big O Notation:**

Big O notation describes the upper bound of an algorithm's running time. It helps us:

* Evaluate algorithm performance based on input size.
* Compare scalability and efficiency.
* Focus on dominant terms that affect performance most as data grows.

|  |  |  |
| --- | --- | --- |
| Case | Linear Search | Binary Search |
| Best Case | O(1) (First element match) | O(1) (Middle element match) |
| Average Case | O(n) | O(log n) |
| Worst Case | O(n) | O(log n) |

Code:

public class EcommerceSearch {

    // Product class

    static 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 "Product[ID=" + productId + ", Name=" + productName + ", Category=" + category + "]";

        }

    }

    // Linear Search

    public static Product linearSearch(Product[] products, int targetId) {

        for (Product product : products) {

            if (product.productId == targetId) {

                return product;

            }

        }

        return null;

    }

    // Binary Search

    public static Product binarySearch(Product[] products, int targetId) {

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

            if (products[mid].productId == targetId) {

                return products[mid];

            } else if (products[mid].productId < targetId) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

    // Selection Sort by productId

    public static void selectionSort(Product[] products) {

        int n = products.length;

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

            int minIndex = i;

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

                if (products[j].productId < products[minIndex].productId) {

                    minIndex = j;

                }

            }

            // Swap

            Product temp = products[i];

            products[i] = products[minIndex];

            products[minIndex] = temp;

        }

    }

    // Main method

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(105, "Shampoo", "Personal Care"),

            new Product(101, "Phone", "Electronics"),

            new Product(103, "Notebook", "Stationery")

        };

        // Linear Search before sorting

        int targetId = 105;

        Product result1 = linearSearch(products, targetId);

        System.out.println("Linear Search Result: " + result1);

        // Sort the array using selection sort

        selectionSort(products);

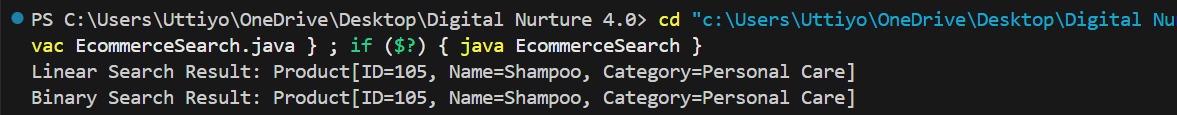
        // Binary Search after sorting

        Product result2 = binarySearch(products, targetId);

        System.out.println("Binary Search Result: " + result2);

    }

}

Output:

**Time Complexity Comparison:**

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

We can use Linear Search if:

* + Data is unsorted.
  + Dataset is small.
  + Search is infrequent.

We can use Binary Search if:

* + Data is sorted.
  + Performance is critical (large datasets).
  + Search is frequent and needs optimization.

**Exercise 3: Sorting Customer Orders**

Bubble Sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they’re in the wrong order.  
It’s easy to implement but very inefficient for large datasets (O(n²) time complexity).

Insertion Sort

Insertion Sort builds the sorted array one item at a time by comparing and inserting elements into the correct position.  
It performs well on small or nearly sorted datasets, but has O(n²) time complexity in the worst case.

Quick Sort

Quick Sort uses divide-and-conquer by selecting a pivot, partitioning the array, and sorting subarrays recursively.  
It is very fast for large datasets, with O(n log n) average time complexity, but O(n²) in the worst case.

Merge Sort

Merge Sort also uses divide-and-conquer by splitting arrays, sorting them recursively, and merging them back.  
It guarantees O(n log n) time complexity even in the worst case but uses extra space (O(n)).

Code:

import java.util.Scanner;

class Order {

String orderId;

String customerName;

double totalPrice;

public Order(String orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String toString() {

return "[" + orderId + "] " + customerName + " - ₹" + totalPrice;

}

}

public class SortOrders {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Order[] orders = {

new Order("O001", "Alice", 1500.0),

new Order("O002", "Bob", 2400.0),

new Order("O003", "Charlie", 900.0),

new Order("O004", "David", 3100.0)

};

System.out.println("Choose Sorting Algorithm:");

System.out.println("1. Bubble Sort");

System.out.println("2. Quick Sort");

System.out.print("Enter choice: ");

int choice = sc.nextInt();

System.out.println("\nBefore Sorting:");

displayOrders(orders);

if (choice == 1) {

bubbleSort(orders);

} else if (choice == 2) {

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

} else {

System.out.println("Invalid choice.");

return;

}

System.out.println("\nAfter Sorting by Total Price:");

displayOrders(orders);

sc.close();

}

static void displayOrders(Order[] orders) {

for (Order o : orders) {

System.out.println(o);

}

}

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) {

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

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);

}

}

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];

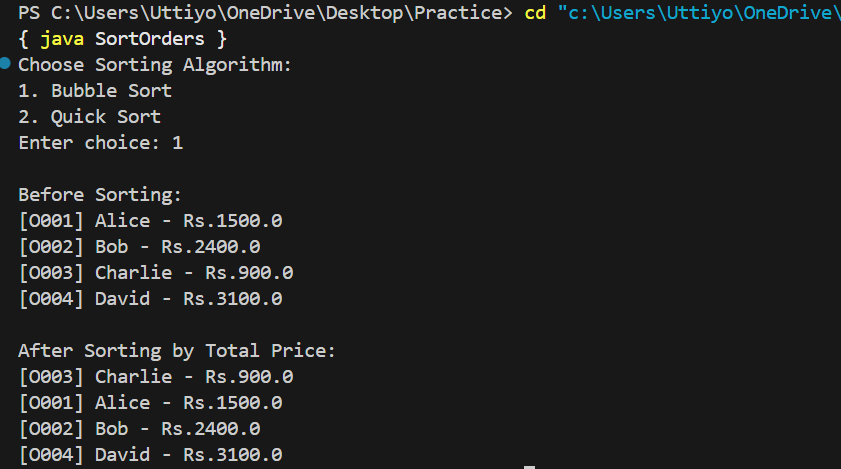
orders[high] = temp;

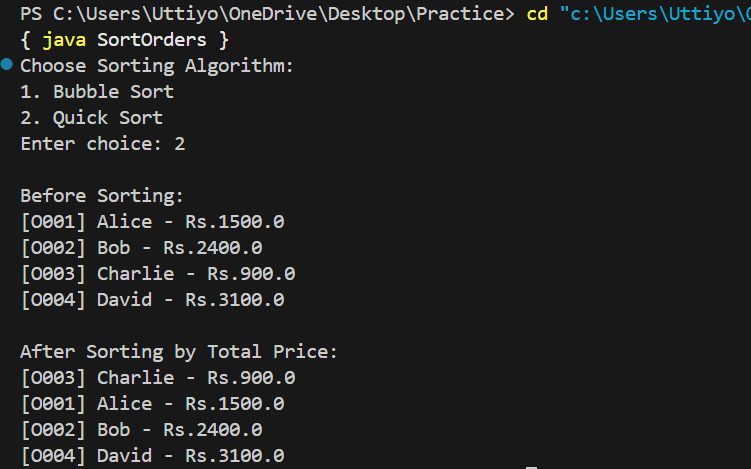
return i + 1;

}

}

**Output:**

****



Bubble Sort

* Time Complexity:
  + Best Case: O(n) when the array is already sorted
  + Average/Worst Case: O(n²) — very inefficient for large lists
* Performance: Slow due to repeated comparisons and swaps
* Use Case: Only suitable for small datasets or for educational purposes

Quick Sort

* Time Complexity:
  + Best/Average Case: O(n log n)
  + Worst Case: O(n²), but this is rare with good pivot selection
* Performance: Very efficient and widely used in practice
* Use Case: Ideal for large datasets; works well in most scenarios

**Why Quick Sort is Generally Preferred Over Bubble Sort?**

Quick Sort is preferred because:

* It handles large arrays much faster than Bubble Sort.
* Its average-case time complexity is **O(n log n)**, which is far better than **Bubble Sort's O(n²)**.
* It efficiently uses recursion and divides the problem into smaller subproblems.

**Exercise 4: Employee Management System**

Understanding Array Representation

* Arrays are a contiguous block of memory, where each element can be accessed directly by its index (O(1) access time).
* They are simple and fast for fixed-size collections, but resizing and insertion/deletion (except at the end) can be costly (O(n)).

**Code:**

import java.util.Scanner;

class Employee {

String id, name, position;

double salary;

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

this.id = id;

this.name = name;

this.position = position;

this.salary = salary;

}

public String toString() {

return "[" + id + "] " + name + " - " + position + " - ₹" + salary;

}

}

public class EmployeeManagementSystem {

static final int MAX = 100;

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

static int count = 0;

static Scanner sc = new Scanner(System.in);

public static void main(String[] args) {

while (true) {

System.out.println("\n1. Add 2. Search 3. Traverse 4. Delete 5. Exit");

int choice = sc.nextInt();

switch (choice) {

case 1 -> add();

case 2 -> search();

case 3 -> traverse();

case 4 -> delete();

case 5 -> { System.out.println("Exiting..."); return; }

default -> System.out.println("Invalid choice.");

}

}

}

static void add() {

if (count >= MAX) {

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

return;

}

System.out.print("Enter ID Name Position Salary: ");

String id = sc.next();

String name = sc.next();

String pos = sc.next();

double sal = sc.nextDouble();

employees[count++] = new Employee(id, name, pos, sal);

System.out.println("Employee added.");

}

static void search() {

System.out.print("Enter Employee ID to search: ");

String id = sc.next();

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

if (employees[i].id.equals(id)) {

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

return;

}

}

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

}

static void traverse() {

if (count == 0) {

System.out.println("No employees.");

return;

}

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

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

}

}

static void delete() {

System.out.print("Enter Employee ID to delete: ");

String id = sc.next();

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

if (employees[i].id.equals(id)) {

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

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

}

employees[--count] = null;

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

return;

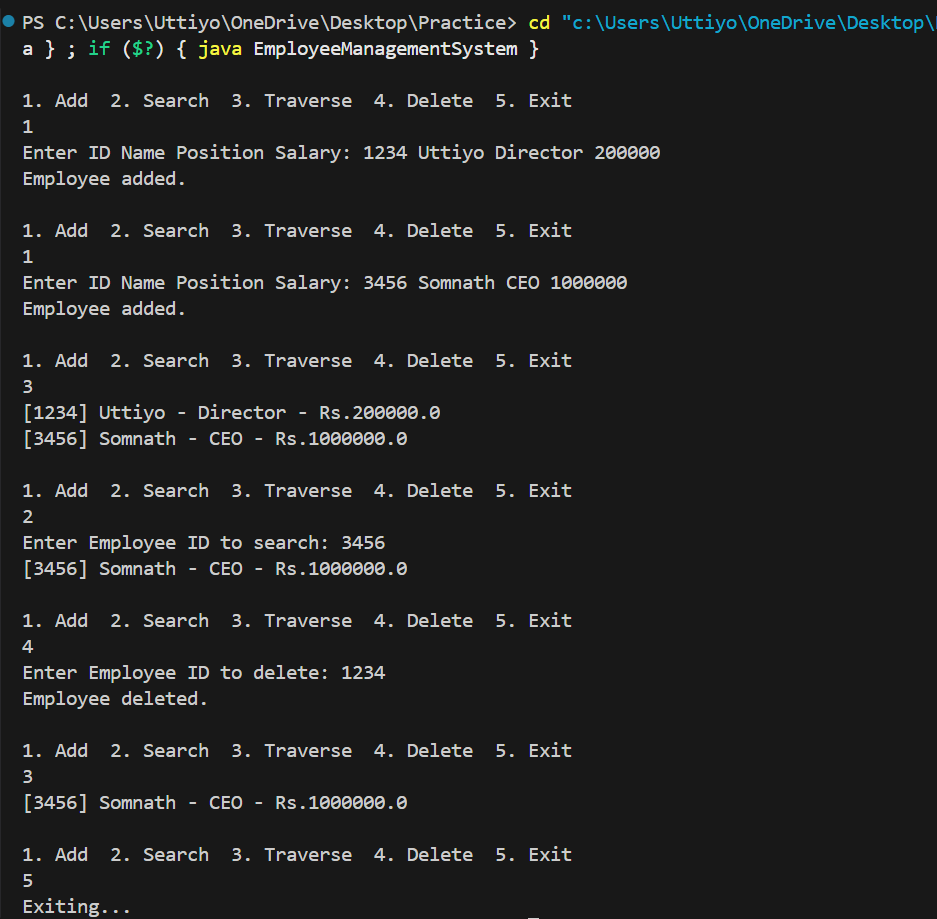
}

}

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

}

}

**Output:**

**Limitations of Arrays**

* **Fixed size**: You must declare the size in advance; resizing is manual and inefficient.
* **Costly deletion/insertion**: Removing or inserting at any position except the end involves shifting elements.
* **Wasted memory**: If the array isn’t full, unused space is wasted.

**When to Use Arrays**

We can use arrays when:

* The number of elements is known and fixed.
* You need fast, constant-time access by index.
* Insertions/deletions are rare or always at the end.

**Exercise 5: Task Management System**

Types of Linked Lists:

1. Singly Linked List:  
   Each node points only to the next node.  
   Simple and uses less memory.
2. Doubly Linked List:  
   Each node has two pointers: one to the next node and one to the previous node.  
   Allows traversal in both directions.

**Code:**

import java.util.Scanner;

// Represents a task

class Task {

String taskId, taskName, status;

Task(String 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 Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.next = null;

}

}

public class TaskManagementSystem {

static Node head = null;

static Scanner sc = new Scanner(System.in);

public static void main(String[] args) {

while (true) {

System.out.println("\n1. Add Task 2. Search Task 3. Traverse Tasks 4. Delete Task 5. Exit");

System.out.print("Enter choice: ");

int choice = sc.nextInt();

sc.nextLine(); // consume newline

switch (choice) {

case 1 -> addTask();

case 2 -> searchTask();

case 3 -> traverseTasks();

case 4 -> deleteTask();

case 5 -> {

System.out.println("Exiting..."); return;

}

default -> System.out.println("Invalid choice.");

}

}

}

static void addTask() {

System.out.print("Enter Task ID: ");

String id = sc.nextLine();

System.out.print("Enter Task Name: ");

String name = sc.nextLine();

System.out.print("Enter Status: ");

String status = sc.nextLine();

Task newTask = new Task(id, name, status);

Node newNode = new Node(newTask);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) temp = temp.next;

temp.next = newNode;

}

System.out.println("Task added.");

}

static void searchTask() {

System.out.print("Enter Task ID to search: ");

String id = sc.nextLine();

Node temp = head;

while (temp != null) {

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

System.out.println(temp.task);

return;

}

temp = temp.next;

}

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

}

static void traverseTasks() {

if (head == null) {

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

return;

}

Node temp = head;

while (temp != null) {

System.out.println(temp.task);

temp = temp.next;

}

}

static void deleteTask() {

System.out.print("Enter Task ID to delete: ");

String id = sc.nextLine();

if (head == null) {

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

return;

}

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

head = head.next;

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

return;

}

Node temp = head;

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

temp = temp.next;

}

if (temp.next == null) {

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

} else {

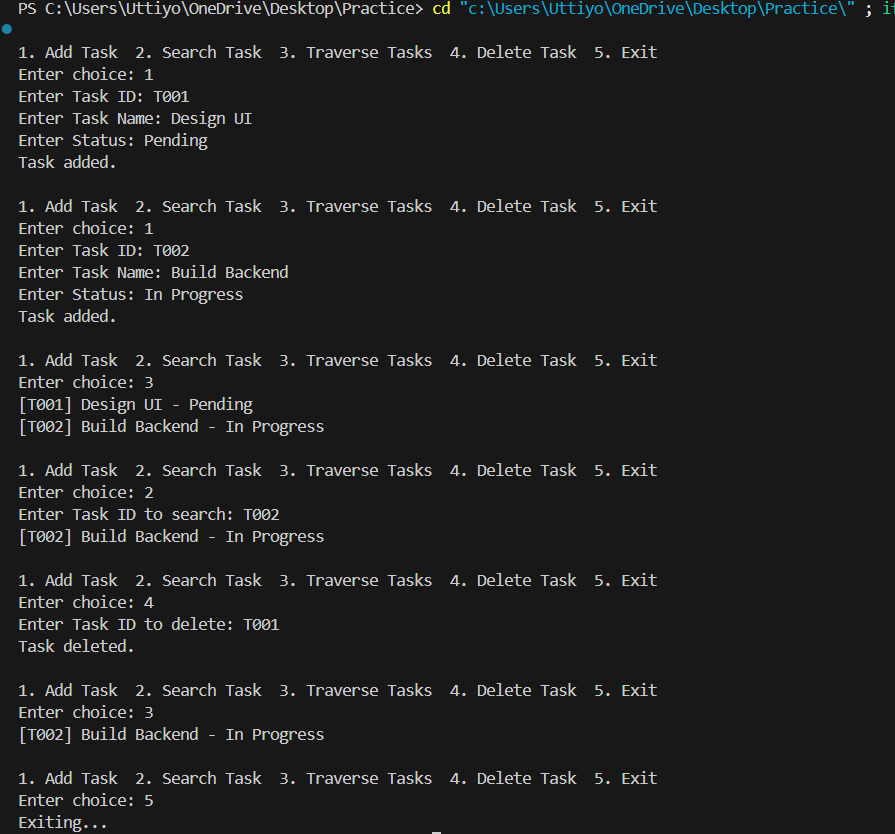
temp.next = temp.next.next;

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

}

}

}

**Output:**

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Explanation |
| Add | O(n) | Add at end requires traversal |
| Search | O(n) | Linear search through nodes |
| Traverse | O(n) | Visits each node once |
| Delete | O(n) | Must find the node and relink |

Advantages of Linked Lists over Arrays

* Dynamic size: Grows and shrinks as needed, no fixed limit.
* Efficient insert/delete: No shifting of elements like arrays.
* No memory wastage: Only uses as much memory as needed.

**Exercise 6: Library Management System**

Linear Search:

* Checks each element one by one.
* Works on unsorted data.
* Simple, but inefficient for large lists.

Binary Search:

* Works only on sorted arrays/lists.
* Repeatedly divides the array and searches in half.
* Much faster than linear search for large sorted datasets.

**Code:**

import java.util.\*;

class Book {

String bookId, title, author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

public String toString() {

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

}

}

public class LibraryManagementSystem {

static Book[] books = {

new Book("B001", "Java Programming", "James Gosling"),

new Book("B002", "Data Structures", "Mark Allen Weiss"),

new Book("B003", "Operating Systems", "Silberschatz"),

new Book("B004", "Database Systems", "Raghu Ramakrishnan")

};

static Scanner sc = new Scanner(System.in);

public static void main(String[] args) {

while (true) {

System.out.println("\n1. Linear Search by Title");

System.out.println("2. Binary Search by Title");

System.out.println("3. Exit");

System.out.print("Enter choice: ");

int choice = sc.nextInt();

sc.nextLine(); // consume newline

switch (choice) {

case 1 -> linearSearch();

case 2 -> binarySearch();

case 3 -> { System.out.println("Exiting..."); return; }

default -> System.out.println("Invalid choice.");

}

}

}

static void linearSearch() {

System.out.print("Enter book title to search: ");

String title = sc.nextLine();

for (Book book : books) {

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

System.out.println("Book found: " + book);

return;

}

}

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

}

static void binarySearch() {

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

System.out.print("Enter book title to search: ");

String title = sc.nextLine();

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) {

System.out.println("Book found: " + books[mid]);

return;

} else if (cmp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

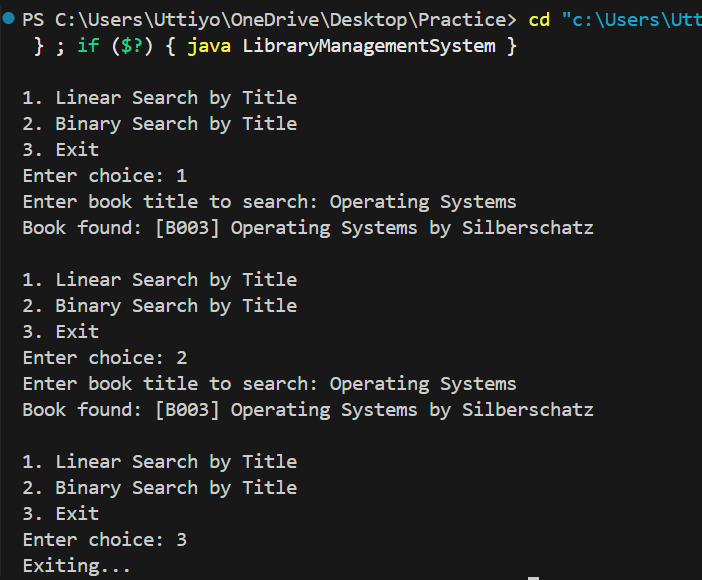
}

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

}

}

**Output:**

****

**Time Complexity Comparison**

|  |  |  |
| --- | --- | --- |
| Search Type | Time Complexity | Use Case |
| Linear Search | O(n) | Works on any unsorted data |
| Binary Search | O(log n) | Requires sorted data |

When to Use Which Algorithm

* Use Linear Search when the book list is unsorted or small.
* Use Binary Search when:
  + The data is pre-sorted.
  + You need fast search performance.
  + You are okay with sorting once (O(n log n)) for repeated fast searches.

**Exercise 7: Financial Forecasting**

* Recursion is when a function calls itself to break a problem into smaller subproblems.
* It’s often used when a problem can be defined in terms of smaller instances of itself (e.g., factorial, Fibonacci).

Recursion simplifies problems by:

1. Breaking them into smaller subproblems, each similar to the original.
2. Reducing code complexity, especially for tasks that involve repetition or branching.

**Code:**

import java.util.Scanner;

public class FinancialForecasting {

// Recursive method to calculate future value

static double forecastRecursive(double value, double rate, int years) {

if (years == 0) return value;

return forecastRecursive(value \* (1 + rate), rate, years - 1);

}

// Optimized version using Math.pow

static double forecastOptimized(double value, double rate, int years) {

return value \* Math.pow(1 + rate, years);

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter current value: ");

double presentValue = sc.nextDouble();

System.out.print("Enter annual growth rate (in %): ");

double rate = sc.nextDouble() / 100;

System.out.print("Enter number of years: ");

int years = sc.nextInt();

double futureRecursive = forecastRecursive(presentValue, rate, years);

double futureOptimized = forecastOptimized(presentValue, rate, years);

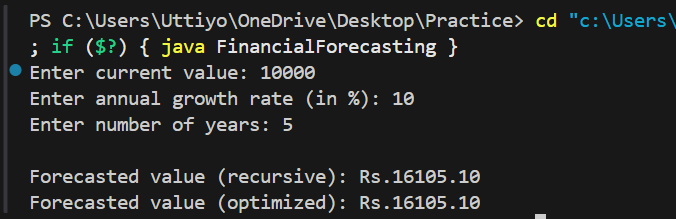
System.out.printf("\nForecasted value (recursive): Rs.%.2f\n", futureRecursive);

System.out.printf("Forecasted value (optimized): Rs.%.2f\n", futureOptimized);

sc.close();

}

}

**Output:**

Time Complexity Analysis

* Each recursive call reduces years by 1.
* So, Time Complexity = O(n) where n is the number of years.
* Space Complexity = O(n) due to recursion stack depth.

**How to Optimize the Recursive Solution**

To avoid excessive computation and stack overflow, replace recursion with the formula:

futureValue = presentValue × Math.pow(1 + rate, years);

This is faster (O(1) time), uses no recursion, and is better for large inputs.