**Exercise 1: Inventory Management System**

**why data structures and algorithms are essential in handling large inventories.**

Data structures and algorithms are essential in handling large inventories because they help store and organize data efficiently, enabling fast access and updates. With the right structures like hash maps or trees, inventory items can be quickly searched, added, or modified. Efficient algorithms also ensure that the system remains scalable and performs well even with thousands of items. This is crucial for maintaining real-time accuracy and responsiveness in inventory management.

**Discussing the types of data structures suitable for this problem**

**Array/Array List** – For storing and accessing products with index-based operations.

**Linked List** – Useful when frequent insertions/deletions are needed. **HashMap** – Ideal for fast lookup using unique product IDs.

**Tree (BST/AVL)** – Best for sorted data and range-based searches.

**Stack/Queue** – Suitable for managing inventory in LIFO or FIFO order.

**Implementing:**

import java.util.HashMap;

import java.util.Scanner;

class Product {

int productId;

String productName;

int quantity;

double price;

// Constructor

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public void display() {

System.out.println("ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: ₹" + price);

}

}

// InventorySystem class to manage inventory operations

class InventorySystem {

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

Scanner sc = new Scanner(System.in);

public void addProduct() {

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

int id = sc.nextInt();

sc.nextLine(); // Consume the newline character

if (inventory.containsKey(id)) {

System.out.println("Error: Product with ID " + id + " already exists.");

return;

}

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

String name = sc.nextLine();

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

int quantity = sc.nextInt();

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

double price = sc.nextDouble();

Product product = new Product(id, name, quantity, price);

inventory.put(id, product);

System.out.println("product added successfully.");

}

public void viewProducts() {

if (inventory.isEmpty()) {

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

return;

}

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

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

p.display();

}

}

// Method to update quantity of a product

public void updateQuantity() {

System.out.print("Enter Product ID to update quantity: ");

int id = sc.nextInt();

if (inventory.containsKey(id)) {

System.out.print("Enter new quantity: ");

int quantity = sc.nextInt();

inventory.get(id).quantity = quantity;

System.out.println("Quantity updated successfully.");

} else {

System.out.println("roduct not found with ID: " + id);

}

}

// Method to remove a product

public void removeProduct() {

System.out.print("Enter Product ID to remove: ");

int id = sc.nextInt();

if (inventory.remove(id) != null) {

System.out.println("🗑 Product removed successfully.");

} else {

System.out.println("roduct not found with ID: " + id);

}

}

// Method to close scanner (optional, good practice)

public void closeScanner() {

sc.close();

}

}

public class Main {

public static void main(String[] args) {

InventorySystem system = new InventorySystem();

int choice;

do {

System.out.println("\n=== Inventory Menu ===");

System.out.println("1. Add Product");

System.out.println("2. View Products");

System.out.println("3. Update Quantity");

System.out.println("4. Remove Product");

System.out.println("5. Exit");

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

while (!system.sc.hasNextInt()) {

System.out.println("nvalid input. Please enter a number between 1 and 5.");

system.sc.next(); // Consume invalid input

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

}

choice = system.sc.nextInt();

system.sc.nextLine(); // Consume newline

switch (choice) {

case 1: system.addProduct(); break;

case 2: system.viewProducts(); break;

case 3: system.updateQuantity(); break;

case 4: system.removeProduct(); break;

case 5:System.out.println(" Exiting. Thank you!");

system.closeScanner();

break;

default:

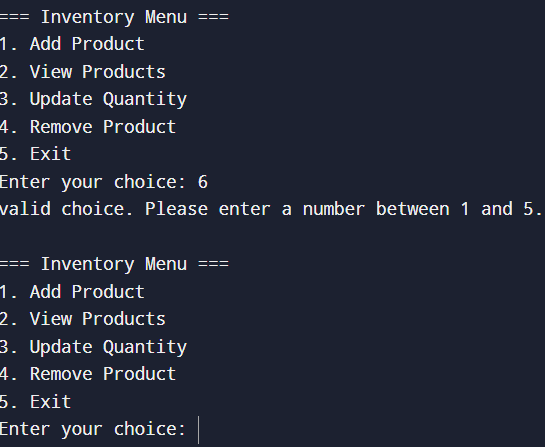
System.out.println("valid choice. Please enter a number between 1 and 5.");

}

} while (choice != 5);

}

}

****

**Analysis:**

**Analyzing the time complexity of each operation (add, update, delete) in your chosen data structure**

**Add (Insertion)** – Average: **O(1)**, Worst-case: **O(n)** (if many collisions).

**Update** – Average: **O(1)**, since it's just replacing the value of an existing key.

**Delete (Removal)** – Average: **O(1)**, Worst-case: **O(n)** in case of collision chains.

**Discussing how you can optimize these operations**

###### Optimize operations in a Hash Map for inventory, it's important to use a good hash function to minimize collisions and ensure even distribution of keys. Maintaining an appropriate load factor helps prevent performance drops by resizing the map before it gets too full. Using unique and consistent keys, such as product IDs, ensures faster and more accurate look-ups. Additionally, initializing the map with a suitable capacity can reduce rehashing and improve overall efficiency.

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

**Explaining Big O notation and how it helps in analyzing algorithms.**

Big O notation is a mathematical way to describe the **time or space complexity** of an algorithm as the input size grows. It helps us understand the **worst-case performance** and efficiency of an algorithm, regardless of hardware or programming language. By using Big O, we can compare different algorithms and choose the one that scales better for large data. For example, O(log n) is faster and more efficient than O(n²) for large inputs.

**Describing the best, average, and worst-case scenarios for search operations.**

**Best Case**: The element is found immediately, such as being the first in a list – **O(1)** time.

**Average Case**: The element is somewhere in the middle – **O(n)** for linear search, **O(log n)** for binary search on sorted data.

**Worst Case**: The element is not found or is the last checked – **O(n)** for linear, **O(log n)** for binary.

**Implementing:**

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

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

for (Product product : products) {

if (product.productId == targetId) {

return product;

}

}

return null;

}

// Binary Search (Requires sorted array)

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

int left = 0, 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;

}

public static void main(String[] args) {

// Creating product array

Product[] products = {

new Product(103, "Shoes", "Fashion"),

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

new Product(102, "Book", "Education"),

new Product(104, "Watch", "Accessories")

};

int searchId = 102;

// Linear Search (on unsorted array)

Product result1 = linearSearch(products, searchId);

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

// Sort the array by productId before Binary Search

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

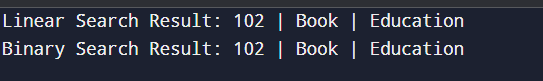
// Binary Search (on sorted array)

Product result2 = binarySearch(products, searchId);

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

}

}



**Analysis:**

**Comparing the time complexity of linear and binary search algorithms**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** | **Condition** |
| --- | --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) | Works on unsorted arrays |
| **Binary Search** | O(1) | O(log n) | O(log n) | Requires sorted array |

**Discussing which algorithm is more suitable for your platform and why.**

For an **e-commerce platform**, **binary search** is more suitable because it offers **faster performance (O(log n))** when searching through large product lists. Since users expect quick search results, especially when browsing thousands of items, binary search ensures better scalability. Although it requires the data to be sorted, sorting can be done once during data load or periodically. In contrast, **linear search is slower (O(n))** and becomes inefficient as the product list grows, making it less ideal for real-time platforms.

**Exercise 3: Sorting Customer Orders**

**Explaining different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).**

### ****Bubble Sort**** Bubble Sort is one of the simplest sorting methods. Imagine looking at a list of numbers and comparing two at a time. If the first one is bigger than the next, you swap them. You keep doing this for the whole list, again and again, until no more swaps are needed. The largest values slowly move to the end — like bubbles rising to the surface — hence the name.

**Example in real life:**  
Think of students standing in a line by height. You compare two students at a time and ask them to switch places if the taller one is before the shorter one. Keep going until everyone is in the right order.

**Time it takes:**

Fastest when things are already sorted (O(n))

Slow when the list is jumbled (O(n²))

### ****Insertion Sort**** Insertion Sort works like how we sort playing cards. You take one card at a time and place it in the right position among the cards already sorted in your hand. It builds the sorted list one item at a time.

**Example in real life:**  
You’re holding a few cards. When you pick a new card, you shift the others around to insert it in the correct spot. That’s what this algorithm does.

**Time it takes:**

Quick if the data is almost sorted (O(n))

Slower if the data is in reverse (O(n²))

### ****Quick Sort**** Quick Sort is much smarter. It picks one element as a "pivot" and puts all smaller items on one side and all bigger items on the other. Then it repeats the same process on both sides. Eventually, the whole list becomes sorted.

### **Example in real life:** Imagine asking your friends to stand to the left if they have less money than you, and to the right if they have more. Then each side keeps doing that again with someone else. That’s how Quick Sort works.

**Time it takes:**

Usually very fast (O(n log n))

Can be slow if bad pivots are picked every time (O(n²))

### ****4. Merge Sort**** Merge Sort splits the list into smaller and smaller parts (even down to single elements), then starts combining those parts in sorted order. It keeps doing this until the whole list is merged and sorted.

**Example in real life:**  
Think of tearing a newspaper into pieces with one word each, sorting each tiny piece, and then sticking them back together in order.

**Time it takes:**Always consistent and fast: O(n log n),But it uses extra memory

**Implementing Bubble Sort:**

import java.util.\*;

class order{

int orderId;

String customerName;

double totalPrice;

int temp = 0;

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 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;

}

}

}

}

public static void main(String[] args) {

order[] orders = {

new order(101, "Bindhu", 700),

new order(102, "Hema", 500),

new order(103, "Siri", 300)

};

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

for (order o : orders) {

System.out.println(o);

}

// Calling bubble sort

bubbleSort(orders);

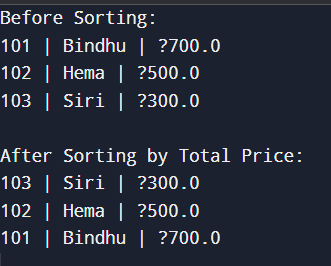
System.out.println("\nAfter Sorting by Total Price:");

for (order o : orders) {

System.out.println(o);

}

}

****

**Implementing Quick Sort :**

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;

}

}

public class Main {

// Quick Sort

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);

}

}

// Partition for Quick Sort

public 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 main(String[] args) {

Order[] orders = {

new Order(101, "kamala", 599.0),

new Order(102, "sriya", 299.0),

new Order(103, "nandhini", 499.0),

new Order(104, "varshini", 799.0),

new Order(105, "anjali", 199.0)

};

System.out.println("Before Quick Sort:");

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

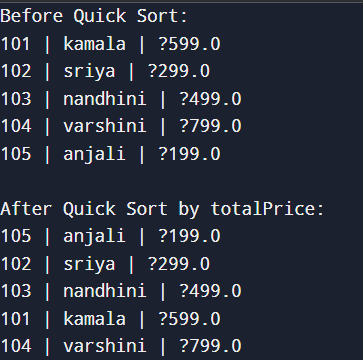
quickSort(orders, 0, orders.length - 1);

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

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

}

}



**Analysis:**

**Comparing the performance (time complexity) of Bubble Sort and Quick Sort.**

| **Sorting Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| **Bubble Sort** | O(n) | O(n²) | O(n²) | O(1) |
| **Quick Sort** | O(n log n) | O(n log n) | O(n²) | O(log n) |

**Discussing why Quick Sort is generally preferred over Bubble Sort.**

Quick Sort is faster and more efficient than Bubble Sort, especially for large datasets.  
It uses a smart divide-and-conquer approach, while Bubble Sort does many unnecessary comparisons.  
That’s why Quick Sort is used in real-world applications, but Bubble Sort is mainly for learning.

**Exercise 4: Employee Management System**

**Explaining how arrays are represented in memory and their advantages**.

Arrays are stored in **contiguous memory locations**, meaning all elements are placed side by side in a single block. Each element can be accessed directly using its index, which makes array access very fast (O(1)). Since the size of an array is fixed during creation, it is memory-efficient and simple to manage. Arrays are easy to traverse using loops and work well with the CPU cache due to their continuous layout. These features make arrays ideal for scenarios where the number of elements is known in advance and fast access is important.

**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 Main {

static Employee[] employees = new Employee[100]; // max 100 employees

static int count = 0; // number of employees

public static void addEmployee(Employee e) {

if (count < employees.length) {

employees[count++] = e;

} else {

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

}

}

public static void searchEmployee(int id) {

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

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

System.out.println("Employee Found: " + employees[i]);

return;

}

}

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

}

//traversing all employees

public static void showAllEmployees() {

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

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

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

}

}

// Delete employee by ID

public static void deleteEmployee(int id) {

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

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

// shift elements to left

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

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

}

employees[--count] = null; // reduce count and remove last

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

return;

}

}

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

}

public static void main(String[] args) {

addEmployee(new Employee(1, "Kanna", "Manager", 70000));

addEmployee(new Employee(2, "Gopala krishna", "Developer", 50000));

addEmployee(new Employee(3, "SUhasini", "Tester", 40000));

showAllEmployees();

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

searchEmployee(2);

System.out.println("\nDeleting ID 1:");

deleteEmployee(1);

System.out.println("\nAfter Deletion:");

showAllEmployees();

}

}

**Analysis**

**Analyzing the time complexity of each operation (add, search, traverse, delete).**

When you add something at the end of an array and there's space, it’s really fast — it just takes a moment. But if you're trying to find something, like searching for an employee by ID, you might have to check each one until you find it, which takes more time. Going through all the elements one by one (traversing) also takes time as the list grows. Deleting is a bit tricky — if you remove something from the middle or beginning, you have to shift everything after it to fill the gap, which slows things down too.

**Discussing the limitations of arrays and when to use them**

Arrays come with a fixed size, so once they're full, you can't just add more unless you create a new one. If you want to insert or delete something in the middle, it can be slow because everything has to shift to make space or fill the gap. Unlike Array List or Linked List, arrays can’t grow or shrink automatically, and they don’t manage memory as flexibly. But if you know how many items you need from the start and want fast access using index numbers, arrays work really well.

**Exercise 5: Task Management System**

**Explaining the different types of linked lists (Singly Linked List, Doubly Linked List)**

A **Singly Linked List** is a linear data structure where each node points only to the next node.It allows traversal in one direction, from head to tail.  
A **Doubly Linked List** has nodes that point to both the next and previous nodes.  
This allows traversal in both forward and backward directions.  
While doubly linked lists use more memory, they make insertion and deletion easier.

**Implementation:**

public class TaskManagementSystem {

static class Task {

int taskId;

String taskName;

String status;

Task(int taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public String toString() {

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

}

}

static class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.next = null;

}

}

// Task Manager class with linked list logic

static class TaskManager {

Node head;

// Add task at end

void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

// Search task by ID

Task searchTask(int taskId) {

Node temp = head;

while (temp != null) {

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

return temp.task;

}

temp = temp.next;

}

return null;

}

// Delete task by ID

boolean deleteTask(int taskId) {

if (head == null) return false;

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

head = head.next;

return true;

}

Node temp = head;

while (temp.next != null) {

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

temp.next = temp.next.next;

return true;

}

temp = temp.next;

}

return false;

}

// Print all tasks

void printTasks() {

Node temp = head;

if (temp == null) {

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

return;

}

while (temp != null) {

System.out.println(temp.task);

temp = temp.next;

}

}

}

// Main method

public static void main(String[] args) {

TaskManager manager = new TaskManager();

// Add sample tasks

manager.addTask(new Task(120, "Write report", "Pending"));

manager.addTask(new Task(150, "Team meeting", "Completed"));

manager.addTask(new Task(170, "Code review", "In Progress"));

// Display tasks

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

manager.printTasks();

// Search task

System.out.println("\nSearching for task with ID 102:");

Task result = manager.searchTask(102);

System.out.println(result != null ? result : "Task not found");

// Delete task

System.out.println("\nDeleting task with ID 101:");

boolean deleted = manager.deleteTask(101);

System.out.println(deleted ? "Task deleted successfully." : "Task not found");

// Display after deletion

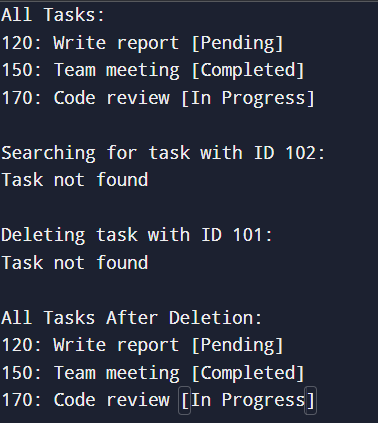
System.out.println("\nAll Tasks After Deletion:");

manager.printTasks();

}

}

**OUTPUT:**

****

**Analysis:**

**Analyzing the time complexity of each operation.**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Add (at end)** | O(n) | Must traverse the entire list to reach the last node. |
| **Search (by ID)** | O(n) | May need to scan all nodes to find the match. |
| **Delete (by ID)** | O(n) | Must find the node and update links, requiring traversal. |

**Discussing when to use each algorithm based on the data set size and order**.

**Insertion Sort**: Best for **small or nearly sorted linked lists**, as it works well with sequential access.

**Merge Sort**: Ideal for **large linked lists**, since it doesn’t require random access and has consistent **O(n log n)** time.

**Bubble Sort / Selection Sort**: Not recommended for linked lists due to excessive pointer swapping and **O(n²)** time.

**Exercise 6: Library Management System**

**Explaining linear search and binary search algorithms.**

**Linear Search** goes through each element one by one to find the target.  
It works on both sorted and unsorted data.  
Time complexity is O(n), and it's simple to implement.  
**Binary Search** divides the sorted array into halves to search efficiently.  
It has a faster time complexity of O(log n) but needs sorted data.

**Implementing:**

import java.util.\*;

class Book {

int bookId;

String title;

String author;

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 Book linearSearch(List<Book> books, String targetTitle) {

for (Book book : books) {

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

return book;

}

}

return null;

}

// Binary Search by title (list must be sorted)

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

int low = 0;

int high = books.size() - 1;

while (low <= high) {

int mid = (low + high) / 2;

Book midBook = books.get(mid);

int cmp = midBook.title.compareToIgnoreCase(targetTitle);

if (cmp == 0) return midBook;

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

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

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

books.add(new Book(1, "Java Basics", "Bindhu"));

books.add(new Book(2, "Data Structures", "hema"));

books.add(new Book(3, "Operating Systems", "siri"));

books.add(new Book(4, "Algorithms", "homitha"));

// Linear Search (works on unsorted list)

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

Book result1 = linearSearch(books, "Data Structures");

System.out.println(result1 != null ? result1 : "Book not found");

// Sort the list by title before binary search

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

// Binary Search (requires sorted list)

System.out.println("\n⚡ Binary Search Result:");

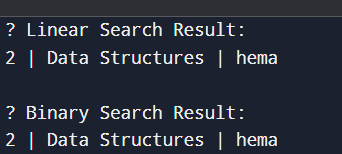
Book result2 = binarySearch(books, "Data Structures");

System.out.println(result2 != null ? result2 : "Book not found");

}

}

**OUTPUT:**



**Analysis:**

**Comparing the time complexity of linear and binary search.**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Notes** |
| --- | --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) | Good for **small or unsorted** data |
| **Binary Search** | O(1) | O(log n) | O(log n) | Requires **sorted** data |

**Discussing when to use each algorithm based on the data set size and order.**

**linear search** for **small or unsorted** data, as it works without sorting.  
It’s simple and effective when the dataset is not large.

**binary search** for **large, sorted** datasets to get faster results.  
Sorting is required, but the search is much quicker with **O(log n)** time.

**Exercise 7: Financial Forecasting**

**Explaining the concept of recursion and how it can simplify certain problems.**

**Recursion** is when a function calls **itself** to solve a smaller part of the problem.  
It continues until a **base case** is reached, which stops the recursion.  
It can **simplify complex problems** like factorials, Fibonacci, or tree traversal.  
Instead of writing loops, recursion breaks a problem into smaller, repeatable steps.

**Implementing:**

public class Main {

public static double forecastValue(double currentValue, double growthRate, int years) {

// Base case: No more years left to forecast

if (years == 0) {

return currentValue;

}

return forecastValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double currentValue = 10000; // Initial value

double growthRate = 0.10; // 10% annual growth rate

int years = 5; // Forecast for 5 years

double futureValue = forecastValue(currentValue, growthRate, years);

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

}

}

**OUTPUT:**

Screenshot 2025-06-21 135316

**Analysis:**

**Discussing the time complexity of recursive algorithm.**

The time complexity of the recursive financial forecasting algorithm is **O(n)**, where n is the number of years. This is because the function calls itself once for each year, reducing the years parameter by 1 with every call until it reaches the base case (when years equals 0). Each recursive call performs a simple constant-time calculation (multiplication and addition)

**Explaining how to optimize the recursive solution to avoid excessive computation.**

To optimize a recursive solution, you can convert it into an **iterative approach** to prevent deep recursion and stack overflow.For problems with **overlapping subproblems**, apply **memoization** to cache results of previous calls.  
This avoids recomputation and significantly improves performance.  
Additionally, use **tail recursion** (if supported by the language) to optimize stack usage.