**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.

**Answer:**

Data structures and algorithms are essential in inventory management systems to ensure fast and efficient handling of large volumes of products. They help in organizing data for quick access, updates, and deletions. For example, using a HashMap allows constant-time operations like searching by productId. This improves system performance and scalability. Without proper structures, the system may slow down as the inventory grows. Hence, choosing the right data structure is key for a reliable and responsive system.

* + Discuss the types of data structures suitable for this problem.

**Answer:**

An ArrayList is easy to use and allows simple iteration over products.  
However, it is slow for searching or updating in large inventories due to linear time complexity.  
A HashMap stores data as key-value pairs, enabling constant-time access.  
It is ideal for operations like search, update, or delete using productId as the key.  
Thus, HashMap is more suitable for efficient inventory management.

1. **Setup:**
   * Create a new project for the inventory management system.
2. **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.

**Product.java:**

package InventoryManagement;

public class Product {

String productId;

String productName;

int quantity;

double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public void displayProduct() {

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

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

}

}

**InventoryManagementTest.java:**

package InventoryManagement;

import java.util.HashMap;

import java.util.\*;

public class InventoryManagementTest {

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

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int choice;

do {

System.out.println("Inventory Management System:");

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

System.out.println("2. Update Product");

System.out.println("3. Delete Product");

System.out.println("4. View Inventory");

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

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

choice = sc.nextInt();

sc.nextLine();

switch (choice) {

case 1 -> addProduct(sc);

case 2 -> updateProduct(sc);

case 3 -> deleteProduct(sc);

case 4 -> displayInventory();

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

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

}

} while (choice != 5);

}

static void addProduct(Scanner sc) {

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

String id = sc.nextLine();

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

String name = sc.nextLine();

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

int qty = sc.nextInt();

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

double price = sc.nextDouble();

sc.nextLine();

Product p = new Product(id, name, qty, price);

inventory.put(id, p);

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

}

static void updateProduct(Scanner sc) {

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

String id = sc.nextLine();

if (inventory.containsKey(id)) {

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

int qty = sc.nextInt();

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

double price = sc.nextDouble();

sc.nextLine();

Product p = inventory.get(id);

p.quantity = qty;

p.price = price;

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

} else {

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

}

}

static void deleteProduct(Scanner sc) {

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

String id = sc.nextLine();

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

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

} else {

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

}

}

static void displayInventory() {

if (inventory.isEmpty()) {

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

} else {

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

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

p.displayProduct();

}

}

}

}

A screenshot of a computer

AI-generated content may be incorrect.

1. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.

**Answer:**

|  |  |
| --- | --- |
| **Operations** | **Time Complexity** |
| Add | O(1) |
| Update | O(1) |
| Delete | O(1) |
| Display | O(n) |

* + Discuss how you can optimize these operations.

**Answer:**

If sorted product IDs are needed, a TreeMap is better than a HashMap as it maintains order with O(log n) operations.

**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.

**Answer:**

Big O notation is used to describe the time or space complexity of an algorithm as the input size grows. It provides an upper bound on the running time, helping us understand the scalability and efficiency of an algorithm.

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

**Answer:**

In linear search, the best-case occurs when the target is the first item, taking O(1) time. On average, it checks about half the elements, making it O(n), and in the worst case (last or not found), it's also O(n). Binary search, on the other hand, is much faster for sorted data with a best, average, and worst-case time complexity of O(log n), except the best-case where the element is found in the middle at O(1). This makes binary search more efficient for large sorted datasets.

1. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
2. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.

**Product.java:**

package EcommercePlatformSearch;

public 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 + ")";

}

}

**LinearSearch.java:**

package EcommercePlatformSearch;

public class LinearSearch {

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

for (Product product : products) {

if (product.productName.equalsIgnoreCase(searchKey)) {

return product;

}

}

return null;

}

}

**BinarySearch.java:**

package EcommercePlatformSearch;

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

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

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

while (left <= right) {

int mid = left + (right - left) / 2;

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

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

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

else right = mid - 1;

}

return null;

}

public static void sortProducts(Product[] products) {

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

}

}

**ProductTest.java:**

package EcommercePlatformSearch;

public class ProductTest {

public static void main(String[] args) {

Product[] products = {

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

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

new Product(3, "Book", "Education"),

new Product(4, "Headphones", "Electronics"),

new Product(5, "Pen", "Stationery")

};

String searchKey = "Book";

// Linear Search

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

Product result1 = LinearSearch.*linearSearch*(products, searchKey);

System.*out*.println(result1 != null ? "Found: " + result1 : "Not Found");

// Binary Search

System.*out*.println("\nBinary Search (after sorting):");

BinarySearch.*sortProducts*(products);

Product result2 = BinarySearch.*binarySearch*(products, searchKey);

System.*out*.println(result2 != null ? "Found: " + result2 : "Not Found");

}

}

A screenshot of a computer

AI-generated content may be incorrect.

1. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.

**Answer:**

Linear Search: O(n) – Time increases linearly with the number of elements.

Binary Search: O(log n) – Time grows logarithmically; much faster on large sorted datasets.

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

**Answer:**

Binary Search is more suitable for an e-commerce platform because it is faster (O(log n)) and efficient for large, sorted datasets. It ensures quicker search results and better scalability compared to Linear Search.

**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).

**Answer:**

Bubble Sort:

* Repeatedly swaps adjacent elements if they are in the wrong order.
* Time Complexity:
  + Best: O(n) (when already sorted)
  + Average & Worst: O(n²)
* Space Complexity: O(1) (in-place)
* Use Case: Simple and easy, but inefficient for large datasets.

Insertion Sort:

* Builds the sorted array one element at a time by inserting into the correct position.
* Time Complexity:
  + Best: O(n)
  + Average & Worst: O(n²)
* Space Complexity: O(1)
* Use Case: Efficient for small or nearly sorted datasets.

Quick Sort:

* Uses divide-and-conquer by selecting a pivot, partitioning the array, and recursively sorting.
* Time Complexity:
  + Best & Average: O(n log n)
  + Worst: O(n²) (rare; happens when the pivot is poorly chosen)
* Space Complexity: O(log n)
* Use Case: Fast in practice, widely used.

Merge Sort:

* Divides the array, sorts both halves, and merges them.
* Time Complexity: O(n log n) in all cases.
* Space Complexity: O(n)
* Use Case: Stable and predictable, preferred when stability is required.

1. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
2. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.

**Order.java:**

package SortingCustomers;

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;

}

public String toString() {

return "OrderID: " + orderId + ", Name: " + customerName + ", Total Price: " + totalPrice;

}

}

**BubbleSort.java:**

package SortingCustomers;

public class BubbleSort {

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;

}

}

}

}

}

**QuickSort.java:**

package SortingCustomers;

public class QuickSort {

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;

}

}

**SortingCustomerOrdersTest.java:**

package SortingCustomers;

public class SortingCustomerOrdersTest {

public static void main(String[] args) {

Order[] orders = {

new Order(101, "Sai", 250.0),

new Order(102, "Babu", 120.0),

new Order(103, "Charan", 300.0),

new Order(104, "Danush", 150.0)

};

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

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

// BubbleSort.bubbleSort(orders);

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

System.*out*.println("\nSorted Orders by Total Price:");

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

}

}

A screenshot of a computer

AI-generated content may be incorrect.

1. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**Answer:**

Bubble Sort is simple but inefficient for large datasets with average and worst-case time complexity of O(n²) and constant space O(1).  
Quick Sort is much faster on average with O(n log n) time and O(log n) space, making it preferable for large datasets.

**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.

**Answer:**

Arrays are stored in contiguous memory locations, allowing constant-time access to elements using their index.  
Each element’s address is calculated using the formula: base address + index × element size.  
Advantages include fast access, cache efficiency, and simplified data management for fixed-size collections.

1. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
2. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.

**Employee.java:**

package EmployeeManagement;

public 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 void display() {

System.*out*.println("ID: " + employeeId + ", Name: " + name +

", Position: " + position + ", Salary: " + salary);

}

}

**EmployeeManagementSystem.java:**

package EmployeeManagement;

public class EmployeeManagementSystem {

private Employee[] employees;

private int count;

public EmployeeManagementSystem(int size) {

employees = new Employee[size];

count = 0;

}

// Add

public void addEmployee(Employee emp) {

if (count < employees.length) {

employees[count++] = emp;

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

} else {

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

}

}

// Search

public Employee searchEmployee(int empId) {

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

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

return employees[i];

}

}

return null;

}

// Display all

public void displayAllEmployees() {

if (count == 0) {

System.*out*.println("No employees to display.");

return;

}

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

employees[i].display();

}

}

// Delete

public void deleteEmployee(int empId) {

boolean found = false;

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

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

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

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

}

employees[count - 1] = null;

count--;

found = true;

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

break;

}

}

if (!found) {

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

}

}

}

**EmployeeManagementSystemTest.java:**

package EmployeeManagement;

public class EmployeeManagementSystemTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

EmployeeManagementSystem system = new EmployeeManagementSystem(5);

// Add

system.addEmployee(new Employee(101, "Alice", "Manager", 75000));

system.addEmployee(new Employee(102, "Bob", "Developer", 60000));

system.addEmployee(new Employee(103, "Charlie", "Designer", 55000));

// Display All

System.*out*.println("\nAll Employees:");

system.displayAllEmployees();

// Search

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

Employee e = system.searchEmployee(102);

if (e != null) e.display();

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

// Delete

System.*out*.println("\nDeleting employee with ID 102:");

system.deleteEmployee(102);

// Display

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

system.displayAllEmployees();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

1. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).

|  |  |
| --- | --- |
| **Operations** | **Time Complexity** |
| Add | O(1) |
| Search | O(n) |
| Traverse | O(n) |
| Delete | O(n) |

* + Discuss the limitations of arrays and when to use them.

**Answer:**

Arrays have a fixed size, so you can't add more elements once it's full.  
Insertion and deletion can be slow because elements may need to be shifted.  
Use arrays when you know the number of items in advance and need fast access by index.

**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).

**Answer:**

Singly Linked List:

* Each node contains data and a pointer to the next node.
* Can be traversed in one direction only.
* Simple structure, less memory usage than doubly linked lists.
* Suitable for adding/removing items from the beginning.

Doubly Linked List:

* Each node contains data, a pointer to the next node, and a pointer to the previous node.
* Can be traversed in both directions.
* More flexible but uses more memory due to the extra pointer.
* Suitable for applications needing frequent insertion/deletion from both ends.

1. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
2. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.

**Task.java:**

package TaskManagement;

public 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 "Task ID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

**TaskNode.java:**

package TaskManagement;

public class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

**TaskLinkedList.java:**

package TaskManagement;

public class TaskLinkedList {

private TaskNode head;

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;

}

}

public Task searchTask(int taskId) {

TaskNode temp = head;

while (temp != null) {

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

return temp.task;

}

temp = temp.next;

}

return null;

}

public boolean deleteTask(int taskId) {

if (head == null) return false;

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

head = head.next;

return true;

}

TaskNode current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

return true;

}

return false;

}

public void displayTasks() {

TaskNode temp = head;

while (temp != null) {

System.*out*.println(temp.task);

temp = temp.next;

}

}

}

**TaskManagementSystemTest.java:**

package TaskManagement;

public class TaskManagementSystemTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

TaskLinkedList taskList = new TaskLinkedList();

taskList.addTask(new Task(1, "Design UI", "Pending"));

taskList.addTask(new Task(2, "Develop Backend", "In Progress"));

taskList.addTask(new Task(3, "Testing", "Not Started"));

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

taskList.displayTasks();

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

Task task = taskList.searchTask(2);

if (task != null) {

System.*out*.println(task);

} else {

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

}

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

boolean deleted = taskList.deleteTask(1);

System.*out*.println("Deleted: " + deleted);

System.*out*.println("\nUpdated Task List:");

taskList.displayTasks();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

1. **Analysis:**
   * Analyze the time complexity of each operation.

|  |  |
| --- | --- |
| **Operations** | **Time Complexity** |
| Add Task | O(n) |
| Search task | O(n) |
| Delete task | O(n) |
| Traverse task | O(n) |

* + Discuss the advantages of linked lists over arrays for dynamic data.

**Answer:**

Linked lists can grow or shrink easily while the program is running.  
Adding or removing items is fast because we don’t need to move other elements.  
They use memory only when needed, so there's no waste.  
We don’t have to decide the size in advance like we do with arrays.

**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.

**Answer:**

Linear Search:

* Checks each element one by one until the target is found or the list ends.
* Time Complexity:
  + Best Case: O(1) (first element is the target)
  + Worst Case: O(n) (target is at the end or not present)
* Usage: Works on unsorted data.

Binary Search:

* Divides the sorted list in half repeatedly to find the target.
* Time Complexity:
  + Best Case: O(1)
  + Worst Case: O(log n)
* Usage: Only works on sorted data.

1. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
2. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).

**Book.java:**

package LibraryManagement;

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;

}

public String toString() {

return "BookID: " + bookId + ", Title: " + title + ", Author: " + author;

}

}

**SearchOperations.java:**

package LibraryManagement;

public class SearchOperations {

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

for (Book book : books) {

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

return book;

}

}

return null;

}

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

int left = 0;

int right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (comparison == 0) {

return books[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**LibraryManagemntTest.java:**

package LibraryManagement;

import java.util.Arrays;

import java.util.Comparator;

public class LibraryManagemntTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

Book[] books = {

new Book(1, "Data Structures", "Mark Weiss"),

new Book(2, "Java Programming", "James Gosling"),

new Book(3, "Algorithms", "Robert Sedgewick")

};

// Linear Search

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

Book foundBook = SearchOperations.linearSearchByTitle(books, "Java Programming");

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

// Binary Search

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

// Binary Search Test

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

Book foundBook2 = SearchOperations.binarySearchByTitle(books, "Algorithms");

System.out.println(foundBook2 != null ? foundBook2 : "Book not found");

}

}

A screenshot of a computer

AI-generated content may be incorrect.

1. **Analysis:**
   * Compare the time complexity of linear and binary search.

**Answer:**

Linear Search has a time complexity of O(n) in the worst case, where *n* is the number of elements.

Binary Search has a time complexity of O(log n), but it requires the data to be sorted.

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

**Answer:**

Linear search is best when the list is small or not sorted, as it's simple and easy to use.  
Binary search works well when the list is already sorted and has many items.

**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.

**Answer:**

Recursion is a method where a function calls itself to solve a problem. Each recursive call should bring the problem closer to a base case that can be solved directly.

It simplifies problems that have repetitive subproblems, such as:

* Calculating factorials
* Traversing tree structures
* Solving dynamic financial projections

1. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
2. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.

**FinalForecasting.java:**

package Forecasting;

public class FinalForecasting {

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

if (years == 0) {

return currentValue;

}

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

}

public static void main(String[] args) {

double currentValue = 10000;

double rate = 0.08;

int years = 5;

double futureValue = *calculateFutureValue*(currentValue, rate, years);

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

}

}

A screenshot of a computer

AI-generated content may be incorrect.

1. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.

**Answer:**

The time complexity of the recursive algorithm is O(n) because it makes one function call for each year. If you want to forecast for 5 years, the function runs 5 times. This is because each year reduces the problem by one, moving toward the base case. So, the number of steps depends directly on the number of years. It grows linearly as the years increase.

* + Explain how to optimize the recursive solution to avoid excessive computation.

**Answer:**

To optimize the recursive solution and avoid excessive computation, we can replace recursion with a simple loop, which uses less memory and runs faster. Recursion stores each call in memory, which can slow things down if the number of years is large. A loop does the same calculation step by step without storing extra calls. This makes the program more efficient and avoids errors like stack overflow. So, using an iterative method is a better choice for large inputs.