**WEEK-1**

**DATA STRUCTURES AND ALGORITHMS**

**EXERCISE-1:** Inventory Management System.

**PROBLEM STATEMENT:**

**Step 1:** Understanding the Problem.

**Why Are Data Structures and Algorithms Important?**

In an inventory management system, efficient storage and retrieval of product data is essential for:

* Fast lookups (e.g., finding product by ID)
* Quick updates (e.g., modifying quantity or price)
* Real-time performance as inventory grows

**Suitable Data Structures:**

| **Data Structure** | **Use Case** | **Pros** |
| --- | --- | --- |
| Array List | Simple list of products | Good for iteration |
| HashMap | Map product Id to Product for fast lookup | O(1) time for get/put/delete |
| Tree Map | Sorted keys (e.g., product Id) | O(log n) for operations |

**Best choice**: HashMap<String, Product> — allows fast access by product Id.

**Step 2–3:** Java Code Implementation (Runnable in JDoodle).

**SOURCE CODE:**

import java.util.\*;

// Product model

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 String toString() {

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

}

}

// Inventory system using HashMap

class Inventory {

private Map<String, Product> products = new HashMap<>();

// Add or update product

public void addOrUpdateProduct(Product product) {

products.put(product.productId, product);

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

}

// Delete product

public void deleteProduct(String productId) {

if (products.containsKey(productId)) {

products.remove(productId);

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

} else {

System.out.println("Product not found: " + productId);

}

}

// Display all products

public void displayAllProducts() {

if (products.isEmpty()) {

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

} else {

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

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

System.out.println(p);

}

}

}

}

// Main class to test

public class Main {

public static void main(String[] args) {

Inventory inventory = new Inventory();

// Adding products

inventory.addOrUpdateProduct(new Product("P001", "Laptop", 10, 1500.00));

inventory.addOrUpdateProduct(new Product("P002", "Mouse", 50, 20.00));

// Updating product

inventory.addOrUpdateProduct(new Product("P002", "Mouse", 45, 18.00));

// Deleting a product

inventory.deleteProduct("P001");

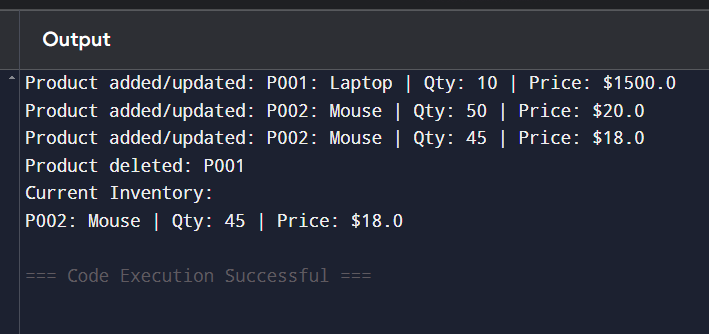
// Display all products

inventory.displayAllProducts();

}

}

**OUTPUT:**

****

**Step 4:** Time Complexity Analysis.

| **Operation** | **Data Structure Used** | **Time Complexity** |
| --- | --- | --- |
| Add/Update | HashMap | **O(1)** average |
| Delete | HashMap | **O(1)** average |
| Display All | Iteration over values | **O(n)** |

**EXERCISE-2:** E-commerce Platform Search Function.

**PROBLEM STATEMENT:**

An e-commerce platform wants to implement a basic product search function. The platform has a list of products (by name), and the user can input a keyword to search. The search function should return all products that contain the keyword (case-insensitive).

**REQUIREMENTS:**

* Create a list of product names (at least 5 items).
* Accept a search keyword from the user.
* Print all products that match the keyword.
* If no products match, display “No products found.”

**SOURCE CODE:**

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

public class EcommerceSearch {

public static void main(String[] args) {

// Step 1: List of product names

List<String> products = new ArrayList<>();

products.add("Wireless Mouse");

products.add("Gaming Keyboard");

products.add("Bluetooth Headphones");

products.add("Smartphone Case");

products.add("USB-C Charger");

// Step 2: Input search keyword

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a search keyword: ");

String keyword = scanner.nextLine().toLowerCase();

// Step 3: Search logic

boolean found = false;

System.out.println("Search Results:");

for (String product : products) {

if (product.toLowerCase().contains(keyword)) {

System.out.println("- " + product);

found = true;

}

}

// Step 4: If no results

if (!found) {

System.out.println("No products found.");

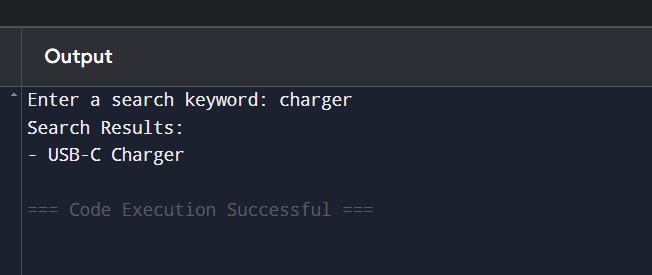
}

scanner.close();

}

}

**OUTPUT:**

****

**EXERCISE-3:** Sorting Customer Orders.

**PROBLEM STATEMENT:**

**Step 1:** Understand Sorting Algorithms.

**Common Sorting Algorithms:**

| **Algorithm** | **Best Time** | **Worst Time** | **Space** | **Notes** |
| --- | --- | --- | --- | --- |
| **Bubble Sort** | O(n) | O(n²) | O(1) | Simple, inefficient for large data |
| **Insertion Sort** | O(n) | O(n²) | O(1) | Good for small or nearly sorted lists |
| **Quick Sort** | O(n logn) | O(n²) | O(logn) | Fast in practice, recursive |
| **Merge Sort** | O(n logn) | O(n log n) | O(n) | Stable, good for linked lists |

**Quick Sort** is generally preferred for performance; **Bubble Sort** is only useful for educational purposes or very small data.

**Step 2–3:** Java Code Implementation.

import java.util.\*;

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;

}

}

class OrderSorter {

// Bubble Sort

public static void bubbleSort(List<Order> orders) {

int n = orders.size();

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

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

if (orders.get(j).totalPrice > orders.get(j + 1).totalPrice) {

// Swap

Order temp = orders.get(j);

orders.set(j, orders.get(j + 1));

orders.set(j + 1, temp);

}

}

}

}

// Quick Sort

public static void quickSort(List<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(List<Order> orders, int low, int high) {

double pivot = orders.get(high).totalPrice;

int i = low - 1;

for (int j = low; j < high; j++) {

if (orders.get(j).totalPrice <= pivot) {

i++;

// Swap

Order temp = orders.get(i);

orders.set(i, orders.get(j));

orders.set(j, temp);

}

}

// Swap pivot

Order temp = orders.get(i + 1);

orders.set(i + 1, orders.get(high));

orders.set(high, temp);

return i + 1;

}

}

public class Main {

public static void main(String[] args) {

List<Order> orders = new ArrayList<>(Arrays.asList(

new Order("O1001", "Alice", 120.50),

new Order("O1002", "Bob", 540.00),

new Order("O1003", "Charlie", 75.25),

new Order("O1004", "Diana", 320.00)

));

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

printOrders(orders);

// Bubble Sort

List<Order> bubbleSortedOrders = new ArrayList<>(orders);

OrderSorter.bubbleSort(bubbleSortedOrders);

System.out.println("\nOrders Sorted by Bubble Sort (Ascending):");

printOrders(bubbleSortedOrders);

// Quick Sort

List<Order> quickSortedOrders = new ArrayList<>(orders);

OrderSorter.quickSort(quickSortedOrders, 0, quickSortedOrders.size() - 1);

System.out.println("\nOrders Sorted by Quick Sort (Ascending):");

printOrders(quickSortedOrders);

}

private static void printOrders(List<Order> orders) {

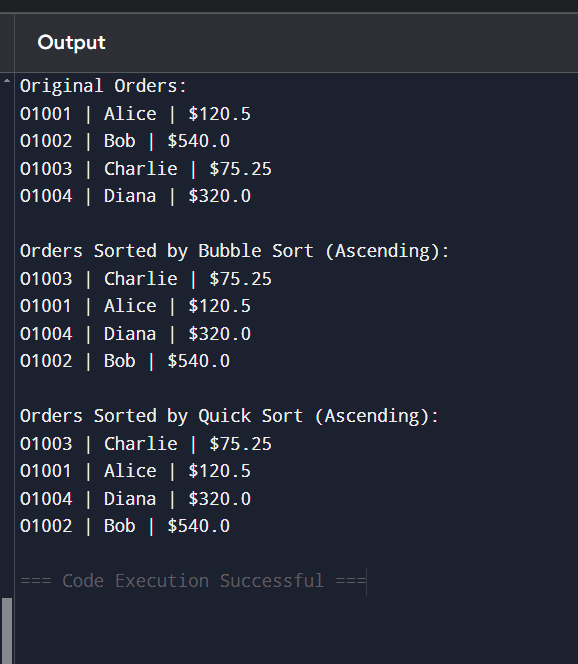
for (Order o : orders) {

System.out.println(o);

}

}

**OUTPUT:**

****

**Step 4:** Performance Analysis.

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

**EXERCISE- 4:** Employee Management System.

**STEP- 1:** Understanding Array Representation.

**How Arrays Work in Memory**

* Arrays are **contiguous blocks of memory** where elements are stored sequentially.
* Accessing an element by index is **O(1)** because the address is computed directly:  
  address = base + index × element\_size

**Advantages of Arrays:**

* Fast random access using index
* Simple and memory-efficient for fixed-size data
* Best for dense collections where order matters

**STEP 2-3:** Java Implementation Using Arrays.

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;

}

}

class EmployeeManagement {

private Employee[] employees;

private int size;

public EmployeeManagement(int capacity) {

employees = new Employee[capacity];

size = 0;

}

// Add employee

public void addEmployee(Employee e) {

if (size < employees.length) {

employees[size++] = e;

System.out.println("Added: " + e);

} else {

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

}

}

// Search employee by ID

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

// Traverse all employees

public void listEmployees() {

System.out.println("Employee List:");

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

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

}

}

// Delete employee by ID

public void deleteEmployee(int id) {

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

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

// Shift elements left

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

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

}

employees[--size] = null;

System.out.println("Deleted employee with ID: " + id);

return;

}

}

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

}

}

// Main class to test

public class Main {

public static void main(String[] args) {

EmployeeManagement em = new EmployeeManagement(5);

em.addEmployee(new Employee(101, "Alice", "Manager", 80000));

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

em.addEmployee(new Employee(103, "Charlie", "Analyst", 55000));

em.listEmployees();

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

Employee result = em.searchEmployee(102);

if (result != null) {

System.out.println("Found: " + result);

} else {

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

}

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

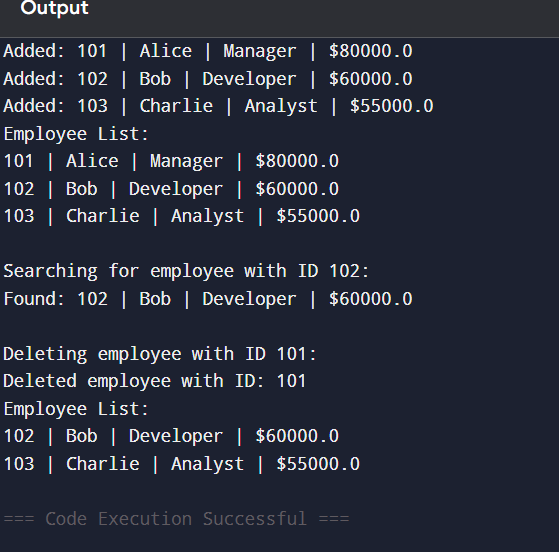
em.deleteEmployee(101);

em.listEmployees();

}

}

**OUTPUT:**

****

**STEP-4:** Time Complexity Analysis.

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| Add | O(1) | If not full |
| Search | O(n) | Linear scan by ID |
| Traverse | O(n) | Visit each employee |
| Delete | O(n) | Search + shift all right elements left |

**EXERCISE-5:** Task Management System.

**STEP-1:** Understand Linked Lists.

**Types of Linked Lists:**

| **Type** | **Description** |
| --- | --- |
| Singly Linked List | Each node points to the next node only |
| Doubly Linked List | Each node points to both next and previous nodes |

**Singly Linked List (used here)**

* Good for insertion and deletion at the head or tail.
* Uses less memory than doubly linked lists.

**STEP-2,3:** Java Implementation – Singly Linked List for Tasks.

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;

}

}

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

class TaskLinkedList {

private TaskNode head;

// Add task at the end

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

System.out.println("Added: " + task);

}

// Search by taskId

public Task searchTask(int taskId) {

TaskNode current = head;

while (current != null) {

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

return current.task;

}

current = current.next;

}

return null;

}

// Delete task by taskId

public void deleteTask(int taskId) {

if (head == null) return;

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

head = head.next;

System.out.println("Deleted task ID: " + taskId);

return;

}

TaskNode current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

System.out.println("Deleted task ID: " + taskId);

} else {

System.out.println("Task ID not found: " + taskId);

}

}

// Traverse all tasks

public void listTasks() {

TaskNode current = head;

if (current == null) {

System.out.println("Task list is empty.");

return;

}

System.out.println("Task List:");

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

}

public class Main {

public static void main(String[] args) {

TaskLinkedList taskList = new TaskLinkedList();

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

taskList.addTask(new Task(2, "Implement API", "In Progress"));

taskList.addTask(new Task(3, "Write Documentation", "Pending"));

taskList.listTasks();

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

Task found = taskList.searchTask(2);

if (found != null) {

System.out.println("Found: " + found);

} else {

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

}

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

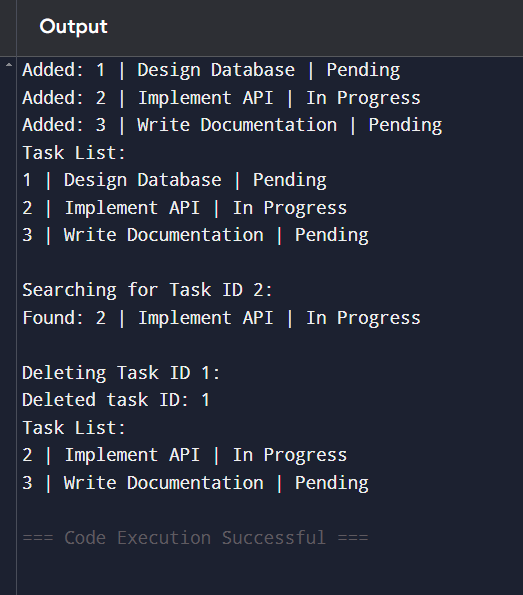
taskList.deleteTask(1);

taskList.listTasks();

}

}

**OUTPUT:**

****

**STEP-4:** Time Complexity Analysis.

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| Add | O(n) | Traverse to end unless using tail pointer |
| Search | O(n) | Must scan each node |
| Delete | O(n) | Must find node and re-link pointers |
| Traverse | O(n) | Visit each node |

**EXERSISE-6:** Library Management System.

**STEP-1:** Understand Search Algorithms.

**Linear Search**

* Checks **each element one-by-one**.
* Does **not require sorting**.
* Use when data is **unsorted** or list is **small**.

**Time Complexity**:

* Best: O(1)
* Worst: O(n)
* Average: O(n)

**Binary Search**

* Divides the list into halves recursively.
* Requires data to be **sorted**.
* Much faster for **large sorted lists**.

**Time Complexity**:

* Best: O(1)
* Worst: O(log n)
* Average: O(log n)

**STEP-2,3:** Java Implementation.

import java.util.\*;

class Book implements Comparable<Book> {

int bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title.toLowerCase(); // normalize for consistent search

this.author = author;

}

public String toString() {

return bookId + " | " + title + " | " + author;

}

@Override

public int compareTo(Book other) {

return this.title.compareTo(other.title);

}

}

public class Main {

// Linear Search by title

public static Book linearSearch(List<Book> books, String title) {

for (Book book : books) {

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

return book;

}

}

return null;

}

// Binary Search by title

public static Book binarySearch(List<Book> sortedBooks, String title) {

int left = 0, right = sortedBooks.size() - 1;

title = title.toLowerCase();

while (left <= right) {

int mid = (left + right) / 2;

int cmp = sortedBooks.get(mid).title.compareTo(title);

if (cmp == 0)

return sortedBooks.get(mid);

else if (cmp < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

public static void main(String[] args) {

List<Book> books = new ArrayList<>(Arrays.asList(

new Book(101, "Java Programming", "James Gosling"),

new Book(102, "Data Structures", "Robert Lafore"),

new Book(103, "Operating Systems", "Andrew Tanenbaum"),

new Book(104, "Algorithms", "Thomas Cormen")

));

System.out.println("Original Book List:");

for (Book b : books) System.out.println(b)

// Linear Search

System.out.println("\nLinear Search for 'Operating Systems':");

Book found1 = linearSearch(books, "Operating Systems");

System.out.println(found1 != null ? "Found: " + found1 : "Book not found");

// Binary Search

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

Collections.sort(books); // Binary search requires sorted list

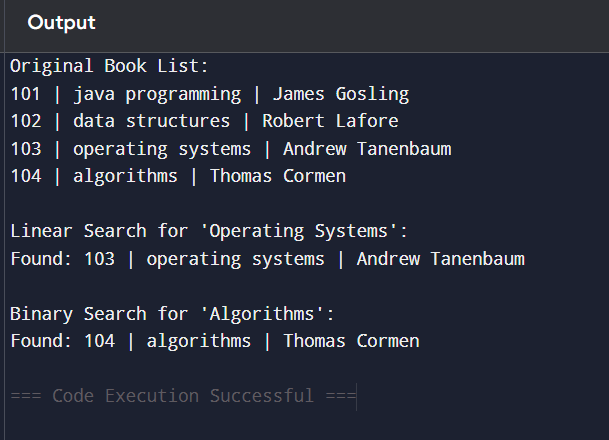
Book found2 = binarySearch(books, "Algorithms");

System.out.println(found2 != null ? "Found: " + found2 : "Book not found");

}

}

**OUTPUT:**

****

**STEP-4:** Complexity & Analysis

| **Operation** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Best Case | O(1) | O(1) |
| Average | O(n) | O(log n) |
| Worst Case | O(n) | O(log n) |
| Requires Sorting | ❌ No | ✅ Yes |

**EXERCUSE-7:**  Financial Forecasting.

**PROBLEM STATEMENT:**

We are developing a financial forecasting module for a business. The goal is to predict the total savings over a given number of months based on:

* Initial investment amount
* Monthly contribution
* Expected monthly growth rate (as a percentage)
* Forecast period (in months)

We need to calculate and display the projected savings at the end of each month using compound growth.

**FORMULA:**

For each month, the new amount is calculated as:

Total = (Previous Total + Monthly Contribution) \* (1 + Growth Rate/100)

**REQUIREMENTS:**

* Accept the following inputs:

1. Initial investment amount
2. Monthly contribution
3. Monthly growth rate (in %)
4. Number of months

* Output the savings at the end of each month.
* Round results to 2 decimal places.

**STEP-1:** Understand Recursive Algorithms

**What is Recursion?**

Recursion is when a method calls itself to solve a smaller version of the same problem.

**Why Use Recursion?**

* Simplifies problems that have repeated patterns, especially where results depend on previous computations (like forecasting, Fibonacci, etc.)

**STEP-2:** Setup

We’ll forecast future values using the formula:

FV(n) = FV(n-1) \* (1 + growthRate)

Where:

* FV(n) = value in year n
* growthRate = annual growth rate (e.g., 5% = 0.05)

**STEP- 3:** Recursive Implementation in Java.

import java.util.HashMap;

public class FinancialForecastingMemo {

static HashMap<Integer, Double> memo = new HashMap<>();

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

if (year == 0) return currentValue;

if (memo.containsKey(year)) return memo.get(year);

double value = forecastValue(year - 1, currentValue, growthRate) \* (1 + growthRate);

memo.put(year, value);

return value;

}

public static void main(String[] args) {

double initialValue = 1000;

double growthRate = 0.08;

int years = 10;

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

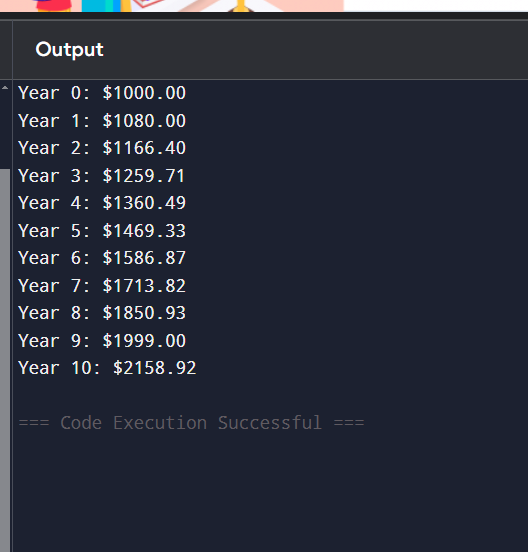
System.out.printf("Year %d: $%.2f%n", i, forecastValue(i, initialValue, growthRate));

}

}

}

**OUTPUT:**

****

**STEP-4:** Analysis

**Time Complexity:**

* Without optimization: O(n) — each call goes one level deeper.
* With memoization: O(n) with saved results, avoiding recomputation.

**Drawbacks of Plain Recursion:**

* Deep recursion can lead to stack overflow for large n.
* Recomputes values multiple times unless optimized.