**Week 1 – Algorithms and Data Structures**

Question 1:

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

**Scenario:**

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

**Steps:**

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

Understanding Asymptotic Notation (Big O):

Big O notation describes the performance (efficiency) of an algorithm as the input size (n) grows.

* It helps answer: *"How much time/memory will this algorithm need for large data?"*
* Binary Search is much faster but only works on sorted arrays.

Search Operation Time Complexities:

| Algorithm | Best Case | Average Case | Worst Case |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

CODE:

package Ecommerce;

**Product.java:**

//Represents a product in the e-commerce platform

public class Product {

int productId; // Unique identifier for the product

String productName; // Name of the product (e.g., "iPhone 14")

String category; // Category (e.g., "Mobile", "Electronics")

// Constructor to initialize product details

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

// Display product information

public void display() {

System.out.println("ID: " + productId + ", Name: " + productName + ", Category: " + category);

}

}

**SearchEngine.java:**

**package** Ecommerce;

//This class contains both Linear Search and Binary Search implementations

**public** **class** SearchEngine {

// Linear search algorithm - checks each item one by one

**public** **static** **int** linearSearch(Product [] products, String targetName) {

// Loop through the entire list of products

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

// Check if the product name matches the target

**if** (products[i].productName.equalsIgnoreCase(targetName)) {

**return** i; // Return the index if found

}

}

**return** -1; // Not found

}

// Binary search algorithm - much faster, but needs sorted array

**public** **static** **int** binarySearch(Product[] products, String targetName) {

**int** left = 0;

**int** right = products.length - 1;

// Loop until the search space is exhausted

**while** (left <= right)

{

// Find the middle index

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

// Compare target with middle product

**int** comparison products[mid].productName.compareToIgnoreCase(targetName);

**if** (comparison == 0)

{

**return** mid; // Match found

}

**else** **if** (comparison < 0) {

left = mid + 1; // Search in the right half

}

**else**

{

right = mid - 1; // Search in the left half

}

}

**return** -1; // Not found

}

}

**ECommercePlatform.java:**

**package** Ecommerce;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** ECommercePlatform {

**public** **static** **void** main(String[] args) {

// Sample product list (can scale to millions in real apps)

Product[] productList = {

**new** Product(101, "iPhone 14", "Mobiles"),

**new** Product(102, "Samsung Galaxy", "Mobiles"),

**new** Product(103, "AirPods Pro", "Audio"),

**new** Product(104, "Sony WH-1000XM5", "Audio"),

**new** Product(105, "OnePlus Nord", "Mobiles"),

};

System.***out***.println("Searching for 'AirPods Pro' using Linear Search:");

// Perform linear search

**int** index1 = SearchEngine.*linearSearch*(productList, "AirPods Pro");

**if** (index1 != -1) {

productList[index1].display(); // Show found product

} **else** {

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

}

// Before binary search, sort the array by productName (required!)

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

System.***out***.println("\n Searching for 'AirPods Pro' using Binary Search:");

// Perform binary search

**int** index2 = SearchEngine.*binarySearch*(productList, "AirPods Pro");

**if** (index2 != -1) {

productList[index2].display(); // Show found product

} **else** {

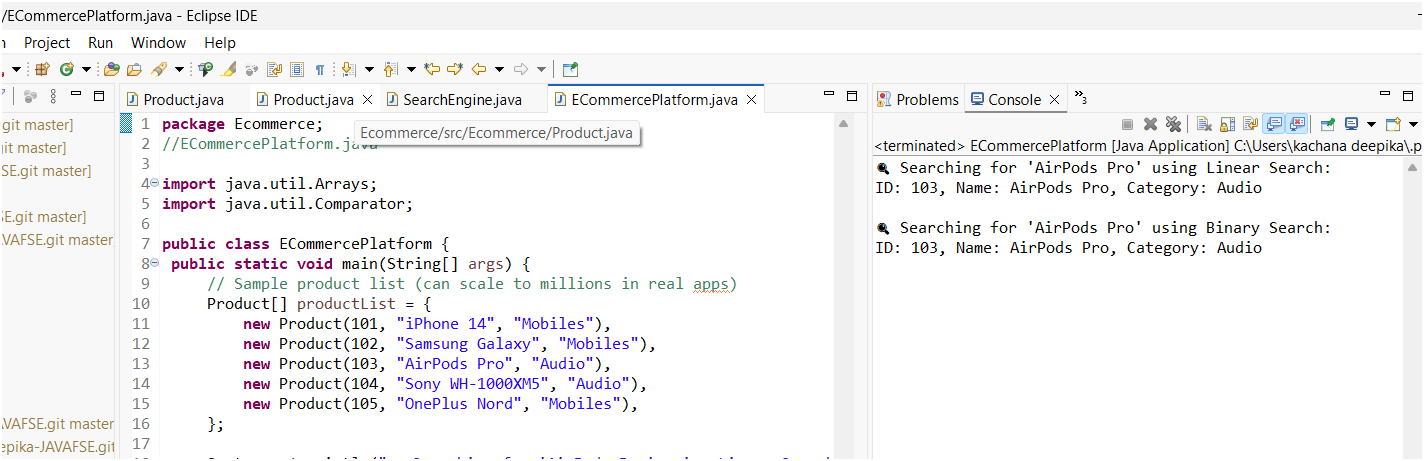
System.***out***.println("Product not found.");

}

}

}

**OUTPUT:**



Question 2:

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

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

**1. Understanding Recursive Algorithm:**

Recursion is a programming technique where a method (or function) calls itself to solve a problem.

* It breaks a big problem into smaller problems.
* The function keeps calling itself with smaller inputs, until it reaches a point where it stops — called the base case.

| **Part** | **What It Does** |
| --- | --- |
| **Base Case** | Stops recursion. Without it → infinite loop |
| **Recursive Case** | Keeps calling itself with smaller input |

**Forecasting Problem:**

This program that **predicts the future value** of an investment given:

* Initial amount
* Annual growth rate (percentage)
* Number of years

**Recursive Formula:**

We have:

* Initial amount A
* Growth rate r (like 5%)
* Time n years

Then future value:

CopyEdit

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

**CODE:**

**package** FinancialForcasting;

**public** **class** FinancialForecast

{

// This is the recursive method to calculate the future value

**public** **static** **double** calculateFutureValue(**double** amount, **double** rate, **int** years) {

// Base case: if number of years is 0, return the original amount

**if** (years == 0) {

**return** amount;

}

// Recursive case: calculate value for one year less, then apply growth

// Example: if years = 3, this becomes FV(2) \* (1 + rate)

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

}

**public** **static** **void** main(String[] args) {

// Initial investment amount

**double** initialAmount = 10000; // ₹10,000

// Annual growth rate (5% written as decimal)

**double** annualGrowthRate = 0.05;

// Number of years to forecast

**int** forecastYears = 5;

// Call the recursive method to calculate the future value

**double** futureValue = *calculateFutureValue*(initialAmount, annualGrowthRate, forecastYears);

// Display the result using printf for formatted currency output

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

}

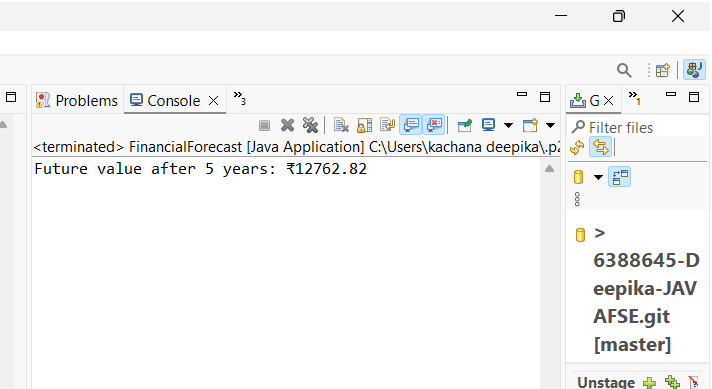
}

**Analysis of Time Complexity:**

Each call leads to **1 more recursive call**, until the year hits 0

T(n) = T(n - 1) + O(1) => Time complexity = O(n)

**OUTPUT:**

****