**Module 2 - Data Structures and Algorithms**

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

**Understand Asymptotic Notation:**

Explain Big O notation and how it helps in analysing algorithm**.**

**ANS:**

Big O notation describes the time or space complexity of an algorithm as a function of input size (n). It helps in evaluating how efficiently an algorithm performs as the data size grows.

Why It Helps?

It lets developers choose the most efficient algorithm by comparing performance in different scenarios (best, average, worst case).

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

**ANS:**

| **Case** | **Description** |
| --- | --- |
| Best Case | The item is found at the beginning (e.g., index 0) |
| Average Case | The item is somewhere in the middle |
| Worst Case | The item is not found or at the end |

**IMPLEMENTATION:**

CODE:

package com.ecommerce.search;

import java.util.Arrays;

import java.util.Scanner;

class Product {

int productId;

String productName;

String category;

Product(int id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

}

public class SearchComparison {

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

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

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

return i;

}

}

return -1;

}

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

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

while (left <= right) {

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

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

if (cmp == 0)

return mid;

else if (cmp < 0)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

public static void main(String[] args) {

Product[] originalProducts = {

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

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

new Product(3, "Watch", "Accessories"),

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

new Product(5, "T-Shirt", "Clothing")

};

Product[] sortedProducts = Arrays.copyOf(originalProducts,originalProducts.length);

Arrays.sort(sortedProducts, (a, b) -> a.productName.compareToIgnoreCase(b.productName));

Scanner scanner = new Scanner(System.in);

System.out.print("Enter product name to search: ");

String searchKey = scanner.nextLine();

int linearIndex = linearSearch(originalProducts, searchKey);

System.out.println("\nUsing Linear Search:");

if (linearIndex != -1)

System.out.println("Product found at index " + linearIndex + ": " + originalProducts[linearIndex].productName);

else

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

int binaryIndex = binarySearch(sortedProducts, searchKey);

System.out.println("\nUsing Binary Search:");

if (binaryIndex != -1)

System.out.println("Product found at index " + binaryIndex + ": " + sortedProducts[binaryIndex].productName);

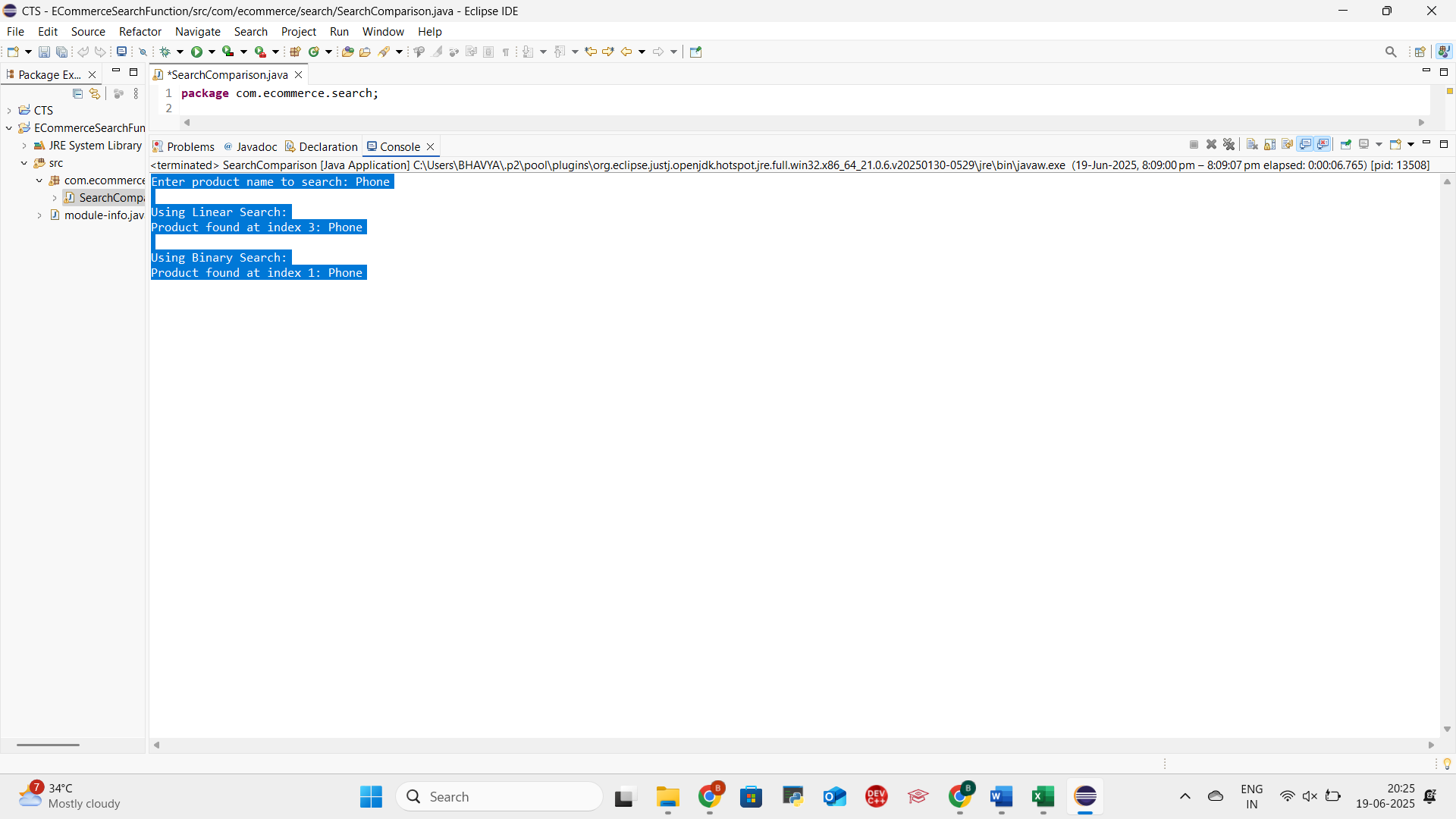
else

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

}

}

OUTPUT SCREENSHOT:

****

**ANALYSIS:**

Compare the time complexity of linear and binary search algorithms.

**ANS:**

| Algorithm | Time Complexity |
| --- | --- |
| Linear Search | Best: O(1), Worst: O(n) |
| Binary Search | Best: O(1), Worst: O(log n) |
|  |  |

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

**ANS:**

Binary Search is faster but works only on sorted data. Linear Search is simpler but slower for large datasets. In an e-commerce platform with many products, Binary Search is more efficient after sorting the list.

**Exercise 7: Financial Forecasting**

**Understand Recursive Algorithms:**

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

**ANS:**

Recursion is a programming technique where a method calls itself to solve a problem. It breaks down a large problem into smaller sub-problems, each of which is a smaller version of the original.

Each recursive call moves closer to a base case, which ends the recursion.

A recursive function has two parts:

1. Base Case – when to stop the recursion
2. Recursive Case – the part where the function calls itself

Recursion makes complex problems:

* Easier to express logically
* More readable and elegant for tasks involving repetition or branching
* Ideal for problems like:
  + Factorial calculation
  + Fibonacci series
  + Tree traversals
  + Searching/sorting (like quicksort/mergeSort)
  + Forecasting and predictions (like this task!)

**Problem Setup**

We will write a **recursive function** that predicts future value based on:

* Current value
* Annual growth rate
* Number of years

**Implementation**

CODE:

package com.module2.forecasting;

import java.util.Scanner;

public class FinancialForecast {

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

if (years == 0)

return currentValue;

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

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.*in*);

System.*out*.print("Enter current investment value (e.g., 10000): ");

double currentValue = scanner.nextDouble();

System.*out*.print("Enter annual growth rate in percentage (e.g., 10 for 10%): ");

double ratePercent = scanner.nextDouble();

double rate = ratePercent / 100;

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

int years = scanner.nextInt();

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

System.*out*.printf("\nThe predicted future value after %d years is: %.2f\n", years, futureValue);

}

}

OUTPUT SCREENSHOT:

A screenshot of a computer

AI-generated content may be incorrect.

**Analysis:**

Discuss the time complexity of your recursive algorithm.

**ANS:**

Time Complexity: O(n)

* The method makes one recursive call per year.
* If years = n, the method will execute n recursive calls until it hits the base case (years == 0).
* Therefore, the total time complexity is:

T(n) = T(n-1) + O(1) → O(n)

Example:

If the user wants to forecast for 10 years, the recursion depth will be 10 — meaning 10 recursive calls will be made.

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

**ANS:**

Recursive solutions are elegant but can become inefficient when:

* The recursion depth is high
* The same subproblems are recalculated repeatedly
* There's a risk of stack overflow due to deep recursion

To optimize, we can use the following techniques:

1. Convert Recursion to Iteration

For problems like financial forecasting — which involve simple repeated calculations — recursion can be replaced with a loop to avoid extra function calls and stack usage.

2. Memoization (for overlapping subproblems)

If the problem has overlapping recursive calls (like Fibonacci), you can store results of subproblems in a cache (array or map) to avoid recalculating.

This is not needed for this particular forecasting problem, but it’s good to know when dealing with more complex recursive problems.

The best way to optimize this financial forecasting recursion is to convert it to an iterative approach, which saves memory and avoids stack overflow.