**Data Structures and Algorithms**

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

1. **Asymptotic Notation:**

**What is Big O Notation?**

**Big O notation** is a mathematical way to describe the **efficiency of an algorithm** in terms of time or space complexity. It shows how the performance of an algorithm changes **as the input size (n) increases**.

* It focuses on the **growth rate** and **ignores constant factors** and lower-order terms.
* It gives us an **upper bound**, i.e., the worst-case time an algorithm could take.
* Helps in comparing algorithms objectively, regardless of programming language or hardware.

**Why is Big O Important?**

* To choose the **right algorithm** that performs efficiently as data grows.
* To avoid algorithms that might work for small input but **fail on large data**.
* To ensure **scalability** of applications like e-commerce platforms, where thousands or millions of items are searched.

**Best, Average, and Worst-Case Scenarios for Search Operations**

When analyzing search algorithms like **linear search** and **binary search**, we examine three scenarios:

**1. Best Case**

* **Definition**: The item is found in the **first attempt**.
* **Linear Search**: O(1) → The item is at the first position.
* **Binary Search**: O(1) → The item is exactly in the middle.

**2. Average Case**

* **Definition**: The item is somewhere in the **middle or randomly placed**.
* **Linear Search**: O(n/2) ≈ O(n)
* **Binary Search**: O(log n)

**3. Worst Case**

* **Definition**: The item is at the **last position** or **not present** at all.
* **Linear Search**: O(n)
* **Binary Search**: O(log n)

1. **Setup:**

// Product.java

class Product {

private final int productId;

private final String productName;

private final String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

@Override

public String toString() {

return productId + " | " + productName + " (" + category + ")";

}

}

1. **Implementation:**

import java.util.Arrays;

import java.util.Comparator;

class Main {

public static void main(String[] args) {

Product[] catalogue = {

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

new Product(102, "T-Shirt", "Apparel"),

new Product(103, "Headphone", "Electronics"),

new Product(104, "Coffee", "Grocery"),

new Product(105, "Shoes", "Footwear")

};

**/\* 1. Linear search on the unsorted array \*/**

System.out.println("Linear → " +

linearSearch(catalogue, "coffee")); // O(n)

**/\* 2. Binary search: first sort by productName \*/**

Arrays.sort(catalogue, Comparator.comparing(

p -> p.getProductName().toLowerCase()));

System.out.println("Binary → " +

binarySearch(catalogue, "Headphone")); // O(log n)

}

**/\* ---------- LINEAR SEARCH (unsorted array) ---------- \*/**

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

for (Product p : products) {

if (p.getProductName().equalsIgnoreCase(needle)) {

return p; // O(1) best, O(n) worst

}

}

return null; // not found

}

**/\* ---------- BINARY SEARCH (sorted array) ---------- \*/**

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

int low = 0, high = products.length - 1;

while (low <= high) { // O(log n)

int mid = (low + high) >>> 1; // unsigned right-shift avoids overflow

int cmp = products[mid]

.getProductName()

.compareToIgnoreCase(needle);

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

if (cmp < 0) low = mid + 1; // search right half

else high = mid - 1; // search left half

}

return null;

}

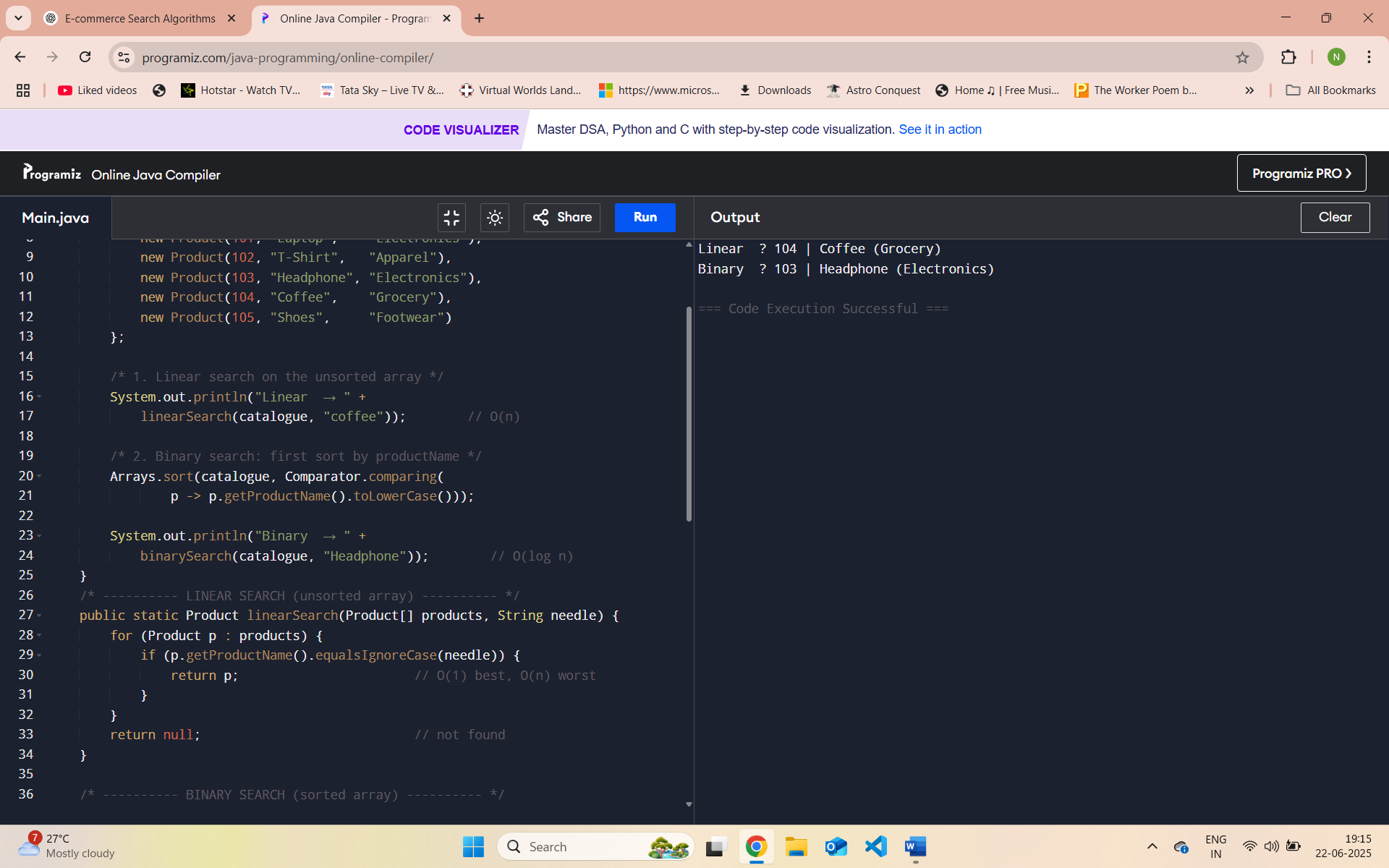
}

1. **Analysis:**

| **Feature** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Time Complexity** | O(n) | O(log n) |
| **Best Case** | O(1) (first element match) | O(1) (middle element match) |
| **Average Case** | O(n) | O(log n) |
| **Worst Case** | O(n) (not found or last item) | O(log n) (not found) |

**Binary search is the better choice** for an e-commerce platform because it is **much faster** for large product lists and provides a **scalable, efficient search experience** when the data is sorted.

1. **Output**



**Exercise 7: Financial Forecasting**

* + 1. **Understand Recursive Algorithms:**

**Recursion** is a programming technique where a function **calls itself** in order to solve a problem by breaking it into smaller subproblems.

A recursive function usually has:

1. **Base case** – the simplest instance where the function can return a result directly (no further recursion).
2. **Recursive case** – the function calls itself with a smaller or simpler input.

**How Recursion Simplifies Problems**

Recursion is useful when:

* The problem has a repeating structure (e.g., tree, graph, sequence).
* You want to avoid complex loops and make the code cleaner and shorter.
* Problems naturally break down into smaller subproblems, such as:
  + Mathematical computations (e.g., factorial, power)
  + Data structure traversal (e.g., trees, graphs)
  + Divide-and-conquer algorithms (e.g., mergesort, quicksort)
  + Financial forecasting, where each year builds upon the result of the previous on

**2.Implementation:**

class Main {

**// Set up**

public static double futureValueRecursive(double currentValue,

double[] growthRates,

int yearsProcessed) {

if (yearsProcessed == growthRates.length) { // base case

return currentValue;

}

double nextValue = currentValue \* (1 + growthRates[yearsProcessed]);

return futureValueRecursive(nextValue, growthRates, yearsProcessed + 1);

}

public static void main(String[] args) {

double presentValue = 10\_000.0; // initial amount ₹10 000

double[] growthRates = { 0.08, 0.06, 0.07,

0.05, 0.09 }; // five-year plan

double forecast = futureValueRecursive(presentValue,

growthRates,

0); // start at year 0

System.out.printf("Predicted value after %d years: ₹%.2f%n",

growthRates.length, forecast);

}

}

1. **Time Complexity and optimization**

* The algorithm processes each year **once**.
* If n is the number of years (i.e., length of growth Rates array), the method performs:
  + **n multiplications**
  + **n recursive calls**
* So the **time complexity is linear**, or:

Time Complexity=O(n)

**How to Optimize a Recursive Solution to Avoid Excessive Computation**

In recursion, **excessive computation** usually occurs due to:

1. **Too many repeated calls** (redundant work),
2. **Deep call stacks** (memory overload),
3. **No memoization or base-case limits**.

**Use an Iterative Approach Instead**

Instead of using recursion, you can use a simple loop to compute the same result without using the call stack:

public static double futureValueIterative(double presentValue, double[] growthRates) {

for (double rate : growthRates) {

presentValue \*= (1 + rate);

}

return presentValue;

}

1. **Output**

A computer screen shot of a computer screen

AI-generated content may be incorrect.