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

**1. Understanding Asymptotic Notation**

**Big O Notation**

Big O notation is a mathematical representation used to describe the **time or space complexity** of an algorithm in relation to the size of the input data (n). It helps analyze how an algorithm behaves as the input grows, without focusing on the specific hardware or programming language.

For example:

* O(1) means constant time.
* O(n) means time increases linearly with input size.
* O(log n) means time increases logarithmically.

This is especially useful in search operations, where performance depends on how efficiently we can find a specific item from a list of elements.

**Best, Average, and Worst-Case Scenarios**

In search algorithms, we analyze three performance cases:

| **Case** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best** | O(1) – First item match | O(1) – Midpoint match |
| **Average** | O(n/2) → O(n) | O(log n) |
| **Worst** | O(n) – Last item match | O(log n) – Keeps halving |

**2. Setup: Product Class for Search**

We define a class Product to store relevant details about items on an e-commerce platform. These attributes are essential for performing search operations.

public class Product {

private int productId;

private String productName;

private 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; }

}

**3. Implementation of Search Algorithms**

**Linear Search (Unsorted Array)**

In linear search, each element is checked one by one until a match is found.

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

for (Product product : products) {

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

return product;

}

}

return null;

}

**Binary Search (Sorted Array)**

Binary search is more efficient but requires the array to be sorted by product name.

import java.util.Arrays;

import java.util.Comparator;

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

Arrays.sort(products, Comparator.comparing(Product::getProductName, String.CASE\_INSENSITIVE\_ORDER));

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

while (left <= right) {

int mid = (left + right) / 2;

String midName = products[mid].getProductName();

int cmp = productName.compareToIgnoreCase(midName);

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

else if (cmp < 0) right = mid - 1;

else left = mid + 1;

}

return null;

}

**4. Analysis and Comparison**

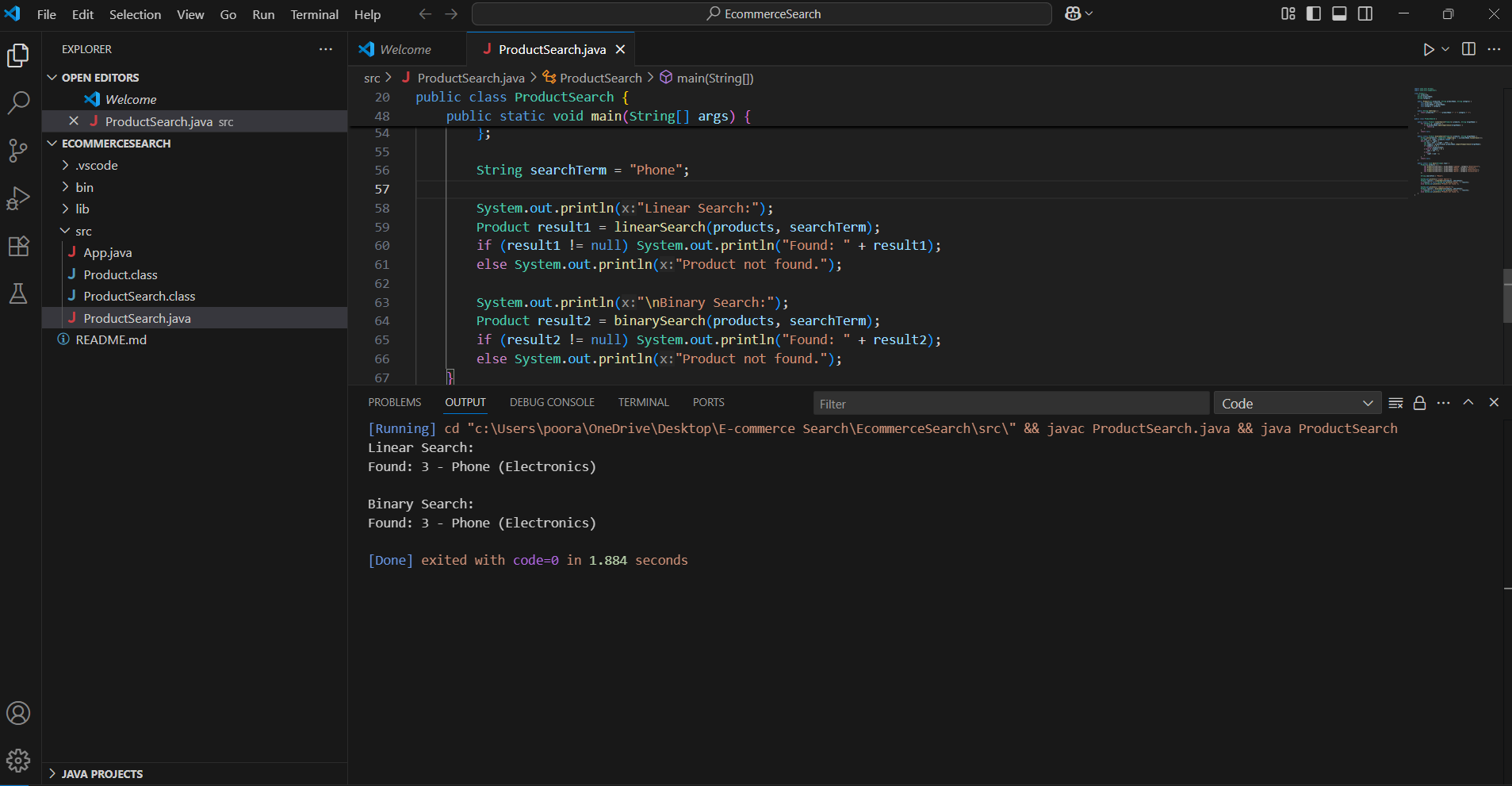
**Time Complexity Comparison**

| **Algorithm** | **Time Complexity** | **Sorted Required?** | **Efficiency** |
| --- | --- | --- | --- |
| Linear Search | O(n) | ❌ No | Slower for large datasets |
| Binary Search | O(log n) | ✅ Yes | Faster, scalable |

**Which is More Suitable?**

For a modern e-commerce platform:

* **Binary Search** is more suitable if the product catalog is large and mostly static or pre-sorted.
* **Linear Search** works fine for small datasets or dynamically changing content but is inefficient for scalability.



**Exercise 7: Financial Forecasting**

**1. Understanding Recursive Algorithms**

**🔹 What is Recursion?**

Recursion is a problem-solving technique where a function **calls itself** to solve smaller instances of a larger problem. Each recursive call breaks down the problem into subproblems until a **base case** is reached, which stops further recursion.

**🔹 Why Use Recursion?**

Recursion is often used when a problem can be defined in terms of **smaller subproblems**, especially in scenarios such as:

* Tree traversals
* Fibonacci sequence
* Factorials
* Financial projections where future values depend on prior ones

Recursion **simplifies the logic** of such problems and makes the code more readable, although it may not always be the most efficient approach.

**2. Setup: Defining the Problem**

We aim to develop a method that calculates the **future financial value** based on the previous year’s value and a **fixed annual growth rate**.

Let:

* FV(n) be the future value at year n.
* FV(n) = FV(n-1) \* (1 + r) where r is the growth rate.
* Base case: FV(0) = initial value

**3. Implementation: Recursive Future Value Calculator**

**🔹 Java Code Example:**

public class FinancialForecast {

public static double futureValue(double initialValue, double rate, int years) {

if (years == 0) {

return initialValue;

}

return futureValue(initialValue, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

double initialValue = 10000; // ₹10,000

double rate = 0.08; // 8% annual growth

int years = 5;

double result = futureValue(initialValue, rate, years);

System.out.printf("Future value after %d years: ₹%.2f", years, result);

}

}

**4. Analysis**

**🔹 Time Complexity**

Each recursive call decreases years by 1 until it reaches 0. Hence, if n is the number of years:

* **Time Complexity**: O(n)
* **Space Complexity**: O(n) (due to function call stack)

**🔹 Drawbacks of Naive Recursion**

* **Stack Overflow**: For large values of years, the recursive stack can grow large.
* **Repetitive Calculations**: In more complex recursive financial models (e.g., variable growth), repeated subproblem computation can degrade performance.

**Optimizing the Recursive Solution**

**1. Memoization (Top-Down Dynamic Programming)**

Store already computed results in a cache (e.g., a map or array). This avoids repeated computation.

**2. Iterative Approach (Bottom-Up)**

Convert the recursive approach to an iterative one using loops.

public static double futureValueIterative(double initialValue, double rate, int years) {

double result = initialValue;

for (int i = 1; i <= years; i++) {

result \*= (1 + rate);

}

return result;

}

* Time Complexity: O(n)
* Space Complexity: O(1)

This is more efficient and avoids recursion overhead.

