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

**Big O Notation:**

Big O Notation describes how an algorithm’s time or space requirements grow as the size of input increases.

| **Big O** | **Description** |  |
| --- | --- | --- |
| O(1) | Constant time |  |
| O(n) | Linear time |  |
| O(log n) | Logarithmic time |  |
| O(n²) | Quadratic time |  |

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

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) (first item) | O(n/2) ≈ O(n) | O(n) |
| Binary Search | O(1) (middle) | O(log n) | O(log n) |

**Code:**

// Product.java

public class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

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

}

}

// LinearSearch.java

public class LinearSearch {

public static Product searchById(Product[] products, int id) {

for (Product p : products) {

if (p.productId == id) {

return p;

}

}

return null;

}

}

// BinarySearch.java

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static Product searchById(Product[] products, int id) {

// Sort the array by productId

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

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

while (left <= right) {

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

if (products[mid].productId == id) {

return products[mid];

} else if (products[mid].productId < id) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

// App.java

public class App {

public static void main(String[] args) {

Product[] products = {

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

new Product(205, "Shoes", "Footwear"),

new Product(150, "Watch", "Accessories"),

new Product(330, "Phone", "Electronics"),

new Product(410, "Bag", "Fashion")

};

// Linear Search

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

Product result1 = LinearSearch.searchById(products, 150);

System.out.println(result1 != null ? result1 : "Product not found");

// Binary Search

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

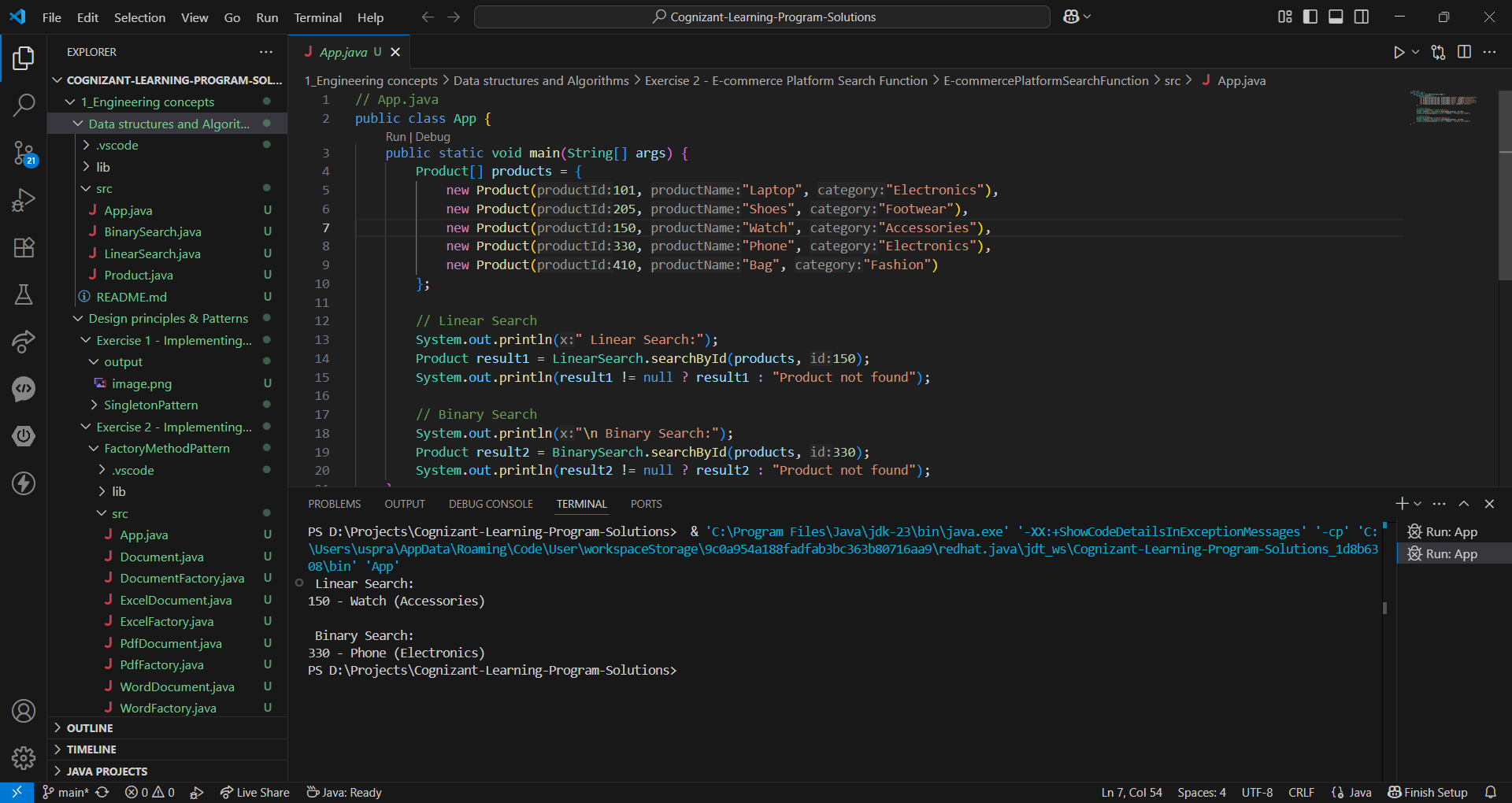
Product result2 = BinarySearch.searchById(products, 330);

System.out.println(result2 != null ? result2 : "Product not found");

}

}

**Output:**

****

**Analysis:**

| **Criteria** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Time Complexity | O(n) | O(log n) |
| Sorted Needed? | No | Yes |
| Easy to Use | Yes | Requires sorting |
| Best Use Case | Small datasets | Large datasets |

**Which is better for e-commerce?**

| **Scenario** | **Recommended Search** |
| --- | --- |
| Small number of products | Linear Search |
| Large product database | Binary Search (with sorted data or use indexing structures like Trees, HashMaps, DB) |
| Advanced needs (partial match) | Use Tries / Full-Text Search engines like Elasticsearch |

**Exercise 7: Financial Forecasting**

**Recursion:**

Recursion is a technique where a function calls itself to solve a smaller version of the original problem. It is especially useful for problems that have a repetitive or sequential nature. It is used to simplify complex problems and some of the problems for which recursion could be used are calculating factorial, generating Fibonacci numbers, tree traversal and forecasting.

**Code:**

// App.java

public class App {

public static double forecastRecursive(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

return forecastRecursive(currentValue \* (1 + growthRate), growthRate, years - 1);

}

public static double forecastIterative(double currentValue, double growthRate, int years) {

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}

public static void main(String[] args) {

double initialValue = 10000.0;

double annualGrowthRate = 0.08;

int years = 5;

double futureValueRecursive = forecastRecursive(initialValue, annualGrowthRate, years);

System.out.printf("Future value using recursion after %d years: ₹%.2f%n", years, futureValueRecursive);

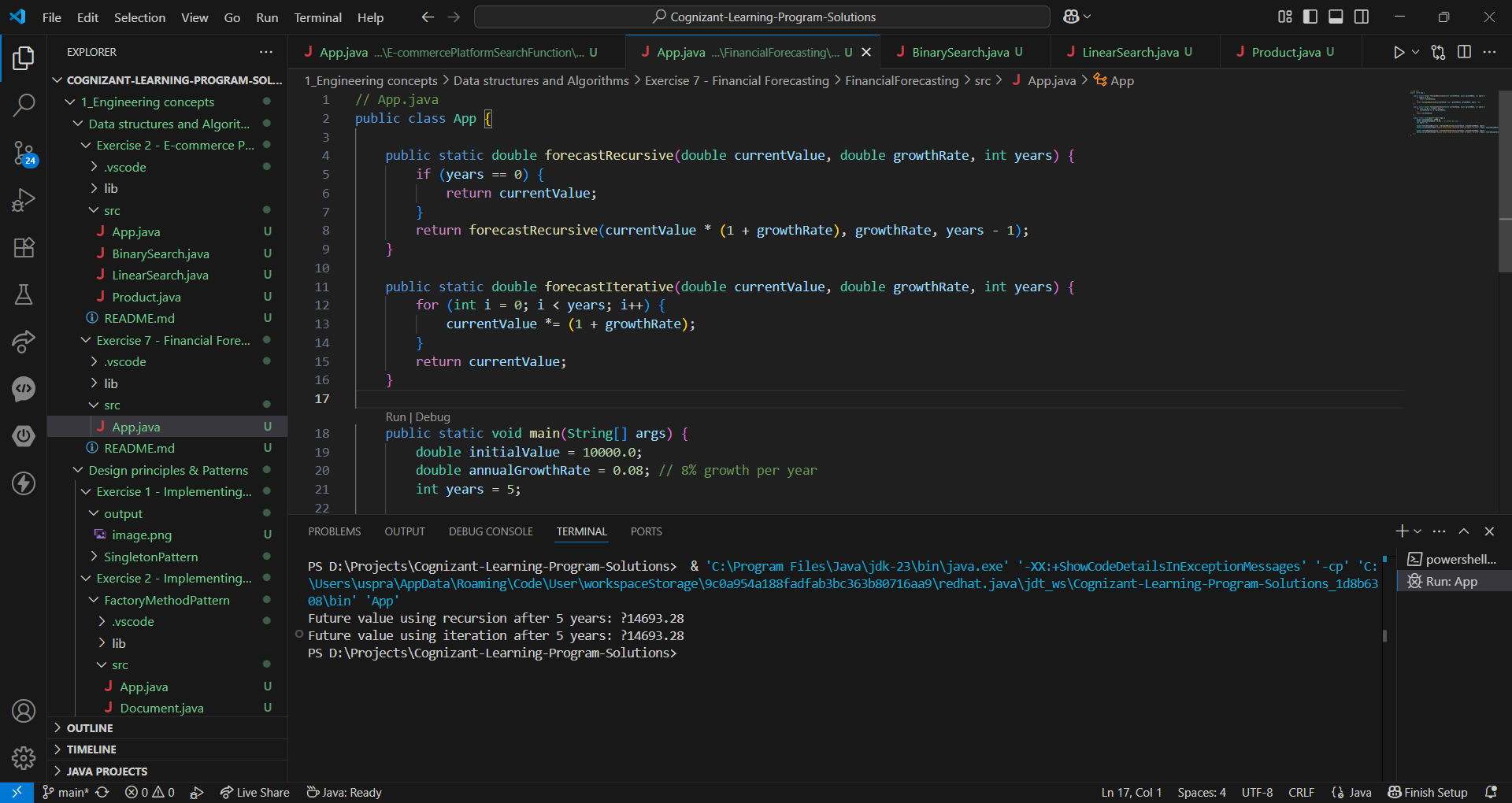
double futureValueIterative = forecastIterative(initialValue, annualGrowthRate, years);

System.out.printf("Future value using iteration after %d years: ₹%.2f%n", years, futureValueIterative);

}

}

**Output:**

****

**Analysis:**

| **Aspect** | **Value** |
| --- | --- |
| Time Complexity | **O(n)** (n recursive calls) |
| Space Complexity | **O(n)** (due to call stack) |

**Optimization Strategy:**

### Problem: Recursion may cause stack overflow for large n.

### Solution: Use iterative approach or tail recursion (Java does not optimize tail recursion, so iteration is safer).

### Optimized Iterative Version:

public static double forecastIterative(double currentValue, double growthRate, int years) {

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}