**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.

## **1. Understand Asymptotic Notation**

🔹 **Big O Notation:**

Big O notation describes the upper bound of an algorithm’s running time or space requirement in terms of input size n. It helps to:  
- Analyze performance regardless of hardware.  
- Compare efficiency of different algorithms.  
- Focus on scalability.

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

• Best Case (Ω): The item is found immediately.  
• Average Case (Θ): The item is found after scanning half the collection.  
• Worst Case (O): The item is not found or is last in the collection.

**2. Setup**

Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

public class Product {

String productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String toString() {

return "ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

**3. Implementation:**

🔹 **Linear Search Implementation:**

**// ProductSerach.java**

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

for (Product p : products) {

if (p.productName.equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

🔹 **Binary Search Implementation (on sorted array):**

**// ProductSearch.java**

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

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

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

while (left <= right) {

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

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

if (compare == 0) {

return products[mid];

} else if (compare < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

**4.Analysis**

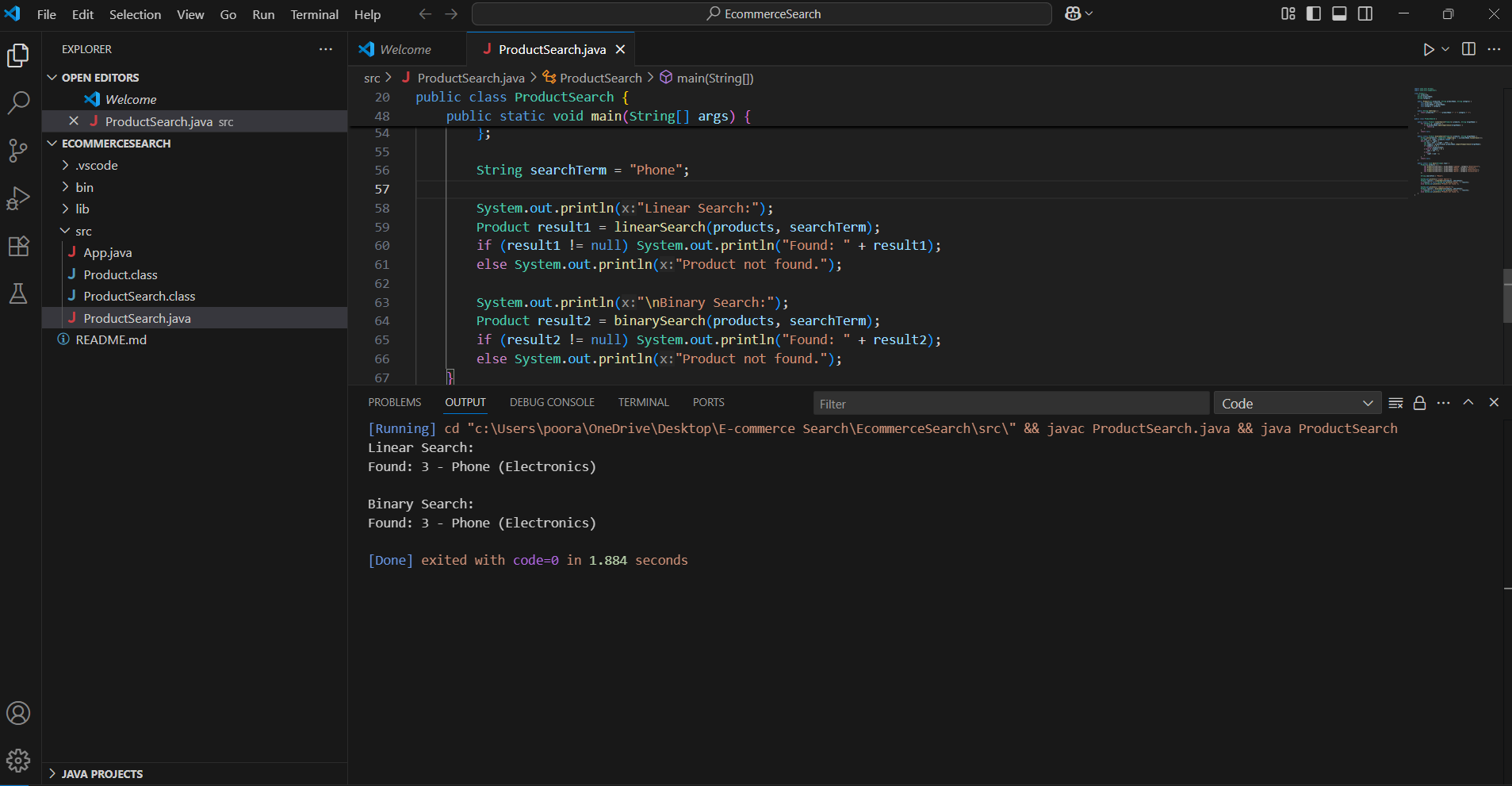
🔹 **Time Complexity Comparison:**

• Linear Search: Best Case – O(1), Average/Worst Case – O(n)  
• Binary Search: Best Case – O(1), Average/Worst Case – O(log n)

🔹 **Recommendation:**

We an use Binary Search for large, sorted datasets to achieve better performance and Linear Search for small or unsorted collections.

**5. Output screenshot**



**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**

**🔹 The concept of recursion and how it can simplify certain problems:**  
 • Recursion is a programming technique in which a function repeatedly calls itself to solve smaller pieces of a larger problem.  
 • In forecasting, this is helpful because each future value depends on the previous year's value — a naturally recursive relationship.  
 • Instead of writing nested loops, recursion offers a cleaner way to model step-by-step annual changes. It simplifies the problem logic and makes the code easier to understand when dealing with repetitive calculations.

1. **Setup**

🔹 **A method to calculate the future value using a recursive approach:**

**// Forecast.java**

public static double calculateFutureValue(double currentValue, double annualGrowthRate, int years) {

if (years == 0) {

return currentValue;

}

return calculateFutureValue(currentValue, annualGrowthRate, years - 1) \* (1 + annualGrowthRate);

}

}

1. **Implementation:**

🔹 **Implementation of a recursive algorithm to predict future values based on past growth rates:**

**// ForecastTest.java**

public class ForecastTest {

public static void main(String[] args) {

double startingAmount = 1500.0;

double growthRate = 0.07;

int totalYears = 6;

double result = Forecast.calculateFutureValue(startingAmount, growthRate, totalYears);

System.out.printf("Estimated value after %d years: ₹%.2f\n", totalYears, result);

}

}

1. **Analysis:**

**🔹 Time complexity of the recursive algorithm:**

* The time complexity is **O(n)**, where n is the number of years to forecast.
* Each recursive call handles a single year, so the function is called n times in total.

**🔹 How to optimize the recursive solution to avoid excessive computation:**

1. **Use Iteration Instead:**  
   Recursion in this case is straightforward, but using a loop (for or while) is more efficient and avoids potential stack overflow errors.
2. **Avoid Redundant Calls:**  
   Although this example does not contain repeated subproblems, in more complex models, **memoization** (caching results) can be used to save time.
3. **Output screenshot:**

