# 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.

## Steps:

1. Understand Asymptotic Notation:

* - Big O notation describes the upper bound of algorithm complexity as input size grows.
* - Best Case: Item is at the beginning (O(1))
* - Average Case: Item is somewhere in the middle (O(n/2) ≈ O(n))
* - Worst Case: Item is at the end or not found (O(n))

2. Setup:

* - Create a class Product with attributes: productId, productName, category.

3. Implementation:

* - Linear Search: Iterate over array and compare productName.
* - Binary Search: Sort the array by productName and apply binary search.

4. Analysis:

* - Linear Search: O(n), no sorting required, simple but slower for large lists.
* - Binary Search: O(log n), but needs sorted input, much faster for large, sorted lists.
* - Binary Search is more efficient for large datasets when products are sorted.

## Code Implementation

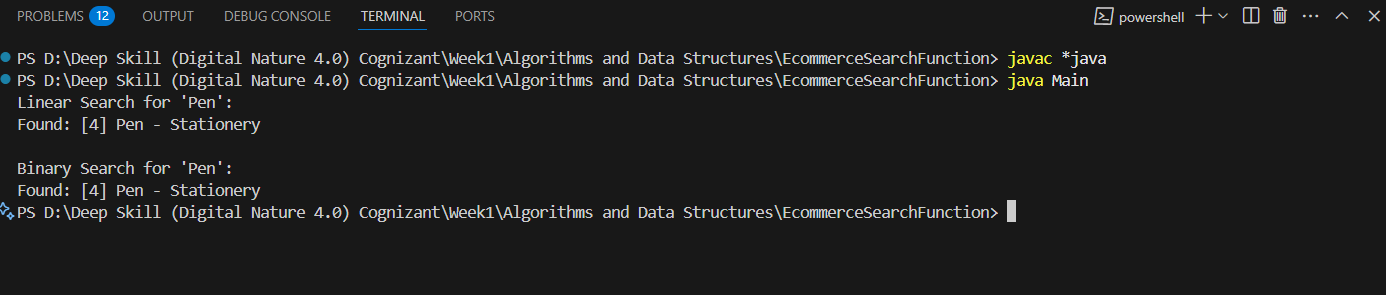
### Product.java

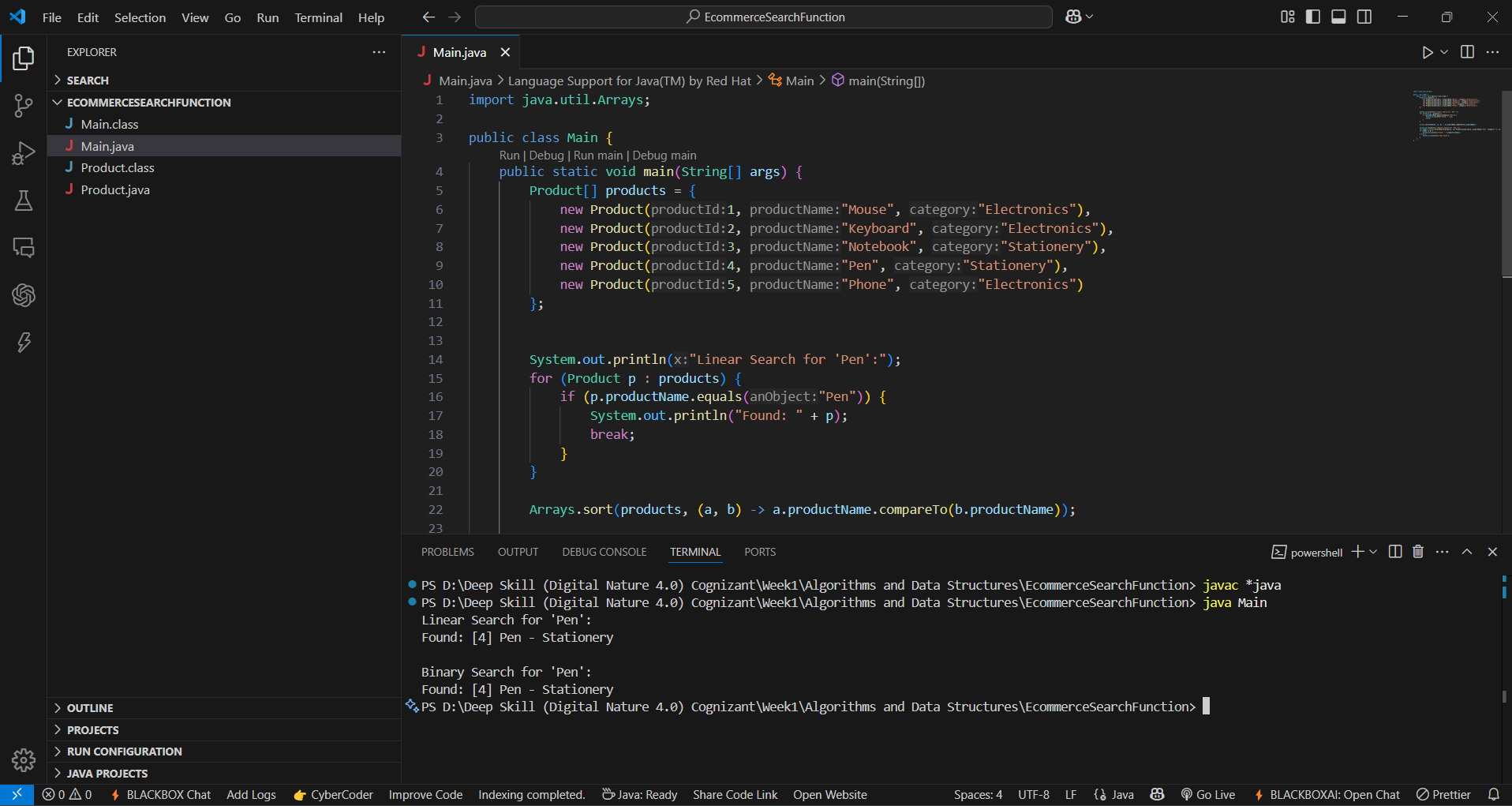
public class Product {  
 int productId;  
 String productName;  
 String category;  
  
 public Product(int productId, String productName, String category) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 }  
  
 public String toString() {  
 return "[" + productId + "] " + productName + " - " + category;  
 }  
}

### Main.java

import java.util.Arrays;  
  
public class Main {  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(1, "Mouse", "Electronics"),  
 new Product(2, "Keyboard", "Electronics"),  
 new Product(3, "Notebook", "Stationery"),  
 new Product(4, "Pen", "Stationery"),  
 new Product(5, "Phone", "Electronics")  
 };  
  
   
 System.out.println("Linear Search for 'Pen':");  
 for (Product p : products) {  
 if (p.productName.equals("Pen")) {  
 System.out.println("Found: " + p);  
 break;  
 }  
 }  
  
   
 Arrays.sort(products, (a, b) -> a.productName.compareTo(b.productName));  
  
   
 System.out.println("\nBinary Search for 'Pen':");  
 int index = Arrays.binarySearch(products, new Product(0, "Pen", ""), (a, b) -> a.productName.compareTo(b.productName));  
 if (index >= 0) {  
 System.out.println("Found: " + products[index]);  
 } else {  
 System.out.println("Not Found");  
 }  
 }  
}

## Output





# 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:

* - Recursion is a technique where a method calls itself to break a problem into smaller subproblems.
* - It simplifies problems like compound interest, Fibonacci, tree traversal, etc.

2. Setup:

* - Create a method to recursively calculate the future financial value using compound interest logic.

3. Implementation:

* - Input: initial value, growth rate, number of years
* - Use recursion: future = value \* (1 + growthRate) for each year

4. Analysis:

* - Time Complexity: O(n), where n is the number of years (each recursive call reduces n by 1)
* - Optimizations: Use memoization or iteration if performance becomes critical.

## Code Implementation

### FinancialForecast.java

public class FinancialForecast {

public static double forecast(double value, double growthRate, int years) {  
 if (years == 0) {  
 return value;  
 }  
 return forecast(value \* (1 + growthRate), growthRate, years - 1);  
 }  
  
 public static void main(String[] args) {  
 double initialValue = 10000.0;   
 double growthRate = 0.10;   
 int years = 5;  
  
 double futureValue = forecast(initialValue, growthRate, years);  
  
 System.out.printf("Future Value after %d years: ₹%.2f%n", years, futureValue);  
 }  
}

## Output

