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

1. **Understand Asymptotic Notation:**

* Big O notation is a way to describe how the performance of an algorithm changes as the size of the input grows. It gives an upper limit on the time or space an algorithm may take, helping us understand its efficiency. For example, an algorithm with O(n) means its runtime grows linearly with the input size, while O(log n) grows much slower. By using Big O, developers can compare different algorithms and choose the one that works best for large or small data sets, even before running the code.
* In search operations, the best, average, and worst-case scenarios describe how many steps an algorithm might take depending on where the target item is located (or if it's not found at all):

**Best Case:** The item is found immediately (e.g., at the start of the list). For linear search, this is O(1). For binary search, the best case is also O(1) if the item happens to be exactly in the middle.

**Average Case**: The item is found somewhere in the middle, or the algorithm goes through about half the list. Linear search takes O(n/2) on average (simplified to O(n)), while binary search still averages O(log n) due to halving the search space each step.

**Worst Case**: The item is not in the list or found last. For linear search, this means checking every element (O(n)). For binary search, the worst case is still efficient at O(log n).

1. **Setup and Implementation:**

**Code:**

package WeekOne.java;

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** "Product{" + "productId=" + productId +", productName='" + productName + '\'' + ", category='" +

category + '\'' +'}';

}

}

**Main Class:**

package WeekOne.java;

import java.util.Arrays;

import java.util.Comparator;

public class ShopSearchApp {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shirt", "Clothing"),

new Product(3, "Phone", "Electronics"),

new Product(4, "Watch", "Accessories"),

new Product(5, "Shoes", "Footwear")

};

// Linear Search

String targetName = "Phone";

Product foundLinear = *linearSearch*(products, targetName);

System.*out*.println("Linear Search Result: " + (foundLinear != null ? foundLinear : "Not Found"));

// Sort products by productName for binary search

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

// Binary Search

Product foundBinary = *binarySearch*(products, targetName);

System.*out*.println("Binary Search Result: " + (foundBinary != null ? foundBinary : "Not Found"));

}

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

for (Product p : products) {

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

return p;

}

}

return null;

}

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

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

while (low <= high) {

int mid = (low + high) / 2;

int comparison = products[mid].productName.compareToIgnoreCase(name);

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

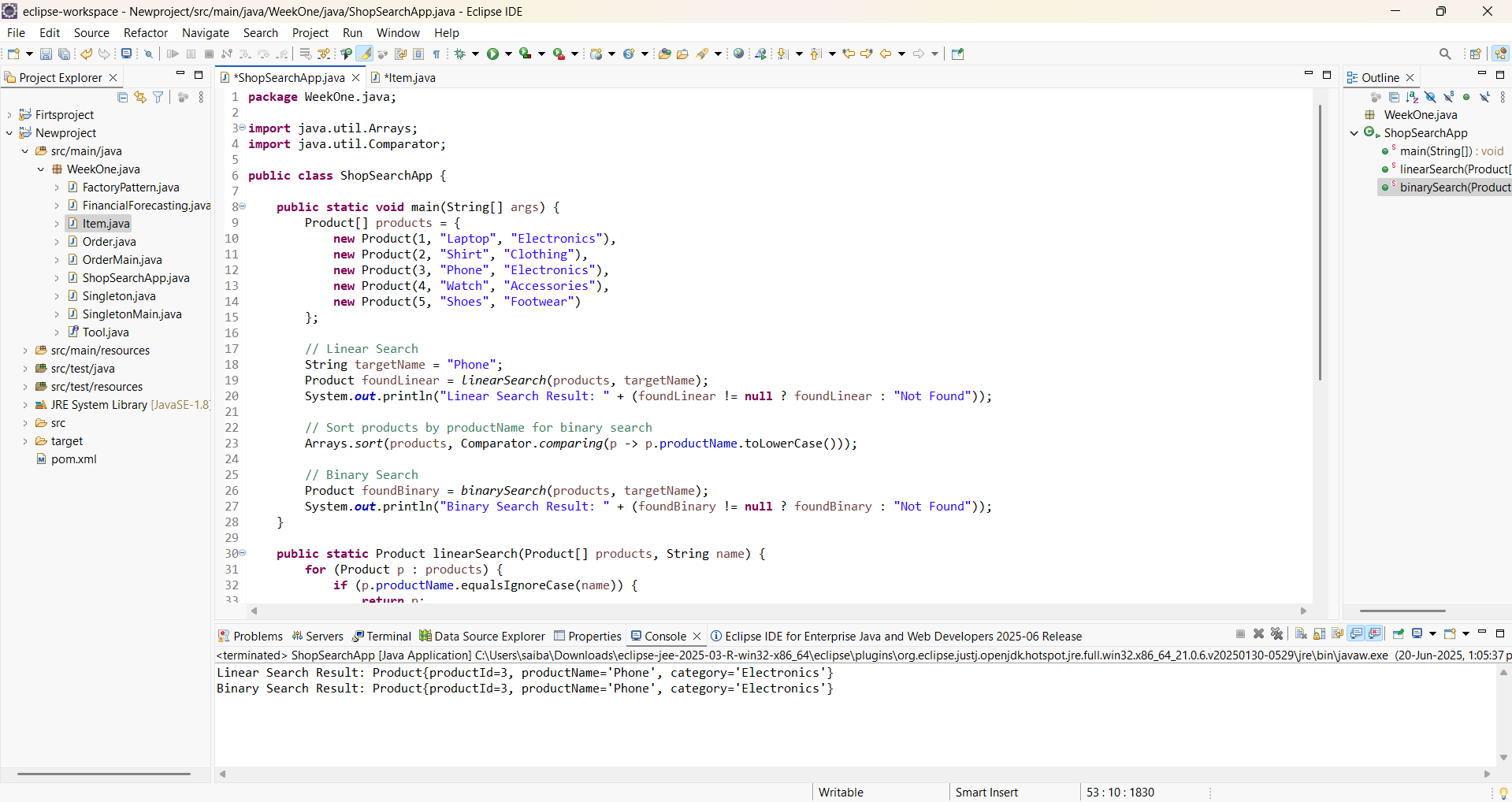
}

return null;

}

}

**Output:**

****

1. **Analysis:**

* Linear search has a time complexity of O(n), meaning it may need to check every element in the list to find a match (or determine it's not there). It's simple and works on unsorted data but becomes slow with large datasets.
* **Binary search**, on the other hand, has a time complexity of **O(log n).** It repeatedly divides the sorted list in half, drastically reducing the number of steps needed. However, it **requires the data to be sorted** first.

# **Exercise 7: Financial Forecasting**

1. **Understand Recursive Algorithms:**

Recursion is a programming technique where a method calls itself to solve smaller parts of a larger problem. It simplifies problems that have a natural repetitive or branching structure, such as computing factorials, Fibonacci numbers, or, in this case, forecasting future financial values. Recursive solutions are often cleaner and easier to understand but can be inefficient without optimization.

1. **Setup and Implementation:**

**Code:**

package WeekOne.java;

import java.util.Scanner;

public class FinancialForecast {

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

if (years == 0) {

return currentValue;

}

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

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.*in*);

System.*out*.print("Enter current value: ");

double currentValue = scanner.nextDouble();

System.*out*.print("Enter annual growth rate (e.g., 0.05 for 5%): ");

double growthRate = scanner.nextDouble();

System.*out*.print("Enter number of years: ");

int years = scanner.nextInt();

double futureValue = *forecastValue*(currentValue, growthRate, years);

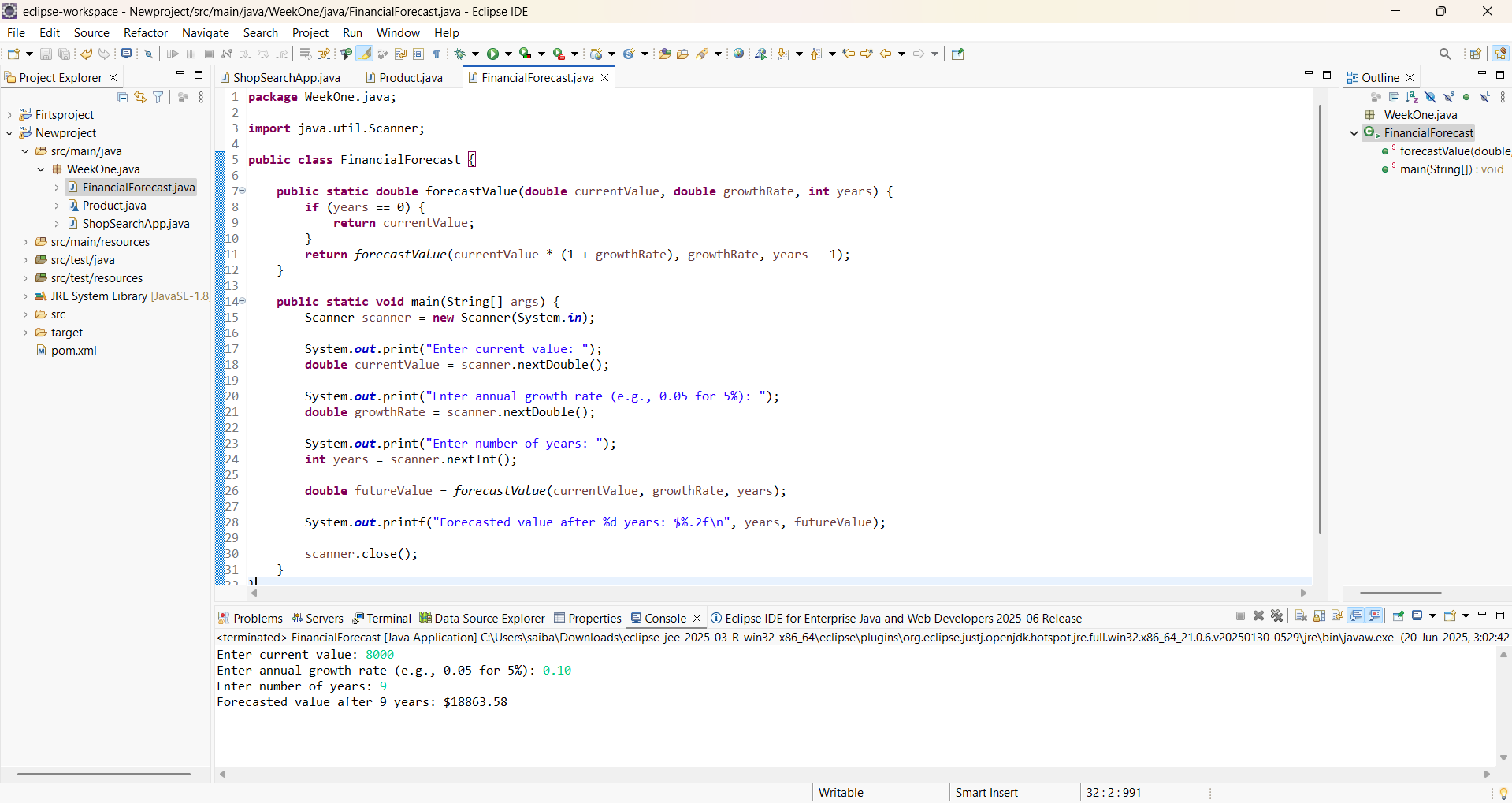
System.*out*.printf("Forecasted value after %d years: $%.2f\n", years, futureValue);

scanner.close();

}

}

**Output:**

****

1. **Analysis:**

* **Discuss the time complexity of your recursive algorithm :**

The time complexity of the recursive financial forecasting algorithm is O(n), where *n* is the number of years being forecasted. This is because the function makes one recursive call for each year, reducing the years value by one each time until it reaches zero. Each call performs a constant-time multiplication, so the overall time grows linearly with the number of years. The space complexity is also O(n) due to the recursive call stack, which can lead to stack overflow for very large values of *n*. For better performance and to avoid memory issues, an iterative approach is often preferred for large forecasts.

* **Explain how to optimize the recursive solution to avoid excessive computation :**

To optimize the recursive solution and avoid excessive computation, you can convert it into an iterative approach, which eliminates the need for repeated function calls and reduces memory usage. In recursion, each call adds a new frame to the call stack, leading to higher space complexity (O(n)) and the risk of a stack overflow for large input values. By using a simple loop instead, you can calculate the future value step by step without recursion, achieving the same O(n) time complexity but with O(1) space complexity. This makes the program more efficient and suitable for larger forecasting ranges.