**Week 1**

**Data structures and Algorithms**

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

**1. Understand Asymptotic Notation:**

* + **Explain Big O notation and how it helps in analyzing algorithms.**
  + Big O notation is a used to represent the efficiency of an algorithm as the size of the input becomes very large.
  + It is used to calculate the upper bound of algorithms running time
  + It is used to represent the worst case time complexity

O(1): Constant time

O(n): Linear time - time increases as the input size increases.

O(log n): Logarithmic time - time increases slowly even with large input.

O(n²): Quadratic time – time increases rapidly as input size increases.

* + **Describe the best, average, and worst-case scenarios for search operations.**

**Linear Search Binary Search**

|  |  |
| --- | --- |
| **Best Case** | **O(1)** |
| **Average Case** | **O(n/2)** |
| **Worst Case** | **O(n)** |

|  |  |
| --- | --- |
| **Best Case** | **O(1)** |
| **Average Case** | **O(log n)** |
| **Worst Case** | **O(log n)** |

**2. Setup:**

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

**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 + ")";

}

}

**3. Implementation:**

* **Implement linear search and binary search algorithms.**
* **Store products in an array for linear search and a sorted array for binary search.**

**searchEngine.java**

import java.util.Arrays;

import java.util.Comparator;

public class SearchEngine {

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Sun screen", "Personal Care"),

new Product(103, "Laptop", "Electronics"),

new Product(104, "Rocking Chair", "Furniture"),

new Product(105, "Recliner", "Furniture")

};

Product result1 = linearSearch(products, "Tablet");

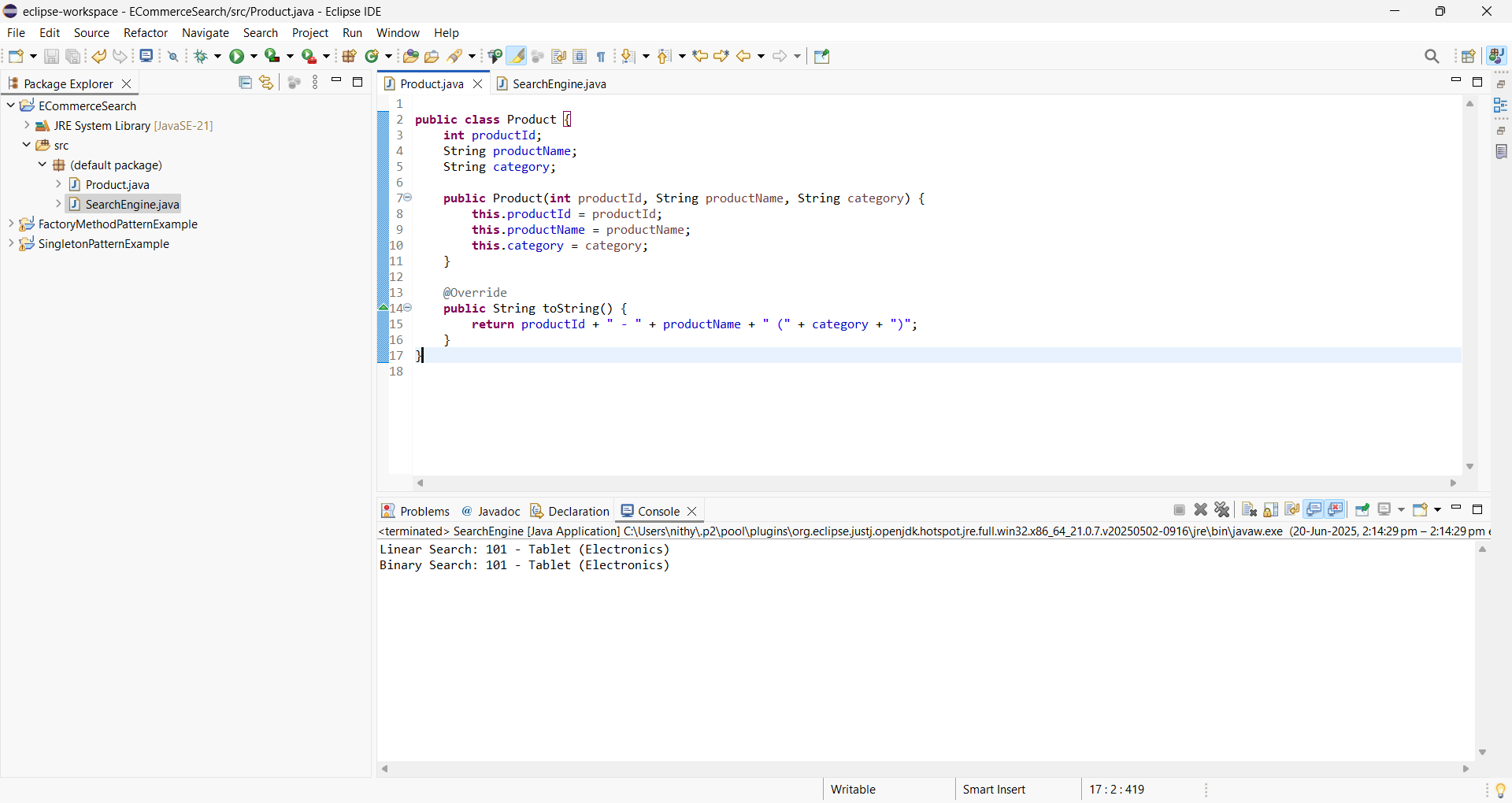
System.out.println("Linear Search: " + (result1 != null ? result1 : "Product not found"));

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

Product result2 = binarySearch(products, "Tablet");

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

}

}

**Exercise 7: Financial Forecasting**

1. **Understand Recursive Algorithms:**

* **Explain the concept of recursion and how it can simplify certain problems.**

Recursion is a technique where a method calls itself to solve a smaller instance of the problem.

1. **Setup:Create a method to calculate the future value using a recursive approach.**

static double forecastRecursive(double initial,

double[] growthRates,

int year) {

}

1. **Implementation:Implement a recursive algorithm to predict future values based on past growth rates.**

**FinancialForecastProject**

public class FinancialForecast {

static double forecastRecursive(double initial,double[] growthRates,int year) {

if (year == 0) {

return initial;

}

double prev = *forecastRecursive*(initial, growthRates, year - 1);

return prev \* (1.0 + growthRates[year - 1]);

}

static void printYearlyForecast(double initial, double[] growthRates,int years) {

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

double value = *forecastRecursive*(initial, growthRates, n);

System.*out*.printf("Year %d → ₹%.2f%n", n, value);

}

}

public static void main(String[] args) {

double initialAmount = 10\_000;

double[] growth = {0.05, 0.08, 0.06, 0.07};

int totalYears = growth.length;

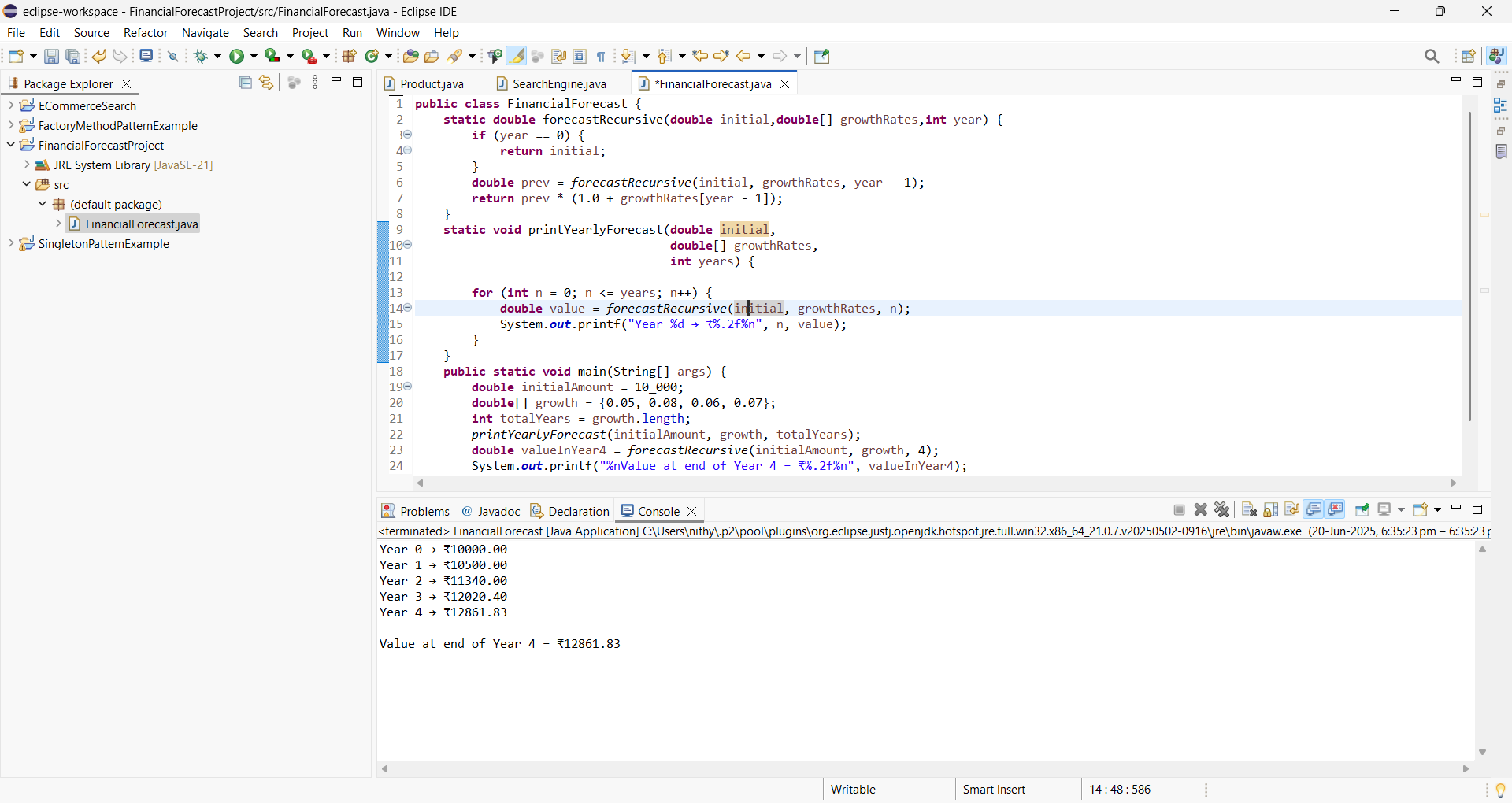
*printYearlyForecast*(initialAmount, growth, totalYears);

double valueInYear4 = *forecastRecursive*(initialAmount, growth, 4);

System.*out*.printf("%nValue at end of Year 4 = ₹%.2f%n", valueInYear4);

}

}

****

1. **Analysis:**

* **Discuss the time complexity of recursive algorithm.**
* **Explain how to optimize the recursive solution to avoid excessive computation.**

The time complexity of recursive algorithm is O(log n)