**Data Structures and Algorithms**

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

**Java Project Name: ECommerceSearchExample**

**Product.java**

package com.ecommerce.search;

public class Product implements Comparable<Product> {

private String productId;

private String productName;

private String category;

private double price;

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

this.productId = productId;

this.productName = productName;

this.category = category;

this.price = price;

}

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

public double getPrice() { return price; }

@Override

public int compareTo(Product other) {

return this.productId.compareTo(other.productId);

}

@Override

public String toString() {

return "Product [ID=" + productId + ", Name=" + productName + ", Category=" + category + ", Price=$" + price + "]";

}

}

**LinearSearch.java**

package com.ecommerce.search;

import java.util.ArrayList;

import java.util.List;

public class LinearSearch {

public static Product searchById(Product[] products, String targetId) {

for (Product product : products) {

if (product.getProductId().equals(targetId)) {

return product;

}

}

return null;

}

public static List<Product> searchByName(Product[] products, String keyword) {

List<Product> results = new ArrayList<>();

for (Product product : products) {

if (product.getProductName().toLowerCase().contains(keyword.toLowerCase())) {

results.add(product);

}

}

return results;

}

}

**BinarySearch.java**

package com.ecommerce.search;

import java.util.Arrays;

public class BinarySearch {

public static Product searchById(Product[] sortedProducts, String targetId) {

int left = 0;

int right = sortedProducts.length - 1;

while (left <= right) {

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

String midId = sortedProducts[mid].getProductId();

if (midId.equals(targetId)) {

return sortedProducts[mid];

} else if (midId.compareTo(targetId) < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**SearchTest.java (Main Class)**

package com.ecommerce.search;

import java.util.Arrays;

import java.util.List;

public class SearchTest {

public static void main(String[] args) {

Product[] products = {

new Product("P100", "iPhone 13", "Electronics", 999.99),

new Product("P200", "Samsung TV", "Electronics", 799.99),

new Product("P300", "Nike Shoes", "Fashion", 120.50),

new Product("P400", "Dell Laptop", "Electronics", 899.99),

new Product("P500", "Adidas Hoodie", "Fashion", 59.99)

};

Product[] sortedProducts = Arrays.copyOf(products, products.length);

Arrays.sort(sortedProducts);

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

long startTime = System.nanoTime();

Product foundLinear = LinearSearch.searchById(products, "P300");

long endTime = System.nanoTime();

System.out.println("Found: " + foundLinear);

System.out.println("Time taken: " + (endTime - startTime) + " ns");

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

startTime = System.nanoTime();

Product foundBinary = BinarySearch.searchById(sortedProducts, "P300");

endTime = System.nanoTime();

System.out.println("Found: " + foundBinary);

System.out.println("Time taken: " + (endTime - startTime) + " ns");

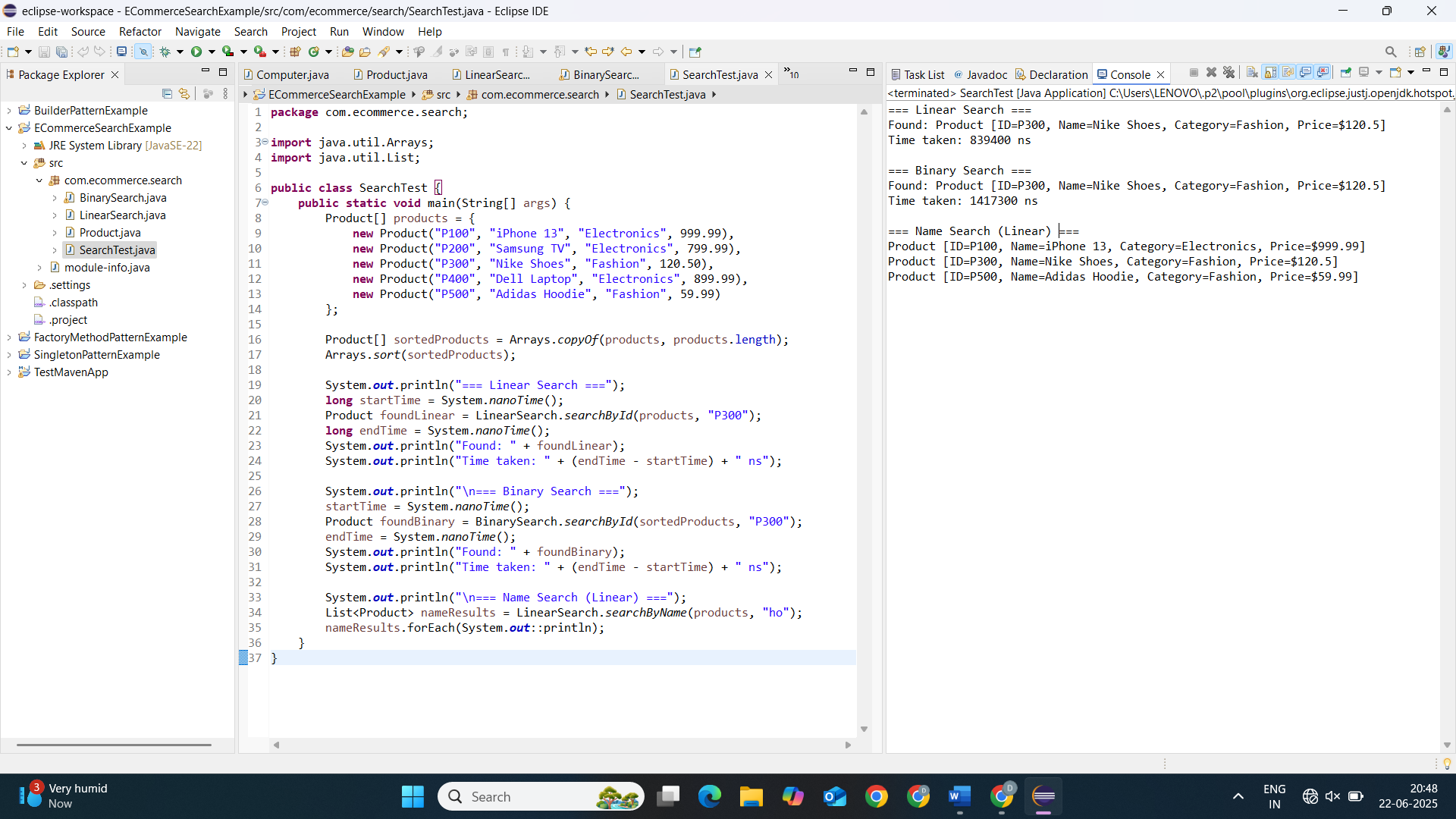
System.out.println("\n=== Name Search (Linear) ===");

List<Product> nameResults = LinearSearch.searchByName(products, "ho");

nameResults.forEach(System.out::println);

}

}



**Understanding Asymptotic Notation**

1. **What is Big O Notation?**

Big O notation describes how an algorithm's runtime or memory usage grows as input size increases. It focuses on **worst-case** behavior and ignores constants.

**Why It Matters**

* Helps compare algorithms objectively.
* Predicts performance at scale.
* Identifies bottlenecks in code.

**Common Complexities**

| **Big O** | **Name** | **Example** |
| --- | --- | --- |
| **O(1)** | Constant | Array access (arr[0]) |
| **O(log n)** | Logarithmic | Binary search |
| **O(n)** | Linear | Linear search |
| **O(n²)** | Quadratic | Nested loops |

**2. Search Operation Scenarios**

**Best Case**

* **Scenario**: Target is found immediately.
* **Linear Search**: O(1) (first element).
* **Binary Search**: O(1) (middle element).

**Average Case**

* **Scenario**: Target is found after searching half the data.
* **Linear Search**: O(n/2) → **O(n)**.
* **Binary Search**: O(log n).

**Worst Case**

* **Scenario**: Target is missing or at the end.
* **Linear Search**: O(n) (check all elements).
* **Binary Search**: O(log n) (split until empty).

**Analysis**

| **Search Type** | **Time Complexity** | **Suitable For** |
| --- | --- | --- |
| **Linear** | O(n) | Small datasets or unsorted data |
| **Binary** | O(log n) | Large datasets but must be **sorted** |
|  |  |  |

**Recommendation:**

* **Binary Search** is **faster** for large, sorted datasets.
* **Linear Search** is better if data changes frequently and sorting is not practical.

**Exercise 7: Financial Forecasting**

**Java Project Name: FinancialForecastingTool**

**ForecastTool.java**

package com.finance;

public class ForecastTool {

public static double predictFutureValue(double presentValue, double growthRate, int years) {

if (years == 0) {

return presentValue;

} else {

return (1 + growthRate) \* predictFutureValue(presentValue, growthRate, years - 1);

}

}

public static void main(String[] args) {

double presentValue = 1000.0; // Initial amount

double growthRate = 0.05; // 5% annual growth

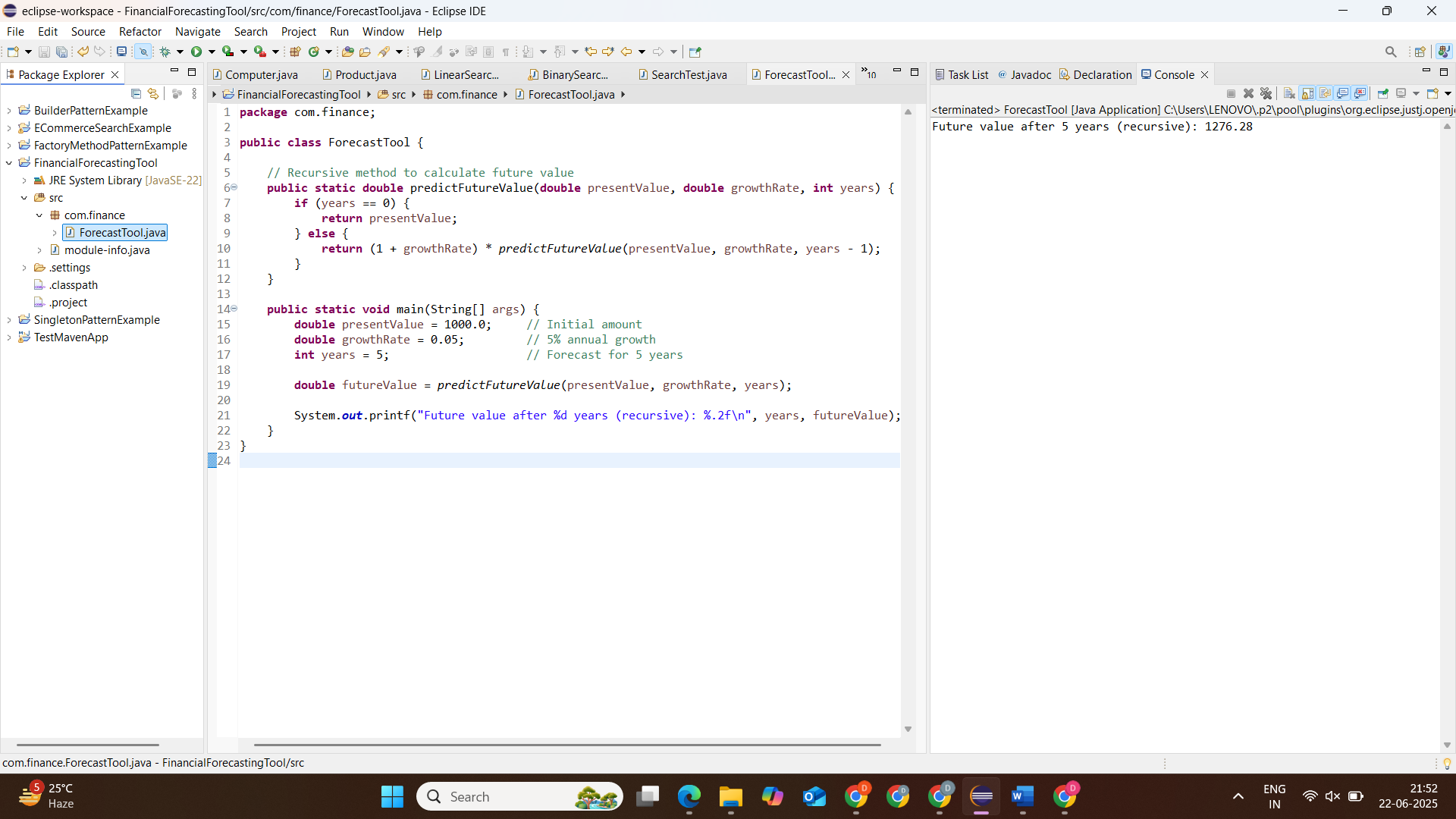
int years = 5; // Forecast for 5 years

double futureValue = predictFutureValue(presentValue, growthRate, years);

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

}

}



**Understanding Recursive Algorithms**

**Recursion**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It simplifies problems that can be broken down into identical subproblems.

**Key Components:**

1. **Base Case**: The stopping condition that prevents infinite recursion
2. **Recursive Case**: The part where the function calls itself with modified parameters

**Why Use Recursion for Financial Forecasting?**

* Naturally models **compound growth** (each year builds on the previous).
* Simplifies calculations where future values depend on past values.

**Analysis of the Recursive Algorithm**

**Time Complexity:**

* The function calls itself **once** per year, so:
  + **Time Complexity**: O(n)
  + **Space Complexity**: O(n) (due to call stack)

For n = 5, the call stack will build up 5 recursive calls.

**Forecast future value based on:**

* Present Value (PV)
* Growth Rate (GR)
* Number of Years (N)

**Formula (Compound Growth):**

Future Value = Present Value \* (1 + Growth Rate)^n

**Optimizing Recursive Computation**

**Problem:**

Recursive methods can be **inefficient** if:

* Too many repeated calculations
* Stack overflow risk for large n

**Optimization Techniques:**

**A. Use Iteration Instead of Recursion:**

public static double predictIteratively(double presentValue, double growthRate, int years) {

double futureValue = presentValue;

for (int i = 1; i <= years; i++) {

futureValue \*= (1 + growthRate);

}

return futureValue;

}

* **Time Complexity**: O(n)
* **Space Complexity**: O(1) – no call stack

**B. Use Memoization (for more complex recursive problems)**

In forecasting, it’s not needed unless subproblems overlap (e.g., in Fibonacci or dynamic programming cases).