**E-commerce Platform Search Function Scenario:**

**1.Understanding Asymptotic Notation**

### **Big O Notation:**

Big O notation describes the **upper bound** of an algorithm’s running time in terms of the input size n. It helps measure **scalability** and **efficiency**.

| **Notation** | **Description** | **Example** |
| --- | --- | --- |
| O(1) | Constant time | Accessing an array element |
| O(n) | Linear time | Linear search |
| O(log n) | Logarithmic time | Binary search |
| O(n log n) | Log-linear time | Merge sort |
| O(n²) | Quadratic time | Bubble sort |

### **Search Case Scenarios:**

| **Case** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best** | O(1) (first match) | O(1) (middle match) |
| **Average** | O(n/2) → O(n) | O(log n) |
| **Worst** | O(n) | O(log n) |

**Product.java**

**package** EcommerceSearch;

**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 + ")";

}

}

**SearchFunctions.java**

**package** EcommerceSearch;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** SearchFunctions {

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

**for** (Product p : products) {

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

**return** p;

}

}

**return** **null**;

}

**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) / 2;

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

**if** (cmp == 0) **return** products[mid];

**else** **if** (cmp < 0) left = mid + 1;

**else** right = mid - 1;

}

**return** **null**;

}

}

**Main.java**

**package** EcommerceSearch;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Product[] products = {

**new** Product(1, "Apple", "Fruits"),

**new** Product(2, "Banana", "Fruits"),

**new** Product(3, "Carrot", "Vegetables"),

**new** Product(4, "Dates", "Dry Fruits")

};

Product result1 = SearchFunctions.*linearSearch*(products, "Carrot");

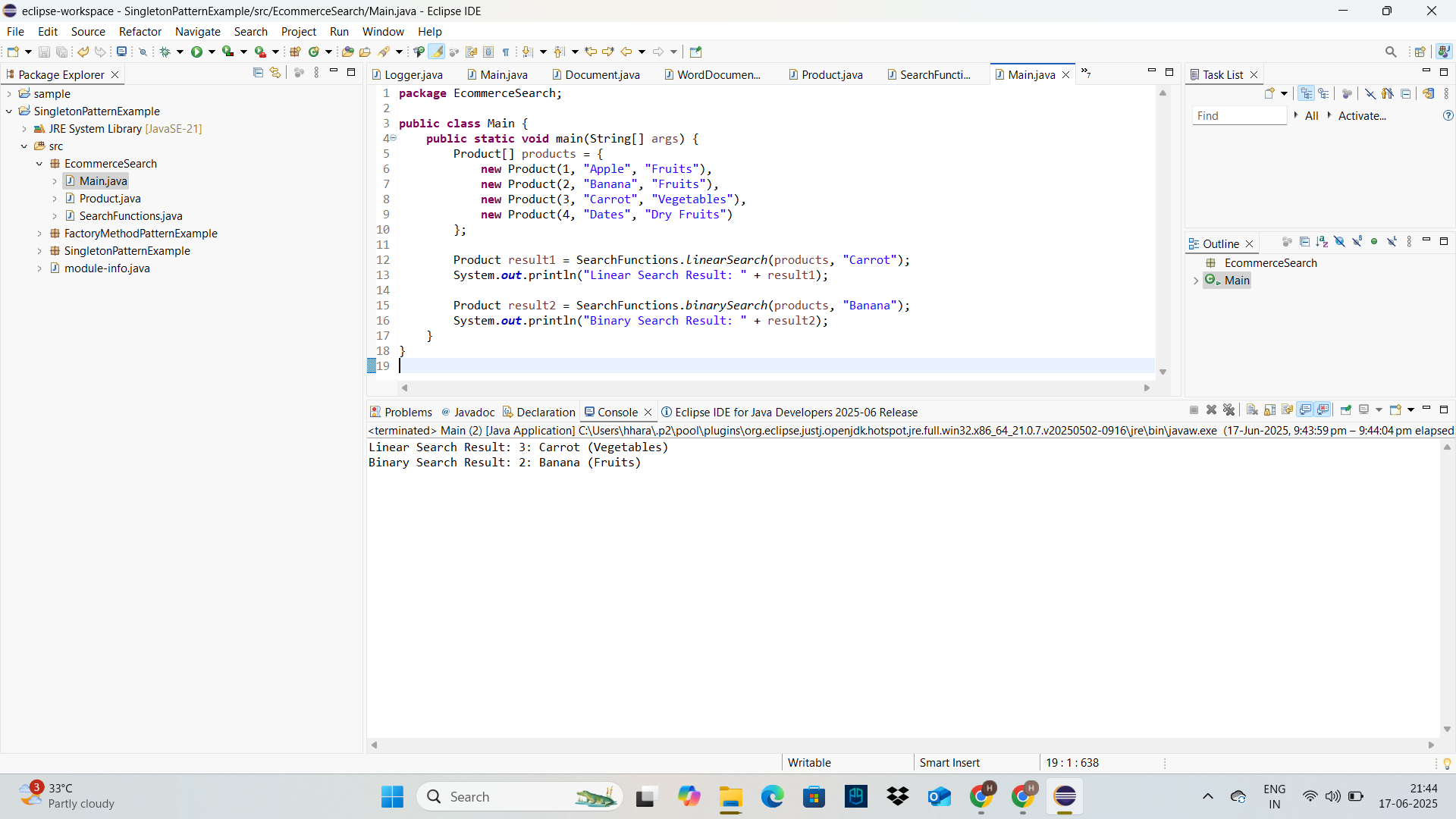
System.***out***.println("Linear Search Result: " + result1);

Product result2 = SearchFunctions.*binarySearch*(products, "Banana");

System.***out***.println("Binary Search Result: " + result2);

}

}

**OUTPUT:**

### **4. Analysis**

#### Time Complexity Comparison

| Algorithm | Time Complexity | Sorted Data Required | Use Case |
| --- | --- | --- | --- |
| Linear Search | O(n)O(n)O(n) | No | Small or unsorted product lists |
| Binary Search | O(log⁡n)O(\log n)O(logn) | Yes | Large and sorted product lists |

#### Which Algorithm is More Suitable?

* Linear Search is simple and doesn't need the array to be sorted. It's good for small datasets or one-off searches.
* Binary Search is much faster on large datasets but requires sorting, which adds overhead if done frequently.

##### Recommendation:

* Use binary search if:  
  + Product data is already sorted or can be kept sorted (e.g., in a database or index).
  + High performance and scalability are required (common in e-commerce platforms).
* Use linear search for:  
  + Small datasets.
  + Prototyping or where sorting isn’t feasible.

**Financial Forecasting**

### **1. Understand Recursive Algorithms**

#### What Is Recursion?

Recursion is a programming technique where a function calls itself to solve a smaller version of the original problem. It often simplifies problems that have a repetitive or self-similar structure.

Example Use Cases:

* Calculating factorials
* Fibonacci sequence
* Tree traversal
* Financial forecasts over time

#### Why Use Recursion in Financial Forecasting?

In forecasting, the future value often depends on the value from the previous period(s). Recursion models this naturally:

FVn=FVn−1×(1+r)FV\_n = FV\_{n-1} \times (1 + r)FVn​=FVn−1​×(1+r)

Where:

* FVnFV\_nFVn​: future value at year nnn
* rrr: growth rate

### **2. Setup**

We’ll assume:

* A base value (initial amount)
* A fixed annual growth rate
* A number of years to forecast

**3. Implementation**

**FinancialForecast.java**

package FinancialForecast;

import java.util.HashMap;

import java.util.Map;

public class FinancialForecast {

public static double futureValueRecursive(double initialValue, double rate, int years) {

if (years == 0) return initialValue;

return *futureValueRecursive*(initialValue, rate, years - 1) \* (1 + rate);

}

public static double futureValueMemo(double initialValue, double rate, int years, Map<Integer, Double> memo) {

if (memo.containsKey(years)) return memo.get(years);

if (years == 0) return initialValue;

double result = *futureValueMemo*(initialValue, rate, years - 1, memo) \* (1 + rate);

memo.put(years, result);

return result;

}

public static double futureValueIterative(double initialValue, double rate, int years) {

double value = initialValue;

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

value \*= (1 + rate);

}

return value;

}

}

**Main.java**

**package** FinancialForecast;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**double** initial = 1000.0;

**double** rate = 0.05; // 5% annual growth

**int** years = 5;

**double** recursive = FinancialForecast.*futureValueRecursive*(initial, rate, years);

System.***out***.println("Future Value (Recursive): $" + String.*format*("%.2f", recursive));

Map<Integer, Double> memo = **new** HashMap<>();

**double** memoized = FinancialForecast.*futureValueMemo*(initial, rate, years, memo);

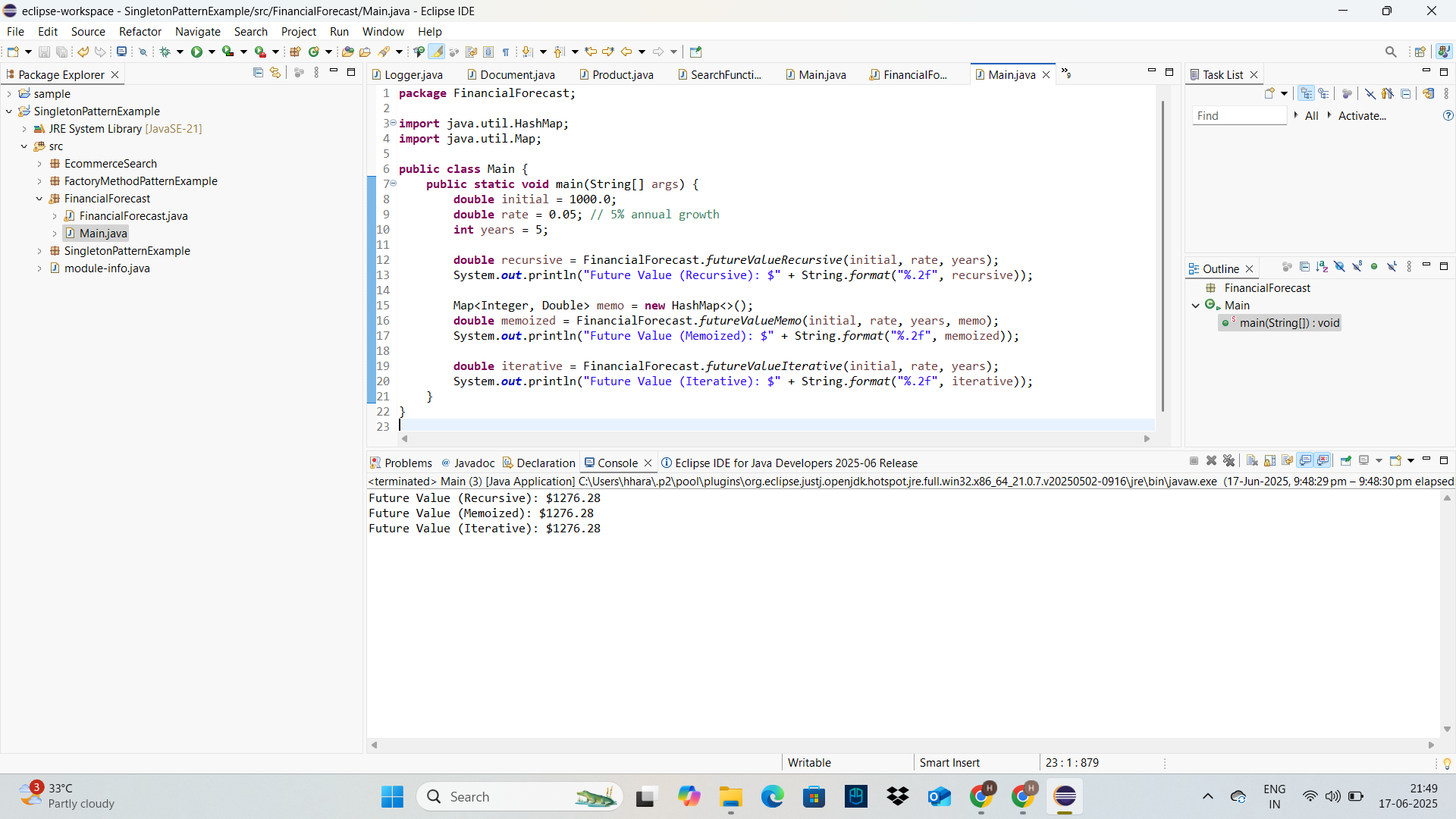
System.***out***.println("Future Value (Memoized): $" + String.*format*("%.2f", memoized));

**double** iterative = FinancialForecast.*futureValueIterative*(initial, rate, years);

System.***out***.println("Future Value (Iterative): $" + String.*format*("%.2f", iterative));

}

}

**OUTPUT:**

### **4. Analysis**

#### **Time Complexity**

* The function calls itself once for each year.
* So, **Time Complexity: O(n)O(n)O(n)**, where nnn is the number of years.

#### **Optimization – Avoid Redundant Computation**

The current implementation is efficient for this simple case, but if each year depended on **multiple** past years (e.g., like Fibonacci), it would result in **exponential** time O(2n)O(2^n)O(2n).