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

**What is Big O Notation?**

Big O notation describes the **upper bound** of an algorithm’s time or space complexity in terms of input size n. It provides a way to **classify algorithms** by how they respond to changes in input size, focusing on **performance and scalability**.

**Search Time Complexities:**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) |
| **Binary Search** | O(1) | O(log n) | O(log n) |

**Linear Search** checks each element until it finds a match.  
**Binary Search** divides the array and eliminates half the data each time, **requires sorted data**.

**Product.java**

package searchfunction;

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 + ")";

}

}

**Search.java**

package searchfunction;

import java.util.\*;

public class Search {

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

for (Product product : products) {

if (product.productName.equalsIgnoreCase(productName)) {

return product;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

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

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void sortByName(Product[] products) {

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

}

}

**Main.java**

package searchfunction;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

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

new Product(4, "Table", "Furniture"),

new Product(5, "Pen", "Stationery")

};

System.out.println("Linear Search for 'Mobile':");

Product foundLinear = Search.*linearSearch*(products, "Mobile");

System.out.println(foundLinear != null ? foundLinear : "Not Found");

Search.*sortByName*(products);

System.out.println("\nBinary Search for 'Mobile':");

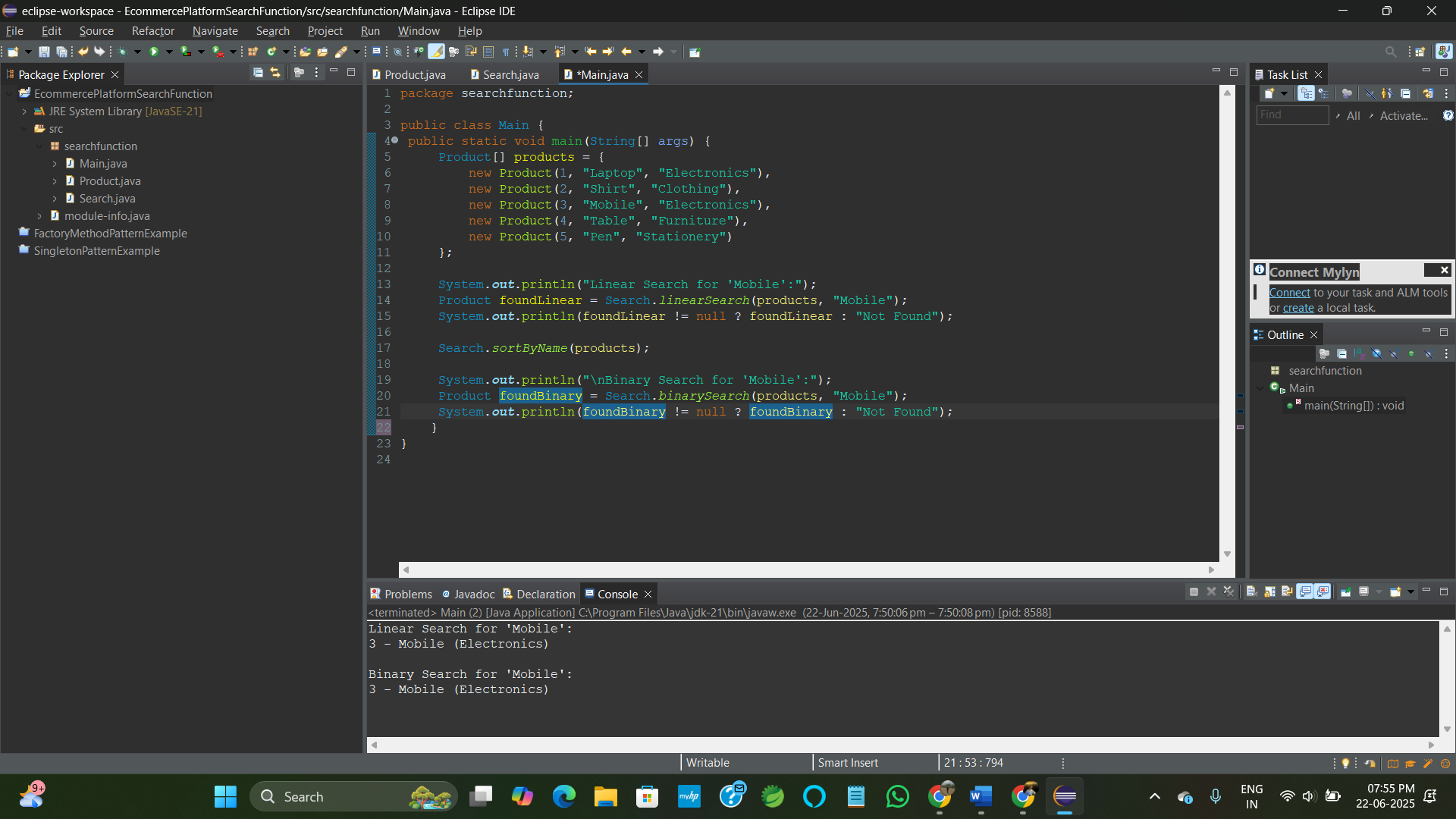
Product foundBinary = Search.*binarySearch*(products, "Mobile");

System.out.println(foundBinary != null ? foundBinary : "Not Found");

}

}

**OUTPUT:**



* For small or unsorted datasets, linear search is acceptable.
* For large datasets, binary search is better if data is sorted (or use a data structure like a hash map or Trie for even faster search).

**Exercise 7: Financial Forecasting**

**What is Recursion?**

Recursion is a programming technique where a method **calls itself** to solve smaller instances of a problem.

* A recursive function breaks down a complex problem into smaller versions of the same problem.
* It usually has
  + **Base case** – the stopping condition.
  + **Recursive case** – the function calls itself with smaller input.

**Why use recursion?**  
It simplifies problems like

* Factorial
* Fibonacci numbers
* Tree traversal
* Forecasting with repeating patterns or growth rates.

**FinancialForecast.java**

package financialforecasting;

public class FinancialForecast {

public static double calculateFutureValue(double presentValue, double rate, int years) {

if (years == 0) {

return presentValue;

}

return (1 + rate) \* calculateFutureValue(presentValue, rate, years - 1);

}

public static void main(String[] args) {

double presentValue = 10000.0;

double rate = 0.05;

int years = 10;

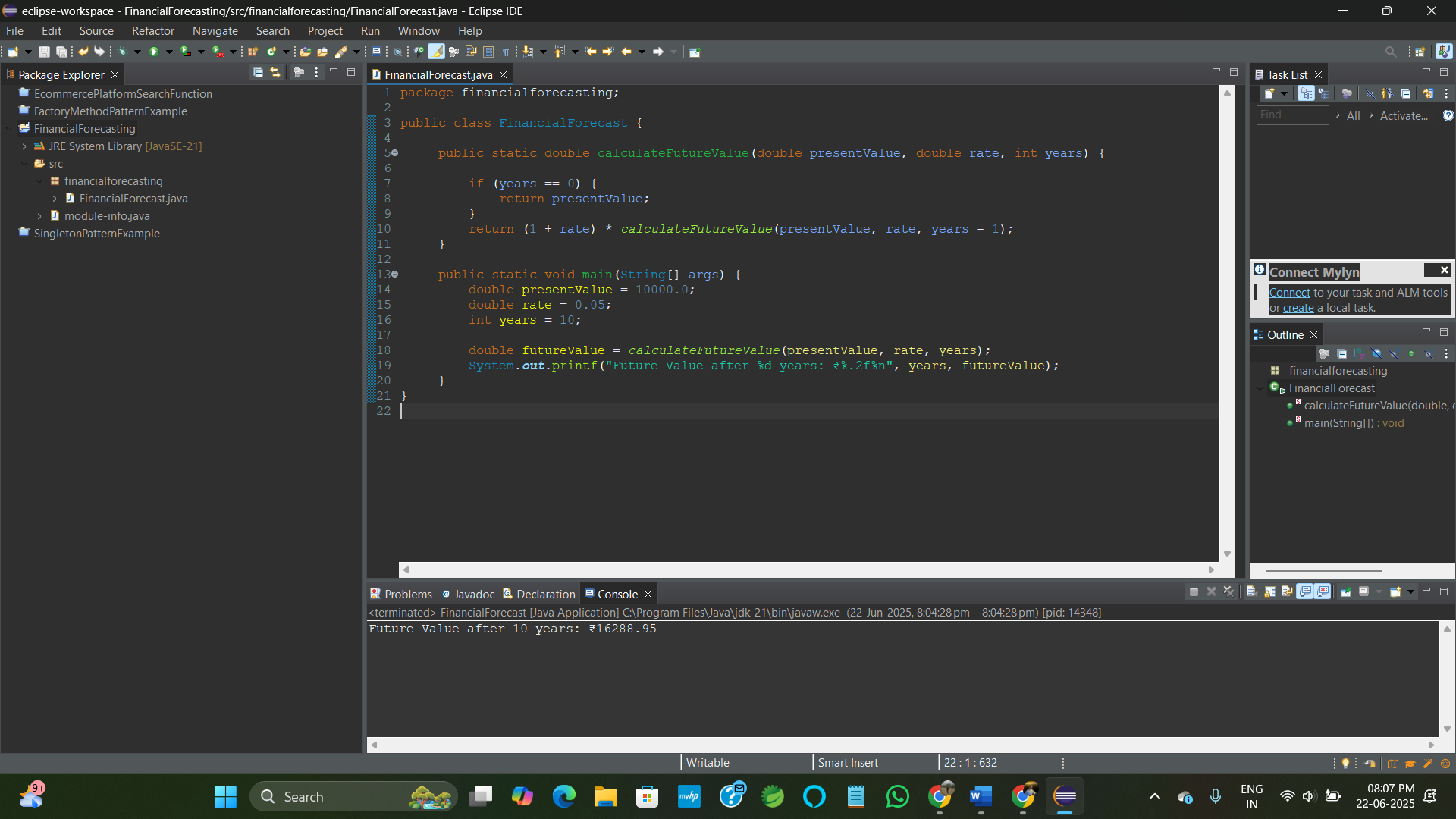
double futureValue = calculateFutureValue(presentValue, rate, years);

System.out.printf("Future Value after %d years: ₹%.2f%n", years, futureValue);

}

}

**OUTPUT:**



**Time Complexity:**

* **T(n) = T(n-1) + O(1)**
* So overall: **O(n)** (linear time)

Each recursive call reduces years by 1, so we make n calls.

**Optimization**

This recursion is **tail-recursive** (calls happen at the end), but in larger systems or deep recursions, it’s better to:

* Use **memoization** (not needed here due to one path of recursion)
* Or convert to an **iterative solution** to avoid stack overflow for large n.

**Iterative Version:**

public static double calculateFutureValueIterative(double presentValue, double rate, int years) {

double futureValue = presentValue;

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

futureValue \*= (1 + rate);

}

return futureValue;

}