**SUPERSET ID : 6375627**

**Week 1 : DATA STRUCTURES AND ALGORITHMS**

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

**Asymptotic Notation**

**Big O Notation:** Big O notation describes the time complexity of algorithms. It indicates how the runtime or space requirements of an algorithm increase as the input size grows. It is useful for analyzing and comparing the efficiency of algorithms with large inputs.

**Best, Average, and Worst-Case Scenarios:**

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

* **Best Case**: The minimum time required for the algorithm to complete (e.g., the element is found in the first comparison).
* **Average Case:** The expected time taken for a random input**.**
* **Worst Case:** The maximum time taken when the input is least favorable (e.g., element is not found, or at the last position).

**Product.java**

package ecommerce;

public class Product {

private int productId;

private String productName;

private String category;

public Product(int id, String name, String cat) {

this.productId = id;

this.productName = name;

this.category = cat;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

*@Override*

public String toString() {

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

}

}

**ProductSearch.java**

package ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class ProductSearch {

// Linear Search

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

for (Product p : products) {

if (p.getProductName().equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

// Binary Search (after sorting)

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

Arrays.*sort*(products, Comparator.*comparing*(Product::getProductName));

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

while (low <= high) {

int mid = (low + high) / 2;

int compare = products[mid].getProductName().compareToIgnoreCase(targetName);

if (compare == 0) {

return products[mid];

} else if (compare < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

} }

**SearchDemo.java**

package ecommerce;

public class SearchDemo {

public static void main(String[] args) {

Product[] catalog = {

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

new Product(102, "Shoes", "Fashion"),

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

new Product(104, "Watch", "Accessories"),

new Product(105, "Tablet", "Electronics")

};

String targetProduct = "Mobile";

System.***out***.println("Using Linear Search:");

Product foundLinear = ProductSearch.*linearSearch*(catalog, targetProduct);

System.***out***.println(foundLinear != null ? foundLinear : "Product not found");

System.***out***.println("\nUsing Binary Search:");

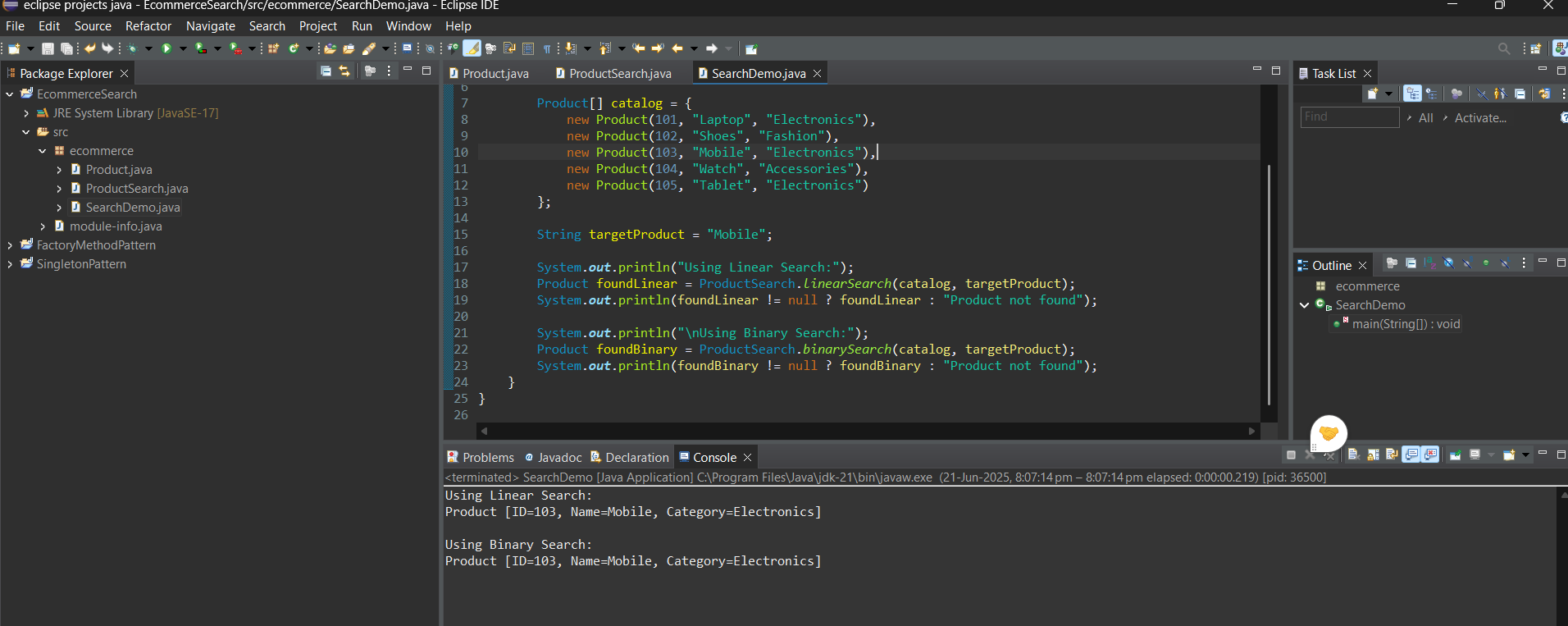
Product foundBinary = ProductSearch.*binarySearch*(catalog, targetProduct);

System.***out***.println(foundBinary != null ? foundBinary : "Product not found");

}

}

**Output:**



**Time Complexity Comparison**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Sorted Required?** |
| --- | --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) | No |
| Binary Search | O(1) | O(log n) | O(log n) | Yes |

In an e-commerce platform, **Binary Search** is more efficient for large datasets because of its O(log n) time complexity. However, it requires the product list to be sorted by product name.

For smaller or unsorted datasets, **Linear Search** is simpler to implement. In real-world applications, platforms often use optimized search techniques such as **HashMap-based lookups for product IDs (O(1))** or database indexing to achieve the fastest possible search performance.

**Exercise 7: Financial Forecasting**

**Recursion:**  
Recursion is a programming technique where a method calls itself repeatedly until a base condition is met. It simplifies problems by breaking them down into smaller, similar sub-problems. Recursion is often used in tasks like factorial calculation, tree traversal, and financial forecasting where a value is repeatedly updated based on a rule.  
It replaces repetitive loops with a clean, self-calling method structure that makes code easier to read for problems with a recursive pattern.

**Java code:**

**FinancialForecast.java**

package forecasting;

public class FinancialForecast {

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

if (years == 0) {

return currentValue;

} else {

return *predictValue*(currentValue \* (1 + growthRate / 100), growthRate, years - 1);

}

}

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 7.5;

int years = 5

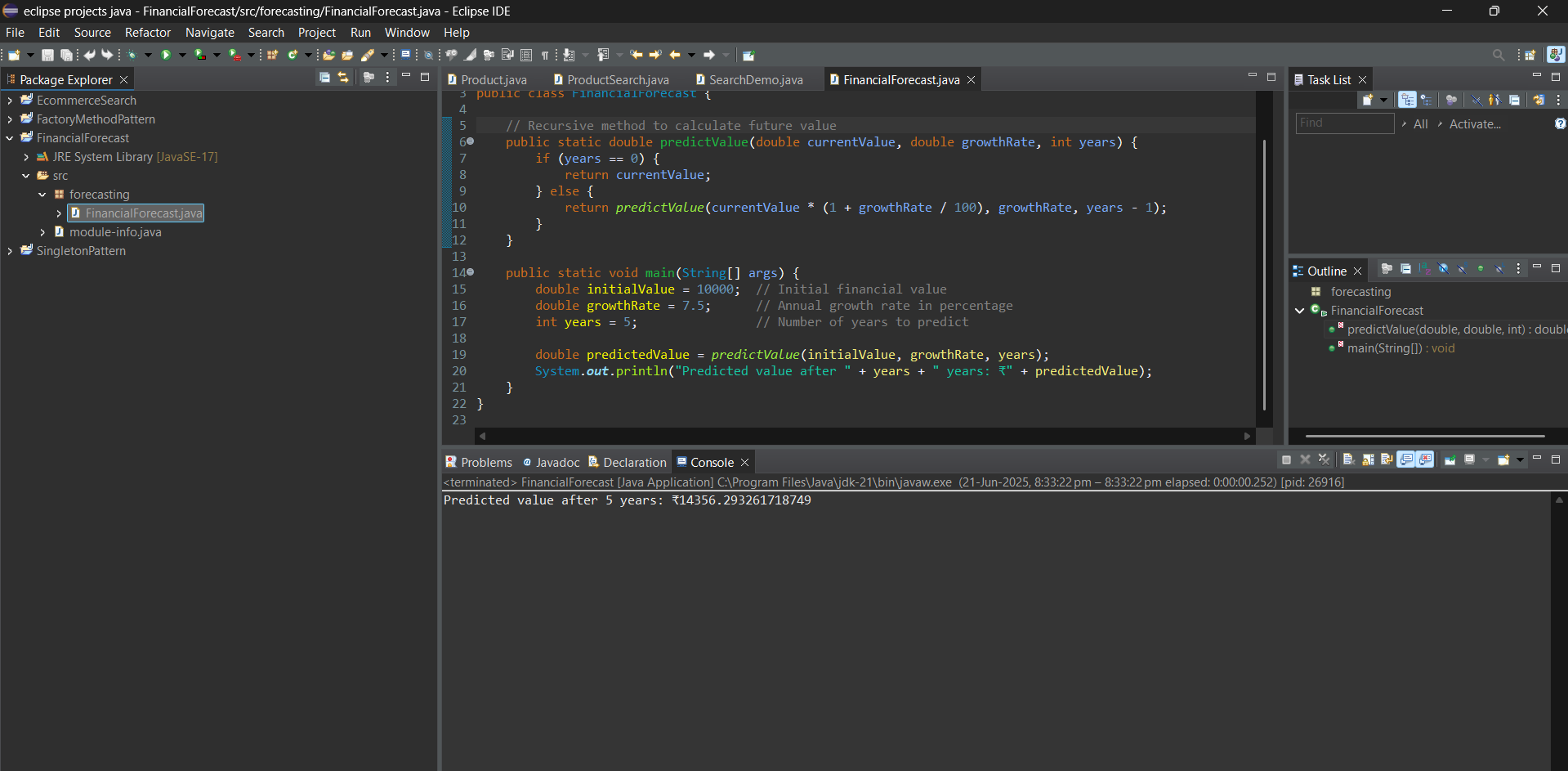
double predictedValue = *predictValue*(initialValue, growthRate, years);

System.***out***.println("Predicted value after " + years + " years: ₹" + predictedValue);

}

}

**Output:**



**Time Complexity Analysis**

* **Time Complexity:** O(n)
  + The function makes **one recursive call per year**, so if years = n, it makes n calls.
* **Space Complexity:** O(n)
  + Due to the call stack growing with each recursive call until the base case is reached.

**Optimization**

**Problem:**  
For large values of years, recursion increases call stack size and may cause stack overflow.

**Optimizations:**

* Convert recursion to iteration using a simple loop.
* Use memoization (in problems with overlapping subproblems like Fibonacci).