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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

1. **Understand Asymptotic Notation**

* **Big O Notation**

Describes how an algorithm’s running time grows as the input size n increases.

* O(1) – constant time
* O(log n) – logarithmic time
* O(n) – linear time
* O(n log n) – linearithmic time
* O(n²) – quadratic time
* **Search Scenarios**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| Linear | O(1) (first element) | O(n) (middle) | O(n) (last/not found) |
| Binary | O(1) (middle match) | O(log n) | O(log n) |

1. **Setup Your Classes**

**Package**Under src, create a package: search.

**Product.java**

package search;

public class Product {

private int productId;

private String productName;

private String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

@Override

public String toString() {

return String.format("Product[id=%d, name=%s, cat=%s]", productId, productName, category);

}

}

1. **Implement Linear & Binary Search**

**SearchAlgorithms.java**

package search;

import java.util.Arrays;

import java.util.Comparator;

public class SearchAlgorithms {

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

for (Product p : products) {

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

return p;

}

}

return null;

}

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

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

while (low <= high) {

int mid = (low + high) / 2;

String midName = products[mid].getProductName();

int cmp = midName.compareToIgnoreCase(targetName);

if (cmp == 0)

return products[mid];

else if (cmp < 0)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

public static void sortByName(Product[] products) {

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

}

}

1. **Test & Compare**

**SearchDemo.java**

package search;

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

public static void main(String[] args) {

Product[] products = {

new Product(101, "Shoes", "Footwear"),

new Product(102, "Laptop", "Electronics"),

new Product(103, "Chair", "Furniture"),

new Product(104, "Mobile", "Electronics"),

new Product(105, "Watch", "Accessories")

};

Product result1 = SearchAlgorithms.linearSearch(products, "Mobile");

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

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

SearchAlgorithms.sortByName(products);

Product result2 = SearchAlgorithms.binarySearch(products, "Laptop");

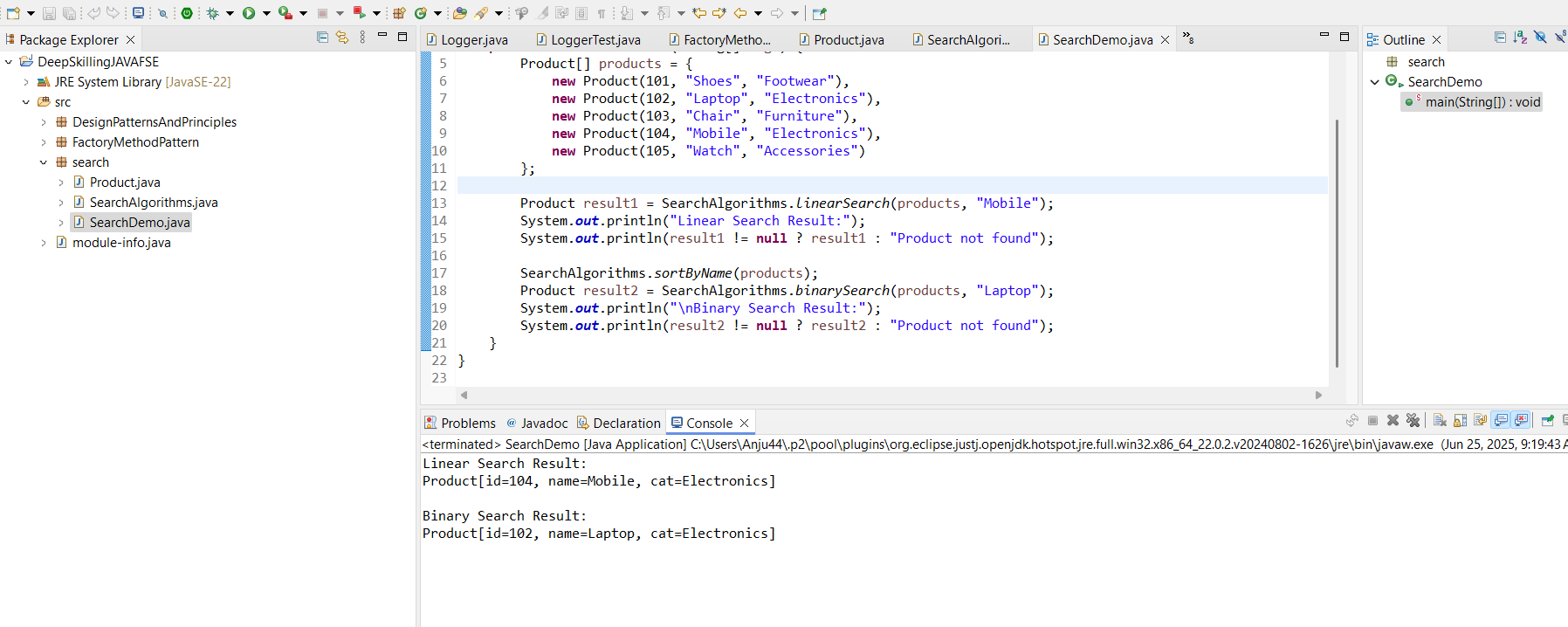
System.out.println("\nBinary Search Result:");

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

}

}

**OUTPUT**

****

1. **Analysis & Recommendation**

**Aspect Linear Search Binary Search**

Time Complexity O(n) O(log n)

Data Requirement Unsorted list Sorted list required

Use Case Small data, simple logic Large datasets, frequent search

Code Simplicity Very simple Requires sorting (O(n log n))

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

1. **Understand Recursive Algorithms**

Recursion is a method where a function calls itself to solve smaller subproblems. It simplifies problems like repetitive growth, tree traversal, etc.

* **Base case**: The condition when recursion stops (e.g., year = 0).
* **Recursive case:** When the function keeps calling itself with reduced input (e.g., year - 1).

To compute future value at year n:

FV(n) = FV(n - 1) × (1 + rate)

Base Case: FV(0) = initial

1. **Setup Your Classes**

**Package**

Under src, create a package: forecasting.

**Forecasting.java**

package forecasting;

public class Forecasting {

public static double forecastRecursive(double initialValue, double rate, int year) {

if (year < 0) throw new IllegalArgumentException("Year cannot be negative");

if (year == 0) return initialValue;

return forecastRecursive(initialValue, rate, year - 1) \* (1 + rate);

}

public static double forecastTailRecursive(double initialValue, double rate, int year) {

return helper(initialValue, rate, year, initialValue);

}

private static double helper(double initial, double rate, int year, double acc) {

if (year == 0) return acc;

return helper(initial, rate, year - 1, acc \* (1 + rate));

}

public static double forecastIterative(double initialValue, double rate, int year) {

double value = initialValue;

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

value \*= (1 + rate);

}

return value;

}

}

1. **Test & Compare**

package forecasting;

public class ForecastingTest {

public static void main(String[] args) {

double initial = 1000.0;

double rate = 0.07; // 7% annual growth

int year = 5;

double vRec = Forecasting.forecastRecursive(initial, rate, year);

double vTail = Forecasting.forecastTailRecursive(initial, rate, year);

double vIter = Forecasting.forecastIterative(initial, rate, year);

System.out.printf("Recursive: %.2f%n", vRec);

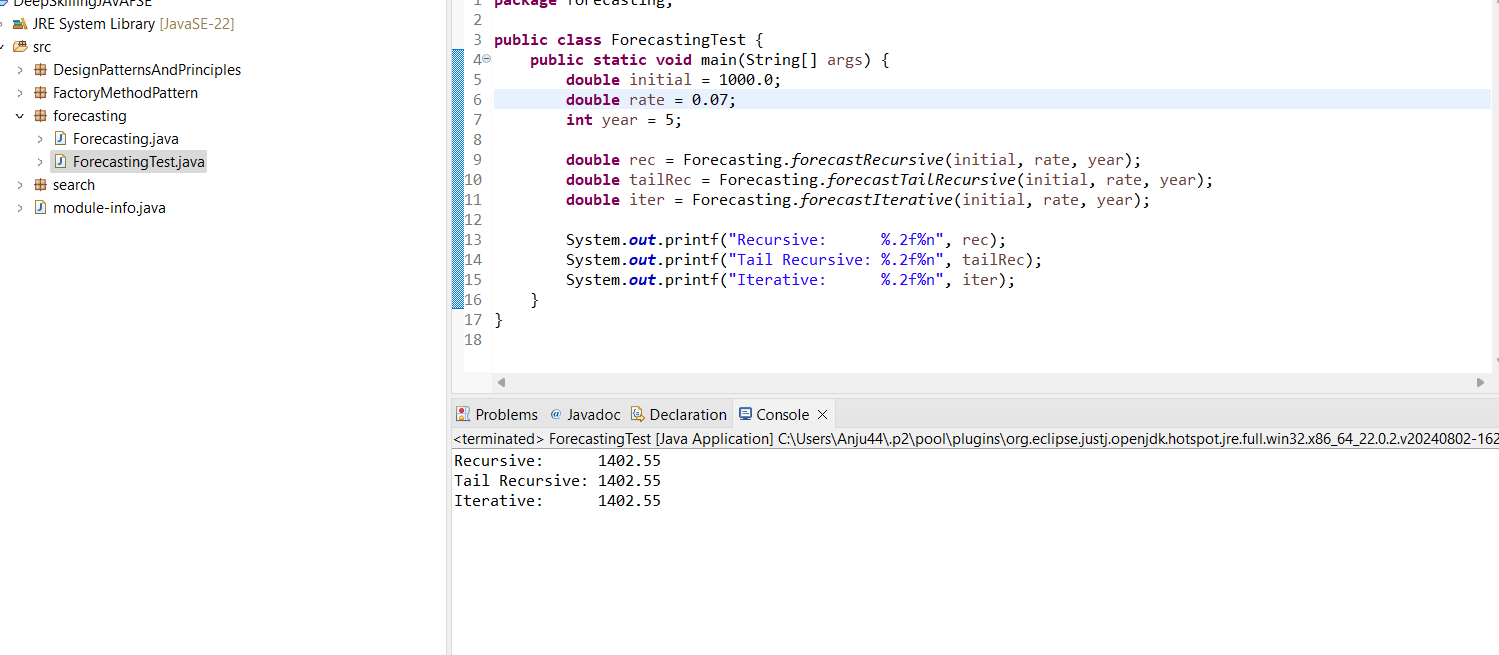
System.out.printf("Tail Recursive: %.2f%n", vTail);

System.out.printf("Iterative: %.2f%n", vIter);

}

}

**OUTPUT**

****

1. **Theoretical Analysis & Optimization**

**Recursion Explained**

* A recursive method calls itself with smaller input.
* Solves problems by breaking them down.

**Recursive Forecast Formula**

FV(n) = FV(n - 1) × (1 + rate)

Base Case: FV(0) = initial

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Version** | **Time Complexity** | **Space Complexity** |
| Recursive | O(n) | O(n) (stack) |
| Tail Recursive | O(n) | O(n)\* (may optimize to O(1)) |
| Iterative | O(n) | O(1) |