**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 grows.
  + **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) (found at first index) | O(n) (on average at n/2) | O(n) (not found or last) |
| **Binary** | O(1) (found at middle) | O(log n) | O(log n) |

**2. Setup Your Classes**

1. **Package**  
   Under src, create a package search.
2. **Product Class**

package search;

public class Product {

private final int productId;

private final String productName;

private final 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);

}

}

**3. Implement Linear & Binary Search**

In the same package, create a SearchUtils class:

package search;

import java.util.Arrays;

public class SearchUtils {

public static int linearSearch(Product[] arr, int targetId) {

for (int i = 0; i < arr.length; i++) {

if (arr[i].getProductId() == targetId) {

return i;

}

}

return -1;

}

public static int binarySearch(Product[] arr, int targetId) {

int lo = 0, hi = arr.length - 1;

while (lo <= hi) {

int mid = lo + (hi - lo) / 2;

int midId = arr[mid].getProductId();

if (midId == targetId) {

return mid;

} else if (midId < targetId) {

lo = mid + 1;

} else {

hi = mid - 1;

}

}

return -1;

}

public static void sortById(Product[] arr) {

Arrays.sort(arr, (a,b) -> Integer.compare(a.getProductId(), b.getProductId()));

}

}

**4. Test & Compare**

Create SearchTest.java:

package search;

import java.util.Random;

public class SearchTest {

public static void main(String[] args) {

Product[] products = new Product[10];

for (int i = 0; i < products.length; i++) {

products[i] = new Product(100 + i, "Item" + i, "Category" + (i%3));

}

int target = 104;

int idxLin = SearchUtils.linearSearch(products, target);

System.out.println("Linear search found at index: " + idxLin

+ ", element: " + (idxLin>=0 ? products[idxLin] : "N/A"));

java.util.List<Product> list = java.util.Arrays.asList(products);

java.util.Collections.shuffle(list, new Random());

products = list.toArray(new Product[0]);

SearchUtils.sortById(products);

int idxBin = SearchUtils.binarySearch(products, target);

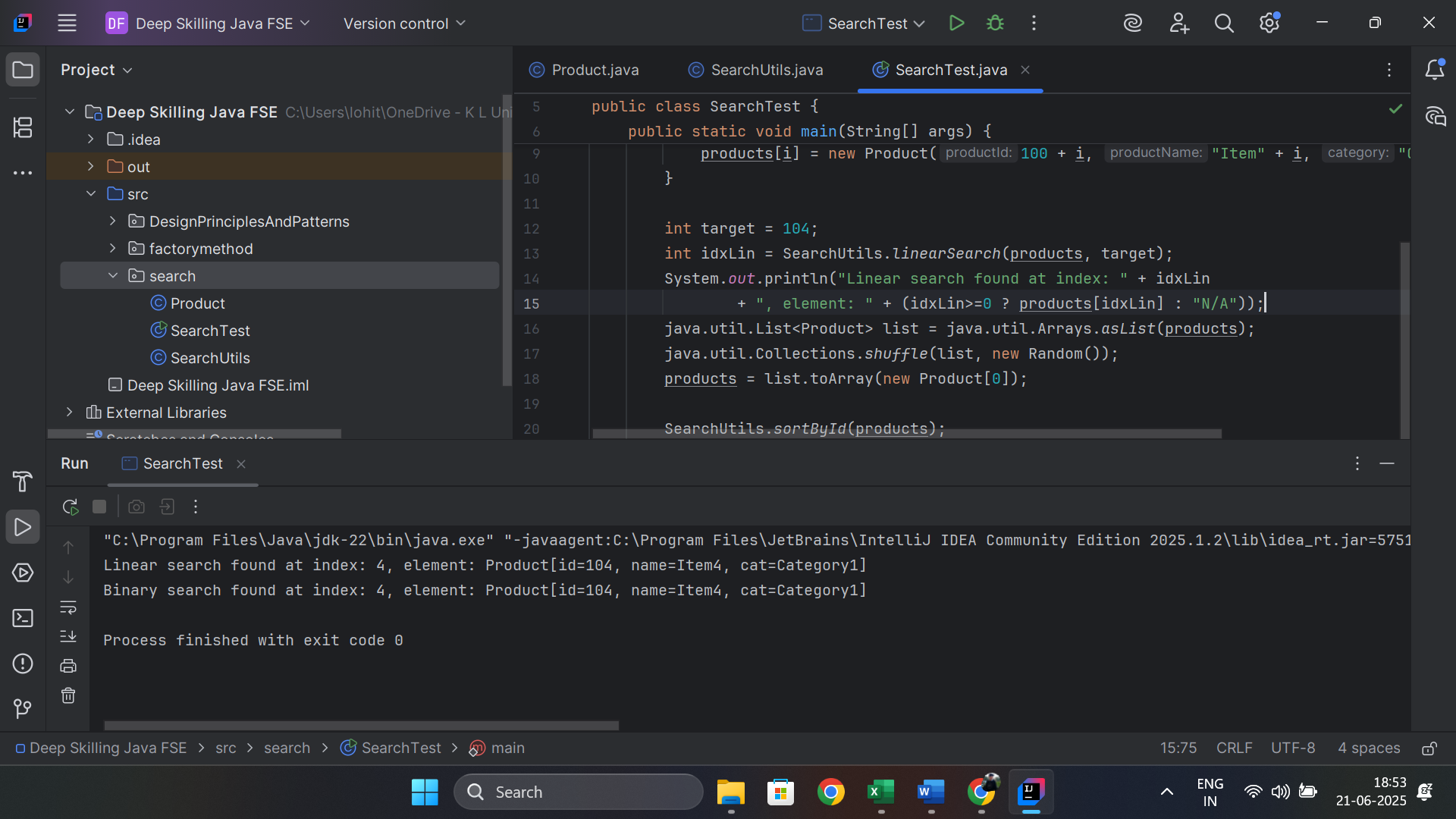
System.out.println("Binary search found at index: " + idxBin

+ ", element: " + (idxBin>=0 ? products[idxBin] : "N/A"));

}

}

**Output**



**5. Analysis & Recommendation**

| **Aspect** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Time Complexity** | O(n) | O(log n) |
| **Data Requirement** | any order | *must* be sorted |
| **Use Case** | small lists | large, immutable/search‑heavy datasets |
| **Overhead** | minimal code | need sorting step (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** means a method calls itself to solve a smaller instance of the same problem.
* You always need:
  1. **Base case** – when to stop recursing
  2. **Recursive case** – how to reduce the problem and recurse

**Why recursion helps here:**  
If we want to compute the future value in year *n* based on the previous year’s value and a growth rate, you can express:

FV(0) = initialValue

FV(n) = FV(n - 1) × (1 + rate)

Each call to FV(n) delegates to FV(n-1) until you hit n == 0.

**2. Setup: Method Signature**

Create a class Forecasting with a method:

/\*\*

\* @param initialValue the value at year 0

\* @param rate annual growth rate (e.g. 0.05 for 5%)

\* @param year the target year (n ≥ 0)

\* @return the forecasted value at year n

\*/

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

**3. Implementation**

package DesignPrinciplesAndPatterns.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;

}

double prevValue = forecastRecursive(initialValue, rate, year - 1);

return prevValue \* (1 + rate);

}

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

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

}

private static double forecastTRHelper(double initialValue, double rate,

int year, double accValue) {

if (year == 0) {

return accValue;

}

return forecastTRHelper(initialValue, rate, year - 1, accValue \* (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;

}

}

**Test it** in a ForecastingTest class:

package DesignPrinciplesAndPatterns.forecasting;

public class ForecastingTest {

public static void main(String[] args) {

double initial = 1000.0;

double rate = 0.07; // 7% per year

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);

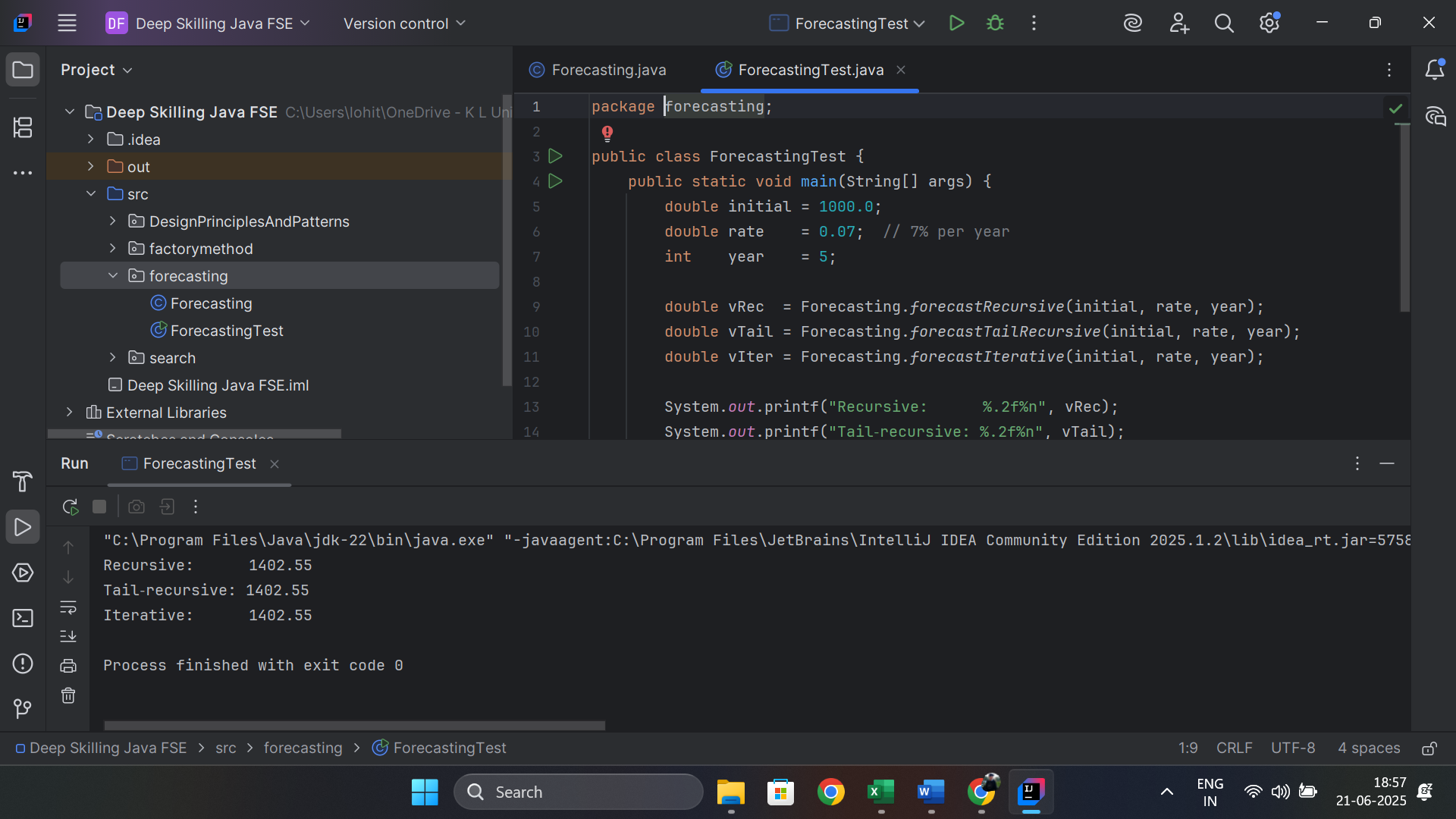
System.out.printf("Tail‑recursive: %.2f%n", vTail);

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

}

}

**Output:**



**4. Analysis & Optimization**

1. **Time Complexity**
   * **Recursive & Iterative** versions both perform *n* multiplications → **O(n)**.
2. **Space Complexity**
   * **Naïve recursion** uses *n* stack frames → **O(n)** space.
   * **Tail recursion** (if optimized by the JVM) can run in **O(1)** stack space.
   * **Iterative** always uses **O(1)** space.
3. **Optimizations**
   * **Use the iterative** version for production: it’s straightforward and constant‑space.
   * **Memoization** isn’t needed here because each forecastRecursive(k) is called exactly once in the chain.
   * If you had overlapping subproblems (e.g., forecasting based on multiple previous years), you’d cache results in an array or map to avoid exponential blow‑up.