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

Big O notation is used to describe the upper bound of an algorithm’s running time or space as the input size grows. It helps analyse and compare algorithms based on efficiency, regardless of hardware.

* Best Case: The item is found at the first try (e.g., first index).
* Average Case: The item is somewhere in the middle.
* Worst Case: The item is not found or is at the last index.

|  |  |  |  |
| --- | --- | --- | --- |
| **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) |

**Code:**

Product.java

package Ecommerce;  
  
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;  
 }  
  
 public String toString() {  
 return productId + " - " + productName + " (" + category + ")";  
 }  
}

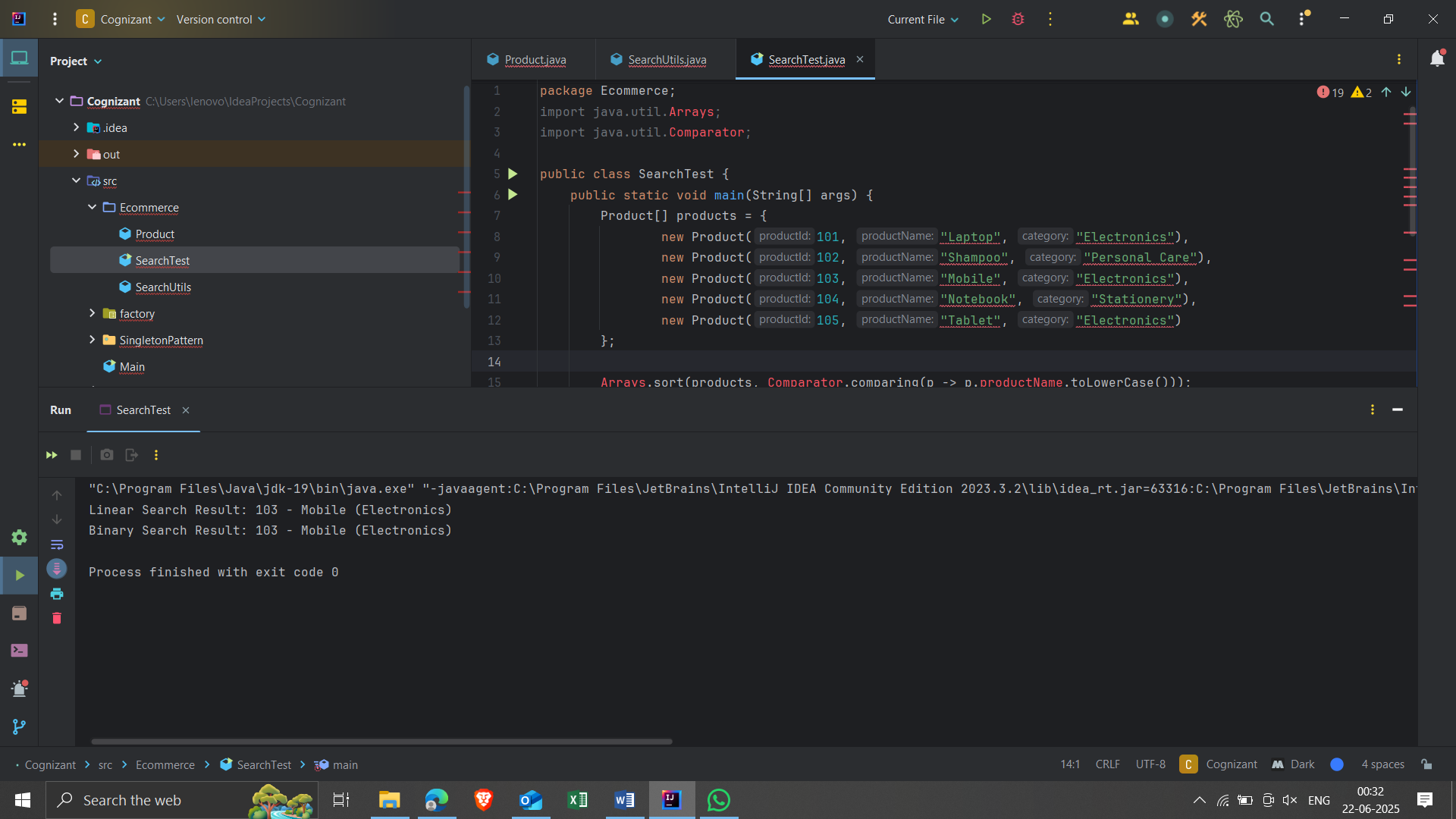
SearchUtils.java

package Ecommerce;  
  
public class SearchUtils {  
 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) {  
 int left = 0, right = products.length - 1;  
 while (left <= right) {  
 int mid = left + (right - left) / 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;  
 }  
}

SearchTest.java

package Ecommerce;  
import java.util.Arrays;  
import java.util.Comparator;  
  
public class SearchTest {  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(101, "Laptop", "Electronics"),  
 new Product(102, "Shampoo", "Personal Care"),  
 new Product(103, "Mobile", "Electronics"),  
 new Product(104, "Notebook", "Stationery"),  
 new Product(105, "Tablet", "Electronics")  
 };  
  
 Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));  
  
 // Linear Search Test  
 Product result1 = SearchUtils.*linearSearch*(products, "Mobile");  
 System.*out*.println("Linear Search Result: " + result1);  
  
 // Binary Search Test  
 Product result2 = SearchUtils.*binarySearch*(products, "Mobile");  
 System.*out*.println("Binary Search Result: " + result2);  
 }  
}

**Output:**



**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Suitable For** |
| **Linear Search** | O(n) | Small data, unsorted list |
| **Binary Search** | O(log n) | Large, sorted list |

**Exercise 7: Financial Forecasting**

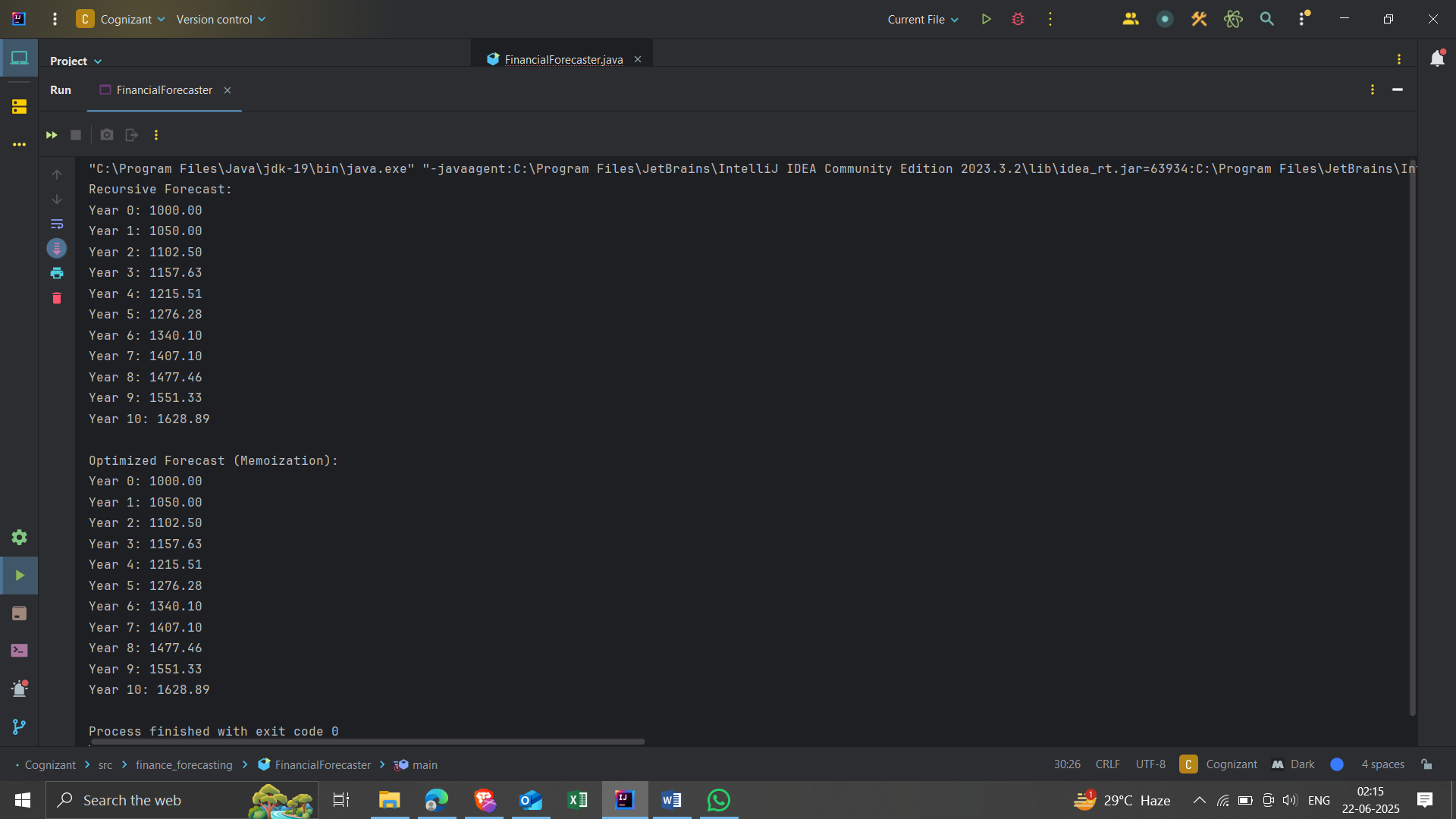
Recursion is a method where a function **calls itself** to solve a smaller version of the original problem. It's useful for problems that can be broken down into smaller, similar subproblems (e.g., Fibonacci sequence, factorial, tree traversal).Best Case: The item is found at the first try (e.g., first index). Recursion helps model **repeated calculations over time**, such as forecasting values for n future periods based on previous values and a growth rate.

**Code:**

FinancialForecaster.java

package finance\_forecasting;  
  
public class FinancialForecaster {  
 // Recursive method to calculate future value  
 public static double futureValueRecursive(double initialValue, double growthRate, int years) {  
 if (years == 0) {  
 return initialValue; // base case  
 }  
 return *futureValueRecursive*(initialValue, growthRate, years - 1) \* (1 + growthRate);  
 }  
  
 // Optimized with memoization  
 public static double futureValueMemo(double initialValue, double growthRate, int years, double[] memo) {  
 if (years == 0) return initialValue;  
 if (memo[years] != 0) return memo[years];  
 memo[years] = *futureValueMemo*(initialValue, growthRate, years - 1, memo) \* (1 + growthRate);  
 return memo[years];  
 }  
  
 public static void main(String[] args) {  
 double initial = 1000.0;  
 double rate = 0.05; // 5%  
 int years = 10;  
  
 System.*out*.println("Recursive Forecast:");  
 for (int i = 0; i <= years; i++) {  
 System.*out*.printf("Year %d: %.2f%n", i, *futureValueRecursive*(initial, rate, i));  
 }  
  
 System.*out*.println("\nOptimized Forecast (Memoization):");  
 double[] memo = new double[years + 1];  
 for (int i = 0; i <= years; i++) {  
 System.*out*.printf("Year %d: %.2f%n", i, *futureValueMemo*(initial, rate, i, memo));  
 }  
 }  
}

**Output:**



**Analysis:**

### Time Complexity:

* **Recursive without memoization**: O(n), but involves **n recursive calls**, each with its own stack frame.
* **Recursive with memoization**: O(n) **with much less overhead** — avoids repeated recalculations.

Optimization Tips:

* **Memoization** stores already-computed results → reduces computation time.
* For **large** n, prefer **iterative** or **DP-based** solutions to avoid stack overflow.