**Data Structure and Algorithm**

**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.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Explain Big O notation and how it helps in analyzing algorithms.**

Ans) Big O notation is a way to describe how fast or slow an algorithm is, based on how much input you give it. Think of it as a tool to measure performance or efficiency, especially when dealing with large data sets.

Instead of counting exact steps, Big O tells you the rate of growth—how the time (or memory) increases as the input grows.

Common Big O examples:

* O(1) → Constant time: Always takes the same amount of time, no matter the input size.
* O(n) → Linear time: Time grows directly with input. If input doubles, time doubles.
* O(n²) → Quadratic time: Time grows fast as input grows. If input doubles, time becomes 4x longer.

It helps programmers compare algorithms and pick the most efficient one, especially when the program needs to handle lots of data.

**Describe the best, average, and worst-case scenarios for search operations.**

When we analyze an algorithm like searching, there are three ways to measure its performance, based on different situations:

Best Case:

* The ideal situation.
* Example: In linear search, the item you’re looking for is the first element in the list.
* Result: Fastest possible time (e.g., O(1) for linear search).

Average Case:

* Usually happens on average.
* Example: The item might be somewhere in the middle of the list.
* Result: Moderate time (e.g., O(n/2), but still simplified to O(n) in Big O).

Worst Case:

* The slowest situation.
* Example: The item is at the end of the list or not in the list at all.
* Result: Longest time (e.g., O(n) for linear search, O(log n) for binary search).

**Product.java**

package ecommerce;

public class Product {

int prodId;

String prodName;

String category;

public Product(int prodId, String prodName, String category) {

this.prodId = prodId;

this.prodName = prodName;

this.category = category;

}

public String toString() {

return "[" + prodId + ", " + prodName + ", " + category + "]";

}

}

package ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchFunction {

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

for (Product product : products) {

if (product.prodName.equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int comparison = products[mid].prodName.compareToIgnoreCase(targetName);

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void sortProductsByName(Product[] products) {

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

}

}

**SearchFunction.java**

**package** ecommerce;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** SearchFunction {

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

**for** (Product product : products) {

**if** (product.prodName.equalsIgnoreCase(targetName)) {

**return** product;

}

}

**return** **null**;

}

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

**int** low = 0;

**int** high = products.length - 1;

**while** (low <= high) {

**int** mid = (low + high) / 2;

**int** comparison = products[mid].prodName.compareToIgnoreCase(targetName);

**if** (comparison == 0) {

**return** products[mid];

} **else** **if** (comparison < 0) {

low = mid + 1;

} **else** {

high = mid - 1;

}

}

**return** **null**;

}

**public** **static** **void** sortProductsByName(Product[] products) {

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

}

}

**TestSearch.java**

package ecommerce;

public class TestSearch {

public static void main(String[] args) {

Product[] products = {

new Product(501, "Laptop", "Electronics"),

new Product(502, "Shoes", "Footwear"),

new Product(503, "Keyboard", "Electronics"),

new Product(504, "T-Shirt", "Apparel"),

new Product(505, "Mobile", "Electronics")

};

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

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

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

SearchFunction.*sortProductsByName*(products);

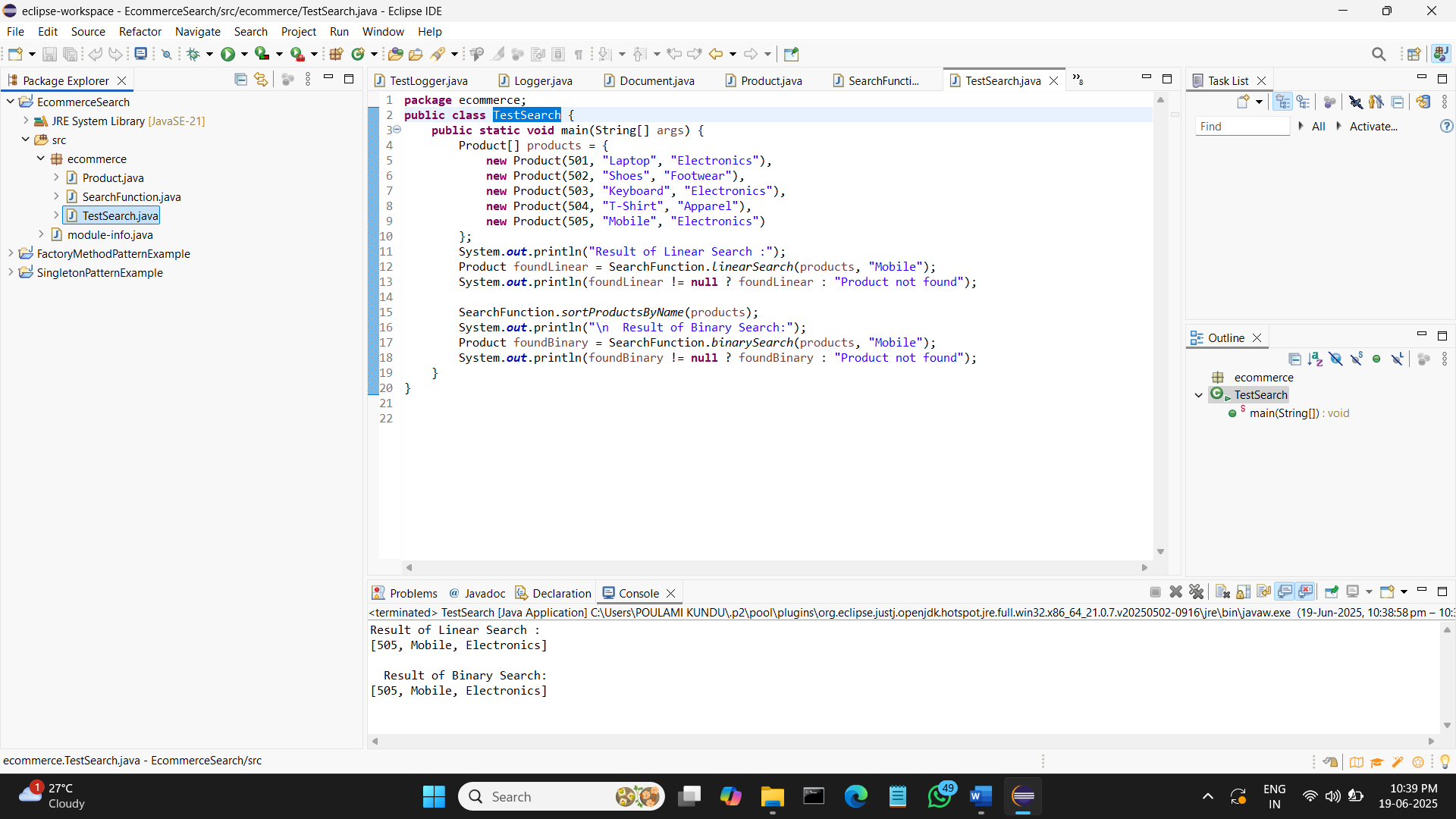
System.*out*.println("\n Result of Binary Search:");

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

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

}

}

**OUTPUT:**

**Compare the time complexity of linear and binary search algorithms.**

Ans)Linear Search:

In a linear search, the algorithm goes through each element in the array one by one to find the target value. If the target is found early, the algorithm stops. If not, it continues checking each element until it either finds the value or reaches the end of the array.

* In the best case, the target is found in the first position, so only one comparison is needed. This gives us a time complexity of O(1).
* In the average case, the target might be located somewhere in the middle of the array, so the algorithm needs to check about half of the elements. Still, we represent this as O(n) in Big O notation.
* In the worst case, the algorithm has to check every element in the array (either the last item is the target or the target isn't present at all), resulting in a time complexity of O(n).

So, linear search can become slow as the size of the data increases because it performs a comparison for each element.

Binary Search:

Binary search is much faster, but it only works on a sorted array. Instead of checking each element, it divides the array in half with each step and compares the middle element to the target.

* In the best case, the target happens to be the middle element of the array, and the algorithm finds it immediately, taking just one step — so the time complexity is O(1).
* In both the average case and worst case, the algorithm repeatedly splits the array in half, which drastically reduces the number of comparisons. This leads to a time complexity of O(log n), where n is the number of elements.

In other words, binary search is logarithmic, meaning that even for very large arrays, it only needs a small number of comparisons to find (or fail to find) the target.

**Discuss which algorithm is more suitable for your platform and why.**

Ans) Binary search is more suitable for an e-commerce platform because it offers much faster search performance on large, sorted datasets, which is typical in such systems. Since products are usually stored in sorted order or indexed, binary search can quickly locate items, improving user experience. Linear search is simple but becomes slow with large data, so it’s only ideal for very small or unsorted lists.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Solution:**

**Forecasting.java**

package forecast;

public class Forecasting {

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

if (years == 0) {

return present;

}

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

}

public static void main(String[] args) {

double present = 11000;

double annualGrowthRate = 0.08;

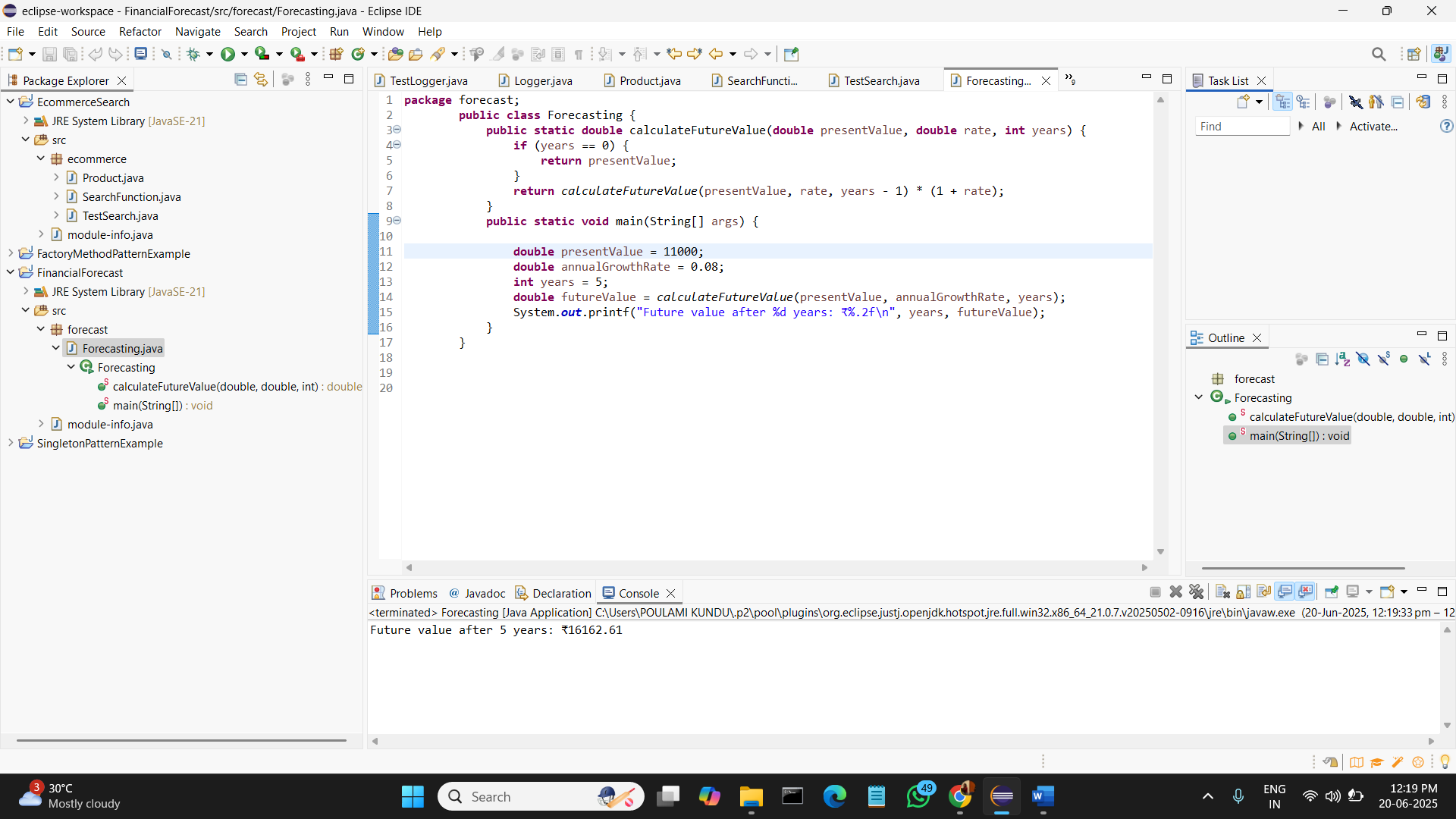
int years = 5;

double futureValue = *calculateFutureValue*(present, annualGrowthRate, years);

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

}

}

 **Explain the concept of recursion and how it can simplify certain problems.**

**Ans)** Recursion is a programming method where a function calls itself to solve smaller parts of a problem. It simplifies problems that have a repetitive or nested structure, like calculating factorials, Fibonacci series, or tree traversals. Instead of using complex loops, recursion breaks the task into smaller steps, making the code cleaner and easier to understand.

**Discuss the time complexity of your recursive algorithm.**

**Ans)** The time complexity of the recursive algorithm to calculate future value is **O(n)**, where n is the number of years. This is because the function calls itself once for each year until it reaches the base case. Although it's simple and easy to write, it can be inefficient for large values of n due to the repeated function calls and increased stack usage. An iterative approach would be more efficient in such cases.

**Explain how to optimize the recursive solution to avoid excessive computation.**

**Ans)** To optimize the recursive solution and avoid excessive computation, you can:

1. Use Iteration Instead of Recursion:  
   Replace recursion with a loop. This avoids the overhead of multiple function calls and reduces memory usage by not using the call stack.
2. Apply Memoization (if values are reused):  
   Store results of previous calculations in a cache (like an array or map). If the same input is needed again, return the stored result instead of recalculating.
3. Tail Recursion (if supported):  
   In some languages, tail recursion can be optimized by the compiler into iteration. Java doesn't support tail call optimization directly, so this may not help much in Java.

For simple problems like financial forecasting where previous results aren’t reused, iteration is usually the most efficient optimization.