**Data Structures and Algorithms**

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

**1. Big O Notation and Its Importance in Algorithm Analysis**

Big O notation is used to describe how the time or space requirement of an algorithm grows as the input size increases. It helps us understand the performance of algorithms in different conditions, especially when working with large amounts of data. By using Big O, we can analyse which algorithm is more efficient and suitable for a particular use case.

For example, if one algorithm takes O(n) time and another takes O(log n), the second one will be faster for large inputs.

**2. Best, Average, and Worst-Case Scenarios for Search Operations**

**In linear search:**

* Best case: The element is found at the first position – Time complexity: O(1).
* Average case: The element is somewhere in the middle – Time complexity: O(n).
* Worst case: The element is at the last position or not found – Time complexity: O(n).

**In binary search (on sorted data):**

* Best case: The element is found at the middle – Time complexity: O(1).
* Average case: The element is found after a few comparisons – Time complexity: O(log n).
* Worst case: The element is not found after maximum divisions – Time complexity: O(log n).

**3. Product Class Setup**

A class named Product is created to represent each item in the e-commerce platform. The class contains the following attributes:

* productId – A unique number to identify the product.
* productName – The name of the product.
* category – The type of product (e.g., Electronics, Fashion).

These attributes are used during search operations to locate specific products.

**4. Implementation of Search Algorithms**

Two algorithms are implemented for searching:

* Linear Search: This method checks each product in the list one by one until the required product is found. It does not require the list to be sorted.
* Binary Search: This method works only on sorted lists. It starts by comparing the target with the middle element and repeatedly divides the list in half until the element is found or the list is empty. It is faster than linear search for large datasets.

In this implementation:

* An array is used to store products.
* For linear search, the array can be unsorted.
* For binary search, the array is sorted based on the product ID.

**5. Time Complexity Comparison**

|  |  |  |
| --- | --- | --- |
| **Search Method** | **Time Complexity** | **Requires Sorted Array** |
| Linear Search | |  | | --- | |  |  |  | | --- | | O(n) | | No |
| Binary Search | |  | | --- | |  |  |  | | --- | | O(log n) | | Yes |

**6. Suitable Algorithm for the E-commerce Platform**

Binary search is more suitable for large e-commerce platforms because it is much faster than linear search when working with sorted product data. It helps in improving the user experience by providing quick search results. Linear search can be used when the data is small or not sorted, but for performance and scalability, binary search is the better choice.

**CODE:**

1. **Product.java**

**package** com.example.search;

**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** **void** display() {

System.***out***.println(productId + " - " + productName + " (" + category + ")");

}

}

1. **SearchDemo.java**

**package** com.example.search;

**import** java.util.\*;

**public** **class** SearchDemo {

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

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

**if** (p.productId == targetId) {

**return** p;

}

}

**return** **null**;

}

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

**int** low = 0;

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

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

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

**if** (products[mid].productId == targetId) {

**return** products[mid];

} **else** **if** (products[mid].productId < targetId) {

low = mid + 1;

} **else** {

high = mid - 1;

}

}

**return** **null**;

}

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

//product list

Product[] products = {

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

**new** Product(205, "Shoes", "Footwear"),

**new** Product(150, "Watch", "Accessories"),

**new** Product(110, "Mobile", "Electronics")

};

System.***out***.print("Enter Product ID to search: ");

**int** searchId = sc.nextInt();

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

Product result1 = *linearSearch*(products, searchId);

**if** (result1 != **null**)

result1.display();

**else**

System.***out***.println("Product not found!");

// Sort for binary search

Arrays.*sort*(products, Comparator.*comparingInt*(p -> p.productId));

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

Product result2 = *binarySearch*(products, searchId);

**if** (result2 != **null**)

result2.display();

**else**

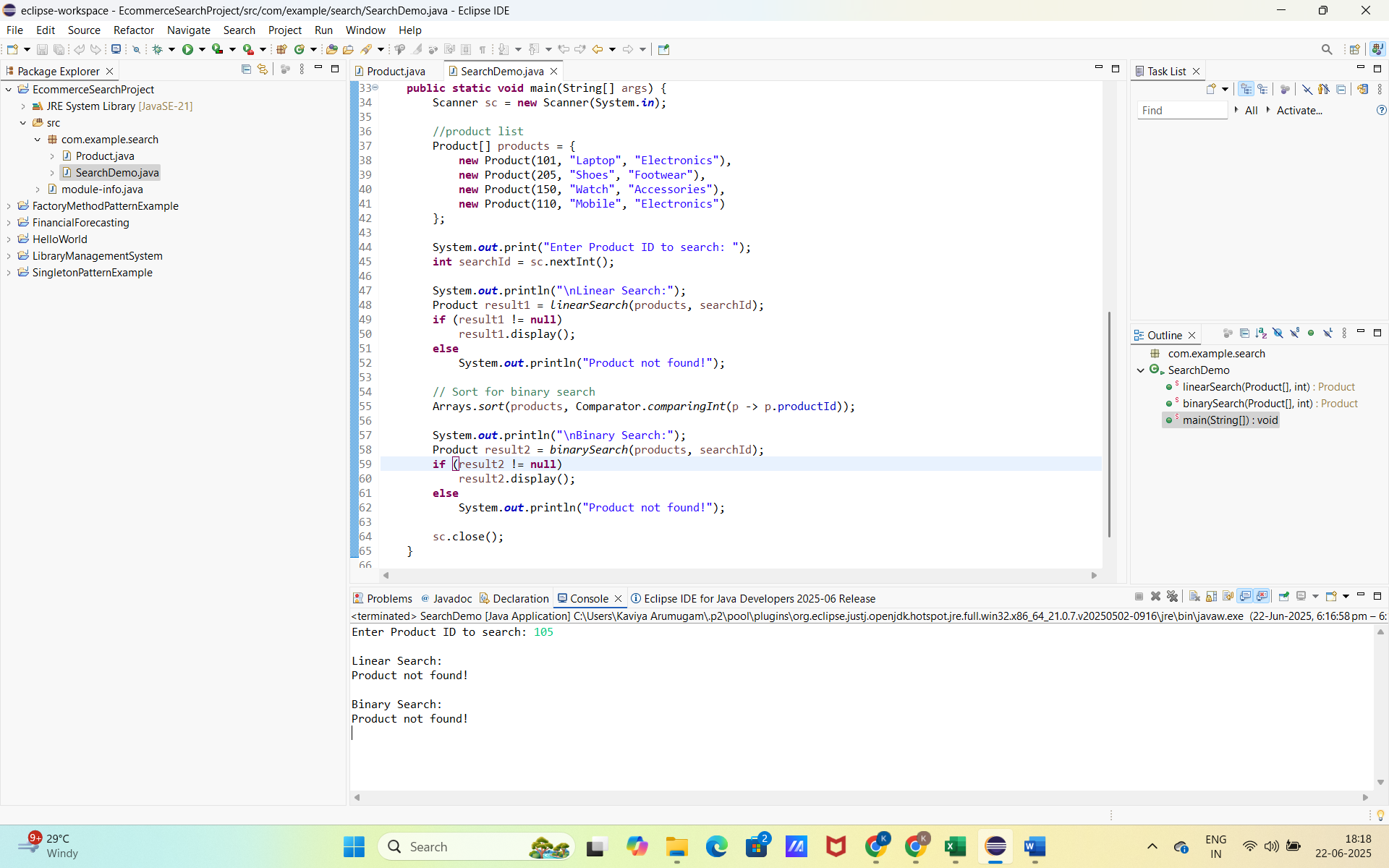
System.***out***.println("Product not found!");

sc.close();

}

}

**OUTPUT:**



A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 7: Financial Forecasting**

**1. Understanding Recursive Algorithms**

Recursion is a programming technique in which a method calls itself to solve a smaller version of the same problem. Recursive algorithms are useful when a problem can be broken down into smaller, similar subproblems. This makes the code cleaner and easier to understand in many cases.

For example, calculating future values based on yearly growth can be expressed recursively by multiplying the value each year by a growth factor until the target year is reached.

**2. Setup**

To predict future values, we create a method that takes three inputs:

* The initial value (starting amount)
* The annual growth rate (in percentage)
* The number of years

The method calculates the future value by applying the growth rate repeatedly for the given number of years using recursion.

**3. Implementation**

A recursive method is implemented where:

* The base case returns the initial value when the number of years is zero.
* The recursive case multiplies the current value by (1 + growth rate) and reduces the number of years by one.

This way, the method continues calling itself until all years are covered, and the final future value is returned.

**4. Analysis**

**Time Complexity:**  
The time complexity of the recursive algorithm is O(n), where *n* is the number of years. This is because the function is called once for each year.

**Optimization:**  
The recursive method is simple and efficient for a small number of years. However, for large values of *n* or cases with repeated calculations (like Fibonacci), recursion can become inefficient due to repeated computations and increased memory usage from the call stack.

To optimize:

* Use iteration instead of recursion for better performance and reduced stack usage.
* Memoization can also be used to store already computed values and avoid redundant calculations.

**CODE:**

1. **FinancialForecast.java**

**package** com.financial.forecasting;

**import** java.util.Scanner;

**public** **class** FinancialForecast {

// Recursive method to calculate future value

**public** **static** **double** forecast(**double** value, **double** rate, **int** years) {

**if** (years == 0) {

**return** value;

}

**return** *forecast*(value \* (1 + rate / 100), rate, years - 1);

}

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.print("Enter initial value (₹): ");

**double** initialValue = sc.nextDouble();

System.***out***.print("Enter annual growth rate (%): ");

**double** growthRate = sc.nextDouble();

System.***out***.print("Enter number of years: ");

**int** years = sc.nextInt();

**double** futureValue = *forecast*(initialValue, growthRate, years);

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

sc.close();

}

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.