**Week-1 Algorithms Data Structures**

1. **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.

**Solution:**

**Understanding Asymptotic Notation:**

**Big O Notation:**

Big O notation is a way to describe the performance (time or space) of an algorithm as the input size grows. It focuses on the worst-case growth rate, helping developers understand how an algorithm behaves with larger inputs.

**Best, average and worst-case scenarios for search operation:**

|  |  |  |  |
| --- | --- | --- | --- |
| Cases | Description | Linear Search | Binary Search |
| Best | Element at the first position for linear or middle for binary. | O(1) | O(1) |
| Average | Element is somewhere in the middle. | O(n/2) | O(log n) |
| Worst | Element is not present in the array | O(n) | O(log n) |

**Set Up:**

**Code:**

**Product.java**

**package** com.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;

}

@Override

**public** String toString() {

**return** "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

**Implementation:**

**SearchDemo.java**

**package** com.search;

**import** java.util.Arrays;

**import** java.util.Comparator;

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

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

Product[] products = {

**new** Product(201, "Apple", "Fruits"),

**new** Product(202, "Milk", "Dairy"),

**new** Product(203, "Bread", "Bakery"),

**new** Product(204, "Eggs", "Dairy"),

**new** Product(205, "Tomato", "Vegetables")

};

System.***out***.println("Linear Search Result (searching 'Bread'):");

Product result1 = *linearSearch*(products, "Bread");

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

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

System.***out***.println("\nBinary Search Result (searching 'Bread'):");

Product result2 = *binarySearch*(products, "Bread");

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

}

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

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

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

**return** product;

}

}

**return** **null**;

}

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

**int** left = 0, right = products.length - 1;

**while** (left <= right) {

**int** mid = (left + right) / 2;

**int** compare = targetName.compareToIgnoreCase(products[mid].productName);

**if** (compare == 0) **return** products[mid];

**else** **if** (compare < 0) right = mid - 1;

**else** left = mid + 1;

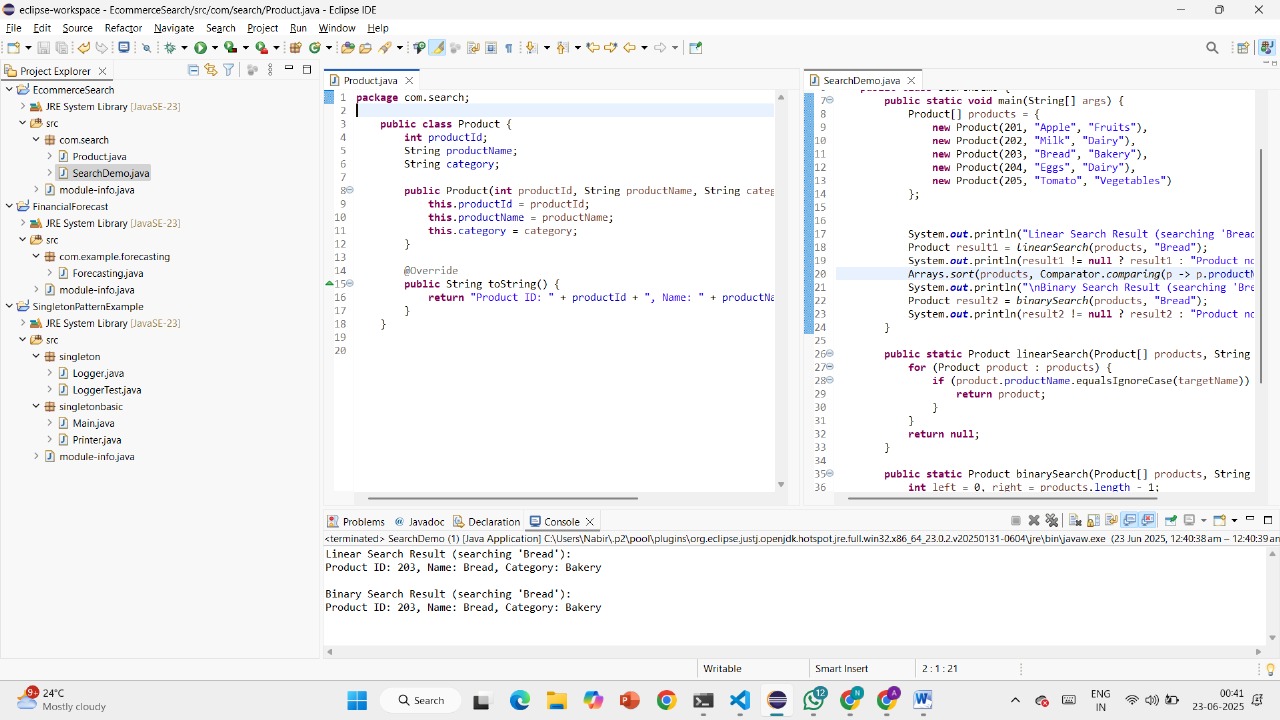
}

**return** **null**;

}

}

**Output:**

****

**Analysis:**

|  |  |
| --- | --- |
| Algorithm | Time Complexity |
| Linear Search | O(n) |
| Binary Search | O(log n) |

We can use Binary Search when the product list is very large. We can sort it only once and can search many times. It is suitable for fast searching of products.

We can use Linear Search when the product list is small. It does not need sorting.

1. **Financial Forecasting**

**Scenario:**

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

**Understanding recursive algorithms:**

In Java, Recursion is a process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. A few Java recursion examples are Inorder, Preorder, Postorder traversals, DFS of graph etc.

**Setup and Implementation:**

**Code:**

**package** com.example.forecasting;

**public** **class** Forecasting {

**public** **static** **double** futureValueRecursive(**double** presentValue, **double** rate, **int** years) {

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

**return** presentValue;

} **else** {

**return** *futureValueRecursive*(presentValue, rate, years - 1) \* (1 + rate);

}

}

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

**double** presentValue = 10000;

**double** rate = 0.05;

**int** years = 5;

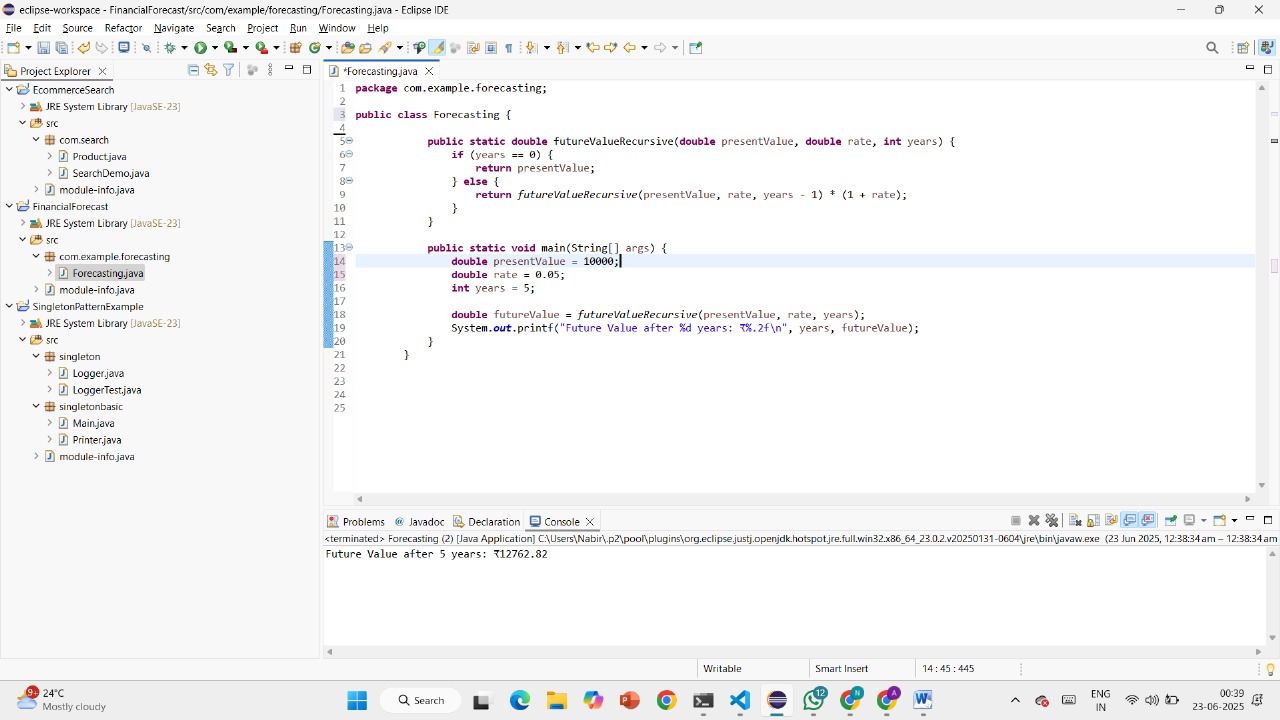
**double** futureValue = *futureValueRecursive*(presentValue, rate, years);

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

}

}

**Output:**

****

**Analysis:**

This code has O(n) time complexity. Each recursive call reduces the n number of years by 1 and therefore the recursive call occurs n times.