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.
      Big O Notation describes the upper bound of an algorithm's running time or space as the input size grows.
    - Describe the best, average, and worst-case scenarios for search operations.

Scenario	**Linear Search**	**Binary Search**
Best Case	O(1) (first match)	O(1) (middle match)
Average Case	O(n/2) ≈ O(n)	O(log n)
Worst Case	O(n)	O(log n)

Linear Search checks each element one by one.

Binary Search repeatedly divides the search interval in half — *requires sorted data*.

2. **Setup:**
    - Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

Product Class:

```
package com.searchFunction;

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 productId + ": " + productName + " [" + 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.

ProductSearch Class:

```
package com.searchFunction;
import java.util.Arrays;
import java.util.Comparator;
public class ProductSearch {

          // Linear Search by productName within a specific category
          public static Product linearSearch(Product[] products, String
name, String category) {
              for (Product product : products) {
                  if (product.category.equalsIgnoreCase(category) &&
                      product.productName.equalsIgnoreCase(name)) {
                      return product;
                  }
              }
              return null;
          }

          // Binary Search by productName in sorted array (filtered by
category beforehand)
          public static Product binarySearch(Product[] sortedProducts,
String name) {
              int left = 0, right = sortedProducts.length - 1;

              while (left <= right) {
                  int mid = (left + right) / 2;
                  int compare =
sortedProducts[mid].productName.compareToIgnoreCase(name);

                  if (compare == 0) return sortedProducts[mid];
                  else if (compare < 0) left = mid + 1;
                  else right = mid - 1;
              }
              return null;
          }
}
```

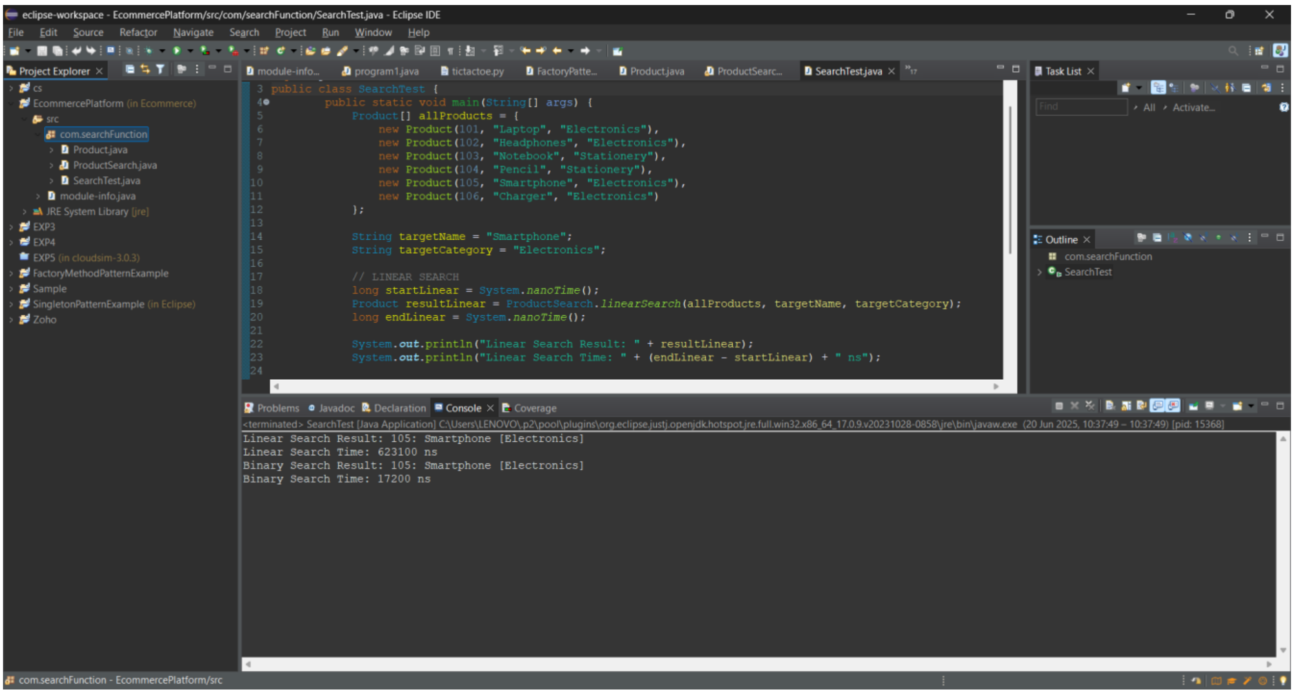
4. **Analysis:**
    - Compare the time complexity of linear and binary search algorithms.

SearchTest Class:

```
package com.searchFunction;
import java.util.*;
public class SearchTest {
          public static void main(String[] args) {
              Product[] allProducts = {
                  new Product(101, "Laptop", "Electronics"),
                  new Product(102, "Headphones", "Electronics"),
                  new Product(103, "Notebook", "Stationery"),
                  new Product(104, "Pencil", "Stationery"),
                  new Product(105, "Smartphone", "Electronics"),
                  new Product(106, "Charger", "Electronics")
```

```
                };

                String targetName = "Smartphone";
                String targetCategory = "Electronics";

                // LINEAR SEARCH
                long startLinear = System.nanoTime();
                Product resultLinear =
ProductSearch.linearSearch(allProducts, targetName, targetCategory);
                long endLinear = System.nanoTime();

                System.out.println("Linear Search Result: " + resultLinear);
                System.out.println("Linear Search Time: " + (endLinear -
startLinear) + " ns");

                // BINARY SEARCH
                Product[] filteredCategory = Arrays.stream(allProducts)
                        .filter(p ->
p.category.equalsIgnoreCase(targetCategory))
                        .toArray(Product[]::new);

                Arrays.sort(filteredCategory, Comparator.comparing(p ->
p.productName.toLowerCase()));

                long startBinary = System.nanoTime();
                Product resultBinary =
ProductSearch.binarySearch(filteredCategory, targetName);
                long endBinary = System.nanoTime();

                System.out.println("Binary Search Result: " + resultBinary);
                System.out.println("Binary Search Time: " + (endBinary -
startBinary) + " ns");
            }

}
```

Output Screenshot:



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

  Linear Search is simple and does not require sorting. Ideal for small or unsorted datasets.

  Binary Search is much faster on **pre-sorted** or **category-filtered** data.

  For an e-commerce platform, where **categories are already defined**, and **products can be sorted**, **Binary Search is highly efficient** and should be preferred.

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.

      Recursion is when a method **calls itself** to solve a smaller subproblem of the original problem.
      It works well when a problem can be broken down into similar sub-problems.

      Example Problem:
      Forecast a future investment value where each year's value is based on a constant growth rate.

      Mathematically:

      $FV(n) = FV(n - 1) * (1 + r)$

      $FV(0) = initialValue$

2. **Setup:**
    - Create a method to calculate the future value using a recursive approach.

FinancialForecast Class:

```
package com.forcasting;
import java.util.HashMap;
public class FinancialForecast {
          // Memoized recursive method
          public static double forecastFutureValue(double initial, double
rate, int year, HashMap<Integer, Double> memo) {
              if (year == 0) return initial;

              // Check if result already exists
              if (memo.containsKey(year)) {
                  return memo.get(year);
              }

              // Calculate recursively
              double previous = forecastFutureValue(initial, rate, year -
1, memo);
              double current = previous * (1 + rate);
              memo.put(year, current);  // Store the result)
              return current;
          }

}
```

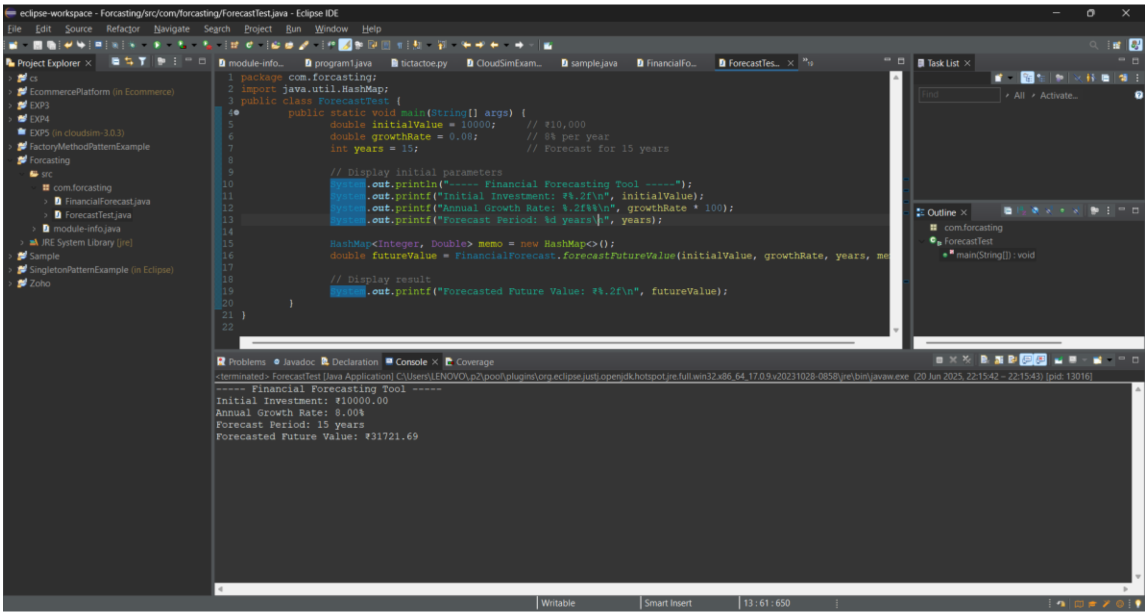
3. **Implementation:**

   - Implement a recursive algorithm to predict future values based on past growth rates.

ForecastTest Class:

```java
package com.forcasting;
import java.util.HashMap;
public class ForecastTest {
            public static void main(String[] args) {
                 double initialValue = 10000;      // ₹10,000
                   double growthRate = 0.08;         // 8% per year
                         int years = 15;                       // Forecast for 15 years

                         // Display initial parameters
                         System.out.println("----- Financial Forecasting Tool -----");
                         System.out.printf("Initial Investment: ₹%.2f\n", initialValue);
                         System.out.printf("Annual Growth Rate: %.2f%%\n", growthRate * 100);
                         System.out.printf("Forecast Period: %d years\n", years);

                         HashMap<Integer, Double> memo = new HashMap<>();
                         double futureValue = FinancialForecast.forecastFutureValue(initialValue, growthRate, years, memo);

                         // Display result
                         System.out.printf("Forecasted Future Value: ₹%.2f\n", futureValue);
                  }
        }
```

Output Screenshot:



4.Analysis:

- Discuss the time complexity of your recursive algorithm.

Time Complexity of Recursive Algorithm

The method:

forecastFutureValue(initial, rate, year, memo)

calls itself recursively for each year from `n` down to `0`. Without memoization, it would recompute values multiple times, leading to inefficiency.

With Memoization (using `HashMap`):

- Each year's value is computed only once.
- Average `O(1)` for `get()` and `put()`.

Time Complexity: `O(n)`
Space Complexity: `O(n)`

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

Optimization: Avoiding Excessive Computation

Using **memoization** avoids redundant calculations by caching already-computed results.

```
if (memo.containsKey(year)) {
   return memo.get(year);
}
```

This ensures each value is computed only once.
Speeds up execution for large `year` values.