**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.

**1. Understand Asymptotic Notation:**

**Big O Notation:**

Big O notation is a mathematical representation used to **describe the performance** (time or space complexity) of an algorithm in the **worst-case** as the input size (n) increases.

It helps developers:

* Compare algorithms.
* Predict scalability.
* Choose the most efficient algorithm for large inputs.

| **Notation** | **Meaning** | **Example** |
| --- | --- | --- |
| O(1) | Constant time | Accessing an array element |
| O(n) | Linear time | Linear search |
| O(log n) | Logarithmic time | Binary search |
| O(n²) | Quadratic time | Nested loops |

**Best, Average, and Worst-Case for Search Operations:**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) (first match) | O(n/2) ≈ O(n) | O(n) (last or not found) |
| Binary Search | O(1) (mid match) | O(log n) | O(log n) |

Binary search requires the array to be sorted.

**2. Setup**

* **Product Class**

**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** displayProduct() {

System.***out***.println("Product ID: " + productId +

", Name: " + productName +

", Category: " + category);

}

}

**3. Implementation**

* **SearchEngine Class (Linear & Binary Search)**

**package** com.example.sort;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** SearchEngine {

**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) / 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**;

}

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

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

}

}

* **Main Class to Demonstrate Functionality**

**public** **class** ECommerceSearchTest {

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

Product[] products = {

**new** Product(1, "Laptop", "Electronics"),

**new** Product(2, "Shoes", "Footwear"),

**new** Product(3, "Book", "Education"),

**new** Product(4, "Camera", "Electronics"),

**new** Product(5, "T-shirt", "Clothing")

};

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

Product found1 = SearchEngine.*linearSearch*(products, "Camera");

**if** (found1 != **null**) found1.displayProduct();

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

SearchEngine.*sortProductsByName*(products);

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

Product found2 = SearchEngine.*binarySearch*(products, "Laptop");

**if** (found2 != **null**) found2.displayProduct();

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

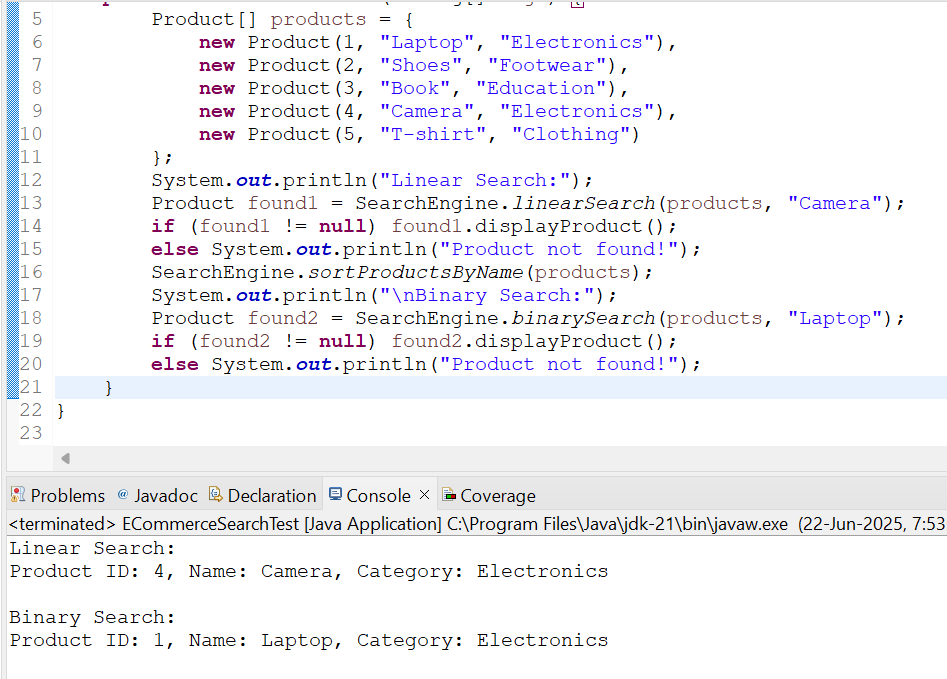
}

}

**4.Analysis**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Requires Sorted Data?** |
| --- | --- | --- | --- | --- |
| **Linear Search** | **O(1)** | **O(n)** | **O(n)** | **No** |
| **Binary Search** | **O(1)** | **O(log n)** | **O(log n)** | **Yes** |

**Output ScreenShot:**

****

**More Suitable:**

**Binary Search** is more suitable for large-scale e-commerce platforms:

1. Fast lookup: O(log n)
2. Efficient with sorted product listings.

**Linear Search** is simple and ideal when:

1. Data is small or unsorted.
2. One-time or rare searches.

------------------------------------------------------------------------------------------------------

**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:**
   1. 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

* **Setup:**

Create a method to calculate the future value using a recursive approach

**package** Forcast;

**import** java.util.HashMap;

**import** java.util.Scanner;

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

**public** **static** **double** forecastFutureValue(**double** initial, **double** rate, **int** year, HashMap<Integer, Double> memo) {

**if** (year == 0) **return** initial;

**if** (memo.containsKey(year)) {

**return** memo.get(year);

}

**double** previous = *forecastFutureValue*(initial, rate, year - 1, memo);

**double** current = previous \* (1 + rate);

memo.put(year, current);

**return** current;

}

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

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

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

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

System.***out***.print("Enter the annual growth rate (e.g., 0.05 for 5%): ");

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

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

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

HashMap<Integer, Double> memo = **new** HashMap<>();

**double** futureValue = *forecastFutureValue*(initialValue, growthRate, years, memo);

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

sc.close();

}

}

* **Implementation:**

Define a recursive method forecastFutureValue(initial, rate, year, memo) that returns the future value.

Use a HashMap to memoize previously computed results and avoid redundant calculations.

**package** Forcast;

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

**public** **class** ForecastTest {

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

**double** initialValue = 10000;

**double** growthRate = 0.08;

**int** years = 15;

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<>();

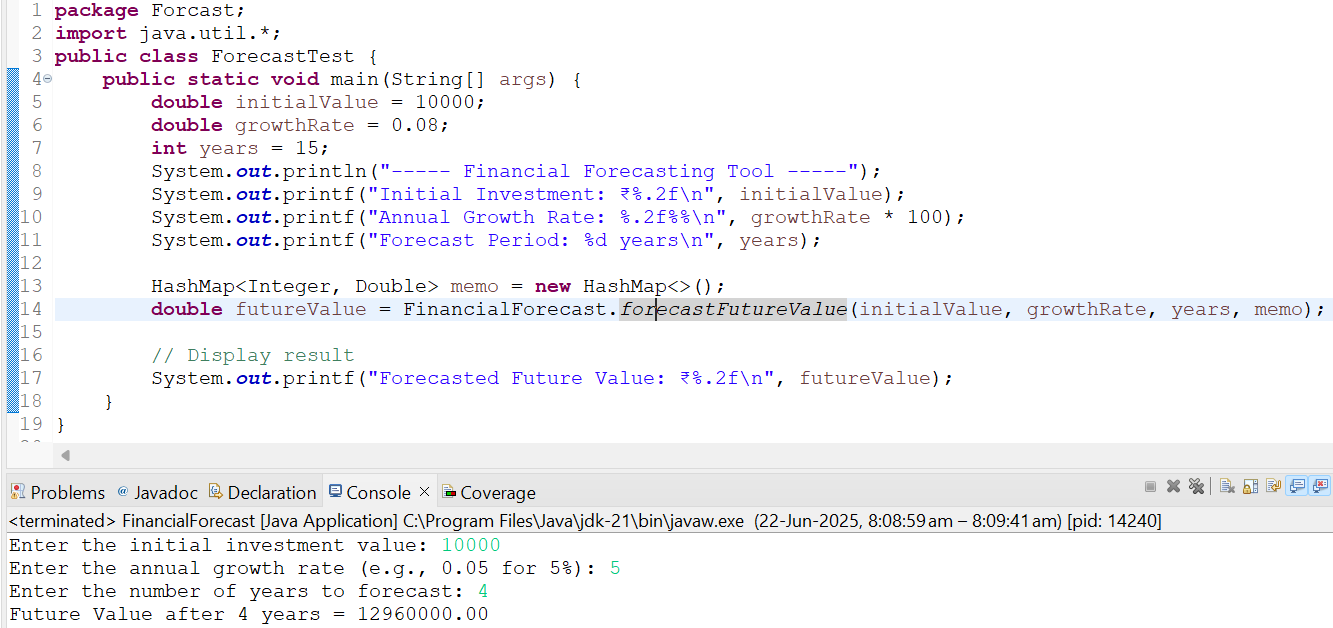
**double** futureValue = FinancialForecast.*forecastFutureValue*(initialValue, growthRate, years, memo);

// Display result

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

}

}

* **Output ScreenShot:**
* **Analysis:**

**1.** **Time Complexity of Recursive Algorithm:**  
 The method forecastFutureValue(initial, rate, year, memo) makes recursive calls from year down to 0.  
 Without memoization, it would perform repeated calculations, resulting in exponential time.

**2.** **With Memoization (using HashMap):**

Each year’s value is computed only once.

HashMap operations (get() and put()) have average-case time complexity of O(1).

**Time Complexity:** O(n)

**Space Complexity:** O(n) — for storing computed results in the memoization map.

**3.Optimization: Avoiding Excessive Computation:**

Memoization prevents redundant computations by caching previous results:

if (memo.containsKey(year)) {

return memo.get(year);

}

Ensures faster execution for large year values by avoiding repeated recursive calls.