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

**Understanding Asymptotic Notation**

In software development, the efficiency of an algorithm is crucial—especially in platforms like e-commerce, where thousands of items must be searchable in real-time. Asymptotic notation provides a way to describe how the execution time (or space) of an algorithm grows with respect to input size n.

* **Big O (O)** describes the upper limit of time required—commonly used to measure the worst-case performance.
* **Best case** is the quickest a function can execute.
* **Average case** is what usually happens with randomized data.
* **Worst case** is the slowest, like searching an absent product in a huge list.

**MarketplaceSearch.java**

public class MarketplaceSearch {

    public static void main(String[] args) {

        Item[] inventory = {

            new Item(1002, "Camera", "Electronics"),

            new Item(1005, "Sneakers", "Footwear"),

            new Item(1001, "Smartphone", "Gadget"),

            new Item(1004, "T-Shirt", "Clothing"),

            new Item(1003, "Book", "Stationery")

        };

        int idToSearch = 1003;

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

        Item result1 = LinearSearchUtil.search(inventory, idToSearch);

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

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

        BinarySearch.sort(inventory);

        Item result2 = BinarySearch.search(inventory, idToSearch);

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

    }

}

**LinearSearchUtil.java**

public class LinearSearchUtil {

    public static Item search(Item[] items, int targetId) {

        for (Item item : items) {

            if (item.getId() == targetId) {

                return item;

            }

        }

        return null;

    }

}

**BinarySearch.java**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

    public static void sort(Item[] items) {

        Arrays.sort(items, Comparator.comparingInt(Item::getId));

    }

    public static Item search(Item[] items, int targetId) {

        int left = 0;

        int right = items.length - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

            int currentId = items[mid].getId();

            if (currentId == targetId) {

                return items[mid];

            } else if (currentId < targetId) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

}

**Item.java**public class Item {

    private int id;

    private String name;

    private String type;

    public Item(int id, String name, String type) {

        this.id = id;

        this.name = name;

        this.type = type;

    }

    public int getId() {

        return id;

    }

    public String getName() {

        return name;

    }

    public String getType() {

        return type;

    }

    @Override

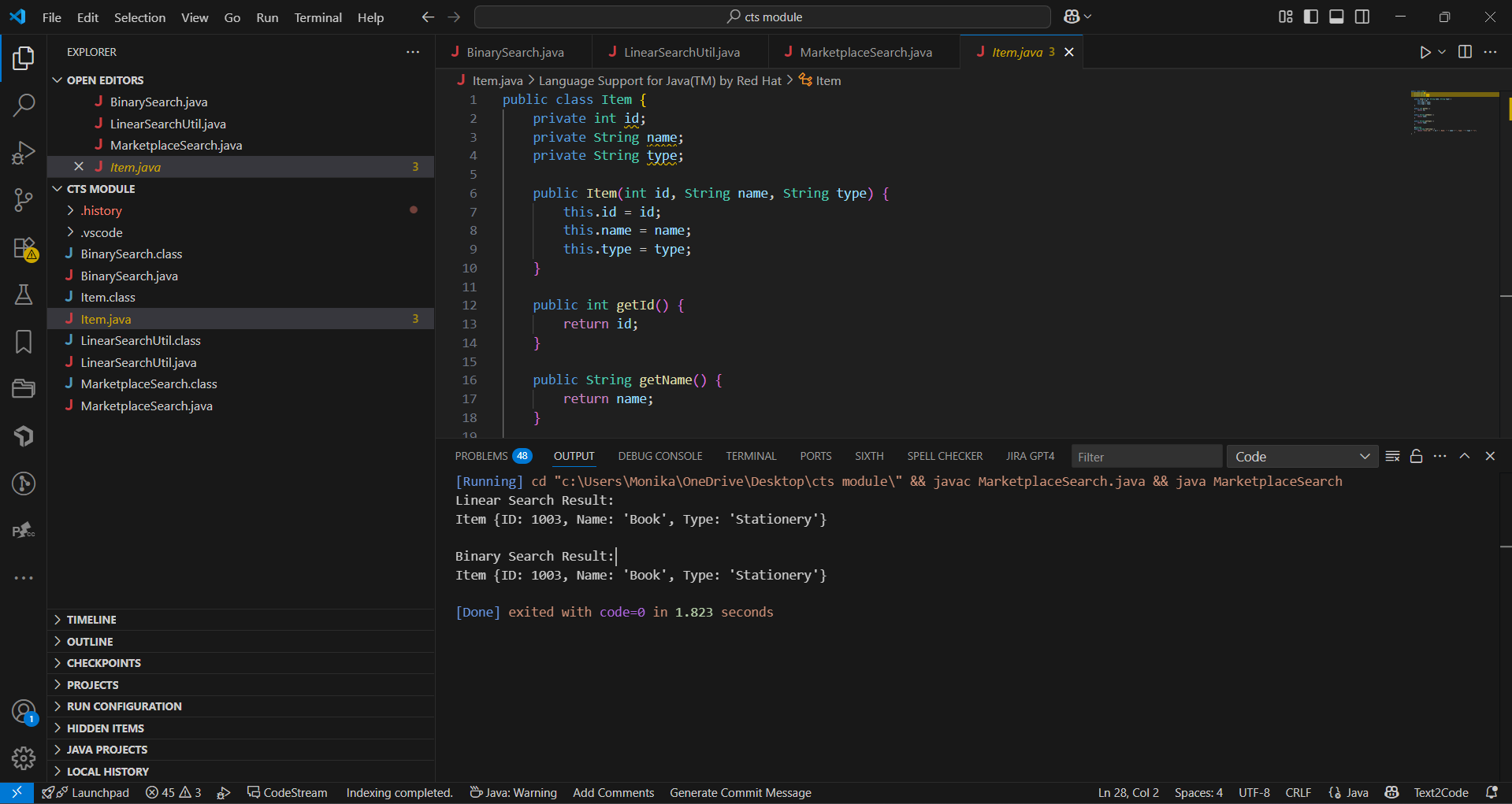
    public String toString() {

        return "Item {ID: " + id + ", Name: '" + name + "', Type: '" + type + "'}";

    }

}

**Output:**

****

**Analysis: Performance Evaluation**

**Linear search** scans items one by one. Its speed drops as the dataset grows. It's simple and doesn't require sorting, making it ideal for small or unsorted datasets.

**Binary search**, on the other hand, is lightning-fast because it reduces the search range in half each step. But it only works on sorted data. Sorting has an initial cost (O(n log n)), but for large and frequently searched lists, it's worth it.

For a high-volume e-commerce platform:

* Use binary search when data is sorted (or after sorting once).
* Use linear search only when the product catalog is small or being modified constantly.

To go further, you could implement hash-based search using HashMap for constant time complexity O(1), ideal for real-time product lookups.

**Exercise 7: Financial Forecasting**

**Understand Recursive Algorithms**

Recursion is a technique where a method calls itself repeatedly to break a problem into smaller sub-problems until it reaches a base condition. This is useful in scenarios like financial forecasting where future values depend on prior values. Recursion simplifies the logic by mimicking natural mathematical progression.

**FinancialForecast.java**

package forecasting;

import java.text.DecimalFormat;

public class FinancialForecast {

public static double predictValue(double currentValue, double growthRate, int years, double yearlyContribution) {

if (years == 0) {

return currentValue;

} else {

double nextValue = currentValue \* (1 + growthRate / 100) + yearlyContribution;

return predictValue(nextValue, growthRate, years - 1, yearlyContribution);

}

}

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 7.5;

int years = 5;

double yearlyContribution = 2000;

double predictedValue = predictValue(initialValue, growthRate, years, yearlyContribution);

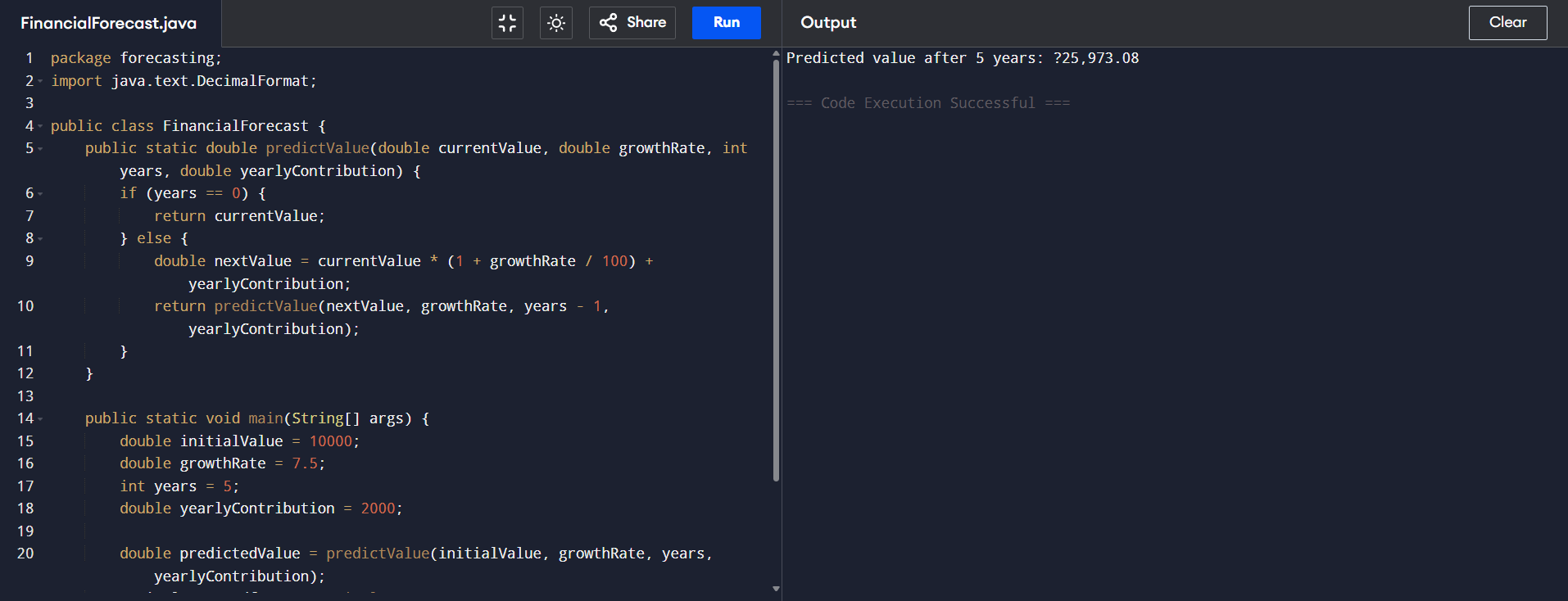
DecimalFormat df = new DecimalFormat("₹#,##0.00");

System.out.println("Predicted value after " + years + " years: " + df.format(predictedValue));

}

}

**Output:**



**Analysis:**  
Time Complexity:

* The time complexity is O(n) where *n* is the number of years.
* Each year involves one recursive call until the base condition is met, resulting in *n* total calls.

Optimization:

* Though recursion works well here, for large years values, Java doesn’t optimize tail recursion.
* You can convert the method to an iterative version to prevent stack overflow.
* Memoization is not needed here as results do not overlap.
* Formatted output and yearly contributions make the model more practical and readable.