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

1.Understanding Asymptotic Notation:

Big O Notation is a mathematical way to describe the efficiency of an algorithm, especially how its time or space requirements grow as the size of the input data increases. It gives an upper bound on the running time (or memory usage), helping you understand the worst-case scenario. Big O tells how slow (or fast) your algorithm becomes as the input grows.

How it helps in analyzing algorithms:

* Compare Algorithms: Helps compare multiple algorithms doing the same task.
* Predict Performance: Gives a rough idea of how your program behaves with large data.
* Optimize Code: Identifies bottlenecks to focus optimization efforts.

Case Analyze:

Linear Search:

* Best case O(1) - Target is the first element.
* Average case O(n) - Target is somewhere in the middle.
* Worst case O(n) - Target is last or not present.

Binary Search:

* Best case O(1) - Target is at the middle index.
* Average case O(log n) - Target is found after several halving steps.
* Worst case O(log n) - Target is not present.

Code:

import java.util.Arrays;

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 String toString() {

        return "[" + productId + ", " + productName + ", " + category + "]";

    }

}

public class SearchAlgorithm{

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

        for(Product p:products){

            if(p.productName.equalsIgnoreCase(productName)){

                return p;

            }

        }

        return null;

    }

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

        Arrays.sort(products, (a, b) -> a.productName.compareToIgnoreCase(b.productName));

        int start = 0;

        int end = products.length - 1;

        while(start<=end){

            int mid = start + (end - start)/2;

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

            if(compare == 0){

                return products[mid];

            }else if(compare > 0){

                start = mid + 1;

            }else{

                end = mid - 1;

            }

        }

        return null;

    }

}

Main Class:

import java.util.Arrays;

public class Search {

    public static void main(String[] args) {

        Product[] products = {

            new Product(1, "batman T-shirts", "Electronics"),

            new Product(2, "Shoe", "Fashion"),

            new Product(3, "Jogger", "Electronics"),

            new Product(4, "Speaker", "Clothing")};

        Product found1 = SearchAlgorithm.linearSearch(products, "shoe");

        System.out.println("Linear Search Result:" + found1);

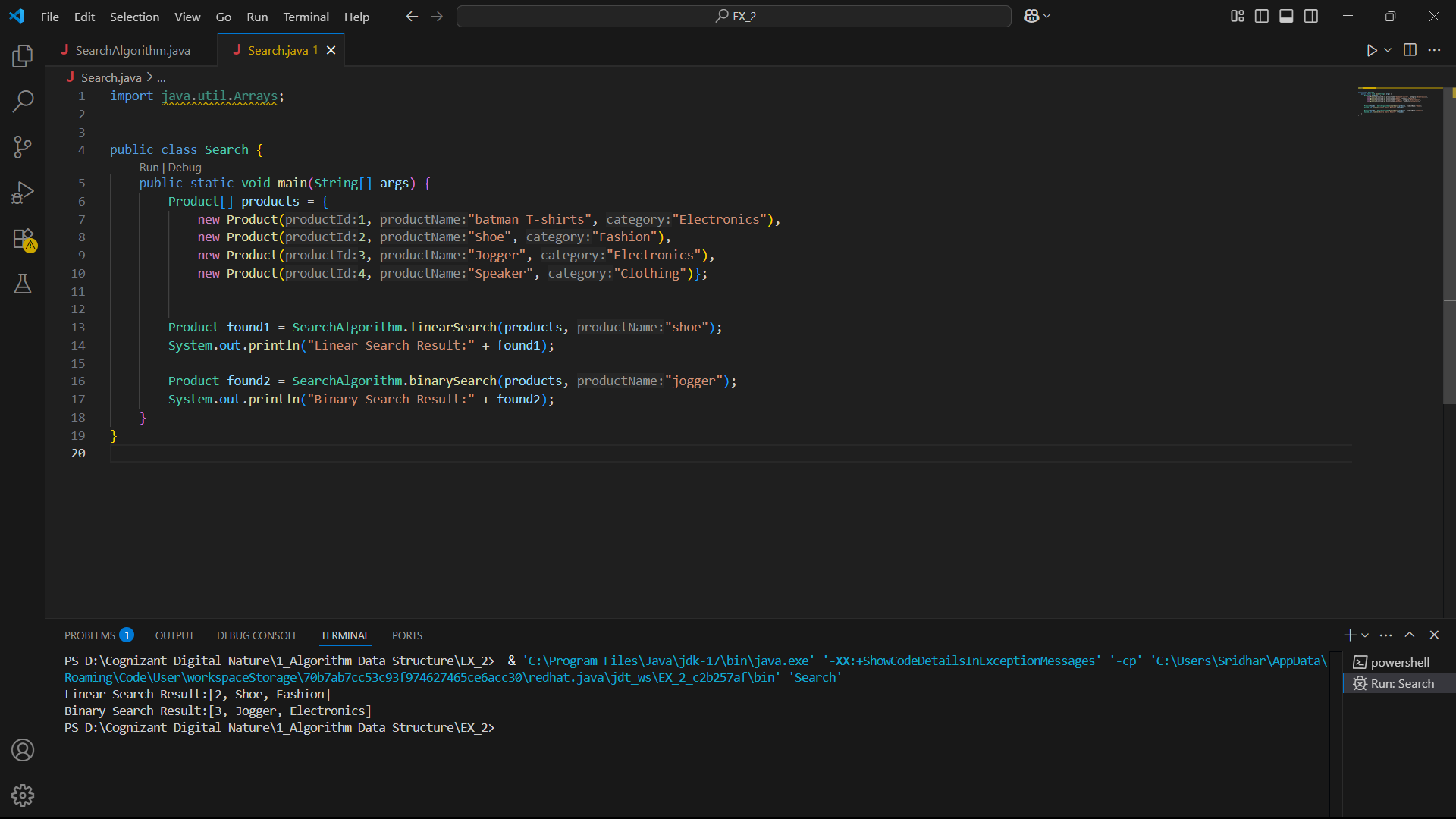
        Product found2 = SearchAlgorithm.binarySearch(products, "jogger");

        System.out.println("Binary Search Result:" + found2);

    }

}

Output:



Comparison of Time Complexity:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Best Case | Average Case | Worst Case | Complexity |
| Linear | O(1) | O(n) | O(n) | Linear |
| Binary | O(1) | O(log n) | O(log n) | Logarithmic |

Conclusion:

* Use binary search for fast and scalable search on sorted data in an e-commerce platform.
* Use linear search only for small or unsorted datasets where simplicity is more important than speed.

**Exercise 7: Financial Forecasting**

1.Recursive Algorithm:

Recursion is a programming technique where a method calls itself to solve smaller instances of the same problem. It simplifies problems that have repetitive or nested sub-problems. It is widely used in problems where the previous value is used to calculate the present value until a certain condition is achieved.

Implementation method:

public class FinancialForecast {

    public static double futureValue(double past\_value, double rate, int years) {

        if (years == 0) {

            return past\_value;

        }

        return futureValue(past\_value, rate, years - 1) \* (1 + rate);

    }

    public static void main(String[] args) {

        double presentValue = 1210;

        double annualRate = 0.05;

        int years = 7;

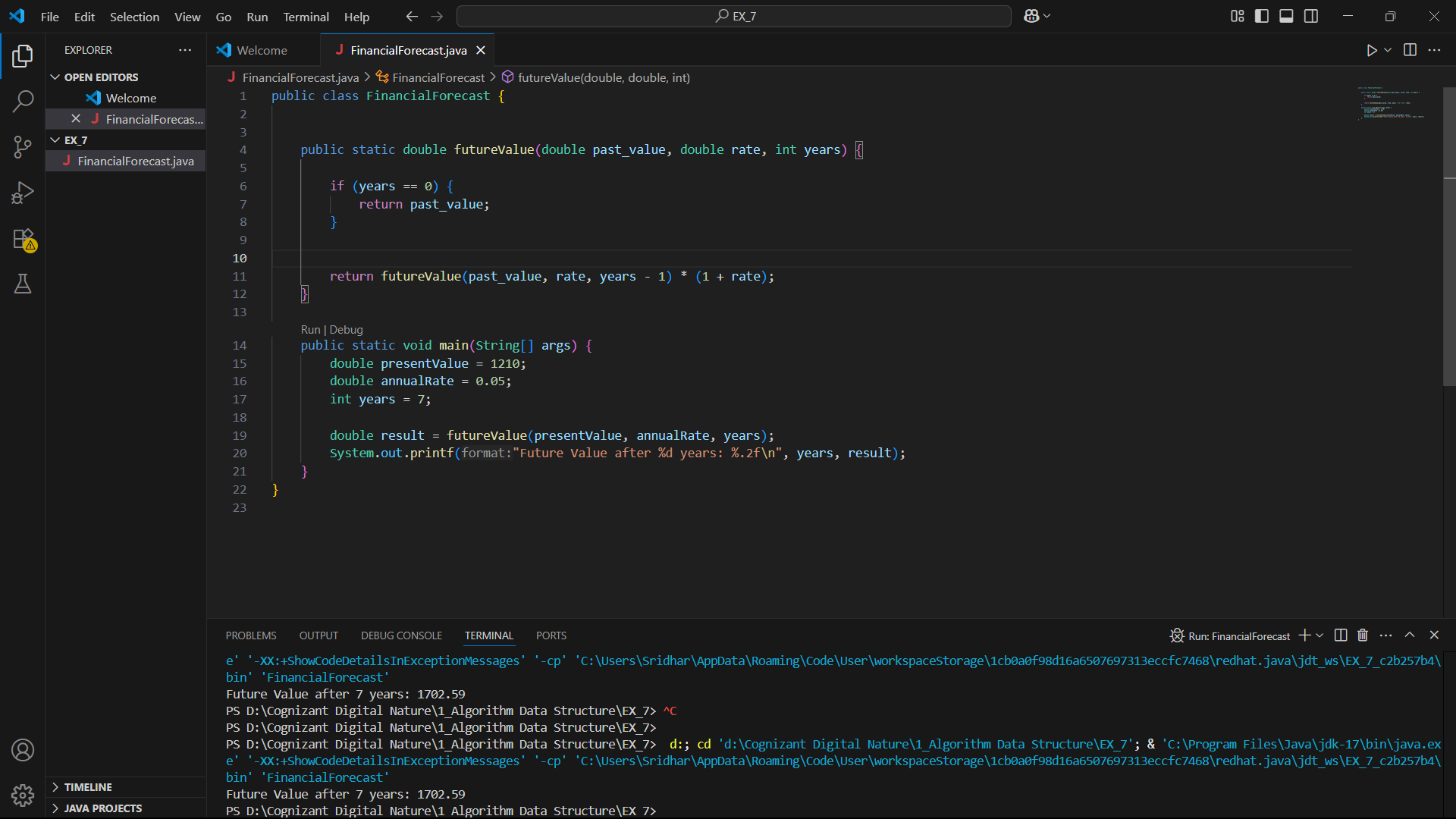
        double result = futureValue(presentValue, annualRate, years);

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

    }

}

Output:



Analysis: Time Complexity

* The time complexity is O(n) because the function makes one recursive call per year.
* There’s no overlapping subproblem.

Optimization:

We can use iteration using the for or while loop, instead of the Recursion, which will avoid the stack overflow and improve performance in large datasets. For large sets the iteration is more preferable as to optimize the code for large n inputs. Below is a iterative version for the same.

Iterative Version:

public static double futureValue(double past\_value, double rate, int years) {

double result = past\_value;

for (int i = 0; i < years; i++) {

result \*= (1 + rate);

}

return result;

}