Week 1 -Algorithms_DataStructures -Hands-On Exercises

Exercise 2: E-commerce Platform Search Function

1. Big O Notation:

***Big-O Notation** is a way to express the **upper bound** of an algorithm's time or space complexity.*

- *It provides a way to describe how the **runtime** or **space requirements** of an algorithm grow as the input size increases.*
- *Allows programmers to compare different algorithms and choose the most efficient one for a specific problem.*
- *Helps in understanding the scalability of algorithms and predicting how they will perform as the input size grows.*
- *Enables developers to optimize code and improve overall performance.*

Best,Average and Worst cases:

**Best case is the function which performs the minimum number of steps on input data of n elements.*

**Worst case is the function which performs the maximum number of steps on input data of size n.*

**Average case is the function which performs an average number of steps on input data of n elements.*

2.CODE:

Java Files: Product.java, SearchDemo.java

Product.java

```
public class Product {

    int productId;

    String productName;
```

```
    String category;

    public Product(int id, String name, String category) {
        this.productId = id;
        this.productName = name;
        this.category = category;
    }

    public String toString() {
        return "ID: " + productId + ", Name: " + productName + ", Category: " + category;
    }
}
```

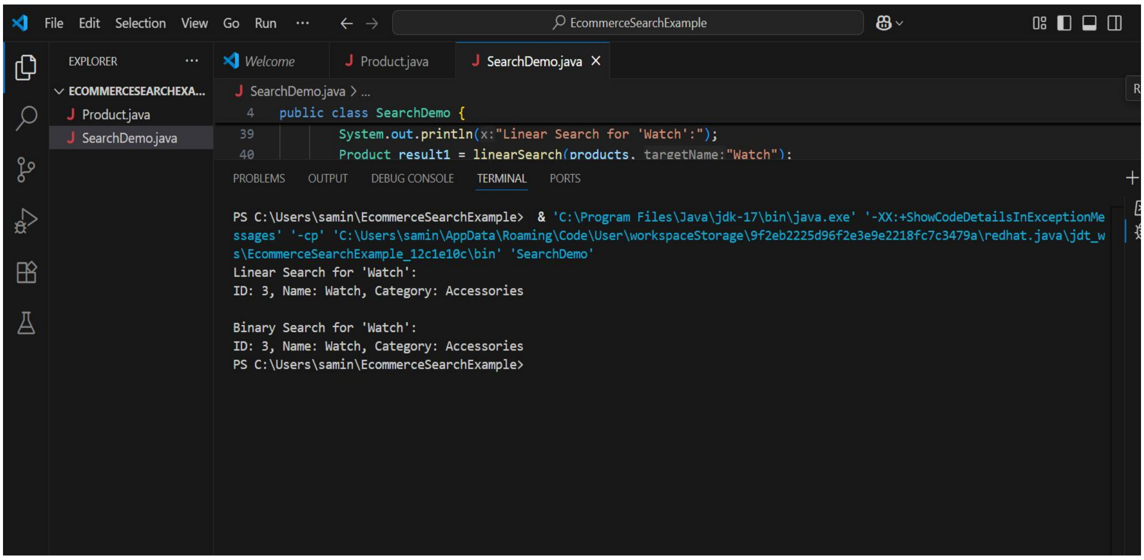
<u>SearchDemo.java</u>

```
import java.util.Arrays;
import java.util.Comparator;

public class SearchDemo {

    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;
        int 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 main(String[] args) {
        Product[] products = {
            new Product(1, "Laptop", "Electronics"),
            new Product(2, "Shoes", "Footwear"),
            new Product(3, "Watch", "Accessories"),
            new Product(4, "Phone", "Electronics"),
            new Product(5, "T-Shirt", "Apparel")
        };
        System.out.println("Linear Search for 'Watch':");
        Product result1 = linearSearch(products, "Watch");
        System.out.println(result1 != null ? result1 : "Product not found");
        Arrays.sort(products, Comparator.comparing(p ->
p.productName.toLowerCase()));
```

```
        System.out.println("\nBinary Search for 'Watch':");
        Product result2 = binarySearch(products, "Watch");
        System.out.println(result2 != null ? result2 : "Product not found");
    }
}
```

OUTPUT:



3.ANALYSIS:

Time complexity:

Algorithm	**Time complexity**
Binary Search	O(n)
Linear Search	O(log n)

=>***Binary search** is better for this platform because it's faster than linear search when the product list is sorted.*

Exercise 7: Financial Forecasting

Recursion:

→*Recursion is when a function calls itself to solve a smaller part of the problem.*

→*It simplifies code for problems like repeated calculations.*

CODE:

Java Files: Forecast.java, Main.java

Forecast.java

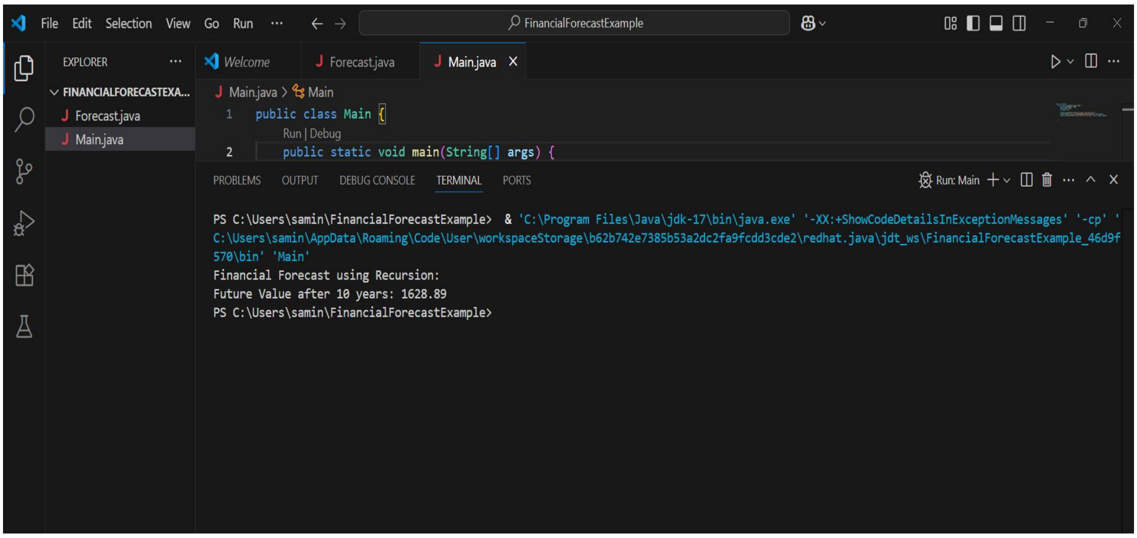
```
public class Forecast {
   public static double futureValueRecursive(double presentValue, double rate, int years) {
       if (years == 0) {
           return presentValue;
       }
       return futureValueRecursive(presentValue, rate, years - 1) * (1 + rate);
   }
   }
```

Main.java

```
   public class Main {
       public static void main(String[] args) {
           double presentValue = 1000;
           double rate = 0.05;
```

```
        int years = 10;


        System.out.println("Financial Forecast using Recursion:");
        double future = Forecast.futureValueRecursive(presentValue, rate, years);
        System.out.printf("Future Value after %d years: %.2f%n", years, future);


    }
}
```

OUTPUT:



3.ANALYSIS:

>> **<u>Time complexity:</u>**

Recursive version- O(n)