Cognizant Digital Nurture 4.0

Name: Siri Chandana Chittipolu

Email: sirichittipolu11@gmail.com

Superset ID: 6386277

Mandatory Hands-On Exercises

Data structures and Algorithms :

Exercise 2: E-commerce Platform Search Function:

Solution:

Step 1:

What is Big O Notation?

- Big O Notation describes the time or space complexity of an algorithm in terms of input size n.
- It tells us how the performance of an algorithm scales as the input size grows.

Big O for Search Algorithms:

Case	Linear Search	Binary Search
Best	O(1) (first element)	O(1) (middle element)
Average	O(n/2) → O(n)	O(log n)
Worst	O(n)	O(log n)

Step 2:

Product.java :

```
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 + ")";
    }
}
```

Step 3:

SearchUtil.java :

```
public class SearchUtil {
    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 low = 0;
    int high = products.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int cmp = products[mid].productName.compareToIgnoreCase(targetName);
        if (cmp == 0) {
            return products[mid];
        } else if (cmp < 0) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return null;
}
}
```

Step 4:

Compare the Time Complexity of Linear and Binary Search Algorithms

Feature	Linear Search	Binary Search
Time Complexity	O(n)	O(log n)
Best Case	O(1) (target at beginning)	O(1) (target at middle)
Average Case	O(n/2) → O(n)	O(log n)
Worst Case	O(n) (target not found or last)	O(log n) (search space halves each step)

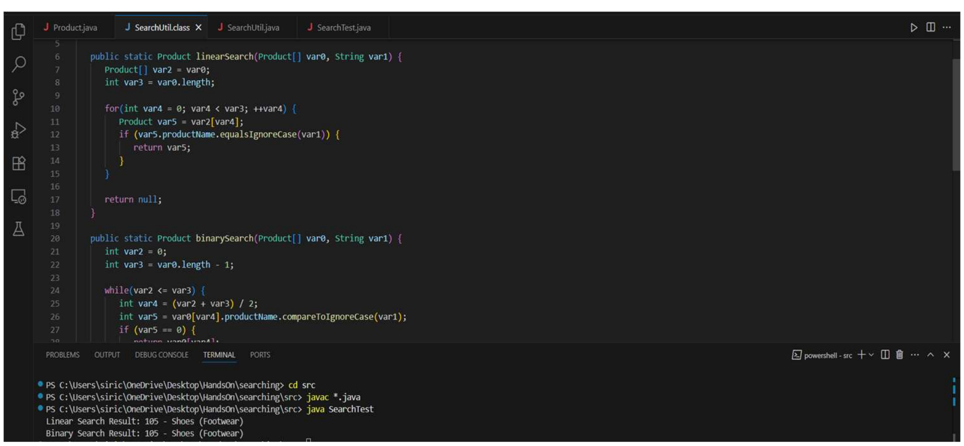
Binary Search is more suitable, because:

1. Performance:
Binary search is significantly faster (O(log n)) than linear search (O(n)) as the product catalog grows.

2. Scalability:
E-commerce platforms typically handle thousands to millions of products — log-based performance is essential for responsiveness.

3. Search Optimization:
Product data can be indexed or pre-sorted (e.g., by product name), which makes binary search practical and efficient.

Output :



Exercise 7: Financial Forecasting :

Solution:

Step 1:

Recursion:

- Recursion is a programming technique where a method calls itself directly or indirectly to solve a problem.
- Simplifies complex problems like tree traversal, mathematical series, and forecasting.
- Good fit when the current output depends on previous results, like forecasting based on prior growth.

Step 2:

Future Value Method:

Let's assume:

- futureValue(years) = currentValue * (1 + growthRate)^years
- We'll write this using recursion.

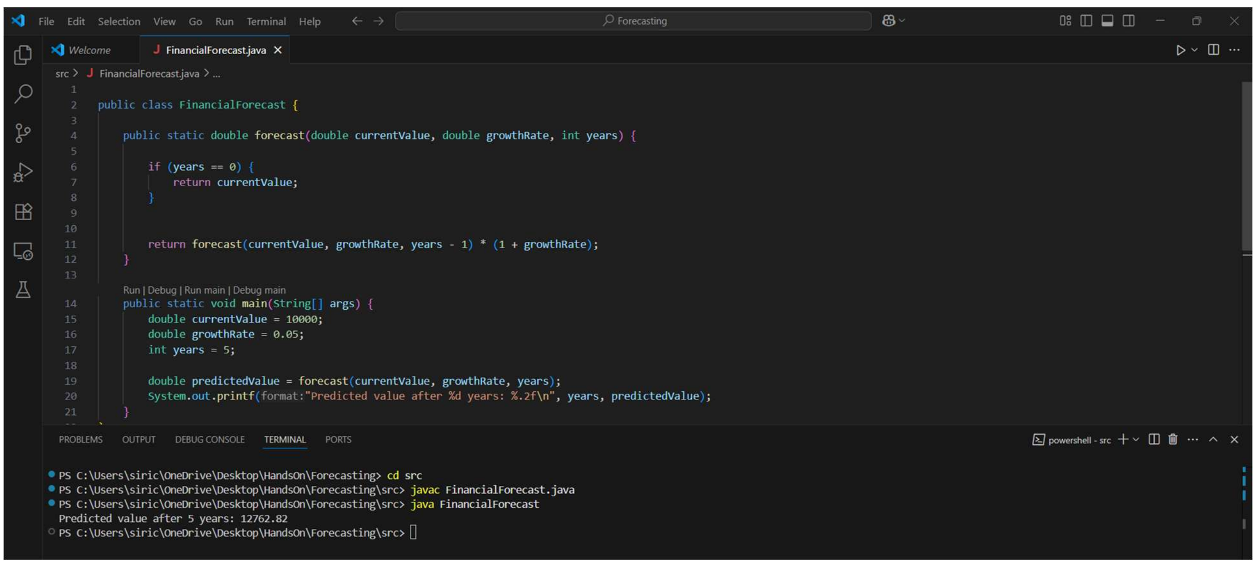
Step 3 :

FinancialForecast.java :

```
public class FinancialForecast {
    public static double forecast(double currentValue, double growthRate, int years)
{
        if (years == 0) {
            return currentValue;
        }
        return forecast(currentValue, growthRate, years - 1) * (1 + growthRate);
    }
    public static void main(String[] args) {
```

```
        double currentValue = 10000;
        double growthRate = 0.05;
        int years = 5;
        double predictedValue = forecast(currentValue, growthRate, years);
        System.out.printf("Predicted value after %d years: %.2f\n", years, 
predictedValue);
    }
}
```

Output :



Step 4 :

Time Complexity of the Recursive Algorithm

Recursive Formula Used:

futureValue(currentValue, growthRate, years) =

futureValue(currentValue, growthRate, years - 1) * (1 + growthRate)

Time Complexity:

- The recursion makes one call per year, reducing years by 1 each time.

- So, for n years, it makes n recursive calls.

Therefore:

- Time Complexity: O(n)
- Space Complexity: O(n) (due to the recursive call stack)

Problem in the Recursive Solution :

- Recursive calls add stack overhead.
- In large input cases, this may lead to stack overflow or performance issues.

Use Iteration Instead of Recursion :

Replace recursion with a loop for constant space.

```
public static double forecastIterative(double currentValue, double growthRate, int years) {
   for (int i = 0; i < years; i++) {
      currentValue *= (1 + growthRate);
   }
   return currentValue;
}
```

- Time Complexity: O(n)
- Space Complexity: O(1)