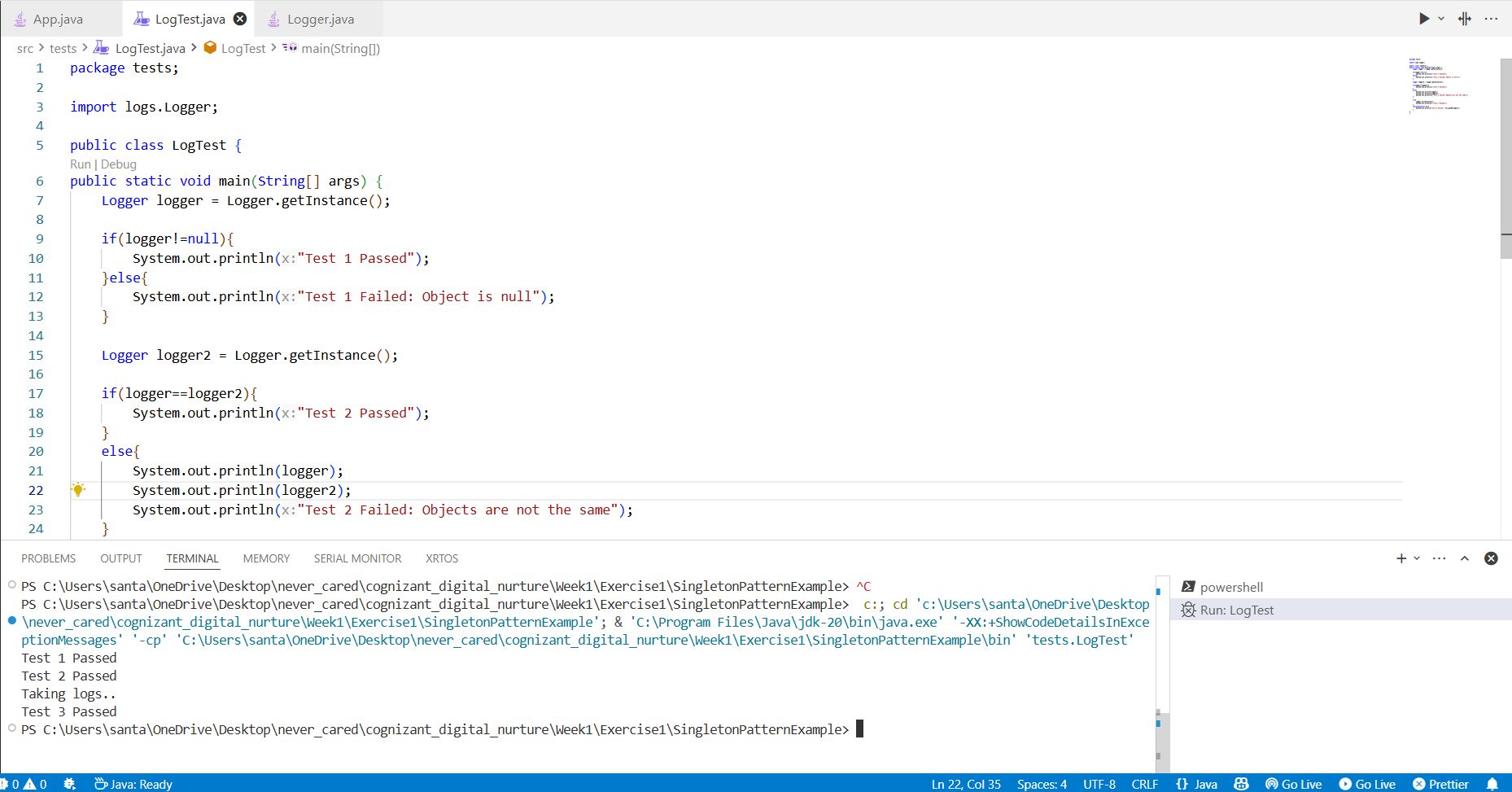
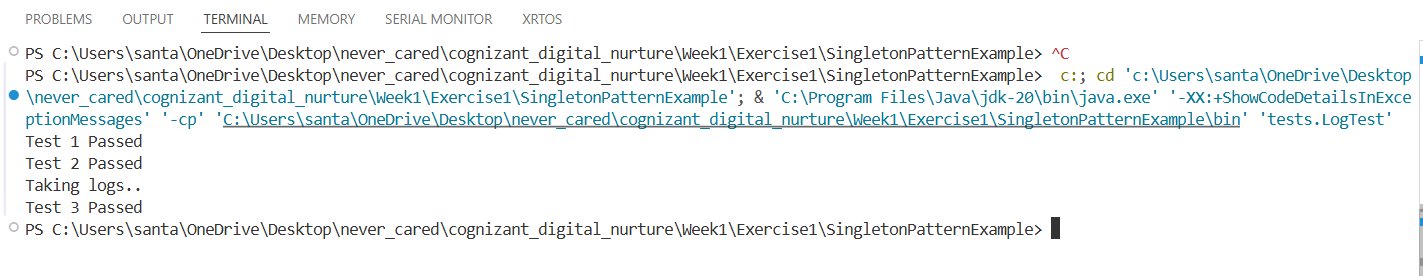
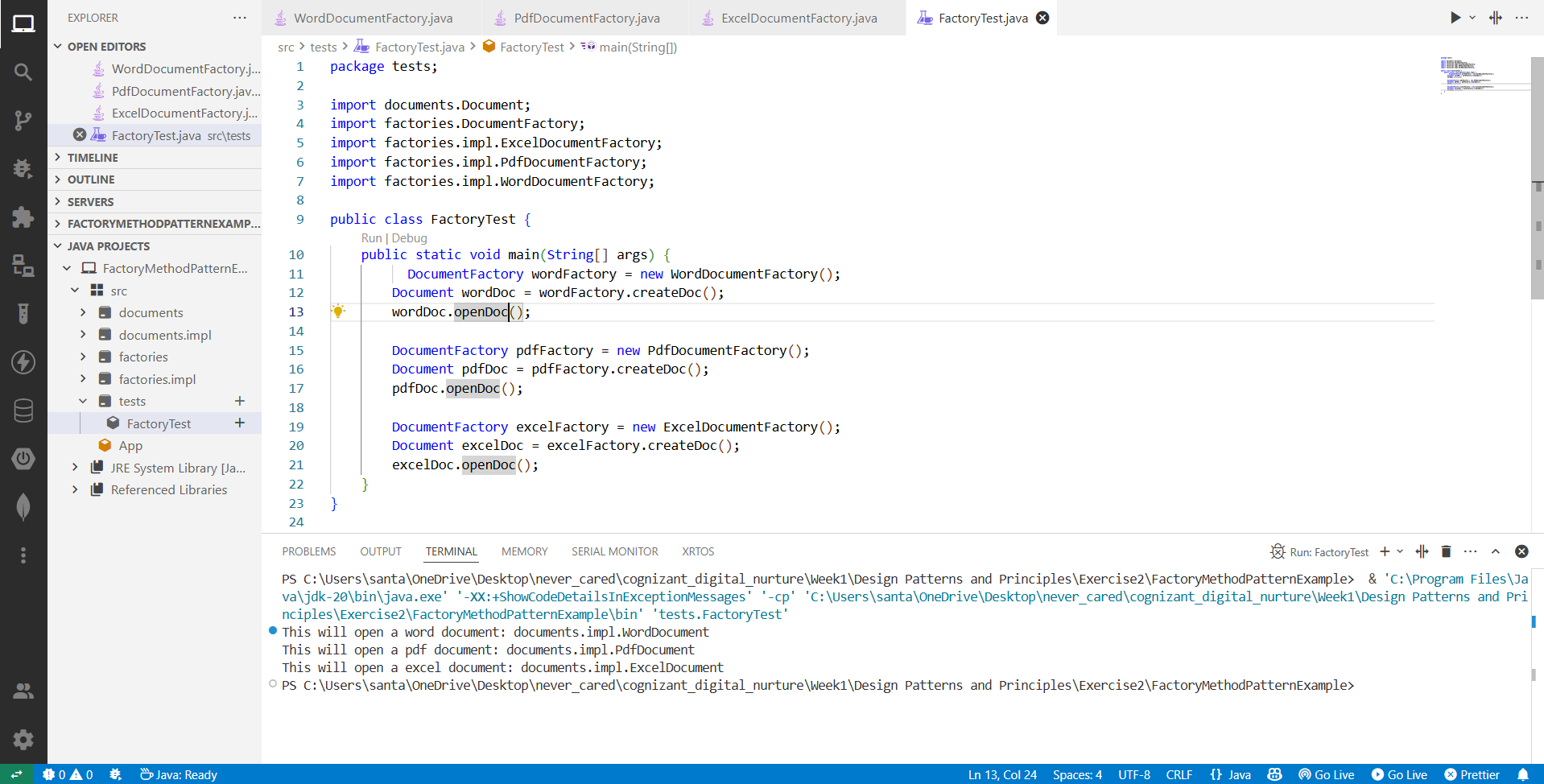
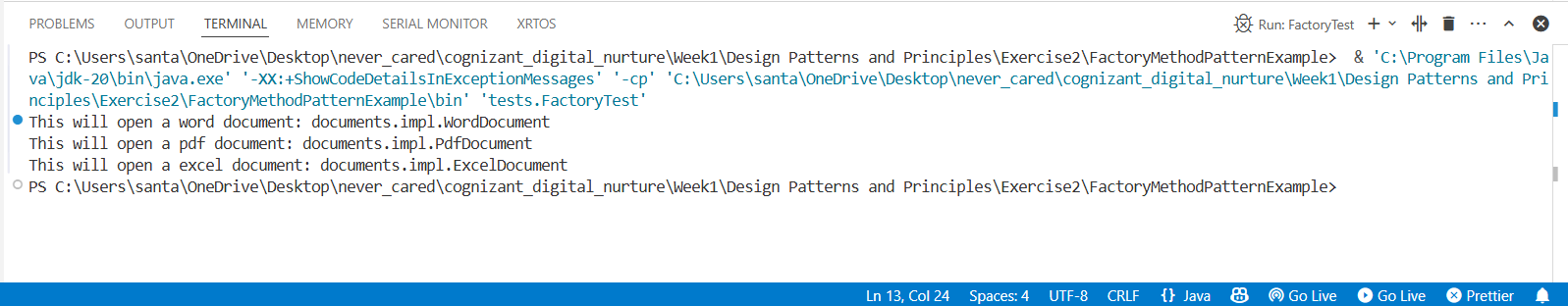
Mandatory Hands On

**Design Patterns and Principles  Exercise 1**

**output**

**Design Patterns and Principles Exercise 2**

**output**

****

**Algorithms\_Data Structures Exercise 2**

1. Explain Big O notation and how it helps in analyzing algorithms.

The Big O Notation is one of the three asymptotic notations used to analyze the time complexity of an algorithms. The Big (O) notation marks the worst case time complexity of an algorithm, meaning what are its performance stats in case the worst possible scenario arises.

For an example, if we take **the linear search** algorithm, if we are searching for our target within an array, there can be three scenarios the target element can be at the very beginning, anywhere in the middle or at the end.

The Big O notation marks the asymptotic upper bound that is the case in which the element is at the end and cannot exceed beyond. Now if O(1) is the time required to check for one particular index and there are total n elements in the array and we are checking every index until the end the time complexity becomes O(N).

1. Describe the best, average, and worst-case scenarios for search operations.

During a search operation such there can be two scenarios

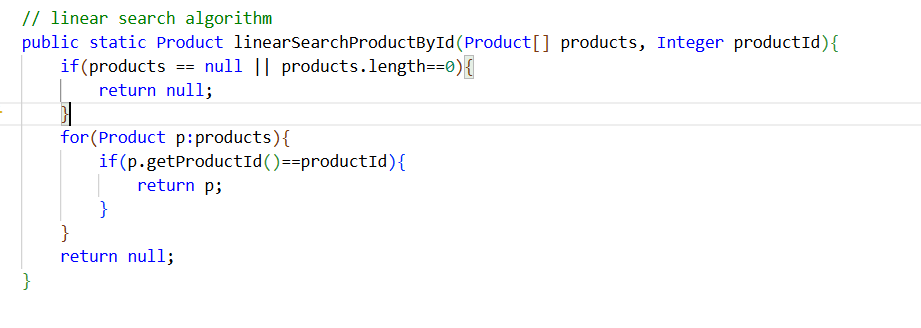
1. The array is unsorted – in this case we use the ***linear search***. From the starting index of the array we check each index, and until the end. Here we can get three scenarios
   1. Best Case:- The best case occurs if the target is at the starting index, it results in a time complexity of O(1).
   2. Worst Case:- The worst case occurs if the target is at the ending of the array, in which we search through the entire array and hence we get a time complexity of O(N)
   3. Average Case:- The average case occurs when the target element is anywhere in between the array. In which case the average case becomes O(N/2) and we can approximate it to O(N) depending on how close or far away it is from the middle.
2. The array is sorted – in case an array is sorted, we are free to use the **binary search**. We choose a middle and check if the target is greater or lesser than the target. In case the middle element is not the target depending on whether the target is larger or smaller than the middle we curve the search space.
   1. Best Case: On the first pass the element is at the middle index. It results in the time complexity of O(1).
   2. Worst Case: The element is either at the beginning or at the end, halving the search space we reduce until there is only one element is left. This operation’s time complexity is derived to

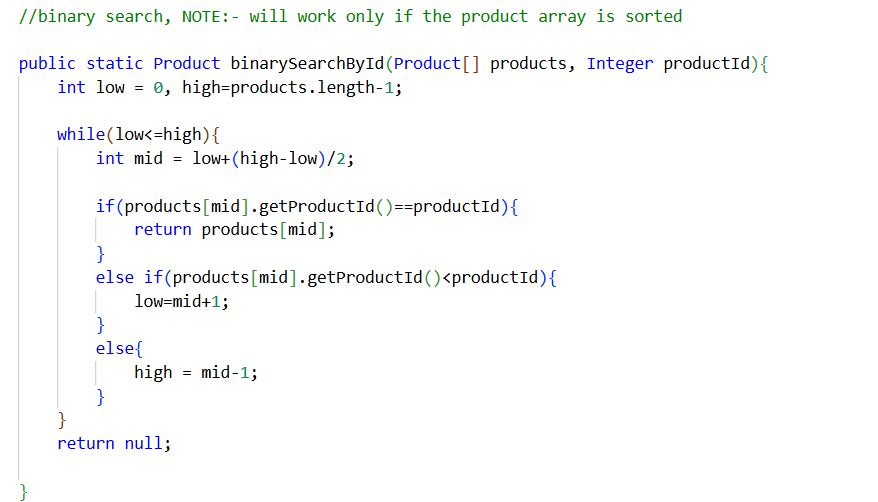
O (log n).

* 1. Average Case: In case the target is anywhere in amongst the array, we curve the search space which gives an average time complexity of O(log n)

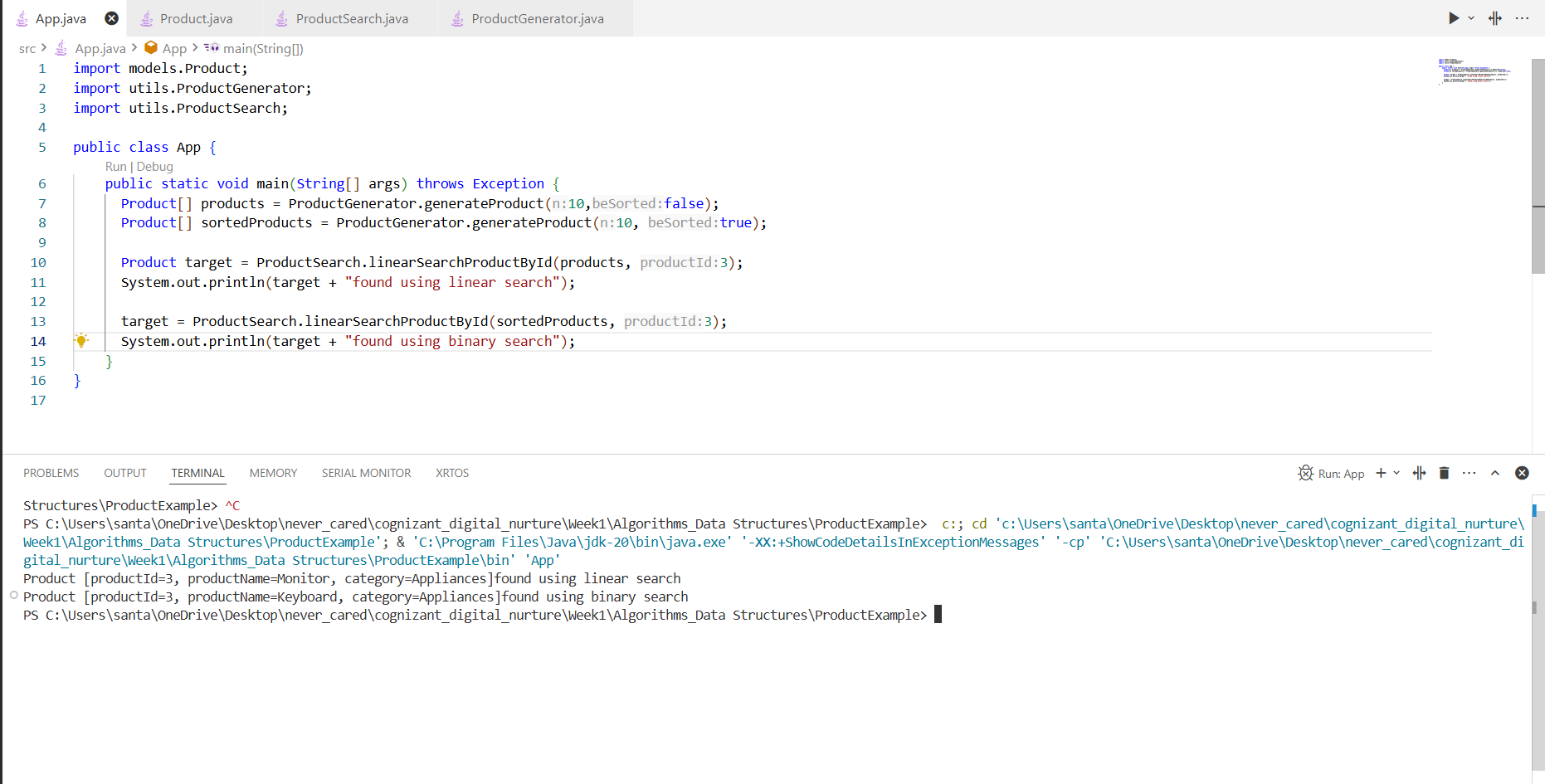
1. Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

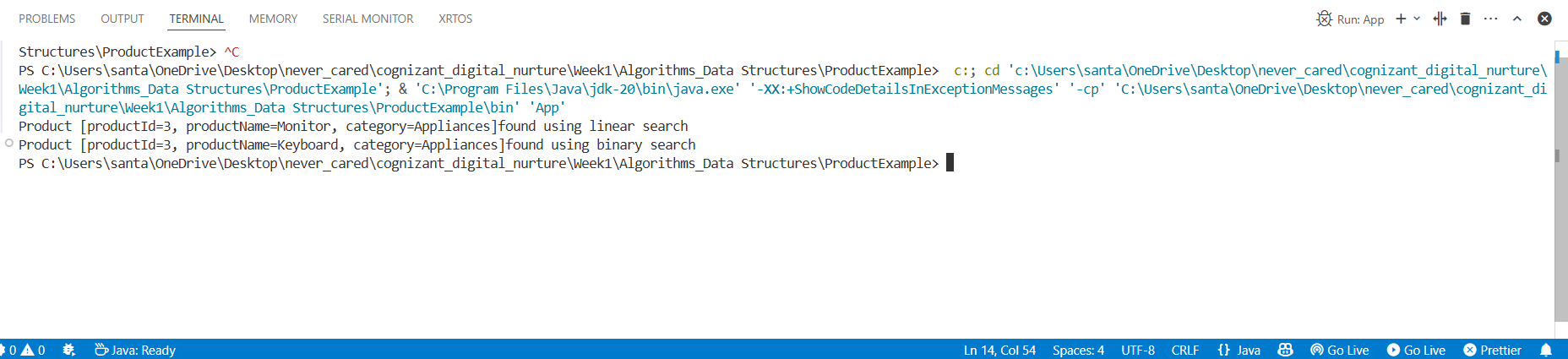
Linear Search Algorithm



 Binary Search Algorithm

**Output**





**Analysis**

**Comparative Analysis between linear and binary search algorithm**

|  |  |
| --- | --- |
| Linear Search | Binary Search |

|  |
| --- |
| Best Case |
| Avg Case |
| Worst Case |

|  |  |
| --- | --- |
| O (1) | O(1) |
| O(N) | O(log n) |
| O(N) | O(log n) |

**Discuss which algorithm is more suitable for your platform and why.**

In case we search using productid, which may be sorted we can use the binary search algorithm if the products are inserted in such a way that the ids are sorted.

If we are searching for a product by name, we need to use the linear search algorithm as the names of the products will not be in a sorted order.

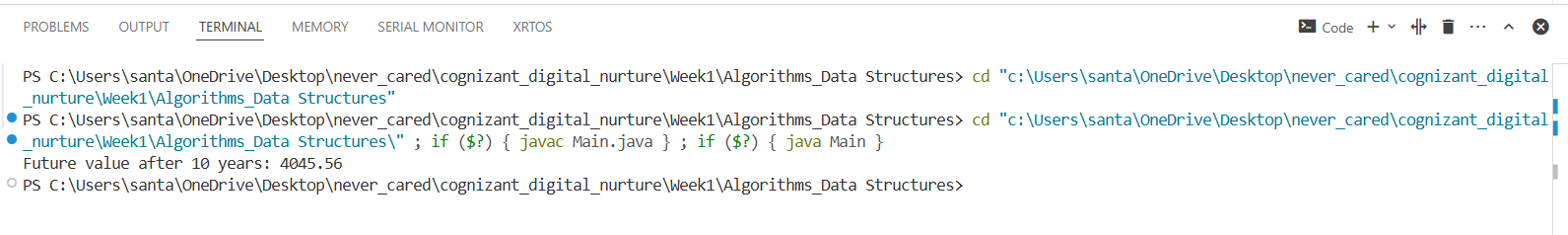
**Algorithms\_Data Structures: Exercise 7**

*Explain the concept of recursion and how it can simplify certain problems.*

Recursion is a technique where in a function calls itself. Originally a large problem that can be solved by solving many smaller problems of the same kind, by breaking them into smaller instances are done using recursion. Recursion is useful for these kinds of cases as a big task can be broken into many smaller tasks.

**Implement a recursive algorithm to predict future values based on past growth rates.**





*Discuss the time complexity of your recursive algorithm.*

Each Operation takes O(1),

T(n) = T(n-1)+O(1), for n recursive calls, hence time complexity

O(N)

*Explain how to optimize the recursive solution to avoid excessive computation.*

We can use memorization recursively and even cache the values if necessary