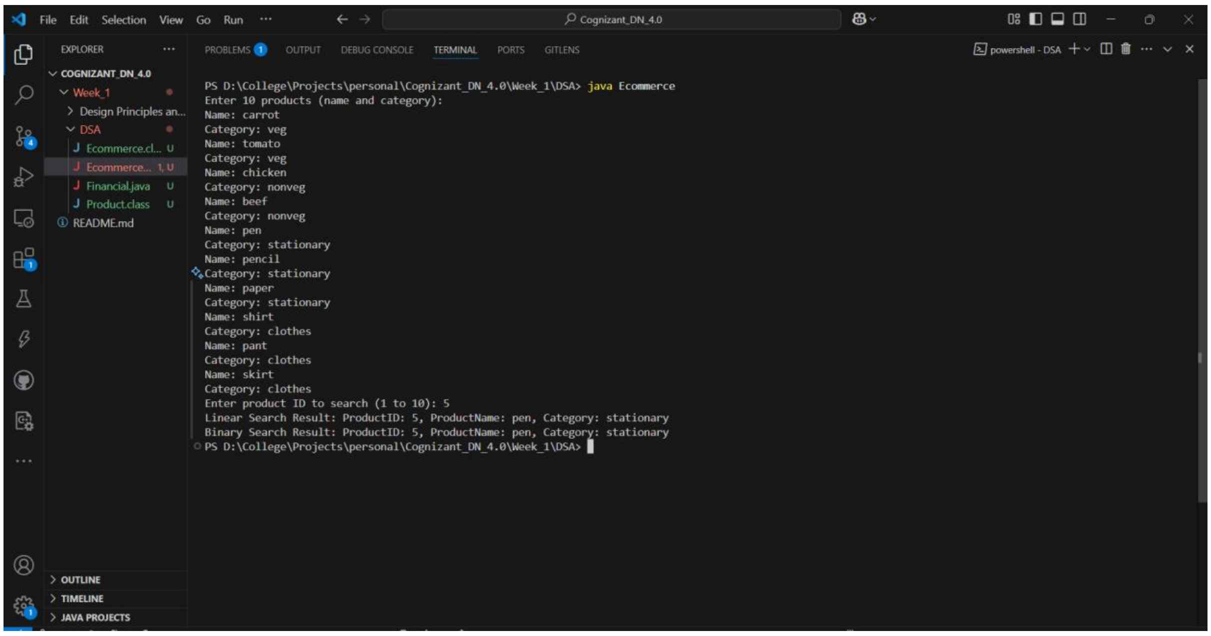
Data Structures and Algorithms

- ## E-Commerce:



```
PS D:\College\Projects\personal\Cognizant_DN_4.0\Week_1\DSA> java Ecommerce
Enter 10 products (name and category):
Name: carrot
Category: veg
Name: tomato
Category: veg
Name: chicken
Category: nonveg
Name: beef
Category: nonveg
Name: pen
Category: stationary
Name: pencil
Category: stationary
Name: paper
Category: stationary
Name: shirt
Category: clothes
Name: pant
Category: clothes
Name: skirt
Category: clothes
Enter product ID to search (1 to 10): 5
Linear Search Result: ProductID: 5, ProductName: pen, Category: stationary
Binary Search Result: ProductID: 5, ProductName: pen, Category: stationary
PS D:\College\Projects\personal\Cognizant_DN_4.0\Week_1\DSA>
```

Asymptotic Notation:

Big O notation is a way to describe the **efficiency of an algorithm** in terms of time or space usage as the input size grows. It helps us understand how the algorithm will perform when handling large amounts of data, by focusing on the **growth rate**, not actual time. “n” is considered as the input and is analysed as follows,

- O(1) - constant time (eg: accessing an array element by index)
- O(log n) - logarithmic (eg: binary search)
- O(n) - linear (eg: scanning an array)
- O(n^2) - quadratic time (slower, often seen in nested loops)

It is important to understand this concept when we pick an algorithm to solve a particular task. For instance, in the e-commerce search, when the number of records increases, predictable performance of the catalog may scale to tens of thousands (or more) of products. In such a case it is important to ensure that the most optimal algorithm is used.

Best/Average/Worst cases for search:

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

- **Linear**-best case: target is at index 0.
- **Binary**-best case: middle element matches immediately.

Analysis & Suitability

1. **Time Complexity**
    - Linear Search:
        - Best: O(1) (first element)
        - Average/Worst: O(n)
    - Binary Search:
    - Best: O(1)
    - Average/Worst: O(log n)
2. **Space Complexity**
    - Both are O(1) extra space (in-place).

Algorithm	**Time Complexity**	**Requirement**
Linear Search	O(n)	Works on unsorted data
Binary Search	O(log n)	Needs sorted data

3. Practical Considerations

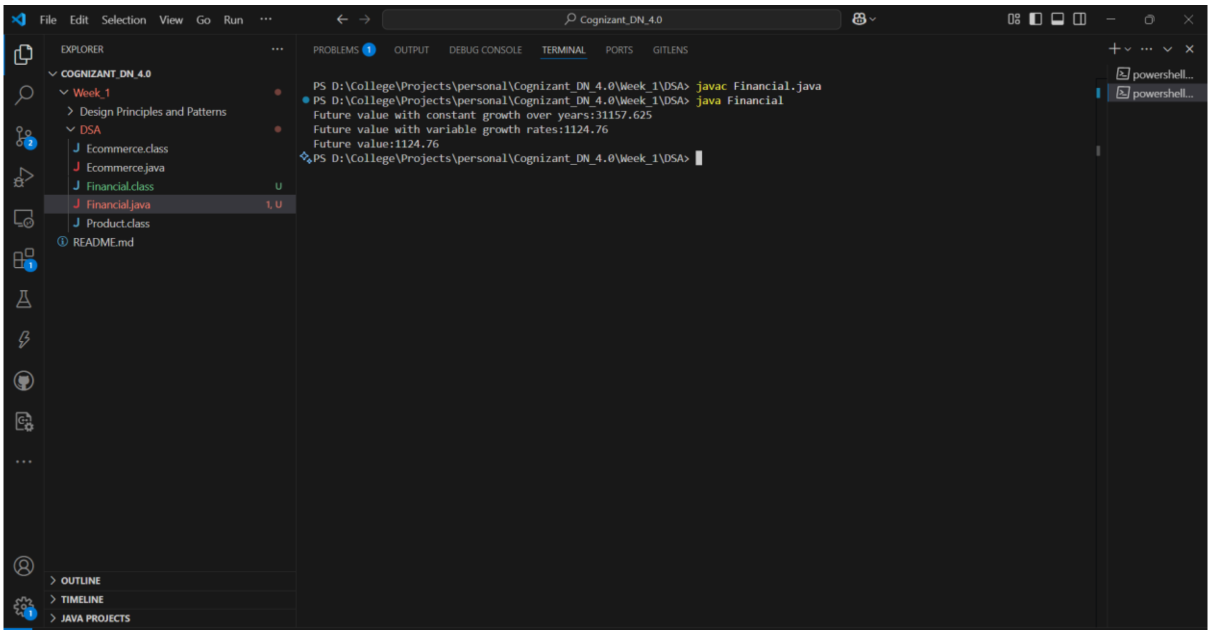
In an e-commerce platform, the choice between linear and binary search depends on how product data is stored and accessed:

- If the platform maintains products in a **sorted structure** (eg: sorted by product ID, price, or name), then **binary search is more suitable**. Its logarithmic time complexity (O(log n)) ensures faster performance as the number of products increases.
- If the product list is **unsorted** or if data changes frequently (eg: due to real-time inventory updates) and sorting is too costly, then **linear search** is the appropriate choice. Though slower (O(n)), it does not require the data to be sorted.

4. E-commerce Fit

- In most real-world e-commerce systems, product data is often organized using sorted structures or indexing mechanisms to allow efficient search operations. Therefore, binary search or even more advanced search techniques (like hash maps) are typically used to ensure optimal performance.
- It is useful when the front-end or service layer frequently looks up products by their unique IDs, maintain a sorted array.
- For category filters or name searches, considering more advanced structures like tries for prefix search, inverted indexes (like ElasticSearch) for full-text search, or a B-Tree index at the database layer.

II. Financial:



```
PS D:\College\Projects\personal\Cognizant_DN_4.0\Week_1\DSA> javac Financial.java
PS D:\College\Projects\personal\Cognizant_DN_4.0\Week_1\DSA> java Financial
Future value with constant growth over years:31157.625
Future value with variable growth rates:1124.76
Future value:1124.76
PS D:\College\Projects\personal\Cognizant_DN_4.0\Week_1\DSA>
```

Recursion:

1. Concept of Recursion and Its Application

Recursion is a programming technique where a method calls itself to solve smaller instances of the same problem. It is particularly effective for problems that can be broken down into repetitive sub-problems, such as mathematical sequences or compound operations over time.

In the provided code, recursion is used to calculate the future value of an investment:

- The method futureValue() handles the case of a **constant annual growth rate**.
- The method fVRates() handles **variable annual growth rates** using an array.

Both methods recursively reduce the problem year by year.

For example,

```
futureValue(){
        return (1 + rate) * futureValue(initialValue, rate, years - 1);
        }
```

This line expresses that the future value at a certain year depends on the result from the previous year, thereby reducing the need for manually managing intermediate calculations.

Recursion helps keep the code concise and readable, especially when the problem naturally involves repetitive logic based on previous states.

2. Time Complexity of the Recursive Algorithms

- **futureValue()** (constant rate):
  This method makes one recursive call per year, decrementing the years counter until it reaches 0.
  Time complexity: O(n), where n is the number of years.

- **fVRates()** (variable rates):
  This method processes each year by recursively moving through the rates array.
  Time complexity: O(n), where n is the length of the rates array.
  Each call performs a constant amount of work (a multiplication and an addition), so the total time scales linearly with the number of years.

3. Optimization of Recursive Solutions

While recursion simplifies problem-solving, it also introduces overhead due to repeated function calls and stack usage. In Java, deep recursion can lead to **stack overflow errors** if the number of recursive calls becomes too large.

To optimize recursive solutions:

- **Convert to Iterative Approach:**
  The method fVIterative() demonstrates this by replacing recursion with a loop:
  ```
  for (double rate : rates) {
     value *= (1 + rate);
  }
  ```
  This approach avoids recursion altogether, improving performance and eliminating the risk of stack overflow.

- **Avoid Redundant Calculations:**
  Although not needed in this example, memoization is a common technique used to cache and reuse previously computed values in recursive algorithms.
  Overall, the iterative method is more efficient and scalable, especially when dealing with large datasets or longer time horizons.