**Module 2 : Data Structures and Algorithms**

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

**Scenario: You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.**

**Solution :**

Step 1:

**Big O Notation:**

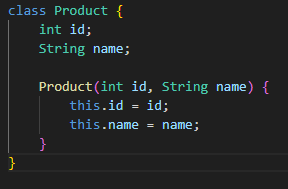
* **Big O** tells how fast or slow an algorithm runs when the input size increases.
* It helps compare which algorithm is more efficient.
* **Example:**
  + If a search takes O(n), it means time increases linearly with input size.
  + If it takes O(log n), it means it's much faster on large inputs.

**Best, Average, and Worst Cases in Search:**

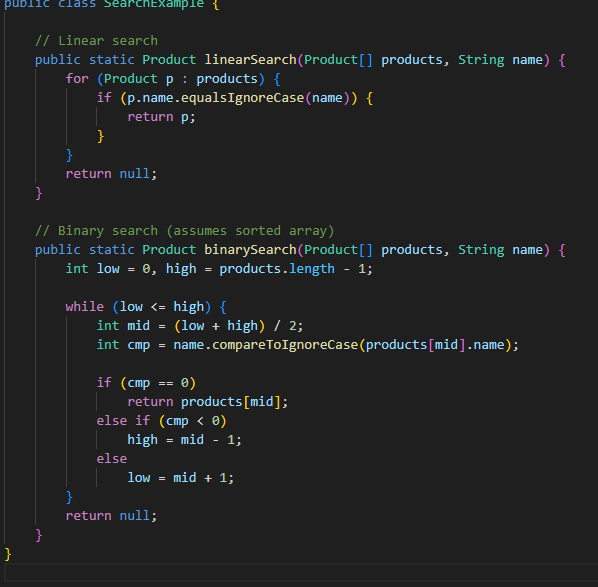
* **Best case:**
  + The item is found at the beginning.
  + E.g., Linear search finds the item in the first position → O(1)
* **Average case:**
  + The item is somewhere in the middle → O(n/2) ≈ O(n)
* **Worst case:**
  + The item is at the end or not present → O(n)

Step2 :

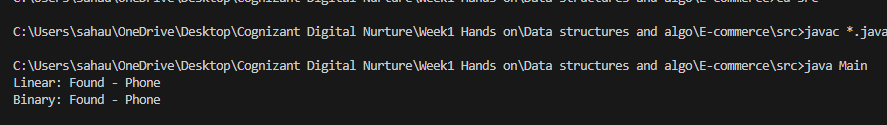
**Code :**

****

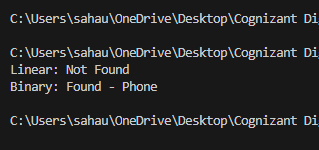
Step3 :

****

****

**Output : **

**On searching for paper shows not found**

****

Step 4:

| **Search Type** | **Time Complexity** | **Sorting Needed?** | **Suitable For Large Data?** |
| --- | --- | --- | --- |
| Linear Search | O(n) | No | Slow for large data |
| Binary Search | O(log n) | Yes | Fast for large sorted data |

**Binary Search** is better if your product list is **sorted**.

**Linear Search** is simple and works even if the list is **not sorted**, but it's **slower** for large inputs.

**Exercise 7: Financial Forecasting**

**Scenario: You are developing a financial forecasting tool that predicts future values based on past data.**

**Solution :**

**Step 1 :**

* Recursion is when a function **calls itself** to solve smaller parts of the problem.
* It's useful when the problem can be broken down into **repetitive subproblems**.

**Example:**

int factorial(int n) {

if (n == 0) return 1;

else return n \* factorial(n - 1);

}

Step2:

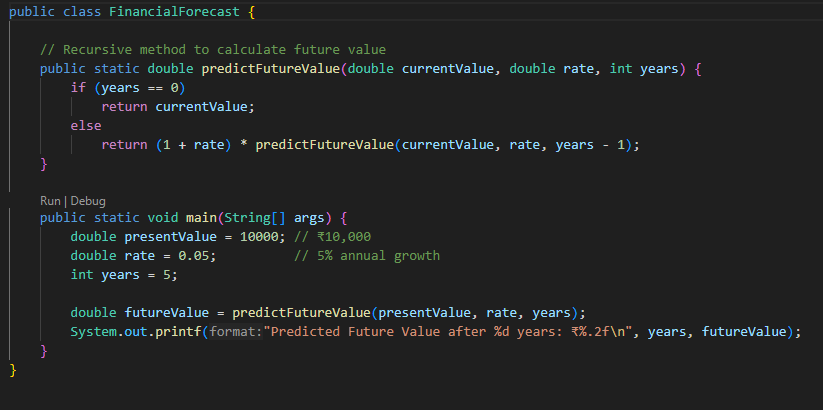
Let’s create a method that calculates **future value** based on:

* current value
* growth rate per year
* number of years

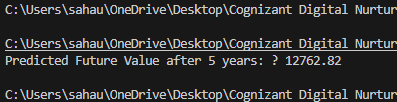
Formula: futureValue = currentValue × (1 + rate) ^ years

Step3:

**Code:**

****

**Output:**

****

Step4:

#### Time Complexity:

* Each recursive call reduces years by 1.
* So, **Time Complexity = O(n)**, where n is the number of years.

#### Problem with Recursion:

* Too many recursive calls can lead to **stack overflow**.
* Recursive calls may repeat same calculations.

#### How to Optimize?

1. **Memoization** – Store results of already calculated years in a map or array.
2. **Use Iteration** – Recursion is simple but loop-based version is safer for large n.