WEEK 1- Algorithms\_Data Structures

# Exercise 2: E-commerce Platform Search Function

## 1. Understand Asymptotic Notation

- Big O Notation:  
 Big O describes the upper bound of the time or space an algorithm uses as the input size grows. It's crucial for comparing the efficiency of different algorithms.  
  
 - O(1): Constant time  
 - O(log n): Logarithmic time  
 - O(n): Linear time  
 - O(n log n): Log-linear time  
 - O(n²): Quadratic time  
  
- Best, Average, and Worst Cases:  
 - Best Case: The ideal scenario where the desired item is found immediately.  
 - Average Case: The expected scenario based on random input.  
 - Worst Case: The least efficient scenario (e.g., item not present or at last index).

## 2. Setup: Product Class

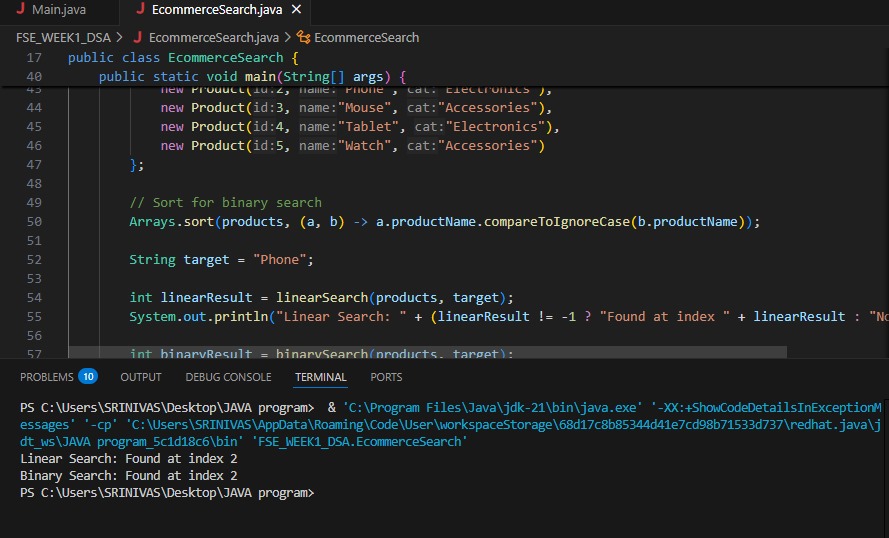
We need a Product class with basic attributes: productId, productName, and category.

## 3. Implementation

Save this code as: EcommerceSearch.java

import java.util.Arrays;  
  
class Product {  
 int productId;  
 String productName;  
 String category;  
  
 public Product(int id, String name, String cat) {  
 this.productId = id;  
 this.productName = name;  
 this.category = cat;  
 }  
}  
  
public class EcommerceSearch {  
   
 public static int linearSearch(Product[] products, String targetName) {  
 for (int i = 0; i < products.length; i++) {  
 if (products[i].productName.equalsIgnoreCase(targetName)) {  
 return i;  
 }  
 }  
 return -1;  
 }  
  
 public static int binarySearch(Product[] products, String targetName) {  
 int low = 0, high = products.length - 1;  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 int compare = products[mid].productName.compareToIgnoreCase(targetName);  
 if (compare == 0) return mid;  
 else if (compare < 0) low = mid + 1;  
 else high = mid - 1;  
 }  
 return -1;  
 }  
  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(1, "Laptop", "Electronics"),  
 new Product(2, "Phone", "Electronics"),  
 new Product(3, "Mouse", "Accessories"),  
 new Product(4, "Tablet", "Electronics"),  
 new Product(5, "Watch", "Accessories")  
 };  
  
 Arrays.sort(products, (a, b) -> a.productName.compareToIgnoreCase(b.productName));  
  
 String target = "Phone";  
  
 int linearResult = linearSearch(products, target);  
 System.out.println("Linear Search: " + (linearResult != -1 ? "Found at index " + linearResult : "Not Found"));  
  
 int binaryResult = binarySearch(products, target);  
 System.out.println("Binary Search: " + (binaryResult != -1 ? "Found at index " + binaryResult : "Not Found"));  
 }  
}

## 4. Analysis

| 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 Search is simple but inefficient for large data.  
- Binary Search is faster but requires the array to be sorted.  
- For large-scale platforms, Binary Search or even better: HashMaps or search indices (e.g., ElasticSearch) are recommended.  


# Exercise 7: Financial Forecasting

## 1. Understand Recursive Algorithms

- Recursion is when a function calls itself to solve a smaller version of the same problem.  
- It simplifies code for problems with a natural recursive structure (e.g., tree traversal, Fibonacci).  
- However, recursion can be less efficient due to repeated calls and stack usage.

## 2. Setup: Recursive Future Value

We'll compute future value using the formula:  
FV = PV × (1 + r)^n  
  
Where:  
- PV = Present Value  
- r = Growth rate  
- n = Number of periods

## 3. Implementation

Save this code as: FinancialForecast.java

public class FinancialForecast {  
  
 public static double futureValueRecursive(double presentValue, double rate, int periods) {  
 if (periods == 0) return presentValue;  
 return (1 + rate) \* futureValueRecursive(presentValue, rate, periods - 1);  
 }  
  
 public static double futureValueIterative(double presentValue, double rate, int periods) {  
 double result = presentValue;  
 for (int i = 0; i < periods; i++) {  
 result \*= (1 + rate);  
 }  
 return result;  
 }  
  
 public static void main(String[] args) {  
 double presentValue = 1000.0;  
 double growthRate = 0.08;  
 int years = 5;  
  
 double futureRecursive = futureValueRecursive(presentValue, growthRate, years);  
 double futureIterative = futureValueIterative(presentValue, growthRate, years);  
  
 System.out.printf("Future Value (Recursive): %.2f\n", futureRecursive);  
 System.out.printf("Future Value (Iterative): %.2f\n", futureIterative);  
 }  
}

## 4. Analysis

- Time Complexity (Recursive): O(n) — one call per year  
- Space Complexity: O(n) due to the recursive call stack  
- Optimized Version: Iterative approach reduces space to O(1)  
  
In real-world financial systems where performance matters, the iterative or memoized version is preferred to avoid stack overflows and repeated computations.  
