**Week 1 :-**

**Skill :- Data structures and Algorithms**

**Hands-on 1 :- Exercise 2: E-commerce Platform Search Function**

## 1. Understand Asymptotic Notation

Big O notation describes how the runtime of an algorithm grows with the input size (n). It focuses on the worst-case scenario and gives an upper bound on time or space complexity.  
  
Best, Average, and Worst Case Scenarios:  
Linear Search:  
- Best Case: O(1)  
- Average Case: O(n)  
- Worst Case: O(n)  
  
Binary Search:  
- Best Case: O(1)  
- Average Case: O(log n)  
- Worst Case: O(log n)

## 3. Implementation

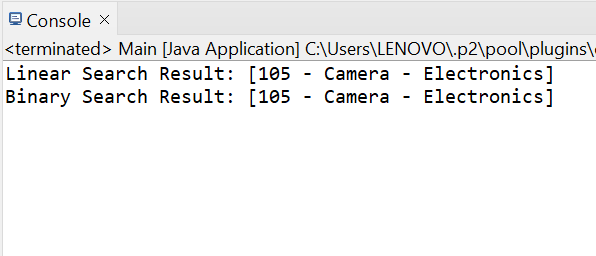
// Product.java  
public class Product {  
 int productId;  
 String productName;  
 String category;  
  
 public Product(int productId, String productName, String category) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 }  
  
 public String toString() {  
 return "[" + productId + " - " + productName + " - " + category + "]";  
 }  
}  
  
// ProductSearch.java  
import java.util.Arrays;  
import java.util.Comparator;  
  
public class ProductSearch {  
  
 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) {  
 Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));  
 int left = 0, right = products.length - 1;  
 while (left <= right) {  
 int mid = (left + right) / 2;  
 int cmp = products[mid].productName.compareToIgnoreCase(targetName);  
 if (cmp == 0) return products[mid];  
 else if (cmp < 0) left = mid + 1;  
 else right = mid - 1;  
 }  
 return null;  
 }

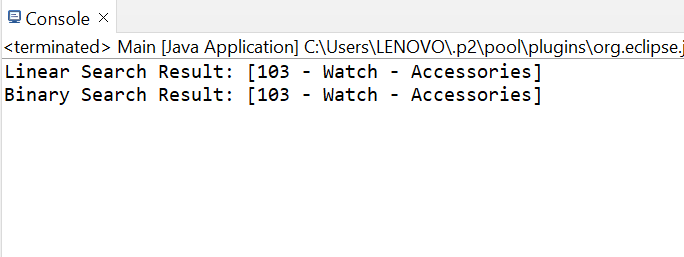
}

// Main.java  
public class Main {  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(101, "Laptop", "Electronics"),  
 new Product(102, "Shoes", "Footwear"),  
 new Product(103, "Watch", "Accessories"),  
 new Product(104, "Bag", "Fashion"),  
 new Product(105, "Camera", "Electronics")  
 };  
  
 Product foundLinear = ProductSearch.linearSearch(products, "Watch");  
 System.out.println("Linear Search Result: " + foundLinear);  
  
 Product foundBinary = ProductSearch.binarySearch(products, "Watch");  
 System.out.println("Binary Search Result: " + foundBinary);  
 }  
}

## 4. Analysis

Time Complexity:  
- Linear Search: O(n)  
- Binary Search: O(log n)  
  
Binary search is more suitable for large datasets where the data can be sorted, making the search more efficient.





Hands-on 2 :- Exercise 7: Financial Forecasting

## 1. Understand Recursive Algorithms

Recursion is when a method calls itself to solve a smaller subproblem of the original. Each call works with smaller input until it reaches a base case. It simplifies problems like tree traversal, factorial, etc.

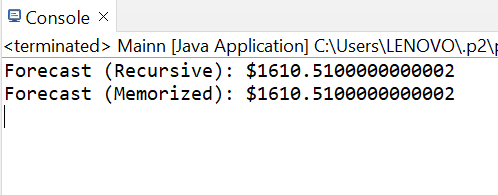
## 3. Implementation

// FinancialForecast.java  
public class FinancialForecast {  
  
 public static double forecastValueRecursive(double initialValue, double growthRate, int years) {  
 if (years == 0) return initialValue;  
 return forecastValueRecursive(initialValue, growthRate, years - 1) \* (1 + growthRate);  
 }  
  
 public static double forecastValueMemoized(double initialValue, double growthRate, int years, double[] memo) {  
 if (years == 0) return initialValue;  
 if (memo[years] != 0) return memo[years];  
 memo[years] = forecastValueMemoized(initialValue, growthRate, years - 1, memo) \* (1 + growthRate);  
 return memo[years];  
 }  
}  
  
// Main.java  
public class Main {  
 public static void main(String[] args) {  
 double initial = 1000.0;  
 double growthRate = 0.10;  
 int years = 5;  
  
 double futureValue = FinancialForecast.forecastValueRecursive(initial, growthRate, years);  
 System.out.println("Forecast (Recursive): $" + futureValue);  
  
 double[] memo = new double[years + 1];  
 double futureValueMemo = FinancialForecast.forecastValueMemoized(initial, growthRate, years, memo);  
 System.out.println("Forecast (Memorized): $" + futureValueMemo);  
 }  
}

## 4. Analysis

### Time Complexity:

* **Naive Recursion**: O(n)
  + One recursive call per year, from n down to 0.
* **Memoized Version**: O(n)
  + Uses cached results to avoid redundant calculations.
  + Useful if there are **multiple** overlapping recursive calls.



|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |