**WEEK – 1**

**DATA STRUCTURE AND ALGORITHMS**

**QUESTIONS - 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.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**EXPLANATION :**

**Big O Notation:**

Big O notation describes the upper bound of an algorithm’s time or space complexity. It gives an estimate of how the algorithm behaves as the input size (n) grows.

**Best, Average, and Worst-Case Scenarios in Search :**

**1. Linear Search**

* **Best Case: O(1)**  
  ➤ Target element is the **first** item in the list.
* **Average Case: O(n)**  
  ➤ Target is somewhere **in the middle**. On average, we check half the list.
* **Worst Case: O(n)**  
  ➤ Target is at the **end** or **not present** at all — we scan the whole list.

**2. Binary Search**

Binary Search only works if the array is **sorted**.

* **Best Case: O(1)**  
  ➤ Target is exactly at the **middle** on the first comparison.
* **Average Case: O(log n)**  
  ➤ We repeatedly divide the array in half to find the element.
* **Worst Case: O(log n)**  
  ➤ Maximum number of divisions needed when the element is near the ends or not found.

**Comparison:**

**Linear vs Binary Search :**

| **Feature** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Time Complexity (Best)** | O(1) (if item is at the beginning) | O(1) (if item is at the middle) |
| **Time Complexity (Avg)** | O(n/2) → O(n) | O(log n) |
| **Time Complexity (Worst)** | O(n) (if item is at the end or not found) | O(log n) |
| **Space Complexity** | O(1) | O(1) |
| **Requires Sorted Data** | ❌ No | ✅ Yes |
| **Data Structure** | Unordered Array/List | Sorted Array/List |
| **Performance** | Slower for large datasets | Much faster for large datasets |
| **Implementation** | Simple | Slightly more complex |

**Which is More Suitable for an E-Commerce Platform?**

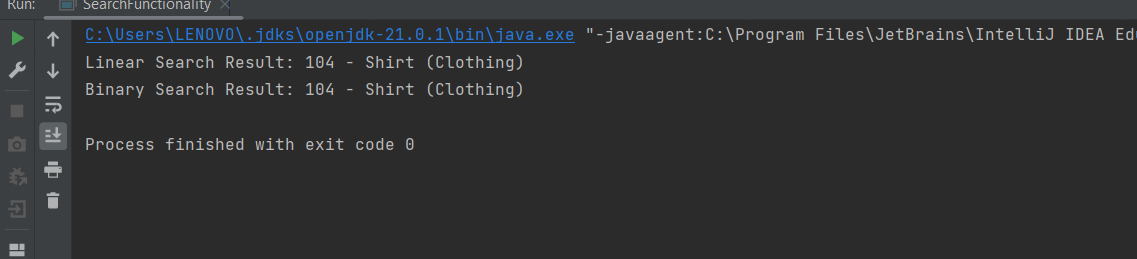
**🔹 Binary Search is more suitable when:**

* You have **millions of products** — Binary Search is **much faster** due to O(log n) time.
* Data is stored in a **sorted or indexed format**.
* You need **scalability and high performance** in search operations.

**CODE :**

import java.util.Arrays;  
import java.util.Comparator;  
  
public class SearchFunctionality {  
  
 // Product class  
 static 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 + ")";  
 }  
 }  
  
 // Linear Search  
 public static Product linearSearch(Product[] products, int targetId) {  
 for (Product product : products) {  
 if (product.productId == targetId) {  
 return product;  
 }  
 }  
 return null;  
 }  
  
 // Binary Search   
 public static Product binarySearch(Product[] products, int targetId) {  
 int low = 0, high = products.length - 1;  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 if (products[mid].productId == targetId) {  
 return products[mid];  
 } else if (products[mid].productId < targetId) {  
 low = mid + 1;  
 } else {  
 high = mid - 1;  
 }  
 }  
 return null;  
 }  
  
   
 public static void sortProducts(Product[] products) {  
 Arrays.*sort*(products, Comparator.*comparingInt*(p -> p.productId));  
 }  
  
   
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(103, "Shoes", "Footwear"),  
 new Product(101, "Laptop", "Electronics"),  
 new Product(104, "Shirt", "Clothing"),  
 new Product(102, "Mobile", "Electronics")  
 };  
  
 int targetId = 104;  
  
 // Linear Search  
 Product resultLinear = *linearSearch*(products, targetId);  
 System.*out*.println("Linear Search Result: " + resultLinear);  
  
 // Binary Search  
 *sortProducts*(products);   
 Product resultBinary = *binarySearch*(products, targetId);  
 System.*out*.println("Binary Search Result: " + resultBinary);  
 }  
}

**OUTPUT :**

****

**Binary Search is More Suitable than Linear Search in Ecommerce Platform .**

**QUESTIONS – 7 :**

**Financial Forecasting :**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**SOLUTION :**

RECURSION :

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem. It simplifies complex problems by breaking them down into simpler subproblems.

**Time Complexity**

**1. Basic Recursion:**

* Time Complexity: O(n) — One recursive call per year.
* Space Complexity: O(n) — Due to recursive call stack.

**2. Memoized Recursion:**

* Time Complexity: O(n) — But faster due to cached values.
* Space Complexity: O(n) — For both cache and call stack.

**Optimization :**

In recursive problems with overlapping subproblems (like Fibonacci), **recalculating values repeatedly** wastes time. Though this specific problem is linear, **memoization** still improves performance slightly, especially for deeper recursive trees or complex financial logic.

**CODE :**

public class FinancialForecast {

public static double calculateFutureValue(int years, double currentValue, double growthRate) {

if (years == 0) {

return currentValue;

}

return calculateFutureValue(years - 1, currentValue, growthRate) \* (1 + growthRate);

}

public static double calculateFutureValueMemo(int years, double currentValue, double growthRate, double[] cache) {

if (years == 0) {

return currentValue;

}

if (cache[years] != 0) {

return cache[years];

}

cache[years] = calculateFutureValueMemo(years - 1, currentValue, growthRate, cache) \* (1 + growthRate);

return cache[years];

}

public static void main(String[] args) {

double currentValue = 10000;

double growthRate = 0.08;

int years = 5;

double futureVal = calculateFutureValue(years, currentValue, growthRate);

System.out.println("Future Value (Recursive): ₹" + futureVal);

double[] cache = new double[years + 1];

double futureValMemo = calculateFutureValueMemo(years, currentValue, growthRate, cache);

System.out.println("Future Value (Memoized): ₹" + futureValMemo);

}

**}**

**OUTPUT :**

