**Week 1- Hands on**

**Name- Adi Jain**

**Superset ID- 6390400**

**Algorithms\_Data Structures**

**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.

**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.

**Solution**

**Big O Notation Explanation**

Big O notation describes how the runtime or space requirements of an algorithm grow relative to input size n. It's used to evaluate algorithm **efficiency**, especially when scaling.

* **O(1):** Constant time
* **O(log n):** Logarithmic time (e.g., binary search)
* **O(n):** Linear time (e.g., linear search)
* **O(n log n):** Linearithmic (e.g., merge sort)
* **O(n²):** Quadratic (e.g., bubble sort)

**Best, Average, and Worst Case for Search**

* **Linear Search:**
  + **Best:** O(1) → Found at the beginning.
  + **Average:** O(n/2) → On average, found in the middle.
  + **Worst:** O(n) → Found at the end or not present.
* **Binary Search:**
  + **Best:** O(1) → Middle element is the target.
  + **Average/Worst:** O(log n) → Each step halves the search space.
  + **Note:** Requires sorted array

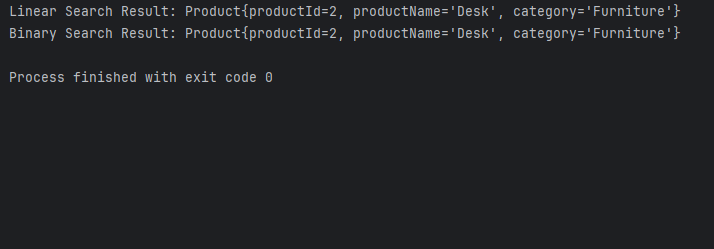
**Code**

package ecommerce\_search;  
  
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;  
 }  
  
 @Override  
 public String toString() {  
 return "Product{" +  
 "productId=" + productId +  
 ", productName='" + productName + '\'' +  
 ", category='" + category + '\'' +  
 '}';  
 }  
}

package ecommerce\_search;  
  
import java.util.Arrays;  
import java.util.Comparator;  
  
public class searchservice {  
   
 public static product linearSearch(product[] products, int targetId) {  
 for (product product : products) {  
 if (product.productId == targetId) {  
 return product;  
 }  
 }  
 return null;  
 }  
   
 public static product binarySearch(product[] products, int targetId) {  
 int left = 0, right = products.length - 1;  
  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 if (products[mid].productId == targetId) {  
 return products[mid];  
 } else if (products[mid].productId < targetId) {  
 left = mid + 1;  
 } else {  
 right = mid - 1;  
 }  
 }  
 return null;  
 }  
  
 public static void sortProductsById(product[] products) {  
 Arrays.*sort*(products, Comparator.*comparingInt*(p -> p.productId));  
 }  
}

package ecommerce\_search;  
  
public class main {  
 public static void main(String[] args) {  
 product[] products = {  
 new product(3, "Laptop", "Electronics"),  
 new product(1, "Phone", "Electronics"),  
 new product(2, "Desk", "Furniture")  
 };  
  
 product foundLinear = searchservice.*linearSearch*(products, 2);  
 System.*out*.println("Linear Search Result: " + foundLinear);  
  
  
 searchservice.*sortProductsById*(products);  
 product foundBinary = searchservice.*binarySearch*(products, 2);  
 System.*out*.println("Binary Search Result: " + foundBinary);  
 }  
}

**Output**



**Analysis**

When analyzing the performance of linear search versus binary search in an e-commerce platform, it's essential to consider both time complexity and scalability. Linear search checks each product sequentially until it finds a match or reaches the end of the list. In the best-case scenario, the desired product is found at the beginning, resulting in a time complexity of **O(1)**. However, on average and in the worst case, it takes **O(n)** time, which becomes inefficient as the number of products grows.

Binary search, in contrast, significantly reduces the number of comparisons by dividing the sorted dataset in half at each step. It has a time complexity of **O(log n)** for both average and worst-case scenarios. This makes it highly efficient for large datasets. The trade-off is that binary search requires the product list to be sorted beforehand, typically by product ID or another comparable field.

For an e-commerce platform, where product catalogs often include thousands or millions of items, **binary search is clearly more suitable**. It provides faster and more predictable performance for exact lookups like product ID searches. While maintaining a sorted list is necessary, this overhead is minor compared to the performance benefits.

Linear search may still be used for small datasets or for searches based on unstructured fields like partial names or descriptions. Overall, binary search is the better choice for optimizing search performance in structured e-commerce systems.

**Exercise 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**

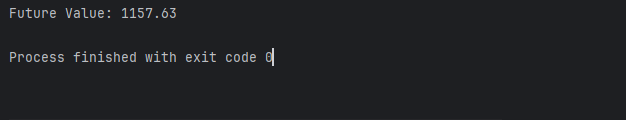
Recursion is a programming technique where a function calls itself to solve smaller instances of a problem. A recursive function typically has:

* A **base case**: when to stop the recursion.
* A **recursive case**: how the function calls itself on a smaller input.
* It can simplify complex problems like tree traversal, mathematical sequences (e.g., Fibonacci), and divide-and-conquer strategies.
* In financial forecasting, recursion can help break down calculations over multiple time periods.

**Code**

public class financialforecasting {  
  
 public static double futureValueRecursive(double v0, double r, int n) {  
  
 if (n == 0) {  
 return v0;  
 }  
  
 return (1 + r) \* *futureValueRecursive*(v0, r, n - 1);  
 }  
  
 public static void main(String[] args) {  
 double initialValue = 1000.0; // V₀  
 double growthRate = 0.05; // r = 5%  
 int periods = 3; // n = 3 years  
  
 double result = *futureValueRecursive*(initialValue, growthRate, periods);  
 System.*out*.printf("Future Value: %.2f\n", result);   
 }  
}

**Output**



**Analysis**

The recursive algorithm for calculating future value uses a simple approach where the function calls itself with a reduced number of periods until it reaches the base case. This recursive method has a **linear time complexity of O(n)**, where n is the number of periods. For each recursive call, one multiplication is performed, and the function continues to call itself n times before reaching the base case. Thus, the number of recursive calls increases directly with the number of periods, making it efficient for small n but potentially problematic for large values.

Despite its simplicity, recursion can become inefficient due to **stack overhead and risk of stack overflow** for large n values (e.g., 1000+). Each recursive call adds a new frame to the call stack, consuming memory. Java, by default, has a limited stack size, and excessive recursion can cause a stack overflow error..

To optimize this, the recursive solution can be converted into an **iterative approach**. The iterative version computes the same result by looping from 0 to n, multiplying the value by (1 + r) in each iteration. This eliminates the need for recursive calls and significantly improves memory efficiency, with a **space complexity of O(1)**.