**WEEK 1**

NAME: ROHIT KARUKOLA

SUPERSET ID : 6383975

**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.

Big O notation describes how the runtime of an algorithm grows with the input size n. It **ignores constants** and focuses on **scalability**.

### **Best, Average, and Worst Cases (for Searching)**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | O(1) – match first | O(n/2) ≈ O(n) | O(n) – last or not found |
| **Binary Search** | O(1) – match mid | O(log n) | O(log n) |

Setup the Product Class

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 productId + " - " + productName + " (" + category + ")";

}

}

Implement Linear & Binary Search

import java.util.Arrays;

import java.util.Comparator;

public class Main {

// Linear Search

public static Product linearSearch(Product[] products, String name) {

for (Product p : products) {

if (p.productName.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

// Binary Search (Assumes array is sorted by productName)

public static Product binarySearch(Product[] products, String name) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = products[mid].productName.compareToIgnoreCase(name);

if (cmp == 0) return products[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(102, "Shoes", "Fashion"),

new Product(103, "Watch", "Accessories"),

new Product(104, "Mobile", "Electronics"),

new Product(105, "Book", "Education")

};

// Sort for binary search

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

// Test Linear Search

Product result1 = linearSearch(products, "Watch");

System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Not Found"));

// Test Binary Search

Product result2 = binarySearch(products, "Watch");

System.out.println("Binary Search Result: " + (result2 != null ? result2 : "Not Found"));

}

}

## Analysis and Comparison

### Time Complexity

| **Search Type** | **Time Complexity** | **Sorted Data Required?** |
| --- | --- | --- |
| Linear Search | O(n) | No |
| Binary Search | O(log n) | Yes (sorted by name) |

For real-world e-commerce platforms, binary search works well for sorted data, but large-scale systems typically use HashMaps or search engines like Elasticsearch for faster and more scalable search. For this exercise, binary search is the more efficient choice compared to linear search.

**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.

### **What is Recursion?**

Recursion is a method where a function **calls itself** to solve smaller instances of a problem.  
 It helps simplify problems like factorials, Fibonacci sequences, and financial forecasting with compound calculations.

## Setup – Recursive Future Value Formula

Let’s assume a simple **compound interest-like growth**:

ini

CopyEdit

FV = PV × (1 + r)^n

Where:

* FV = future value
* PV = present value
* r = growth rate
* n = number of years

CODE:

public class Forecast {

// Recursive method to calculate future value

public static double futureValue(double presentValue, double rate, int years) {

if (years == 0) {

return presentValue;

}

return futureValue(presentValue \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

double pv = 10000; // initial investment

double rate = 0.08; // 8% annual growth

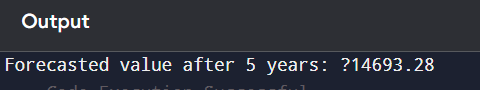
int years = 5; // forecast for 5 years

double result = futureValue(pv, rate, years);

System.out.printf("Forecasted value after %d years: ₹%.2f", years, result);

}

}



## **Analysis**

### **Time Complexity:**

* The time complexity is **O(n)** since the function makes one recursive call per year.

### **Optimization:**

To avoid overhead from recursion:

* Convert to an **iterative approach**
* Or use **memoization** if intermediate results repeat (e.g., in other recursive problems like Fibonacci)