**Mandatory HandsOn**

**Exercise 2: E-commerce Platform Search Function**

**Big O Notation (Asymptotic Notation)**

Big O Notation describes the upper bound of an algorithm’s running time. It tells us how an algorithm’s performance scales with input size. We will be able to compare the algorithms without the need to implement them. It is useful to predict algorithm’s performance as the dataset grows.

**Search Operation Cases**

**Best Case:**

The item is found in the first attempt so O(1) for both linear and binary search algorithms.

**Average Case:**

The item is somewhere in the middle O(n/2) ~ O(n) for linear, O(log n) for binary.

**Worst Case:**

The item is nto found in the list or is the last one .O(n) for linear and O(log n) for binary.

**Program:**

import java.util.Arrays;

class Product {

int id;

String name;

String category;

Product(int id, String name, String category) {

this.id = id;

this.name = name;

this.category = category;

}

public String toString() {

return id + " - " + name + " - " + category;

}

}

public class SimpleSearch {

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

for (int i = 0; i < products.length; i++) {

if (products[i].name.equalsIgnoreCase(name)) {

return i;

}

}

return -1;

}

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

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

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, "Shoes", "Footwear"),

new Product(3, "Watch", "Accessories"),

new Product(4, "Mobile", "Electronics"),

new Product(5, "Shirt", "Clothing")

};

String searchName = "Watch";

int index1 = linearSearch(products, searchName);

if (index1 != -1)

System.out.println("Linear Search: " + products[index1]);

else

System.out.println("Linear Search: Product not found");

Arrays.sort(products, (a, b) -> a.name.compareToIgnoreCase(b.name));

int index2 = binarySearch(products, searchName);

if (index2 != -1)

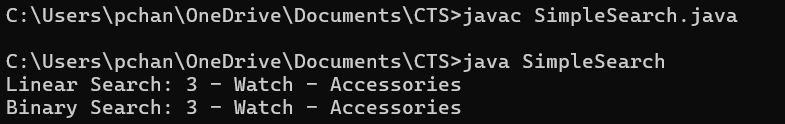
System.out.println("Binary Search: " + products[index2]);

else

System.out.println("Binary Search: Product not found");

}

}



**Analysis:**

Binary Search is better for large datasets that are frequently searched but not frequently updated. Linear Search is acceptable for small or unsorted datasets or when insertion and deletion happen often. We can also use advanced structures like HashMaps or search trees for real time, large-scale product searches in E-Commerce Platforms.

**Exercise 7: Financial Forecasting**

**Recursion**

Recursion is a technique where a method calls itself again and again to solve smaller sub-problems. It is useful for problems like factorial, Fibonacci and Financial Forecasting.

In Financial forecasting it simplifies repeated calculations like compound growth over time.

**Program:**

public class FinancialForecast {

static double futureValueRecursive(double presentValue, double rate, int years) {

if (years == 0)

return presentValue;

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

}

static double futureValueIterative(double presentValue, double rate, int years) {

double value = presentValue;

for (int i = 1; i <= years; i++) {

value = value \* (1 + rate);

}

return value;

}

public static void main(String[] args) {

double presentValue = 10000;

double annualGrowthRate = 0.08;

int years = 5;

double resultRecursive = futureValueRecursive(presentValue, annualGrowthRate, years);

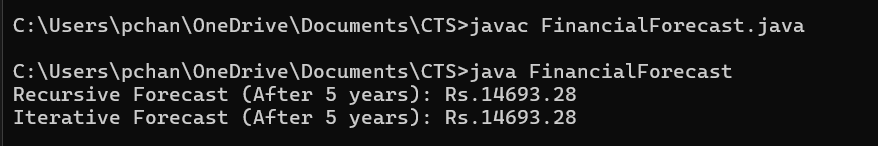
System.out.printf("Recursive Forecast (After %d years): Rs.%.2f\n", years, resultRecursive);

double resultIterative = futureValueIterative(presentValue, annualGrowthRate, years);

System.out.printf("Iterative Forecast (After %d years): Rs.%.2f\n", years, resultIterative);

}

}



**Analysis:**

The recursive approach in financial forecasting calculates the future value by repeatedly calling the same method for each year until it reaches the year 0. It is simple and easy with a time complexity of O(n), where n is the number of years. Recursion can use more memory due to repeated function calls. An iteration uses a loop instead and is more efficient for large values and avoids memory issues. Both methods gives the same result but iteration is better for performance in real applicatio