**Week 1 – Mandatory Hands on:**

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

Big O Notation

Big O notation is a mathematical technique for expressing the performance of an algorithm as a function of time or space, respectively, as the size of the input (n) increases. It provides an upper limit on the time consumed, informing us of the growth of the algorithm regardless of machine or programming environment.

**Product.java**

package Harsha;

import java.util.Arrays;

import java.util.Comparator;

public class Product {

static class Item {

int productId;

String productName;

String category;

public Item(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String toString() {

return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

public static Item linearSearch(Item[] items, String targetName) {

for (Item item : items) {

if (item.productName.equalsIgnoreCase(targetName)) {

return item;

}

}

return null;

}

public static Item binarySearch(Item[] items, String targetName) {

Arrays.*sort*(items, Comparator.*comparing*(i -> i.productName.toLowerCase()));

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

while (low <= high) {

int mid = (low + high) / 2;

int cmp = items[mid].productName.compareToIgnoreCase(targetName);

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

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

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Item[] items = {

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

new Item(102, "Shirt", "Clothing"),

new Item(103, "Phone", "Electronics"),

new Item(104, "Shoes", "Footwear")

};

String target = "Phone";

System.*out*.println("Linear Search Result:");

Item result1 = *linearSearch*(items, target);

System.*out*.println(result1 != null ? result1 : "Product not found.");

System.*out*.println("\nBinary Search Result: ");

Item result2 = *binarySearch*(items, target);

System.*out*.println(result2 != null ? result2 : "Product not found.");

System.*out*.println("\nTime Complexity:");

System.*out*.println("Linear Search: Best O(1), Worst O(n)");

System.*out*.println("Binary Search: Best O(1), Worst O(log n)");

System.*out*.println("\nRecommendation:");

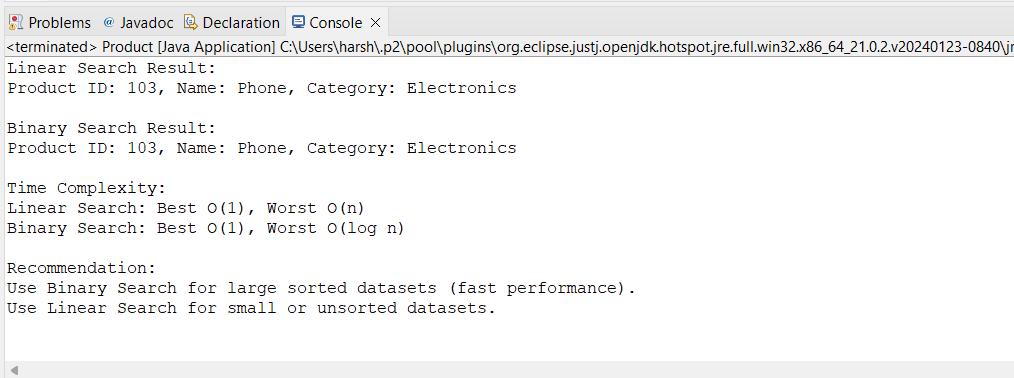
System.*out*.println("Use Binary Search for large sorted datasets (fast performance).");

System.*out*.println("Use Linear Search for small or unsorted datasets.");

}

}

**OUTPUT:**

****

**Exercise 7: Financial Forecasting**

Recursion is a programming methodology in which a function calls itself to compute smaller instances of a problem until a base case is reached. It is especially useful for problems that can be decomposed into smaller subproblems of similar type (e.g., Fibonacci, factorial, tree traversals).  
  
In forecasting finance, recursion can be applied to determine the future value of an investment such that every year is built on the accumulation of the previous year's growth.

**Financial.java**

package Harsha;

public class Financial {

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

if (years == 0)

return presentValue;

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

}

public static void main(String[] args) {

double presentValue = 30000;

double rate = 0.06;

int years = 6;

double future = *futureValue*(presentValue, rate, years);

System.*out*.println("Present Value: " + presentValue);

System.*out*.println("Annual Growth Rate: " + (rate \* 100) + "%");

System.*out*.println("Years: " + years);

System.*out*.println("Predicted Future Value: " + future);

}

}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**