**Week 1 – Hands-on : Data structures and Algorithms**

Exercise 2 : E-commerce Platform Search Function

**File**: Main.java

import com.Ecommerce.Product;

class Main {

    public static void main(String[] args) {

        Product myStore = new Product();

        Product product1 = new Product(1, "Laptop", "Electronics");

        Product product2 = new Product(2, "Mobile", "Electronics");

        Product product3 = new Product(3, "Tooth Brush", "Lifestyle");

        myStore.addProduct(product1);

        myStore.addProduct(product2);

        myStore.addProduct(product3);

        System.out.println("Using Linear search -- O(n) complexity");

        Product productSearch1 = myStore.linearSearch(3);

        Product.printProducts(productSearch1);

        System.out.println();

        System.out.println("Using Binary search -- O(log n) complexity");

        Product productSearch2 = myStore.binarySearch(1);

        Product.printProducts(productSearch2);

        System.out.println();

        System.out.println("Using Binary search -- O(log n) complexity");

        Product productSearch3 = myStore.binarySearch(10);

        Product.printProducts(productSearch3);

    }

**File**: Product.java

package com.Ecommerce;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class Product {

    int productId;

    String productName;

    String category;

    List<Product> listForBinSearch;

    List<Product> listForLinearSearch;

    public Product() {

        listForBinSearch = new ArrayList<>();

        listForLinearSearch = new ArrayList<>();

    }

    // creates new product with values given

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

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    // to add a new product to the lists

    public void addProduct(Product product) {

        listForBinSearch.add(product);

        listForLinearSearch.add(product);

    }

    // Linear Search Implementation

    public Product linearSearch(int id) {

        for (Product product : listForLinearSearch) {

            if (product.productId == id) {

                return product;

            }

        }

        return null;

    }

    // Binary Search Implementation

    public Product binarySearch(int id) {

        Collections.sort(listForBinSearch, Comparator.comparingInt(p -> productId));

        int start = 0;

        int end = listForBinSearch.size() - 1;

        while (start <= end) {

            int mid = start + (end - start) / 2;

            int midProductId = listForBinSearch.get(mid).productId;

            if (midProductId == id) {

                return listForBinSearch.get(mid);

            } else if (midProductId < id) {

                start = mid + 1;

            } else {

                end = mid - 1;

            }

        }

        return null;

    }

    // printing a product

    public static void printProducts(Product product) {

        if (product == null) {

            System.out.println("Product not found for ID :( ");

        } else {

            System.out.println("Product ID : " + product.productId);

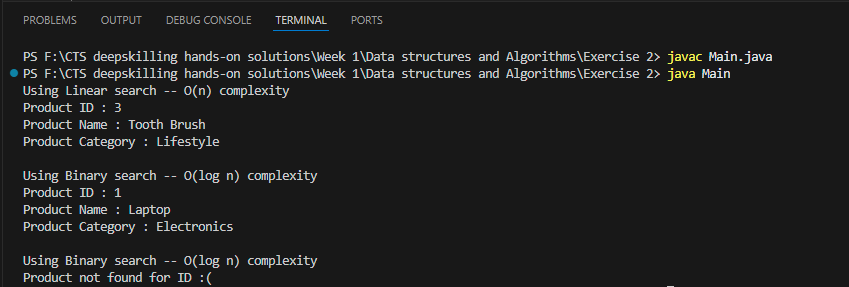
            System.out.println("Product Name : " + product.productName);

            System.out.println("Product Category : " + product.category);

        }

    }

}

OUTPUT :  


**Analysis** :

* **Binary Search (O(logn))** is more suitable for our E-commerce platform because for larger datasets in worst case **Linear Search(O(n))** has to scan through ‘n’ number of products, if n=100 then linear search would do **100 iterations**, But if we use binary search for the same search space it does **only 7 iterations** **.**
* Therefore **Binary Search is the best option** for this scenario.

Exercise 7 : Financial Forecasting

**File**: Main.java

import com.Finance.FinancialForecast;

class Main {

    public static void main(String[] args) {

        int foreCastYears = 10;

        double startingValue = 10000; // initial amount

        double growthRate = 0.25; // annual growth rate percentage (25%);

        double result = FinancialForecast.forecastValue(foreCastYears, startingValue, growthRate);

        System.out.printf("Forecasted value after %d years is Rs. %.2f", foreCastYears, result);

    }

}

**File**: FinancialForecast.java

package com.Finance;

public class FinancialForecast {

    public static double forecastValue(int year, double initialValue, double annualGrowthRate) {

        if (year == 0) {

            return initialValue;

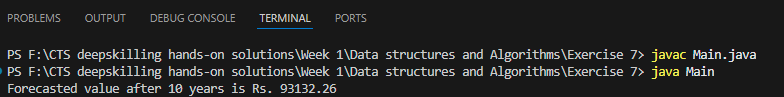
        }

        return forecastValue(year - 1, initialValue, annualGrowthRate) \* (1 + annualGrowthRate);

    }

}

OUTPUT:



**Analysis** :

* **Time Complexity** – The recursive algorithm has time complexity of O(n), where n is the number of years, Since it makes one recursive call for each year.
* We can optimize the recursive solution using **Memoization.**