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

**Big-O Notation:**

Big O notation is a mathematical notation used to find upper bound of time taken by a data structure or algorithm. It provides a way to compare the performance of different algorithms and data structures, and to predict their behaviour as the input size increases.

Product.java

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;

    }

    public String toString() {

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

    }

}

LinearSearch.java

public class LinearSearch {

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

      for (Product p : products) {

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

              return p;

          }

      }

      return null;

    }

}

BinarySearch.java

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

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

        int left = 0, right = products.length - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

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

            if (cmp == 0)

                return products[mid];

            else if (cmp < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

    public static void sortProducts(Product[] products) {

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

    }

}

Main.java

public class Main {

    public static void main(String[] args) {

       Product[] products = {

            new Product(3, "Smartphone", "Electronics"),

            new Product(1, "T-shirt", "Clothing"),

            new Product(5, "Blender", "Home"),

            new Product(2, "Laptop", "Electronics"),

            new Product(4, "Shoes", "Footwear")

        };

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

        Product result1 = LinearSearch.linearSearch(products, "Smartphone");

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

        BinarySearch.sortProducts(products);

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

        Product result2 = BinarySearch.binarySearch(products, "Laptop");

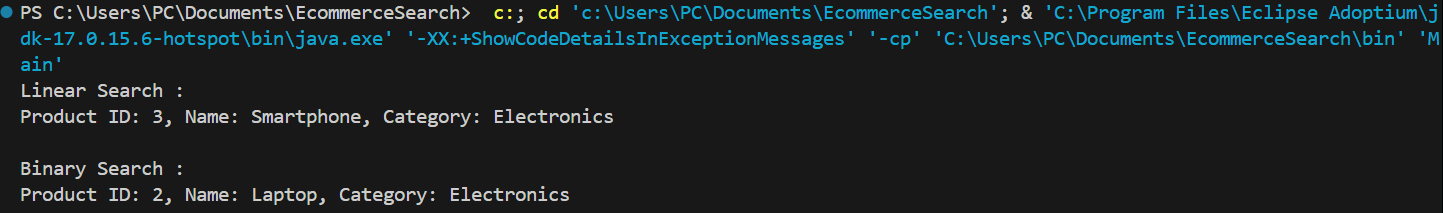
        System.out.println(result2 != null ? result2 : "Product not found");

    }

}

Output:

| **Search Type** |  | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- | --- |
| **Linear Search** |  | O(1) | O(n) | O(n) |
| **Binary Search** |  | O(1) | O(log n) | O(log n) |

Suitability: For large databases, binary search is faster and more scalable, hence it is suitable for this scenario.

**Exercise 7: Financial Forecasting:**

Recursion:

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

App.java

public class App {

    public static double futureValue(double initialValue, double growthRate, int years) {

        if (years == 0) {

            return initialValue;

        }

        return futureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

    }

    public static void main(String[] args) {

        double initial = 10000;

        double growthRate = 0.08;

        int years = 5;

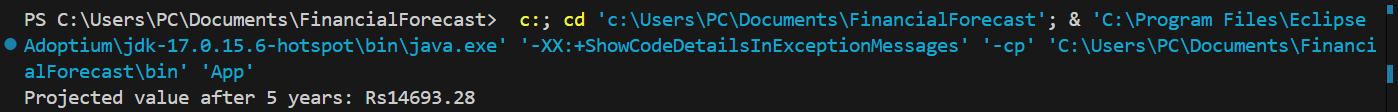
        double result = futureValue(initial, growthRate, years);

        System.out.printf("Projected value after %d years: Rs%.2f\n", years, result);

    }

}

Output:



Time Complexity Analysis:

Each year calls the function recursively until base case is heat. Therefore time complexity depends on the number of years or n. Hence, time complexity: O(n)

Optimization:

Using linear iteration can avoid recursion and call stack overhead.

public static double futureValueIterative(double initialValue, double growthRate, int years) {

    double result = initialValue;

    for (int i = 0; i < years; i++) {

        result \*= (1 + growthRate);

    }

    return result;

}