Superset id-6361766

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

Code:-

public class ProductSearch {

    static 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 static Product linearSearch(Product[] *list*, String *target*) {

        for (Product p : list) {

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

                return p;

            }

        }

        return null;

    }

    public static Product binarySearch(Product[] *list*, String *target*) {

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

        while (low <= high) {

            int mid = (low + high) / 2;

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

            if (compare == 0) {

                return list[mid];

            } else if (compare < 0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return null;

    }

    public static void main(String[] *args*) {

        Product[] items = {

            new Product(1, "Laptop", "Electronics"),

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

            new Product(3, "Shoes", "Fashion"),

            new Product(4, "Book", "Stationery"),

            new Product(5, "Watch", "Accessories")

        };

        java.util.Arrays.sort(items, (a, b) -> a.name.compareToIgnoreCase(b.name));

        String searchItem = "Shoes";

        Product found1 = linearSearch(items, searchItem);

        if (found1 != null) {

            System.out.println("Linear Search: Found " + found1.name + " in " + found1.category);

        } else {

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

        }

        Product found2 = binarySearch(items, searchItem);

        if (found2 != null) {

            System.out.println("Binary Search: Found " + found2.name + " in " + found2.category);

        } else {

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

        }

    }

}

Output:-

Linear Search: Found Shoes in Fashion

Binary Search: Found Shoes in Fashion

**Analysis:-**

#### ****Linear Search****

Linear search goes through the list one element at a time.

**Best Case**: The target is found at the first position — **O(1)** time.

**Worst Case**: The target is at the end or not present — **O(n)** time.

**Space Complexity**: **O(1)**, as it uses no extra memory apart from variables.

Linear search works on **unsorted arrays**, and it’s simple to implement. However, it becomes inefficient when the dataset is large.

#### ****Binary Search****

Binary search requires the array to be **sorted** by the key being searched. It repeatedly divides the array in half to narrow down the search.

**Best Case**: The target is found at the middle — **O(1)** time.

**Worst Case**: The target is not found after repeated halving — **O(log n)** time.

**Space Complexity**: **O(1)** for the iterative version (used here).

Binary search is highly efficient for large, sorted datasets. It performs significantly better than linear search in terms of time complexity but requires the data to be sorted beforehand.

**Exercise 7: Financial Forecasting**

**Code:-**

*//6361766*

public class Ff {

    public static double forecast(double *value*, double *rate*, int *years*) {

        if (years == 0) {

            return value;

        } else {

            return forecast(value \* (1 + rate), rate, years - 1);

        }

    }

    public static void main(String[] *args*) {

        double initialValue = 1000.0;

        double growthRate = 0.10; *// 10% interest is used we can make it any.*

        int years = 5;

        double futureValue = forecast(initialValue, growthRate, years);

        System.out.printf("Future Value after %d years: %.2f\n", years, futureValue);

    }

}

### ****Analysis:-****

### ****Time Complexity****

The recursive method calls itself once for each year.

So, the time complexity is **O(n)**, where n is the number of years.

### ****Space Complexity****

Each recursive call adds a new frame to the stack.

So, the space complexity is also **O(n)** due to the call stack.

### ****Optimization****

Although this example is already **tail-recursive**, for very large values of n, you can:

**Use iteration** instead of recursion to save stack space.

public static double forecastIterative(double value, double rate, int years) {

for (int i = 0; i < years; i++) {

value \*= (1 + rate);

}

return value;

}

This reduces space complexity to **O(1)**.