1. E-Commerce Platform search function  
   In an e-commerce platform, search functionality is one of the most critical features, enabling users to quickly locate products from a large inventory. The efficiency of this search directly impacts user experience and system performance. Therefore, understanding search algorithms and time complexity analysis is crucial.

**Asymptotic Notation and Big O**

Asymptotic notation, particularly Big O notation, is used to describe the efficiency of algorithms as input size grows. It allows us to analyze how an algorithm scales in terms of time (speed) and space (memory).

Key Cases:

* Best Case (Ω): Minimum time taken (e.g., element is first in the list)
* Average Case (Θ): Expected time for random input
* Worst Case (O): Maximum time taken (element is last or not present)

Code:

import java.util.Arrays;

import java.util.Comparator;

class Product {

    int productId;

    String productName;

    String category;

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

        this.productId = id;

        this.productName = name;

        this.category = cat;

    }

}

public class Ecommerce{

    // Linear search by product name

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

        for (Product p : products) {

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

                return p;

            }

        }

        return null;

    }

    // Binary search by product name

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

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

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

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

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

            else right = mid - 1;

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(102, "Shoes", "Footwear"),

            new Product(103, "Keyboard", "Electronics"),

            new Product(104, "Book", "Education")

        };

        // Linear search

        Product foundLinear = linearSearch(products, "Shoes");

        if (foundLinear != null)

            System.out.println("Linear Search: Found " + foundLinear.productName);

        else

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

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

        // Binary search

        Product foundBinary = binarySearch(products, "Shoes");

        if (foundBinary != null)

            System.out.println("Binary Search: Found " + foundBinary.productName);

        else

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

    }

}

Output:



Analysis:

| **Method** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n/2) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

* Binary search is faster, but requires a sorted array.

1. Financial Forecasting

Financial forecasting involves predicting future values based on historical performance. It is commonly used in budgeting, investment planning, and business strategy. One simple way to model such predictions is through recursive functions, especially when modeling compound growth.

**What is Recursion?**

Recursion is a technique where a function calls itself to solve smaller sub-problems. It is particularly useful when a problem can be broken down into similar sub-problems.

In the code, the compound growth formula used is:

FV(n) = FV(n-1) \* (1 + r)

At each recursive step, the investment grows by a fixed rate.

Code:

public class FinanceForecasting {

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

        if (years == 0) {

            return presentValue;

        }

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

    }

    public static void main(String[] args) {

        double presentValue = 1000; // Initial amount

        double growthRate = 0.05;   // 5% growth per year

        int years = 5;

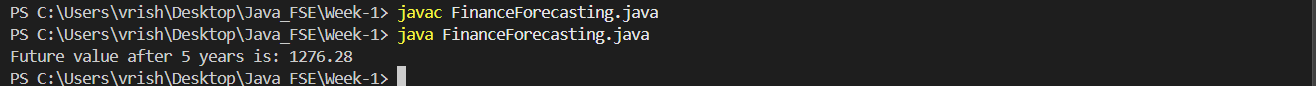
        double futureValue = calculateFutureValue(presentValue, growthRate, years);

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

    }

}

Output:



Analysis:

* The recursive method has O(n) time complexity (linear), because it makes one recursive call per year.
* It can be optimized using memorization(DP) or convert it to iteration if needed.