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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**SOLUTION:**

**1.Explain Big O notation and how it helps in analyzing algorithms**

**🔹 What is Big O Notation?**

**Big O notation is a way to describe how the performance of an algorithm changes as the input size grows.**

| **Notation** | **Meaning** | **Example** |
| --- | --- | --- |
| **O(1)** | **Constant time** | **Accessing array[i]** |
| **O(n)** | **Linear time** | **Linear search** |
| **O(log n)** | **Logarithmic time** | **Binary search** |
| **O(n²)** | **Quadratic time** | **Bubble sort** |

**🔹 Best, Average, and Worst Case (for search)**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | **O(1) – first item** | **O(n/2) ≈ O(n)** | **O(n) – last/not found** |
| **Binary Search** | **O(1) – middle** | **O(log n)** | **O(log n)** |

**🔸 Binary search is only possible on a sorted array.**

**2.**

**PRODUCT.JAVA:-**

**package search;**

**public class Product {**

**private int productId;**

**private String productName;**

**private String category;**

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

**this.productId = productId;**

**this.productName = productName;**

**this.category = category;**

**}**

**public int getProductId() {**

**return productId;**

**}**

**public String getProductName() {**

**return productName;**

**}**

**public String getCategory() {**

**return category;**

**}**

**public String toString() {**

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

**}**

**}**

**SEARCHING:-**

**package search;**

**import java.util.Arrays;**

**import java.util.Comparator;**

**public class SearchService {**

**// Linear Search by productName**

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

**for (Product p : products) {**

**if (p.getProductName().equalsIgnoreCase(name)) {**

**return p;**

**}**

**}**

**return null;**

**}**

**// Binary Search by productName (array must be sorted)**

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

**Arrays.sort(products, Comparator.comparing(Product::getProductName));**

**int low = 0, high = products.length - 1;**

**while (low <= high) {**

**int mid = (low + high) / 2;**

**int result = products[mid].getProductName().compareToIgnoreCase(name);**

**if (result == 0) {**

**return products[mid];**

**} else if (result < 0) {**

**low = mid + 1;**

**} else {**

**high = mid - 1;**

**}**

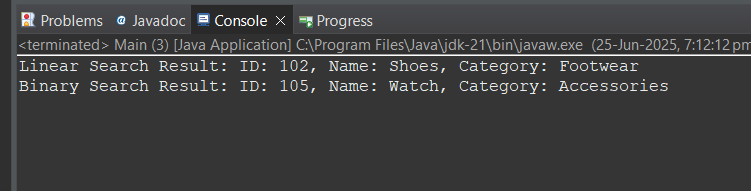
**}**

**return null;**

**}**

**}**

**OUTPUT:**

****

**4.**

**Time Complexity Comparison**

| **Search Type** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| **Linear Search** | **O(n)** | **No sorting required** |
| **Binary Search** | **O(log n)** | **Requires sorted array (O(n log n))** |

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**SOLUTION:**

**1.** Explain the concept of recursion and how it can simplify certain problems.

**Recursion is when a function calls itself to solve a smaller part of the original problem.**

** you want to compute the value of investment over n years.**

** Instead of looping year by year, recursion can calculate:**

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

**2.**

**package forecast;**

**public class ForecastCalculator {**

**// Recursive method to calculate future value**

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

**// Base case**

**if (years == 0) {**

**return presentValue;**

**}**

**// Recursive case**

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

**}**

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

**double presentValue = 10000; // Initial investment**

**double annualRate = 0.05; // 5% growth rate**

**int years = 10; // Forecast for 10 years**

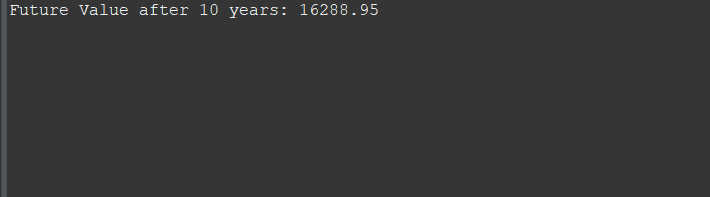
**double result = *futureValue*(presentValue, annualRate, years);**

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

**}**

**}**

**OUTPUT:**

****

**4.**

* + Discuss the time complexity of your recursive algorithm.
  + Explain how to optimize the recursive solution to avoid excessive computation.

**⏱️ Time Complexity**

**Each recursive call decreases years by 1:**

* **Time Complexity: O(n) → where n = number of years**
* **Space Complexity: O(n) → due to recursive call stack**

**Optimization Suggestion: Use Iteration or Memoization**

**To avoid deep recursion and stack overflow for large n, use:**

**Iterative Version (Alternative):**

**java**

**CopyEdit**

**public static double futureValueIterative(double presentValue, double rate, int years) {**

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

**presentValue \*= (1 + rate);**

**}**

**return presentValue;**

**}**

**This has the same O(n) time complexity but O(1) space — no call stack overhead.**