**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. Asymptotic Notation**

**What is Big O Notation?**

Big O notation is used to describe the performance or complexity of an algorithm in terms of time or space. It helps determine how scalable or efficient an algorithm is, especially for large input sizes. It usually refers to the **worst-case scenario**.

**Time Complexities**

| **Algorithm** | **Time Complexity** |
| --- | --- |
| Linear Search | O(n) |
| Binary Search | O(log n) |

2. Product Class Setup:

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 "[" + productId + ", " + productName + ", " + category + "]";

}

}

3. Search Function Implementation:

import java.util.Arrays;

import java.util.Comparator;

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 "[" + productId + ", " + productName + ", " + category + "]";

}

}

public class Main {

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

for (Product p : products) {

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

return p;

}

}

return null;

}

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(1, "Laptop", "Electronics"),

new Product(2, "Shirt", "Fashion"),

new Product(3, "Book", "Education"),

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

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

};

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

Product result1 = linearSearch(products, "Book");

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

// Sorting the array for binary search

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

System.out.println("\n=== Binary Search ===");

Product result2 = binarySearch(products, "Book");

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

}

}

Output:

**4. Analysis and Recommendation**

**Time Complexity Comparison**

| **Algorithm** | **Time Complexity** | **Requires Sorted Array** |
| --- | --- | --- |
| Linear Search | O(n) | No |
| Binary Search | O(log n) | Yes |

* **Linear Search** is simple and requires no sorting but is inefficient for large datasets.
* **Binary Search** is much more efficient but requires the product list to be sorted first.

**Recommendation Suggestion:**  
Use **Binary Search** for fast performance if products can be stored or retrieved in a **sorted order**. It's ideal for scalable search in e-commerce platforms.

**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. Understanding Recursive Algorithms**

**Recursion** is a programming technique where a function calls itself to solve smaller instances of a problem. It is particularly useful for problems that have a naturally **repetitive or self-similar structure**.

* Simplifies code for problems like **factorials**, **Fibonacci series**, or **tree traversals**.
* Breaks complex tasks into smaller subproblems.
* Useful when past data is used to derive future data using a repeated logic.

**2. Setup**

**Formula:**

FutureValue(n) = CurrentValue × (1 + growthRate)^n

Where:

* n = number of years
* growthRate = expected yearly growth rate (e.g., 0.05 for 5%)

3. Implementation in Java

public class FinancialForecast {

public static double forecastValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

return forecastValue(currentValue, growthRate, years - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

double currentValue = 10000;

double growthRate = 0.08; // 8% growth per year

int years = 5;

double futureValue = forecastValue(currentValue, growthRate, years);

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

}

}

Output:

**4. Analysis**

**Time Complexity**

The recursive function forecastValue calls itself once per year, so:

* **Time Complexity:** O(n), where n is the number of years
* **Space Complexity:** O(n), due to the call stack

**Optimization Suggestions**

* For this specific problem, recursion is **not very heavy**, as each call does minimal computation.
* However, for large n, **iterative solutions** or **memoization** can be considered to improve performance and reduce stack overflow risk.