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

**Solution:**

### **What is Big O Notation?**

Big O notation is a way to describe how fast or slow an algorithm is based on input size.  
 It doesn't give actual time, but shows how time grows as data grows.

For example:

* O(1) → constant time (very fast, time doesn’t change with input size)
* O(n) → linear time (slowly increases as input grows)
* O(log n) → logarithmic time (very efficient for large data)

### **Best, Average, and Worst Cases:**

Search operations are fundamental to computer science and are used to locate a specific element within a collection of data. The performance of a search operation can vary significantly depending on the nature of the data structure and the position of the target element. The efficiency of a search is typically evaluated in three scenarios: **best case**, **average case**, and **worst case**.

**1. Best-Case Scenario**

**Definition:**  
 The best-case scenario occurs when the target element is found immediately, typically at the first position being checked.

**Explanation:**  
 In this scenario, the algorithm performs the minimum possible number of operations. For example, in a linear search, this would mean the desired element is located at the beginning of the list. This results in optimal performance, with the time complexity being **O(1)** — constant time.

### **2. Average-Case Scenario**

**Definition:**  
 The average-case scenario represents the expected performance over a set of typical inputs, assuming that the target element is equally likely to be located at any position in the dataset.

**Explanation:**  
 In linear search, the average case would imply that the target element is located somewhere in the middle of the list. On average, the algorithm would need to examine half of the elements. The time complexity in this case is **O(n)**, where *n* is the number of elements in the dataset.

### **3. Worst-Case Scenario**

**Definition:**  
 The worst-case scenario occurs when the search algorithm must examine every element in the dataset before finding the target or determining that it does not exist.

**Explanation:**  
 This happens when the element is located at the end of the data structure or is not present at all. Consequently, the algorithm performs the maximum number of comparisons, resulting in the highest time consumption. For linear search, this also leads to a time complexity of **O(n)**.

Hence, when searching for a product:

* **Best Case:** We find the item at the start (very fast).
* **Average Case:** It's somewhere in the middle.
* **Worst Case:** It's at the end or not found at all.

**Code:**

import java.util.\*;

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 static int linearSearch(Product[] pr, String name) {

for (int i = 0; i < pr.length; i++) {

if (pr[i].productName.equalsIgnoreCase(name)) {

return i;

}

}

return -1;

}

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (compare == 0) return mid;

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

else right = mid - 1;

}

return -1;

}

}

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Shampoo", "Personal Care"),

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

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

new Product(4, "Mobile", "Electronics")

};

int index1 = Product.linearSearch(products, "Book");

System.out.println("Linear Search: Found at index " + index1);

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

int index2 = Product.binarySearch(products, "Laptop");

System.out.println("Binary Search: Found at index " + index2);

}

}

## **Analysis**

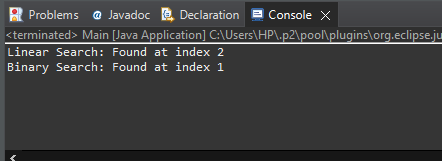
### **Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Description** |
| Linear Search | O(n) | Slower for large product lists |
| Binary Search | O(log n) | Much faster if list is sorted |

### **Which is Better?**

* Use **Linear Search** when:
  + The product list is small.
  + The list is **unsorted**.
* Use **Binary Search** when:
  + You have **a large list**.
  + You can **sort it once** and search many times.

**Output:**



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

**Solution :**

### **Concept of Recursion**

**Definition:**  
 Recursion is a programming technique where a function calls itself, either directly or indirectly, in order to solve a problem. Each recursive call works on a smaller or simpler instance of the original problem until it reaches a base case — a condition under which the recursion stops.

### **Key Components of Recursion**

1. **Recursive Case:**  
    This defines how the problem can be broken down into smaller subproblems of the same type.
2. **Base Case:**  
    This is the condition under which the recursion terminates. It prevents infinite recursion and ensures that the function returns a result without further recursive calls.

**Code:**

public class Main {

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

if (years == 0) {

return amount;

}

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

}

public static void main(String[] args) {

// TODO Auto-generated method stub

double currentValue = 10000;

double growthRate = 0.10;

int numberOfYears = 5;

double futureValue = calculateFutureValue(currentValue, growthRate, numberOfYears);

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

}

}

#### **Time Complexity**

Since our method calls itself once per year, the time complexity is O(n) where n = number of years.

Each year we do one multiplication and call the method again with years - 1.

#### **Optimization Idea**

If we were doing something like calculating Fibonacci, recursion would become slow because of repeating calculations.

In our case, it's okay because:

* No overlapping problems
* Only one call per level
* But if needed, we could:
* Use a loop instead of recursion (iterative way).

**Output:**

