**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.

ANSWER :

1. Understand Asymptotic Notation

Big O Notation:

1. Big O Notation describes the upper bound of an algorithm's running time.
2. It helps understand how the algorithm scales with input size n.
3. It hides constants and lower-order terms to focus on the growth rate.

| **Complexity** | **Example** | **Growth** |
| --- | --- | --- |
| O(1) | Constant time | Same regardless of input |
| O(log n) | Binary Search | Grows slowly |
| O(n) | Linear Search | Grows linearly |
| O(n²) | Bubble Sort | Grows quickly |

**Best, Average, and Worst Case for Search Operations**

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

### ****2. Setup: Product Class in Java****

public class Product {

int productId;

String productName;

String category;

public Product(int id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

public String toString() {

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

}

}

### ****3. Implementation****

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

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

for (Product product : products) {

if (product.productName.equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

#### ****Binary Search (Sorted by productName)****

import java.util.Arrays;import java.util.Comparator;

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

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

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

while (low <= high) {

int mid = (low + high) / 2;

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

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

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

else high = mid - 1;

}

return null;

}

**4. Analysis: Time Complexity Comparison**

| **Algorithm** | **Time Complexity** | **Requires Sorting?** | **Suitable For** |
| --- | --- | --- | --- |
| Linear Search | O(n) | No | Small or unsorted datasets |
| Binary Search | O(log n) | Yes (O(n log n)) | Large, sorted datasets |

Conclusion :- **Binary search** is much faster for large datasets **but requires sorted data**.

**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.

**ANSWER :**

1. Recursive Algorithms

Recursion :-

Recursion is when a function calls itself to solve a smaller version of the problem.

It works by defining a base case (stop condition) and a recursive case (repeat step).

Use of Recursion :-

- Simplifies problems that have repeating patterns or can be broken down into smaller subproblems.

- Useful in mathematical series, financial projections, tree/graph traversal, etc.

2. Setup: Method to Calculate Future Value

Formula for future value prediction using compound interest :

FV = PV × (1 + r)^n

Where:

- FV = Future Value

- PV = Present Value

- r = growth rate per period

- n = number of periods (years)

3. Implementation

public class FinancialForecast {

public static double futureValue(double presentValue, double growthRate, int years) {

if (years == 0) {

return presentValue; // Base case

} else {

return futureValue(presentValue \* (1 + growthRate), growthRate, years - 1); // Recursive case

}

}

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05; // 5% annual growth

int years = 5;

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

System.out.printf("Future value after %d years = ₹%.2f\n", years, futureVal);

}

}

**4. Complexity and Optimization**

Time Complexity = O(n), where n is the number of years.

Optimized Iterative Version:

public static double futureValueIterative(double presentValue, double growthRate, int years) {

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

presentValue \*= (1 + growthRate);

}

return presentValue;

}