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

**1. Understanding Asymptotic Notation**

**Big O Notation**

Big O notation is used to describe the **time or space complexity** of an algorithm in terms of input size (**n**). It gives an upper bound on the growth rate of an algorithm, helping us understand its **scalability** and **performance**.

* Example:
  + A linear search has a time complexity of **O(n)** – time grows linearly with input size.
  + A binary search has a time complexity of **O(log n)** – very efficient for large datasets.

**Best, Average, and Worst Case Scenarios**

Search operations can have different performance in different cases:

* **Best Case**: The desired product is found on the first attempt.
  + Linear Search: O(1)
  + Binary Search: O(1)
* **Average Case**: The product is somewhere in the middle.
  + Linear Search: O(n/2) ≈ O(n)
  + Binary Search: O(log n)
* **Worst Case**: The product is at the end (or not present).
  + Linear Search: O(n)
  + Binary Search: O(log n)

**2. Setup**

**Product Class in Java**

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;

}

@Override

public String toString() {

return productId + " - " + productName + " (" + category + ")";

}

}

**3. Implementation**

**Linear Search**

public class SearchFunctions {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

}

**Binary Search**

* Assumes array is sorted by productName.

import java.util.Arrays;

import java.util.Comparator;

public class SearchFunctions {

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

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

if (compare == 0)

return products[mid];

else if (compare < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

public static void sortProducts(Product[] products) {

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

}

}

**4. Analysis**

**Time Complexity Comparison**

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

**Suitability for E-commerce Search**

* **Linear Search** is simple and doesn't require sorting, but **slow for large datasets**.
* **Binary Search** is **much faster** on large datasets but requires the data to be sorted, which adds overhead when updating the product list.

**Recommendation**:  
For an e-commerce platform with **frequent search queries and large product catalogs**, **binary search (on sorted arrays or using data structures like binary search trees or hash maps)** is more efficient.

In real-world systems, we typically use:

* **Hash-based searches** (e.g., HashMap) for constant time lookup.
* **Full-text search engines** (e.g., Elasticsearch) for more complex queries.

**Exercise 7: Financial Forecasting**

**1. Understand Recursive Algorithms**

What is Recursion?

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem until it reaches a base case. It is especially helpful in problems that can be broken down into smaller, similar sub-problems.

Why Use Recursion?

Recursion can simplify code, especially for problems involving:

* Repetitive calculations
* Divide-and-conquer logic (like in Fibonacci, tree traversals)
* Mathematical computations

For financial forecasting, recursion can be used to repeatedly apply a growth rate over a number of time periods.

**2. Setup: Future Value Formula**

We can model financial forecasting recursively using the formula:

Future Value=Current Value×(1+Growth Rate)^n

Where:

* n is the number of years (or periods)
* Growth Rate is expressed as a decimal (e.g., 5% → 0.05)

**3. Implementation**

Java Implementation of Recursive Forecasting

public class FinancialForecast {

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

if (years == 0) {

return currentValue; // base case

}

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

}

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

return currentValue \* Math.pow(1 + growthRate, years);

}

public static void main(String[] args) {

double currentValue = 10000.0;

double growthRate = 0.07; // 7% growth

int years = 5;

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

double optimizedResult = futureValueOptimized(currentValue, growthRate, years);

System.out.println("Recursive Forecast: $" + recursiveResult);

System.out.println("Optimized Forecast: $" + optimizedResult);

}

}

**4. Analysis**

**Time Complexity of Recursive Algorithm**

The recursive method performs one multiplication and one recursive call per year:

* **Time Complexity**: O(n)
* **Space Complexity**: O(n) due to call stack

Each function call is dependent on the previous one, so for n years, n calls are made.

**Optimizing the Recursive Solution**

To avoid excessive computation and **stack overflow** risks, consider:

1. **Memoization (for more complex cases)**: Cache results to avoid redundant calculations.
2. **Iterative Approach**: Use a loop instead of recursion to compute results in O(n) time but with O(1) space.
3. **Mathematical Optimization**: Use the power formula:

Future Value=Current Value×(1+Growth Rate)^n

This uses Math.pow() and gives an **O(1)** time and space complexity.