**Week 1:**

**Module 2: Data Structure And Algorithm**

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

**1. Understand Asymptotic Notation**

**Big O Notation:** Big O notation is a mathematical representation used to describe the upper bound of an algorithm's time complexity. It provides a way to express how the runtime of an algorithm grows relative to the size of the input data. This notation is crucial for analyzing the efficiency of algorithms, especially when dealing with large datasets.

* **O(1):** Constant time - the execution time remains the same regardless of the input size.
* **O(n):** Linear time - the execution time grows linearly with the input size.
* **O(log n):** Logarithmic time - the execution time grows logarithmically as the input size increases.
* **O(n^2):** Quadratic time - the execution time grows quadratically with the input size.

**Best, Average, and Worst-Case Scenarios for Search Operations:**

* **Linear Search:**
  + **Best Case:** O(1) - The target element is the first element in the array.
  + **Average Case:** O(n) - The target element is found somewhere in the middle of the array.
  + **Worst Case:** O(n) - The target element is not present in the array, requiring a full scan.
* **Binary Search:**
  + **Best Case:** O(1) - The target element is the middle element of the sorted array.
  + **Average Case:** O(log n) - The search space is halved with each iteration.
  + **Worst Case:** O(log n) - The target element is not present, but the search space is reduced logarithmically.

**2. Setup**

Create a class **Product** with attributes for searching, such as **productId**, **productName**, and **category**.

public class Product {

private String productId;

**3. Implementation**

Implement linear search and binary search algorithms. Store products in an array for linear search and a sorted array for binary search.

import java.util.Arrays;

Bookmark messageCopy messageExport

**Code:**

import java.util.Arrays;

// Product class

class Product implements Comparable<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 int compareTo(Product other) {

return this.productName.compareToIgnoreCase(other.productName); // for binary search

}

@Override

public String toString() {

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

}

}

// Main class

public class Main {

// 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 (on sorted array)

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// Main method

public static void main(String[] args) {

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(102, "Shampoo", "Beauty"),

new Product(103, "Notebook", "Stationery"),

new Product(104, "Mouse", "Electronics"),

new Product(105, "Book", "Education")

};

// Linear Search

System.out.println("🔍 Linear Search:");

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

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

// Binary Search

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

Arrays.sort(products); // must sort for binary search

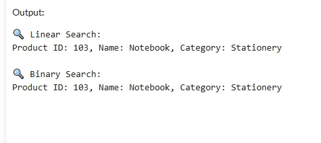
Product result2 = binarySearch(products, "Notebook");

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

    }

}

**Output:**

****

**Exercise 7: Financial Forecasting:**

**1. Understand Recursive Algorithms**

**Concept of Recursion:** Recursion is a programming technique where a function calls itself in order to solve a problem. It simplifies complex problems by breaking them down into smaller, more manageable subproblems. Each recursive call typically works on a smaller instance of the same problem, and the recursion continues until it reaches a base case, which is a condition that stops the recursion.

**Benefits of Recursion:**

* **Simplicity:** Recursive solutions can be more straightforward and easier to understand than their iterative counterparts.
* **Reduction of Code:** Recursive functions can often be written in fewer lines of code.
* **Natural Fit for Certain Problems:** Problems that can be defined in terms of smaller subproblems, such as tree traversals or combinatorial problems, are often more naturally expressed using recursion.

**2. Setup**

**Method to Calculate Future Value Using a Recursive Approach:** To calculate the future value based on a past growth rate, we can define a recursive function. The future value $ FV $ can be calculated using the formula:

$ FV = PV \times (1 + r)^n $

Where:

* PV is the present value (initial investment),
* r is the growth rate (as a decimal),
* n is the number of periods (years).

In a recursive approach, we can express this as:

$ FV(n) = FV(n-1) \times (1 + r) $

With the base case being $ FV(0) = PV $.

**3. Implementation**

Starts with initial investment (pv)

For each recursive call:

• Multiplies by (1 + rate) (annual growth)

• Decrements years until base case is reached

**4. Analysis**

**Time Complexity:** The time complexity of the recursive algorithm is $ O(n) $, where $ n $ is the number of periods. This is because the function makes $ n $ recursive calls, each reducing the problem size by 1 until it reaches the base case.

**Optimization:** To optimize the recursive solution and avoid excessive computation, we can use **memoization**. This technique involves storing the results of expensive function calls and reusing them when the same inputs occur again. This can significantly reduce the number of recursive calls.

By using memoization, the time complexity can be reduced to $ O(n) $ in terms of the number of unique calls, but the actual number of computations is significantly reduced, making it more efficient for larger values of $ n $.

Bookmark messageCopy messageExport

**Code:**

public class FinancialForecasting {

// Recursive method to calculate future value

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

if (years == 0) {

return initialValue;

}

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

}

public static void main(String[] args) {

double initialValue = 1000.0; // starting value

double growthRate = 0.05; // 5% growth rate

int years = 5;

double result = futureValue(initialValue, growthRate, years);

System.out.printf("Predicted future value after %d years: %.2f\n", years, result);

}

}

**Output:**

