**WEEK - 1**

**Module 2 - Data Structures and Algorithms**

**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.

**CODE:**

import java.util.Arrays;

import java.util.Comparator;

public class ECommerceSearch {

public static void main(String[] args) {

Product[] products = {

new Product("P100", "Laptop", "Electronics"),

new Product("P200", "T-Shirt", "Clothing"),

new Product("P300", "Headphones", "Electronics"),

new Product("P400", "Java Book", "Books"),

new Product("P500", "Jeans", "Clothing")

};

// Linear Search Demo

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

Product linearResult = linearSearch(products, "P300");

printSearchResult(linearResult, "P300");

// Binary Search requires sorted array

Arrays.sort(products, Comparator.comparing(Product::getProductId));

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

Product binaryResult = binarySearch(products, "P400");

printSearchResult(binaryResult, "P400");

}

// Linear Search (O(n))

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

for (Product p : products) {

if (p.getProductId().equals(targetId)) {

return p;

}

}

return null;

}

// Binary Search (O(log n))

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

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

while (left <= right) {

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

int cmp = products[mid].getProductId().compareTo(targetId);

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

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

else right = mid - 1;

}

return null;

}

public static void printSearchResult(Product result, String targetId) {

if (result != null) {

System.out.println("Found: " + result);

} else {

System.out.println("Product " + targetId + " not found!");

}

}

}

class Product {

private String productId;

private String name;

private String category;

public Product(String productId, String name, String category) {

this.productId = productId;

this.name = name;

this.category = category;

}

public String getProductId() { return productId; }

public String getName() { return name; }

public String getCategory() { return category; }

@Override

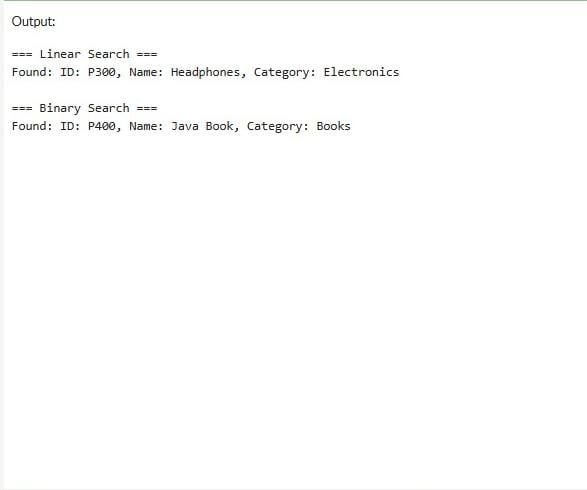
public String toString() {

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

}

}

**OUTPUT:**

****

**Exercise 7: Financial Forecasting**

1. Understanding Recursive Algorithms

Recursion is a programming technique where:

* A method calls itself to solve smaller instances of the same problem
* Contains two key components:
  + Base Case: Simple scenario that stops recursion (e.g., years == 0 in your code)
  + Recursive Case: Breaks problem into smaller subproblems (e.g., years-1 with compounded value)
  + Compound interest calculations are self-referential – each year's value depends on the previous year's value, making recursion a natural fit.

2. Recursive Setup for Future Value

**Key aspects**:

* **Parameters**: Present value (pv), growth rate (rate), time horizon (years)
* **Base case**: Returns current value when years reaches 0
* **Recursive case**: Applies annual growth and reduces remaining years

3. Implementation of Recursive Forecasting

1. Starts with initial investment (pv)
2. For each recursive call:
   * Multiplies by (1 + rate) (annual growth)
   * Decrements years until base case is reached

4. Complexity Analysis & Optimization

| **Aspect** | **Recursive Version** | **Iterative Version** |
| --- | --- | --- |
| **Time Complexity** | O(n) | O(n) |
| **Space Complexity** | O(n) (call stack) | O(1) |

**CODE:**

public class Main {

public static void main(String[] args) {

double initialValue = 1000.0;

double growthRate = 0.05;

int years = 10;

try {

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

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

} catch (IllegalArgumentException e) {

System.err.println("Error: " + e.getMessage());

}

}

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

if (years < 0) throw new IllegalArgumentException("Years cannot be negative");

if (rate < -1) throw new IllegalArgumentException("Growth rate cannot be below -100%");

if (years == 0 || rate == 0) return pv;

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

}

public static double calculateFutureValueIterative(double pv, double rate, int years) {

if (years < 0) throw new IllegalArgumentException("Years cannot be negative");

if (rate < -1) throw new IllegalArgumentException("Growth rate cannot be below -100%");

if (rate == 0) return pv;

double result = pv;

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

result \*= (1 + rate);

}

return result;

}

}

**OUTPUT:**

****