**Cognizant Deep Nurture 4.0 Hands-on Exercise    
   
   
   
   
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.

1. Setup the Product Class

// File: Product.java

public 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;

}

public String toString() {

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

}

}

2. Implement Search Algorithms

// File: SearchEngine.java

import java.util.Arrays;

import java.util.Comparator;

public class SearchEngine {

// Linear search by productName

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

for (Product product : products) {

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

return product;

}

}

return null;

}

// Binary search by productName (array must be sorted)

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

// Sort array by productName for binary search

public static void sortProducts(Product[] products) {

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

}

}

3.Test the Search Function

// File: Main.java

public class Main {

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Shoes", "Fashion"),

new Product(103, "Book", "Education"),

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

new Product(105, "Tablet", "Electronics")

};

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

Product result1 = SearchEngine.linearSearch(products, "Phone");

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

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

SearchEngine.sortProducts(products); // must sort before binary search

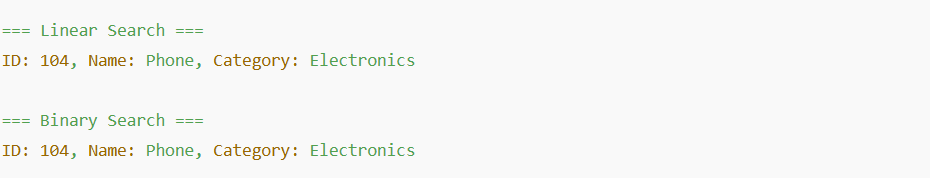
Product result2 = SearchEngine.binarySearch(products, "Phone");

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

}

}

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:**
   * 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.

   
Recursive Implementation in Java

// File: FinancialForecast.java

public class FinancialForecast {

// Recursive method to calculate future value

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

// Base case

if (years == 0) {

return presentValue;

}

// Recursive case

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

}

public static void main(String[] args) {

double presentValue = 10000.0; // Initial investment

double rate = 0.05; // 5% annual growth

int years = 10; // Forecasting for 10 years

double futureValue = calculateFutureValue(presentValue, rate, years);

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

}

}   
   
   
   
   
Memoization

import java.util.HashMap;

public class OptimizedForecast {

static HashMap<Integer, Double> memo = new HashMap<>();

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

if (years == 0) return presentValue;

if (memo.containsKey(years)) return memo.get(years);

double value = (1 + rate) \* calculateFutureValue(presentValue, rate, years - 1);

memo.put(years, value);

return value;

}

public static void main(String[] args) {

double future = calculateFutureValue(10000.0, 0.05, 10);

System.out.printf("Future Value (optimized): %.2f%n", future);

}

}

Iteration

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

double result = presentValue;

for (int i = 1; i <= years; i++) {

result \*= (1 + rate);

}

return result;

}

OUTPUT: 

da8c10bd07aa7b52f5bf3f5625885bd4.png