**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.

**Step:**

**1.Understand Asymptotic Notation:**

1) Big O notation describes the **performance** or **complexity** of an algorithm as the size of the input (n) grows. It gives an upper bound of how long an algorithm takes in the worst case. Best case, Worst case and Average case.

\* Best**-**case**:** The best possible performance (least time).

\* Average**-**case**:** The expected performance for a random input.

\* Worst**-**case**:** The slowest performance.

2) 1. Linear search

* Best-case: **O (1)** (element is the first item).
* Average-case: **O(n)**.
* Worst-case: **O(n)** (element is the last item or not found).

2. Binary search

* Best-case: **O (1)** (element is the middle).
* Average-case: **O (log n)**.
* Worst-case: **O (log n)**.

**2.Setup:**

package Ecommerce;

public class Product {

private int productId;

private String productName;

private String category;

public Product (int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

@Override

public String toString() {

return "Product {" + "productId=" + productId + ", productName='" + productName + '\'' + ", category='" + category + '\'' + '}';

}

}

**3.Implementation:**

import java.util.Arrays;

public class Searchdemo {

public static void main (String [] args) {

Product [] products = {

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

new Product (102, "Smartphone", "Electronics"),

new Product (103, "Headphones", "Accessories"),

new Product (104, "Keyboard", "Accessories"),

new Product (105, "Mouse", "Accessories")

};

// Linear Search

Product resultLinear = *linearSearch*(products, 103);

System.out.println("Linear Search Result: " + resultLinear);

// For binary search: ensure array is sorted by productId

Arrays.*sort*(products, (p1, p2) -> Integer.*compare*(p1. getProductId (), p2. getProductId ()));

Product resultBinary = *binarySearch*(products, 103);

System.out.println("Binary Search Result: " + resultBinary);

}

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

for (Product p: products) {

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

return p;

}

}

return null;

}

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int midId = products[mid].getProductId();

if (midId == targetId) {

return products[mid];

} else if (midId < targetId) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

}

**4.Analysis:**

Time complexity

| **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) |

I will choose Binary Search for the e‑commerce platform. Because it is quicker to find the datasets, in ecommerce data there are so many data to find so I choose binary search. It is highly effective because it works with O (log n) time.

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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.

**1)Recursion algorithm**: Recursion is when a method calls itself. It’s a powerful approach that can simplify problems like financial forecasting, where future values can be defined based on prior results.

**2)Setup and Implementation:**

package myfirstproject;

public class FinancialForecast {

public static void main (String [] args) {

double presentValue = 1000.0;

double annualGrowthRate = 0.05; // 5%

int years = 10;

// Calling the recursive method

double result = *futureValue*(presentValue, annualGrowthRate, years);

System.*out*.println(" ");

System.*out*.println("Future Value after " + years + " years: " + result);

}

// This method sets up and implements the recursion

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

if (years == 0) {

return presentValue; // Base case

}

// Recursive call

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

}

}

**4)Analysis:**

Time complexity: O (n)

For optimization we can use formula or loops,

Formula

futureValue = presentValue \* Math.pow(1 + rate, years);

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**