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

Sparshak Ghosh

Mandatory Hands-on

**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.

**ANSWERS**

1. **Big O notation** is used to describe the time or space complexity of an algorithm in terms of input size (n). It helps us analyze the runtime and memory usage as the input size increases, such as O (1) for representing constant time and O (log n) for logarithmic time. Also, it ignores lower-order terms to give a general idea of performance.

Best, Average, and Worst-Case Scenarios for Search Algorithms:

1. **Best Case:** Element is found at the first index of the data structure.
2. **Average Case:** Element is found somewhere in the middle of the data structure.
3. **Worst Case:** Element not found or found at the end, the whole data structure is traversed.

**2.**

Sparshak Ghosh

Mandatory Hands-on

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;

}

}

**3.**

**Linear Search (Array can be unsorted.)**

public class LinearSearch

{

public static Product search (Product[] products, int id)

{

for (Product product : products)

{

if (product.getProductId() == id)

return product;

}

return null;

}

}

**Binary Search (Array has to be sorted in any order.)**

Mandatory Hands-on

Sparshak Ghosh

public class BinarySearch

{

public static Product searchById(Product[] products, int id)

{

int left = 0;

int right = products.length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

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

if (midId == id)

{

return products[mid];

}

else if (midId < id)

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

return null;

}

}

**Storing Data for Linear Search and a sorted array for Binary Search.**

import java.util.\*;

public class Test

{

public static void main(String[] args)

{

Product[] products = {

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

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

new Product(103, "Shoes", "Footwear"),

new Product(102, "Shirt", "Clothing")

};

int searchId = 103;

Product result1 = LinearSearch.search(products, searchId);

if (result1 != null)

Mandatory Hands-on

Sparshak Ghosh

{

System.out.println("LinearSearch found: " + result1.getProductName());

}

else

{

System.out.println("LinearSearch found: none");

}

Product[] sortedProducts = Arrays.copyOf(products, products.length);

Arrays.sort(sortedProducts, (a, b) -> a.getProductId() - b.getProductId());

Product result2 = BinarySearch.searchById(sortedProducts, searchId);

if (result2 != null)

{

System.out.println("BinarySearch found: " + result2.getProductName());

}

else

{

System.out.println("BinarySearch found: none");

}

}

}

**4.**

Time Complexity Comparison

**Linear Search**

* **Best Case:** O (1) – when the element is at the beginning
* **Average Case:** O (n) – element is somewhere in the middle
* **Worst Case:** O (n) – element is at the end or not present

**Binary Search**

* **Best Case:** O (1) – when the element is in the middle
* **Average Case:** O (log n)
* **Worst Case:** O (log n)

As the search needs to be optimized for faster performance, we can use Binary Search as the data will be sorted usually ( using ProductID or ProductName ).

**Exercise 7: Financial Forecasting**

Sparshak Ghosh

Mandatory Hands-on

**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.

**ANSWERS**

**1.**

Recursion is a programming technique where a method calls itself to solve smaller instances of the same problem. It is useful for problems that involve repeated calculations. In financial forecasting, recursion can be used to calculate the future value of an investment by applying a growth rate over several periods.

**2.**

The future value of an investment is calculated using the formula: FV = PV / (1 + r)n

Which is FV(n) = FV(n - 1) / (1 + r) with the base case: FV(0) = PV

public class FinancialForecast

{

public static double forecast(double presentValue, double rate, int years)

{

if (years == 0)

return presentValue;

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

}

}

**3.**

Sparshak Ghosh

public class FinancialForecast

{

public static double forecast(double presentValue, double rate, int years)

{

if (years == 0)

return presentValue;

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

}

public static void main(String[] args)

{

double pv = 1000.0;

double rate = 0.05;

int years = 5;

double futureValue = forecast(pv, rate, years);

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

}

}

**4.**

Time Complexity: O(n), where n is the number of years.

**Optimizing the Recursive Solution**

The recursive solution can be optimized by converting it to an iterative approach. Recursion can lead to excessive computation and stack memory usage when the number of recursive calls becomes large.

public static double forecastIterative(double presentValue, double rate, int years)

{

double result = presentValue;

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

{

result \*= (1 + rate);

}

return result;

}

Mandatory Hands-on