**Algorithms\_Data Structures**

**Week 1**

**Mandatory Hands-on**

**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.

**Solution:**

**Step 1: Understand Asymptotic Notation**

**What is Big O Notation?**

Big O notation describes the **upper bound of an algorithm's runtime or space requirement** as the input size grows. It helps developers:

* Analyze and compare the performance of algorithms.
* Identify bottlenecks.
* Design scalable solutions.

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

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) – first item match | O(n/2) ≈ O(n) | O(n) – last item or not found |
| Binary Search | O(1) – middle item match | O(log n) | O(log n) – repeated halving |

**Step 2: Setup**

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;

}

@Override

public String toString() {

return productId + " - " + productName + " - " + category;

}

}

**Step 3: Implementation**

**Linear Search**

public class LinearSearch {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

}

**Binary Search (on productName, case-insensitive)**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

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

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

**Step 4: Analysis**

| **Algorithm** | **Time Complexity** | **When to Use** |
| --- | --- | --- |
| **Linear Search** | O(n) | Unsorted lists, small datasets, rarely used |
| **Binary Search** | O(log n) | Sorted arrays, large datasets |

**Which is More Suitable?**

For an **e-commerce platform** with potentially **thousands or millions of products**, **binary search** is much more efficient:

* **Faster lookups** with O(log n) time.
* Requires sorted data – maintain sorted list or use a binary search tree / Trie / search index for even better performance.

**Example Usage:**

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

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

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

new Product(105, "T-shirt", "Fashion")

};

Product result1 = LinearSearch.search(products, "Phone");

Product result2 = BinarySearch.search(products, "Phone");

System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Not Found"));

System.out.println("Binary Search Result: " + (result2 != null ? result2 : "Not Found"));

}

}

**Output:**

Linear Search Result: 104 - Phone - Electronics

Binary Search Result: 104 - Phone - Electronics

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Solution:**

**Step 1: Understand Recursive Algorithms**

**What is Recursion?**

**Recursion** is a technique where a function calls itself to solve smaller instances of a problem. It’s useful for problems that can be divided into similar subproblems (e.g., Fibonacci series, factorials, tree traversal).

**Example of recursion in financial forecasting:**  
To predict a future value based on compound growth:

futureValue(n) = futureValue(n - 1) \* (1 + growthRate)

**Step 2: Setup**

We will create a method that calculates the future value using the formula:

FV = PV \* (1 + r)^n

Where:

* FV: future value
* PV: present value
* r: growth rate (e.g., 0.05 for 5%)
* n: number of years

**Step 3: Implementation**

**Recursive Method**

public class FinancialForecast {

// Recursive function to calculate future value

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

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

double presentValue = 10000; // ₹10,000

double annualRate = 0.05; // 5% annual growth

int years = 5;

double result = futureValue(presentValue, annualRate, years);

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

}

}

**Output:**

Future Value after 5 years: ₹12762.82

This means ₹10,000 invested at 5% annual growth will become ₹12,762.82 after 5 years.

**Step 4: Analysis**

**Time Complexity:**

* **Recursive Time Complexity**: O(n), where n is the number of years.
* **Space Complexity**: O(n) due to the call stack from recursion.

**Optimization:**

* Use **memoization** or **convert to iteration** to avoid repeated calculations and stack overflow for large n.

**Optimized (Iterative) Version:**

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

double result = presentValue;

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

result \*= (1 + rate);

}

return result;

}

Submitted By:  
Name: Shreyansh Kumar  
Superset Id: 6422403