**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.**

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.

**=**

**BIG O:**

*Big O notation* is stated as upper bound of complexity for an algorithm which is a way to measure the efficiency of an algorithm, especially how its runtime or memory usage changes with input size.

It is concerned with the leading term of the complexity, excluding constant factors and lower-order terms, to give an upper bound on the performance of the algorithm. Basically it describes the worst-case scenario or the maximum amount of resources an algorithm might use. Big O notation helps predict how an algorithm will perform with very large inputs. Big O notation allows developers to compare the efficiency of different algorithms without needing to run them on specific hardware or with specific implementations. It also helps to predict how an algorithm will behave with large datasets thst allows developers to choose the best algorithm to solve a task.

**Best, Average, and Worst Case in Search:**

**LINEAR SEARCH:**

Best Case: O(1) – First element s searched

Average Case: O(n)

Worst Case: O(n) - Last element or not found

Space complexity: O(1)

**BANARY SEARCH:**

Best Case: O(1) – Middle element searched

Average Case: O(log n)

Worst Case: O(log n) - element at first position

Space complexity: O(1)

**CODE PART:**

**PRODUCT CLASS**

public class Product {

int id;

String name;

String category;

Product(int id, String name, String category) {

this.id = id;

this.name = name;

this.category = category;

}

public String toString() {

return id + " - " + name + " (" + category + ")";

}

}

**Search class**

import java.util.Arrays;

public class Search {

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

for (Product p : products) {

if (p.name.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

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

Arrays.sort(products, (a, b) -> a.name.compareToIgnoreCase(b.name));

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (result == 0) {

return products[mid];

} else if (result < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

}

**Main class**

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "TShirt", "Clothing"),

new Product(3, "Phone", "Electronics"),

new Product(4, "Football", "Sports"),

new Product(5, "Toothpaste", "Stationery")

};

// Linear search

Product foundLinear = Search.linearSearch(products, "Football");

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

// Binary search

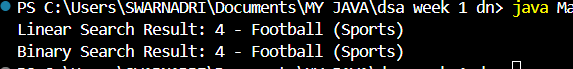
Product foundBinary = Search.binarySearch(products, "Football");

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

}

}

**OUTPUT:**

****

|  |  |  |
| --- | --- | --- |
| Linear Search: | O(n) |  |

|  |  |  |
| --- | --- | --- |
| Binary Search: | O(log n) |  |

* For our e-commerce platform, **binary search** will be a better choice because it is significantly faster compared to linear search on large product datasets. As our platform may deal with hundreds or thousands of items that need quick user search responses, so, binary search will give better performance. Although it requires sorting, that cost is small compared to the speed benefit during searches.

QUESTION

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**=**

**RECURSION:**

**Recursion** is process when a function in a program **calls itself** to solve a problem.

It goes on solving smaller and smaller versions of the same problem until it gets the **easiest part**, which it already knows how to solve. That is called the **base case**.

**How it simplifies:**

It keeps breaking the problem into **smaller and smaller subproblems**, until it reaches a **base case**.

* It **simplifies** problems that repeat in a similar way.
* It Makes code **shorter and easier to read**. Avoids writing long repetitive code
* Makes code **cleaner** and more **elegant**

**PREDICTING PROBLEM USING RECURSION:**

Suppose, we are selling a new product online.We notice that customers who buy the product tend to refer it to others.

We assume that:

In the first month, sale is 100 customers.

Each customer refers 1 new customer every month.So, each month, the number of new sales = total customers from last month. Total sales = old + new sales.

**CODE:**

public class ProductSales {

public static int predictSales(int month) {

if (month == 1) return 100; // base case: first month sales

return 2 \* predictSales(month - 1);

}

public static void main(String[] args) {

int targetMonth = 5;

int totalSales = predictSales(targetMonth);

System.out.println("Predicted product sales in month " + targetMonth + ": " + totalSales);

}

}

**OUTPUT:**



**COMPLEXITY:**

Each call reduces month by 1, and there’s **only** one recursive call per step. It grows linearly with the number of months.

**Time Complexity: O(n)**

where

* n is the number of months

So, it grows linearly with the number of months.

**How to reduce complexity:**

we may take 2 approaches…

1. if the code has a lot of repeated calculations, en we may use memorization method that the concept of dynamc programming. We store some values and use them rather than repeating the same calculation again and again.
2. We may convert the recursion to iteration.