**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.

**Solution**

**Big O Notation**

Big O describes the worst-case performance of an algorithm in terms of input size (n).

|  |  |  |
| --- | --- | --- |
| Algorithm | Time Complexity | Description |
| Linear Search | O(n) | Checks every element |
| Binary Search | O(log n) | Divides the array into halves each step |

**Best, Average, Worst Case**

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Best Case | Average Case | Worst Case |
| Linear Search | O(1) | O(n/2) ≈ O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

**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 "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

    }

}

**SearchDemo.java**

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

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

        for (Product product : products) {

            if (product.productId == targetId) {

                return product;

            }

        }

        return null;

    }

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

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

        while (left <= right) {

            int mid = (left + right) / 2;

            if (products[mid].productId == targetId) {

                return products[mid];

            } else if (products[mid].productId < targetId) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(101, "Shoes", "Fashion"),

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

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

        };

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

        Product result1 = linearSearch(products, 103);

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

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

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

        Product result2 = binarySearch(products, 103);

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

    }

}

**Output**

**A computer screen shot of a black background

AI-generated content may be incorrect.**

**Linear Search**

* **Time Complexity:** O(n)
* **Space Complexity:** O(1)
* **Use case:** Small datasets or unsorted data.

**Binary Search**

* **Time Complexity:** O(log n)
* **Space Complexity:** O(1)
* **Use case:** Large datasets where data is **sorted**.

|  |  |
| --- | --- |
| **Search Method** | **Best When...** |
| **Linear Search** | Dataset is small or unsorted |
| **Binary Search** | Dataset is large and pre-sorted by ID |

**For an optimized e-commerce platform**, **binary search is preferred**, provided the product list is sorted by productId.

**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.

**Solution**

What is Recursion?

* Recursion is when a method calls itself to solve a smaller part of the same problem.
* It’s useful for problems that follow a repetitive or breakdown pattern (e.g., calculating interest over years, Fibonacci, factorial, etc.).

Example

To calculate future value with compound growth: FV = PV × (1 + r)^n

Can be expressed recursively as: FV(n) = FV(n-1) × (1 + r)

Base case: FV(0) = initial value

**FinancialForecast.java**

public class FinancialForecast {

    public static double calculateFutureValue(double initialAmount, double growthRate, int years) {

        if (years == 0) {

            return initialAmount;

        }

        return calculateFutureValue(initialAmount, growthRate, years - 1) \* (1 + growthRate);

    }

    public static void main(String[] args) {

        double initialInvestment = 10000;

        double annualGrowthRate = 0.07;

        int years = 5;

        double futureValue = calculateFutureValue(initialInvestment, annualGrowthRate, years);

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

    }

}

**Output**

**A black screen with white text

AI-generated content may be incorrect.**

**Time Complexity:**

* The time complexity is **O(n)** for n years (1 recursive call per year).
* Space complexity is also **O(n)** due to the call stack.

**Optimization Suggestions:**

1. **Memoization** (if repeated sub-problems):
   * Store intermediate results in a map/array.
   * Not strictly needed here since each call is unique.
2. **Iterative Alternative** (recommended for large n):

public static double calculateFutureValueIterative(double initialAmount, double growthRate, int years) {

double value = initialAmount;

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

value \*= (1 + growthRate);

}

return value;

}

**Time Complexity:** O(n)

**Space Complexity:** O(1)

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Time Complexity** | **Space Complexity** | **Use Case** |
| Recursive | O(n) | O(n) | Clear logic, small n |
| Iterative | O(n) | O(1) | Large n, performance critical |