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

**Big O Notation**

* Big O notation describes the **upper bound** of an algorithm's running time as input size increases.
* It helps us **compare algorithms** based on efficiency and **predict performance**.

| **Notation** | **Meaning** |
| --- | --- |
| O(1) | Constant time (best) |
| O(log n) | Logarithmic time |
| O(n) | Linear time |
| O(n log n) | Linearithmic |
| O(n²) | Quadratic (slower) |

**Best, Average, and Worst Case for Search**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) (1st element) | O(n/2) ≈ O(n) | O(n) (last or not found) |
| Binary Search | O(1) (middle) | O(log n) | O(log n) |

**Implementation:**

import java.util.\*;

// Product class

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 + ")";

    }

}

public class ECommerceSearch {

    // Linear search

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

        for (Product p : products) {

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

                return p;

            }

        }

        return null;

    }

    // Binary search

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

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

        while (left <= right) {

            int mid = (left + right) / 2;

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

            if (cmp == 0)

                return products[mid];

            else if (cmp < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

    // Main method

    public static void main(String[] args) {

        Product[] productList = {

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

            new Product(2, "Shoes", "Fashion"),

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

            new Product(4, "Watch", "Accessories")

        };

        // Linear search

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

        Product result1 = linearSearch(productList, "Phone");

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

        // Sort before binary search

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

        System.out.println("\nBinary Search Result:");

        Product result2 = binarySearch(productList, "Phone");

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

    }

}

**Output:**

A white background with black text

AI-generated content may be incorrect.

**Analysis:**

**⏱️ Time Complexities**

| **Algorithm** | **Time Complexity** |
| --- | --- |
| Linear Search | O(n) |
| Binary Search | O(log n) |

**Which is better for e-commerce search?**

* **Binary Search is faster (O(log n))**, but requires a **sorted array**.
* **Linear Search is more flexible**, works on **unsorted data**.

**Conclusion**

* If product data is **rarely changing and sorted** we can use **Binary Search**.
* If the array is **dynamic or unsorted** we should use **Linear Search** or consider advanced techniques like **Hashing** or **Tries** for better performance.

**Exercise 7: Financial Forecasting**

**What is Recursion?**

* **Recursion** is a method where a function calls **itself** to solve smaller subproblems of the original problem.
* Useful for problems that can be broken down into **similar subproblems**, like calculating factorials, Fibonacci numbers, or, in this case, **predicting future values** based on a **recurring growth rate**.

**Implementation:**

public class FinancialForecast {

    // Recursive method

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

        if (years == 0)

            return initialAmount;

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

    }

    // Optimized

    public static double predictWithMemo(double initialAmount, double growthRate, int years, double[] memo) {

        if (years == 0)

            return initialAmount;

        if (memo[years] != 0)

            return memo[years];

        memo[years] = predictWithMemo(initialAmount, growthRate, years - 1, memo) \* (1 + growthRate);

        return memo[years];

    }

    public static void main(String[] args) {

        double initialAmount = 10000;

        double growthRate = 0.05; // 5%

        int years = 5;

        // Plain recursion

        double futureValue = predictFutureValue(initialAmount, growthRate, years);

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

        // Optimized

        double[] memo = new double[years + 1];

        double futureValueMemo = predictWithMemo(initialAmount, growthRate, years, memo);

        System.out.println("Future Value (Memoized): " + futureValueMemo);

    }

}

**Output:**

**A close-up of a computer screen

AI-generated content may be incorrect.**

**Analysis**

**Time Complexity**

* **Naive recursion**: O(n)  
  Each call waits for the next, leading to a **linear chain of calls**.
* **Memoized version**: O(n)  
  Same number of calls, but avoids **redundant computations** by storing results.

**Why Optimization is Needed?**

* For **large values of n**, recursion can become slow and even cause **stack overflow**.
* Use **memoization** or convert to an **iterative solution** to optimize performance.