**Data structures and Algorithms**

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

**Explanation:**

**Understanding Asymptotic Notation**

**What is Big O Notation?**

Big O notation is a mathematical tool used in computer science to describe the upper bound (worst-case scenario) of an algorithm's time or space complexity as the input size (n) grows. It helps us abstract away constants and less significant terms, focusing on how the algorithm scales with larger datasets. For example, if an algorithm takes 4n^2 + 2n + 7 steps, Big O simplifies this to O(n^2), since the n^2 term dominates as n increases.

**Why is Big O Important?**

* **Compares Efficiency:** Lets you compare different algorithms for the same task and choose the most scalable one.
* **Predicts Performance:** Helps predict how an algorithm will behave as data size increases, which is crucial for e-commerce platforms dealing with large product catalogs.
* **Guides Optimization:** Highlights which parts of your code may need improvement for better performance.

**Big O Notation Examples**

* **O(1):** Constant time (e.g., accessing an array element by index)
* **O(log n):** Logarithmic time (e.g., binary search in a sorted array)
* **O(n):** Linear time (e.g., linear search through an array)
* **O(n^2):** Quadratic time (e.g., bubble sort)[.](https://www.simplilearn.com/big-o-notation-in-data-structure-article)

**Best, Average, and Worst-Case Scenarios in Search Operations**

When analyzing search algorithms, we consider three scenarios:

|  |  |  |  |
| --- | --- | --- | --- |
| **Scenario** | **Description** | **Example (Linear Search)** | **Big O Notation** |
| **Best-case** | The minimum time the algorithm can take | Target is first item | O(1) |
| **Average-case** | The expected time over all possible positions of the target | Target is in the middle | O(n) |
| **Worst-case** | The maximum time the algorithm can take (often used for Big O analysis) | Target is last or absent | O(n) |

* **Best-case:** The algorithm finds the target immediately (e.g., first item in a list). For linear search, this is O(1).
* **Average-case:** The algorithm finds the target somewhere in the middle. For linear search, this is O(n).
* **Worst-case:** The algorithm searches the entire dataset (target is last or not present). For linear search, this is O(n).

For **binary search** (on a sorted array):

* **Best-case:** Target is at the middle index, found on the first try (O(1)).
* **Average-case/Worst-case:** Each step halves the search space, so it takes O(log n) time.

Big O notation provides a standardized way to describe and compare the efficiency of algorithms, especially as input sizes grow. By focusing on best, average, and worst-case scenarios, you can analyze and predict the performance of search operations, ensuring your e-commerce platform remains fast and scalable[.](https://www.simplilearn.com/big-o-notation-in-data-structure-article)

**Code :**

import java.util.\*;

public class ECommerceSearch {

    // Product class definition

    static 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;

        }

        @Override

        public String toString() {

            return "[" + productId + "] " + productName + " (" + category + ")";

        }

    }

    // Linear search implementation

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

        for (int i = 0; i < products.length; i++) {

            if (products[i].getProductId() == targetId) {

                return i;

            }

        }

        return -1;

    }

    // Binary search implementation

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

        int low = 0, high = products.length - 1;

        while (low <= high) {

            int mid = low + (high - low) / 2;

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

            if (midId == targetId) {

                return mid;

            } else if (midId < targetId) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return -1;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // User input for products

        System.out.print("Enter number of products: ");

        int n = sc.nextInt();

        sc.nextLine();

        Product[] products = new Product[n];

        for (int i = 0; i < n; i++) {

            System.out.println("Enter details for product " + (i + 1) + ":");

            System.out.print("Product ID: ");

            int id = sc.nextInt();

            sc.nextLine();

            System.out.print("Product Name: ");

            String name = sc.nextLine();

            System.out.print("Category: ");

            String category = sc.nextLine();

            products[i] = new Product(id, name, category);

        }

        // User input for search

        System.out.print("\nEnter Product ID to search: ");

        int searchId = sc.nextInt();

        // Linear search

        int idxLinear = linearSearch(products, searchId);

        if (idxLinear != -1) {

            System.out.println("Linear Search: Product found at index " + idxLinear + ": " + products[idxLinear]);

        } else {

            System.out.println("Linear Search: Product not found.");

        }

        // Sort products by productId for binary search

        Arrays.sort(products, Comparator.comparingInt(Product::getProductId));

        // Binary search

        int idxBinary = binarySearch(products, searchId);

        if (idxBinary != -1) {

            System.out.println("Binary Search: Product found at index " + idxBinary + ": " + products[idxBinary]);

        } else {

            System.out.println("Binary Search: Product not found.");

        }

        sc.close();

    }

}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Exercise 7: Financial Forecasting**

**Explanation :**

**Understanding Recursive Algorithms**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It is particularly useful for problems that can be broken down into similar subproblems, allowing for elegant and concise solutions. A recursive function typically has:

* **Base Case:** The simplest scenario, which is solved directly and terminates the recursion.
* **Recursive Case:** The function calls itself with modified parameters, moving closer to the base case with each call.

For example, in calculating the factorial of a number n, the base case is 0!=1, and the recursive case is n!=n×(n−1)!.

**Code:**

import java.util.Scanner;

public class FinancialForecast {

    // Recursive method to calculate future value

    public static double forecastValue(int year, double initialValue, double growthRate) {

        if (year == 0) {

            return initialValue;

        } else {

            return forecastValue(year - 1, initialValue, growthRate) \* (1 + growthRate / 100);

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // User input

        System.out.print("Enter initial value (e.g., current profit or revenue): ");

        double initialValue = sc.nextDouble();

        System.out.print("Enter annual growth rate (%): ");

        double growthRate = sc.nextDouble();

        System.out.print("Enter number of years to forecast: ");

        int years = sc.nextInt();

        // Forecast calculation

        double result = forecastValue(years, initialValue, growthRate);

        System.out.printf("Forecasted value after %d years: %.2f\n", years, result);

        sc.close();

    }

}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Analysis**

**Time Complexity:**

* The **time complexity** is **O(n)** because each recursive call decreases year by 1 until it hits 0.
* For n years, it performs n multiplications.

**Optimization:**

* Recursive algorithms can be inefficient if they recompute the same values multiple times. In this case, since each call only depends on the previous period, the overhead is minimal. However, for more complex recursive relations (e.g., Fibonacci sequence), memoization or converting to an iterative approach can significantly reduce computation time.
* **Memoization** or **Iterative approach** can optimize it.

**Iterative version (alternative):**

public static double forecastValueIterative(int years, double initialValue, double growthRate) {

double value = initialValue;

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

value \*= (1 + growthRate / 100);

}

return value;

}