**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.

**Step -** **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.

**1.1 Big O Notation**

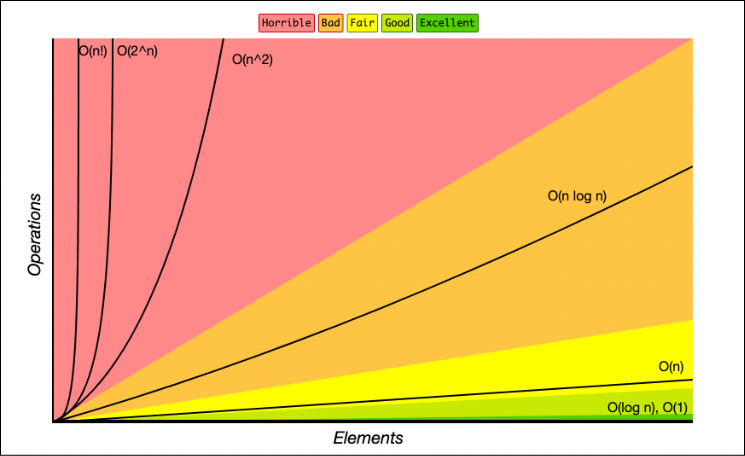
Big O notation describes the upper bound of an algorithm's running time in terms of input size n. It shows how well an algorithm scales.

**Why is it Useful?**

Big O notation helps us:

* Compare the efficiency of different algorithms.
* Predict the scalability of an algorithm as data grows.
* Choose the most optimalalgorithm for a given problem.
* Identify potential bottlenecks in time or memory usage.

The following graph illustrates Big O complexity:



**How It Analyzes Algorithms**

Suppose you're comparing two search algorithms:

* **Linear Search**: O(n)
* **Binary Search**: O(log n)

If you search through 1,000,000 elements:

* Linear Search might take up to 1,000,000 comparisons (in worst case).
* Binary Search will take about 20 comparisons(in worst case) (since log₂(1,000,000) ≈ 20).

Big O helps you choose the faster, more scalable approach by analyzing growth.

**1.2 Describe the best, average, and worst-case scenarios for search operations.**

**1.2.1. Linear Search**

How It Works:

* Traverse each element from start to end until the target is found or the list ends.

Best Case

* Scenario: The target element is the first element in the array.
* Time Complexity: O(1) (only 1 comparison needed)

Average Case

* Scenario: The target element is somewhere in the middle of the array.
* Time Complexity: O(n/2) → simplifies to O(n)  
  (On average, n/2 comparisons are required)

Worst Case

* Scenario 1: The element is at the last position.
* Scenario 2: The element is not present.
* Time Complexity: O(n)  
  (You must check every element)

**1.2.2. Binary Search**

How It Works:

* Repeatedly divide the sorted array into halves and compare the middle element.

Best Case

* Scenario: The target element is exactly at the middle index on the first try.
* Time Complexity: O(1)

Average Case

* Scenario: The target element is located somewhere not at the edges or center, requiring multiple splits.
* Time Complexity: O(log n)  
  (Log base 2 because the array is halved each time)

Worst Case

* Scenario 1: The element is not present after full log-based divisions.
* Scenario 2: Found only after maximum number of splits.
* Time Complexity: O(log n)

**Step – 2. Setup**

Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

**Product.java**

package Week1\_DataStructuresandAlgorithms.Handson2\_EcommercePlatformSearchFunction.Code;

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 Product() {

        // Default constructor

    }

    // Setters for productId, productName, and category

    public void setProductId(int productId) {

        this.productId = productId;

    }

    public void setProductName(String productName) {

        this.productName = productName;

    }

    public void setCategory(String category) {

        this.category = category;

    }

    // Getters for productId, productName, and category

    public int getProductId() {

        return productId;

    }

    public String getProductName() {

        return productName;

    }

    public String getCategory() {

        return category;

    }

}

**Step – 3. Implementation**

Implement linear search and binary search algorithms.

Store products in an array for linear search and a sorted array for binary search.

**3.1 Linear Search (Unsorted Array / Sorted Array)**

**LinearSearch.java**

package Week1\_DataStructuresandAlgorithms.Handson2\_EcommercePlatformSearchFunction.Code;

import java.util.List;

public class LinearSearch {

    public static Product search(List<Product> products, String targetName) {

        for (Product product : products) {

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

                return product;

            }

        }

        return null;

    }

}

**3.2** **Binary Search (Sorted Array by Name) (works only on sorted things)**

**BinarySearch.java**

package Week1\_DataStructuresandAlgorithms.Handson2\_EcommercePlatformSearchFunction.Code;

import java.util.ArrayList;

import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<Product> products = new ArrayList<>();

        products.add(new Product(101, "Laptop", "Electronics"));

        products.add(new Product(102, "Tablet", "Electronics"));

        Product p1 = new Product();

        p1.setProductId(103);

        p1.setProductName("Phone");

        p1.setCategory("Electronics");

        products.add(p1);

        Product p2 = new Product();

        p2.setProductId(104);

        p2.setProductName("Smartwatch");

        p2.setCategory("Electronics");

        products.add(p2);

        products.add(new Product(105, "Headphones", "Accessories"));

        String target = "Phone";

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

        Product result1 = LinearSearch.search(products, target);

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

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

        Product result2 = BinarySearch.search(products, target);

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

    }

}

**3.3 Logger.java**

package Week1\_DataStructuresandAlgorithms.Handson2\_EcommercePlatformSearchFunction.Code;

public class Logger {

    private static Logger instance;

    private Logger() {

        // private constructor to prevent instantiation

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void logProduct(Product product) {

        System.out.println("Product Found:");

        System.out.println("ID       : " + product.getProductId());

        System.out.println("Name     : " + product.getProductName());

        System.out.println("Category : " + product.getCategory());

    }

}

**3.4 Main.java**

package Week1\_DataStructuresandAlgorithms.Handson2\_EcommercePlatformSearchFunction.Code;

import java.util.ArrayList;

import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<Product> products = new ArrayList<>();

        products.add(new Product(101, "Laptop", "Electronics"));

        products.add(new Product(102, "Tablet", "Electronics"));

        Product p1 = new Product();

        p1.setProductId(103);

        p1.setProductName("Phone");

        p1.setCategory("Electronics");

        products.add(p1);

        Product p2 = new Product();

        p2.setProductId(104);

        p2.setProductName("Smartwatch");

        p2.setCategory("Electronics");

        products.add(p2);

        products.add(new Product(105, "Headphones", "Accessories"));

        String target = "Phone";

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

        Product result1 = LinearSearch.search(products, target);

        if (result1 != null) {

            Logger.getInstance().logProduct(result1);

        } else {

            System.out.println("Product not found");

        }

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

        Product result2 = BinarySearch.search(products, target);

        if (result2 != null) {

            Logger.getInstance().logProduct(result2);

        } else {

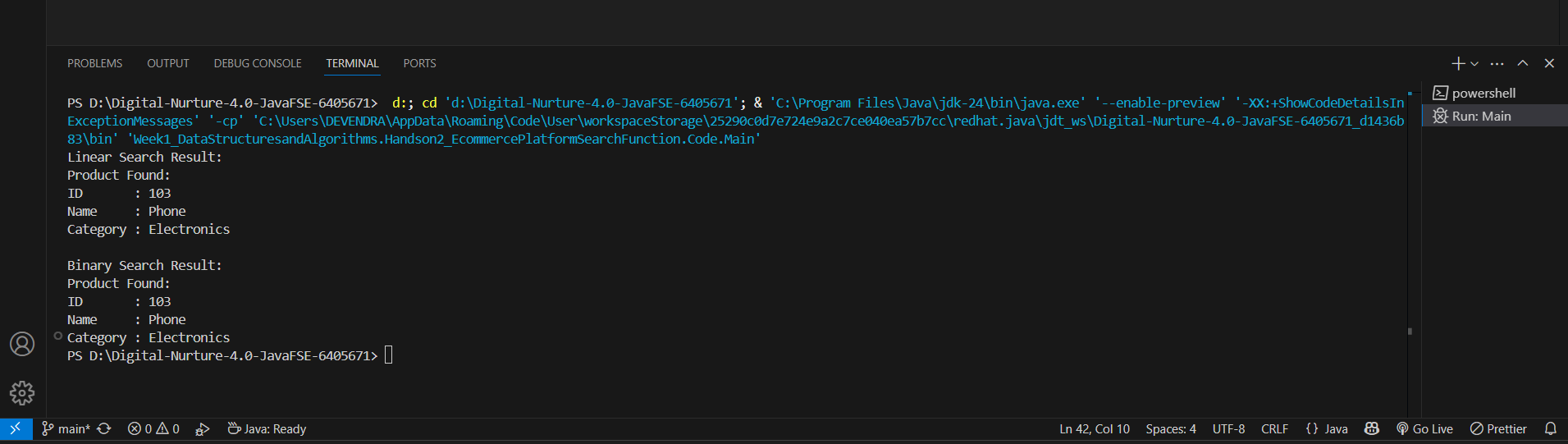
            System.out.println("Product not found");

        }

    }

}

**Output:**

****

**Step – 4. Analysis:**

Compare the time complexity of linear and binary search algorithms.

Discuss which algorithm is more suitable for your platform and why**.**

**4.1 Compare the time complexity of linear and binary search algorithms.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Aspect** | |  | | --- | | **Linear Search** |  |  | | --- | |  | | | **Binary Search** | | --- |  |  | | --- | |  | |
| |  | | --- | | **Time Complexity (Best)** |  |  | | --- | |  | | |  | | --- | | O(1) — target at the beginning |  |  | | --- | |  | | |  | | --- | | O(1) — target is the middle element |  |  | | --- | |  | |
| |  | | --- | | **Time Complexity (Average)** |  |  | | --- | |  | | |  | | --- | | O(n) — target anywhere in array |  |  | | --- | |  | | |  | | --- | | **O(log n) — divide and** search **recursively** |  |  | | --- | |  | |
| |  | | --- | | **Time Complexity (Worst)** |  |  | | --- | |  | | |  | | --- | | O(n) — target not found |  |  | | --- | |  | | |  | | --- | | O(log n) — target not found after full division |  |  | | --- | |  | |
| |  | | --- | | **Input Requirement** |  |  | | --- | |  | | |  | | --- | | Unsorted or sorted array |  |  | | --- | |  | | |  | | --- | | Sorted array only |  |  | | --- | |  | |
| |  | | --- | | **Efficiency (Large n)** |  |  | | --- | |  | | |  | | --- | | Slower |  |  | | --- | |  | | Much faster |
| |  | | --- | | **Scalability** |  |  | | --- | |  | | |  | | --- | | Poor for largedatasets |  |  | | --- | |  | | Excellent for large datasets |

**4.2 Discuss which algorithm is more suitable for your platform and why.**

**Best Fit for E-commerce Platform: Binary Search**

**Reasons:**

Requires pre-sorted data.

Sorting takes time (O(n log n)), but it's often a one-time cost or done during insertion

1. **Large Product Catalog**
   * E-commerce platforms have thousands or millions of products.
   * Binary Search drastically reduces search time (logarithmic scale).
2. **Sorted Listings**
   * Products are often sorted by name, price, or category — matching Binary Search's needs.
3. **User Experience**
   * Faster search results mean better responsiveness and customer satisfaction.
4. **Scalability**
   * As more products are added, Binary Search maintains high performance.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Step – 1. Understand Recursive Algorithms:**

Explain the concept of recursion and how it can simplify certain problems.

**Recursion** is a programming technique where a method calls itself to solve a smaller instance of the same problem. It simplifies problems that have a repetitive or self-similar structure.

**Example**: Calculating the *future value* of an investment recursively, based on a fixed growth rate.

**Step – 2. Setup:**

Create a method to calculate the future value using a recursive approach.

**Recursive Future Value Calculation:**

Let’s assume:

* Initial amount: A
* Growth rate: r (e.g., 5% → 0.05)
* Number of years: n

The future value after n years is:

FV (n) = A . (1 + r)^n

Can be broke down recursively:

FV (n) = (1 + r) . FV(n – 1)

**Recursive Function:**

public static double FV(double amount, double rate, int years) {

// Base case: if no years left, return the amount

if (years == 0) {

return amount;

}

// Recursive case: apply the growth rate and call recursively for one less year

return (1 + rate) \* FV(amount, rate, years - 1);

}

**Step – 3. Implementation:**

Implement a recursive algorithm to predict future values based on past growth rates.

package Week1\_DataStructuresandAlgorithms.Handson7\_FinancialForecasting.Code;

**Recursive code:**

public class FinancialForecastingRecEx {

    public double futureValue(double amount, double rate, int years) {

        if (years == 0) {

            return amount;

        }

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

    }

}

**Iterative code:**package Week1\_DataStructuresandAlgorithms.Handson7\_FinancialForecasting.Code;

public class FinancialForecastingIterativeEx {

    public double futureValueIterative(double amount, double rate, int years) {

    double result = amount;

    for (int i = 0; i < years; i++) {

        result \*= (1 + rate);

    }

    return result;

}

}

**Logger:**

package Week1\_DataStructuresandAlgorithms.Handson7\_FinancialForecasting.Code;

import Week1\_DataStructuresandAlgorithms.Handson2\_EcommercePlatformSearchFunction.Code.Product;

public class Logger {

    private static Logger instance;

    private Logger() {

        // private constructor to prevent instantiation

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void logProduct(Product product) {

        System.out.println("Product Found:");

        System.out.println("ID       : " + product.getProductId());

        System.out.println("Name     : " + product.getProductName());

        System.out.println("Category : " + product.getCategory());

    }

    public void log(String message) {

        System.out.println(message);

    }

}

**Main:**

package Week1\_DataStructuresandAlgorithms.Handson7\_FinancialForecasting.Code;

public class Main {

public static void main(String[] args) {

        FinancialForecastingRecEx recObj = new FinancialForecastingRecEx();

        FinancialForecastingIterativeEx iterObj = new FinancialForecastingIterativeEx();

        double initialAmount = 10000;

        double annualGrowthRate = 0.05;

        int numberOfYears = 5;

        double futureValRec = recObj.futureValue(initialAmount, annualGrowthRate, numberOfYears);

        double futureValIter = iterObj.futureValueIterative(initialAmount, annualGrowthRate, numberOfYears);

        Logger logger = Logger.getInstance();

        logger.log("Future Value Calculation using Recursion:");

        logger.log(String.format("Future Value after %d years: %.2f", numberOfYears, futureValRec));

        logger.log("Future Value Calculation using Iteration:");

        logger.log(String.format("Future Value after %d years: %.2f", numberOfYears, futureValIter));

        logger.log("Both methods yield the same result: " + (futureValRec == futureValIter));

    }

}

Output:



**Step - 4. Analysis**

Discuss the time complexity of your recursive algorithm.

Explain how to optimize the recursive solution to avoid excessive computation.

**Algorithm Complexity discussion:**

The recursive function:

futureValue(amount, rate, years)

calls itself exactly once per recursive step until years == 0.

**Time Complexity: O(n)**

* Each recursive call reduces years by 1.
* A total of n recursive calls are made (where n = number of years).
* No overlapping subproblems → each step does a constant-time calculation and one recursive call.

**Space Complexity: O(n)**

* Each recursive call adds a new frame to the call stack.
* So, for n years, the maximum depth of the stack is n.

**Problem: Excessive Computation in Recursion**

Though this recursion is simple and linear, it becomes inefficient when:

* n (years) is very large.
* Stack overflow risk due to deep recursion (especially in languages with small call stack limits like Java).
* Recursion overhead: pushing/popping frames repeatedly.

**Optimization Techniques**

**1. Convert to Iterative Solution**

Instead of recursive calls, use a for loop:

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

double result = amount;

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

result \*= (1 + rate);

}

return result;

}

**Time Complexity**: O(n)

**Space Complexity**: O(1)

No stack overhead, faster, safer for large n.

**2.Use Exponentiation (Optimized Mathematical Approach)**

Since:

FV(n) = amount . (1 + rate)^n

Use Math.pow():

public static double futureValueMath(double amount, double rate, int years) {

return amount \* Math.pow(1 + rate, years);

}

**Time Complexity**: O(1)

Best performance and precision (built-in optimized exponentiation).

Comparison:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Approach** | |  | | --- | | **Time Complexity** |  |  | | --- | |  | | |  | | --- | | **Space Complexity** |  |  | | --- | |  | | |  | | --- | | **Risk of Stack Overflow** |  |  | | --- | |  | | | **Efficiency** | | --- |  |  | | --- | |  | |
| |  | | --- | | Recursive |  |  | | --- | |  | | |  | | --- | | O(n) |  |  | | --- | |  | | |  | | --- | | O(n) |  |  | | --- | |  | | Yes | Moderate |
| |  | | --- | | Iterative |  |  | | --- | |  | | |  | | --- | | O(n) |  |  | | --- | |  | | |  | | --- | | O(1) |  |  | | --- | |  | | No | |  | | --- | | Better |  |  | | --- | |  | |
| |  | | --- | | Using Buit in func: Math.pow() |  |  | | --- | |  | | |  | | --- | | O(1) |  |  | | --- | |  | | |  | | --- | | O(1) |  |  | | --- | |  | | |  | | --- | | No |  |  | | --- | |  | | Best |

**Optimizing the Recursive Solution:**

**Recursive solutions can be inefficient due to:**

* Repeated function calls
* Stack overflow risk for large n

**Optimized Approaches:**

**a. Memoization**

* Store already computed values in a map or array.
* Useful if the problem has overlapping subproblems (not strictly needed here since every input is unique).

**b. Convert to Iteration (Tail Recursion / Loop)**

* As above iterative example given.