**Algorithms\_Data Structures (Mandatory Hands On)**

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

**Big O Notation** is a mathematical way to describe the **time or space complexity** of an algorithm in terms of input size n. It focuses on the **growth rate** as input increases, helping us analyze and compare algorithms for performance.

| Complexity | Name | Example Use Case |
| --- | --- | --- |
| O(1) | Constant time | Accessing an array element by index |
| O(n) | Linear time | Linear search in an array |
| O(log n) | Logarithmic time | Binary search on a sorted array |
| O(n²) | Quadratic time | Bubble sort, nested loops on array |

| Search Type | 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) |

Code:

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**;**

    }

}

SearchUtil.java

*public* class SearchUtil {

*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 **-** left) **/** 2**;**

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

**return** products[mid]**;**

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

                left **=** mid **+** 1**;**

            } **else** {

                right **=** mid **-** 1**;**

            }

        }

**return** null**;**

    }

}

SearchTest.java

*import* java**.**util**.**Arrays**;**

*public* class SearchTest {

*public* *static* *void* main(String[] **args**) {

        Product[] products **=** {

**new** Product(102**,** "Keyboard"**,** "Electronics")**,**

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

**new** Product(103**,** "Laptop"**,** "Electronics")**,**

**new** Product(104**,** "Chair"**,** "Furniture")

        }**;**

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

        Product found1 **=** SearchUtil**.**linearSearch(products**,** 103)**;**

        System**.***out***.**println(found1 **!=** null **?** found1 **:** "Product not found")**;**

        Arrays**.**sort(products**,** (a**,** b) **->** a**.***productId* **-** b**.***productId*)**;**

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

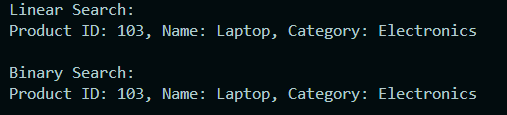
        Product found2 **=** SearchUtil**.**binarySearch(products**,** 103)**;**

        System**.***out***.**println(found2 **!=** null **?** found2 **:** "Product not found")**;**

    }

}

Output:



| Algorithm | Time Complexity | Sorted Required | Suitability |
| --- | --- | --- | --- |
| Linear Search | O(n) | No | Simple, but slower for large data |
| Binary Search | O(log n) | Yes | Much faster, efficient, but requires sorting |

**Exercise 7: Financial Forecasting**

Recursion is a programming technique where a function **calls itself** to solve smaller instances of the same problem. It simplifies problems that have a **repeating pattern or structure**, like factorial, Fibonacci, or financial forecasting with compound interest.

FV=PV×(1+r)^n

Where:

* FV = future value
* PV = present value
* r = annual growth rate
* n = number of years

Code:

*public* class FinancialForecast {

*public* *static* *double* calculateFutureValue(*double* **presentValue,** *double* **rate,** *int* **years**) {

**if** (years **==** 0) {

**return** presentValue**;**

        }

**return** calculateFutureValue(presentValue**,** rate**,** years **-** 1) **\*** (1 **+** rate)**;**

    }

*public* *static* *void* main(String[] **args**) {

*double* presentValue **=** 10000**;**

*double* rate **=** 0.05**;**

*int* years **=** 10**;**

*double* futureValue **=** calculateFutureValue(presentValue**,** rate**,** years)**;**

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

    }

}

Output:



Time Complexity

The recursive function has O(n) time complexity, where n is the number of years.

Each recursive call reduces years by 1 until it hits 0.

Although this example is simple and linear, if you extend this to more complex financial models (like recursive Fibonacci-based investment growth), use memoization or convert to iteration to avoid exponential time complexity.

*public* *static* *double* calculateFutureValueIterative(*double* presentValue**,** *double* rate**,** *int* years) {

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

        presentValue **\*=** (1 **+** rate)**;**

    }

**return** presentValue**;**

}