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

**Steps:**

**1. Understand Asymptotic Notation:**

* Explain Big O notation and how it helps in analyzing algorithms.

**Big O Notation**

Big O notation describes the **upper bound** of an algorithm's running time or space requirement in terms of the input size n. It helps developers understand how algorithms scale and compare their efficiency.

|  |  |  |
| --- | --- | --- |
| **Notation** | **Description** | **Example** |
| O(1) | Constant time | Accessing array element |
| O(log n) | Logarithmic time | Binary Search |
| O(n) | Linear time | Linear Search |
| O(n²) | Quadratic time | Nested loops |

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

**Best, Average, and Worst-Case in Searching:**

|  |  |  |
| --- | --- | --- |
| **Case** | **Linear Search** | **Binary Search** |
| Best Case | O(1) – First match | O(1) – Middle element match |
| Average Case | O(n/2) = O(n) | O(log n) |
| Worst Case | O(n) – Last element | O(log n) |

**2. Setup:**

**Output:**

Product{ID=3, Name='Shoes', Category='Footwear'}

**3. Implementation:**

**Output:**

Linear Search Result: Product{ID=3, Name='Tablet', Category='Electronics'}

Binary Search Result: Product{ID=3, Name='Tablet', Category='Electronics'}

**4. Analysis**

**Time Complexity Comparison:**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **When to Use** |
| Linear Search | O(n) | Unsorted or small dataset |
| Binary Search | O(log n) | Sorted large dataset |

**Which is More Suitable?**

* Binary Search is more efficient for large and sorted datasets, making it ideal for frequently accessed product catalogs in an e-commerce platform.
* Linear Search is easier to implement and better for small or unsorted datasets, or when only a few searches are performed.

**Exercise 7: Financial Forecasting:**

**1. Understand Recursive Algorithms:**

**🔹 What is Recursion?**

Recursion is a programming technique where a method calls itself to solve smaller instances of a problem.

**🔹 How It Helps**

Recursion simplifies problems that have repetitive sub-structure, such as:

* Fibonacci series
* Tree traversals
* Financial predictions with compound patterns

**2. Setup:**

We will define a method to recursively calculate the future value based on:

* P = initial amount
* r = growth rate (as a decimal, e.g., 0.05 for 5%)
* n = number of years

📌 Formula:

FV = P × (1 + r)^n

**3. Implementation:**

Future value after 5 years: ₹12762.82

**4. Analysis:**

**Time Complexity:**

* The recursive method makes one call per year.
* So, Time Complexity = O(n) where n = number of years.

**1. Use Iteration Instead**

Recursion adds overhead due to function calls and stack usage. You can convert to a loop:

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

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

principal \*= (1 + rate);

}

return principal;

}

**2. Use Memorization (if overlapping subproblems exist)**

For more complex financial models (e.g., multi-factor forecasting), **store previously computed values** to avoid redundant calls.