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

**Big O Notation:** Big O notation is a mathematical notation used to describe the time or space complexity of an algorithm in terms of input size (n). It helps us estimate how the performance of an algorithm scales as the input grows.

**Common Big O complexities**:

* O(1) – Constant time (e.g., accessing an array element)
* O(n) – Linear time (e.g., linear search)
* O(log n) – Logarithmic time (e.g., binary search)
* O(n²) – Quadratic time (e.g., bubble sort)

**the Best, Average, and Worst-Case Scenarios for Search Operations**

In algorithm analysis, the performance of a search operation can vary depending on the location of the target element and the nature of the data. These variations are captured in **three scenarios**: best case, average case, and worst case.

**🔹 1. Best-Case Scenario:**

The **best-case scenario** represents the situation where the search algorithm finds the desired element at the **earliest possible position** in the dataset.

* **Linear Search**: The best case occurs when the element is at the **first position** of the array or list.  
  **Time Complexity**: O(1) – only one comparison is needed.
* **Binary Search**: The best case happens when the element is found at the **middle index** on the very first comparison.  
  **Time Complexity**: O(1) as well.

**🔹 2. Average-Case Scenario:**

The **average-case scenario** refers to the situation where the element could be **anywhere** in the dataset, and all positions are **equally likely**.

* **Linear Search**: On average, the element is found after searching **n/2** elements.  
  **Time Complexity**: O(n)
* **Binary Search**: The average number of steps is **log₂(n)** since it divides the dataset in half at each step.  
  **Time Complexity**: O(log n)

**🔹 3. Worst-Case Scenario:**

The **worst-case scenario** occurs when the element is either at the **last position** or is **not present at all**, requiring the algorithm to do the **maximum amount of work**.

* **Linear Search**: The algorithm must check **every single element** in the list.  
  **Time Complexity**: O(n)
* **Binary Search**: The algorithm performs the **maximum number of halving steps** before concluding the element is not found.  
  **Time Complexity**: 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;

}

@Override

**public** String toString() {

**return** "[" + productId + ", " + productName + ", " + category + "]";

}

}

SearchUtils.java

**public** **class** SearchUtils {

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

**for** (Product product : products) {

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

**return** product;

}

}

**return** **null**;

}

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

**int** low = 0;

**int** high = products.length - 1;

**while** (low <= high) {

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

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

**if** (comparison == 0) **return** products[mid];

**else** **if** (comparison < 0) low = mid + 1;

**else** high = mid - 1;

}

**return** **null**;

}

}

Main.java

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** Main {

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

Product[] products = {

**new** Product(101, "Laptop", "Electronics"),

**new** Product(102, "Shoes", "Fashion"),

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

**new** Product(104, "Tablet", "Electronics"),

**new** Product(105, "Watch", "Accessories")

};

Product result1 = SearchUtils.*linearSearch*(products, "Phone");

System.***out***.println("Linear Search Result: " + result1);

// Sort before Binary Search

Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName.toLowerCase()));

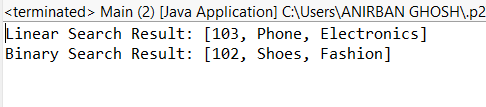
Product result2 = SearchUtils.*binarySearch*(products, "Shoes");

System.***out***.println("Binary Search Result: " + result2);

}

}

**Output:**



**Time Complexity of Linear and Binary Search Algorithms**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |  |
| --- | --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) |
| **Binary Search** | O(1) | O(log n) | O(log n) |  |

**Which Algorithm Is More Suitable and Why?**

For an **e-commerce platform**, where search performance is critical **Binary Search** is more suitable because The product list is **pre-sorted** or indexed for performing **frequent searches** on **large datasets** where Speed and scalability are important.

**Exercise 7: Financial Forecasting**

**Recursion:** Recursion is a programming technique where a method calls itself to solve smaller instances of the same problem.Recursion simplifies problems that have repeating patterns or can be broken into smaller subproblems.

Examples include factorial, Fibonacci, and in this case — forecasting future values based on past trends**.**

Assume we want to forecast the value of an investment given:

A starting value

A fixed growth rate

A number of periods (years, months, etc.)

**Code:**

Forecast.java

**public** **class** Forecast {

// Recursive method to calculate future value

**public** **static** **double** predictFutureValue(**double** currentValue, **double** growthRate, **int** periods) {

**if** (periods == 0) {

**return** currentValue;

}

**return** *predictFutureValue*(currentValue \* (1 + growthRate), growthRate, periods - 1);

//formula

}

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

**double** initialInvestment = 10000;

**double** annualGrowthRate = 0.05;

**int** years = 5;

**double** futureValue = *predictFutureValue*(initialInvestment, annualGrowthRate, years);

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

}

}

**Output:**



**Time Complexity:**

The recursive function makes one call for each period.

So the time complexity is: **O(n)**, where n = number of periods

**How to Optimize the Recursive Solution to Avoid Excessive Computation**

Recursive solutions, while simple and elegant, can become inefficient due to repeated calculations and high memory usage from deep recursion calls. To optimize such solutions, we can use the following techniques:

**1. Memoization**: Store the results of already computed recursive calls in a data structure (like an array or map). Avoids recalculating the same values, thus improving performance significantly.

**2. Convert to Iterative Approach:** Replace recursion with loops (for or while) to reduce call stack overhead. This approach uses less memory and is generally faster. Ideal for simple linear recursions, such as calculating compound growth.

**3. Tail Recursion:** Optimize the recursive function to use **tail recursion** (where the recursive call is the last operation). Some compilers/VMs can optimize tail-recursive functions into iterations internally.