**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.

**Steps:**

1. **Understand Asymptotic Notation:**
   1. Explain Big O notation and how it helps in analyzing algorithms.
   2. Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   1. Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   1. Implement linear search and binary search algorithms.
   2. Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   1. Compare the time complexity of linear and binary search algorithms.
   2. Discuss which algorithm is more suitable for your platform and why.

**Understand Asymptotic Notation**

**Big O Notation**

**Big O notation** describes the performance (time or space complexity) of an algorithm in terms of input size n.

* **O(1):** Constant time – doesn’t depend on input size.
* **O(n):** Linear time – performance grows proportionally with input.
* **O(log n):** Logarithmic – performance increases slowly as input size increases.
* **O(n log n):** Linearithmic – slightly slower than linear, often in sorting.
* **O(n²):** Quadratic – performance worsens quickly with more data.

**1. Linear Search**

In **Linear Search**, we scan the array **sequentially** from the beginning.

**Time Complexity: O(n)**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Description** | **Time Complexity** |
| **Best Case** | Target is the **first** element in the array. | O(1) |
| **Average Case** | Target is found around the **middle** of the array. | O(n/2) → O(n) |
| **Worst Case** | Target is at the **last** index or **not present at all**. | O(n) |

**2. Binary Search**

In **Binary Search**, the array **must be sorted**, and we repeatedly divide the search interval in half.

**Time Complexity: O(log n)**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Description** | **Time Complexity** |
| **Best Case** | Target is found at the **middle** on the first check. | O(1) |
| **Average Case** | Target is somewhere in the array; takes log₂(n) steps to find. | O(log n) |
| **Worst Case** | Target is **not present** or found after log₂(n) divisions. | O(log n) |

**Comparison Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **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) |

**Code**

Product.java

**package** com;

**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;

}

}

SearchFunctions.java

**package** com;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** SearchFunctions {

// Linear Search: No need to sort

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

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

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

**return** product;

}

}

**return** **null**;

}

// Binary Search: Requires sorted array by product name

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

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

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

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

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

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

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

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

**else** high = mid - 1;

}

**return** **null**;

}

}

ECommerceSearchDemo.java

**package** com;

**public** **class** ECommerceSearchDemo {

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

Product[] products = {

**new** Product(1, "Laptop", "Electronics"),

**new** Product(2, "Shoes", "Fashion"),

**new** Product(3, "Watch", "Accessories"),

**new** Product(4, "Camera", "Electronics"),

**new** Product(5, "Book", "Stationery")

};

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

Product result1 = SearchFunctions.*linearSearch*(products, "Camera");

System.***out***.println(result1 != **null** ? "Found: " + result1 : "Not Found");

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

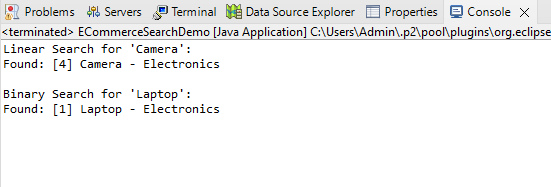
Product result2 = SearchFunctions.*binarySearch*(products, "Laptop");

System.***out***.println(result2 != **null** ? "Found: " + result2 : "Not Found");

}

}

**Output:**



|  |  |  |
| --- | --- | --- |
| **Feature** | **Linear Search** | **Binary Search** |
| Time Complexity | O(n) | O(log n) |
| Requirement | Unsorted array | Sorted array |
| Setup Time | None | O(n log n) for sorting |
| Use Case | Small or unsorted data | Large, sorted datasets |

**Conclusion**

For an **e-commerce platform**, where thousands or millions of products exist:

* Use **binary search** for high-speed performance.
* Maintain sorted arrays or use **binary search trees** or **hash maps** for even faster operations.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   1. Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   1. Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   1. Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   1. Discuss the time complexity of your recursive algorithm.
   2. Explain how to optimize the recursive solution to avoid excessive computation.

**What is Recursion?**

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

**Why Use Recursion?**

* Simplifies problems that have **repetitive sub-structure**, such as in:
  + Tree traversal
  + Factorials
  + Fibonacci numbers
  + Financial forecasting (e.g., compounding returns)

**Code:**

FinancialForecast.java

**package** com;

**public** **class** FinancialForecast {

// Recursive method to calculate future value

**public** **static** **double** forecastValueRecursive(**double** currentValue, **double** rate, **int** periods) {

// Base Case: no more periods to grow

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

**return** currentValue;

}

**return** *forecastValueRecursive*(currentValue \* (1 + rate), rate, periods - 1);

}

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

**double** initialValue = 1000.0;

**double** growthRate = 0.05;

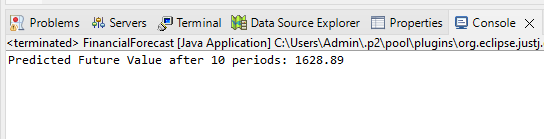
**int** futurePeriods = 10;

**double** futureValue = *forecastValueRecursive*(initialValue, growthRate, futurePeriods);

System.***out***.printf("Predicted Future Value after %d periods: %.2f\n", futurePeriods, futureValue);

}

}



**Analysis**

**Time Complexity**

* Each recursive call reduces periods by 1.
* So for n periods, we make n recursive calls.
* **Time Complexity**: O(n)

**Space Complexity**

* Each call adds to the call stack.
* **Space Complexity**: O(n) (due to recursion stack)

**Optimization: Avoiding Excessive Recursion**

Recursion can be **inefficient** for large n. Optimizations include:

**Iterative Approach (Tail Recursion Alternative)**

public static double forecastValueIterative(double currentValue, double rate, int periods) {

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

currentValue \*= (1 + rate);

}

return currentValue;

}

* Same result as recursion.
* **Time Complexity**: O(n)
* **Space Complexity**: O(1)

**Summary**

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Time Complexity** | **Space Complexity** | **Notes** |
| Recursive | O(n) | O(n) | Simple but less efficient |
| Iterative | O(n) | O(1) | Preferred in real-world |