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

**SOLUTION:**

**Big O Notation**

In programming, Big O describes how an algorithm's run time or space grows with the size of the input n.

|  |  |  |
| --- | --- | --- |
| **Case** | **Description** | **Example** |
| Best Case | Minimal time to search (e.g., first element match) | O(1) |
| Average Case | Average position match | O(n) for linear, O(log n) for binary |
| Worst Case | Last or no match | O(n) for linear, O(log n) for binary |

**Product.java**

**package** com.ecommerce.search;

**public** **class** Product {

**private** **int** productId;

**private** String productName;

**private** String category;

**public** Product(**int** productId, String productName, String category) {

**this**.productId = productId;

**this**.productName = productName;

**this**.category = category;

}

**public** **int** getProductId() {

**return** productId;

}

**public** String getProductName() {

**return** productName;

}

**public** String getCategory() {

**return** category;

}

@Override

**public** String toString() {

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

}

}

**SearchService.java**

**package** com.ecommerce.search;

**public** **class** SearchService {

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

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

**if** (p.getProductName().equalsIgnoreCase(targetName)) {

**return** p;

}

}

**return** **null**;

}

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

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

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

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

**int** compare = products[mid].getProductName().compareToIgnoreCase(targetName);

**if** (compare == 0) {

**return** products[mid];

} **else** **if** (compare < 0) {

low = mid + 1;

} **else** {

high = mid - 1;

}

}

**return** **null**;

}

}

**SearchTest.java**

**package** com.ecommerce.search;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** SearchTest {

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

Product[] products = {

**new** Product(11, "Sandals", "Footwear"),

**new** Product(12, "Frock", "Clothing"),

**new** Product(13, "Mouse", "Electronics"),

**new** Product(14, "TextBooks", "Education"),

**new** Product(15, "Keychain", "Accessories")

};

String target1 = "Mouse";

Product result1 = SearchService.*linearSearch*(products, target1);

System.***out***.println("Linear Search Result: " + (result1 != **null** ? result1 : "Product not found"));

Arrays.*sort*(products, Comparator.*comparing*(Product::getProductName));

String target2 = "Mouse";

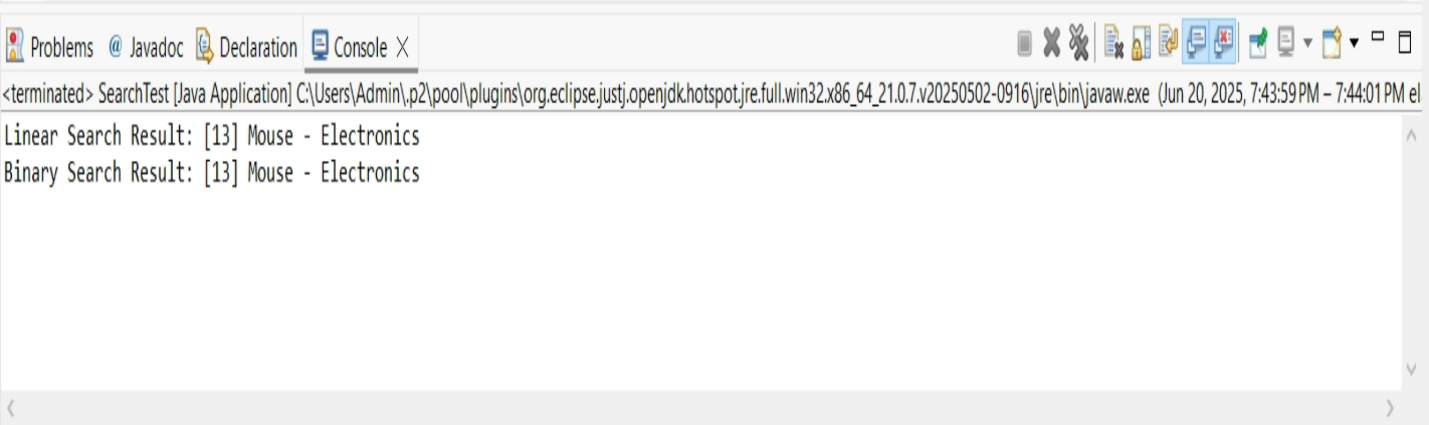
Product result2 = SearchService.*binarySearch*(products, target2);

System.***out***.println("Binary Search Result: " + (result2 != **null** ? result2 : "Product not found"));

}

}

**OUTPUT:**

****

### Comparing Time Complexity

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Linear Search** | **Binary Search** |
| **Time Complexity** | O(n) | O(log n) |
| **Best Case** | O(1) (first element match) | O(1) (middle element match) |
| **Average Case** | O(n/2) → O(n) | O(log n) |
| **Worst Case** | O(n) (last or no match) | O(log n) |
| **Data Requirement** | Works on unsorted data | Requires sorted data |
| **Implementation** | Simple | Slightly more complex |
| **Flexibility** | Good for small or dynamic collections | Best for large, static, sorted collections |

### B. Which Algorithm is More Suitable for Your Platform?

1. **Large Product Catalogs**: E-commerce platforms typically store thousands or millions of products. Binary search’s O(log n) time makes it faster and scalable.
2. **Sorted Indexes Are Common**: Products can be sorted by name, category, or price, enabling the use of binary search or tree-based structures.
3. **High Performance Is Critical**: Fast search leads to better user experience and conversion rates.

#### **When Linear Search Might Be Used**:

* If the dataset is **small**.
* If the data **is not sorted** and you want a **quick and simple implementation**.
* For **temporary lists** or **testing purposes**.

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

**SOLUTION:**

### Concept of Recursion:

Recursion is when a function **calls itself** to solve smaller subproblems. It's often useful for problems like:

* Factorial calculation
* Fibonacci series
* Tree/graph traversal
* Financial predictions with recurring calculations (like compound interest or growth modeling)

**Example :**  
To predict value in n years:  
**futureValue(n) = futureValue(n-1) × (1 + growthRate)**

**ForecastCalculator.java**

**package** com.financial.forecast;

**public** **class** ForecastCalculator {

**public** **static** **double** calculateFutureValue(**double** currentValue, **double** growthRate, **int** years) {

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

**return** currentValue;

}

**return** *calculateFutureValue*(currentValue \* (1 + growthRate), growthRate, years - 1);

}

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

**double** initialValue = 1000.0;

**double** growthRate = 0.08;

**int** years = 5;

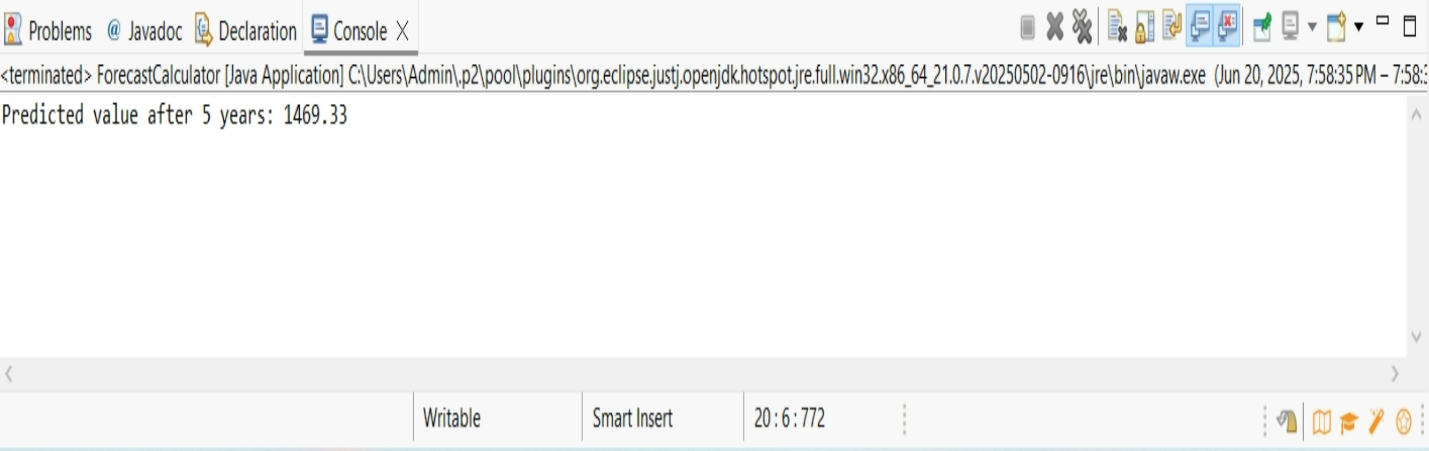
**double** predictedValue = *calculateFutureValue*(initialValue, growthRate, years);

System.***out***.printf("The Predicted value after %d years: %.2f%n", years, predictedValue);

}

}

**OUTPUT:**

****

### A. ****Time Complexity****

The recursive function calls itself **once per year**, so:

* **Time Complexity:** O(n), where n = number of years
* **Space Complexity:** O(n) (due to call stack in recursion)

### B. ****Optimization Discussion****

#### **Problem with Recursion:**

* For large n, recursion can cause **stack overflow** or unnecessary recomputation.
* Example: If we used recursion for a Fibonacci-based forecast, we’d face **exponential time** without memoization.

#### **Solution:**

1. **Use Memoization**: Store results of previous calculations.
2. **Use Iterative Approach** (preferred for long-term forecasting).