Algorithms Data Structures:

**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.

Explanation:

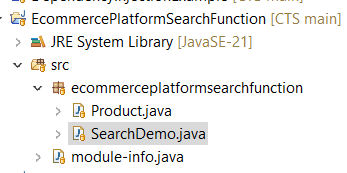
**1  |  Asymptotic notation in a nutshell**

**Big O (upper bound).** Describes how the *maximum* amount of work grows with input size *n*.  
**Big Ω (lower bound).** Describes a guaranteed *minimum* amount of work.  
**Θ (tight bound).** When an algorithm is both O(f(n)) and Ω(f(n)).

For a single search operation, we usually quote Big O:

| **Case** | **Linear search** | **Binary search (array must be sorted)** |
| --- | --- | --- |
| **Best** | **O(1)** – first element is the target | **O(1)** – middle element is the target |
| **Average** | O(n/2) ≈ **O(n)** | O(log n) |
| **Worst** | **O(n)** – target last / absent | **O(log n)** – target deepest leaf / absent |

2. SET UP:



3. IMPLEMENTATION:

**package** ecommerceplatformsearchfunction;

**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; }

**public** String toString() {

**return** "ID:" + productId + " Name:" + productName + " Cat:" + category;

}

}

package ecommerceplatformsearchfunction;

import java.util.\*;

import java.util.Arrays;

public class SearchDemo {

public static Product linearSearch(Product[] data, int id) {

for (Product p : data) if (p.getProductId() == id) return p;

return null;

}

public static Product binarySearch(Product[] sorted, int id) {

int lo = 0, hi = sorted.length - 1;

while (lo <= hi) {

int mid = (lo + hi) >>> 1;

int cmp = Integer.compare(sorted[mid].getProductId(), id);

if (cmp == 0) return sorted[mid];

if (cmp < 0) lo = mid + 1;

else hi = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] inventory = {

new Product(105, "Laptop", "Electronics"),

new Product(21, "Shoes", "Footwear"),

new Product(78, "Book", "Books"),

new Product(302, "Camera", "Electronics"),

new Product(4, "Watch", "Accessories")

};

Product result1 = linearSearch(inventory, 78);

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

Product[] sorted = Arrays.copyOf(inventory, inventory.length);

Arrays.sort(sorted, Comparator.comparingInt(Product::getProductId));

Product result2 = binarySearch(sorted, 78);

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

}

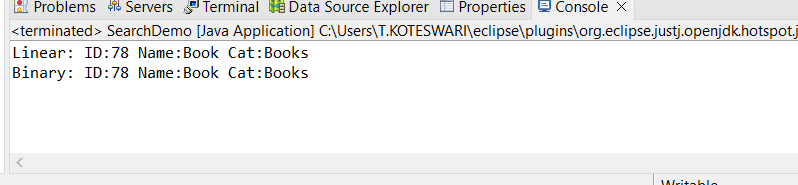
}

4.ANALYSIS:

| **Factor** | **Linear search** | **Binary search** |
| --- | --- | --- |
| **Work per step** | 1 comparison | 1–2 comparisons |
| **Steps needed** | up to *n* | up to log₂ *n* |
| **Extra constraint** | none | input must stay sorted |

As *n* grows, log₂ *n* rises **much** more slowly than *n* (e.g., 1 000 000 items ⇒ log₂ *n* ≈ 20).  
Hence binary search is dramatically faster for large collections, *provided* you can keep the array sorted (or are willing to pay the cost of sorting once then performing many queries).

OUTPUT:



**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:

1. Understanding Recursive Algorithms:

**Recursion** is a technique in programming where a method calls itself to solve a smaller version of the original problem. It is especially useful for problems that can be broken down into similar sub-problems. Each recursive call works on a reduced input and eventually reaches a base case that stops the recursion.

In financial forecasting, recursion can be used to calculate the **future value** by applying a growth formula repeatedly over time. Instead of using a loop, recursion helps express this repetitive growth more naturally.

2. Setup: Method for Future Value Calculation:

We can define a function futureValue(amount, rate, years) that recursively calculates the amount after a given number of years, assuming a constant annual growth rate.

3. Implementation in Java

**package** financialforecasting;

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

**public** **static** **double** futureValueRecursive(**double** value, **double** rate, **int** years) {

**if** (years == 0) **return** value;

**return** *futureValueRecursive*(value, rate, years - 1) \* (1 + rate);

}

**public** **static** **double** futureValueMemo(**double** value, **double** rate, **int** years, **double**[] memo) {

**if** (years == 0) **return** value;

**if** (memo[years] != 0) **return** memo[years];

memo[years] = *futureValueMemo*(value, rate, years - 1, memo) \* (1 + rate);

**return** memo[years];

}

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

**double** initial = 10000;

**double** rate = 0.08;

**int** years = 5;

**double** result1 = *futureValueRecursive*(initial, rate, years);

System.***out***.println("Recursive Forecast: " + result1);

**double**[] memo = **new** **double**[years + 1];

**double** result2 = *futureValueMemo*(initial, rate, years, memo);

System.***out***.println("Memoized Forecast: " + result2);

}

}

**4. Analysis**

**Time Complexity**

* The time complexity of this recursive function is **O(n)**, where n is the number of years.
* This is because the function makes **one recursive call per year**, each performing a simple multiplication and subtraction.

**Optimization Techniques**

While the above recursive solution works for small inputs, it can be inefficient for very large n due to:

* **Call stack limitations** (stack overflow risk)
* **Repeated calculations** (in other recursive problems)

To optimize:

1. **Tail Recursion (if supported by the language)** – Some languages optimize tail-recursive functions to avoid stack overflows.
2. **Memoization** – Store already computed results to avoid redundant calculations (though not needed here due to no overlapping subproblems).
3. **Iterative Approach** – Convert recursion to iteration for better performance:

java

CopyEdit

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

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

amount \*= (1 + rate);

}

return amount;

}

This runs in **O(n)** time and **O(1)** space.

OUTPUT:

