**Week – 1 Algorithms and Data Structures**

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

**Big O Notation** is used to describe how the time or space required by an algorithm increases with the size of the input n. It helps us analyse and compare algorithms in terms of performance and scalability, especially for large inputs.

**Search Algorithm Complexities:**

**Linear Search:**

Best Case: O(1) — when the target element is found at the beginning of the array.

Average Case: O(n/2) which simplifies to O(n) — on average, the element is in the middle.

Worst Case: O(n) — when the element is at the end or not present at all.

**Binary Search:**

Best Case: O(1) — when the middle element is the target.

Average Case: O(log n) — it divides the search space in half at each step.

Worst Case: O(log n) — continues halving until the element is found or search space is empty.

**Code:**

**(Product.java)**

**package** package1;

**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;

}

**public** String toString() {

**return** "ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

**(Linearsearch.java)**

**package** package1;

**public** **class** Linearsearch{

**public** **static** Product search(Product[] products, **int** targetId) {

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

**if** (p.productId == targetId) {

**return** p;

}

}

**return** **null**;

}

}

**(Binarysearch.java)**

**package** package1;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** Binarysearch {

**public** **static** Product search(Product[] products, **int** targetId) {

Arrays.*sort*(products, Comparator.*comparingInt*(p -> p.productId));

**int** low = 0;

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

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

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

**if** (products[mid].productId == targetId) {

**return** products[mid];

} **else** **if** (products[mid].productId < targetId) {

low = mid + 1;

} **else** {

high = mid - 1;

}

}

**return** **null**;

}

}

**(Searchmethods.java)**

**package** package1;

**public** **class** Searchmethods{

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

Product[] products = {

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

**new** Product(205, "Shampoo", "Beauty"),

**new** Product(150, "Chair", "Furniture"),

**new** Product(120, "Book", "Stationery"),

};

System.***out***.println(" Linear Search for ID 150:");

Product result1 = Linearsearch.*search*(products, 150);

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

System.***out***.println("\nBinary Search for ID 150:");

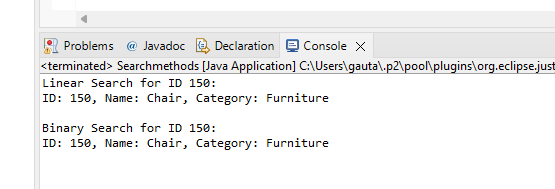
Product result2 = Binarysearch.*search*(products, 150);

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

}

}

**Output:**



**Analysis:**

**Linear Search has a time complexity of:**

Best Case: O(1) – when the element is at the start.

Average/Worst Case: O(n) – when the element is in the middle, at the end, or not present.

**Binary Search has a time complexity of:**

Best Case: O(1) – when the element is in the middle.

Average/Worst Case: O(log n) – due to halving the search space at each step.

Binary search is more suitable for an e-commerce platform as it offers faster performance (O(log n)) on large, sorted product data.

**Exercise 7: Financial Forecasting**

**Recursion** is a technique where a method calls itself to solve smaller instances of a problem.  
It simplifies problems that have a repeating pattern or can be broken into subproblems ( like factorial, Fibonacci, financial predictions).

To recursively calculate the **future value** using the formula:

* Future Value = Present Value × (1 + growthRate)^n

**Code:**

**package** package1;

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

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

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

**return** presentValue;

}

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

}

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

**double** presentValue = 10000;

**double** growthRate = 0.05;

**int** years = 5;

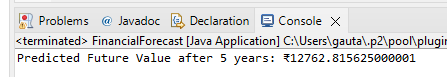
**double** futureValue = *calculateFutureValue*(presentValue, growthRate, years);

System.***out***.println("Predicted Future Value after " + years + " years: ₹" + futureValue);

}

}

**Output**:



**Analysis:**

**Time Complexity**:  
The recursive algorithm runs once per year, so the time complexity is **O(n)** where n is the number of years.

**Optimization**:  
For simple linear recursion like this, optimization is not critical.  
But for more complex recursive problems (like overlapping subproblems), use **memorization** or convert it to **iteration** to reduce computation and avoid stack overflow.