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

### **Understanding Big O Notation**

**Big O Notation** is a mathematical concept used in computer science to describe the efficiency of algorithms in terms of **time** or **space complexity** relative to the **input size** (commonly denoted as *n*). It helps predict how an algorithm will perform as the size of the input grows, focusing on **asymptotic behavior**—what happens when *n* becomes very large.

**Case Scenarios:**

* **Best Case:** The item is found at the first position. For linear search, this is O(1).
* **Average Case:** The item is somewhere in the middle. For linear search, it averages O(n/2), simplified to O(n).
* **Worst Case:** The item is at the end or not found. For linear search, O(n); for binary search (in sorted data), O(log n) in all cases.

**Code :**

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return productId + " - " + productName + " (" + category + ")";

}

}

public class ProductSearch {

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[] sortedProducts, String targetName) {

int left = 0;

int right = sortedProducts.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int compare = sortedProducts[mid].productName.compareToIgnoreCase(targetName);

if (compare == 0) {

return sortedProducts[mid];

} else if (compare < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(301, "Smartwatch", "Electronics"),

new Product(302, "Blender", "Kitchen"),

new Product(303, "Gaming Chair", "Furniture"),

new Product(304, "Pen Drive", "Electronics"),

new Product(305, "Office Bag", "Accessories")

};

Product[] sortedProducts = products.clone();

java.util.Arrays.sort(sortedProducts, java.util.Comparator.comparing(p -> p.productName.toLowerCase()));

String searchTerm = "Gaming Chair";

Product result1 = linearSearch(products, searchTerm);

Product result2 = binarySearch(sortedProducts, searchTerm);

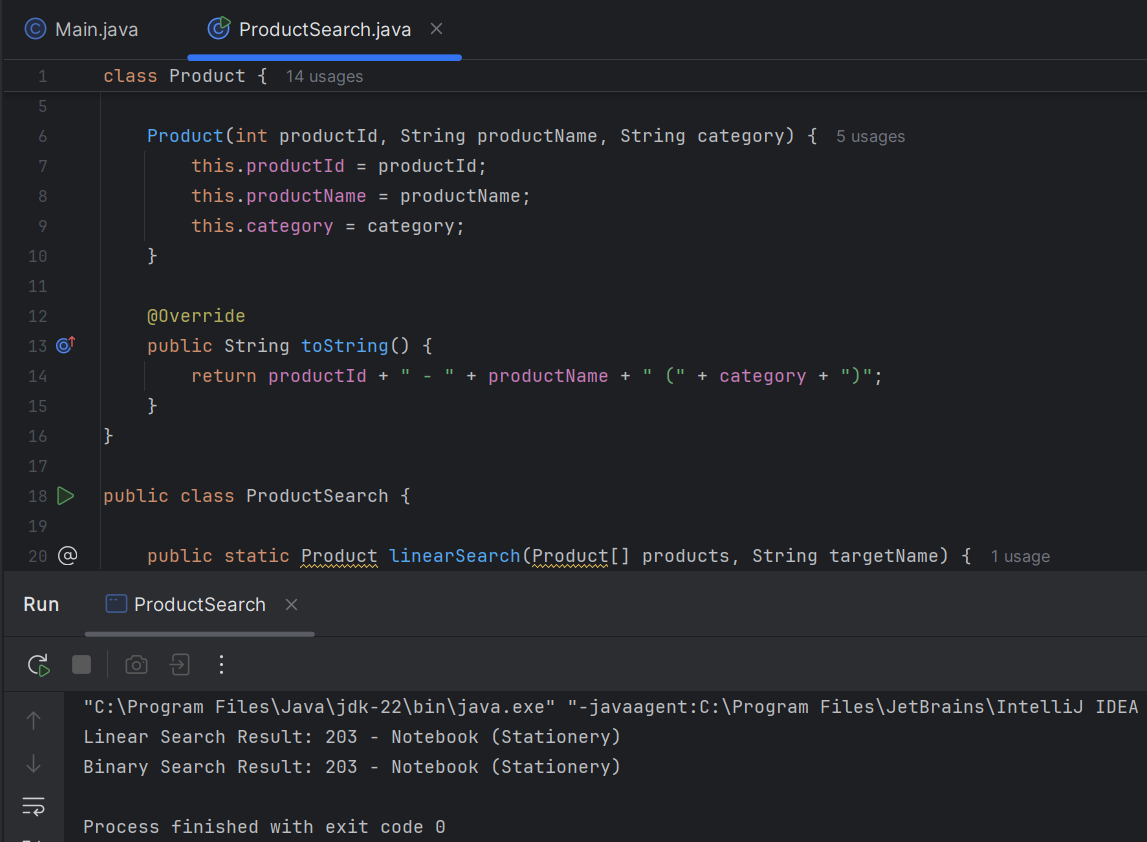
System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Not found"));

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

}

}

**Output :**



**Analysis :**

Algorithm Best Case Average Case Worst Case

Linear Search O(1) O(n) O(n)

Binary Search O(log n) O(log n) O(log n)

1. Linear Search is simple and doesn't require sorted data, but it's inefficient for large datasets.
2. Binary Search is significantly faster (O(log n)) but works only on sorted arrays.

**Conclusion:**

For an e-commerce platform where fast product searches are crucial, binary search is more suitable, especially when product listings are pre-sorted or indexed.

**Exercise 7: Financial Forecasting**

Recursion is a programming technique where a function calls itself to solve a smaller version of the same problem. It’s often useful when the solution to a big problem depends on solutions to smaller subproblems.

For example, instead of using loops, we can solve problems like factorials, Fibonacci numbers, or even growth predictions using recursion. In forecasting, recursion can model how values evolve over time, based on previous values and a growth rate.

It’s elegant and simple, but recursion can sometimes become inefficient if not handled properly, especially with large datasets or deep call stacks.

**Implementation :**

public class Forecast {

public static double forecastValue(double currentValue, double growthRate, int periods) {

if (periods == 0) {

return currentValue;

}

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

}

public static void main(String[] args) {

double initialValue = 1000.0;

double growthRate = 0.05;

int futurePeriods = 5;

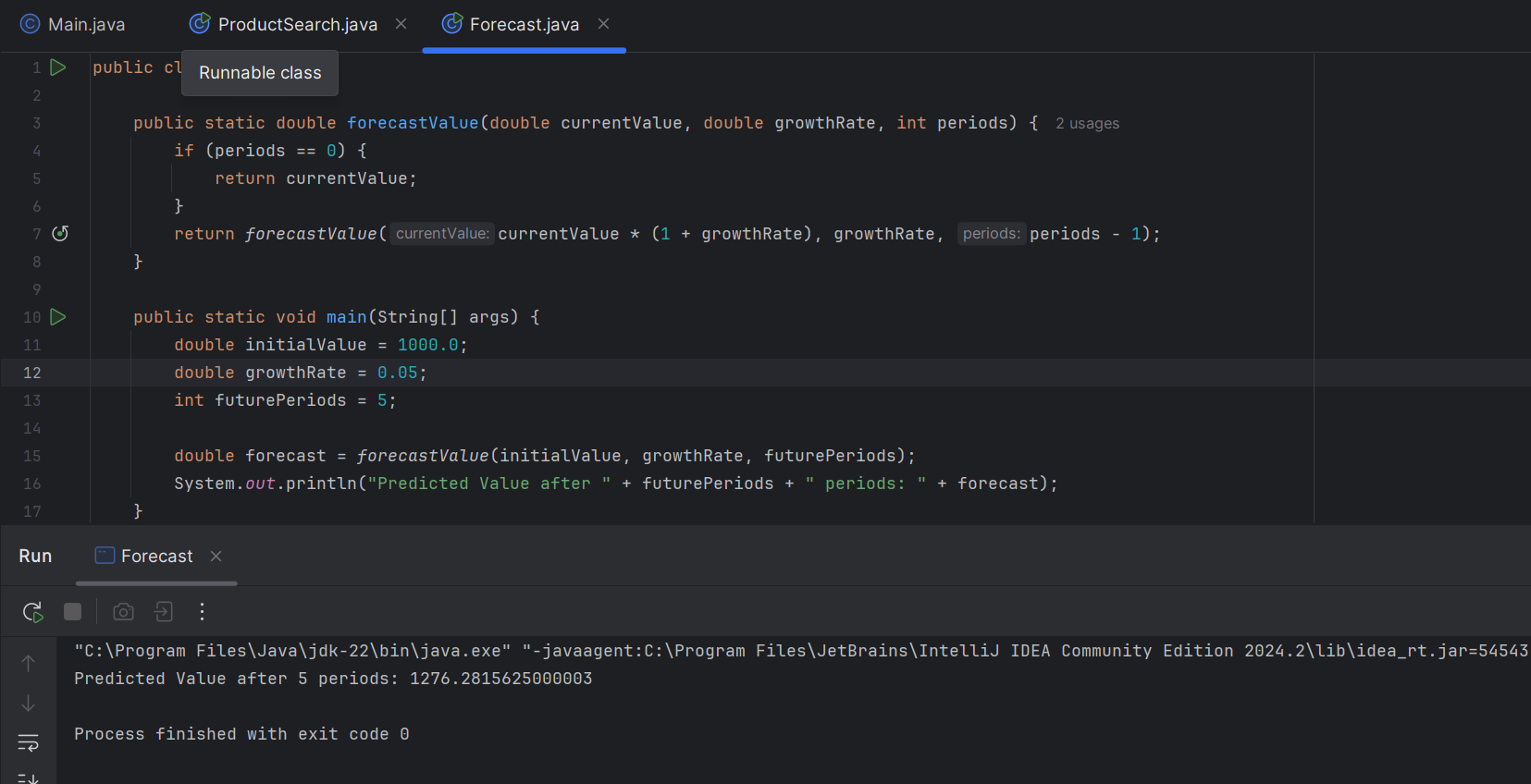
double forecast = forecastValue(initialValue, growthRate, futurePeriods);

System.out.println("Predicted Value after " + futurePeriods + " periods: " + forecast);

}

}

**Output :**



### **4. Analysis**

#### **Time Complexity**

This recursive function has a time complexity of **O(n)**, where n is the number of periods. That’s because it performs one call per period.

#### **Optimization**

While the current recursion is fairly efficient and doesn’t repeat calculations, recursion can sometimes cause performance issues due to deep call stacks or redundant calls (especially in problems like Fibonacci series).

To optimize:

* **Use iteration instead of recursion** if performance is a concern.
* Or use **memoization** (storing already-computed values) for recursive functions that repeat work.

In this case, since each call is unique and doesn't recompute values, it's safe. But for very large periods (e.g., forecasting 1,000 steps), switching to a loop might be more efficient and safer in terms of memory.