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

**Understanding:**

**Big O Notation**

Big O notation describes how an algorithm’s performance changes with the size of the input, focusing on the worst-case scenario. For example, O(n) indicates that the time needed increases linearly as the input grows.

**How Big O Helps**

By using Big O, we can compare algorithms to see which one works best with large data, helping us pick the most efficient one for the task at hand.

**Best, Average, and Worst-case Scenarios**

Best-case: The item is found immediately, such as being the first in the list.

Average-case: The algorithm typically goes through half of the items to find the target.

Worst-case: The item is the last one or not found, requiring a check of every item.

**Code:**

class Product {

int id;

String name;

String category;

Product(int id, String name, String category) {

this.id = id;

this.name = name;

this.category = category;

}

}

public class Main {

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

for (Product p : products) {

if (p.id == id) {

return p;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (products[mid].id == id) {

return products[mid];

} else if (products[mid].id < id) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Smartphone", "Electronics"),

new Product(2, "Tablet", "Electronics"),

new Product(3, "Desk", "Furniture"),

new Product(4, "Notebook", "Stationery"),

new Product(5, "Shampoo", "Beauty")

};

int searchId = 4;

Product result1 = linearSearch(products, searchId);

System.out.println(result1 != null

? "Found Product (Linear Search): " + result1.name + " from category " + result1.category + " with ID " + result1.id

: "Product not found in linear search");

Product result2 = binarySearch(products, searchId);

System.out.println(result2 != null

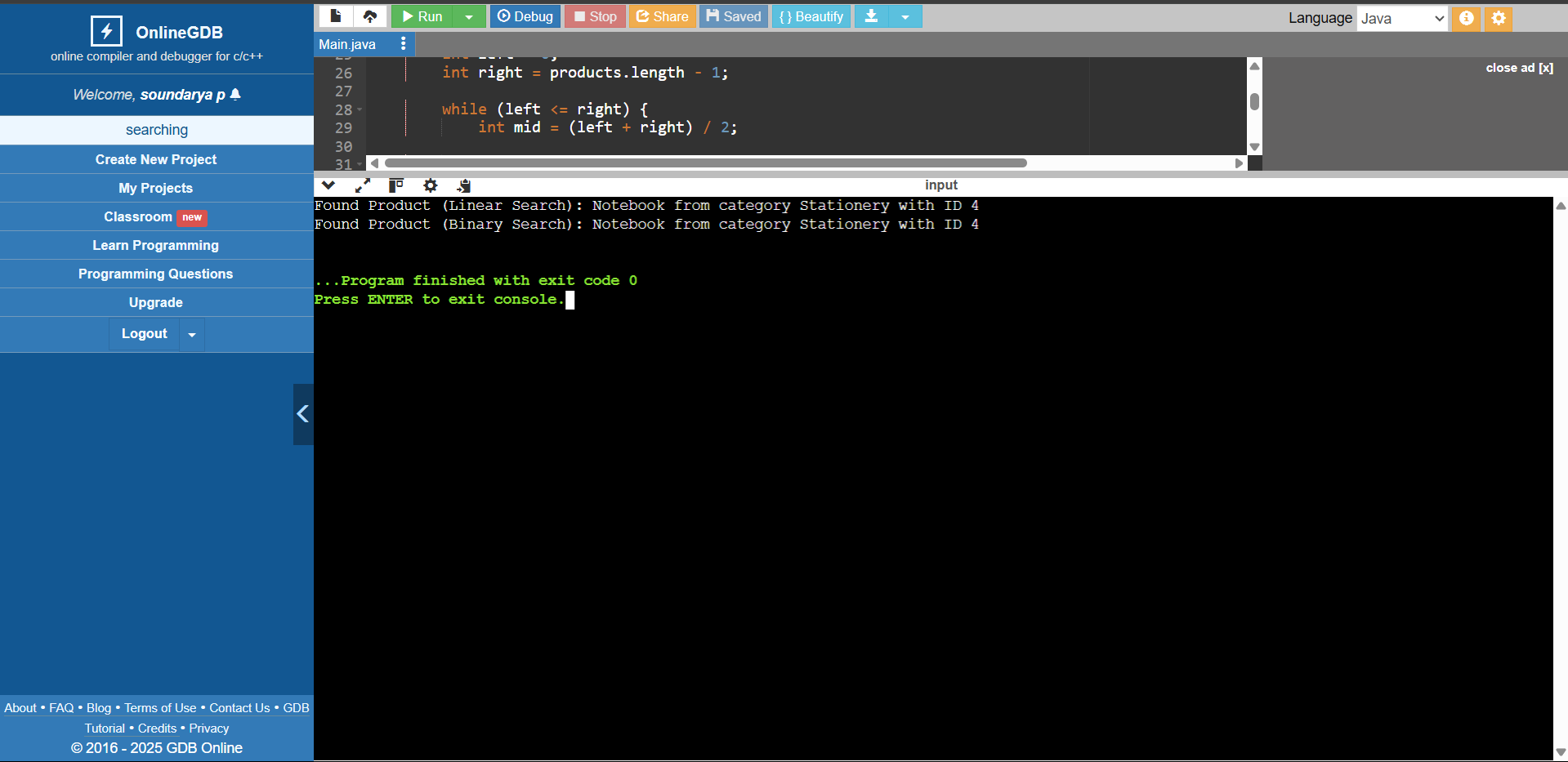
? "Found Product (Binary Search): " + result2.name + " from category " + result2.category + " with ID " + result2.id

: "Product not found in binary search");

}

}

**Output:**

****

**Analysis:**

**Time Complexity**

Linear Search:

The time complexity of linear search is O(n), meaning it goes through every element in the list one by one until it finds the target or reaches the end.

Binary Search:

Binary search has a time complexity of O(log n). It efficiently narrows down the search by dividing the list in half with each step, making it much faster for large datasets.

**Suitable algorithm:**

Linear Search:

It works well for small datasets or unsorted lists where you don't need to organize the data before searching.

Binary Search:

This is best used when the data is already sorted and you need to search quickly, especially with large datasets.

**Exercise 7: Financial Forecasting**

**Understanding:**

* Recursion means a function calling itself.
* It always needs a base case to stop.
* It helps write cleaner and simpler code.
* Each recursive call solves a smaller part.
* Too many recursive calls can crash the program.
* Sometimes loops are better.
* Use recursion when the problem has repeating sub-patterns.

**Code:**

public class Main {

static double forecast(double amount, double rate, int year) {

if (year == 0) return amount;

return forecast(amount \* (1 + rate), rate, year - 1);

}

public static void main(String[] args) {

double amt = 10000;

double r = 0.1;

int y = 5;

double future = forecast(amt, r, y);

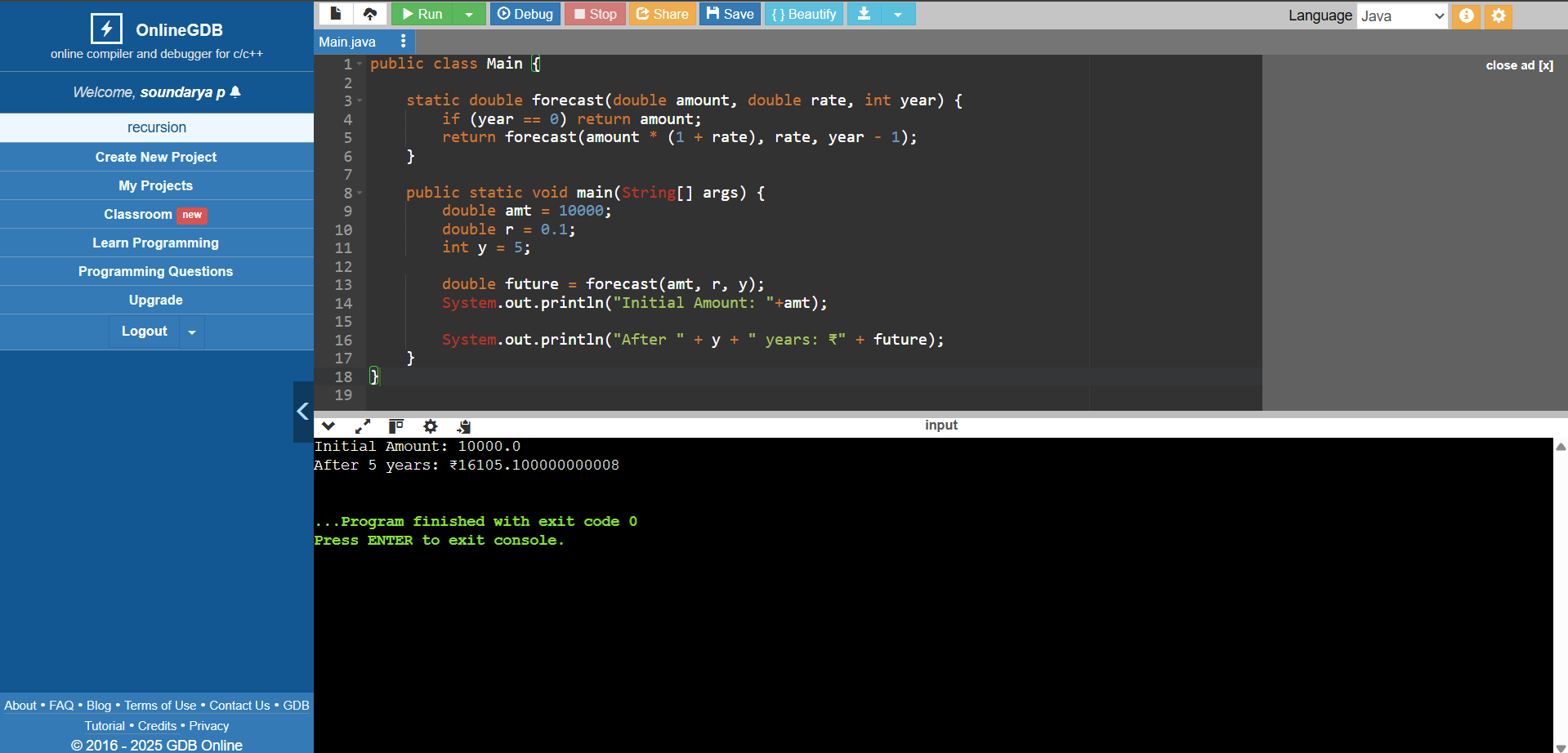
System.out.println("Initial Amount: "+amt);

System.out.println("After " + y + " years: ₹" + future);

}

}

**Output:**

****

**Analysis:**

Time Complexity: O(n)

* One recursive call per year.
* Each call does 1 multiplication.

Space Complexity: O(n)

* Because it uses the call stack for recursion n times.

**Optimized solution:**

Convert to Looping statements (Iteration)

Recursion uses stack space and can lead to stack overflow or slower performance.

Time Complexity: O(n)

Space Complexity: O(1)