**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, "Laptop", "Electronics"),

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

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

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

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

};

int searchId = 5;

Product result1 = linearSearch(products, searchId);

System.out.println("Linear Search Result: " + (result1 != null

? result1.id + " - " + result1.name + " (" + result1.category + ")"

: "Not Found"));

Product result2 = binarySearch(products, searchId);

System.out.println("Binary Search Result: " + (result2 != null

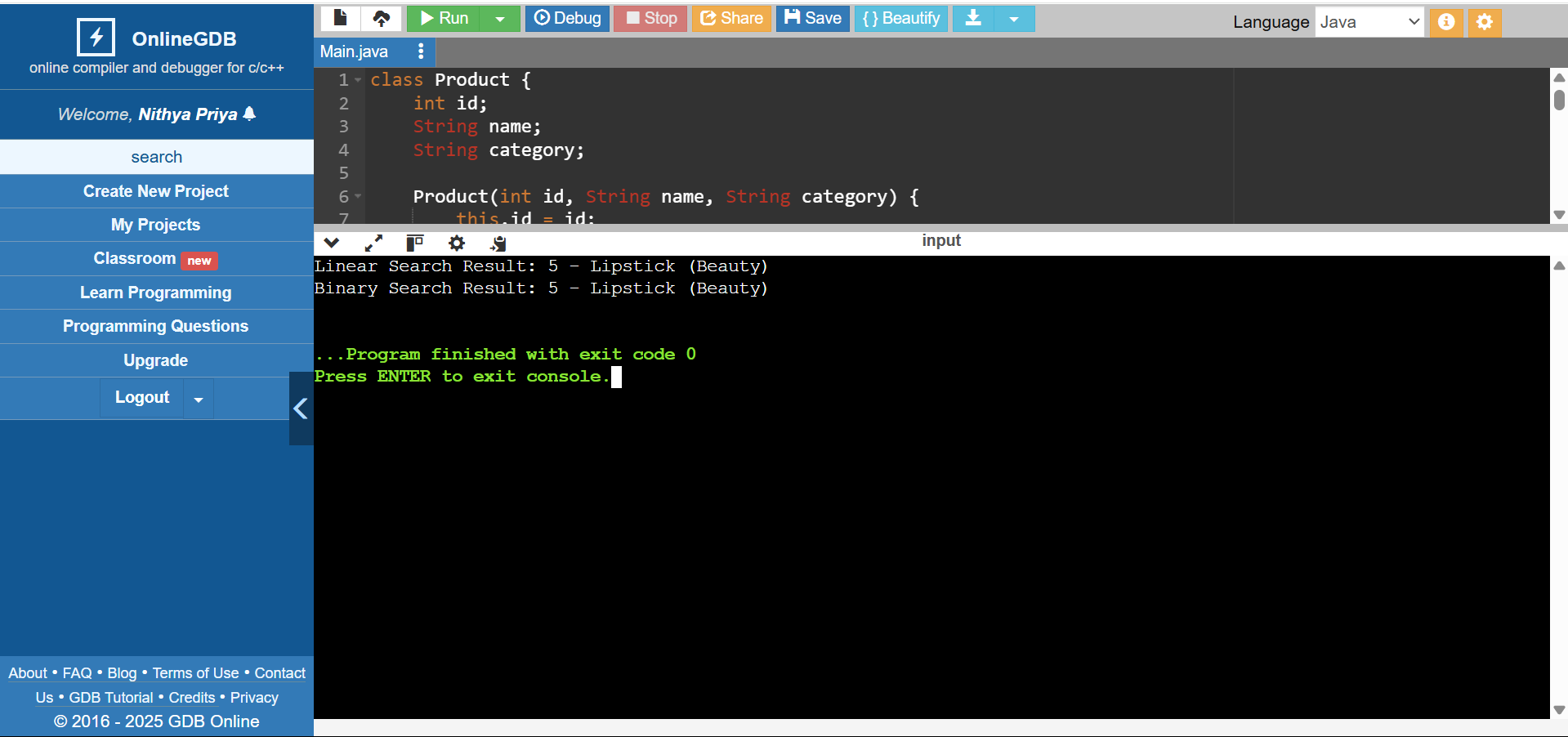
? result2.id + " - " + result2.name + " (" + result2.category + ")"

: "Not Found"));

}

}

**Output:**

****

**Analysis:**

**Time Complexity**

Linear Search:

O(n), where n is the number of elements. It checks each element one by one.

Binary Search:

O(log n). It divides the list in half with each step, making it faster for large datasets.

**Algorithm Suitability**

Linear Search: Good for small or unsorted datasets.

Binary Search: Ideal for large, sorted datasets.

**Exercise 7: Financial Forecasting**

**Understanding:**

* Recursion breaks a problem into smaller versions of itself.
* It’s important to have a base case to stop the recursion.
* Recursion is helpful for solving problems with repeating sub-patterns.
* Too many recursive calls can lead to memory overflow.
* For some problems, loops can be more efficient than recursion.
* Each recursive call tackles a smaller part of the problem.
* It makes code cleaner and simpler when dealing with complex problems.

**Code:**

public class Main {

static double forecast(double amt, double rate, int years) {

if (years == 0) return amt;

return (1 + rate) \* forecast(amt, rate, years - 1);

}

public static void main(String[] args) {

double amt = 10000;

double r = 0.1;

int y = 5;

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

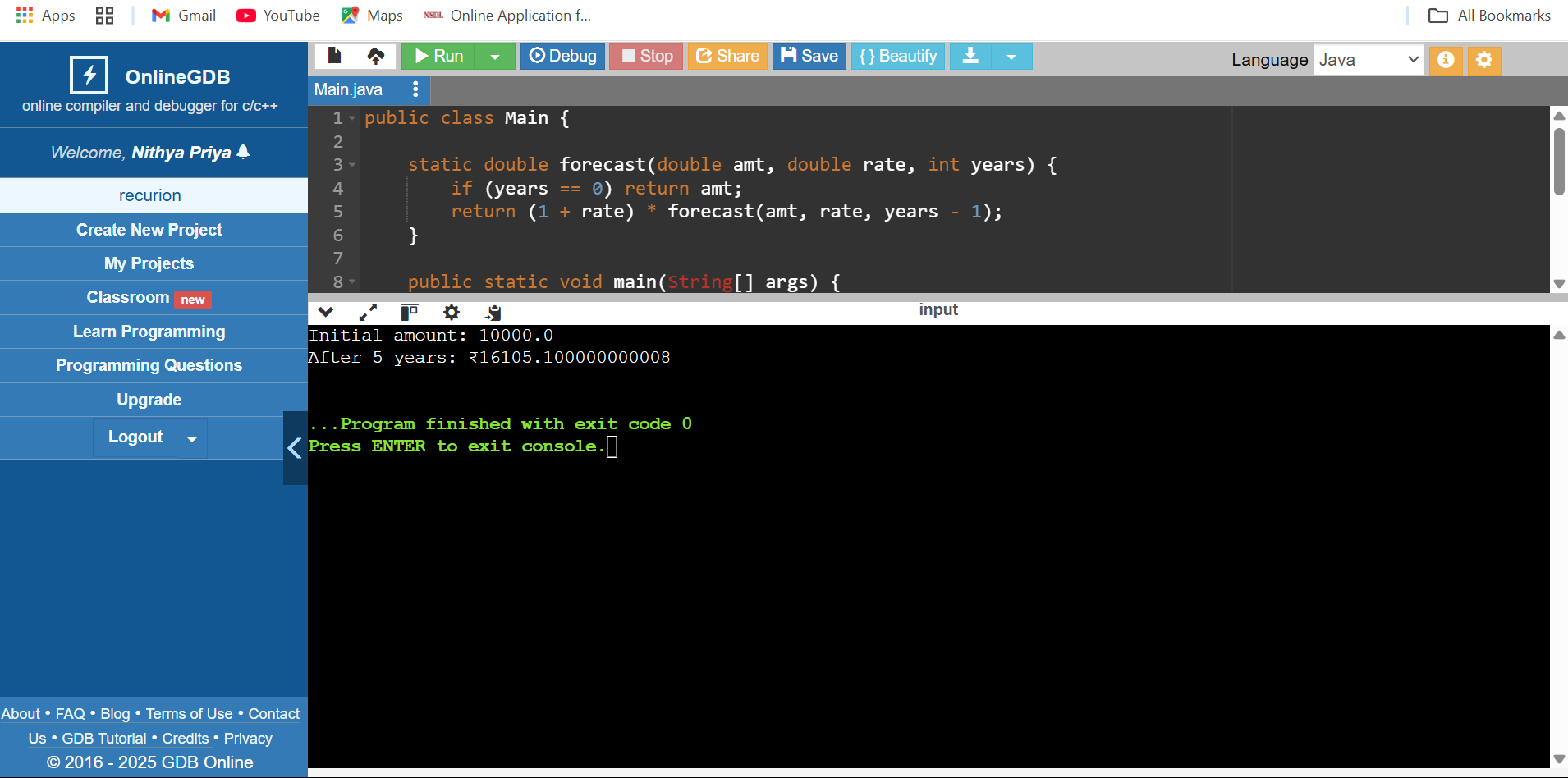
System.out.println("Initial amount: " +amt);

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 Iteration (Loop)

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

Time Complexity: O(n)

Space Complexity: O(1)