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

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 productId + " - " + productName + " - " + category;

}

}  
import java.util.Arrays;

import java.util.Comparator;

public class ProductSearch {

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

for (Product p : products) {

if (p.productId == id) return p;

}

return null;

}

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

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (products[mid].productId == id) return products[mid];

else if (products[mid].productId < id) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void sortById(Product[] products) {

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

}

}

public class SearchTest {

public static void main(String[] args) {

Product[] items = {

new Product(104, "Laptop", "Electronics"),

new Product(101, "Shirt", "Clothing"),

new Product(105, "Book", "Education"),

new Product(102, "Shoes", "Footwear"),

new Product(103, "Watch", "Accessories")

};

Product result1 = ProductSearch.linearSearch(items, 105);

if (result1 != null) {

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

} else {

System.out.println("Product not found");

}

ProductSearch.sortById(items);

Product result2 = ProductSearch.binarySearch(items, 105);

if (result2 != null) {

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

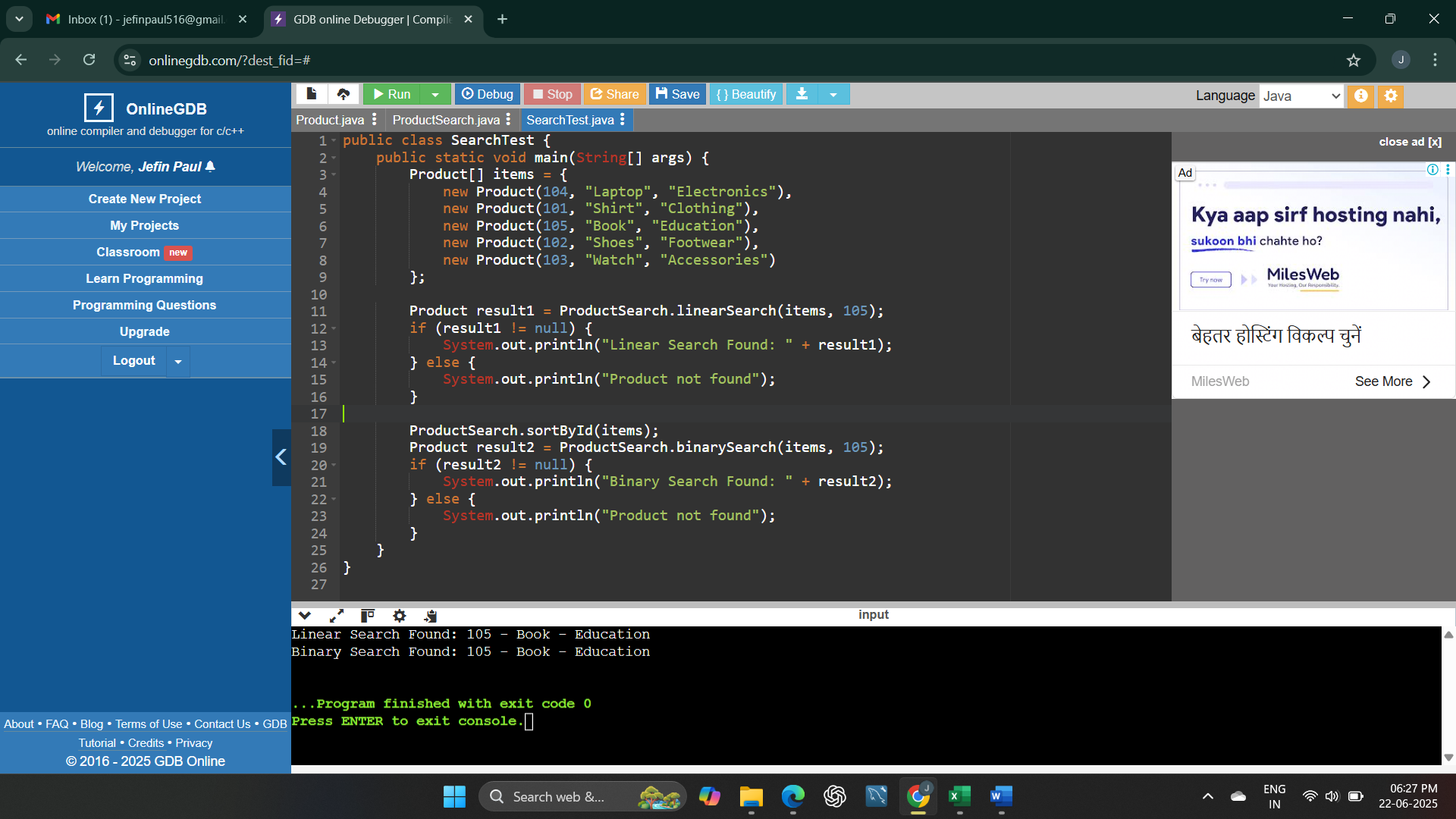
} else {

System.out.println("Product not found");

}

}

}



**Understand Asymptotic Notation**

**Big O notation** is used to describe how the running time or space requirements of an algorithm grow as the size of the input increases. It doesn’t focus on exact timing but gives us an idea of how efficient or scalable an algorithm is when the input becomes large.

It helps in comparing different algorithms in terms of their performance, especially in worst-case scenarios.

**Best, Average, and Worst-Case Scenarios:**

* **Best Case**: The fastest possible scenario (e.g., the element is found on the first try).
* **Average Case**: The performance expected for a typical or random input.
* **Worst Case**: The slowest situation (e.g., element is not found or is at the last position).

These help in understanding how consistent and reliable an algorithm is across different inputs.

Analysis

Time Complexity:

Algorithm Best Case Average Case Worst Case

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

Binary Search O(1) O(log n O(log n)

Linear Search checks each element one by one. In the worst case, it has to check all elements.

Binary Search works only on sorted arrays and keeps dividing the search range in half, making it much faster for large datasets.

Which is More Suitable:

For an e-commerce platform, performance and speed are very important, especially when searching through thousands or millions of products.

Binary Search is more suitable when the product list is already sorted or can be sorted once and reused. It is significantly faster and efficient.

However, if the data is constantly changing (like frequent additions or deletions), Linear Search may be more flexible, but it’s slower for large inputs.

In most real-world applications, we go for binary search or more advanced search algorithms, often with indexing or database support, to handle huge data quickly.

**Exercise 7: Financial Forecasting**

public class FinancialForecast {

public static double predictFutureValue(double presentValue, double rate, int years) {

if (years == 0) return presentValue;

return predictFutureValue(presentValue \* (1 + rate), rate, years - 1);

}

public static double predictFutureValueOptimized(double presentValue, double rate, int years) {

return presentValue \* Math.pow(1 + rate, years);

}

}

public class ForecastTest {

public static void main(String[] args) {

double presentValue = 10000;

double annualRate = 0.08;

int years = 5;

double recursiveResult = FinancialForecast.predictFutureValue(presentValue, annualRate, years);

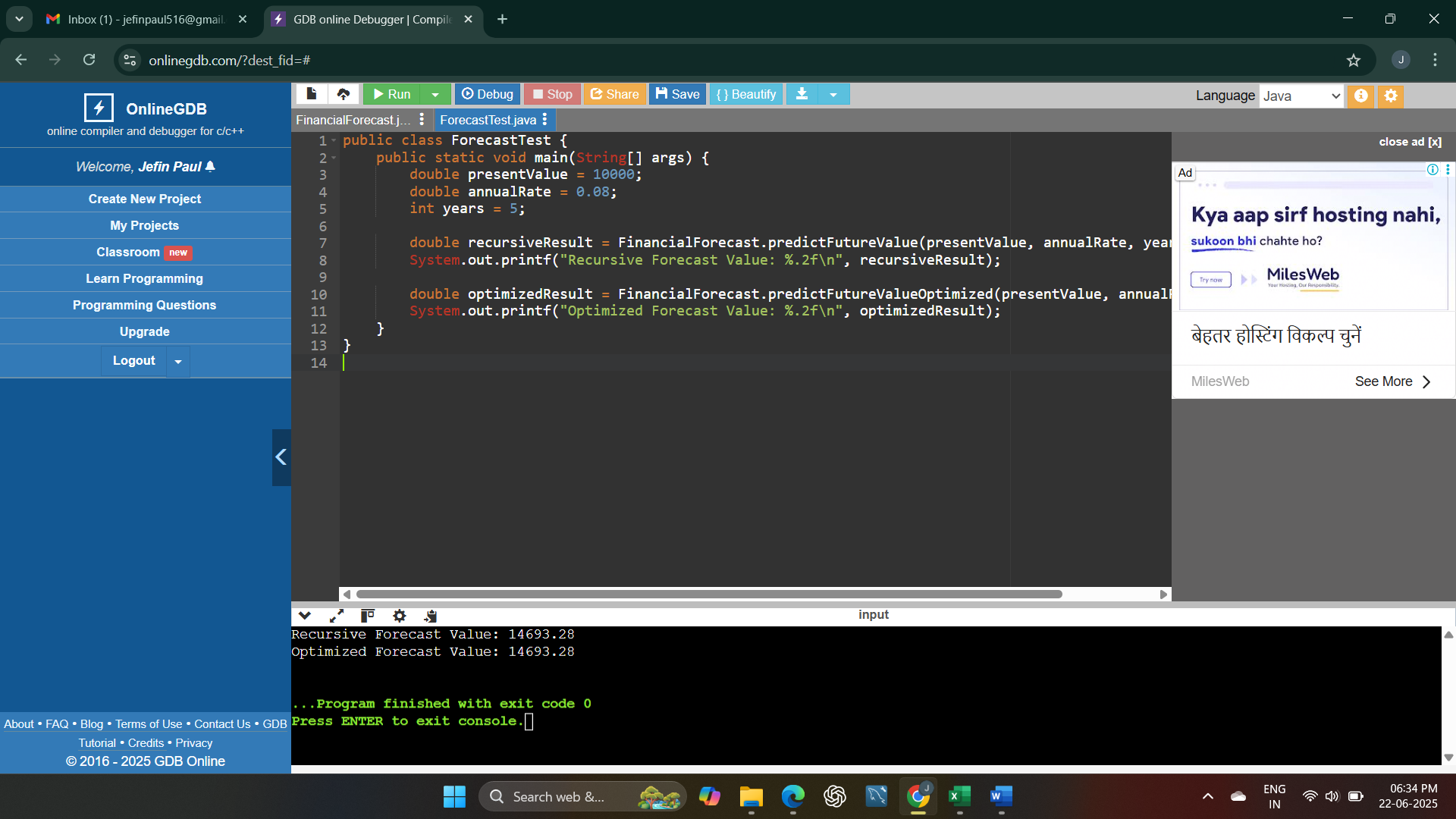
System.out.printf("Recursive Forecast Value: %.2f\n", recursiveResult);

double optimizedResult = FinancialForecast.predictFutureValueOptimized(presentValue, annualRate, years);

System.out.printf("Optimized Forecast Value: %.2f\n", optimizedResult);

}

}

  
**Understand Recursive Algorithms**

**Recursion** is a programming concept where a method calls itself to solve a smaller part of the same problem. It continues doing so until it reaches a base case — a simple, stopping condition.

Recursion is helpful when a problem can be broken down into smaller, similar subproblems. For example, calculating future values over a period of time can be done recursively by calculating each year's value based on the previous year.

It makes the code shorter and easier to understand for problems that follow a repetitive pattern, like tree traversal, factorials, or financial forecasting.

**Analysis**

**Time Complexity of Recursive Algorithm:**

The recursive method for predicting future value:

public static double predictFutureValue(double presentValue, double rate, int years)

has a time complexity of **O(n)**, where n is the number of years. That's because for each year, the method makes one recursive call until it reaches year 0.

**Optimization Strategy:**

Although the recursive approach is easy to understand, it becomes inefficient for large inputs due to deep recursive calls, which can cause **stack overflow** or **excessive computation**.

To optimize it, we can either:

* Use the **mathematical formula** directly with Math.pow(), which gives the result in **O(1)** time.
* Or apply **memoization** to store previously calculated values, though that's more useful when values overlap (not needed in simple compound interest).

In financial forecasting, especially when we deal with large time periods or real-time systems, it's better to avoid recursion and use the **optimized direct formula** for better performance.