**MANDATORY JAVA HANDS-ON**

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

Big O notation is used to estimate the worst case time complexity of an algorithm.

It helps to understand the maximum time needed for algorithm to execute.

Two types of searching techniques are:

1. Linear Search

Best Case: O(1)

Average Case: O(n)

Worst Case: O(n)

1. Binary Search

Best Case: O(1)

Average Case: O(log n)

Worst Case: O(log n)

Main.java

public class Main {  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(101, "Laptop", "Electronics"),  
 new Product(102, "Shirt", "Clothing"),  
 new Product(103, "Book", "Education"),  
 new Product(104, "Phone", "Electronics")  
 };  
  
 System.*out*.println("LINEAR SEARCH");  
 Product p1 = LinearSearch.*search*(products, "Shirt");  
 if (p1 != null) {  
 System.*out*.println( p1.productName+" FOUND");  
 } else {  
 System.*out*.println("Not Found");  
 }  
  
 System.*out*.println("BINARY SEARCH");  
 Product p2 = BinarySearch.*search*(products, "Phone");  
 if (p2 != null) {  
 System.*out*.println( p2.productName+" FOUND");  
 } else {  
 System.*out*.println("Not Found");  
 }  
 }  
}

Product.java

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;  
 }  
}

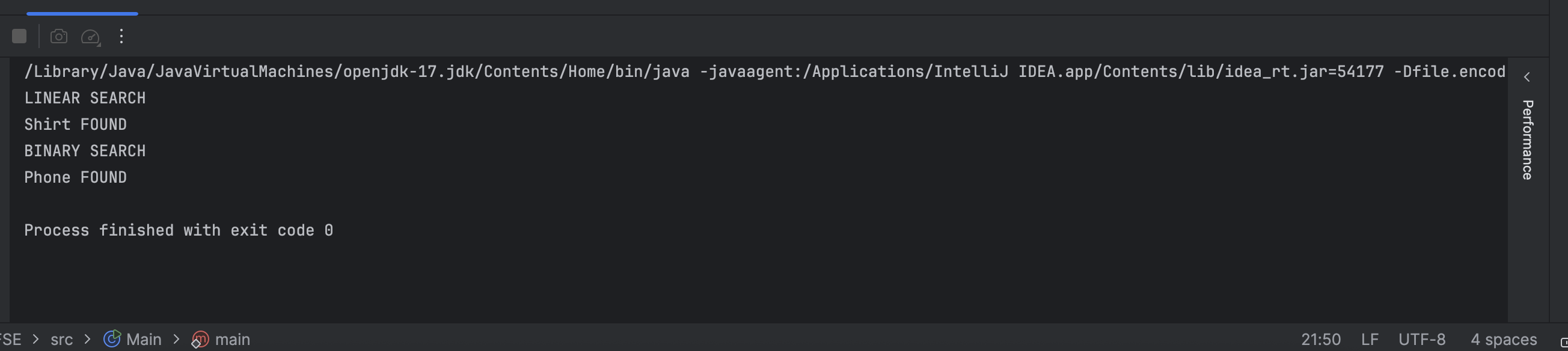
LinearSearch.java

public class LinearSearch {  
 public static Product search(Product[] products, String targetName) {  
 for (Product p : products)  
 if (p.productName.equals(targetName))  
 return p;  
 return null;  
 }  
}

BinarySearch.java

import java.util.Arrays;  
import java.util.Comparator;  
  
public class BinarySearch {  
 public static Product search(Product[] products, String targetName) {  
 Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName));  
  
 int left = 0, right = products.length - 1;  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 int cmp = products[mid].productName.compareTo(targetName);  
  
 if (cmp == 0) return products[mid];  
 if (cmp < 0) left = mid + 1;  
 else right = mid - 1;  
 }  
 return null;  
 }  
}

OUTPUT:



Analysis:

Binary Search [O(log n)] will be most suitable for the E-commerce platform because it will halve the search space. We can make it more efficient if we sort the products during the insertion phase.

**Exercise 7: Financial Forecasting**

**RECURSION:**

Recursion is a method where a function calls itself to solve smaller instances of a problem. It's useful when a problem can be broken down into similar subproblems.

Let financial formula be:

**FutureValue = PresentValue \* (1 + rate)^years**

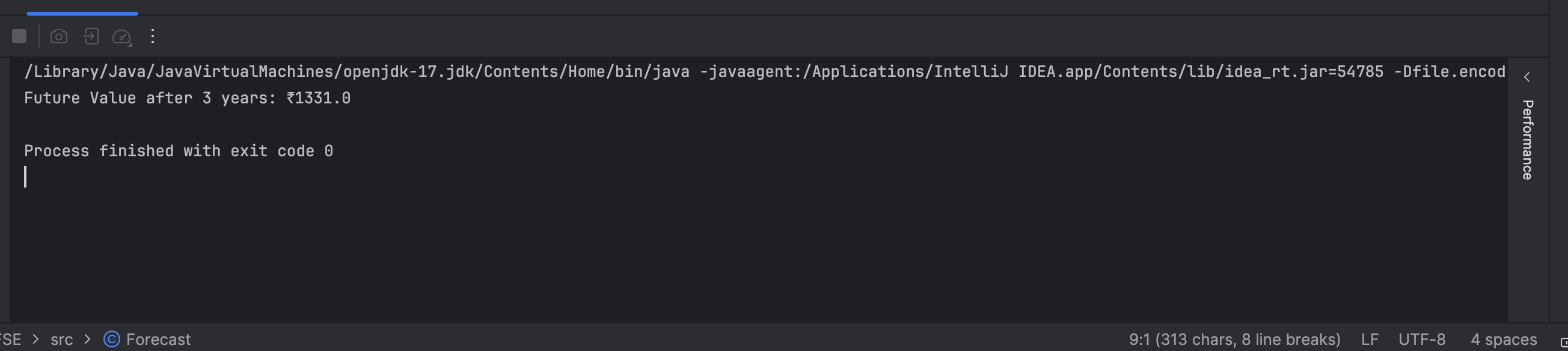
Main.java

public class Main {  
 public static void main(String[] args) {  
 double presentValue = 1000; // Initial investment  
 double rate = 0.10; // 10% growth rate per year  
 int years = 3;  
¯  
 double futureValue = Forecast.*calculateFutureValue*(presentValue, rate, years);  
  
 System.*out*.println("Future Value after " + years + " years: ₹" + futureValue);  
 }  
}

Forecast.java

public class Forecast {  
 // Recursive method to calculate future value  
 public static double calculateFutureValue(double presentValue, double rate, int years) {  
 if (years == 0)  
 return presentValue;  
 return *calculateFutureValue*(presentValue \* (1 + rate), rate, years - 1);  
 }  
}

OUTPUT:



1.**Time Complexity**: O(n) — because it performs one recursive call per year.

OPTIMIZATION:

We can optimize the recursion process by storing every calculated value of the subsequent years in an array, so that we can retrieve it when it is called in future. It is said to be Dynamic Programming.