# **DATA STRUCTURES AND ALGORITHMS**

## **1. E-commerce Platform Search Function**

### **Code:**

### **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 int search(Product[] products, String productName) {

for (int i = 0; i < products.length; i++) {

if (products[i].productName.equalsIgnoreCase(productName)) {

return i;

}

}

return -1;

}

}

### **BinarySearch.java**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static int search(Product[] products, String productName) {

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

int result = productName.compareToIgnoreCase(products[mid].productName);

if (result == 0)

return mid;

if (result > 0)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

}

### **Main.java**

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(101, "Laptop", "Electronics"),

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

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

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

new Product(105, "Shirt", "Clothing")

};

String target = "Phone";

int linearIndex = LinearSearch.search(products, target);

System.out.println("Linear Search Index: " + linearIndex);

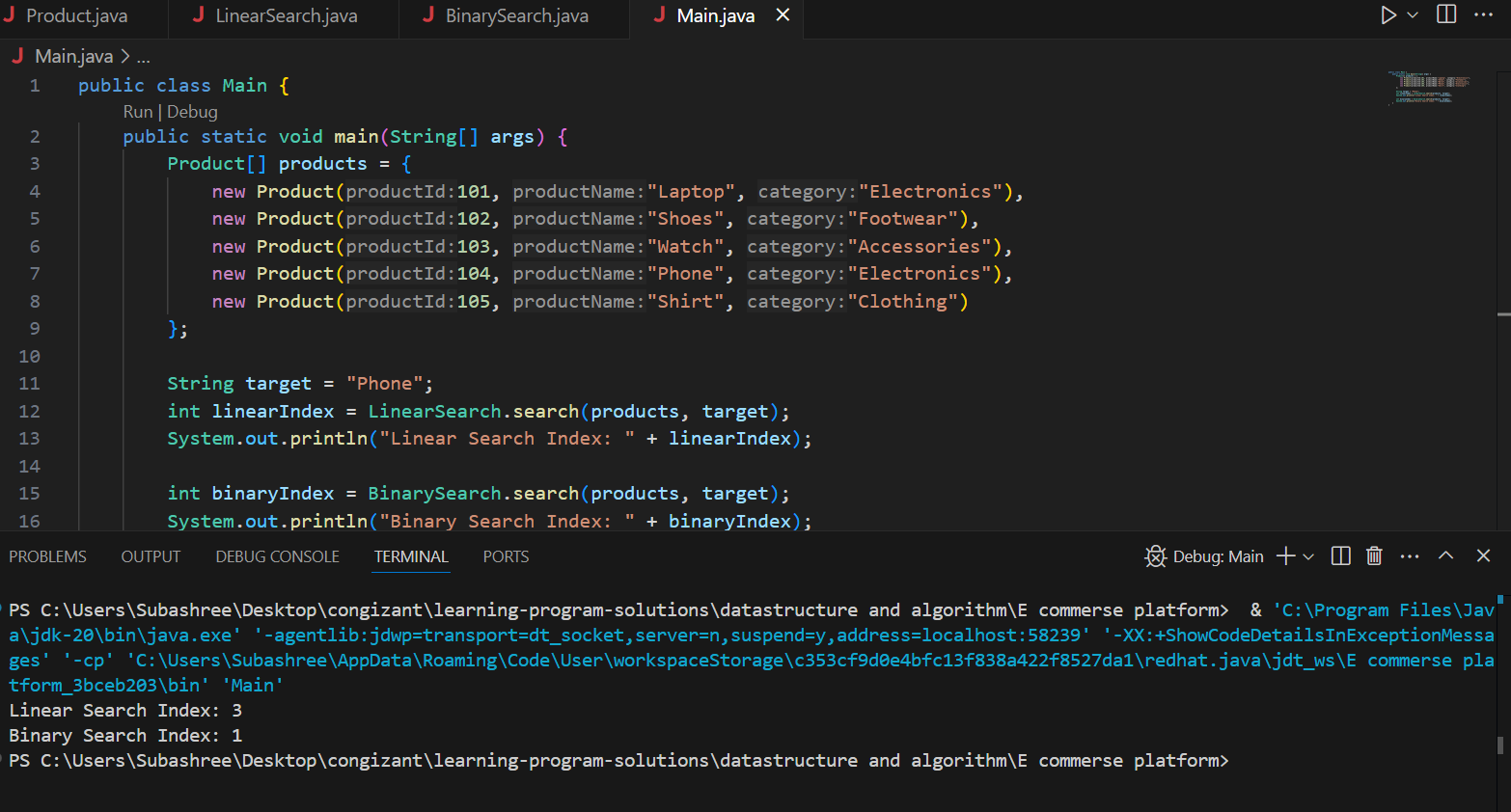
int binaryIndex = BinarySearch.search(products, target);

System.out.println("Binary Search Index: " + binaryIndex);

}

}

### Output:



**Big O Notation & Algorithm Analysis:**

Big O notation is used to describe the upper bound of an algorithm’s running time in terms of input size. It helps us predict the scalability and performance of algorithms by expressing time or space complexity.

* **Best Case**: Fastest scenario (example: element found at the first index).
* **Average Case**: Expected performance across all inputs.
* **Worst Case**: Slowest performance, example: element not found.

**Linear vs Binary Search Time Complexity:**

* **Linear Search:**
  + Best Case: O(1)
  + Average Case: O(n)
  + Worst Case: O(n)
* **Binary Search (on sorted array):**
  + Best Case: O(1)
  + Average Case: O(log n)
  + Worst Case: O(log n)

**Most Suitable Algorithm:**

Binary search is more suitable for platforms where the product list is sorted and frequent searches are performed. It provides faster lookups compared to linear search. However, if the product list is frequently changing and not sorted, linear search may be simpler to use.

## 2. Financial Forecasting

### Code:

### **FinancialForecast.java**

public class FinancialForecast {

public static double predictFutureValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

return predictFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

}

### **Main.java**

public class Main {

public static void main(String[] args) {

double currentValue = 10000;

double growthRate = 0.05;

int years = 5;

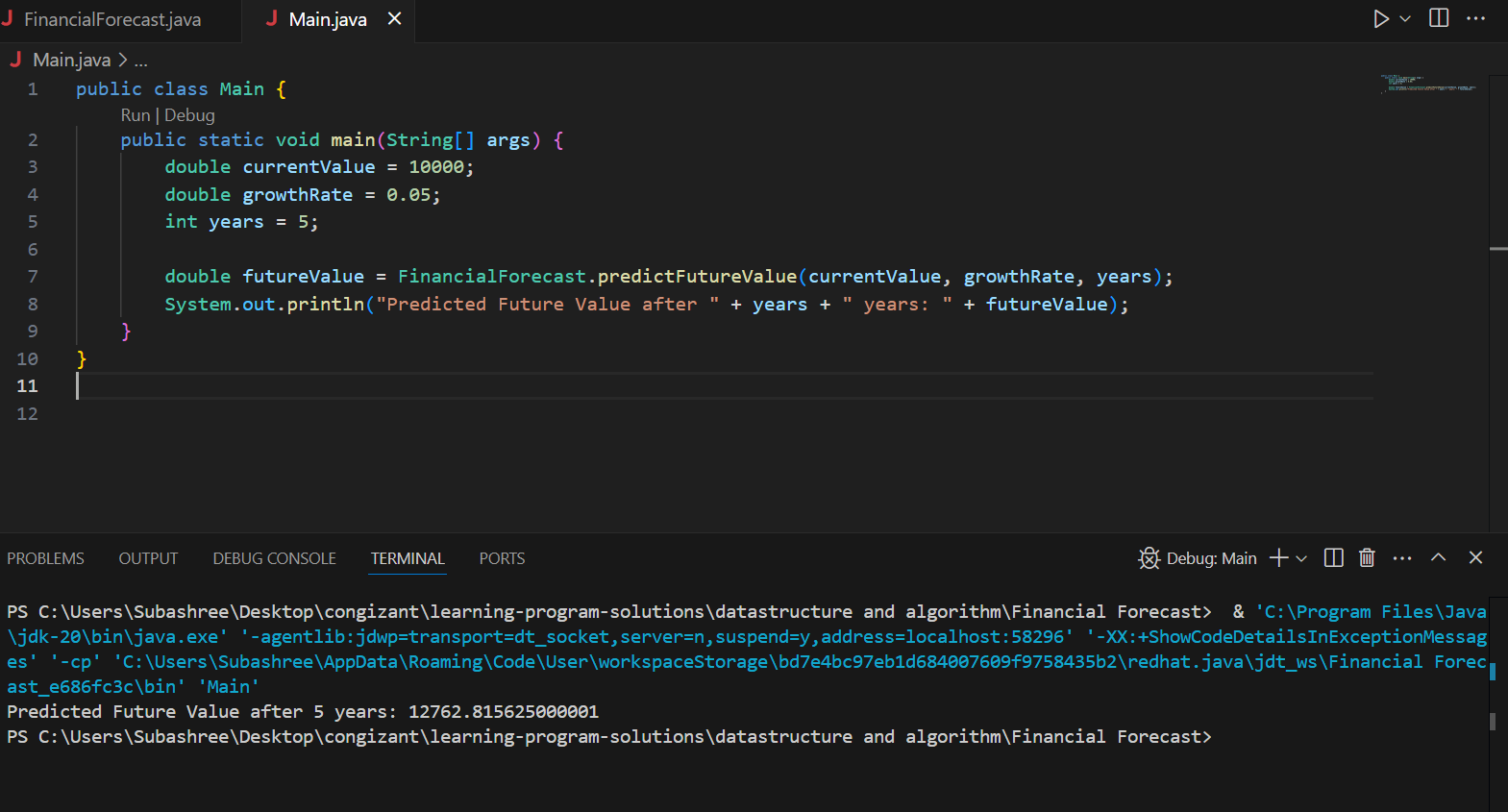
double futureValue = FinancialForecast.predictFutureValue(currentValue, growthRate, years);

System.out.println("Predicted Future Value after " + years + " years: " + futureValue);

}

}

### Output:



**Concept of Recursion:**

Recursion is a method where a function calls itself to solve smaller instances of the same problem. It simplifies code for problems that have a repetitive or nested structure, such as tree traversal, factorial calculation, and forecasting based on repeated patterns.

**Time Complexity Analysis:**

The recursive function runs once for each year, so its time complexity is **O(n)**, where n is the number of years. It is linear in terms of the depth of recursion.

**Optimization:**

To optimize recursive solutions:

* Use **memoization** to store previously computed values.
* Alternatively, use an **iterative approach** to avoid call stack overhead and reduce the risk of stack overflow in large inputs.

Optimized (iterative) approach:

public static double predictFutureValueIterative(double currentValue, double growthRate, int years) {

for (int i = 0; i < years; i++) {

currentValue \*= (1 + growthRate); }

return currentValue;}