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

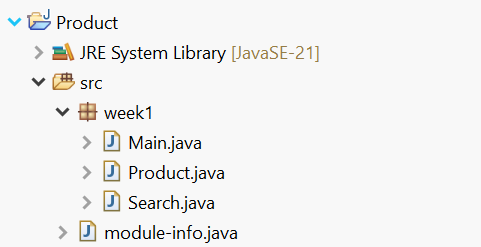
**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Project Structure:**

****

**Big O Notation:**

Big O notation is a way to understand how fast or slow an algorithm will be as the size of the input grows. It gives us a general idea of the algorithm’s efficiency, especially when dealing with large amounts of data. Instead of focusing on exact timing, Big O tells us how the runtime increases—whether it stays constant, grows slowly, or gets really big as more data comes in.

Some examples include :  
O(n) – Where time increases as the input size increases.

O(1) – Where time remains constant irrespective of size  
O(n^2) – Where time increases quadratically with the input size.

**Time Complexities of Search Operations:**

* In a linear search, the best-case scenario occurs when the target element is found at the very beginning of the list, resulting in a time complexity of O(1).
* In The average-case scenario if we assume the target is somewhere in the middle of the list, the complexity result in O(n), where n is the size of the search space.
* The worst-case occurs when the target is either at the end of the list or not present at all, leading to a full scan of all elements, which also results in O(n) time complexity.
* In a binary search, it requires the data to be sorted, the best-case scenario happens when the middle element is the target, giving O(1) time, as we always find the middle element in the binary search.
* The average-case scenario involves several halving steps to locate the target, resulting in O(log n) time, Where n is the size of the search space.
* The worst-case scenario occurs when the target is not found or is found after the maximum number of splits, which still results in O(log n) time complexity.

**Code:**

**Product.java:**

package week1;

public class Product {

int productId;

String productName;

String category;

Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

}

**Search.java:**

package week1;

public class Search {

public static Product linearSearch(Product[] products, String target) {

for (Product product : products) {

if (product.productName.equalsIgnoreCase(target)) {

return product; // found

}

}

return null; // not found

}

public static Product binarySearch(Product[] products, String target) {

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

while (low <= high) {

int mid = (low + high) / 2;

int rs = products[mid].productName.compareToIgnoreCase(target);//Comparing the products[mid] name to the target product name

// if both string are equal comparetoIgnoreCase return 0 else abs difference btw strings

if (rs == 0) return products[mid];//found

else if (rs < 0) low = mid + 1;

else high = mid - 1;

}

return null; //not found

}

}

**Main.java:**

package week1;

import java.util.\*;

public class Main {

public static void main(String args[]) {

Scanner sc = new Scanner(System.*in*);

System.*out*.println("Please Enter the size of the products table: ");

int sz = sc.nextInt();

Product[] products = new Product[sz];

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

int id;

String pname;

String pcategory;

System.*out*.println("Please Enter Product id: ");

id = sc.nextInt();

System.*out*.println("Please Enter Product name: ");

pname = sc.next();

System.*out*.println("Please Enter Product Category: ");

pcategory = sc.next();

Product product = new Product(id,pname,pcategory);

products[i] = product;

}

System.*out*.println("Please Enter Product to be Searched: ");

String target = sc.next();

Product linearsrch = Search.*linearSearch*(products, target);

Product binarysrch = Search.*binarySearch*(products, target);

if(linearsrch==null) {

System.*out*.println("The product is not found by Linear Search");

}else if(binarysrch==null) {

System.*out*.println("The product is not found by Binary Search");

}

else {

System.*out*.println("The Result by Linear Search is :" + linearsrch.productName);

System.*out*.println("The Result by Binary Search is :" + binarysrch.productName);

}

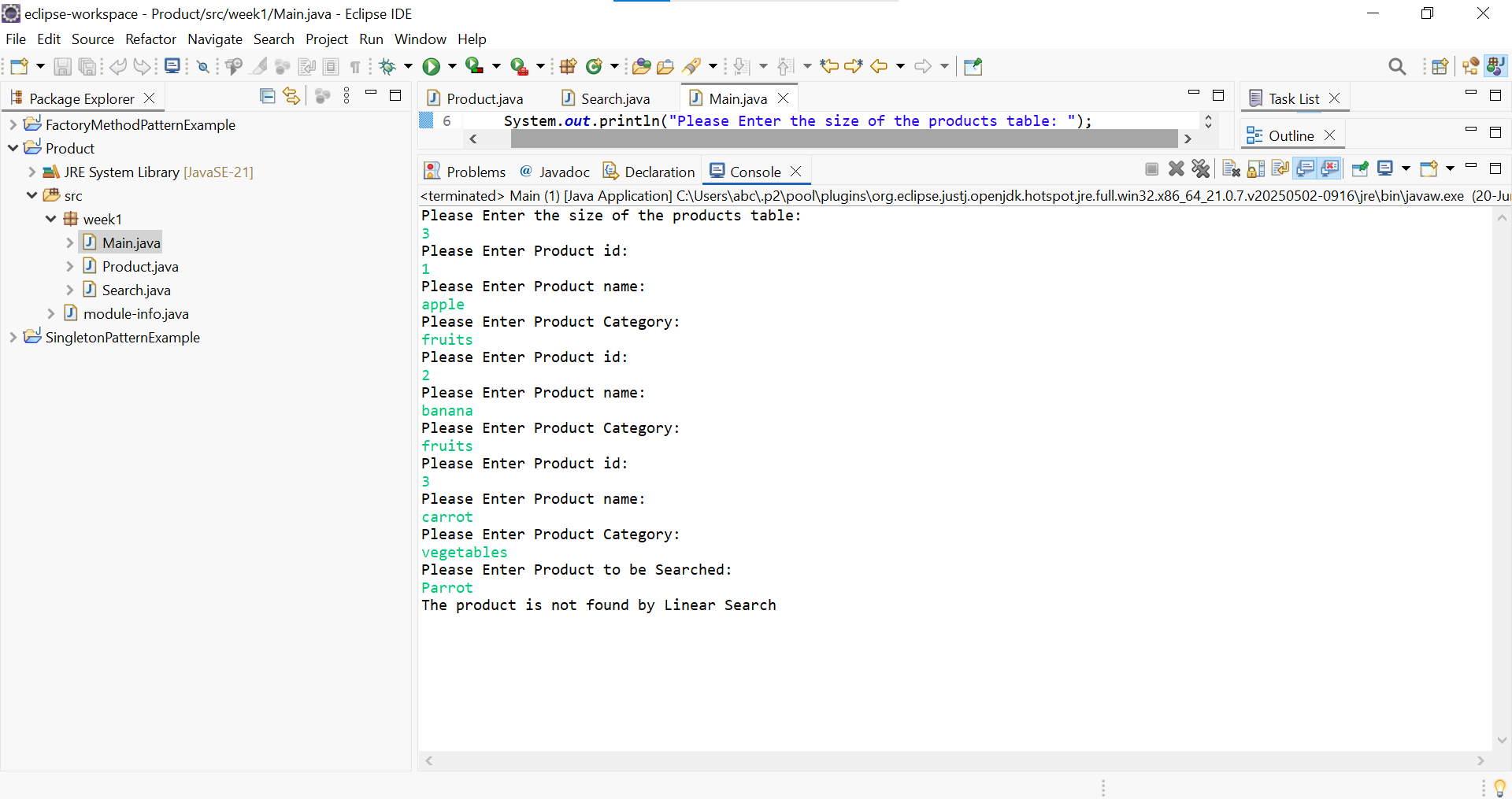
sc.close();

}

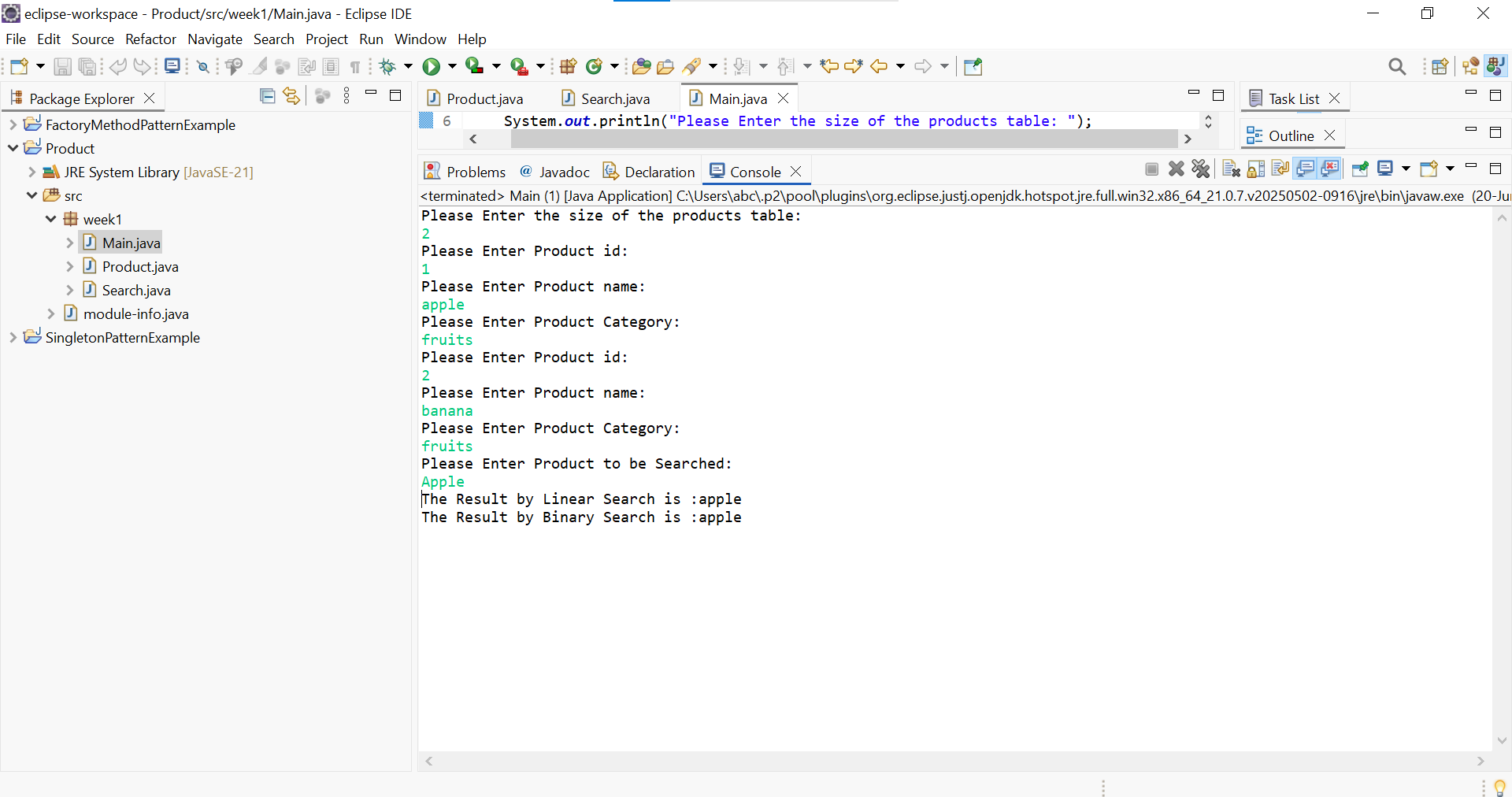
}

**Output:**

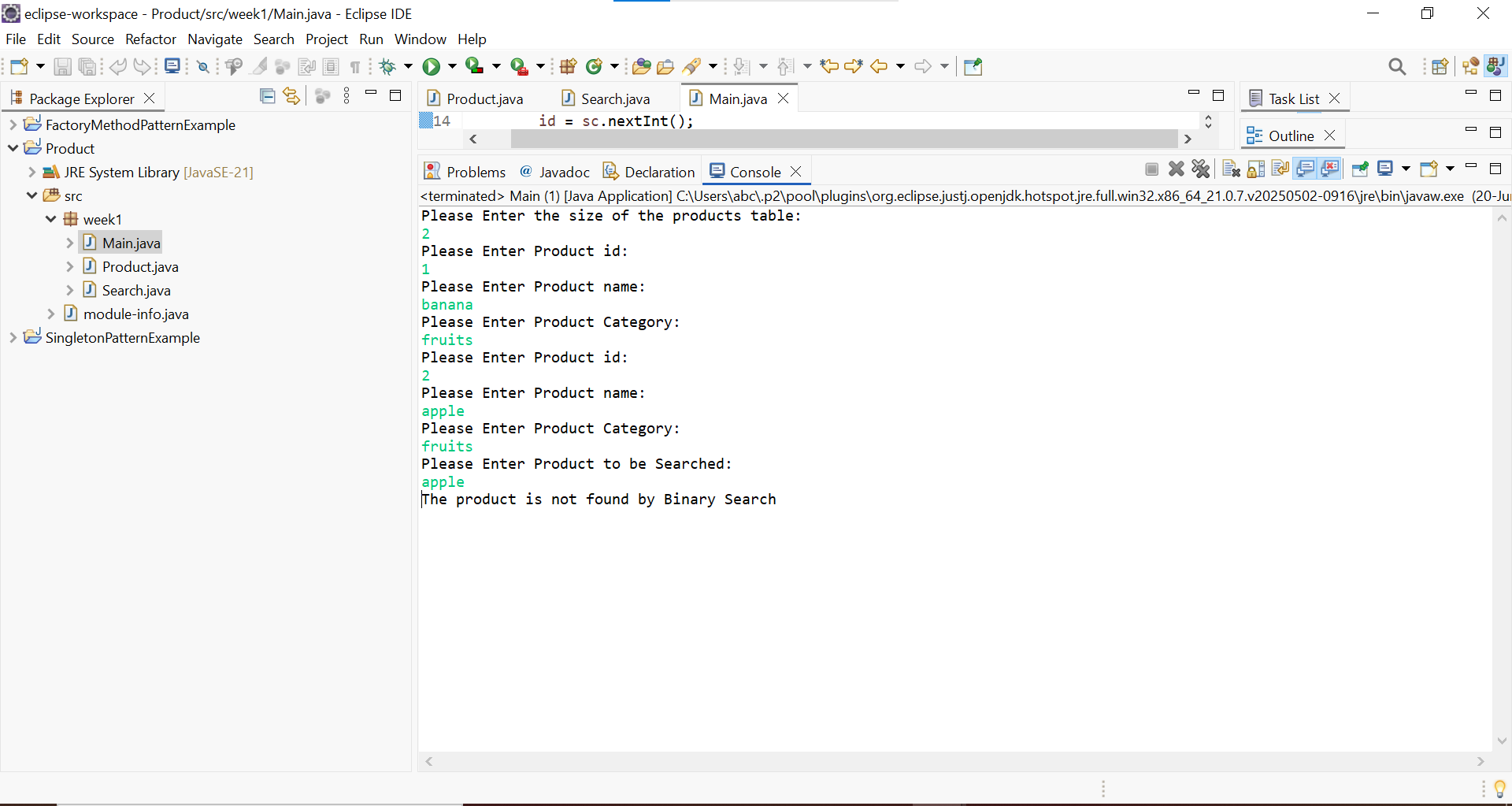
**The Input is Sorted and value is not present in the Array:**



**The Input is Sorted and value is present in the Array:**



**The Input is not Sorted and value is present in the Array:**



**Analysis:**

The Time Complexity for the **linear Search is O(n)**, whereas for the **Binary Search is**

**O(log n)**.  
For the ecommerce platform, in real time, the data will come dynamically, and the search operation must be faster, so for that purpose we must use **Binary Search,** but for this purpose the data needs to be stored in a sorted manner. If we manage to store it in a sorted manner we can use the Binary Search and our application will run faster.

**2**) **Exercise 7: Financial Forecasting**

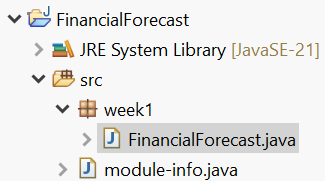
**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Project Structure:**

****

**Recursion:**

Recursion is a programming technique in which a function calls itself to solve smaller pieces of the same problem, breaking a complex task into more manageable steps. By defining a clear base case (the simplest instance of the problem) and a recursive case (how to reduce the problem toward the base case), we can often express efficient solutions that would otherwise require intricate loops or auxiliary data structures.

For example, id we want to traverse a tree data structure if we traverse a tree with loop, it requires another data structure (queue) but in recursion we can simply solve one instance of the problem, and it can be applied to the rest of the problem.

**Code:**

For solving this problem, I’ve assumed that the previous growth rates are in the Fibonacci sequence(fib(n) = fib(n-1) + fin(n-2)) and solved the problem. As Fibonacci depends on the previous values to determine the current value

package week1;

import java.util.\*;

public class FinancialForecast {

public static double forecast(double a, double b, int n) {

if (n == 0) return a;

if (n == 1) return b;

return *forecast*(a, b, n - 1) + *forecast*(a, b, n - 2);

}

public static void main(String[] args) {

double v1 = 10000; // Value at first year

double v2 = 11000; // Value at Second Year

// I've assumed the growth of the Business to be in Fibonacci Sequence

Scanner sc = new Scanner(System.*in*);

System.*out*.println("Please enter the year you want to see the Amount:");

int n = sc.nextInt();

double rs = *forecast*(v1, v2, n);

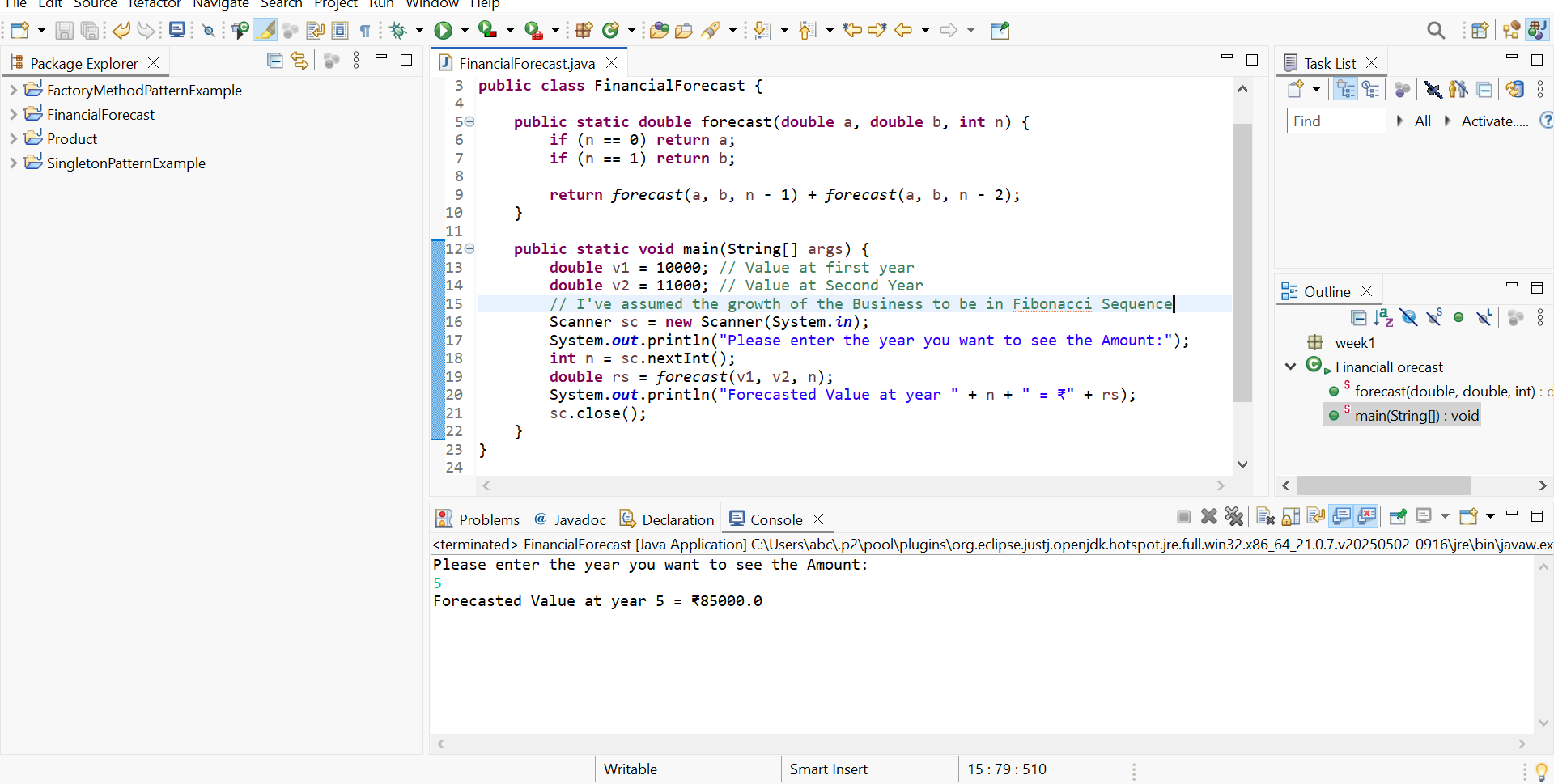
System.*out*.println("Forecasted Value at year " + n + " = ₹" + rs);

sc.close();

}

}

**Output:**



**Analysis:**

The time complexity for this problem can be calculated by using the recurrence relation  
**fib(n) = fib(n-1) + fib(n-2)**

So, for every n we have to solve two other sub problems, so at first level its two sub- problems then it increases to 4 and 8 and so on.

So the complexity of this becomes **O(2^n).**

The complexity of this problem can be reduced by using **Dynamic Programming**, where we store the calculated result in a data structure.