**SUPERSET ID:** 6374833

NAME:JAHNAVI NARAYANASETTY

**ALGORITHMS DATA STRUCTURES**

**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.

**Why Data Structures and Algorithms Are Essential:**

* A warehouse may store **thousands** of products.
* Operations like **searching, updating, adding, or deleting** items must be efficient.
* Proper use of data structures:
  + Reduces **retrieval time** (e.g., O(1) lookup in HashMap vs. O(n) in a List).
  + Ensures **faster updates and maintenance** of records.
  + Supports **scalability** as inventory size increases.

**Suitable Data Structures:**

| **Data Structure** | **Use Case** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| ArrayList | Small inventories, ordered data | Easy iteration | Slower search (O(n)), no fast lookup |
| HashMap | Large inventories, fast access by productId | O(1) average time for search/update | No order, uses more memory |
| TreeMap | Sorted inventories | Keeps sorted order | O(log n) for operations |
| LinkedList | When frequent insert/delete is needed | Fast insertion/deletion | Slow search |

**Best Choice:** HashMap<Integer, Product> — Fast retrieval and updates using productId as key.

1. **Setup:**
   * Create a new project for the inventory management system.
2. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.

**Code for the above problem statement:**

**1.Product.java**

public class Product {

int productId;

String productName;

int quantity;

double price;

public Product(int productId, String productName, int quantity, double price) {

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String toString() {

return "Product ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: " + price;

}

}

**2.InventoryManager.java**

import java.util.HashMap;

public class InventoryManager {

HashMap<Integer, Product> inventory = new HashMap<>();

public void addProduct(Product p) {

if (inventory.containsKey(p.productId)) {

System.out.println("Product already exists.");

} else {

inventory.put(p.productId, p);

System.out.println("Product added.");

}

}

public void updateProduct(int productId, int quantity, double price) {

if (inventory.containsKey(productId)) {

Product p = inventory.get(productId);

p.quantity = quantity;

p.price = price;

System.out.println("Product updated.");

} else {

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

}

}

public void deleteProduct(int productId) {

if (inventory.remove(productId) != null) {

System.out.println("Product deleted.");

} else {

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

}

}

public void displayInventory() {

if (inventory.isEmpty()) {

System.out.println("Inventory is empty.");

} else {

for (Product p : inventory.values()) {

System.out.println(p);

}

}

}

}

**3.Main.java**

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

Scanner scanner = new Scanner(System.in);

int choice;

do {

System.out.println("\nInventory Management Menu:");

System.out.println("1. Add Product");

System.out.println("2. Update Product");

System.out.println("3. Delete Product");

System.out.println("4. Display Inventory");

System.out.println("5. Exit");

System.out.print("Enter your choice: ");

choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.print("Enter Product ID: ");

int addId = scanner.nextInt();

scanner.nextLine();

System.out.print("Enter Product Name: ");

String name = scanner.nextLine();

System.out.print("Enter Quantity: ");

int qty = scanner.nextInt();

System.out.print("Enter Price: ");

double price = scanner.nextDouble();

Product p = new Product(addId, name, qty, price);

manager.addProduct(p);

break;

case 2:

System.out.print("Enter Product ID to update: ");

int updateId = scanner.nextInt();

System.out.print("Enter New Quantity: ");

int newQty = scanner.nextInt();

System.out.print("Enter New Price: ");

double newPrice = scanner.nextDouble();

manager.updateProduct(updateId, newQty, newPrice);

break;

case 3:

System.out.print("Enter Product ID to delete: ");

int deleteId = scanner.nextInt();

manager.deleteProduct(deleteId);

break;

case 4:

System.out.println("\nCurrent Inventory:");

manager.displayInventory();

break;

case 5:

System.out.println("Exiting program.");

break;

default:

System.out.println("Invalid choice! Try again.");

}

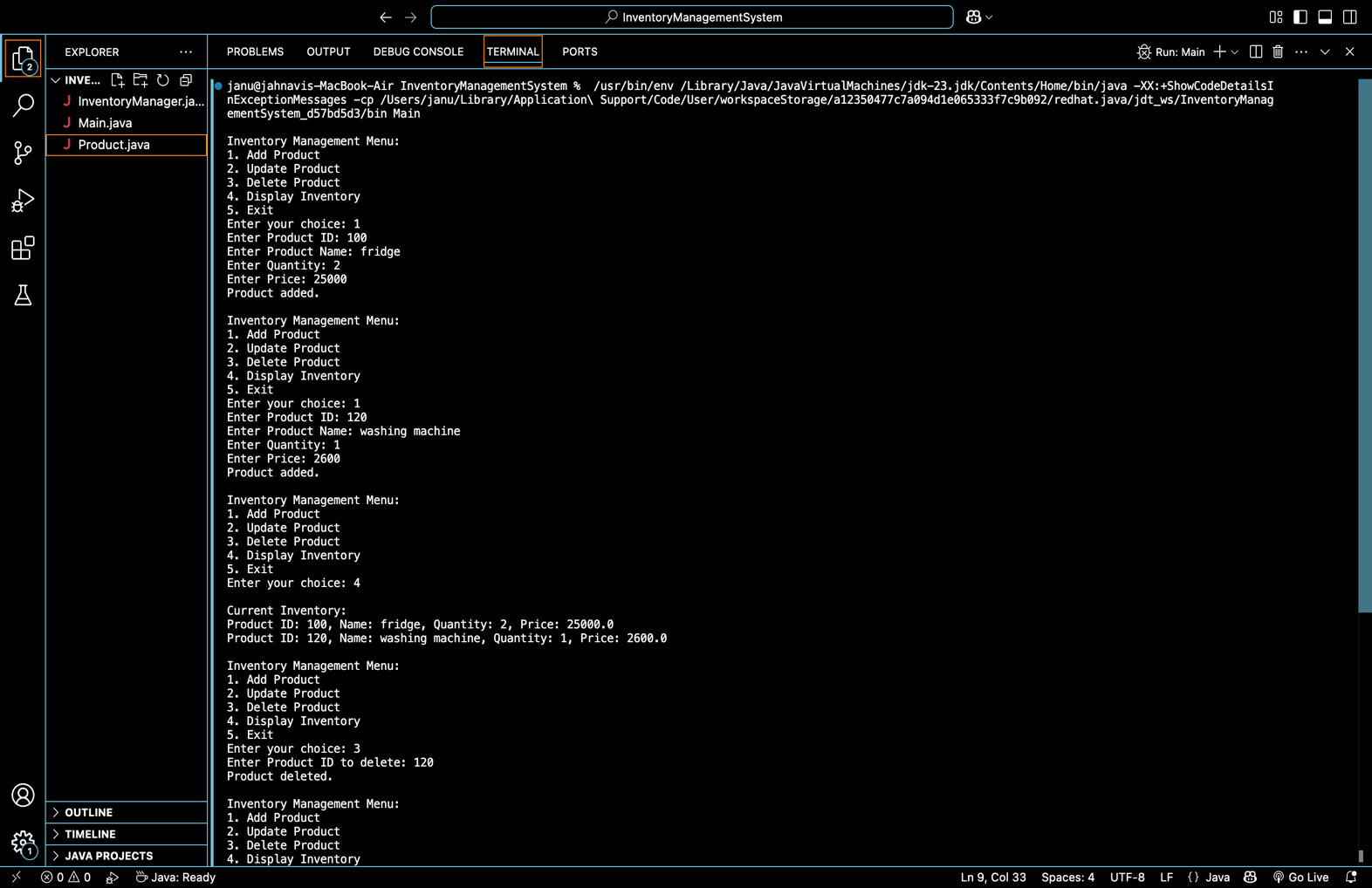
} while (choice != 5);

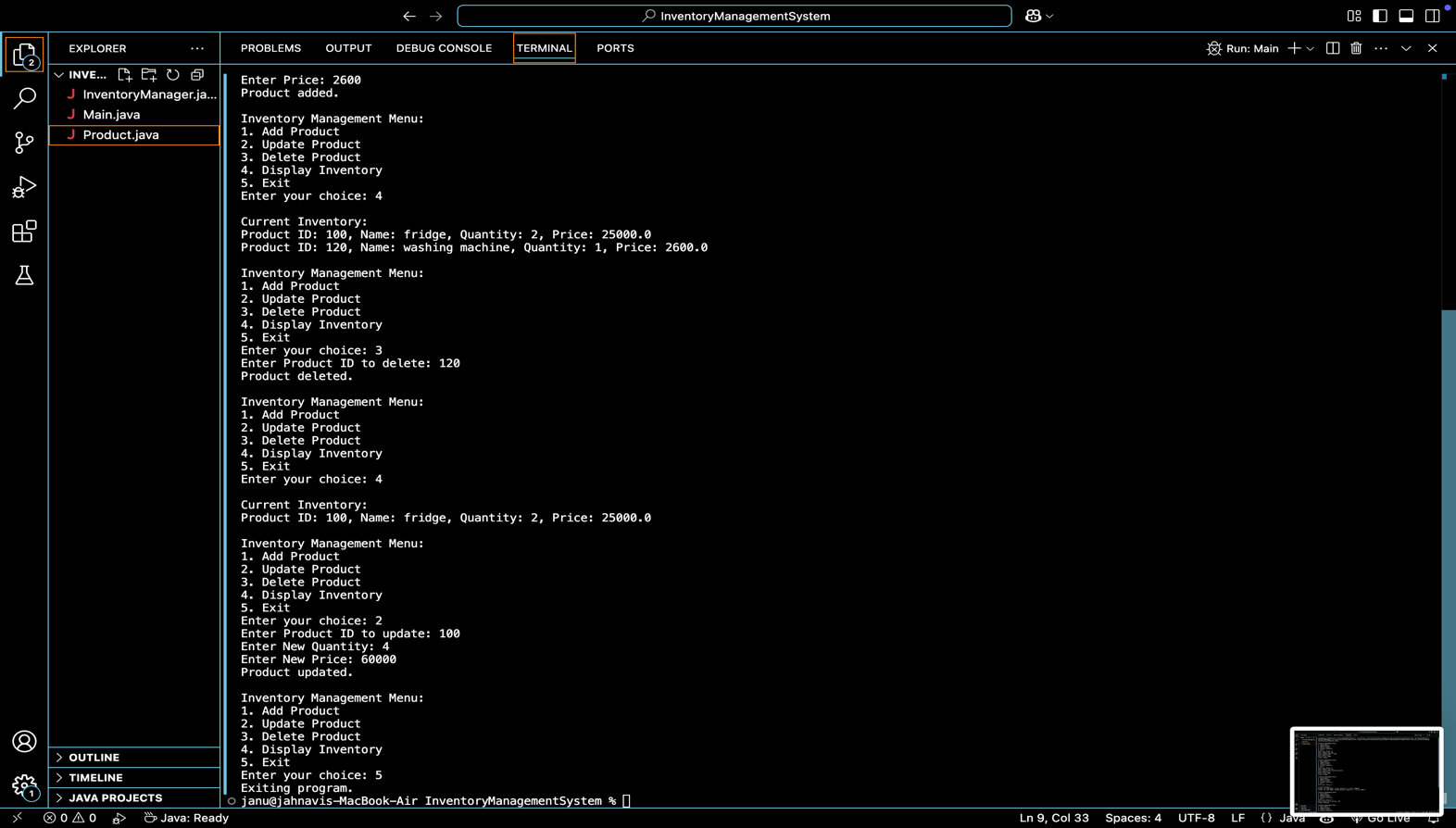
scanner.close();

}

}

OUTPUT:





**4.Analysis:**

* + Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
  + Discuss how you can optimize these operations

**Time Complexity (Using HashMap<Integer, Product>):**

| **Operation** | **Complexity** | **Explanation** |
| --- | --- | --- |
| Add product | O(1) average | Direct insertion by key |
| Update product | O(1) average | Direct lookup and modify |
| Delete product | O(1) average | Direct removal by key |
| Display inventory | O(n) | Must iterate through all values |

**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.

### ****Understand Asymptotic Notation****

#### Big O Notation

Big O notation describes the **worst-case time complexity** of an algorithm as input size n grows. It helps compare algorithm performance regardless of machine speed.

| **Algorithm** | **Big O Time Complexity** |
| --- | --- |
| Linear Search | O(n) – scans each item |
| Binary Search | O(log n) – divides the array each step |

#### Best, Average, Worst Case (for Searching)

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) (first match) | O(n/2) ≈ O(n) | O(n) (last or not found) |
| Binary Search | O(1) (middle) | O(log n) | O(log n) |

1. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
2. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.

**1.Product.java**

public class Product {

int productId;

String productName;

String category;

public Product(int id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

public String toString() {

return productId + " - " + productName + " (" + category + ")";

}

}

1. **SearchAlgorithms.java**

public class SearchAlgorithms {

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

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

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

return i;

}

}

return -1;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (compare == 0) return mid;

else if (compare < 0) left = mid + 1;

else right = mid - 1;

}

return -1;

}

public static void sortByName(Product[] products) {

int n = products.length;

for (int i = 0; i < n-1; i++) {

for (int j = 0; j < n-i-1; j++) {

if (products[j].productName.compareToIgnoreCase(products[j+1].productName) > 0) {

Product temp = products[j];

products[j] = products[j+1];

products[j+1] = temp;

}

}

}

}

}

**3.Main.java**

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

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

new Product(3, "Shirt", "Clothing"),

new Product(4, "Shoes", "Footwear"),

new Product(5, "Charger", "Accessories")

};

Scanner scanner = new Scanner(System.in);

System.out.print("Enter product name to search: ");

String searchName = scanner.nextLine();

System.out.println("\n-- Linear Search --");

int linearResult = SearchAlgorithms.linearSearch(products, searchName);

if (linearResult != -1)

System.out.println("Found: " + products[linearResult]);

else

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

)

SearchAlgorithms.sortByName(products);

System.out.println("\n-- Binary Search --");

int binaryResult = SearchAlgorithms.binarySearch(products, searchName);

if (binaryResult != -1)

System.out.println("Found: " + products[binaryResult]);

else

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

}

}

OUTPUT:



**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.

#### What is Recursion?

Recursion is a technique where a function **calls itself** to solve smaller instances of the same problem. It's useful for problems that can be broken down into similar subproblems (like growth calculations, factorials, etc.).

Example:  
To compute compound growth:

Copy code

FV(n) = FV(n-1) \* (1 + r)

Where:

* FV = future value
* r = growth rate (e.g., 0.1 for 10%)
* n = number of periods (e.g., years)

1. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
2. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.

### 1.Forecast.java

public class Forecast {

public static double futureValue(double initial, double rate, int years) {

if (years == 0) return initial;

return futureValue(initial, rate, years - 1) \* (1 + rate);

}

}

**2.Main.java**

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter initial investment amount: ");

double initial = sc.nextDouble();

System.out.print("Enter annual growth rate (in %): ");

double rate = sc.nextDouble() / 100;

System.out.print("Enter number of years: ");

int years = sc.nextInt();

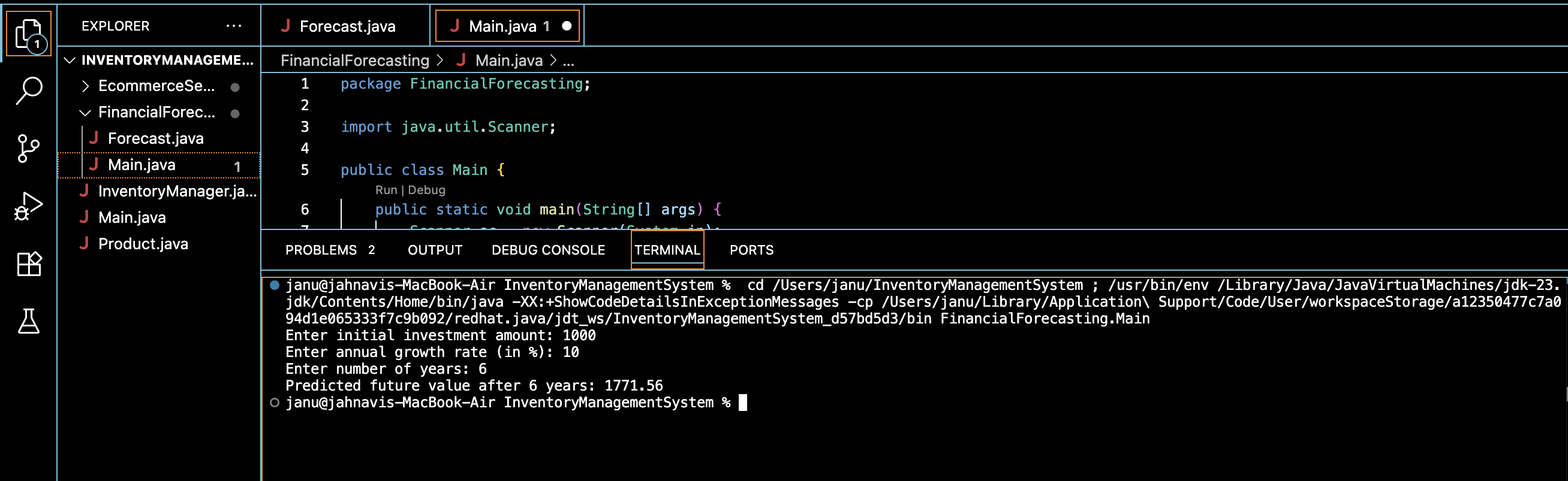
double result = Forecast.futureValue(initial, rate, years);

System.out.printf("Predicted future value after %d years: %.2f\n", years, result);

}

}

OUTPUT:



1. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

#### Time Complexity

* O(n) — one recursive call per year

#### Space Complexity

* Also O(n) due to recursive stack