**Exercise 1: Inventory Management System**

**Importance of Data Structures and Algorithms in Handling Large Inventories:**

Data structures and algorithms are essential in handling large inventories because they enable:

* Efficient data management
* Fast data retrieval
* Automated inventory tracking and optimization
* Scalability and improved performance

This leads to increased efficiency, accuracy, and customer satisfaction, while reducing costs.

**Suitable Data Structures:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Use Case** | **Pros** | **Cons** |
| ArrayList | If order matters, few deletions | Fast iteration | Slow lookup by ID |
| HashMap | When fast lookup by productId is needed | O(1) operations | Slightly more memory usage |
| TreeMap | If sorted productId list is needed | Sorted access | O(log n) operations |

**Code:**

**//Product.java**

public class Product {

private int productId;

private String productName;

private int quantity;

private 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 int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public int getQuantity() {

return quantity;

}

public double getPrice() {

return price;

}

public void setProductId(int productId) {

this.productId = productId;

}

public void setProductName(String productName) {

this.productName = productName;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public void setPrice(double price) {

this.price = price;

}

@Override

public String toString() {

return "Product{" +

"productId=" + productId +

", productName='" + productName + '\'' +

", quantity=" + quantity +

", price=" + price +

'}';

}

}

**//InventoryManager.java**

import java.util.\*;

public class InventoryManager {

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

public void addProduct(Product product) {

inventory.put(product.getProductId(), product);

}

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

Product product = inventory.get(productId);

if (product != null) {

product.setProductName(name);

product.setQuantity(quantity);

product.setPrice(price);

}

else {

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

}

}

public void deleteProduct(int productId) {

if (inventory.containsKey(productId)) {

inventory.remove(productId);

}

else {

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

}

}

public void viewAllProducts() {

if (inventory.isEmpty()) {

System.out.println("No products in inventory.");

} else {

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

System.out.println(product);

}

}

}

}

**//App.java**

public class App {

public static void main(String[] args) throws Exception {

InventoryManager manager = new InventoryManager();

Product p1 = new Product(101, "Laptop", 50, 70000);

Product p2 = new Product(102, "Mouse", 200, 500);

manager.addProduct(p1);

manager.addProduct(p2);

manager.viewAllProducts();

manager.updateProduct(101, "Gaming Laptop", 40, 85000);

manager.viewAllProducts();

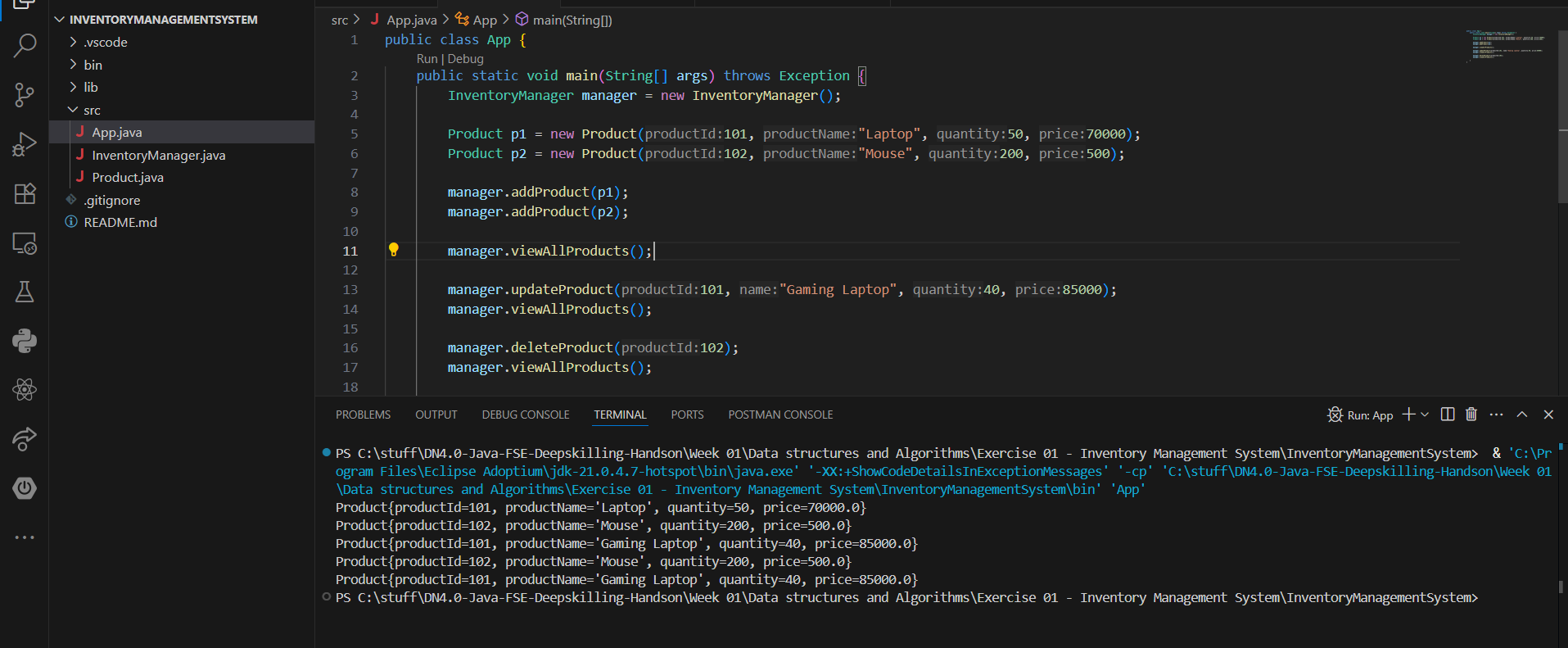
manager.deleteProduct(102);

manager.viewAllProducts();

}

}

**Output:**

****

**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Complexity** | **Reason** |
| addProduct() | O(1) | HashMap insertion |
| updateProduct() | O(1) | HashMap access and update |
| deleteProduct() | O(1) | HashMap deletion |
| viewAllProducts() | O(n) | Iterating over all products |

**Optimization:**

* Use TreeMap if **sorted order of productId** is important.
* For **frequent search by product name**, use an additional HashMap<String, List<Product>>

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

**Big O Notation:**

Big O notation describes the upper bound of an algorithm’s running time as the input size grows. It helps you understand how well your code will perform as your dataset scales.

* O(1) – Constant time
* O(log n) – Logarithmic time
* O(n) – Linear time
* O(n log n) – Linearithmic time
* O(n²) – Quadratic time

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| **Linear Search** | O(1) (found early) | O(n/2) ≈ O(n) | O(n) |
| **Binary Search** | O(1) (found at mid) | O(log n) | O(log n) |

Binary search is faster but requires a sorted array.

**Code:**

**//Product.java**

public class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

}

**//** **SearchUtility.java**

public class SearchUtility {

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

for (Product product : products) {

if (product.getProductId() == productId) {

return product;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

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

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**//App.java**

import java.util.Arrays;

import java.util.Comparator;

public class App {

public static void main(String[] args) throws Exception {

Product[] products = {

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

new Product(102, "Shirt", "Clothing"),

new Product(103, "Phone", "Electronics"),

new Product(104, "Shoes", "Footwear"),

new Product(105, "Watch", "Accessories" )

};

Product foundProduct = SearchUtility.linearSearch(products, "Shoes");

if (foundProduct != null) {

System.out.println("Linear Search Found: " + foundProduct.getProductName() + " in " + foundProduct.getCategory());

}

else {

System.out.println("Product not found using Linear Search.");

}

foundProduct = SearchUtility.linearSearch(products, "Laptop");

if (foundProduct != null) {

System.out.println("Linear Search Found: " + foundProduct.getProductName() + " in " + foundProduct.getCategory());

}

else {

System.out.println("Product not found using Linear Search.");

}

Arrays.sort(products, Comparator.comparing(Product::getProductName, String.CASE\_INSENSITIVE\_ORDER));

foundProduct = SearchUtility.binarySearch(products, "Shirt");

if (foundProduct != null) {

System.out.println("Binary Search Found: " + foundProduct.getProductName() + " in " + foundProduct.getCategory());

}

else {

System.out.println("Product not found using Binary Search.");

}

foundProduct = SearchUtility.binarySearch(products, "Headphones");

if (foundProduct != null) {

System.out.println("Binary Search Found: " + foundProduct.getProductName() + " in " + foundProduct.getCategory());

}

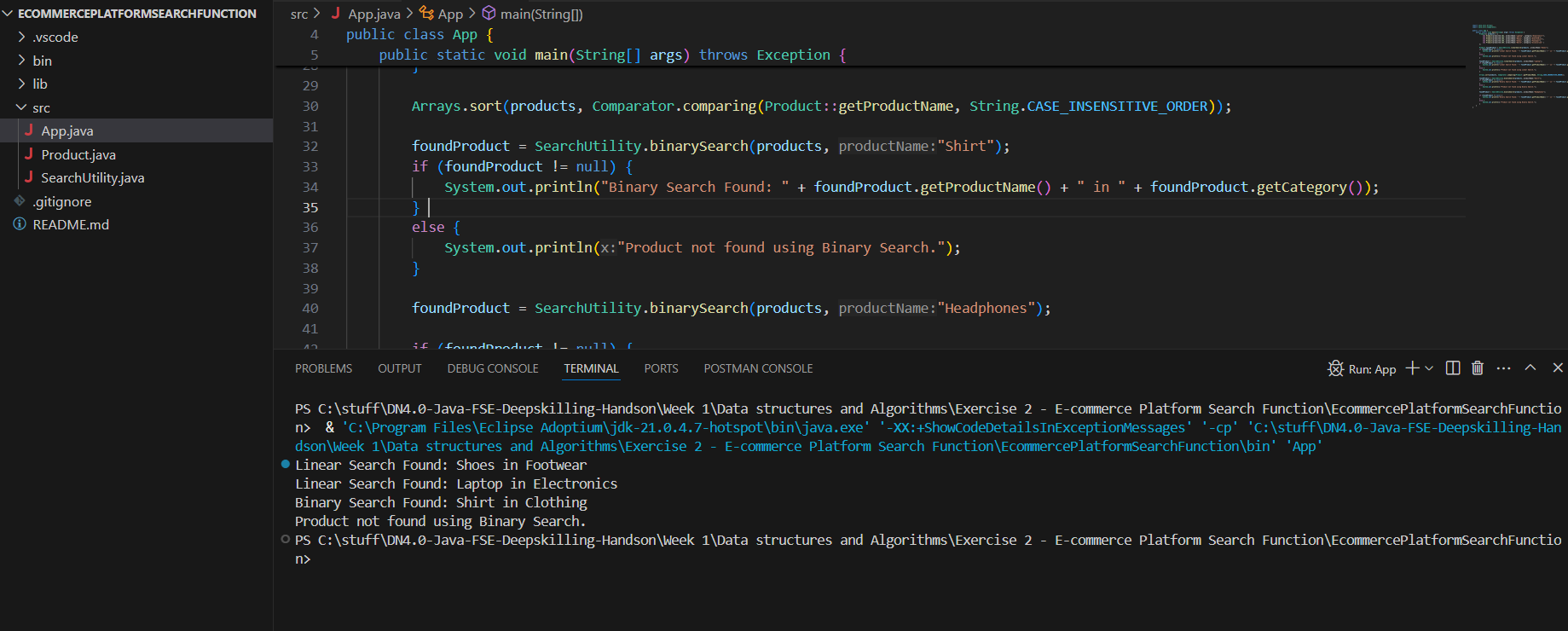
else {

System.out.println("Product not found using Binary Search.");

}

}

}

**Output:**

**Analysis:**

**Time Complexity:**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Data Requirement** |
| Linear Search | O(n) | Unsorted Array |
| Binary Search | O(log n) | Sorted Array |

**When to Use What?**

* Use linear search when:
  + The array is small.
  + Products are not sorted.
  + Real-time updates (insert/delete) are frequent.
* Use binary search when:
  + Data is large and relatively static.
  + You can afford to keep the array sorted.
  + You prioritize search speed.

**Exercise 7: Financial Forecasting**

**Recursion Concept:**

Recursion is a fundamental concept in programming where a function calls itself repeatedly until it reaches a base case that stops the recursion.

Key elements of recursion:

1. Base case: A condition that stops the recursion.
2. Recursive call: The function calls itself**.**

How recursion works:

1. The function is called with an initial input.
2. The function checks the base case. If true, it returns a value.
3. If not, the function calls itself with a modified input.
4. This process repeats until the base case is reached.

In Financial Forecasting:

We can recursively predict the **future value** using a fixed growth rate:

FVn​ = FVn−1 ​× (1+r)

Where:

* FVn = future value at year n
* r = annual growth rate
* Base case: FV0 = initial value

**Code:**

**//FinancialForecast.java**

import java.util.HashMap;

public class FinancialForecast {

public static double calculateFutureValue(double presentValue, double interestRate, int years) {

if(years==0) {

return presentValue;

}

return calculateFutureValue(presentValue, interestRate, years-1) \* (1 + interestRate);

}

public static double memoizedCalculateFutureValue(double presentValue, double interestRate, int years, HashMap<Integer, Double> memo) {

if(years == 0) {

return presentValue;

}

if (memo.containsKey(years)) {

return memo.get(years);

}

double futureValue = memoizedCalculateFutureValue(presentValue, interestRate, years-1, memo)\*(1+ interestRate);

memo.put(years, futureValue);

return futureValue;

}

}

**// App.java**

import java.util.HashMap;

public class App {

public static void main(String[] args) {

double presentValue = 6700.0;

double interestRate = 0.17;

int years = 10;

double futureValue = FinancialForecast.calculateFutureValue(presentValue, interestRate, years);

System.out.println("Future Value Calculation using basic Recursion:");

System.out.printf("Future Value after %d years: %.2f%n", years, futureValue);

HashMap <Integer, Double> memo = new HashMap<>();

futureValue = FinancialForecast.memoizedCalculateFutureValue(presentValue, interestRate, years, memo);

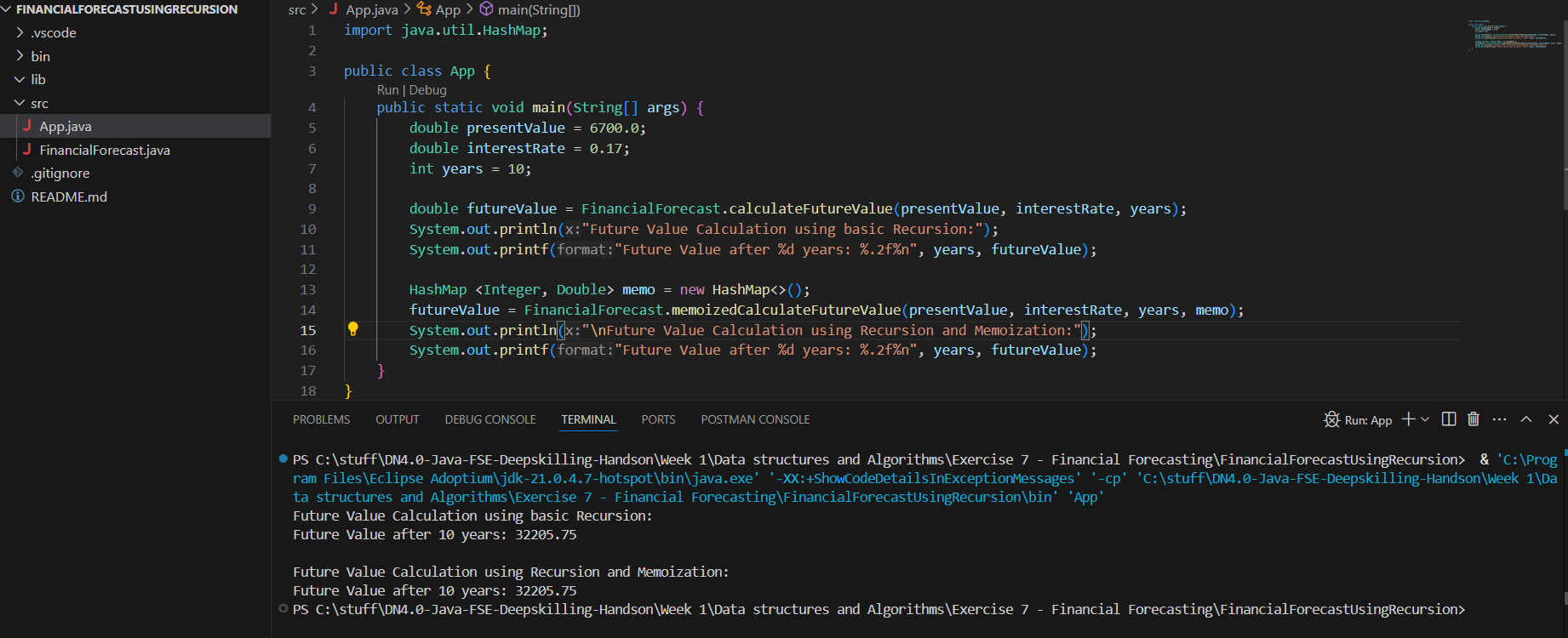
System.out.println("\nFuture Value Calculation using Recursion and Memoization:");

System.out.printf("Future Value after %d years: %.2f%n", years, futureValue);

}

}

**Output:**



**Analysis:**

**Time Complexity:**

Time Complexity: O(n) — since it makes a recursive call for each year.

Space Complexity: O(n) — due to function call stack.

**Optimization:**

Use Memoization (only useful for variable rates or overlapping subproblems): If growth rate changes per year, we can memoize the computed results.