**Exercise 1: Inventory Management System**

**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 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 ID: " + productId + ", Name: " + productName +

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

}

}

**Inventory.java:**

import java.util.HashMap;

public class Inventory {

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

public void addProduct(Product product) {

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

}

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

Product p = products.get(productId);

if (p != null) {

p.setProductName(name);

p.setQuantity(quantity);

p.setPrice(price);

} else {

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

}

}

public void deleteProduct(int productId) {

products.remove(productId);

}

public void displayAllProducts() {

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

System.out.println(p);

}

}

public Product getProduct(int productId) {

return products.get(productId);

}

}

**Main.java**

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Inventory inventory = new Inventory();

Scanner scanner = new Scanner(System.in);

int choice;

do {

System.out.println("\n--- Inventory Management ---");

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

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

System.out.println("3. Display All Products");

System.out.println("4. Exit");

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

choice = scanner.nextInt();

scanner.nextLine(); // consume newline

switch (choice) {

case 1:

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

int id = scanner.nextInt();

scanner.nextLine();

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

String name = scanner.nextLine();

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

int quantity = scanner.nextInt();

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

double price = scanner.nextDouble();

Product p = new Product(id, name, quantity, price);

inventory.addProduct(p);

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

break;

case 2:

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

int updateId = scanner.nextInt();

scanner.nextLine();

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

String newName = scanner.nextLine();

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

int newQty = scanner.nextInt();

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

double newPrice = scanner.nextDouble();

inventory.updateProduct(updateId, newName, newQty, newPrice);

break;

case 3:

System.out.println("Current Inventory:");

inventory.displayAllProducts();

break;

case 4:

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

break;

default:

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

}

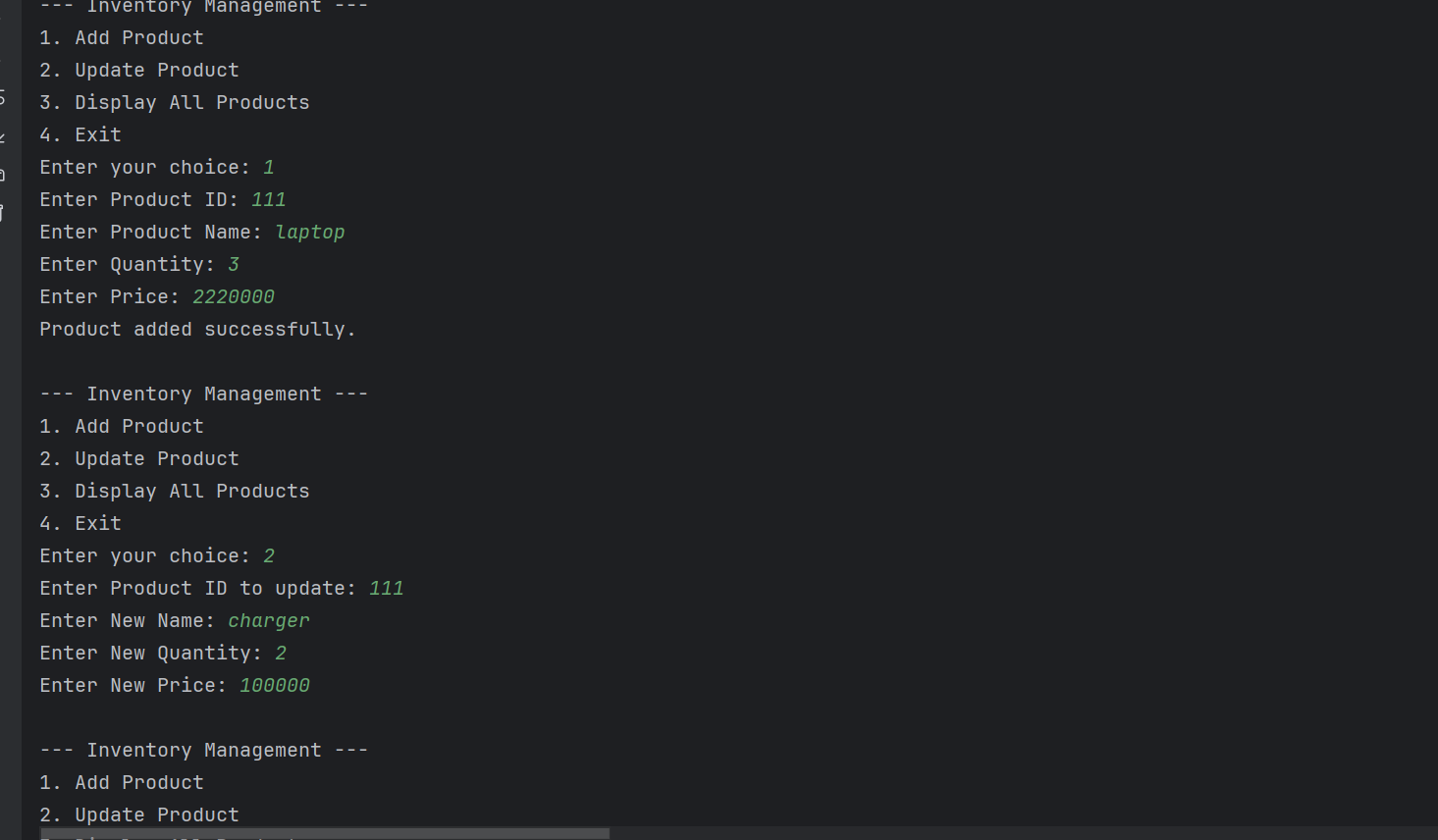
} while (choice != 4);

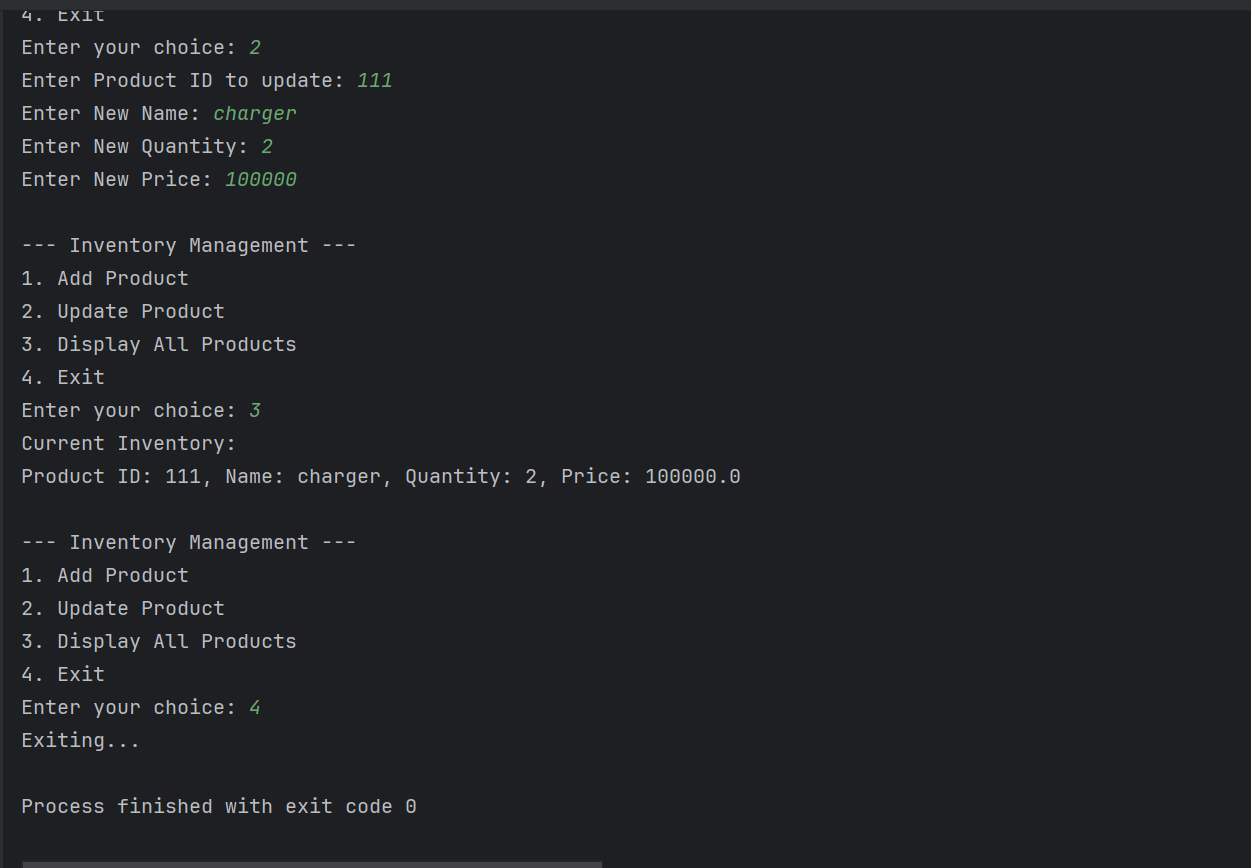
scanner.close();

}

}

**OUTPUT:**





**Time complexity :**

| **Operation** | ****Time Complexity**** |
| --- | --- |
| Add | ****O(1)** (amortized)** |
| Update | ****O(1)**** |
| Delete | ****O(1)**** |
| Search/Get | ****O(1)**** |
| Display all | ****O(n)**** |

**Optimization:**

### ****Add Product:**** Before adding, check if the product already exists to avoid duplicates.

if (!products.containsKey(productId)) {

products.put(productId, product);

} else {

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

}

### ****2. Update Product:****

Check if the product exists before updating it.

if (products.containsKey(productId)) {

Product p = products.get(productId);

p.setProductName(name);

p.setQuantity(quantity);

p.setPrice(price);

} else {

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

}

### ****3. Delete Product**** Check if the product is actually there before deleting.

if (products.containsKey(productId)) {

products.remove(productId);

System.out.println("Deleted successfully.");

} else {

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

}

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

**Explain Big O notation and how it helps in analyzing algorithm?**

Big O notation describes how the performance (time or space) of an algorithm grows with the input size n. It helps identify best algorithms for large datasets by focusing on the most significant term and ignoring constants.

It helps Compare algorithms ,Estimate **scalability,**Design **efficient systems.**

### Best, Average, and Worst-Case Scenarios:

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | O(1) (1st match) | O(n) | O(n) |
| **Binary Search** | O(1) (middle match) | O(log n) | O(log n) |

****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;**

**}**

**@Override**

**public String toString() {**

**return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;**

**}**

**}**

****LinearSearch.java:****

**public class LinearSearch {**

**public static Product search(Product[] products, String name) {**

**for (Product p : products) {**

**if (p.productName.equalsIgnoreCase(name)) {**

**return p;**

**}**

**}**

**return null;**

**}}**

****BinarySearch.java:****

**import java.util.Arrays;**

**import java.util.Comparator;**

**public class BinarySearch {**

**public static Product search(Product[] products, String name) {**

**Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase())); // Sort alphabetically**

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

**while (low <= high) {**

**int mid = (low + high) / 2;**

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

**if (cmp == 0) return products[mid];**

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

**else high = mid - 1;**

**}**

**return null;**

**}**

**}**

****Main.java:****

**public class Main {**

**public static void main(String[] args) {**

**Product[] products = {**

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

**new Product(102, "Shampoo", "Toothpaste"),**

**new Product(103, "Shoes", "Footwear"),**

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

**};**

**System.out.println("Linear Search Result:");**

**Product result1 = LinearSearch.search(products, "Shoes");**

**System.out.println(result1 != null ? result1 : "Product Not Found");**

**System.out.println("\nBinary Search Result:");**

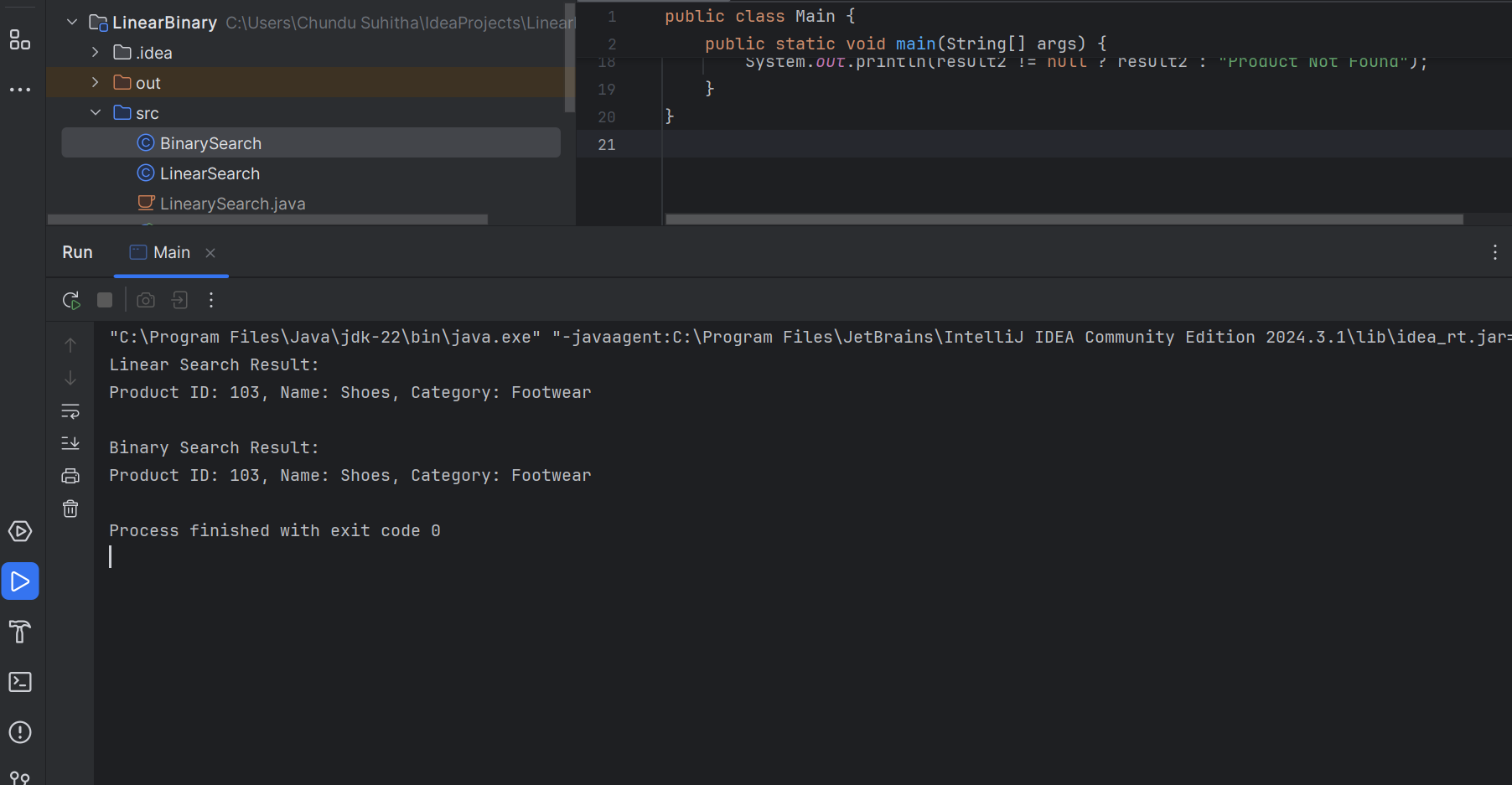
**Product result2 = BinarySearch.search(products, "Shoes");**

**System.out.println(result2 != null ? result2 : "Product Not Found");**

**}**

**}**

****OUTPUT:****



**Linear Search** – O(n):

Checks each item one by one

Slower for large datasets.

**Binary Search** – O(log n):

Repeatedly divides the list in half

Much faster for sorted data

**Exercise 3: Sorting Customer Orders:**

**Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort)?**

**Bubble Sort:** Bubble Sort is a simple comparison-based algorithm where adjacent elements are compared and swapped if they are in the wrong order. This process is repeated for all elements until the array is sorted. It's easy to understand but inefficient for large datasets due to its time complexity of **O(n²)** in average and worst cases.

**Insertion Sort:** Insertion Sort builds the final sorted array one element at a time. It takes each element and inserts it into its correct position among the previously sorted elements. It is efficient for small or nearly sorted datasets and has a best-case time complexity of **O(n)** but a worst-case of **O(n²)**.

**Quick Sort:** Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into two parts—elements less than the pivot and elements greater than the pivot. It then recursively sorts the subarrays. Quick Sort is efficient for large datasets with an average time complexity of **O(n log n)** but has a worst-case of **O(n²)** if the pivot is poorly chosen.

**Merge Sort:** Merge Sort is another divide-and-conquer algorithm that divides the array into halves, recursively sorts them, and then merges the sorted halves. It has a consistent time complexity of **O(n log n)** in all cases and is stable, but it uses additional memory space of **O(n)** for the temporary arrays used during merging.

**Order.java:**

public class Order {

int orderId;

String customerName;

double totalPrice;

public Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice; }

@Override

public String toString() {

return "Order ID: " + orderId + ", Customer: " + customerName + ", Total Price: ₹" + totalPrice; }}

**BubbleSort.java:**

class BubbleSort {

public static void sort(Order[] orders) {

int n = orders.length;

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

boolean swapped = false;

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

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break;}

}

}

**QuickSort.java:**

class QuickSort {

public static void sort(Order[] orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

sort(orders, low, pi - 1);

sort(orders, pi + 1, high);

} }

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = low - 1;

for (int j = low; j < high; j++) {

if (orders[j].totalPrice <= pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

}

**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 number of orders: ");

int n = sc.nextInt();

sc.nextLine();

Order[] orders = new Order[n];

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

System.out.println("Enter details for order " + (i + 1));

System.out.print("Order ID: ");

int id = sc.nextInt();

sc.nextLine();

System.out.print("Customer Name: ");

String name = sc.nextLine();

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

double price = sc.nextDouble();

sc.nextLine();

orders[i] = new Order(id, name, price);

}

System.out.println("\n🔽 Sorted Orders using Bubble Sort:");

BubbleSort.sort(orders);

for (Order o : orders) {

System.out.println(o);

}

System.out.println("\n🔽 Sorted Orders using Quick Sort:");

QuickSort.sort(orders, 0, n - 1);

for (Order o : orders) {

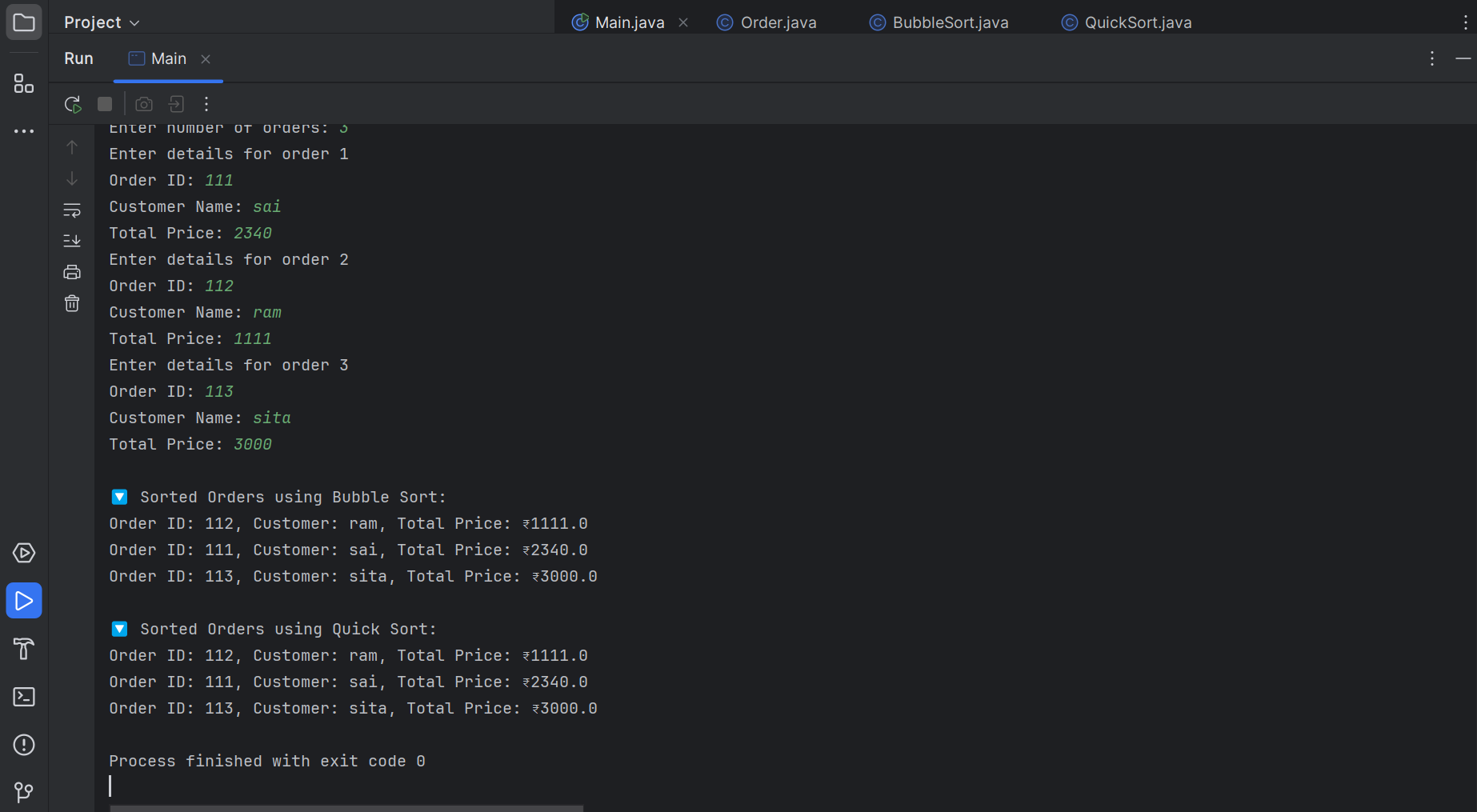
System.out.println(o);

}

sc.close(); }

}

**OUTPUT:**



|  |  |  |  |
| --- | --- | --- | --- |
| ****Operation**** | **Best Case** | **Average Case** | **Worst Case** |
| **Add** (at end) | O(1) | O(1) | O(1) |
| **Add** (at position) | O(1) | O(n) | O(n) |
| **Search** (unsorted) | O(1) | O(n) | O(n) |
| **Search** (sorted) | O(1) | O(log n) | O(log n) |
| **Traverse** | O(n) | O(n) | O(n) |
| **Delete** | O(1) | O(n) | O(n) |

### ****Limitations of Arrays****

1. **Fixed Size:**
   1. Arrays require a predefined size at creation.
   2. Cannot grow or shrink dynamically, leading to memory wastage or overflow.

**2.Inefficient Memory Use:**

* 1. May allocate more space than needed (static memory).
  2. May allocate more space than needed (static memory).
  3. May lead to unused slots.

****3.Lack of Flexibility:****

* 1. Can only store elements of the same data type.
  2. For heterogeneous data or complex structures, arrays become cumbersome.

### ****When to Use Arrays:****

* **Use arrays when:**
  + You know the fixed size in advance.
  + You need fast index-based access (random access is O(1)).
  + Memory usage is predictable.
  + You are working with primitive types and performance matters.
* **Avoid arrays when:**
  + You need dynamic resizing .
  + Insertions/deletions are frequent.
  + You want to store different data types or use key-based access (use Map or List).

**Exercise 5: Task Management System:**

**Task.java:**

public class Task {

int taskId;

String taskName;

String status;

public Task(int taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

@Override

public String toString() {

return "Task ID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

**Node.java:**

public class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

**TaskList.java:**

public class TaskList {

Node head;

public void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public Task searchTask(int id) {

Node temp = head;

while (temp != null) {

if (temp.task.taskId == id) {

return temp.task;

}

temp = temp.next;

}

return null;

}

public void deleteTask(int id) {

if (head == null) return;

if (head.task.taskId == id) {

head = head.next;

return;

}

Node prev = null;

Node curr = head;

while (curr != null && curr.task.taskId != id) {

prev = curr;

curr = curr.next; }

if (curr != null) {

prev.next = curr.next;

} else {

System.out.println("Task not found."); }

}

public void traverseTasks() {

if (head == null) {

System.out.println("Task list is empty.");

return; }

Node temp = head;

while (temp != null) {

System.out.println(temp.task);

temp = temp.next;

}

}}

**Main.java:**

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

TaskList taskList = new TaskList();

Scanner sc = new Scanner(System.in);

while (true) {

System.out.println("\n1. Add Task\n2. Search Task\n3. Delete Task\n4. View All Tasks\n5. Exit");

System.out.print("Choose: ");

int choice = sc.nextInt();

sc.nextLine();

switch (choice) {

case 1:

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

int id = sc.nextInt();

sc.nextLine();

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

String name = sc.nextLine();

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

String status = sc.nextLine();

taskList.addTask(new Task(id, name, status));

break;

case 2:

System.out.print("Enter Task ID to search: ");

int sid = sc.nextInt();

Task found = taskList.searchTask(sid);

System.out.println(found != null ? found : "Task Not Found");

break;

case 3:

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

int did = sc.nextInt();

taskList.deleteTask(did);

break;

case 4:

taskList.traverseTasks();

break;

case 5:

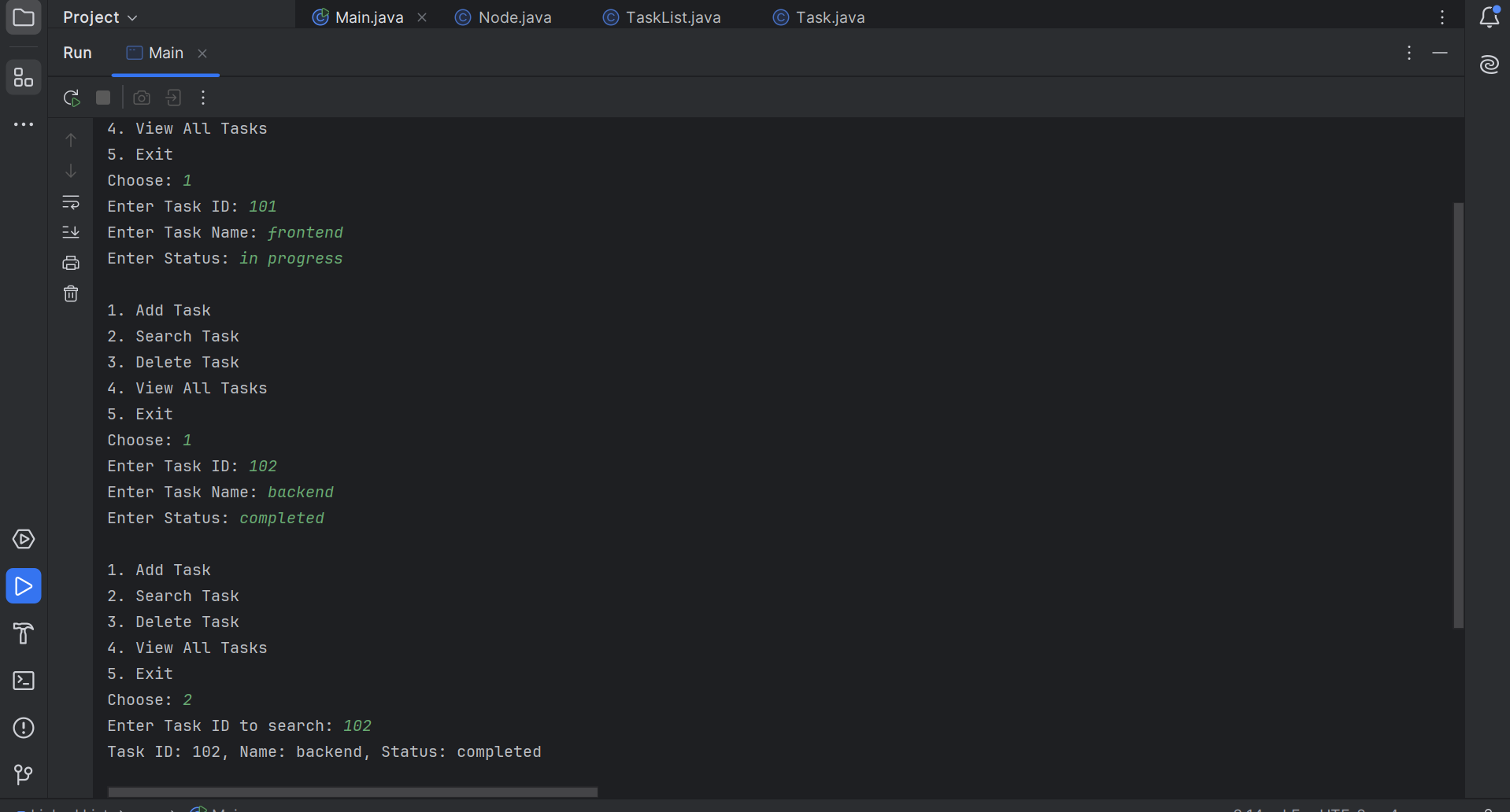
System.out.println("Exiting..."); return;

default:

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

} }}}

**Output:**



**Exercise 6: Library Management System**

**Book.java:**

**public class Book {**

**int bookId;**

**String title;**

**String author;**

**public Book(int bookId, String title, String author) {**

**this.bookId = bookId;**

**this.title = title;**

**this.author = author;**

**}**

**@Override**

**public String toString() {**

**return "Book ID: " + bookId + ", Title: " + title + ", Author: " + author;**

**}**

**}**

****LinearSearch.java:****

**class LinearSearch {**

**public static Book search(Book[] books, String title) {**

**for (Book b : books) {**

**if (b.title.equalsIgnoreCase(title)) {**

**return b; }**

**}**

**return null; }**

**}**

****BinarySearch.java:****

**import java.util.Arrays;**

**import java.util.Comparator;**

**class BinarySearch {**

**public static Book search(Book[] books, String title) {**

**Arrays.sort(books, Comparator.comparing(b -> b.title.toLowerCase()));**

**int low = 0, high = books.length - 1;**

**while (low <= high) {**

**int mid = (low + high) / 2;**

**int cmp = books[mid].title.compareToIgnoreCase(title);**

**if (cmp == 0) return books[mid];**

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

**else high = mid - 1;**

**}**

**return null;**

**}**

**}**

****Main.java:****

**import java.util.Scanner;**

**public class Main {**

**public static void main(String[] args) {**

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

**Book[] books = {**

**new Book(101, "The Cognizant", "Paulo Coelho"),**

**new Book(102, "1984", "George Orwell"),**

**new Book(103, "Python programming", "Harper Lee"),**

**new Book(104, "Amazing", "Herman Melville")**

**};**

**System.out.print("Enter book title to search (Linear): ");**

**String title1 = sc.nextLine();**

**Book result1 = LinearSearch.search(books, title1);**

**System.out.println(result1 != null ? result1 : "Book not found.");**

**System.out.print("\nEnter book title to search (Binary): ");**

**String title2 = sc.nextLine();**

**Book result2 = BinarySearch.search(books, title2);**

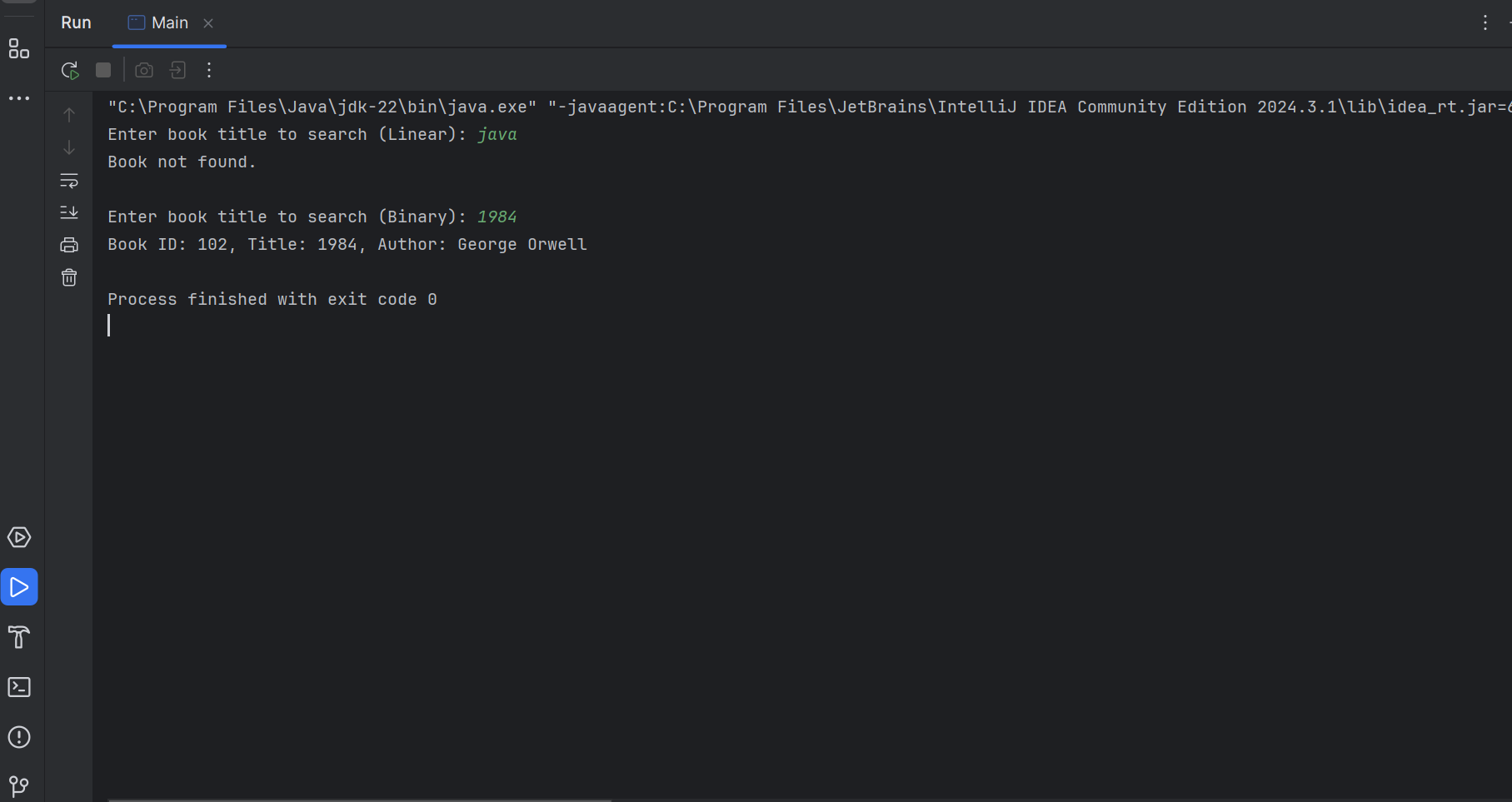
**System.out.println(result2 != null ? result2 : "Book not found.");**

**sc.close();**

**}**

**}**

****OUTPUT:****



**Exercise 7: Financial Forecasting:**

**FinancialForecast.java:**

import java.util.Scanner;

public class FinancialForecast {

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

if (years == 0) {

return currentValue;

}

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

}

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

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

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

double initialValue = sc.nextDouble();

System.out.print("Enter annual growth rate (as a percentage): ");

double rate = sc.nextDouble();

double growthRate = rate / 100;

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

int years = sc.nextInt();

double futureRecursive = forecastRecursive(initialValue, growthRate, years);

double futureIterative = forecastIterative(initialValue, growthRate, years);

System.out.printf("\n🔁 Recursive Forecast Value after %d years: ₹%.2f", years, futureRecursive);

System.out.printf("\n🔄 Iterative Forecast Value after %d years: ₹%.2f\n", years, futureIterative);

sc.close();

}

}

**OUTPUT:**

