**WEEK-1**

**Data Structures and Algorithms**

**Exercise 1: Inventory Management System**

**Problem Understanding:**  
In warehouses, managing thousands of items requires efficient data handling. Operations like searching, updating, and deletion must be quick to maintain real-time inventory status.

**Data Structure Used:**  
I used a **HashMap** to store product details using productId as the key. This provides average-case time complexity of **O(1)** for most operations like insert, search, and delete.

**Code Implementation Summary:**

* Product class encapsulates details (ID, name, quantity, price).
* Inventory class handles add, update, delete, and display operations using HashMap.

**Time Complexity:**

* add() – O(1)
* update() – O(1)
* remove() – O(1)
* showAll() – O(n)

**Optimization Discussion:**  
HashMap was ideal for fast access. For sorted outputs, a TreeMap could be considered but with O(log n) cost.

**Code:**

**Java Project (Package)** :Inventory Management System

**Product.java:**

import java.util.HashMap;

import java.util.Map;

class Product {

    private final String id;

    private String name;

    private int qty;

    private double cost;

    public Product(String id, String name, int qty, double cost) {

        this.id = id;

        this.name = name;

        this.qty = qty;

        this.cost = cost;

    }

    public String getId() { return id; }

    public void setName(String name) { this.name = name; }

    public void setQty(int qty) { this.qty = qty; }

    public void setCost(double cost) { this.cost = cost; }

    public String toString() {

        return "\nProductId: "+id + " \nName: " + name + " \nQuantity:" + qty + " \n Cost: RS" + cost;

    }

}

class Inventory {

    private final Map<String, Product> stock = new HashMap<>();

    public void add(Product p) {

        stock.put(p.getId(), p);

    }

    public void update(String id, String name, int qty, double cost) {

        Product p = stock.get(id);

        if (p != null) {

            p.setName(name);

            p.setQty(qty);

            p.setCost(cost);

        }

    }

    public void remove(String id) {

        stock.remove(id);

    }

    public void showAll() {

        if (stock.isEmpty()) {

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

            return;

        }

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

            System.out.println(p);

        }

    }

}

**Main.java:**public class Main {

    public static void main(String[] args) {

        Inventory inv = new Inventory();

        inv.add(new Product("P101", "Mouse", 100, 499.99));

        inv.add(new Product("P102", "Keyboard", 50, 899.50));

        inv.showAll();

        inv.update("P101", "Wireless Mouse", 80, 549.00);

        inv.remove("P102");

        inv.showAll();

    }

}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

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

**Big O Understanding:**  
Big O notation helps in analyzing algorithm efficiency based on input size.

**Implemented Algorithms:**

* **Linear Search** on unsorted array: O(n)
* **Binary Search** on sorted array: O(log n)

**Code Summary:**

* Store class maintains unsorted and sorted arrays.
* Search is demonstrated using both methods for productName.

**Analysis:**

* Linear Search: Best when data is unsorted or small.
* Binary Search: Best for large sorted datasets.

**Conclusion:**  
Binary search is preferred for faster lookups on large product lists, provided the data is sorted.

**Code:**

**Java Project (Package)** :E Commerce Platform

**Product.java:**

import java.util.Arrays;

import java.util.Comparator;

class Product {

    String productId;

    String productName;

    String category;

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

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    public String toString() {

        return productId + " | " + productName + " | " + category;

    }

}

class Store {

    Product[] unsorted;

    Product[] sorted;

    int size;

    Store(int capacity) {

        unsorted = new Product[capacity];

        sorted = new Product[capacity];

        size = 0;

    }

    void insert(Product p) {

        if (size < unsorted.length) {

            unsorted[size] = p;

            sorted[size] = p;

            size++;

        }

    }

    void sortByName() {

        Arrays.sort(sorted, 0, size, Comparator.comparing(o -> o.productName));

    }

    Product linearSearch(String name) {

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

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

                return unsorted[i];

            }

        }

        return null;

    }

    Product binarySearch(String name) {

        int low = 0, high = size - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

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

            if (cmp == 0) return sorted[mid];

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

            else high = mid - 1;

        }

        return null;

    }

    void displayAll() {

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

            System.out.println(unsorted[i]);

        }

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

        Store s = new Store(10);

        s.insert(new Product("A101", "Laptop", "Electronics"));

        s.insert(new Product("A102", "Sneakers", "Footwear"));

        s.insert(new Product("A103", "Phone", "Electronics"));

        s.insert(new Product("A104", "Shirt", "Clothing"));

        s.sortByName();

        Product found1 = s.linearSearch("Sneakers");

        System.out.println();

        System.out.println(found1 != null ? "\nLinear Found: " + found1 : "Not Found");

        Product found2 = s.binarySearch("Phone");

        System.out.println(found2 != null ? "\nBinary Found: " + found2 : "Not Found");

    }

}

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Exercise 3: Sorting Customer Orders**

**Sorting Algorithms Implemented:**

* **Bubble Sort** – Simple but inefficient (O(n²))
* **Quick Sort** – Efficient divide-and-conquer (Best: O(n log n))

**Code Design:**

* OrderSorter implements both sorting algorithms.
* Sorting is based on totalPrice of orders.

**Performance:**

* Bubble Sort is okay for small datasets but not scalable.
* Quick Sort is better for e-commerce systems handling high-volume orders.

**Conclusion:**  
Quick Sort is more efficient and should be the default in real-world applications.

**Code:**

**Java Project (Package)** :Sorting Customer Orders

**Order.java:**

class Order {

    String orderId;

    String customerName;

    double totalPrice;

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

        this.orderId = orderId;

        this.customerName = customerName;

        this.totalPrice = totalPrice;

    }

    public String toString() {

        return orderId + " | " + customerName + " | $" + totalPrice;

    }

}

class OrderSorter {

    void bubble(Order[] data) {

        int n = data.length;

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

            boolean swapped = false;

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

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

                    Order temp = data[j];

                    data[j] = data[j + 1];

                    data[j + 1] = temp;

                    swapped = true;

                }

            }

            if (!swapped) break;

        }

    }

    void quick(Order[] arr, int left, int right) {

        if (left >= right) return;

        double pivot = arr[(left + right) / 2].totalPrice;

        int i = left, j = right;

        while (i <= j) {

            while (arr[i].totalPrice < pivot) i++;

            while (arr[j].totalPrice > pivot) j--;

            if (i <= j) {

                Order t = arr[i];

                arr[i] = arr[j];

                arr[j] = t;

                i++; j--;

            }

        }

        quick(arr, left, j);

        quick(arr, i, right);

    }

    void display(Order[] data) {

        for (Order o : data) System.out.println(o);

    }

}

**Main.java:**

import java.util.Arrays;

public class Main {

    public static void main(String[] args) {

        Order[] orders = {

            new Order("O101", "Alice", 299.99),

            new Order("O102", "Bob", 149.50),

            new Order("O103", "Charlie", 899.00),

            new Order("O104", "Diana", 499.75)

        };

        OrderSorter sorter = new OrderSorter();

        System.out.println("Using Bubble Sort:");

        Order[] bubbleSorted = Arrays.copyOf(orders, orders.length);

        sorter.bubble(bubbleSorted);

        sorter.display(bubbleSorted);

        System.out.println("\nUsing Quick Sort:");

        Order[] quickSorted = Arrays.copyOf(orders, orders.length);

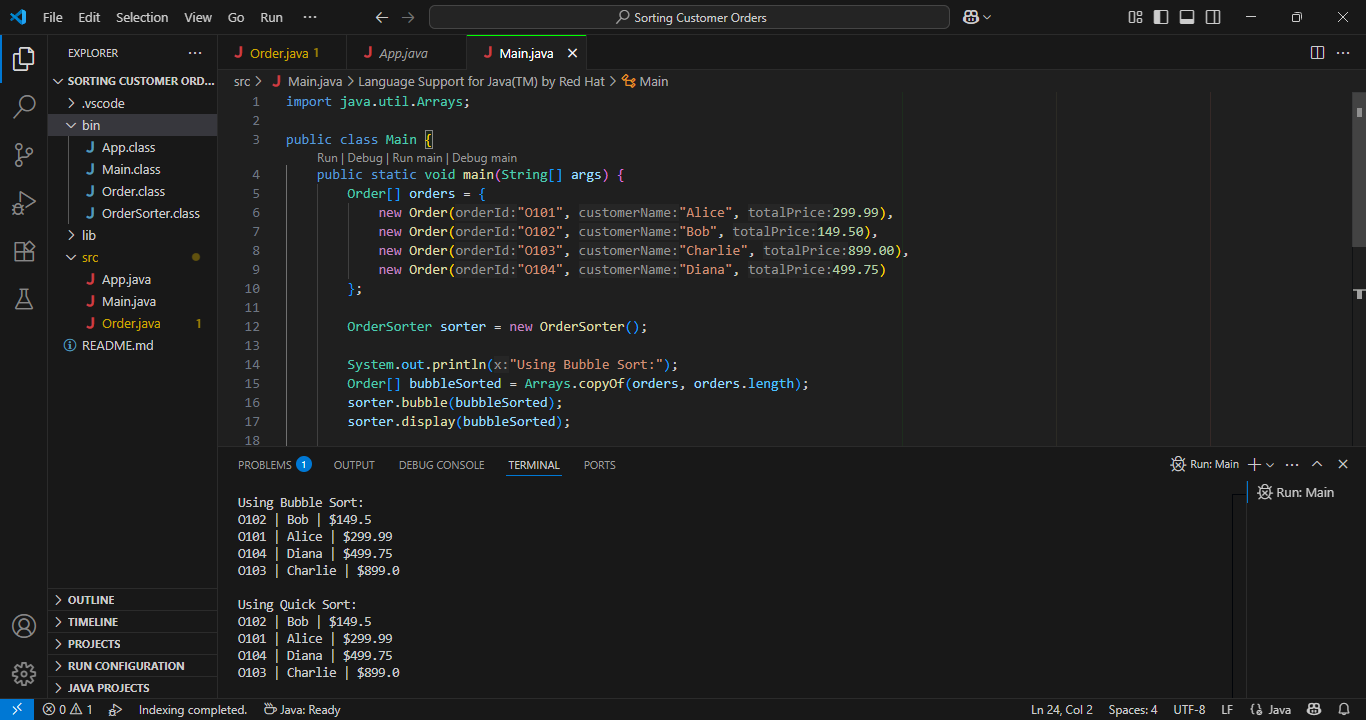
        sorter.quick(quickSorted, 0, quickSorted.length - 1);

        sorter.display(quickSorted);

    }

}

**Output:**

****

**Exercise 4: Employee Management System**

**Array Use Case:**  
I used an array to store fixed-size employee records. Arrays provide fast index-based access (**O(1)**), which is useful in limited capacity systems.

**Key Operations:**

* Add: O(1)
* Search: O(n)
* Delete: O(n)
* Traverse: O(n)

**Limitation of Arrays:**

* Fixed size – can’t scale dynamically.
* Insertion and deletion are costly due to element shifting.

**When to Use:**  
Best suited when the number of employees is small and fixed. For dynamic records, an ArrayList or LinkedList would be better.

**Code:**

**Java Project (Package)** :Employee Management System

**Employee.java:**

class Employee {

    String employeeId;

    String name;

    String position;

    double salary;

    Employee(String employeeId, String name, String position, double salary) {

        this.employeeId = employeeId;

        this.name = name;

        this.position = position;

        this.salary = salary;

    }

    public String toString() {

        return employeeId + " | " + name + " | " + position + " | $" + salary;

    }

}

class StaffRegistry {

    private Employee[] records;

    private int count;

    StaffRegistry(int capacity) {

        records = new Employee[capacity];

        count = 0;

    }

    boolean add(Employee e) {

        if (count >= records.length) return false;

        records[count++] = e;

        return true;

    }

    Employee get(String empId) {

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

            if (records[i].employeeId.equals(empId)) return records[i];

        }

        return null;

    }

    void showAll() {

        if (count == 0) {

            System.out.println("No employees available.");

            return;

        }

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

            System.out.println(records[i]);

        }

    }

    boolean remove(String empId) {

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

            if (records[i].employeeId.equals(empId)) {

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

                    records[j] = records[j + 1];

                }

                records[--count] = null;

                return true;

            }

        }

        return false;

    }

}

**Main.java:**public class Main {

    public static void main(String[] args) {

StaffRegistry sr = new StaffRegistry(5);

        sr.add(new Employee("E001", "John", "Manager", 75000));

        sr.add(new Employee("E002", "Sara", "Developer", 60000));

        sr.add(new Employee("E003", "David", "Analyst", 50000));

        System.out.println("Employee Records:");

        sr.showAll();

        System.out.println("\nSearching for E002:");

        Employee found = sr.get("E002");

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

        System.out.println("\nRemoving E001:");

        boolean success = sr.remove("E001");

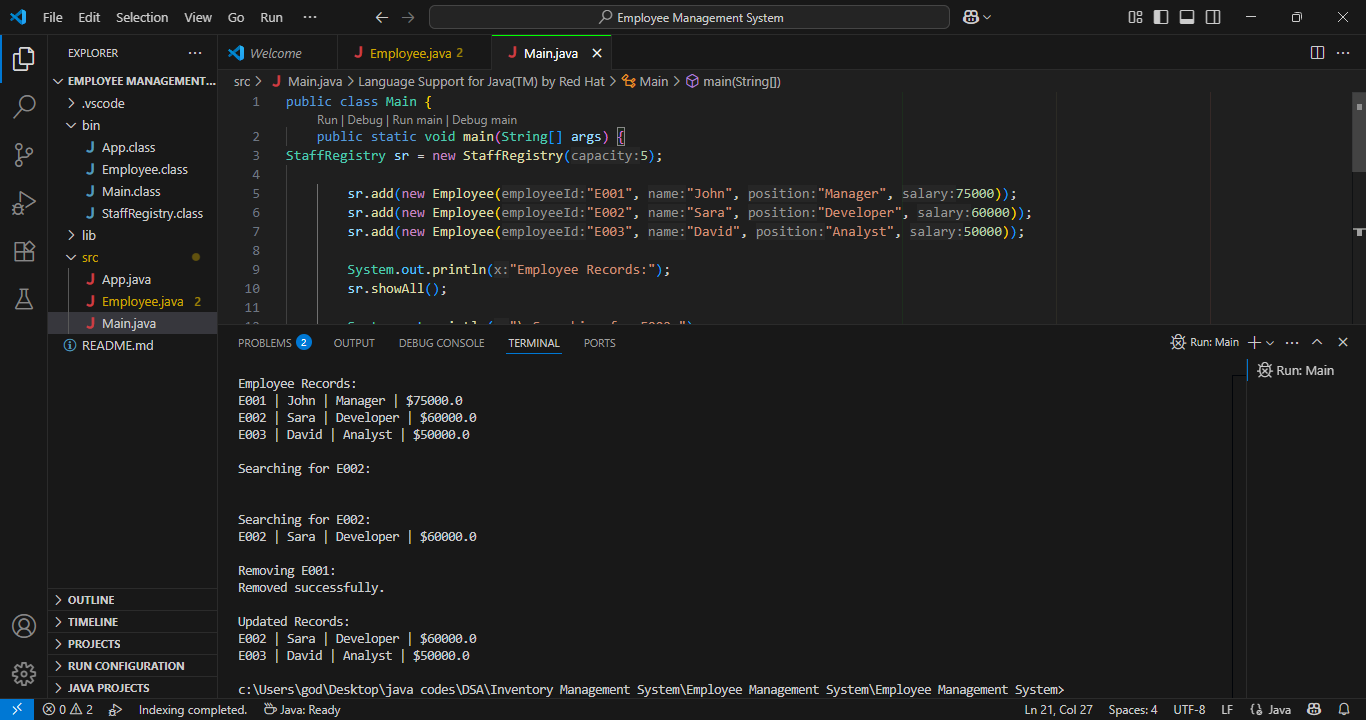
        System.out.println(success ? "Removed successfully." : "Employee not found.");

        System.out.println("\nUpdated Records:");

        sr.showAll();    }

}

**Output:**

****

**Exercise 5: Task Management System**

**Linked List Used:**  
Implemented using a **singly linked list** which supports efficient insertions and deletions.

**Advantages:**

* Dynamic size
* O(1) insertion at head
* No need to shift elements

**Time Complexity:**

* Add: O(n) (at end)
* Search: O(n)
* Delete: O(n)

**Conclusion:**  
Linked lists are more efficient than arrays when dealing with dynamic tasks and frequent modifications.

**Code:**

**Java Project (Package)** :Sorting Customer Orders

**Task.java:**

class Task {

    String taskId;

    String taskName;

    String status;

    Task next;

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

        this.taskId = taskId;

        this.taskName = taskName;

        this.status = status;

        this.next = null;

    }

    public String toString() {

        return "[" + taskId + "] " + taskName + " - " + status;

    }

}

class TaskList {

    private Task head;

    public void add(String taskId, String taskName, String status) {

        Task t = new Task(taskId, taskName, status);

        if (head == null) {

            head = t;

        } else {

            Task walk = head;

            while (walk.next != null) walk = walk.next;

            walk.next = t;

        }

    }

    public boolean remove(String taskId) {

        if (head == null) return false;

        if (head.taskId.equals(taskId)) {

            head = head.next;

            return true;

        }

        Task prev = head, curr = head.next;

        while (curr != null) {

            if (curr.taskId.equals(taskId)) {

                prev.next = curr.next;

                return true;

            }

            prev = curr;

            curr = curr.next;

        }

        return false;

    }

    public Task find(String taskId) {

        Task node = head;

        while (node != null) {

            if (node.taskId.equals(taskId)) return node;

            node = node.next;

        }

        return null;

    }

    public void show() {

        if (head == null) {

            System.out.println("No tasks in list.");

            return;

        }

        Task node = head;

        while (node != null) {

            System.out.println(node);

            node = node.next;

        }

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

     TaskList tl = new TaskList();

        tl.add("T001", "Design UI", "Pending");

        tl.add("T002", "Write Backend", "Ongoing");

        tl.add("T003", "Test System", "Pending");

        System.out.println("All Tasks:");

        tl.show();

        System.out.println("\nSearching for Task T002:");

        Task t = tl.find("T002");

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

        System.out.println("\nRemoving Task T001:");

        boolean removed = tl.remove("T001");

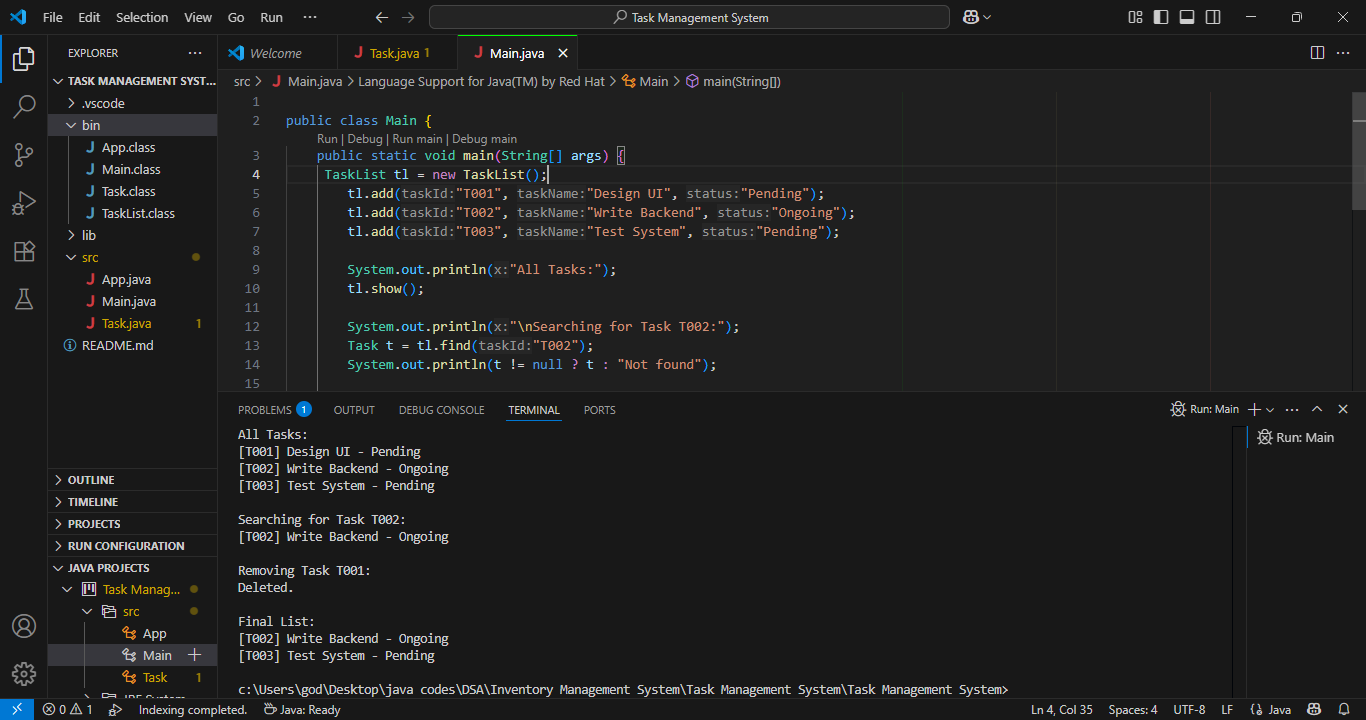
        System.out.println(removed ? "Deleted." : "Not Found.");

        System.out.println("\nFinal List:");

        tl.show();

    }

}

**Output:  
**

**Exercise 6: Library Management System**

**Search Algorithms Used:**

* **Linear Search**: Simple, no need to sort – O(n)
* **Binary Search**: Requires sorted data – O(log n)

**Code Flow:**

* BookShelf maintains book data.
* Searching is done on both unsorted and sorted collections.

**Recommendation:**

* Use Linear Search when the dataset is small or changes frequently.
* Use Binary Search when working with large and stable datasets.

**Code:**

**Java Project (Package)** :Library Management System

**Book.java:**

import java.util.Arrays;

import java.util.Comparator;

class Book {

    String bookId;

    String title;

    String author;

    Book(String bookId, String title, String author) {

        this.bookId = bookId;

        this.title = title;

        this.author = author;

    }

    public String toString() {

        return bookId + " | " + title + " | " + author;

    }

}

class BookShelf {

    Book[] books;

    int count;

    BookShelf(int capacity) {

        books = new Book[capacity];

        count = 0;

    }

    void insert(Book b) {

        if (count < books.length) {

            books[count++] = b;

        }

    }

    Book findLinear(String keyTitle) {

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

            if (books[i].title.equalsIgnoreCase(keyTitle)) {

                return books[i];

            }

        }

        return null;

    }

    void sortByTitle() {

        Arrays.sort(books, 0, count, Comparator.comparing(b -> b.title.toLowerCase()));

    }

    Book findBinary(String keyTitle) {

        int low = 0, high = count - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int result = books[mid].title.compareToIgnoreCase(keyTitle);

            if (result == 0) return books[mid];

            if (result < 0) low = mid + 1;

            else high = mid - 1;

        }

        return null;

    }

    void displayAll() {

        if (count == 0) {

            System.out.println("No books found.");

            return;

        }

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

            System.out.println(books[i]);

        }

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

        BookShelf shelf = new BookShelf(10);

        shelf.insert(new Book("B101", "The Alchemist", "Paulo Coelho"));

        shelf.insert(new Book("B102", "Invisible Man", "Ralph Ellison"));

        shelf.insert(new Book("B103", "Clean Code", "Robert Martin"));

        shelf.insert(new Book("B104", "To Kill a Mockingbird", "Harper Lee"));

        System.out.println("Available Books:");

        shelf.displayAll();

        System.out.println("\nLinear Search for 'Clean Code':");

        Book found1 = shelf.findLinear("Clean Code");

        System.out.println(found1 != null ? found1 : "Book not found.");

        System.out.println("\nBinary Search for 'Invisible Man':");

        shelf.sortByTitle();

        Book found2 = shelf.findBinary("Invisible Man");

        System.out.println(found2 != null ? found2 : "Book not found.");

    }

}

**Output:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Exercise 7: Financial Forecasting**

**Concept:**  
Used recursion to calculate projected future value based on growth rate.

**Optimization:**

* Basic recursive method: O(n) time and space
* Optimized version: Uses **memoization** to store previously computed results and avoid redundant work.

**Conclusion:**  
Recursion simplifies code, but for performance-critical tasks, memoization or iterative logic should be preferred.

**Code:**

**Java Project (Package)** :Financial Forecasting

**ForecastTool.java:**

class ForecastTool {

    double project(double base, double rate, int years) {

        if (years == 0) return base;

        return project(base, rate, years - 1) \* (1 + rate);

    }

    double optimizedProject(double base, double rate, int years, double[] cache) {

        if (years == 0) return base;

        if (cache[years] != 0) return cache[years];

        return cache[years] = optimizedProject(base, rate, years - 1, cache) \* (1 + rate);

    }

    void displayForecast(double base, double rate, int period) {

        System.out.println("Year-wise Financial Forecast:");

        for (int y = 0; y <= period; y++) {

            double value = project(base, rate, y);

            System.out.printf("Year %d: %.2f%n", y, value);

        }

    }

    void displayOptimizedForecast(double base, double rate, int period) {

        System.out.println("\nOptimized Forecast:");

        double[] cache = new double[period + 1];

        for (int y = 0; y <= period; y++) {

            double value = optimizedProject(base, rate, y, cache);

            System.out.printf("Year %d: %.2f%n", y, value);

        }

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

        ForecastTool tool = new ForecastTool();

        double initialValue = 10000.0;

        double annualGrowth = 0.07;

        int years = 10;

        tool.displayForecast(initialValue, annualGrowth, years);

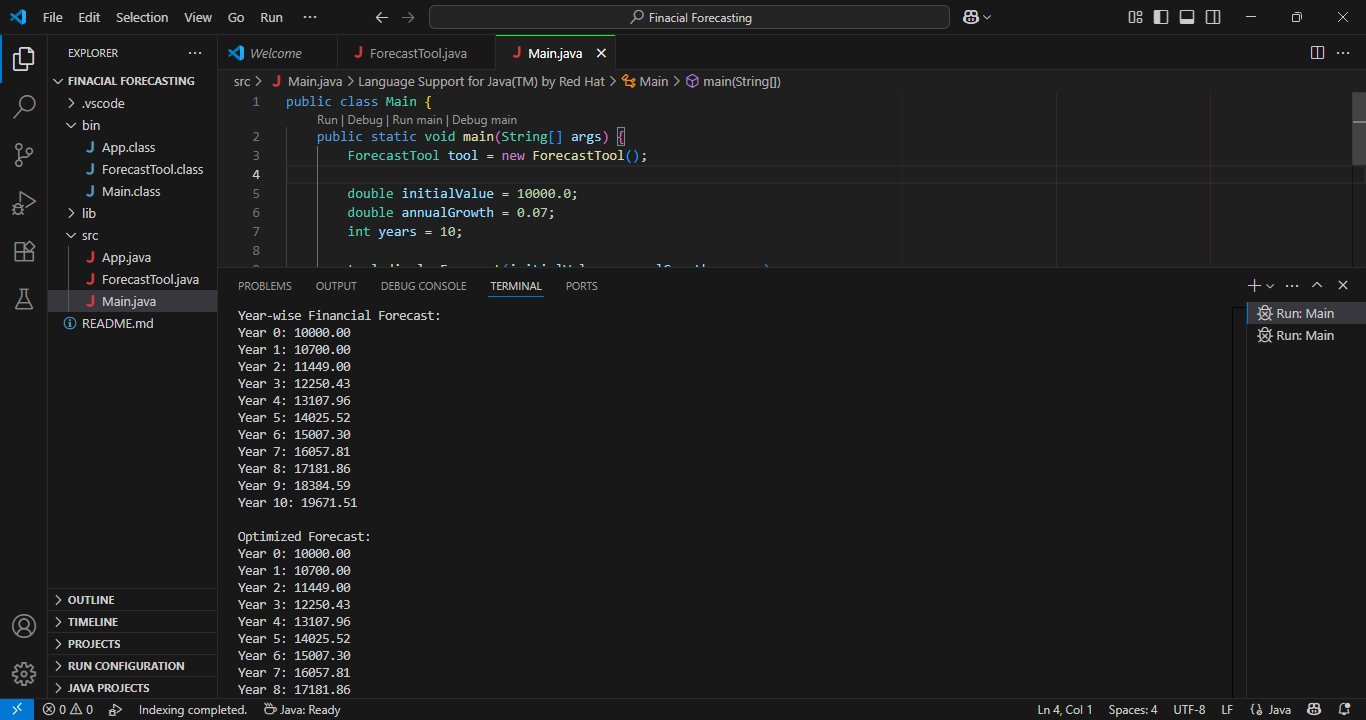
        tool.displayOptimizedForecast(initialValue, annualGrowth, years);

    }

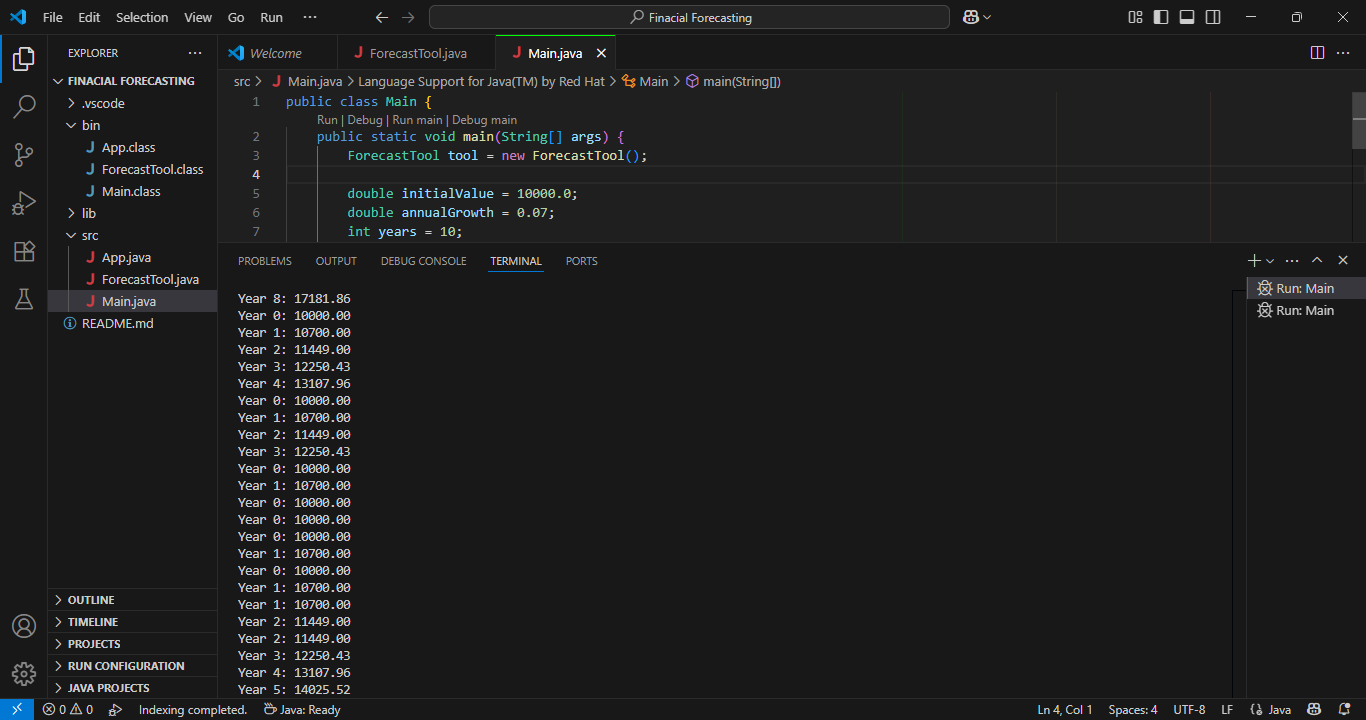
}

**Output:**

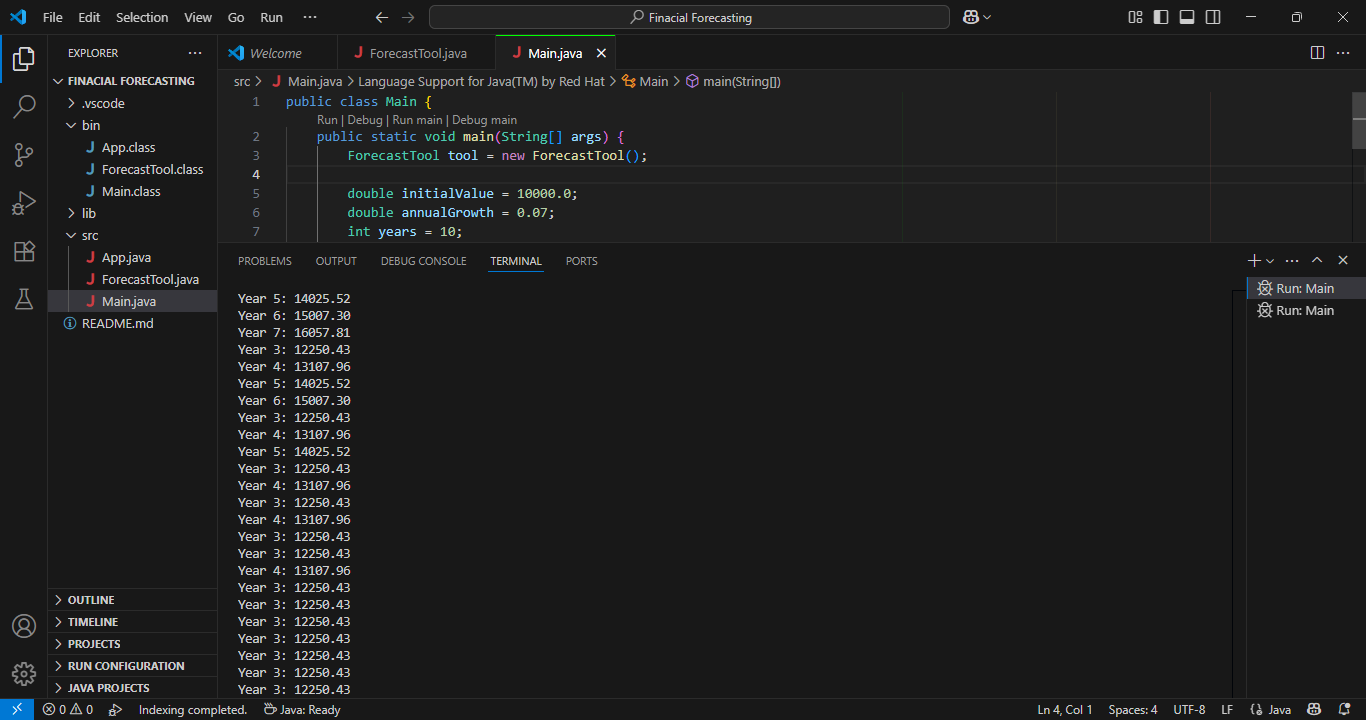
**1.**

****

**2.**

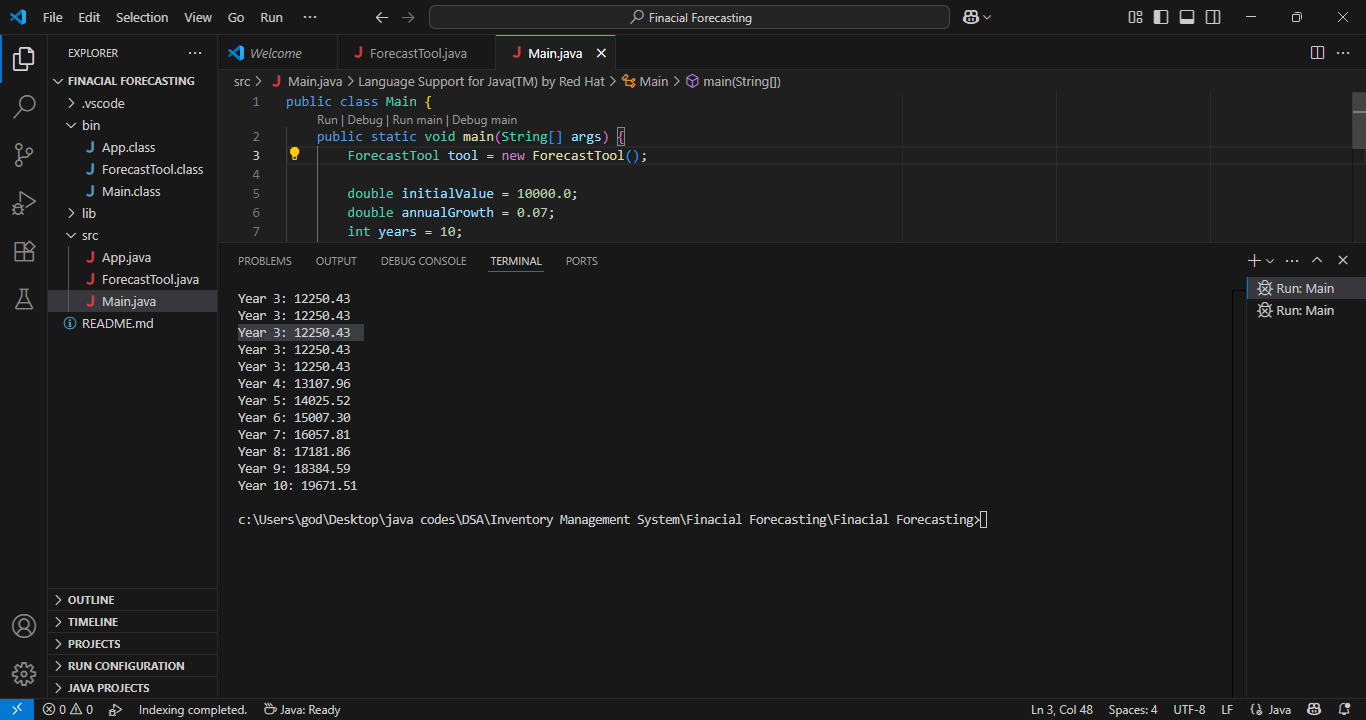
****

**3.**

****

**P.T.O**

**4.**

****