**Week-01 HandsOn Solution**

* **Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

* **Code:**

using System;

public class Logger{

    private static Logger? instance;

private static readonly object lockObj = new object();

private Logger() { }

    public static Logger Instance

    {

        get

        {

            lock (lockObj)

            {

                if (instance == null)

                    instance = new Logger();

                return instance;

            }

        }

}

    public void Log(string message){

        Console.WriteLine("Log: " + message);

    }

}

public class Program{

    public static void Main()

    {

        Logger log1 = Logger.Instance;

        Logger log2 = Logger.Instance;

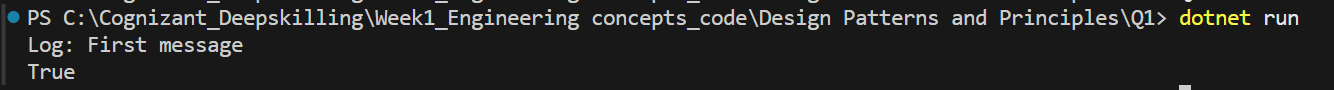
        log1.Log("First message");

        Console.WriteLine(log1 == log2);

    }

}

* **Output:**



**Exercise 2: Implementing the Factory Method Pattern**

* **Code:**

using System;

interface IDocument{

    void Open();

}

class WordDocument : IDocument{

    public void Open() => Console.WriteLine("Word document opened.");

}

class PdfDocument : IDocument{

    public void Open() => Console.WriteLine("PDF document opened.");

}

abstract class DocumentFactory{

    public abstract IDocument CreateDocument();

}

class WordFactory : DocumentFactory{

    public override IDocument CreateDocument() => new WordDocument();

}

class PdfFactory : DocumentFactory{

    public override IDocument CreateDocument() => new PdfDocument();

}

public class Program{

    public static void Main(){

        DocumentFactory factory = new PdfFactory();

        IDocument doc = factory.CreateDocument();

        doc.Open();

    }

}

* **Output:**



**Exercise 3: Implementing the Builder Pattern**

* **Code:**

using System;

class Computer{

    public string CPU { get; private set; }

    public string RAM { get; private set; }

public string Storage { get; private set; }

private Computer() { }

    public class Builder{

        private Computer computer = new Computer();

        public Builder SetCPU(string cpu) { computer.CPU = cpu; return this; }

        public Builder SetRAM(string ram) { computer.RAM = ram; return this; }

        public Builder SetStorage(string storage) { computer.Storage = storage; return this; }

        public Computer Build() => computer;

    }

}

public class Program{

    public static void Main(){

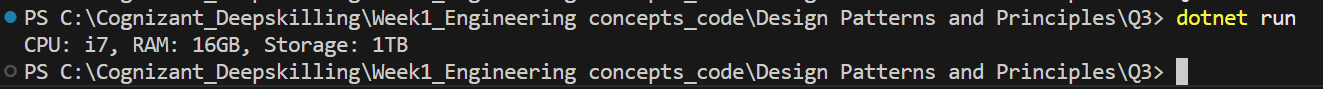
        var pc = new Computer.Builder().SetCPU("i7").SetRAM("16GB").SetStorage("1TB").Build();

        Console.WriteLine($"CPU: {pc.CPU}, RAM: {pc.RAM}, Storage: {pc.Storage}");

    }

}

* **Output:**



**Exercise 4: Implementing the Adapter Pattern**

* **Code:**

using System;

interface IPaymentProcessor{

void ProcessPayment();

}

class OldPaymentGateway{

public void MakePayment() => Console.WriteLine("Payment done using old gateway.");

}

class PaymentAdapter : IPaymentProcessor{

private OldPaymentGateway gateway = new OldPaymentGateway();

public void ProcessPayment() => gateway.MakePayment();

}

public class Program{

public static void Main(){

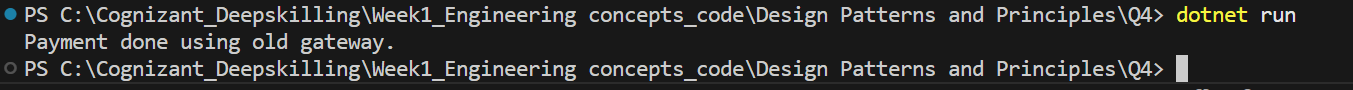
IPaymentProcessor processor = new PaymentAdapter();

processor.ProcessPayment();

}

}

* **Output:**



**Exercise 5: Implementing the Decorator Pattern**

* **Code:**

using System;

interface INotifier{

    void Send(string message);

}

class EmailNotifier : INotifier{

    public void Send(string message) => Console.WriteLine("Email: " + message);

}

abstract class NotifierDecorator : INotifier{

    protected INotifier wrappee;

    public NotifierDecorator(INotifier notifier) { wrappee = notifier; }

    public virtual void Send(string message) => wrappee.Send(message);

}

class SMSNotifier : NotifierDecorator{

    public SMSNotifier(INotifier notifier) : base(notifier) { }

    public override void Send(string message)

    {

        base.Send(message);

        Console.WriteLine("SMS: " + message);

    }

}

public class Program{

    public static void Main(){

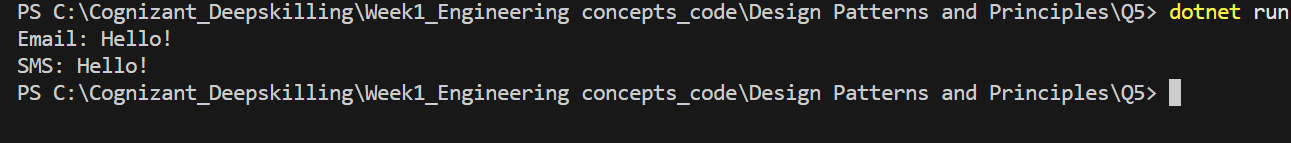
        INotifier notifier = new SMSNotifier(new EmailNotifier());

        notifier.Send("Hello!");

    }

}

* **Output:**



**Exercise 6: Implementing the Proxy Pattern**

* **Code:**

using System;

interface IImage{

    void Display();

}

class RealImage : IImage{

private string filename;

    public RealImage(string filename){

        this.filename = filename;

        LoadFromDisk();

}

    private void LoadFromDisk(){

        Console.WriteLine("Loading " + filename);

}

    public void Display(){

        Console.WriteLine("Displaying " + filename);

    }

}

class ProxyImage : IImage{

    private RealImage? realImage;

private string filename;

    public ProxyImage(string filename){

        this.filename = filename;

}

    public void Display(){

        if (realImage == null)

        {

            realImage = new RealImage(filename);

        }

        realImage.Display();

    }

}

class Program{

    static void Main(){

        IImage img = new ProxyImage("image1.jpg");

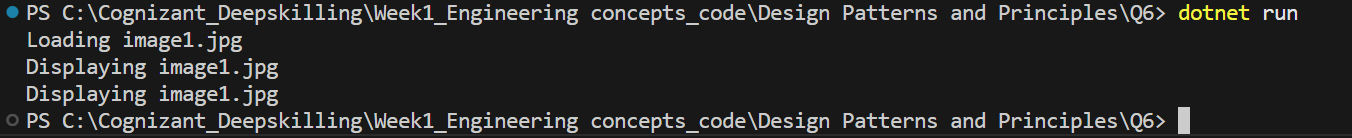
        img.Display();

        img.Display();

    }

}

* **Output:**



**Exercise 7: Implementing the Observer Pattern**

* **Code:**

using System;

using System.Collections.Generic;

interface IObserver{

    void Update(float price);

}

interface IStock{

    void Register(IObserver observer);

    void Unregister(IObserver observer);

    void NotifyObservers();

}

class StockMarket : IStock{

    private List<IObserver> observers = new();

private float price;

    public void Register(IObserver observer) => observers.Add(observer);

    public void Unregister(IObserver observer) => observers.Remove(observer);

    public void SetPrice(float price){

        this.price = price;

        NotifyObservers();

}

    public void NotifyObservers(){

        foreach (var o in observers)

        {

            o.Update(price);

        }

    }

}

class MobileApp : IObserver{

    public void Update(float price)

    {

        Console.WriteLine("Mobile App: Stock price updated to " + price);

    }

}

class Program{

    static void Main(){

        var stock = new StockMarket();

        var mobile = new MobileApp();

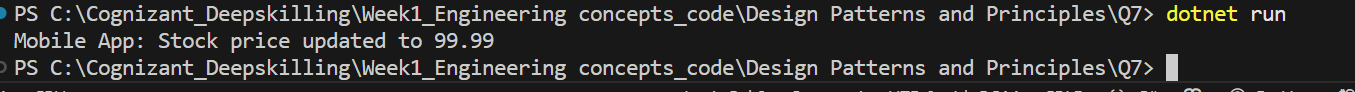
        stock.Register(mobile);

        stock.SetPrice(99.99f);

    }

}

* **Output:**



**Exercise 8: Implementing the Strategy Pattern**

* **Code:**

using System;

interface IPaymentStrategy{

    void Pay();

}

class CreditCardPayment : IPaymentStrategy{

    public void Pay() => Console.WriteLine("Paid using Credit Card");

}

class PayPalPayment : IPaymentStrategy{

    public void Pay() => Console.WriteLine("Paid using PayPal");

}

class PaymentContext{

private IPaymentStrategy strategy;

    public PaymentContext(IPaymentStrategy strategy){

        this.strategy = strategy;

}

    public void Pay(){

        strategy.Pay();

    }

}

class Program{

    static void Main(){

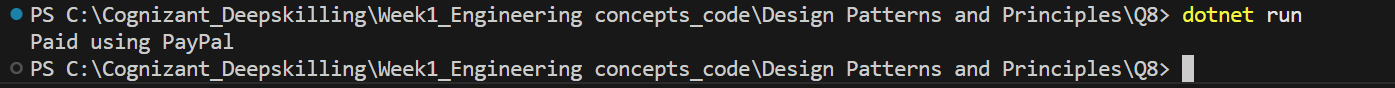
        var context = new PaymentContext(new PayPalPayment());

        context.Pay();

    }

}

* **Output:**



**Exercise 9: Implementing the Command Pattern**

* **Code:**

using System;

interface ICommand{

    void Execute();

}

class Light{

    public void On() => Console.WriteLine("Light is ON");

    public void Off() => Console.WriteLine("Light is OFF");

}

class LightOnCommand : ICommand{

    private Light light;

    public LightOnCommand(Light light) { this.light = light; }

    public void Execute() => light.On();

}

class LightOffCommand : ICommand{

    private Light light;

    public LightOffCommand(Light light) { this.light = light; }

    public void Execute() => light.Off();

}

class RemoteControl{

    private ICommand command;

    public void SetCommand(ICommand cmd) => command = cmd;

    public void PressButton() => command.Execute();

}

class Program{

    static void Main(){

        var light = new Light();

        var remote = new RemoteControl();

        remote.SetCommand(new LightOnCommand(light));

        remote.PressButton();

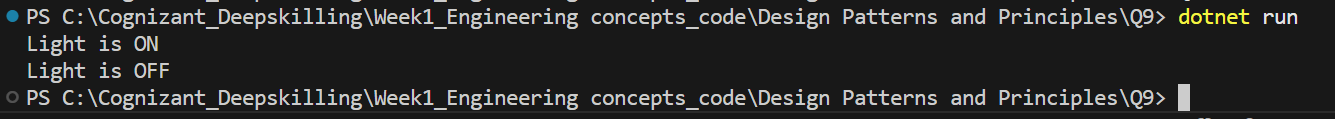
        remote.SetCommand(new LightOffCommand(light));

        remote.PressButton();

    }

}

* **Output:**



**Exercise 10: Implementing the MVC Pattern**

* **Code:**

using System;

class Student

{

    public string Name, Id, Grade;

    public Student(string name, string id, string grade) => (Name, Id, Grade) = (name, id, grade);

}

class StudentView

{

    public void Show(string name, string id, string grade) =>

        Console.WriteLine($"Student Details:\nName: {name}\nID: {id}\nGrade: {grade}\n");

}

class StudentController

{

    Student s; StudentView v;

    public StudentController(Student s, StudentView v) => (this.s, this.v) = (s, v);

    public void SetGrade(string g) => s.Grade = g;

    public void Display() => v.Show(s.Name, s.Id, s.Grade);

}

class Program

{

    static void Main()

    {

        var student = new Student("Gaurav", "22052206", "A");

        var view = new StudentView();

        var controller = new StudentController(student, view);

        controller.Display();

        Console.WriteLine("Updating Grade from A to A+...\n");

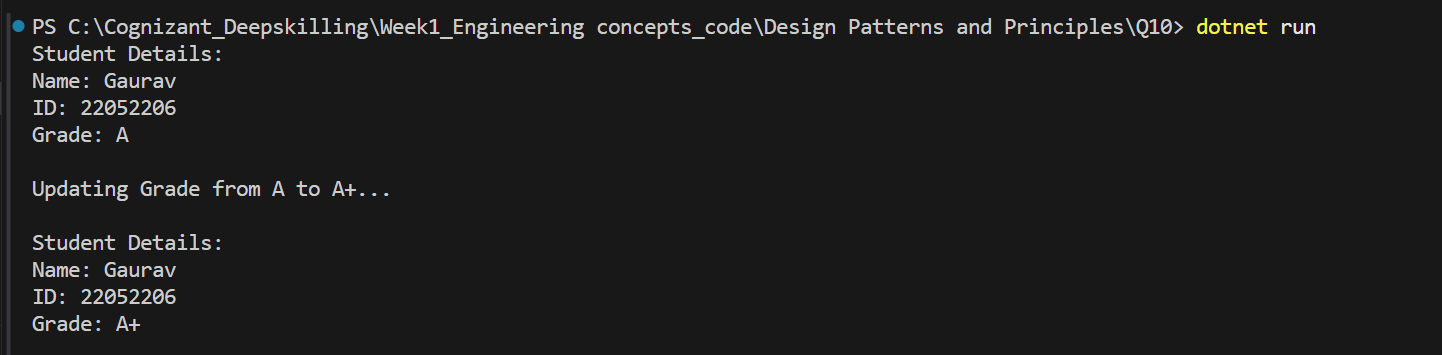
        controller.SetGrade("A+");

        controller.Display();

    }

}

* **Output:**



**Exercise 11: Implementing Dependency Injection**

* **Code:**

using System;

interface ICustomerRepository { string FindCustomerById(int id); }

class CustomerRepositoryImpl : ICustomerRepository

{

    public string FindCustomerById(int id)

    {

        if (id == 22052206) return $"ID: {id}, Name: Gaurav Kumar";

        return $"Customer #{id} not found";

    }

}

class CustomerService

{

    ICustomerRepository repo;

    public CustomerService(ICustomerRepository repo) => this.repo = repo;

    public void PrintCustomer(int id) => Console.WriteLine(repo.FindCustomerById(id));

}

class Program

{

    static void Main()

    {

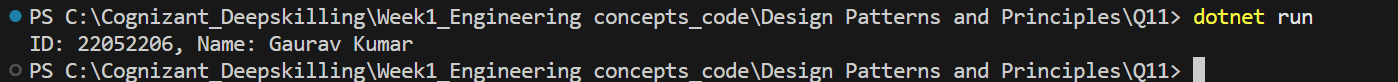
        var service = new CustomerService(new CustomerRepositoryImpl());

        service.PrintCustomer(22052206);

    }

}

* **Output:**



* **Algorithms and Data Structures**

**Exercise 1: Inventory Management System**

* **Code:**

using System;

using System.Collections.Generic;

class Product {

    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public int Quantity { get; set; }

public double Price { get; set; }

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

        ProductId = productId;

        ProductName = productName;

        Quantity = quantity;

        Price = price;

}

    public override string ToString() {

        return $"ID: {ProductId}, Name: {ProductName}, Qty: {Quantity}, Price: {Price}";

    }

}

class InventoryManager {

private Dictionary<int, Product> inventory = new Dictionary<int, Product>();

    public void AddProduct(Product product) {

        inventory[product.ProductId] = product;

}

    public void UpdateProduct(int productId, int newQty, double newPrice) {

        if (inventory.ContainsKey(productId)) {

            inventory[productId].Quantity = newQty;

            inventory[productId].Price = newPrice;

        }

}

    public void DeleteProduct(int productId) {

        inventory.Remove(productId);

}

    public void DisplayInventory() {

        foreach (var item in inventory.Values) {

            Console.WriteLine(item);

        }

    }

}

class Program {

    static void Main(string[] args) {

        InventoryManager manager = new InventoryManager();

        manager.AddProduct(new Product(1, "Keyboard", 10, 500.0));

        manager.AddProduct(new Product(2, "Mouse", 15, 300.0));

        manager.DisplayInventory();

        Console.WriteLine("\nUpdating product 2...");

        manager.UpdateProduct(2, 20, 320.0);

        Console.WriteLine("\nDeleting product 1...");

        manager.DeleteProduct(1);

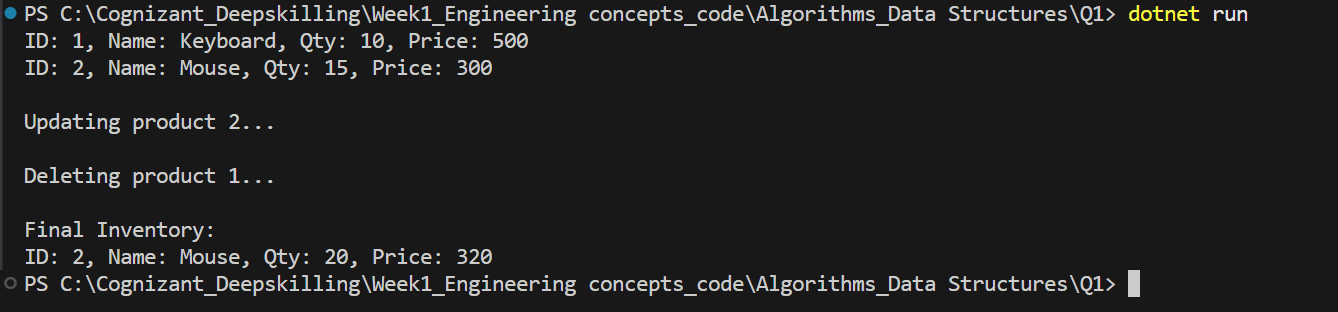
        Console.WriteLine("\nFinal Inventory:");

        manager.DisplayInventory();

    }

}

* **Output:**



* **Analysis:**

Using Dictionary<int, Product> (i.e., HashMap):

* Add, Update, Delete - O(1) average time.
* Optimized for large datasets via constant-time hashing.

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

* **Code:**

using System;

class Product {

    public int ProductId { get; set; }

    public string ProductName { get; set; }

public string Category { get; set; }

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

        ProductId = productId;

        ProductName = productName;

        Category = category;

}

    public override string ToString() {

        return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";

    }

}

class SearchFunction {

    public static int LinearSearch(Product[] products, string name) {

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

            if (products[i].ProductName.Equals(name, StringComparison.OrdinalIgnoreCase)) {

                return i;

            }

        }

        return -1;

}

    public static int BinarySearch(Product[] products, string name) {

        int left = 0, right = products.Length - 1;

        while (left <= right) {

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

            int cmp = string.Compare(products[mid].ProductName, name, StringComparison.OrdinalIgnoreCase);

            if (cmp == 0)

                return mid;

            else if (cmp < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return -1;

    }

}

class Program {

    static void Main(string[] args) {

        Product[] products = {

            new Product(1, "Apple", "Fruits"),

            new Product(2, "Banana", "Fruits"),

            new Product(3, "Carrot", "Vegetables"),

            new Product(4, "Mango", "Fruits")

        };

        Console.WriteLine("=== Linear Search ===");

        int index = SearchFunction.LinearSearch(products, "Carrot");

        Console.WriteLine(index != -1 ? $"Found: {products[index]}" : "Product not found");

        Array.Sort(products, (p1, p2) => p1.ProductName.CompareTo(p2.ProductName));

        Console.WriteLine("\n=== Binary Search ===");

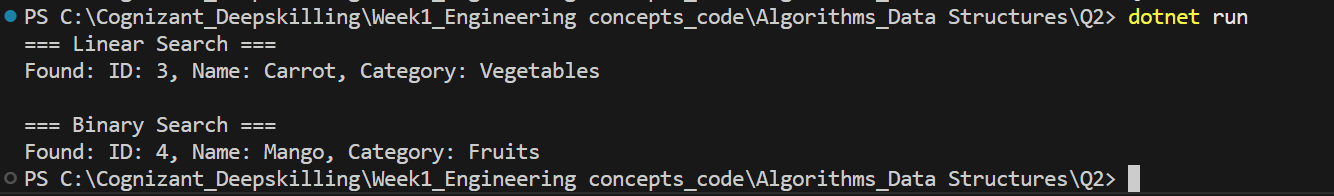
        index = SearchFunction.BinarySearch(products, "Mango");

        Console.WriteLine(index != -1 ? $"Found: {products[index]}" : "Product not found");

    }

}

* **Output:**



* **Analysis:**
* Linear Search: **O(n)**
* Binary Search: ****O(log n)**** (on **sorted** data)
* Use Binary Search for **performance-critical** scenarios with **sorted** data.

**Exercise 3: Sorting Customer Orders**

* **Code:**

using System;

class Order {

    public int OrderId { get; set; }

    public string CustomerName { get; set; }

public double TotalPrice { get; set; }

    public Order(int id, string name, double price) {

        OrderId = id;

        CustomerName = name;

        TotalPrice = price;

}

    public override string ToString() {

        return $"OrderID: {OrderId}, Name: {CustomerName}, Price: {TotalPrice}";

    }

}

class Program {

    static void BubbleSort(Order[] orders) {

        int n = orders.Length;

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

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

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

                    var temp = orders[j];

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

                    orders[j + 1] = temp;

                }

            }

        }

}

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

        if (low < high) {

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

            QuickSort(orders, low, pi - 1);

            QuickSort(orders, pi + 1, high);

        }

}

    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++;

                var temp = orders[i];

                orders[i] = orders[j];

                orders[j] = temp;

            }

        }

        var temp2 = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp2;

        return i + 1;

}

    static void PrintOrders(Order[] orders) {

        foreach (var order in orders) {

            Console.WriteLine(order);

        }

    }

    static void Main(string[] args) {

        Order[] orders = {

            new Order(101, "Alice", 5000),

            new Order(102, "Bob", 2500),

            new Order(103, "Charlie", 8000)

        };

        Console.WriteLine("Bubble Sort:");

        var bubbleSorted = (Order[])orders.Clone();

        BubbleSort(bubbleSorted);

        PrintOrders(bubbleSorted);

        Console.WriteLine("\nQuick Sort:");

        var quickSorted = (Order[])orders.Clone();

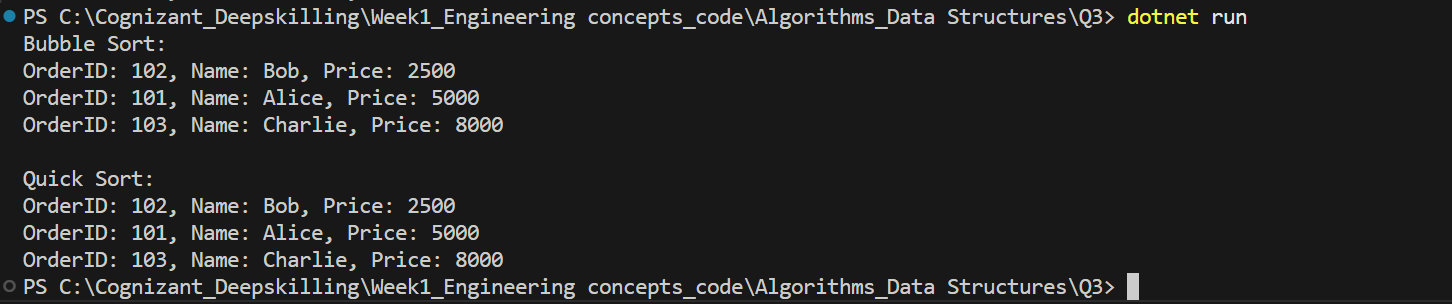
        QuickSort(quickSorted, 0, quickSorted.Length - 1);

        PrintOrders(quickSorted);

    }

}

* **Output:**



* **Analysis:**
* Bubble Sort: ****O(n²)****
* Quick Sort: ****O(n log n)**** average case
* Quick Sort is better for **larger datasets** due to divide-and-conquer strategy.

**Exercise 4: Employee Management System**

* **Code:**

using System;

class Employee {

    public int EmployeeId;

    public string Name;

    public string Position;

public double Salary;

    public Employee(int id, string name, string pos, double sal) {

        EmployeeId = id;

        Name = name;

        Position = pos;

        Salary = sal;

}

    public void Display() {

        Console.WriteLine($"ID: {EmployeeId}, Name: {Name}, Position: {Position}, Salary: {Salary}");

    }

}

class Program {

    static void Main(string[] args) {

        Employee[] employees = new Employee[100];

        int count = 0;

        employees[count++] = new Employee(1, "Alice", "Manager", 60000);

        employees[count++] = new Employee(2, "Bob", "Engineer", 45000);

        Console.WriteLine("All Employees:");

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

            employees[i].Display();

        }

        Console.WriteLine("\nSearching Employee with ID 2:");

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

            if (employees[i].EmployeeId == 2) {

                employees[i].Display();

                break;

            }

        }

        Console.WriteLine("\nDeleting Employee with ID 1:");

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

            if (employees[i].EmployeeId == 1) {

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

                    employees[j] = employees[j + 1];

                }

                count--;

                break;

            }

        }

        Console.WriteLine("\nEmployees after deletion:");

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

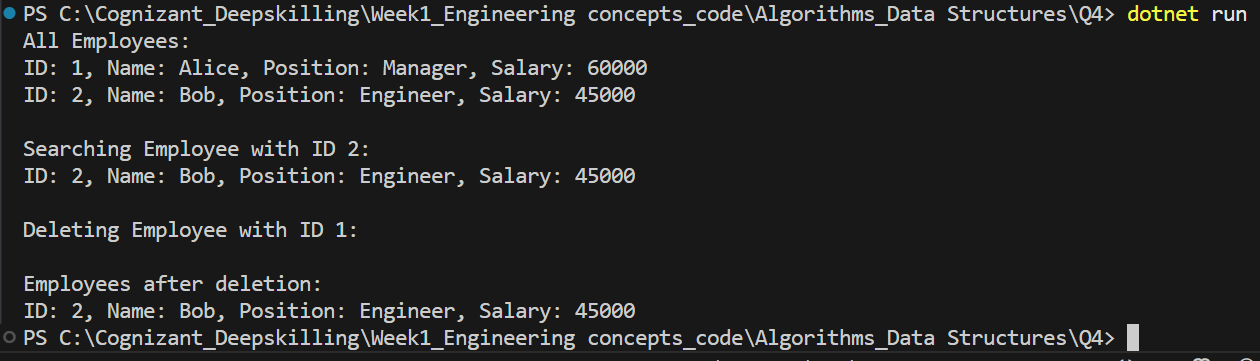
            employees[i].Display();

        }

    }

}

* **Output:**



* **Analysis:**

· Add/Search/Delete: **O(n)** worst-case

· Arrays are fast for **indexed access**, but not efficient for **dynamic data** or frequent **insertions/deletions**.

**Exercise 5: Task Management System**

* **Code:**

using System;

class Task

{

    public int TaskId;

    public string TaskName;

    public string Status;

public Task? Next;

    public Task(int id, string name, string status)

    {

        TaskId = id;

        TaskName = name;

        Status = status;

        Next = null;

}

    public void Display() =>

        Console.WriteLine($"ID: {TaskId}, Name: {TaskName}, Status: {Status}");

}

class TaskManager

{

private Task? head;

    public void AddTask(int id, string name, string status)

    {

        var newTask = new Task(id, name, status);

        if (head == null) head = newTask;

        else

        {

            var temp = head;

            while (temp.Next != null) temp = temp.Next;

            temp.Next = newTask;

        }

}

    public void DisplayTasks()

    {

        var temp = head;

        while (temp != null)

        {

            temp.Display();

            temp = temp.Next;

        }

}

    public void DeleteTask(int id)

    {

        Task? temp = head, prev = null;

        if (temp != null && temp.TaskId == id)

        {

            head = temp.Next;

            return;

        }

        while (temp != null && temp.TaskId != id)

        {

            prev = temp;

            temp = temp.Next;

        }

        if (temp == null || prev == null) return;

        prev.Next = temp.Next;

    }

}

class Program

{

    static void Main()

    {

        var manager = new TaskManager();

        manager.AddTask(1, "Design", "Pending");

        manager.AddTask(2, "Code", "In Progress");

        manager.AddTask(3, "Test", "Pending");

        Console.WriteLine("All Tasks:");

        manager.DisplayTasks();

        Console.WriteLine("\nDeleting Task with ID 2...");

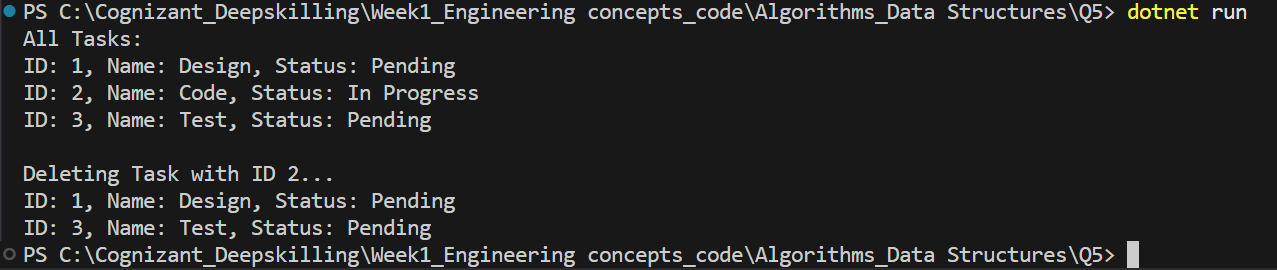
        manager.DeleteTask(2);

        manager.DisplayTasks();

    }

}

* **Output:**



* **Analysis:**

· Add: **O(n)** (if added at end)

· Delete/Search: **O(n)**

· Linked Lists are **better** for **dynamic size** and **frequent insertions/deletions** without shifting elements.

**Exercise 6: Library Management System**

* **Code:**

using System;

class Book {

    public int BookId;

    public string Title;

public string Author;

    public Book(int id, string title, string author) {

        BookId = id;

        Title = title;

        Author = author;

}

    public void Display() {

        Console.WriteLine($"ID: {BookId}, Title: {Title}, Author: {Author}");

    }

}

class Program {

    static int LinearSearch(Book[] books, string title) {

        for (int i = 0; i < books.Length; i++) {

            if (books[i].Title.Equals(title, StringComparison.OrdinalIgnoreCase)) return i;

        }

        return -1;

}

    static int BinarySearch(Book[] books, string title) {

        int low = 0, high = books.Length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int cmp = string.Compare(books[mid].Title, title, StringComparison.OrdinalIgnoreCase);

            if (cmp == 0) return mid;

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

            else high = mid - 1;

        }

        return -1;

}

    static void Main(string[] args) {

        Book[] books = {

            new Book(1, "Algorithms", "Cormen"),

            new Book(2, "Data Structures", "Lafore"),

            new Book(3, "Clean Code", "Martin")

        };

        Console.WriteLine("Linear Search for 'Data Structures':");

        int index = LinearSearch(books, "Data Structures");

        if (index != -1) books[index].Display();

        Array.Sort(books, (b1, b2) => b1.Title.CompareTo(b2.Title));

        Console.WriteLine("\nBinary Search for 'Clean Code':");

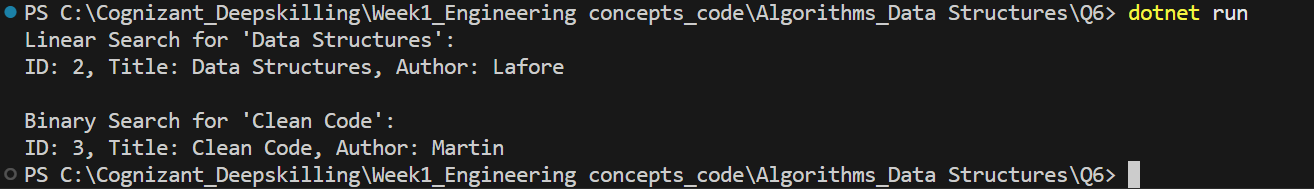
        index = BinarySearch(books, "Clean Code");

        if (index != -1) books[index].Display();

    }

}

* **Output:**



* **Analysis:**

· Linear: **O(n)** – good for **unsorted** or **small data sets**.

· Binary: **O(log n)** – works **only** on **sorted arrays**.

· Binary Search is better for **frequent search operations** with **large sorted data**.

**Exercise 7: Financial Forecasting**

* **Code:**

using System;

class Program {

    static double PredictFutureValue(double currentValue, double growthRate, int years) {

        if (years == 0) return currentValue;

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

}

    static double PredictUsingLoop(double currentValue, double growthRate, int years) {

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

            currentValue \*= (1 + growthRate);

        }

        return currentValue;

}

    static void Main(string[] args) {

        double initial = 10000;

        double rate = 0.1;

        int years = 5;

        double futureRecursive = PredictFutureValue(initial, rate, years);

        double futureIterative = PredictUsingLoop(initial, rate, years);

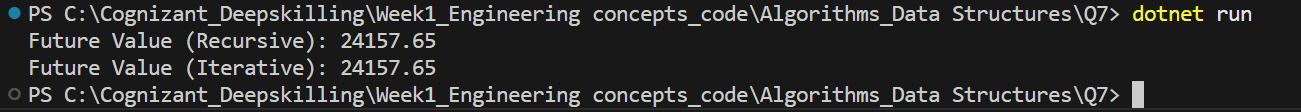
        Console.WriteLine($"Future Value (Recursive): {futureRecursive}");

        Console.WriteLine($"Future Value (Iterative): {futureIterative}");

    }

}

* **Output:**



* **Analysis:**

· Recursive: **O(n)** time, **O(n)** space (stack)

· Optimized (iterative): **O(n)** time, **O(1)** space

· Use **loops** or **memoization** to reduce stack overhead.