Exercise 1: Inventory Management System

1. Understand the Problem

- Data structures and algorithms are critical for handling large inventories to ensure fast lookup, addition, deletion, and updates.
- Suitable data structures:
    - **Dictionary** (HashMap in Java): Fast O(1) average time for lookup, insert, delete.
    - **List**: Useful for traversal but slow (O(n)) for lookups by ID.

2. Setup

- Create a new C# project.
- Define a Product class.

3. Implementation (C#)

```
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; }

}
```

```
class Inventory {

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

```
    public void AddProduct(Product product) => products[product.ProductId] = product;
```

```
    public void UpdateProduct(int id, int quantity, double price) {

        if (products.ContainsKey(id)) {

            products[id].Quantity = quantity;

            products[id].Price = price;
```

```
        }
    }
```

```
    public void DeleteProduct(int id) => products.Remove(id);
```

```
    public void PrintInventory() {
        foreach (var item in products.Values)
            Console.WriteLine($"{item.ProductId} - {item.ProductName} - {item.Quantity} - ₹{item.Price}");
    }
}
```

```
class Program {
    static void Main(string[] args) {
        Inventory inventory = new Inventory();
```

```
        inventory.AddProduct(new Product {
            ProductId = 101,
            ProductName = "Pen",
            Quantity = 50,
            Price = 10.5
        });
```

```
        inventory.AddProduct(new Product {
            ProductId = 102,
            ProductName = "Notebook",
            Quantity = 30,
            Price = 45.0
        });
```

```
        Console.WriteLine("Inventory:");
```

```
        inventory.PrintInventory();
```

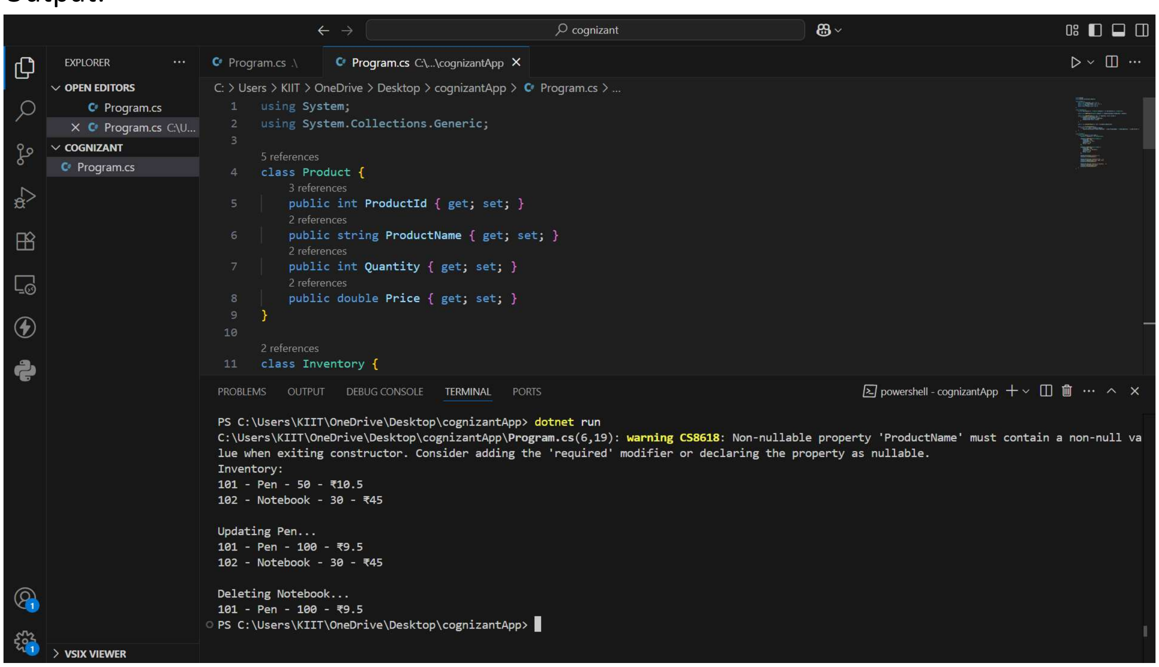
```
        Console.WriteLine("\nUpdating Pen...");

        inventory.UpdateProduct(101, 100, 9.5);

        inventory.PrintInventory();
```

```
        Console.WriteLine("\nDeleting Notebook...");

        inventory.DeleteProduct(102);

        inventory.PrintInventory();

    }

}
```

4. Analysis

- Add, Update, Delete: **O(1)**
- Traverse: **O(n)**
- Optimized with Dictionary for quick access.

Output:



Exercise 2: E-commerce Platform Search Function

1. Understand Asymptotic Notation

- **Big O** describes performance in worst-case.
- **Linear Search**: Best O(1), Avg/Worst O(n).
- **Binary Search**: Best O(1), Avg/Worst O(log n).

2. Setup

- Create Product class with productId, productName, category.

3. Implementation

```
using System;


class Product {

    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public string Category { get; set; }

}
```

```
class Program {

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

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

            if (products[i].ProductName == name)

                return i;

        return -1;

    }
```

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

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

        while (left <= right) {

            int mid = (left + right) / 2;

            int result = string.Compare(products[mid].ProductName, name);

            if (result == 0) return mid;
```

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

            else right = mid - 1;

        }

        return -1;

    }
```

```
    static void Main(string[] args) {

        Product[] products = new Product[] {

            new Product { ProductId = 1, ProductName = "Bag", Category = "Stationery" },

            new Product { ProductId = 2, ProductName = "Book", Category = "Stationery" },

            new Product { ProductId = 3, ProductName = "Pen", Category = "Stationery" },

            new Product { ProductId = 4, ProductName = "Pencil", Category = "Stationery" },

            new Product { ProductId = 5, ProductName = "Ruler", Category = "Stationery" }

        };
```

```
        // NOTE: Binary Search requires sorted array

        Array.Sort(products, (a, b) => string.Compare(a.ProductName, b.ProductName));
```

```
        string searchName = "Pen";
```

```
        int linearIndex = LinearSearch(products, searchName);

        int binaryIndex = BinarySearch(products, searchName);
```

```
        Console.WriteLine($"Linear Search: {(linearIndex >= 0 ? $"Found at index {linearIndex}" : "Not
found")}");

        Console.WriteLine($"Binary Search: {(binaryIndex >= 0 ? $"Found at index {binaryIndex}" : "Not
found")}");

    }

}
```

4. Analysis

- Linear Search: **O(n)**
- Binary Search: **O(log n)**, requires sorted array.
- Use binary search for sorted lists.

Output:

```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(5,19): warning CS8618: Non-nullable property 'ProductName' must contain a non-null va
lue when exiting constructor. Consider adding the 'required' modifier or declaring the property as nullable.
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(6,19): warning CS8618: Non-nullable property 'Category' must contain a non-null value
 when exiting constructor. Consider adding the 'required' modifier or declaring the property as nullable.
Linear Search: Found at index 2
Binary Search: Found at index 2
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 3: Sorting Customer Orders

1. Understand Sorting Algorithms

- Bubble Sort: Simple, O(n^2)
- Quick Sort: Efficient, O(n log n) average

2. Setup

- Class: Order with orderId, customerName, totalPrice

3. Implementation

```
using System;

using System.Collections.Generic;


class Order {

    public int OrderId { get; set; }

    public string CustomerName { get; set; }

    public double TotalPrice { get; set; }

}
```

```
class Program {

    static void BubbleSort(List<Order> orders) {

        int n = orders.Count;

        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)

                (orders[j], orders[j + 1]) = (orders[j + 1], orders[j]);

}
```

```
static void QuickSort(List<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(List<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++;

            (orders[i], orders[j]) = (orders[j], orders[i]);

        }

    }

    (orders[i + 1], orders[high]) = (orders[high], orders[i + 1]);

    return i + 1;

}
```

```
static void PrintOrders(List<Order> orders, string title) {

    Console.WriteLine($"\n{title}:");

    foreach (var order in orders)

        Console.WriteLine($"{order.OrderId} - {order.CustomerName} - ₹{order.TotalPrice}");

}
```

```
    static void Main(string[] args) {

        var orders = new List<Order> {

            new Order { OrderId = 1, CustomerName = "Alice", TotalPrice = 250.0 },

            new Order { OrderId = 2, CustomerName = "Bob", TotalPrice = 100.0 },

            new Order { OrderId = 3, CustomerName = "Charlie", TotalPrice = 450.0 },

            new Order { OrderId = 4, CustomerName = "Diana", TotalPrice = 300.0 }

        };
```

```
        // Bubble Sort

        var bubbleSorted = new List<Order>(orders);

        BubbleSort(bubbleSorted);

        PrintOrders(bubbleSorted, "Bubble Sorted Orders");
```

```
        // Quick Sort

        var quickSorted = new List<Order>(orders);

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

        PrintOrders(quickSorted, "Quick Sorted Orders");

    }

}
```

4. Analysis

- Bubble Sort: O(n^2)
- Quick Sort: O(n log n) average, O(n^2) worst
- Quick Sort is preferred for large datasets

Output:

```
alue when exiting constructor. Consider adding the 'required' modifier or declaring the property as nullable.

Bubble Sorted Orders:
2 - Bob - ₹100
1 - Alice - ₹250
4 - Diana - ₹300
3 - Charlie - ₹450

Quick Sorted Orders:
2 - Bob - ₹100
1 - Alice - ₹250
4 - Diana - ₹300
3 - Charlie - ₹450
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 4: Employee Management System

1. Understand Array Representation

- Stored in contiguous memory blocks.
- Fast indexing: O(1)

2. Setup

- Create Employee class: employeeId, name, position, salary

3. Implementation

```
using System;


class Employee {

    public int EmployeeId { get; set; }

    public string Name { get; set; }

    public string Position { get; set; }

    public double Salary { get; set; }

}
```

```
class EmployeeSystem {

    Employee[] employees = new Employee[100];

    int count = 0;
```

```
    public void Add(Employee emp) => employees[count++] = emp;
```

```
    public Employee Search(int id) {

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

            if (employees[i].EmployeeId == id) return employees[i];

        return null;

    }
```

```
    public void Traverse() {
```

```
        Console.WriteLine("\nEmployee List:");

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

            Console.WriteLine($"{employees[i].EmployeeId} - {employees[i].Name} - {employees[i].Position} - ₹{employees[i].Salary}");

    }
```

```
    public void Delete(int id) {

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

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

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

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

                count--;

                break;

            }

        }

    }

}
```

```
class Program {

    static void Main(string[] args) {

        EmployeeSystem system = new EmployeeSystem();
```

```
        system.Add(new Employee { EmployeeId = 1, Name = "Alice", Position = "Manager", Salary = 60000 });

        system.Add(new Employee { EmployeeId = 2, Name = "Bob", Position = "Developer", Salary = 50000 });

        system.Add(new Employee { EmployeeId = 3, Name = "Charlie", Position = "Designer", Salary = 40000 });
```

```
        system.Traverse();
```

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

        var emp = system.Search(2);

        if (emp != null)
```

```
            Console.WriteLine($"Found: {emp.Name} - {emp.Position} - ₹{emp.Salary}");

        else

            Console.WriteLine("Employee not found.");
```

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

        system.Delete(1);

        system.Traverse();

    }

}
```

4. Analysis

- Add: O(1), Search: O(n), Traverse: O(n), Delete: O(n)
- Limitation: Fixed size, inefficient deletes

Output:

```
Employee List:
1 - Alice - Manager - ₹60000
2 - Bob - Developer - ₹50000
3 - Charlie - Designer - ₹40000

Searching for Employee with ID 2:
Found: Bob - Developer - ₹50000

Deleting Employee with ID 1...

Employee List:
2 - Bob - Developer - ₹50000
3 - Charlie - Designer - ₹40000
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 5: Task Management System

1. Understand Linked Lists

- **Singly Linked List**: One direction
- **Doubly Linked List**: Two-way navigation

. Setup

- Task: taskId, taskName, status

```
using System;


class Task {

    public int TaskId { get; set; }

    public string TaskName { get; set; }

    public string Status { get; set; }

}
```

```
class TaskNode {

    public Task Task;

    public TaskNode Next;

}
```

```
class TaskList {

    TaskNode head;
```

```
    public void Add(Task task) {

        TaskNode newNode = new TaskNode { Task = task, Next = head };

        head = newNode;

    }
```

```
    public Task Search(int id) {

        TaskNode current = head;

        while (current != null) {

            if (current.Task.TaskId == id) return current.Task;
```

```
            current = current.Next;
        }
        return null;
    }
```

```
    public void Traverse() {
        Console.WriteLine("\nTask List:");
        TaskNode current = head;
        while (current != null) {
            Console.WriteLine($"{current.Task.TaskId} - {current.Task.TaskName} - {current.Task.Status}");
            current = current.Next;
        }
    }
```

```
    public void Delete(int id) {
        TaskNode current = head, prev = null;
        while (current != null) {
            if (current.Task.TaskId == id) {
                if (prev == null) head = current.Next;
                else prev.Next = current.Next;
                return;
            }
            prev = current;
            current = current.Next;
        }
    }
}
```

```
class Program {
    static void Main(string[] args) {
        TaskList taskList = new TaskList();
```

```
        taskList.Add(new Task { TaskId = 1, TaskName = "Design UI", Status = "Pending" });

        taskList.Add(new Task { TaskId = 2, TaskName = "Develop Backend", Status = "In Progress" });

        taskList.Add(new Task { TaskId = 3, TaskName = "Write Tests", Status = "Pending" });
```

```
        taskList.Traverse();
```

```
        Console.WriteLine("\nSearching for Task ID 2...");

        var task = taskList.Search(2);

        if (task != null)

            Console.WriteLine($"Found: {task.TaskName} - {task.Status}");

        else

            Console.WriteLine("Task not found.");
```

```
        Console.WriteLine("\nDeleting Task ID 1...");

        taskList.Delete(1);

        taskList.Traverse();

    }

}
```

4. Analysis

- Add: O(1), Search: O(n), Traverse: O(n), Delete: O(n)
- Advantage: No fixed size, easy insert/delete

Output:

```
exiting constructor. Consider adding the 'required' modifier or declaring the field as nullable.

Task List:
3 - Write Tests - Pending
2 - Develop Backend - In Progress
1 - Design UI - Pending

Searching for Task ID 2...
Found: Develop Backend - In Progress

Deleting Task ID 1...

Task List:
3 - Write Tests - Pending
2 - Develop Backend - In Progress
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 6: Library Management System

1. Search Algorithms

- **Linear Search**: O(n)
- **Binary Search**: O(log n) if sorted

2. Setup

- Class: Book with bookId, title, author

3. Implementation

```
using System;


class Book {
    public int BookId { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
}
```

```
class Program {
    static int LinearSearch(Book[] books, string title) {
        for (int i = 0; i < books.Length; i++)
            if (books[i].Title == title)
                return i;
        return -1;
    }
```

```
    static int BinarySearch(Book[] books, string title) {
        int left = 0, right = books.Length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            int cmp = string.Compare(books[mid].Title, title);
            if (cmp == 0) return mid;
            else if (cmp < 0) left = mid + 1;
```

```
            else right = mid - 1;

        }

        return -1;

    }
```

```
    static void Main(string[] args) {

        Book[] books = new Book[] {

            new Book { BookId = 1, Title = "Algorithms", Author = "Cormen" },

            new Book { BookId = 2, Title = "Clean Code", Author = "Robert C. Martin" },

            new Book { BookId = 3, Title = "Design Patterns", Author = "GoF" },

            new Book { BookId = 4, Title = "Effective Java", Author = "Joshua Bloch" },

            new Book { BookId = 5, Title = "Introduction to Algorithms", Author = "CLRS" }

        };
```

```
        // Binary search needs a sorted array by title

        Array.Sort(books, (a, b) => string.Compare(a.Title, b.Title));
```

```
        string searchTitle = "Clean Code";
```

```
        int linearIndex = LinearSearch(books, searchTitle);

        int binaryIndex = BinarySearch(books, searchTitle);
```

```
        Console.WriteLine($"Linear Search: {(linearIndex >= 0 ? $"Found at index {linearIndex}" : "Not found")}");

        Console.WriteLine($"Binary Search: {(binaryIndex >= 0 ? $"Found at index {binaryIndex}" : "Not found")}");

    }

}
```

4. Analysis

- Use binary search for sorted data

- Linear is fine for small/unsorted datasets

Output:

```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(5,19): warning CS8618: Non-nullable property 'Title' must contain a non-null value wh
en exiting constructor. Consider adding the 'required' modifier or declaring the property as nullable.
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(6,19): warning CS8618: Non-nullable property 'Author' must contain a non-null value w
hen exiting constructor. Consider adding the 'required' modifier or declaring the property as nullable.
Linear Search: Found at index 1
Binary Search: Found at index 1
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 7: Financial Forecasting

1. Recursive Algorithms

- Recursive: Function calls itself for smaller problems

2. Setup

- Predict future value based on growth rate

3. Implementation

```
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 void Main(string[] args) {

        double currentValue = 1000.0;     // Initial amount

        double growthRate = 0.1;          // 10% annual growth

        int years = 5;
```

```
        double futureValue = PredictFutureValue(currentValue, growthRate, years);
```

```
        Console.WriteLine($"Future value after {years} years: ₹{futureValue:F2}");

    }
```

```
}
```

4. Analysis

- Time Complexity: O(n)
- Optimization: Use memoization or convert to iteration to reduce stack calls

Output:

```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
Future value after 5 years: ₹1610.51
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```