**Week 1 Assignment Solutions  
Data Structures & Algorithms**

**SuperSet ID: 6365365**

1. **Inventory Management System**

### Why Data Structures & Algorithms Are Essential:

Handling large inventories involves:

**Fast retrieval** of product data

**Efficient storage** and memory usage

**Quick updates** and modifications

Efficient data structures (like HashMap or List) paired with algorithms help maintain **performance** even when the inventory grows large.

### Suitable Data Structures:

**List / ArrayList** (List<T> in C#):  
Useful for small inventories or when order matters.

**Dictionary** (Dictionary<TKey, TValue> in C#):  
Ideal for fast lookups using a key like productId.  
Average time complexity for Add, Update, Delete: **O(1)**.

Inventory.cs

using System;

using System.Collections.Generic;

namespace InventoryManagementSystem

{

    public class Inventory

    {

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

        public void AddProduct(Product product)

        {

            if (products.ContainsKey(product.ProductId))

            {

                Console.WriteLine("Product ID already exists.");

                return;

            }

            products[product.ProductId] = product;

            Console.WriteLine("Product added.");

        }

        public void UpdateProduct(int productId, int quantity, double price)

        {

            if (products.ContainsKey(productId))

            {

                products[productId].Quantity = quantity;

                products[productId].Price = price;

                Console.WriteLine("Product updated.");

            }

            else

            {

                Console.WriteLine("Product not found.");

            }

        }

        public void DeleteProduct(int productId)

        {

            if (products.Remove(productId))

            {

                Console.WriteLine("Product deleted.");

            }

            else

            {

                Console.WriteLine("Product not found.");

            }

        }

        public void DisplayAll()

        {

            foreach (var product in products.Values)

            {

                Console.WriteLine(product);

            }

        }

    }

}

Product.cs

namespace InventoryManagementSystem

{

    public 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}";

        }

    }

}

Program.cs

using System;

namespace InventoryManagementSystem

{

    class Program

    {

        static void Main(string[] args)

        {

            Inventory inventory = new Inventory();

            inventory.AddProduct(new Product(101, "Mouse", 25, 299.99));

            inventory.AddProduct(new Product(102, "Keyboard", 15, 499.99));

            Console.WriteLine("\nInventory:");

            inventory.DisplayAll();

            inventory.UpdateProduct(101, 30, 279.99);

            inventory.DeleteProduct(102);

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

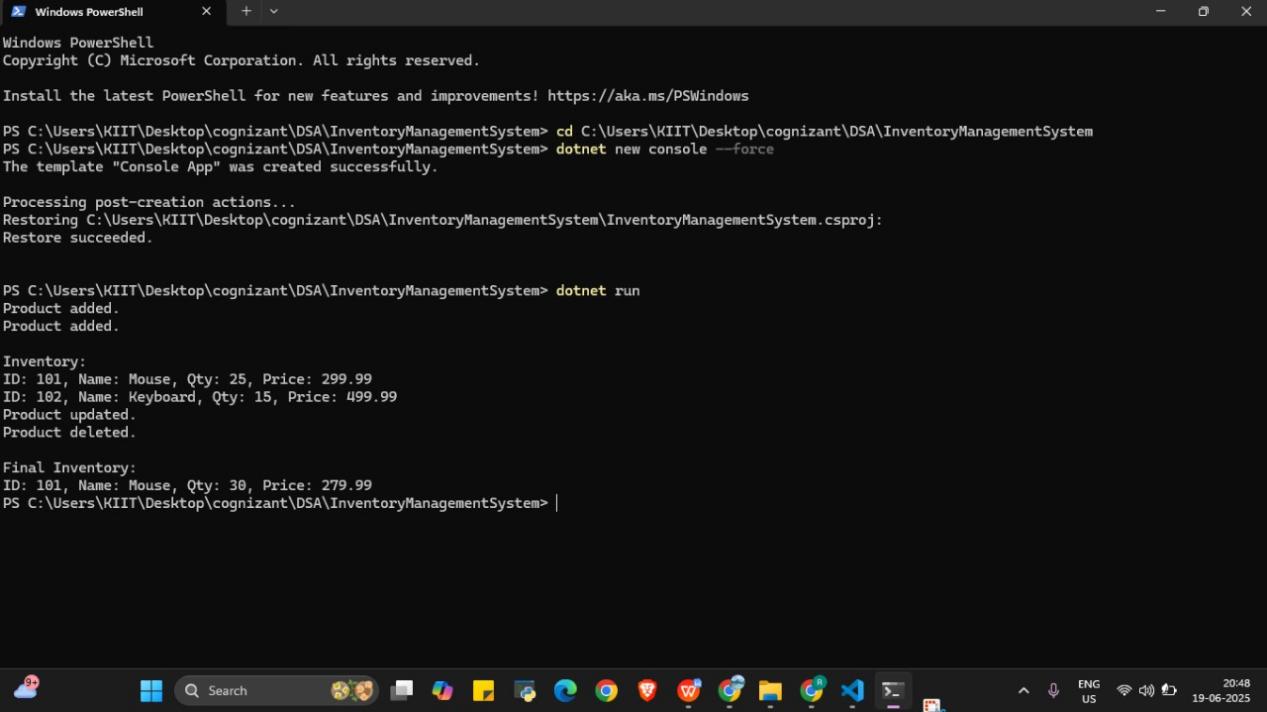
            inventory.DisplayAll();

        }

    }

}

Output



**Analysis-**

| **Operation** | **Data Structure Used** | **Time Complexity** |
| --- | --- | --- |
| Add Product | Dictionary | **O(1)** average |
| Update Product | Dictionary | **O(1)** average |
| Delete Product | Dictionary | **O(1)** average |
| Display All | Dictionary.Values | **O(n)** |

#### Optimization Tips

For larger inventories, persist data using databases (e.g., SQLite or SQL Server).

Add validation logic (e.g., preventing negative quantity).

Use async/await with file/database I/O for better responsiveness.

1. **E-Commerce Platform Search Function**

**Asymptotic notation**

Big O notation describes the upper bound of an algorithm's running time or space requirement in terms of input size n, helping us analyze and compare algorithm efficiency.

**Linear Search**:

Best case: O(1) (item is at the beginning)

Average case: O(n)

Worst case: O(n) (item is not found or at the end)

**Binary Search** (requires sorted data):

Best case: O(1) (middle element is the target)

Average case: O(log n)

Worst case: O(log n)

Binary search is more efficient on large sorted datasets.

Product.cs

public class Product

{

    public int Id { get; set; }

    public string Name { get; set; }

    public Product(int id, string name)

    {

        Id = id;

        Name = name;

    }

}

Search.cs

using System;

public class Search

{

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

    {

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

        {

            if (products[i].Name.Equals(target, StringComparison.OrdinalIgnoreCase))

                return i;

        }

        return -1;

    }

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

    {

        int left = 0;

        int right = products.Length - 1;

        while (left <= right)

        {

            int mid = (left + right) / 2;

            int comparison = string.Compare(products[mid].Name, target, StringComparison.OrdinalIgnoreCase);

            if (comparison == 0)

                return mid;

            else if (comparison < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return -1;

    }

}

Program.cs

using System;

class Program

{

    static void Main(string[] args)

    {

        Product[] productsLinear = {

            new Product(1, "Mouse"),

            new Product(2, "Keyboard"),

            new Product(3, "Monitor"),

            new Product(4, "Printer")

        };

        Product[] productsBinary = {

            new Product(2, "Keyboard"),

            new Product(1, "Mouse"),

            new Product(3, "Monitor"),

            new Product(4, "Printer")

        };

        // Sort for binary search

        Array.Sort(productsBinary, (p1, p2) => p1.Name.CompareTo(p2.Name));

        string target = "Monitor";

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

        int linearIndex = Search.LinearSearch(productsLinear, target);

        Console.WriteLine(linearIndex != -1 ? $"Found at index {linearIndex}" : "Not found");

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

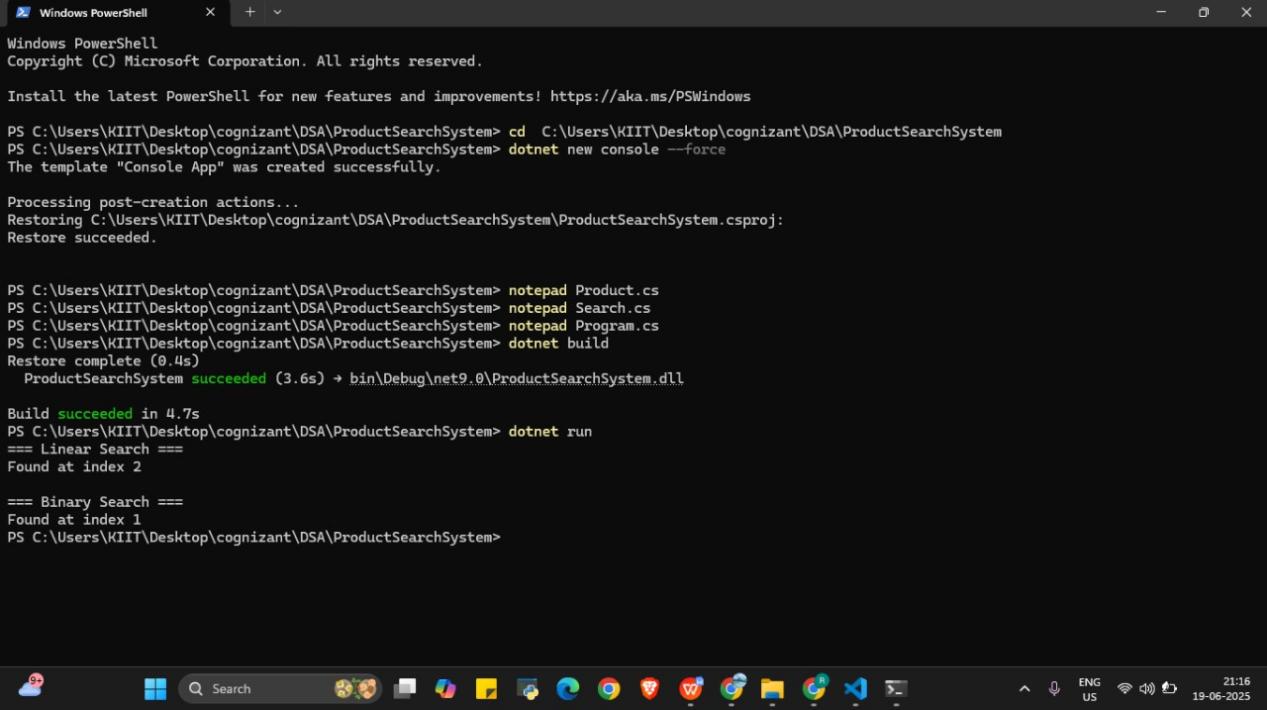
        int binaryIndex = Search.BinarySearch(productsBinary, target);

        Console.WriteLine(binaryIndex != -1 ? $"Found at index {binaryIndex}" : "Not found");

    }

}

Output



**Analysis:**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** | **Sorted Data Required** |
| --- | --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) | ❌ No |
| **Binary Search** | O(1) | O(log n) | O(log n) | ✅ Yes |

**Linear Search** checks each element one-by-one until it finds the target.

**Binary Search** divides the search space in half with each step, which is much faster — but **only works on sorted data**.

#### ****Recommendation****

For an **e-commerce platform** where fast and scalable search is critical:

✅ **Binary Search is more suitable** — especially when users are searching large, sorted product datasets by ID or price.

However, ensure that:

The data is **kept sorted** (e.g., using databases or in-memory sort).

Or use **advanced structures like binary search trees or indexing in DB**.

**Hybrid approach**:

Use **Linear Search** for small or unsorted subsets (like filtering by name or category).

Use **Binary Search** (or better yet, search indexes) for large-scale, performance-critical queries.

1. **Sorting Customer Orders**

## ****1. Understanding Sorting Algorithms****

### Bubble Sort:

Compares adjacent elements and swaps them if out of order.

**Time Complexity:** Best: O(n), Avg/Worst: O(n²)

**Space Complexity:** O(1)

**Use:** Simple but inefficient for large datasets.

### Insertion Sort:

Builds the sorted list by inserting elements one at a time in the correct place.

**Time Complexity:** Best: O(n), Avg/Worst: O(n²)

**Use:** Good for small or nearly sorted arrays.

### Quick Sort:

Uses divide-and-conquer. Picks a pivot and partitions elements.

**Time Complexity:** Best/Avg: O(n log n), Worst: O(n²)

**Use:** Fastest in practice for many datasets.

**Merge Sort:**

Recursively divides the list and merges them in sorted order.

**Time Complexity:** Always O(n log n)

**Space Complexity:** O(n)

**Use:** Stable sort and works well for linked lists.

Order.cs

public class Order

{

    public int OrderId { get; set; }

    public string CustomerName { get; set; }

    public double TotalPrice { get; set; }

    public override string ToString()

    {

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

    }

}

SortAlgorithms.cs

using System;

using System.Collections.Generic;

public static class SortAlgorithms

{

    public 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)

                {

                    var temp = orders[j];

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

                    orders[j + 1] = temp;

                }

            }

        }

    }

    public 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);

        }

    }

    private 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++;

                var temp = orders[i];

                orders[i] = orders[j];

                orders[j] = temp;

            }

        }

        var temp1 = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp1;

        return i + 1;

    }

}

Program.cs

using System;

using System.Collections.Generic;

class Program

{

    static void Main()

    {

        var orders = new List<Order>

        {

            new Order { OrderId = 1, CustomerName = "Alice", TotalPrice = 120.5 },

            new Order { OrderId = 2, CustomerName = "Bob", TotalPrice = 75.0 },

            new Order { OrderId = 3, CustomerName = "Charlie", TotalPrice = 300.0 },

            new Order { OrderId = 4, CustomerName = "David", TotalPrice = 150.75 }

        };

        Console.WriteLine("Original Orders:");

        PrintOrders(orders);

        // Bubble Sort

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

        SortAlgorithms.BubbleSort(bubbleSortedOrders);

        Console.WriteLine("\nBubble Sorted Orders:");

        PrintOrders(bubbleSortedOrders);

        // Quick Sort

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

        SortAlgorithms.QuickSort(quickSortedOrders, 0, quickSortedOrders.Count - 1);

        Console.WriteLine("\nQuick Sorted Orders:");

        PrintOrders(quickSortedOrders);

    }

    static void PrintOrders(List<Order> orders)

    {

        foreach (var order in orders)

        {

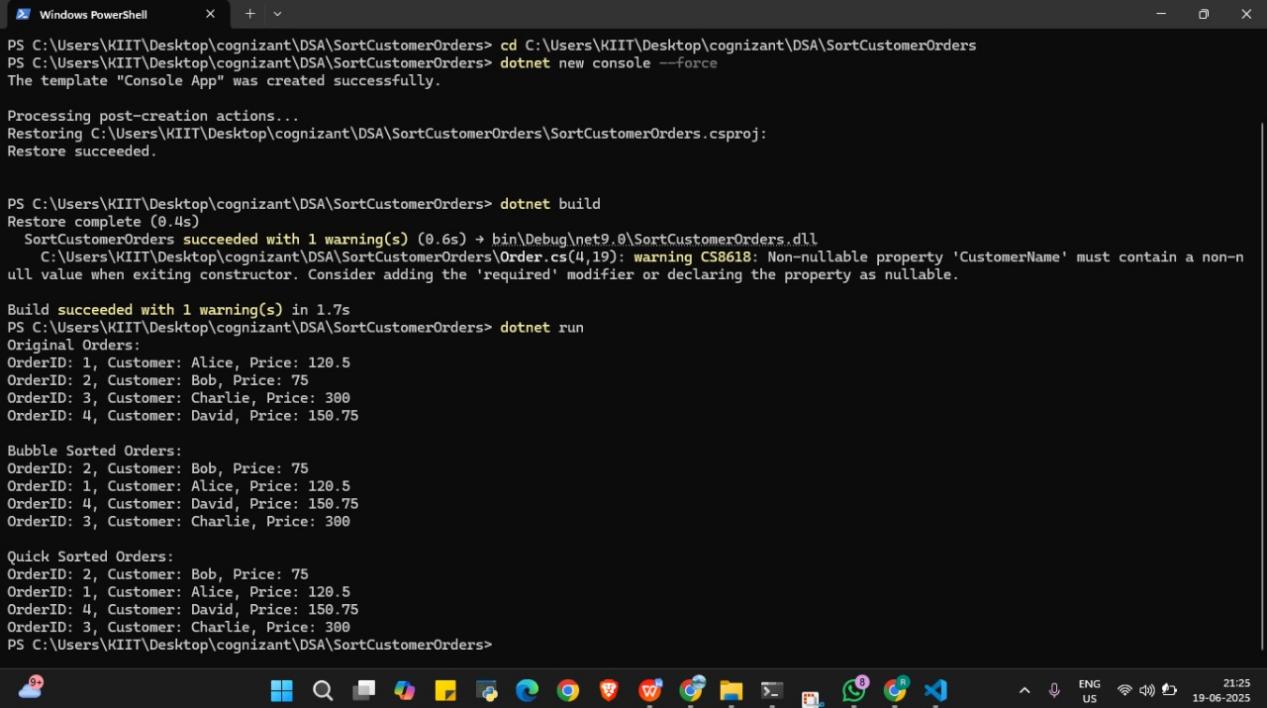
            Console.WriteLine(order);

        }

    }

}

Output



## ****Analysis****

### Time Complexity Comparison:

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space** |
| --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) |

### Why Quick Sort is Preferred:

**Much faster on average** due to divide-and-conquer approach.

**In-place sorting** without extra space (unlike Merge Sort).

**Efficient for large datasets**, unlike Bubble Sort which is only good for small or sorted data.

✅ **Conclusion:**  
Quick Sort is preferred over Bubble Sort because it is **more efficient, scalable, and practical** in most real-world use cases.

1. **Employee Management System**

### ****Understand Array Representation****

**Explanation:**

Arrays in memory are stored in **contiguous (continuous) memory blocks**. Each element is placed next to the previous one.

The **index** of the array acts like an offset to access memory locations quickly.

**Advantages of arrays:**

**Fast access:** Any element can be accessed instantly using its index. Time complexity = **O(1)**.

**Efficient iteration:** Easy to loop through elements sequentially.

**Memory efficiency:** No overhead for pointers or object wrappers (unlike some collections)

Employee.cs

namespace EmployeeManagementSystem

{

    public class Employee

    {

        public int EmployeeId { get; set; }

        public string Name { get; set; }

        public string Position { get; set; }

        public double Salary { get; set; }

        public Employee(int id, string name, string position, double salary)

        {

            EmployeeId = id;

            Name = name;

            Position = position;

            Salary = salary;

        }

        public void Display()

        {

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

        }

    }

}

EmployeeManager.cs

using System;

namespace EmployeeManagementSystem

{

    public class EmployeeManager

    {

        private Employee[] employees;

        private int count;

        public EmployeeManager(int size)

        {

            employees = new Employee[size];

            count = 0;

        }

        public void AddEmployee(Employee emp)

        {

            if (count < employees.Length)

            {

                employees[count++] = emp;

                Console.WriteLine("Employee added.");

            }

            else

            {

                Console.WriteLine("Employee list is full.");

            }

        }

        public void DisplayEmployees()

        {

            Console.WriteLine("Employee List:");

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

            {

                employees[i].Display();

            }

        }

        public Employee SearchById(int id)

        {

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

            {

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

                {

                    return employees[i];

                }

            }

            return null;

        }

        public void DeleteById(int id)

        {

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

            {

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

                {

                    // Shift remaining employees left

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

                    {

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

                    }

                    employees[--count] = null;

                    Console.WriteLine("Employee deleted.");

                    return;

                }

            }

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

        }

    }

}

Program.cs

using System;

using EmployeeManagementSystem;

class Program

{

    static void Main(string[] args)

    {

        EmployeeManager manager = new EmployeeManager(5);

        manager.AddEmployee(new Employee(101, "Alice", "Manager", 75000));

        manager.AddEmployee(new Employee(102, "Bob", "Developer", 60000));

        manager.AddEmployee(new Employee(103, "Carol", "Designer", 50000));

        manager.DisplayEmployees();

        Console.WriteLine("\nSearching for Employee ID 102:");

        var emp = manager.SearchById(102);

        if (emp != null)

            emp.Display();

        else

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

        Console.WriteLine("\nDeleting Employee ID 101:");

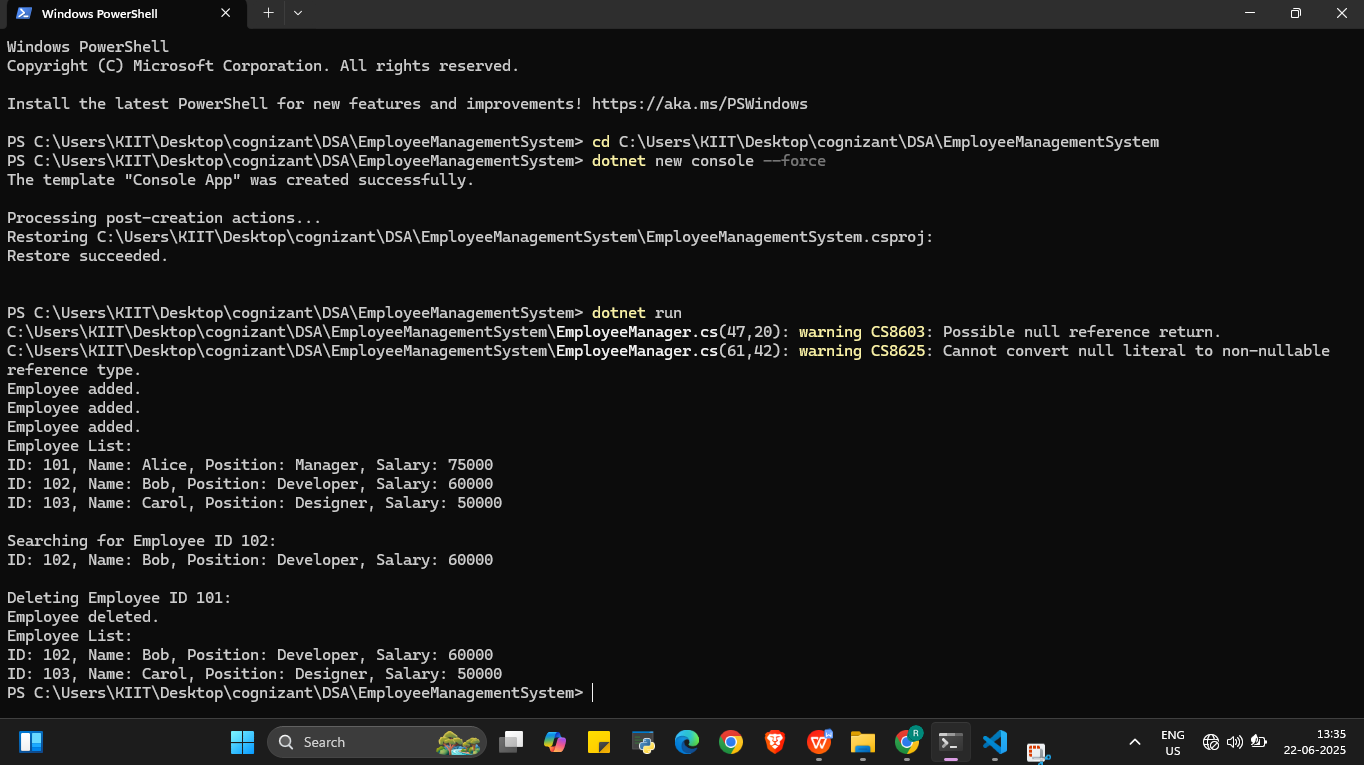
        manager.DeleteById(101);

        manager.DisplayEmployees();

    }

}

Output



### ****Analysis****

#### ****Time Complexity of Operations:****

| **Operation** | **Time Complexity** | **Description** |
| --- | --- | --- |
| Add | **O(1)** (if space available) | Insert at the next free index. |
| Search | **O(n)** | Linear search through the array. |
| Traverse | **O(n)** | Visiting all elements in sequence. |
| Delete | **O(n)** | Find + shift all elements after the deleted one. |

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

**Fixed size**: Cannot grow or shrink after declaration.

**Costly deletions/inserts**: Require shifting elements.

**Wasted memory**: If array size is much larger than needed.

**No built-in flexibility**: Unlike dynamic collections like List<T>.

#### ****When to use arrays:****

When the number of items is **known and fixed**.

When **speed of access** by index is important.

In **memory-constrained** environments.

1. **Task Management System**

### ****Understand Linked Lists****

#### Types of Linked Lists:

**Singly Linked List**:

Each node points to the next node.

Unidirectional.

Efficient for traversal and insertion at the head.

**Doubly Linked List**:

Each node points to both next and previous nodes.

Bidirectional traversal.

Takes more memory (extra pointer for previous).

Task.cs

public class Task

{

    public int TaskId { get; set; }

    public string TaskName { get; set; }

    public string Status { get; set; }

    public Task(int taskId, string taskName, string status)

    {

        TaskId = taskId;

        TaskName = taskName;

        Status = status;

    }

}

TaskLinkedList.cs

using System;

public class TaskLinkedList

{

    private TaskNode head;

    public void AddTask(Task task)

    {

        TaskNode newNode = new TaskNode(task);

        if (head == null)

        {

            head = newNode;

            return;

        }

        TaskNode current = head;

        while (current.Next != null)

        {

            current = current.Next;

        }

        current.Next = newNode;

    }

    public void TraverseTasks()

    {

        TaskNode current = head;

        while (current != null)

        {

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

            current = current.Next;

        }

    }

    public Task SearchTask(int taskId)

    {

        TaskNode current = head;

        while (current != null)

        {

            if (current.Task.TaskId == taskId)

                return current.Task;

            current = current.Next;

        }

        return null;

    }

    public void DeleteTask(int taskId)

    {

        if (head == null) return;

        if (head.Task.TaskId == taskId)

        {

            head = head.Next;

            return;

        }

        TaskNode current = head;

        while (current.Next != null && current.Next.Task.TaskId != taskId)

        {

            current = current.Next;

        }

        if (current.Next != null)

        {

            current.Next = current.Next.Next;

        }

    }

}

TaskNode.cs

public class TaskNode

{

    public Task Task { get; set; }

    public TaskNode Next { get; set; }

    public TaskNode(Task task)

    {

        Task = task;

        Next = null;

    }

}

Program.cs

using System;

class Program

{

    static void Main(string[] args)

    {

        TaskLinkedList taskList = new TaskLinkedList();

        taskList.AddTask(new Task(1, "Design Database", "Pending"));

        taskList.AddTask(new Task(2, "Implement UI", "In Progress"));

        taskList.AddTask(new Task(3, "Write Tests", "Pending"));

        Console.WriteLine("All Tasks:");

        taskList.TraverseTasks();

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

        Task searchResult = taskList.SearchTask(2);

        if (searchResult != null)

            Console.WriteLine($"Found: {searchResult.TaskName}, Status: {searchResult.Status}");

        else

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

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

        taskList.DeleteTask(2);

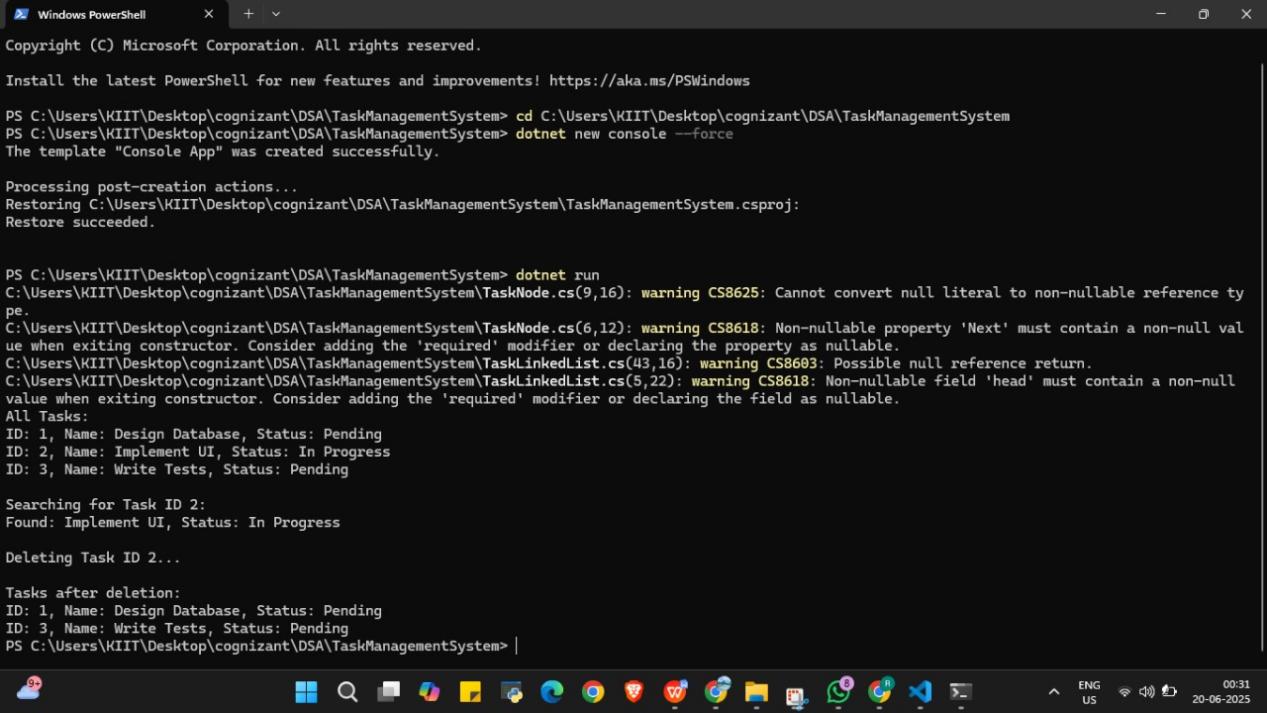
        Console.WriteLine("\nTasks after deletion:");

        taskList.TraverseTasks();

    }

}

Output



### ****Analysis****

### Time Complexity:

| **Operation** | **Time Complexity** |
| --- | --- |
| Add Task | O(n) (to end) |
| Search Task | O(n) |
| Traverse Tasks | O(n) |
| Delete Task | O(n) |

#### Linked List vs Array:

| **Feature** | **Linked List** | **Array** |
| --- | --- | --- |
| Size | Dynamic | Fixed |
| Insertion/Deletion | O(1) (at head) | O(n) (shift needed) |
| Random Access | ❌ No (sequential) | ✅ Yes |
| Memory Usage | More (pointers) | Less |

**Conclusion:** Linked Lists are better when:

Frequent insertions/deletions are needed.

The number of elements is dynamic.  
Arrays are better when:

Random access is required.

Number of elements is known/fixed.

1. **Library Management System**

#### ****Linear Search:****

**Definition**: A simple search algorithm that checks every element in the list one by one.

**Time Complexity**:

Best Case: O(1) (first element)

Average/Worst Case: O(n)

**When to Use**: When the data is **unsorted** or the list is **small.**

#### ****Binary Search:****

#### ****Definition****: An efficient algorithm that repeatedly divides the sorted list into halves to find the target.

**Time Complexity**:

Best Case: O(1)

Average/Worst Case: O(log n)

**When to Use**: When the list is **sorted**.

Book.cs

public class Book

{

    public int BookId { get; set; }

    public string Title { get; set; }

    public string Author { get; set; }

    public Book(int bookId, string title, string author)

    {

        BookId = bookId;

        Title = title;

        Author = author;

    }

    public override string ToString()

    {

        return $"ID: {BookId}, Title: {Title}, Author: {Author}";

    }

}

Search.cs

using System;

using System.Collections.Generic;

public static class Search

{

    public static Book LinearSearch(List<Book> books, string title)

    {

        foreach (Book book in books)

        {

            if (book.Title.Equals(title, StringComparison.OrdinalIgnoreCase))

                return book;

        }

        return null;

    }

    public static Book BinarySearch(List<Book> books, string title)

    {

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

        while (low <= high)

        {

            int mid = (low + high) / 2;

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

            if (comparison == 0)

                return books[mid];

            else if (comparison < 0)

                low = mid + 1;

            else

                high = mid - 1;

        }

        return null;

    }

}

Program.cs

using System;

using System.Collections.Generic;

class Program

{

    static void Main()

    {

        List<Book> books = new List<Book>

        {

            new Book(101, "C# Basics", "Alice"),

            new Book(102, "Data Structures", "Bob"),

            new Book(103, "Algorithms", "Charlie"),

            new Book(104, "Operating Systems", "David"),

            new Book(105, "Networks", "Eve")

        };

        Console.WriteLine("---- LINEAR SEARCH ----");

        var result1 = Search.LinearSearch(books, "Algorithms");

        Console.WriteLine(result1 != null ? result1.ToString() : "Book not found");

        Console.WriteLine("\n---- BINARY SEARCH ----");

        books.Sort((b1, b2) => b1.Title.CompareTo(b2.Title)); // Ensure list is sorted

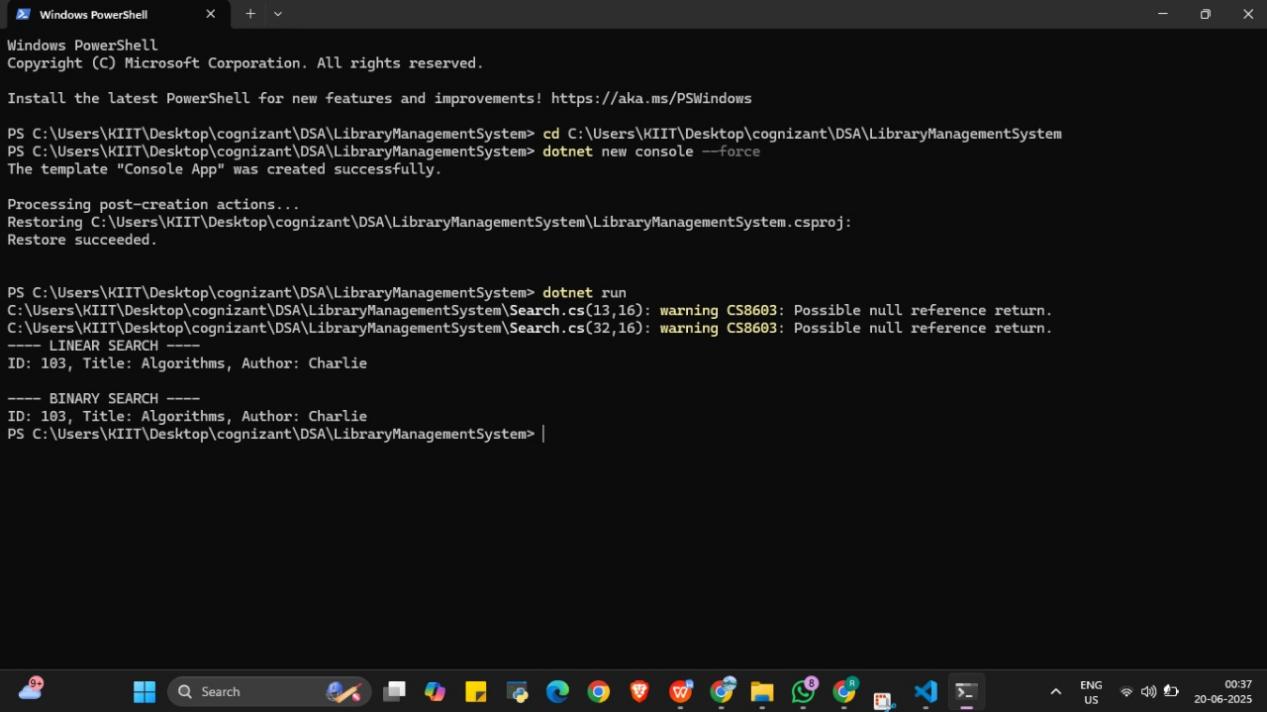
        var result2 = Search.BinarySearch(books, "Algorithms");

        Console.WriteLine(result2 != null ? result2.ToString() : "Book not found");

    }

}

Output



**Analysis**

| **Aspect** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Time Complexity** | O(n) | O(log n) |
| **Data Requirement** | Unsorted or sorted | Must be **sorted** |
| **Implementation** | Simple | Requires sort & logic |
| **Use Case** | Small datasets, unsorted lists | Large sorted datasets |

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

Use **linear search** when:

The list is **unsorted**

The dataset is **small**

You need a quick, simple solution

Use **binary search** when:

The list is **already sorted**

The dataset is **large**

Performance matters (especially in frequent searches)

1. **Financial Forecasting**

### ****Understand Recursive Algorithms****

**Recursion** is a technique where a function calls itself to solve a smaller instance of the same problem. It simplifies problems that can be broken down into similar subproblems.

**Example**: Calculating factorial:

csharp

CopyEdit

int Factorial(int n)

{

if (n == 0) return 1;

return n \* Factorial(n - 1);

}

**Why recursion?**

Cleaner and shorter code for problems with repetitive structures.

Useful in financial forecasting when each year depends on the result of the previous year.

Forecaster.cs

public static class Forecaster

{

    // Recursive method to calculate future value

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

    {

        if (years == 0)

            return currentValue;

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

    }

    // Optimized version using memoization (optional)

    public static double PredictWithMemo(double currentValue, double growthRate, int years, Dictionary<int, double> memo)

    {

        if (years == 0)

            return currentValue;

        if (memo.ContainsKey(years))

            return memo[years];

        double futureValue = PredictWithMemo(currentValue, growthRate, years - 1, memo) \* (1 + growthRate);

        memo[years] = futureValue;

        return futureValue;

    }

}

Program.cs

using System;

using System.Collections.Generic;

class Program

{

    static void Main()

    {

        double initialValue = 10000; // Starting capital

        double growthRate = 0.08;    // 8% annual growth

        int years = 5;

        Console.WriteLine("---- Financial Forecast using Recursion ----");

        double futureValue = Forecaster.PredictFutureValue(initialValue, growthRate, years);

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

        Console.WriteLine("\n---- Financial Forecast with Memoization ----");

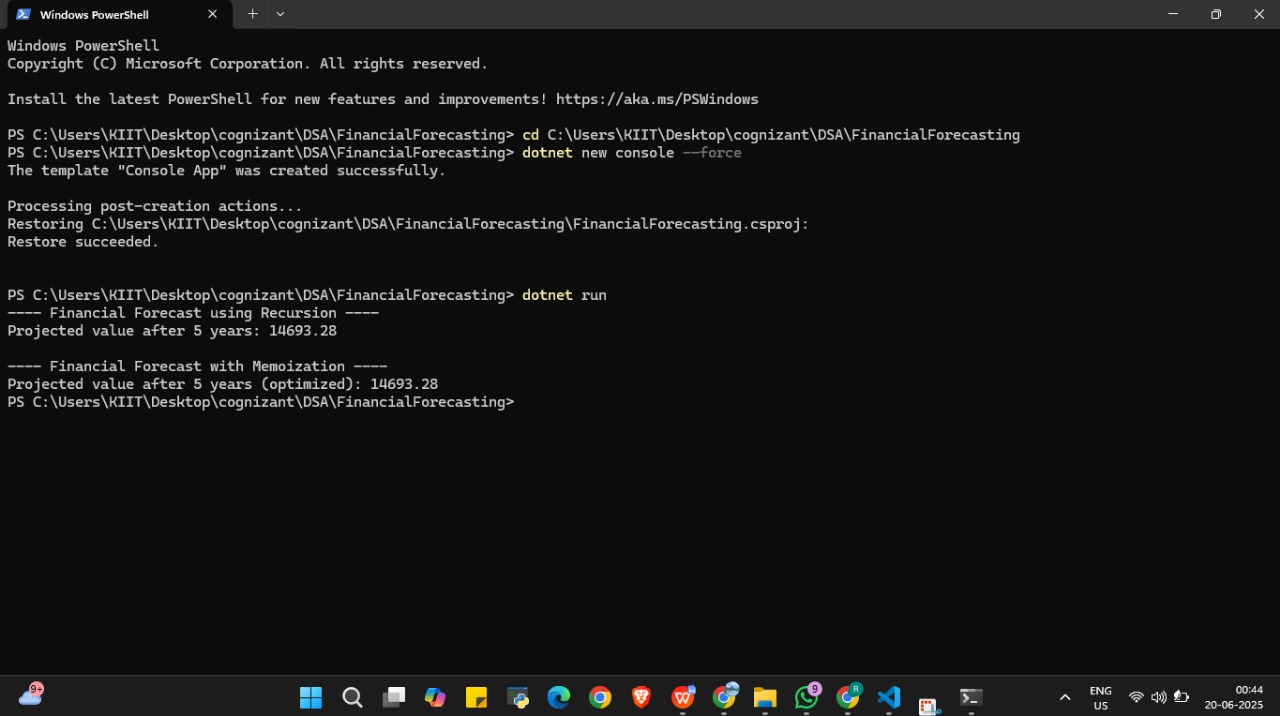
        double optimizedValue = Forecaster.PredictWithMemo(initialValue, growthRate, years, new Dictionary<int, double>());

        Console.WriteLine($"Projected value after {years} years (optimized): {optimizedValue:F2}");

    }

}

Output



### ****Analysis****

#### ****Time Complexity:****

The time complexity is **O(n)** because each recursive call reduces years by 1 until it reaches 0. So, it makes n calls.

#### ****Space Complexity:****

Also **O(n)** due to the call stack storing n recursive frames.

### ****Optimization Tips****

Recursion is elegant but can be inefficient for large n due to:

Stack overflow

Repeating calculations (in other recursive problems)

#### ****Optimized Version: Iterative Approach****

You can optimize by converting recursion to iteration:

static double PredictFutureValueIterative(double initialValue, double growthRate, int years)

{

double value = initialValue;

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

{

value \*= (1 + growthRate);

}

return value;

}

This iterative method:

Has **O(n)** time complexity.

Uses **O(1)** space (no recursive stack).

### Summary

| **Method** | **Time Complexity** | **Space Complexity** | **Pros** | **Cons** |
| --- | --- | --- | --- | --- |
| Recursive | O(n) | O(n) | Simple, elegant | Stack overhead |
| Iterative | O(n) | O(1) | Efficient, no stack overflow | Slightly more verbose code |