Complete Beginner's Guide to Data Structures and Algorithms in C#

Table of Contents

Introduction

What are Data Structures?

Data structures are ways to organize and store data in a computer so that it can be accessed and manipulated efficiently. Think of them like different types of containers - some are better for certain tasks than others.

What are Algorithms?

Algorithms are step-by-step instructions for solving problems or performing tasks. They tell the computer exactly what to do with the data.

Why Learn DSA?

- Write more efficient code
- Solve complex problems systematically

- Essential for technical interviews
- Foundation for advanced programming concepts

Setting Up Your Environment

Prerequisites

- Basic C# knowledge (variables, loops, methods, classes)
- Visual Studio or Visual Studio Code installed
- .NET SDK installed

Creating Your First Project

```bash
dotnet new console -n DSALearning
cd DSALearning
```

Basic Concepts

Big O Notation (Simplified)

Big O describes how the runtime of an algorithm grows as the input size increases:

- **O(1)** - Constant time: Same time regardless of input size
- **O(n)** - Linear time: Time increases proportionally with input
- **O(n^2)** - Quadratic time: Time increases with square of input
- **O(log n)** - Logarithmic time: Very efficient, time increases slowly

Real-world analogy: Finding a book in a library

- O(1): You know the exact shelf location
- O(n): You check every book one by one
- O(log n): You use the card catalog system

Arrays

What is an Array?

An array is a collection of elements stored in contiguous memory locations. Each element can be accessed using an index.

When to Use Arrays

- When you know the size beforehand
- When you need fast access to elements by index
- When memory usage is a concern

C# Array Examples

```csharp
// Declaration and initialization
int[] numbers = new int[5]; // Creates array of size 5
int[] scores = {85, 92, 78, 96, 88}; // Initialize with values

// Accessing elements
Console.WriteLine(scores[0]); // Output: 85 (first element)
Console.WriteLine(scores[4]); // Output: 88 (last element)

// Modifying elements
scores[2] = 95; // Changes 78 to 95

// Getting array length
Console.WriteLine($"Array length: {scores.Length}");

// Iterating through array
for (int i = 0; i < scores.Length; i++)
{
    Console.WriteLine($"Score {i + 1}: {scores[i]}");
}

// Using foreach (easier way)
foreach (int score in scores)
{
    Console.WriteLine($"Score: {score}");
}
```

Common Array Operations

```csharp
```

```
public class ArrayOperations
{
    // Find maximum element
    public static int FindMax(int[] arr)
    {
        if (arr.Length == 0) return 0;

        int max = arr[0];
        for (int i = 1; i < arr.Length; i++)
        {
            if (arr[i] > max)
                max = arr[i];
        }
        return max;
    }

    // Find sum of all elements
    public static int FindSum(int[] arr)
    {
        int sum = 0;
        foreach (int num in arr)
        {
            sum += num;
        }
        return sum;
    }

    // Reverse an array
    public static void ReverseArray(int[] arr)
    {
        int start = 0;
        int end = arr.Length - 1;

        while (start < end)
        {
            // Swap elements
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;

            start++;
            end--;
        }
```

```
    }
}
```

Lists

What is a List?

A List is like an array but can grow and shrink during runtime. It's more flexible than arrays.

When to Use Lists

- When you don't know the final size
- When you need to frequently add/remove elements
- When you want built-in methods for common operations

C# List Examples

```csharp
```

```csharp
using System.Collections.Generic;

// Creating a list
List<int> numbers = new List<int>();
List<string> names = new List<string> {"Alice", "Bob", "Charlie"};

// Adding elements
numbers.Add(10);
numbers.Add(20);
numbers.Add(30);

// Adding multiple elements
numbers.AddRange(new int[] {40, 50});

// Accessing elements
Console.WriteLine(names[0]); // Output: Alice

// Removing elements
names.Remove("Bob"); // Removes first occurrence
names.RemoveAt(0);   // Removes element at index 0

// Checking if element exists
if (numbers.Contains(20))
{
    Console.WriteLine("Found 20!");
}

// Getting count
Console.WriteLine($"List has {numbers.Count} elements");

// Iterating
foreach (int num in numbers)
{
    Console.WriteLine(num);
}
```

Practical List Example: Student Grade Manager

```
csharp
```

```
public class GradeManager
{
    private List<int> grades;

    public GradeManager()
    {
        grades = new List<int>();
    }

    public void AddGrade(int grade)
    {
        if (grade >= 0 && grade <= 100)
        {
            grades.Add(grade);
            Console.WriteLine($"Added grade: {grade}");
        }
        else
        {
            Console.WriteLine("Invalid grade! Must be between 0-100");
        }
    }

    public double GetAverage()
    {
        if (grades.Count == 0) return 0;

        int sum = 0;
        foreach (int grade in grades)
        {
            sum += grade;
        }
        return (double)sum / grades.Count;
    }

    public int GetHighestGrade()
    {
        if (grades.Count == 0) return 0;

        int highest = grades[0];
        foreach (int grade in grades)
        {
            if (grade > highest)
                highest = grade;
```

```
        }
        return highest;
    }

    public void DisplayGrades()
    {
        Console.WriteLine("All grades:");
        for (int i = 0; i < grades.Count; i++)
        {
            Console.WriteLine($"Grade {i + 1}: {grades[i]}");
        }
    }
}

// Usage example
static void Main()
{
    GradeManager gm = new GradeManager();
    gm.AddGrade(85);
    gm.AddGrade(92);
    gm.AddGrade(78);

    Console.WriteLine($"Average: {gm.GetAverage():F2}");
    Console.WriteLine($"Highest: {gm.GetHighestGrade()}");
    gm.DisplayGrades();
}
```

Stacks

What is a Stack?

A stack follows the LIFO (Last In, First Out) principle. Think of it like a stack of plates - you can only add or remove from the top.

When to Use Stacks

- Undo operations in applications
- Function call management
- Expression evaluation
- Backtracking algorithms

Key Operations

- **Push**: Add element to top
- **Pop**: Remove and return top element
- **Peek/Top**: Look at top element without removing
- **IsEmpty**: Check if stack is empty

C# Stack Implementation

```csharp
using System.Collections.Generic;

// Using built-in Stack
Stack<int> stack = new Stack<int>();

// Push elements
stack.Push(10);
stack.Push(20);
stack.Push(30);

// Pop elements (removes and returns)
int top = stack.Pop(); // Returns 30
Console.WriteLine($"Popped: {top}");

// Peek (look without removing)
int topElement = stack.Peek(); // Returns 20
Console.WriteLine($"Top element: {topElement}");

// Check if empty
if (stack.Count > 0)
{
    Console.WriteLine("Stack is not empty");
}
```

Custom Stack Implementation

```csharp
```

```csharp
public class MyStack<T>
{
    private List<T> items;

    public MyStack()
    {
        items = new List<T>();
    }

    public void Push(T item)
    {
        items.Add(item);
        Console.WriteLine($"Pushed: {item}");
    }

    public T Pop()
    {
        if (IsEmpty())
        {
            throw new InvalidOperationException("Stack is empty");
        }

        T item = items[items.Count - 1];
        items.RemoveAt(items.Count - 1);
        return item;
    }

    public T Peek()
    {
        if (IsEmpty())
        {
            throw new InvalidOperationException("Stack is empty");
        }

        return items[items.Count - 1];
    }

    public bool IsEmpty()
    {
        return items.Count == 0;
    }
```

```
    public int Count => items.Count;
}
```

Practical Stack Example: Balanced Parentheses Checker

```csharp
```

```
public class ParenthesesChecker
{
    public static bool IsBalanced(string expression)
    {
        Stack<char> stack = new Stack<char>();

        foreach (char ch in expression)
        {
            // Push opening brackets
            if (ch == '(' || ch == '{' || ch == '[')
            {
                stack.Push(ch);
            }
            // Check closing brackets
            else if (ch == ')' || ch == '}' || ch == ']')
            {
                if (stack.Count == 0)
                    return false; // No matching opening bracket

                char top = stack.Pop();
                if (!IsMatchingPair(top, ch))
                    return false;
            }
        }

        return stack.Count == 0; // All brackets should be matched
    }

    private static bool IsMatchingPair(char opening, char closing)
    {
        return (opening == '(' && closing == ')') ||
               (opening == '{' && closing == '}') ||
               (opening == '[' && closing == ']');
    }
}

// Usage
Console.WriteLine(ParenthesesChecker.IsBalanced("()")); // True
Console.WriteLine(ParenthesesChecker.IsBalanced("({[]})")); // True
Console.WriteLine(ParenthesesChecker.IsBalanced("({[}])")); // False
```

Queues

What is a Queue?

A queue follows the FIFO (First In, First Out) principle. Think of it like a line at a store - first person in line gets served first.

When to Use Queues

- Task scheduling
- Breadth-first search in graphs
- Handling requests in web servers
- Print job management

Key Operations

- **Enqueue**: Add element to rear
- **Dequeue**: Remove and return front element
- **Front/Peek**: Look at front element without removing
- **IsEmpty**: Check if queue is empty

C# Queue Examples

```csharp
```

```csharp
using System.Collections.Generic;

// Using built-in Queue
Queue<string> queue = new Queue<string>();

// Enqueue elements
queue.Enqueue("First");
queue.Enqueue("Second");
queue.Enqueue("Third");

// Dequeue elements
string first = queue.Dequeue(); // Returns "First"
Console.WriteLine($"Dequeued: {first}");

// Peek at front
string front = queue.Peek(); // Returns "Second"
Console.WriteLine($"Front element: {front}");

// Count elements
Console.WriteLine($"Queue size: {queue.Count}");
```

Practical Queue Example: Print Job Manager

```
csharp
```

```
public class PrintJob
{
    public string DocumentName { get; set; }
    public string UserName { get; set; }
    public int Pages { get; set; }

    public PrintJob(string docName, string userName, int pages)
    {
        DocumentName = docName;
        UserName = userName;
        Pages = pages;
    }

    public override string ToString()
    {
        return $"{DocumentName} by {UserName} ({Pages} pages)";
    }
}

public class PrintQueue
{
    private Queue<PrintJob> jobs;

    public PrintQueue()
    {
        jobs = new Queue<PrintJob>();
    }

    public void AddJob(PrintJob job)
    {
        jobs.Enqueue(job);
        Console.WriteLine($"Added to queue: {job}");
    }

    public void ProcessNextJob()
    {
        if (jobs.Count == 0)
        {
            Console.WriteLine("No jobs in queue");
            return;
        }

        PrintJob job = jobs.Dequeue();
```

```
        Console.WriteLine($"Processing: {job}");

        // Simulate printing time
        System.Threading.Thread.Sleep(job.Pages * 100);
        Console.WriteLine($"Completed: {job.DocumentName}");
    }

    public void ShowQueue()
    {
        Console.WriteLine($"\nJobs in queue: {jobs.Count}");
        int position = 1;
        foreach (PrintJob job in jobs)
        {
            Console.WriteLine($"{position}. {job}");
            position++;
        }
    }
}

// Usage example
static void Main()
{
    PrintQueue pq = new PrintQueue();

    pq.AddJob(new PrintJob("Report.pdf", "Alice", 10));
    pq.AddJob(new PrintJob("Invoice.docx", "Bob", 2));
    pq.AddJob(new PrintJob("Presentation.pptx", "Charlie", 25));

    pq.ShowQueue();

    pq.ProcessNextJob();
    pq.ProcessNextJob();

    pq.ShowQueue();
}
```

Linked Lists

What is a Linked List?

A linked list is a collection of nodes where each node contains data and a reference (link) to the next node. Unlike arrays, elements are not stored in contiguous memory.

Types of Linked Lists

1. **Singly Linked List**: Each node points to the next node
2. **Doubly Linked List**: Each node has pointers to both next and previous nodes
3. **Circular Linked List**: Last node points back to the first node

When to Use Linked Lists

- When you frequently insert/delete at the beginning
- When the size varies significantly
- When you don't need random access to elements

Singly Linked List Implementation

```csharp
```

```
public class Node<T>
{
    public T Data { get; set; }
    public Node<T> Next { get; set; }

    public Node(T data)
    {
        Data = data;
        Next = null;
    }
}

public class LinkedList<T>
{
    private Node<T> head;
    private int count;

    public LinkedList()
    {
        head = null;
        count = 0;
    }

    // Add element at the beginning
    public void AddFirst(T data)
    {
        Node<T> newNode = new Node<T>(data);
        newNode.Next = head;
        head = newNode;
        count++;
    }

    // Add element at the end
    public void AddLast(T data)
    {
        Node<T> newNode = new Node<T>(data);

        if (head == null)
        {
            head = newNode;
        }
        else
        {
```

```
        Node<T> current = head;
        while (current.Next != null)
        {
            current = current.Next;
        }
        current.Next = newNode;
    }
    count++;
}

// Remove first occurrence of data
public bool Remove(T data)
{
    if (head == null) return false;

    // If head needs to be removed
    if (head.Data.Equals(data))
    {
        head = head.Next;
        count--;
        return true;
    }

    Node<T> current = head;
    while (current.Next != null)
    {
        if (current.Next.Data.Equals(data))
        {
            current.Next = current.Next.Next;
            count--;
            return true;
        }
        current = current.Next;
    }

    return false;
}

// Check if element exists
public bool Contains(T data)
{
    Node<T> current = head;
    while (current != null)
    {
```

```
            if (current.Data.Equals(data))
                return true;
            current = current.Next;
        }
        return false;
    }

    // Display all elements
    public void Display()
    {
        if (head == null)
        {
            Console.WriteLine("List is empty");
            return;
        }

        Node<T> current = head;
        while (current != null)
        {
            Console.Write($"{current.Data} -> ");
            current = current.Next;
        }
        Console.WriteLine("null");
    }

    public int Count => count;
}

// Usage example
static void Main()
{
    LinkedList<int> list = new LinkedList<int>();

    list.AddFirst(10);
    list.AddFirst(20);
    list.AddLast(30);
    list.AddLast(40);

    list.Display(); // Output: 20 -> 10 -> 30 -> 40 -> null

    Console.WriteLine($"Contains 30: {list.Contains(30)}");

    list.Remove(10);
```

```
    list.Display(); // Output: 20 -> 30 -> 40 -> null
}
```

Trees

What is a Tree?

A tree is a hierarchical data structure with nodes connected by edges. It has a root node and child nodes, forming a parent-child relationship.

Tree Terminology

- **Root**: Top node with no parent
- **Parent**: Node with children
- **Child**: Node with a parent
- **Leaf**: Node with no children
- **Height**: Length of longest path from root to leaf
- **Depth**: Length of path from root to a specific node

Binary Tree

A binary tree is a tree where each node has at most two children (left and right).

Binary Tree Implementation

```
csharp
```

```
public class TreeNode<T>
{
    public T Data { get; set; }
    public TreeNode<T> Left { get; set; }
    public TreeNode<T> Right { get; set; }

    public TreeNode(T data)
    {
        Data = data;
        Left = null;
        Right = null;
    }
}

public class BinaryTree<T>
{
    private TreeNode<T> root;

    public BinaryTree()
    {
        root = null;
    }

    public BinaryTree(T rootData)
    {
        root = new TreeNode<T>(rootData);
    }

    // Tree traversal methods
    public void PreOrderTraversal(TreeNode<T> node)
    {
        if (node != null)
        {
            Console.Write($"{node.Data} ");
            PreOrderTraversal(node.Left);
            PreOrderTraversal(node.Right);
        }
    }

    public void InOrderTraversal(TreeNode<T> node)
    {
        if (node != null)
        {
```

```
            InOrderTraversal(node.Left);
            Console.Write($"{node.Data} ");
            InOrderTraversal(node.Right);
        }
    }

    public void PostOrderTraversal(TreeNode<T> node)
    {
        if (node != null)
        {
            PostOrderTraversal(node.Left);
            PostOrderTraversal(node.Right);
            Console.Write($"{node.Data} ");
        }
    }

    public TreeNode<T> GetRoot()
    {
        return root;
    }

    public void SetRoot(TreeNode<T> node)
    {
        root = node;
    }
}

// Example usage
static void Main()
{
    BinaryTree<int> tree = new BinaryTree<int>();

    // Manually building a tree
    //       1
    //      / \
    //     2   3
    //    / \
    //   4   5

    TreeNode<int> root = new TreeNode<int>(1);
    root.Left = new TreeNode<int>(2);
    root.Right = new TreeNode<int>(3);
    root.Left.Left = new TreeNode<int>(4);
    root.Left.Right = new TreeNode<int>(5);
```

```
    tree.SetRoot(root);

    Console.WriteLine("Pre-order traversal:");
    tree.PreOrderTraversal(tree.GetRoot()); // Output: 1 2 4 5 3

    Console.WriteLine("\nIn-order traversal:");
    tree.InOrderTraversal(tree.GetRoot()); // Output: 4 2 5 1 3

    Console.WriteLine("\nPost-order traversal:");
    tree.PostOrderTraversal(tree.GetRoot()); // Output: 4 5 2 3 1
}
```

Basic Sorting Algorithms

1. Bubble Sort

Repeatedly steps through the list, compares adjacent elements and swaps them if they're in the wrong order.

```csharp
```

```csharp
public static void BubbleSort(int[] arr)
{
    int n = arr.Length;

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

        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                // Swap elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }

        // If no swapping occurred, array is sorted
        if (!swapped)
            break;
    }
}

// Usage
int[] numbers = {64, 34, 25, 12, 22, 11, 90};
Console.WriteLine("Original: " + string.Join(", ", numbers));

BubbleSort(numbers);
Console.WriteLine("Sorted: " + string.Join(", ", numbers));
// Output: 11, 12, 22, 25, 34, 64, 90
```

2. Selection Sort

Finds the minimum element and places it at the beginning, then repeats for the remaining array.

```csharp
```

```csharp
public static void SelectionSort(int[] arr)
{
    int n = arr.Length;

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

        // Find minimum element in remaining array
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }

        // Swap minimum element with first element
        if (minIndex != i)
        {
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}
```

3. Insertion Sort

Builds the sorted array one element at a time by inserting each element into its correct position.

```csharp
```

```csharp
public static void InsertionSort(int[] arr)
{
    for (int i = 1; i < arr.Length; i++)
    {
        int key = arr[i];
        int j = i - 1;

        // Move elements greater than key one position ahead
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}
```

Basic Search Algorithms

1. Linear Search

Searches through each element one by one until the target is found.

```
csharp
```

```csharp
public static int LinearSearch(int[] arr, int target)
{
    for (int i = 0; i < arr.Length; i++)
    {
        if (arr[i] == target)
        {
            return i; // Return index of found element
        }
    }
    return -1; // Element not found
}

// Usage
int[] numbers = {10, 25, 30, 45, 50};
int index = LinearSearch(numbers, 30);

if (index != -1)
    Console.WriteLine($"Found 30 at index {index}");
else
    Console.WriteLine("Element not found");
```

2. Binary Search

Efficiently searches a sorted array by repeatedly dividing the search space in half.

```csharp
```

```csharp
public static int BinarySearch(int[] arr, int target)
{
    int left = 0;
    int right = arr.Length - 1;

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

        if (arr[mid] == target)
            return mid;

        if (arr[mid] < target)
            left = mid + 1;  // Search right half
        else
            right = mid - 1; // Search left half
    }

    return -1; // Element not found
}

// Usage (array must be sorted)
int[] sortedNumbers = {10, 25, 30, 45, 50, 60, 75};
int index = BinarySearch(sortedNumbers, 45);

if (index != -1)
    Console.WriteLine($"Found 45 at index {index}");
else
    Console.WriteLine("Element not found");
```

Time and Space Complexity

Time Complexity Comparison

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$

Space Complexity

- **Arrays**: O(1) - fixed size
- **Dynamic Arrays/Lists**: O(n) - grows with elements
- **Linked Lists**: O(n) - one node per element
- **Stacks/Queues**: O(n) - depends on number of elements

Practice Problems

Problem 1: Array Rotation

Rotate an array to the right by k positions.

```csharp
```

```
public static void RotateArray(int[] arr, int k)
{
    int n = arr.Length;
    k = k % n; // Handle k > n

    // Reverse entire array
    Reverse(arr, 0, n - 1);

    // Reverse first k elements
    Reverse(arr, 0, k - 1);

    // Reverse remaining elements
    Reverse(arr, k, n - 1);
}

private static void Reverse(int[] arr, int start, int end)
{
    while (start < end)
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

// Test
int[] arr = {1, 2, 3, 4, 5, 6, 7};
RotateArray(arr, 3);
Console.WriteLine(string.Join(", ", arr)); // Output: 5, 6, 7, 1, 2, 3, 4
```

Problem 2: Valid Anagram

Check if two strings are anagrams of each other.

```
csharp
```

```csharp
public static bool IsAnagram(string s1, string s2)
{
    if (s1.Length != s2.Length)
        return false;

    Dictionary<char, int> charCount = new Dictionary<char, int>();

    // Count characters in first string
    foreach (char c in s1.ToLower())
    {
        charCount[c] = charCount.GetValueOrDefault(c, 0) + 1;
    }

    // Decrease count for characters in second string
    foreach (char c in s2.ToLower())
    {
        if (!charCount.ContainsKey(c))
            return false;

        charCount[c]--;
        if (charCount[c] == 0)
            charCount.Remove(c);
    }

    return charCount.Count == 0;
}

// Test
Console.WriteLine(IsAnagram("listen", "silent")); // True
Console.WriteLine(IsAnagram("hello", "world"));   // False
```

Problem 3: Two Sum Problem

Find two numbers in an array that add up to a target sum.

```
csharp
```

```csharp
public static int[] TwoSum(int[] nums, int target)
{
    Dictionary<int, int> map = new Dictionary<int, int>();

    for (int i = 0; i < nums.Length; i++)
    {
        int complement = target - nums[i];

        if (map.ContainsKey(complement))
        {
            return new int[] { map[complement], i };
        }

        map[nums[i]] = i;
    }

    return new int[] {}; // No solution found
}

// Test
int[] nums = {2, 7, 11, 15};
int target = 9;
int[] result = TwoSum(nums, target);
Console.WriteLine($"Indices: [{result[0]}, {result[1]}]"); // Output: [0, 1]
```

Next Steps

Intermediate Topics to Explore

1. **Advanced Data Structures**
    - Hash Tables/Dictionaries
    - Heaps
    - Graphs
    - Tries
2. **Advanced Algorithms**
    - Quick Sort and Merge Sort
    - Dynamic Programming
    - Breadth-First Search (BFS)
    - Depth-First Search (DFS)

3. **Algorithm Design Techniques**
   - Divide and Conquer
   - Greedy Algorithms
   - Backtracking

Recommended Resources

- **Books**: "Introduction to Algorithms" by Cormen
- **Online Platforms**: LeetCode, HackerRank, Codewars
- **Practice**: Start with easy problems and gradually increase difficulty

Tips for Success

1. **Practice Regularly**: Solve at least one problem daily
2. **Understand, Don't Memorize**: Focus on understanding the logic
3. **Start Simple**: Master basic concepts before moving to advanced topics
4. **Code by Hand**: Practice writing code without an IDE sometimes
5. **Review Complexity**: Always analyze time and space complexity

---

Advanced Data Structures

Hash Tables (Dictionaries in C#)

What is a Hash Table?

A hash table stores key-value pairs and provides very fast lookup, insertion, and deletion operations (average O(1) time complexity).

When to Use Hash Tables

- Fast lookups by key
- Counting occurrences
- Caching/memoization
- Removing duplicates

C# Dictionary Examples

```
csharp
```

```
using System.Collections.Generic;

// Creating a dictionary
Dictionary<string, int> ages = new Dictionary<string, int>();

// Adding key-value pairs
ages["Alice"] = 25;
ages["Bob"] = 30;
ages.Add("Charlie", 35);

// Accessing values
Console.WriteLine($"Alice's age: {ages["Alice"]}");

// Safe access with TryGetValue
if (ages.TryGetValue("David", out int davidAge))
{
    Console.WriteLine($"David's age: {davidAge}");
}
else
{
    Console.WriteLine("David not found");
}

// Checking if key exists
if (ages.ContainsKey("Bob"))
{
    Console.WriteLine("Bob exists in dictionary");
}

// Iterating through dictionary
foreach (var kvp in ages)
{
    Console.WriteLine($"{kvp.Key}: {kvp.Value}");
}

// Removing elements
ages.Remove("Charlie");
```

Practical Example: Word Counter

```
csharp
```

```
public class WordCounter
{
    public static Dictionary<string, int> CountWords(string text)
    {
        Dictionary<string, int> wordCount = new Dictionary<string, int>();

        // Split text into words and convert to lowercase
        string[] words = text.ToLower()
            .Split(new char[] { ' ', '.', ',', '!', '?', ';', ':' },
                StringSplitOptions.RemoveEmptyEntries);

        foreach (string word in words)
        {
            if (wordCount.ContainsKey(word))
            {
                wordCount[word]++;
            }
            else
            {
                wordCount[word] = 1;
            }

            // Or use this shorter version:
            // wordCount[word] = wordCount.GetValueOrDefault(word, 0) + 1;
        }

        return wordCount;
    }

    public static void DisplayTopWords(Dictionary<string, int> wordCount, int topN)
    {
        var sortedWords = wordCount.OrderByDescending(kvp => kvp.Value).Take(topN);

        Console.WriteLine($"Top {topN} most frequent words:");
        int rank = 1;
        foreach (var kvp in sortedWords)
        {
            Console.WriteLine($"{rank}. {kvp.Key}: {kvp.Value} times");
            rank++;
        }
    }
}
```

```
// Usage
string text = "The quick brown fox jumps over the lazy dog. The dog was really lazy.";
var wordCount = WordCounter.CountWords(text);
WordCounter.DisplayTopWords(wordCount, 5);
```

Recursion

What is Recursion?

Recursion is when a function calls itself to solve a smaller version of the same problem.

Components of Recursion

1. **Base Case**: The condition that stops the recursion
2. **Recursive Case**: The function calling itself with modified parameters

Simple Recursion Examples

```
csharp
```

```
// Factorial: n! = n × (n-1)!
public static int Factorial(int n)
{
    // Base case
    if (n <= 1)
        return 1;

    // Recursive case
    return n * Factorial(n - 1);
}

// Fibonacci: F(n) = F(n-1) + F(n-2)
public static int Fibonacci(int n)
{
    // Base cases
    if (n <= 1)
        return n;

    // Recursive case
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}

// Sum of array elements
public static int SumArray(int[] arr, int index = 0)
{
    // Base case
    if (index >= arr.Length)
        return 0;

    // Recursive case
    return arr[index] + SumArray(arr, index + 1);
}

// Usage
Console.WriteLine($"5! = {Factorial(5)}"); // Output: 120
Console.WriteLine($"Fibonacci(6) = {Fibonacci(6)}"); // Output: 8

int[] numbers = {1, 2, 3, 4, 5};
Console.WriteLine($"Sum = {SumArray(numbers)}"); // Output: 15
```

Tree Recursion Example: Binary Tree Operations

```csharp
```

```csharp
public class BinaryTreeOperations
{
    // Calculate height of tree
    public static int GetHeight(TreeNode<int> node)
    {
        if (node == null)
            return -1; // or 0, depending on definition

        int leftHeight = GetHeight(node.Left);
        int rightHeight = GetHeight(node.Right);

        return 1 + Math.Max(leftHeight, rightHeight);
    }

    // Count total nodes in tree
    public static int CountNodes(TreeNode<int> node)
    {
        if (node == null)
            return 0;

        return 1 + CountNodes(node.Left) + CountNodes(node.Right);
    }

    // Find maximum value in tree
    public static int FindMax(TreeNode<int> node)
    {
        if (node == null)
            throw new ArgumentException("Tree is empty");

        int max = node.Data;

        if (node.Left != null)
            max = Math.Max(max, FindMax(node.Left));

        if (node.Right != null)
            max = Math.Max(max, FindMax(node.Right));

        return max;
    }

    // Check if tree contains a value
    public static bool Contains(TreeNode<int> node, int value)
    {
```

```
        if (node == null)
            return false;

        if (node.Data == value)
            return true;

        return Contains(node.Left, value) || Contains(node.Right, value);
    }
}
```

Advanced Sorting Algorithms

Merge Sort

Divide and conquer algorithm that splits the array in half, sorts each half, then merges them back together.

```csharp
```

```csharp
public static void MergeSort(int[] arr, int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        MergeSort(arr, left, mid);
        MergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        Merge(arr, left, mid, right);
    }
}

private static void Merge(int[] arr, int left, int mid, int right)
{
    // Create temporary arrays
    int[] leftArray = new int[mid - left + 1];
    int[] rightArray = new int[right - mid];

    // Copy data to temporary arrays
    Array.Copy(arr, left, leftArray, 0, leftArray.Length);
    Array.Copy(arr, mid + 1, rightArray, 0, rightArray.Length);

    // Merge the temporary arrays back into arr[left..right]
    int i = 0, j = 0, k = left;

    while (i < leftArray.Length && j < rightArray.Length)
    {
        if (leftArray[i] <= rightArray[j])
        {
            arr[k] = leftArray[i];
            i++;
        }
        else
        {
            arr[k] = rightArray[j];
            j++;
        }
        k++;
    }
```

```csharp
    // Copy remaining elements
    while (i < leftArray.Length)
    {
        arr[k] = leftArray[i];
        i++;
        k++;
    }

    while (j < rightArray.Length)
    {
        arr[k] = rightArray[j];
        j++;
        k++;
    }
}

// Usage
int[] arr = {38, 27, 43, 3, 9, 82, 10};
Console.WriteLine("Original: " + string.Join(", ", arr));

MergeSort(arr, 0, arr.Length - 1);
Console.WriteLine("Sorted: " + string.Join(", ", arr));
```

Quick Sort

Another divide and conquer algorithm that picks a pivot element and partitions the array around it.

```csharp
```

```
public static void QuickSort(int[] arr, int low, int high)
{
    if (low < high)
    {
        // Partition the array and get pivot index
        int pivotIndex = Partition(arr, low, high);

        // Recursively sort elements before and after partition
        QuickSort(arr, low, pivotIndex - 1);
        QuickSort(arr, pivotIndex + 1, high);
    }
}

private static int Partition(int[] arr, int low, int high)
{
    int pivot = arr[high]; // Choose last element as pivot
    int i = low - 1; // Index of smaller element

    for (int j = low; j < high; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            Swap(arr, i, j);
        }
    }

    Swap(arr, i + 1, high);
    return i + 1;
}

private static void Swap(int[] arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

// Usage
int[] arr2 = {10, 7, 8, 9, 1, 5};
Console.WriteLine("Original: " + string.Join(", ", arr2));
```

```
QuickSort(arr2, 0, arr2.Length - 1);
Console.WriteLine("Sorted: " + string.Join(", ", arr2));
```

Graph Basics

What is a Graph?

A graph consists of vertices (nodes) connected by edges. Graphs can represent relationships, networks, paths, and many other structures.

Types of Graphs

- **Directed vs Undirected**: Edges have direction or not
- **Weighted vs Unweighted**: Edges have weights/costs or not
- **Connected vs Disconnected**: All nodes reachable or not

Graph Representation in C#

```
csharp
```

```
public class Graph
{
    private Dictionary<int, List<int>> adjacencyList;

    public Graph()
    {
        adjacencyList = new Dictionary<int, List<int>>();
    }

    // Add vertex
    public void AddVertex(int vertex)
    {
        if (!adjacencyList.ContainsKey(vertex))
        {
            adjacencyList[vertex] = new List<int>();
        }
    }

    // Add edge (undirected)
    public void AddEdge(int source, int destination)
    {
        AddVertex(source);
        AddVertex(destination);

        adjacencyList[source].Add(destination);
        adjacencyList[destination].Add(source); // For undirected graph
    }

    // Display graph
    public void DisplayGraph()
    {
        foreach (var vertex in adjacencyList)
        {
            Console.Write($"{vertex.Key}: ");
            Console.WriteLine(string.Join(", ", vertex.Value));
        }
    }

    // Breadth-First Search (BFS)
    public void BFS(int startVertex)
    {
        HashSet<int> visited = new HashSet<int>();
        Queue<int> queue = new Queue<int>();
```

```
        visited.Add(startVertex);
        queue.Enqueue(startVertex);

        Console.Write("BFS traversal: ");

        while (queue.Count > 0)
        {
            int vertex = queue.Dequeue();
            Console.Write($"{vertex} ");

            foreach (int neighbor in adjacencyList[vertex])
            {
                if (!visited.Contains(neighbor))
                {
                    visited.Add(neighbor);
                    queue.Enqueue(neighbor);
                }
            }
        }
        Console.WriteLine();
    }

    // Depth-First Search (DFS)
    public void DFS(int startVertex)
    {
        HashSet<int> visited = new HashSet<int>();
        Console.Write("DFS traversal: ");
        DFSHelper(startVertex, visited);
        Console.WriteLine();
    }

    private void DFSHelper(int vertex, HashSet<int> visited)
    {
        visited.Add(vertex);
        Console.Write($"{vertex} ");

        foreach (int neighbor in adjacencyList[vertex])
        {
            if (!visited.Contains(neighbor))
            {
                DFSHelper(neighbor, visited);
            }
        }
```

```csharp
    }
}

// Usage
Graph graph = new Graph();
graph.AddEdge(0, 1);
graph.AddEdge(0, 2);
graph.AddEdge(1, 3);
graph.AddEdge(1, 4);
graph.AddEdge(2, 5);
graph.AddEdge(2, 6);

graph.DisplayGraph();
graph.BFS(0); // Output: BFS traversal: 0 1 2 3 4 5 6
graph.DFS(0); // Output: DFS traversal: 0 1 3 4 2 5 6
```

Dynamic Programming Basics

What is Dynamic Programming?

Dynamic Programming (DP) is an optimization technique that solves complex problems by breaking them down into simpler subproblems and storing the results to avoid redundant calculations.

When to Use Dynamic Programming

- Problem has overlapping subproblems
- Problem has optimal substructure
- You can define the problem recursively

Classic DP Example: Fibonacci with Memoization

```csharp
```

```
public class DynamicProgramming
{
    // Fibonacci with memoization (top-down DP)
    private static Dictionary<int, long> fibMemo = new Dictionary<int, long>();

    public static long FibonacciMemo(int n)
    {
        if (n <= 1) return n;

        if (fibMemo.ContainsKey(n))
            return fibMemo[n];

        fibMemo[n] = FibonacciMemo(n - 1) + FibonacciMemo(n - 2);
        return fibMemo[n];
    }

    // Fibonacci with tabulation (bottom-up DP)
    public static long FibonacciTab(int n)
    {
        if (n <= 1) return n;

        long[] dp = new long[n + 1];
        dp[0] = 0;
        dp[1] = 1;

        for (int i = 2; i <= n; i++)
        {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }

    // Coin Change Problem
    public static int CoinChange(int[] coins, int amount)
    {
        int[] dp = new int[amount + 1];
        Array.Fill(dp, amount + 1); // Initialize with max value
        dp[0] = 0; // Base case

        for (int i = 1; i <= amount; i++)
        {
            foreach (int coin in coins)
```

```
            {
                if (coin <= i)
                {
                    dp[i] = Math.Min(dp[i], dp[i - coin] + 1);
                }
            }
        }

        return dp[amount] > amount ? -1 : dp[amount];
    }

    // Longest Common Subsequence
    public static int LCS(string text1, string text2)
    {
        int m = text1.Length;
        int n = text2.Length;
        int[,] dp = new int[m + 1, n + 1];

        for (int i = 1; i <= m; i++)
        {
            for (int j = 1; j <= n; j++)
            {
                if (text1[i - 1] == text2[j - 1])
                {
                    dp[i, j] = dp[i - 1, j - 1] + 1;
                }
                else
                {
                    dp[i, j] = Math.Max(dp[i - 1, j], dp[i, j - 1]);
                }
            }
        }

        return dp[m, n];
    }
}

// Usage examples
Console.WriteLine($"Fibonacci(40) = {DynamicProgramming.FibonacciMemo(40)}");

int[] coins = {1, 3, 4};
int amount = 6;
Console.WriteLine($"Minimum coins for {amount}: {DynamicProgramming.CoinChange(coins, amount)}");
```

```
string text1 = "abcde";
string text2 = "ace";
Console.WriteLine($"LCS length: {DynamicProgramming.LCS(text1, text2)}");
```

More Practice Problems

Problem 4: Remove Duplicates from Sorted Array

csharp

```csharp
public static int RemoveDuplicates(int[] nums)
{
    if (nums.Length == 0) return 0;

    int writeIndex = 1;

    for (int readIndex = 1; readIndex < nums.Length; readIndex++)
    {
        if (nums[readIndex] != nums[readIndex - 1])
        {
            nums[writeIndex] = nums[readIndex];
            writeIndex++;
        }
    }

    return writeIndex;
}

// Test
int[] nums = {1, 1, 2, 2, 2, 3, 4, 4, 5};
int newLength = RemoveDuplicates(nums);
Console.WriteLine($"New length: {newLength}");
Console.WriteLine("Array: " + string.Join(", ", nums.Take(newLength)));
// Output: New length: 5, Array: 1, 2, 3, 4, 5
```

Problem 5: Valid Palindrome

csharp

```csharp
public static bool IsPalindrome(string s)
{
    int left = 0;
    int right = s.Length - 1;

    while (left < right)
    {
        // Skip non-alphanumeric characters
        while (left < right && !char.IsLetterOrDigit(s[left]))
            left++;

        while (left < right && !char.IsLetterOrDigit(s[right]))
            right--;

        // Compare characters (case-insensitive)
        if (char.ToLower(s[left]) != char.ToLower(s[right]))
            return false;

        left++;
        right--;
    }

    return true;
}

// Test
Console.WriteLine(IsPalindrome("A man, a plan, a canal: Panama")); // True
Console.WriteLine(IsPalindrome("race a car")); // False
```

Problem 6: Maximum Subarray Sum (Kadane's Algorithm)

csharp

```
public static int MaxSubarraySum(int[] nums)
{
    int maxSoFar = nums[0];
    int maxEndingHere = nums[0];

    for (int i = 1; i < nums.Length; i++)
    {
        // Either extend existing subarray or start new one
        maxEndingHere = Math.Max(nums[i], maxEndingHere + nums[i]);

        // Update global maximum
        maxSoFar = Math.Max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
}

// Test
int[] nums1 = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
Console.WriteLine($"Maximum subarray sum: {MaxSubarraySum(nums1)}"); // Output: 6
```

Problem 7: Reverse Linked List

```csharp
```

```csharp
public static Node<T> ReverseLinkedList<T>(Node<T> head)
{
    Node<T> prev = null;
    Node<T> current = head;

    while (current != null)
    {
        Node<T> nextTemp = current.Next;
        current.Next = prev;
        prev = current;
        current = nextTemp;
    }

    return prev; // New head
}

// Test with our LinkedList class
LinkedList<int> list = new LinkedList<int>();
list.AddLast(1);
list.AddLast(2);
list.AddLast(3);
list.AddLast(4);

Console.WriteLine("Original:");
list.Display(); // 1 -> 2 -> 3 -> 4 -> null

// Note: This requires modifying our LinkedList class to expose the head
// or creating a separate method
```

Best Practices and Interview Tips

Code Quality Best Practices

```csharp
```

```csharp
// Good: Clear variable names and comments
public static int FindFirstDuplicate(int[] numbers)
{
    HashSet<int> seen = new HashSet<int>();

    foreach (int number in numbers)
    {
        if (seen.Contains(number))
        {
            return number; // Found first duplicate
        }
        seen.Add(number);
    }

    return -1; // No duplicate found
}

// Handle edge cases
public static int BinarySearchSafe(int[] arr, int target)
{
    // Edge case: null or empty array
    if (arr == null || arr.Length == 0)
        return -1;

    int left = 0;
    int right = arr.Length - 1;

    while (left <= right)
    {
        // Avoid integer overflow
        int mid = left + (right - left) / 2;

        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }

    return -1;
}
```

Interview Problem-Solving Strategy

1. **Understand the Problem**
   - Ask clarifying questions
   - Work through examples
   - Identify edge cases
2. **Plan Your Approach**
   - Start with brute force
   - Think of optimizations
   - Consider time/space tradeoffs
3. **Code Systematically**
   - Write clean, readable code
   - Use meaningful variable names
   - Handle edge cases
4. **Test Your Solution**
   - Walk through with examples
   - Consider edge cases
   - Analyze complexity

Common Interview Questions by Category

Arrays & Strings:

- Two Sum, Three Sum
- Merge Intervals
- String Permutations
- Sliding Window problems

Linked Lists:

- Reverse Linked List
- Detect Cycle
- Merge Two Sorted Lists
- Remove Nth Node from End

Trees & Graphs:

- Tree Traversals
- Validate Binary Search Tree
- Lowest Common Ancestor
- Graph BFS/DFS

Dynamic Programming:

- Climbing Stairs
- Coin Change
- Longest Palindromic Substring
- Edit Distance

Sorting & Searching:

- Merge Sort, Quick Sort
- Binary Search variations
- Find Peak Element
- Search in Rotated Array

Performance Optimization Tips

Memory Management

```csharp
```

```
// Use StringBuilder for string concatenation
public static string JoinWords(string[] words)
{
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < words.Length; i++)
    {
        sb.Append(words[i]);
        if (i < words.Length - 1)
            sb.Append(" ");
    }

    return sb.ToString();
}

// Use capacity when you know the size
List<int> numbers = new List<int>(1000); // Pre-allocate capacity
```

Algorithm Optimization

```
csharp
```

```csharp
// Use early termination when possible
public static bool ContainsSum(int[] arr, int targetSum)
{
    HashSet<int> seen = new HashSet<int>();

    foreach (int num in arr)
    {
        int complement = targetSum - num;
        if (seen.Contains(complement))
        {
            return true; // Found pair, no need to continue
        }
        seen.Add(num);
    }

    return false;
}

// Cache expensive computations
private static Dictionary<string, int> stringLengthCache = new Dictionary<string, int>();

public static int GetExpensiveStringLength(string input)
{
    if (stringLengthCache.ContainsKey(input))
    {
        return stringLengthCache[input];
    }

    // Simulate expensive computation
    int length = input.Length;
    stringLengthCache[input] = length;
    return length;
}
```