Exercises: Traverse a Graph

This document defines the **in-class exercises** assignments for the "Data Structures" course @ Software University.

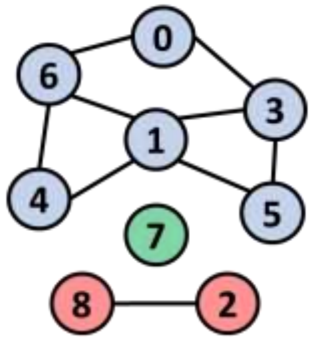
Tra verse a Graph to Find Its Connected Components

The first part of this lab aims to implement the **DFS algorithm** (Depth-First-Search) to **traverse a graph** and find its **connected components** (nodes connected to each other either directly, or through other nodes). The graph nodes are numbered from **`0`** to **`n-1`**. The graph comes from the console in the following format:

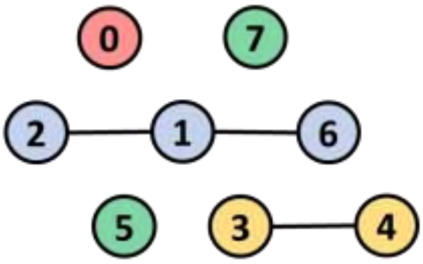
- First line: number of lines **n**
- Next **n** lines: list of child nodes for the nodes **`0`** … **`n-1`** (separated by a space)

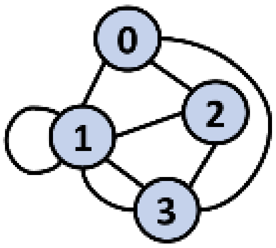
Print the connected components in the same format as in the examples below:

Input	Graph	Output

9
3 6
3 4 5 6
8
0 1 5
1 6
1 3
0 1 4
 	
2
		Connected component: 6 4 5 1 3 0
		Connected component: 8 2
		Connected component: 7

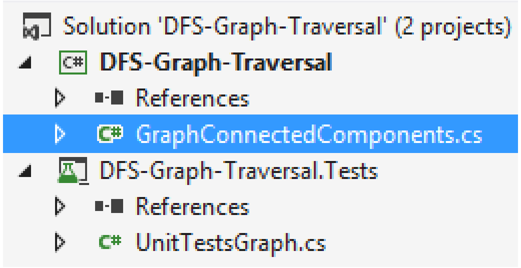
1 0		Connected component: 0
0	(empty graph)	Connected component:

7
 	
2 6
1
4
3
 	
1
		Connected component: 0
		Connected component: 2 6 1
		Connected component: 4 3
		Connected component: 5

4
1 2 3
0 1 2 3 3
0 1 3
0 1 1 2
		Connected component: 3 2 1 0

Problem 1. Graph Traversal – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished class **`GraphConnectedComponents`** and **unit tests** for its functionality. The project holds the following assets:



Solution 'DFS-Graph-Traversal' (2 projects)
- DFS-Graph-Traversal
  - References
  - GraphConnectedComponents.cs
- DFS-Graph-Traversal.Tests
  - References
  - UnitTestsGraph.cs

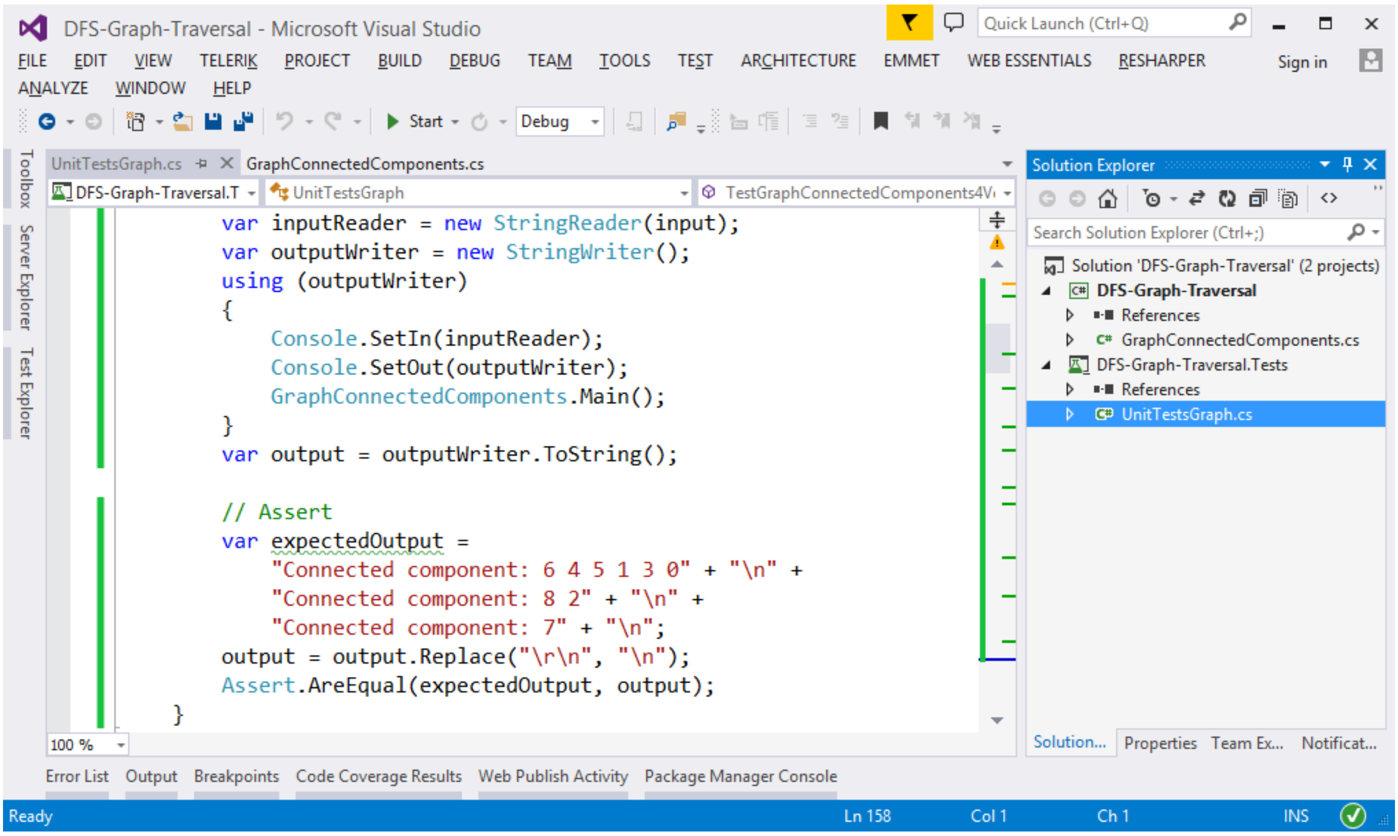
The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**.

The unfinished **`GraphConnectedComponents`** class stays in the file **`GraphConnectedComponents.cs`**:

`GraphConnectedComponents.cs`

```
public class GraphConnectedComponents
{
    public static void Main()
    {
        // TODO: implement me
    }
}
```

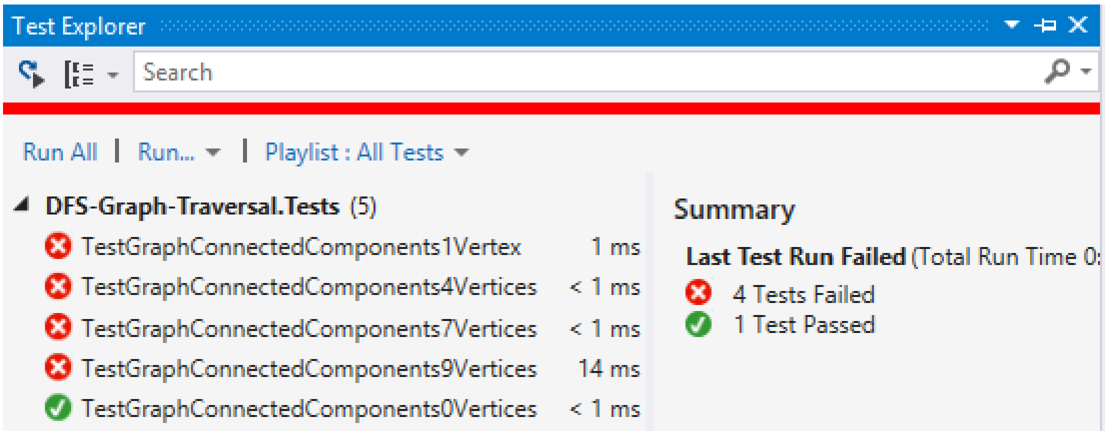
The project comes with **unit tests** covering the functionality of the **`GraphConnectedComponents`** class:



```
        var inputReader = new StringReader(input);
        var outputWriter = new StringWriter();
        using (outputWriter)
        {
            Console.SetIn(inputReader);
            Console.SetOut(outputWriter);
            GraphConnectedComponents.Main();
        }
        var output = outputWriter.ToString();

        // Assert
        var expectedOutput =
            "Connected component: 6 4 5 1 3 0" + "\n" +
            "Connected component: 8 2" + "\n" +
            "Connected component: 7" + "\n";
        output = output.Replace("\r\n", "\n");
        Assert.AreEqual(expectedOutput, output);
    }
```

Problem 2. Run the Unit Tests to Ensure They Initially Fail

Run the unit tests from the **`DFS-Graph-Traversal.Tests`** project. Open the "**Test Explorer**" window (Menu → Test → Windows → Test Explorer) and run all tests. The expected behavior is that all tests should fail:



DFS-Graph-Traversal.Tests (5)

✗ TestGraphConnectedComponents1Vertex	1 ms
✗ TestGraphConnectedComponents4Vertices	< 1 ms
✗ TestGraphConnectedComponents7Vertices	< 1 ms
✗ TestGraphConnectedComponents9Vertices	14 ms
✓ TestGraphConnectedComponents0Vertices	< 1 ms

Summary

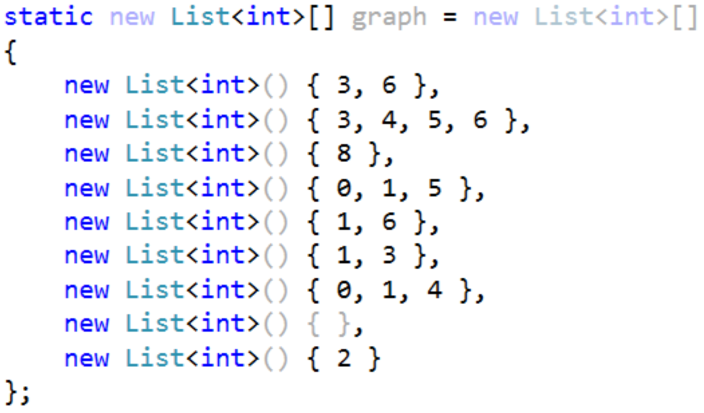
Last Test Run Failed (Total Run Time 0:

✗ 4 Tests Failed
✓ 1 Test Passed

This is quite normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

Problem 3. Define a Sample Graph

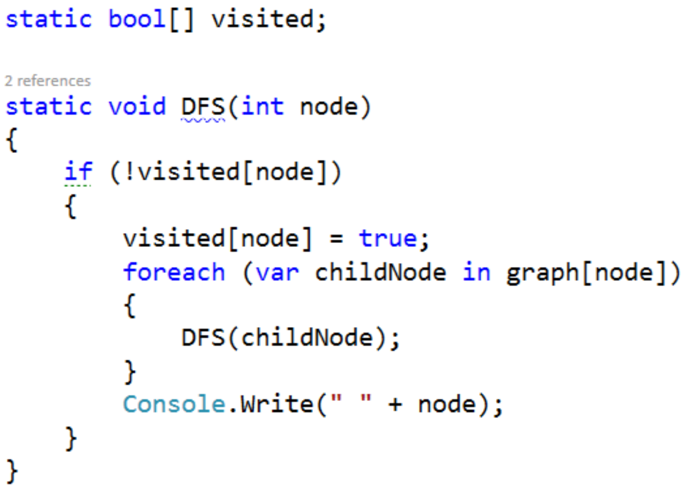
The first step is to define a sample graph. It will be used to test the code during the development:



```
static new List<int>[] graph = new List<int>[]
{
    new List<int>() { 3, 6 },
    new List<int>() { 3, 4, 5, 6 },
    new List<int>() { 8 },
    new List<int>() { 0, 1, 5 },
    new List<int>() { 1, 6 },
    new List<int>() { 1, 3 },
    new List<int>() { 0, 1, 4 },
    new List<int>() { },
    new List<int>() { 2 }
};
```

Problem 4. Implement the DFS Algorithm

The next step is to implement the **DFS** (Depth-First-Search) algorithm to traverse recursively all connected nodes reachable from specified start node:



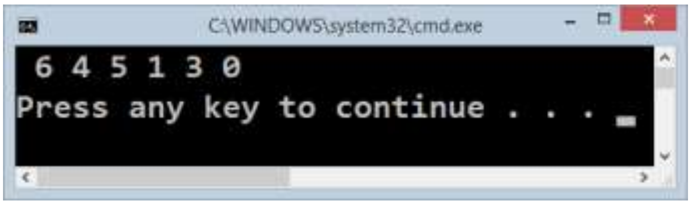
```
static bool[] visited;

2 references
static void DFS(int node)
{
    if (!visited[node])
    {
        visited[node] = true;
        foreach (var childNode in graph[node])
        {
            DFS(childNode);
        }
        Console.Write(" " + node);
    }
}
```

Problem 5. Test the DFS Algorithm

Now, test whether the DFS algorithm implementation. Invoke it starting from node 0. It should print the connected component, holding the node 0:

```
public static void Main()
{
    visited = new bool[graph.Length];
    DFS(0);
    Console.WriteLine();
}
```

Now run the code above. It should find the first connected component in the graph, holding the node 0:



```
 6 4 5 1 3 0
Press any key to continue . . .
```

Problem 6. Find All Connected Components

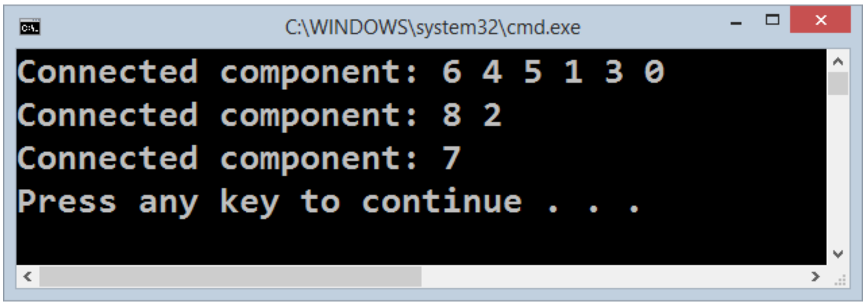
Now, we have DFS algorithm implemented, which finds the connected component holding all nodes reachable from given starting node. This is good, but we want to find all connected components. We can just run the DFS algorithm many times from each node (which was not visited already):

```
static void FindGraphConnectedComponents()
{
    visited = new bool[graph.Length];
    for (int startNode = 0; startNode < graph.Length; startNode++)
    {
        if (!visited[startNode])
        {
            Console.Write("Connected component:");
            DFS(startNode);
            Console.WriteLine();
        }
    }
}
```

Now let's test the above code. Just call it from the main method:

```
public static void Main()
{
    FindGraphConnectedComponents();
}
```

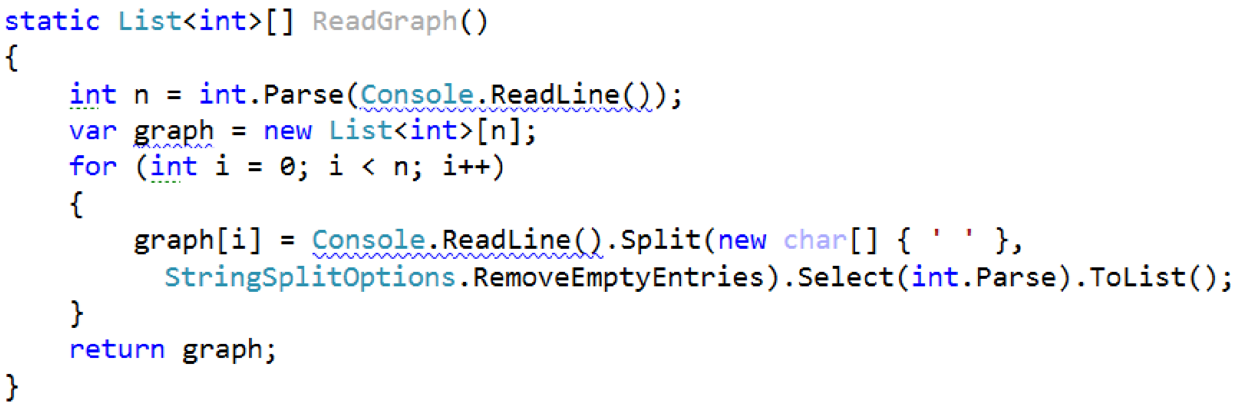
The output is as expected. It prints all connected components in the graph:



```
Connected component: 6 4 5 1 3 0
Connected component: 8 2
Connected component: 7
Press any key to continue . . .
```

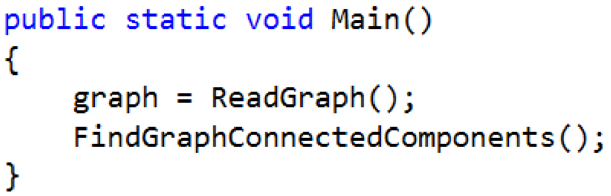
Problem 7. Read the Input Data from the Console

Usually, when we solve problems, we work on hard-coded sample data (in our case the **graph** is hard-coded) and we write the code step by step, test it continuously and finally, when the code is ready and it works well, we change the hard-coded input data with a logic that reads it. Let's implement the data entry logic (read graph from the console):



```
static List<int>[] ReadGraph()
{
    int n = int.Parse(Console.ReadLine());
    var graph = new List<int>[n];
    for (int i = 0; i < n; i++)
    {
        graph[i] = Console.ReadLine().Split(new char[] { ' ' },
          StringSplitOptions.RemoveEmptyEntries).Select(int.Parse).ToList();
    }
    return graph;
}
```

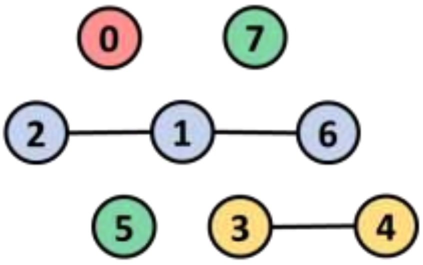
Modify the main method to read the graph from the console instead using the hard-coded graph:



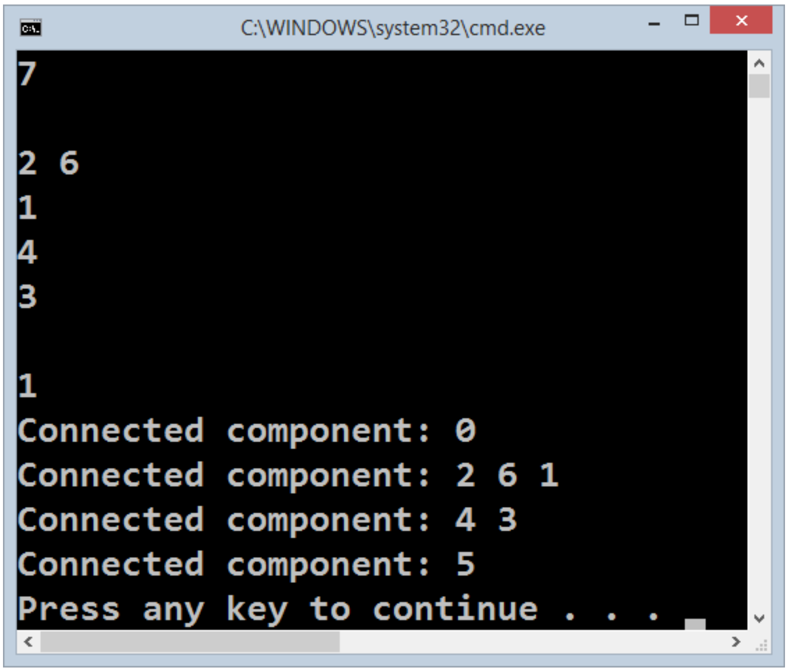
```
public static void Main()
{
    graph = ReadGraph();
    FindGraphConnectedComponents();
}
```

Now test the program. Run it ([Ctrl] + [F5]). Enter a sample graph data and check the output:

Input	Graph	Expected Output

7
 	
2 6
1
4
3
 	
1
		Connected component: 0
		Connected component: 2 6 1
		Connected component: 4 3
		Connected component: 5

Seems like it runs correctly:

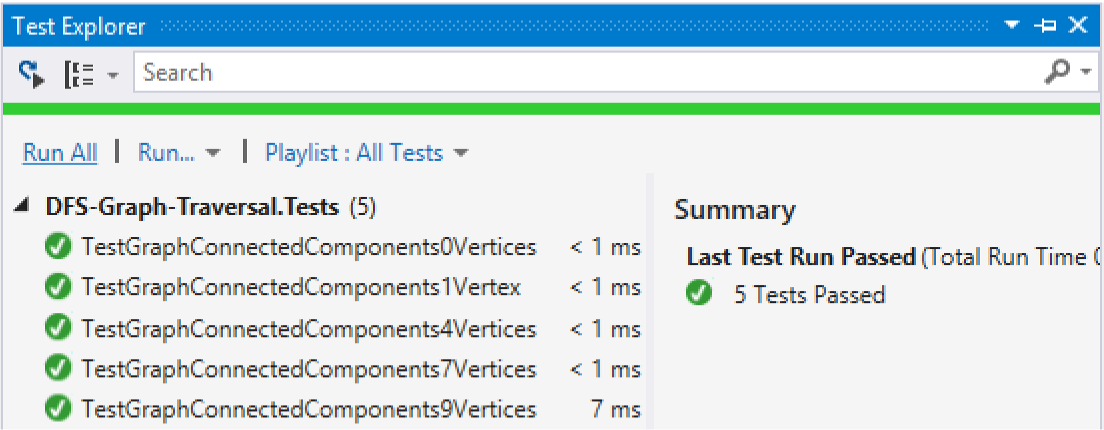


```
7

2 6
1
4
3

1
Connected component: 0
Connected component: 2 6 1
Connected component: 4 3
Connected component: 5
Press any key to continue . . .
```

We are ready for the unit tests.

Problem 8. Run the Unit Tests

Seems like we solved the graph problem. Let's run the unit tests that come with the program skeleton:



```
DFS-Graph-Traversal.Tests (5)
  ✓ TestGraphConnectedComponents0Vertices   < 1 ms
  ✓ TestGraphConnectedComponents1Vertex     < 1 ms
  ✓ TestGraphConnectedComponents4Vertices   < 1 ms
  ✓ TestGraphConnectedComponents7Vertices   < 1 ms
  ✓ TestGraphConnectedComponents9Vertices     7 ms

Summary
Last Test Run Passed (Total Run Time (
✓ 5 Tests Passed
```

Congratulations! You have implemented the DFS algorithm to find all connected components in a graph.