Exercises: Graph Algorithms

This document defines the **in-class exercises** assignments for the ["Algorithms" course @ Software University](https://softuni.bg).

For the following exercises you are given a Visual Studio solution "**Graph-Algorithms-Lab**" holding portions of the source code + unit tests. You can download it from the course's page.

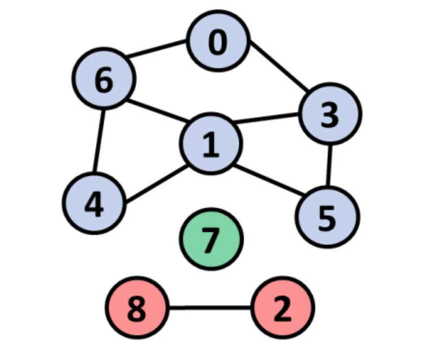
Part I – Traverse a Graph to Find Its Connected Components

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)

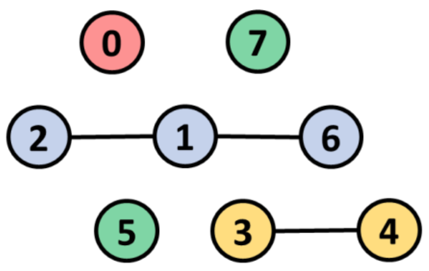
Print the connected components in the same format as in the examples below:

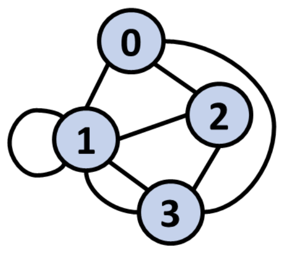
Input	Graph	Output

`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`

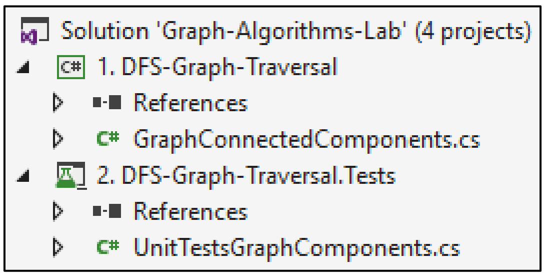
`0`	(empty graph)	`Connected component:`

`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 with DFS – 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 'Graph-Algorithms-Lab' (4 projects)
- 1. DFS-Graph-Traversal
  - References
  - GraphConnectedComponents.cs
- 2. DFS-Graph-Traversal.Tests
  - References
  - UnitTestsGraphComponents.cs

The project skeleton opens correctly in **Visual Studio 2015** 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`**:

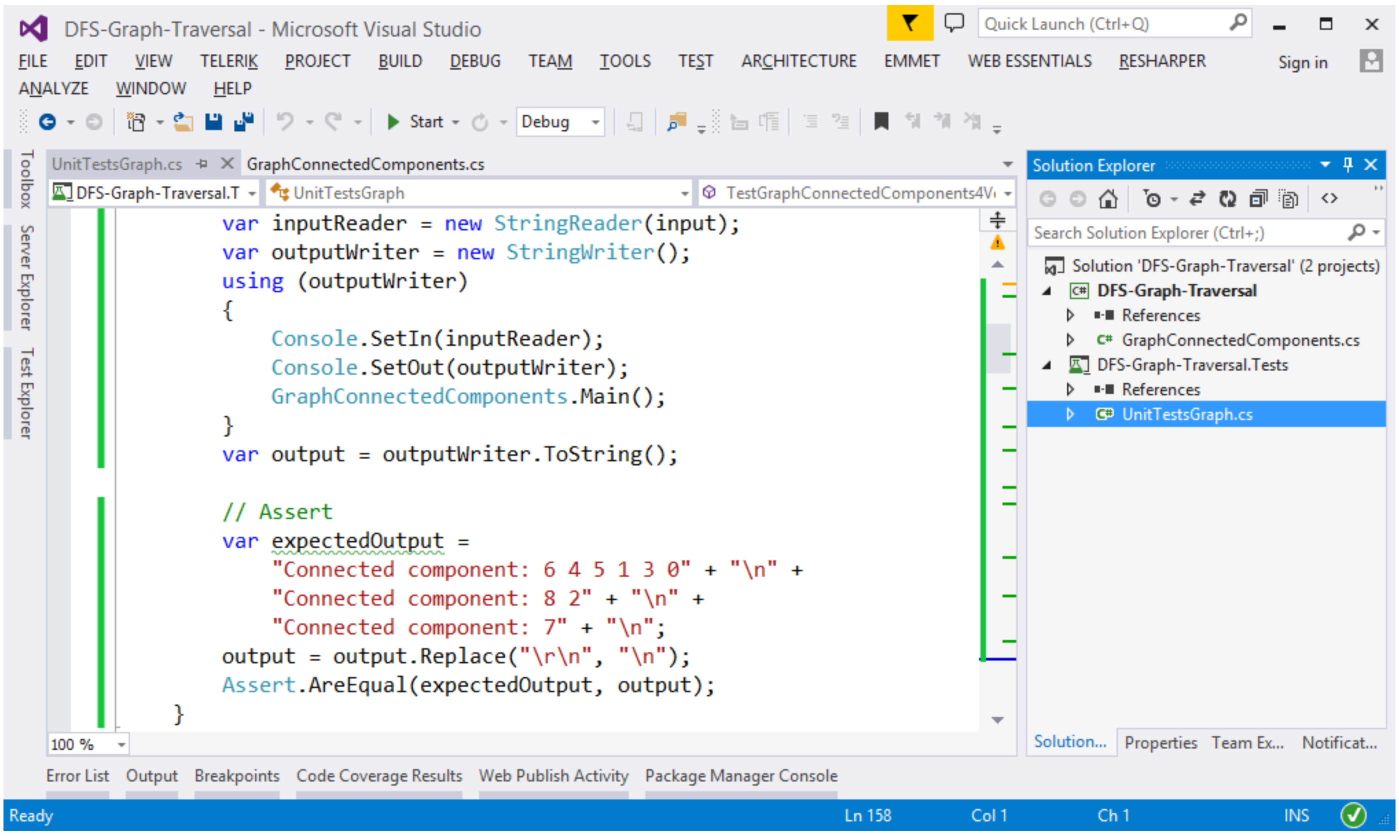
`GraphConnectedComponents.cs`

```
public class GraphConnectedComponents
{
    static 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 }
    };

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

    private 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;
    }

    private static void FindGraphConnectedComponents()
```

```
        {
            // TODO: implement this
        }
    }
```

The project skeleton holds a **hard-coded graph** (the first example from the sample input / output), available for testing purposes during the development. We will first test with this graph, then with the unit tests.

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

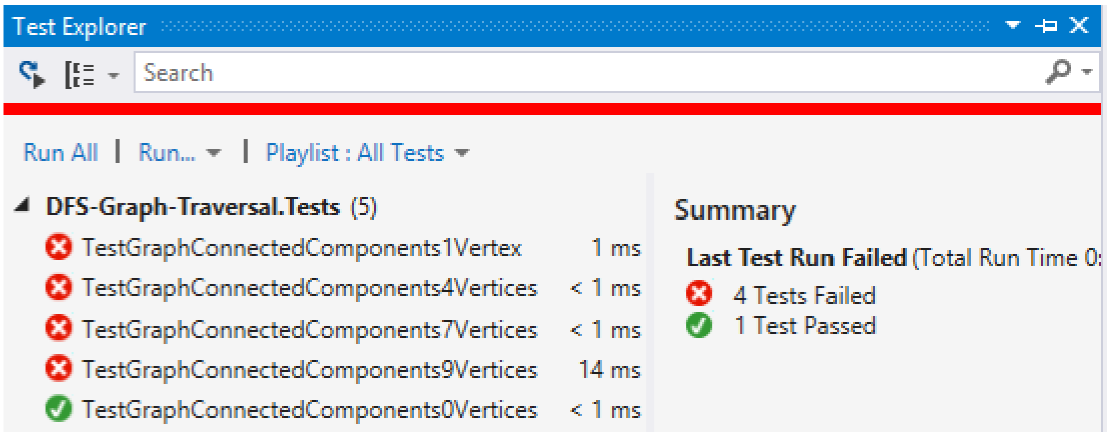


```
            var inputReader = new StringReader(input);
            var outputWriter = new StringWriter();
            using (outputWriter)
            {
                Console.SetIn(inputReader);
                Console.SetOut(outputWriter);
                GraphConnectedComponents.Main();
    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

DFS-Graph-Traversal - Microsoft Visual Studio
UnitTestsGraph.cs
GraphConnectedComponents.cs
DFS-Graph-Traversal.T
UnitTestsGraph
TestGraphConnectedComponents4V
Solution Explorer
Solution 'DFS-Graph-Traversal' (2 projects)
DFS-Graph-Traversal
References
GraphConnectedComponents.cs
DFS-Graph-Traversal.Tests
References
UnitTestsGraph.cs
Ready
Ln 158
Col 1
Ch 1
INS

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. Implement the DFS Algorithm

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

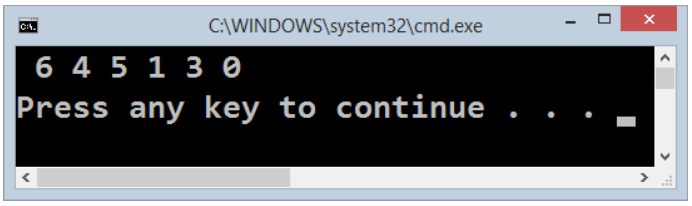
```
static bool[] visited;

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 4. Test the DFS Algorithm

Now, test the DFS algorithm implementation from the console-based C# project. Invoke the **`DFS()`** method 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();
}
```

Run the code above by **[Ctrl + F5]**. It should print the first connected component in the graph, holding the node 0:



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

Problem 5. Find All Connected Components

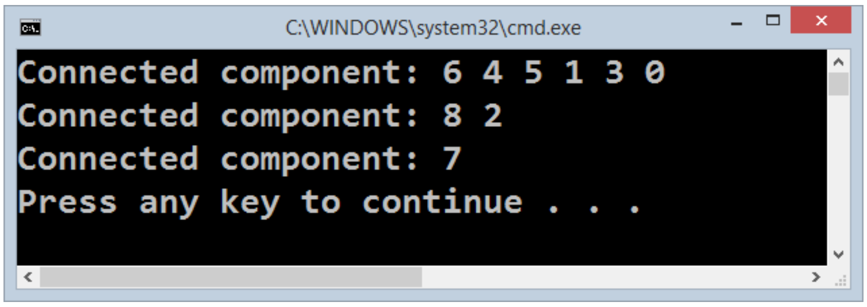
Now, we have DFS algorithm implemented, which finds the connected component holding all nodes reachable from given starting node. This is good, but we want to **find all connected components**. We can just run the DFS algorithm many times from each node taken as a start (which was not visited already):

```
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 6. 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). We already have the method below:

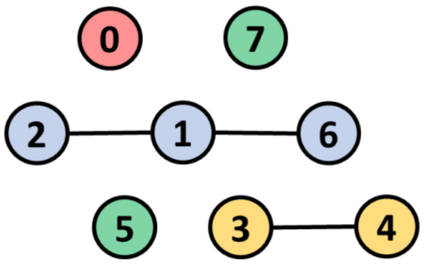
```
static List<int>[] ReadGraph()
{
    int n = int.Parse(Console.ReadLine());
    var graph = new List<int>[n];
    for (int i = 0; i < n; i++)
    {
        graph[i] = Console.ReadLine().Split(new char[] { ' ' },
          StringSplitOptions.RemoveEmptyEntries).Select(int.Parse).ToList();
    }
 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

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

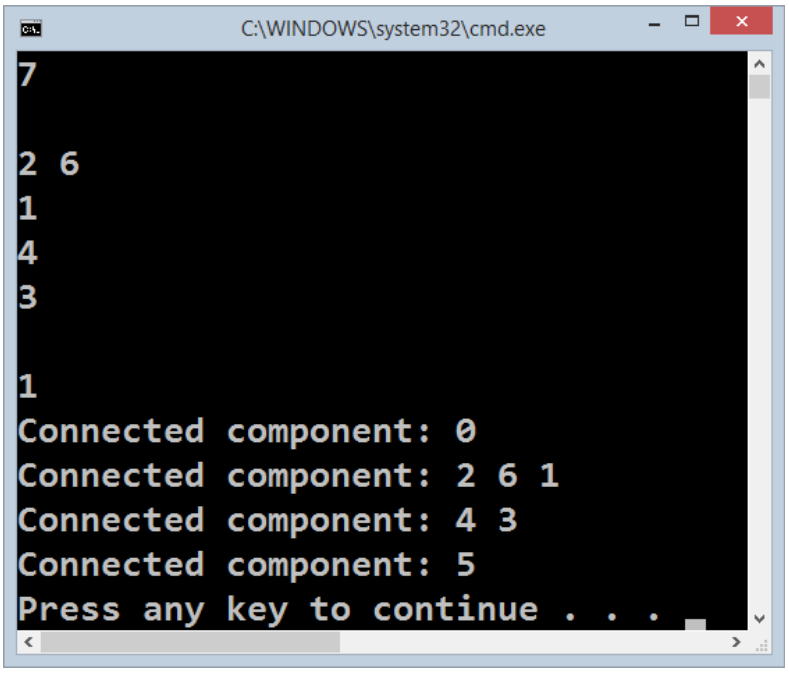
```
public static void Main()
{
    graph = ReadGraph();
    FindGraphConnectedComponents();
}
```

Now test the program. Run it by [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`
		`Connected component: 7`

Seems like it runs correctly:

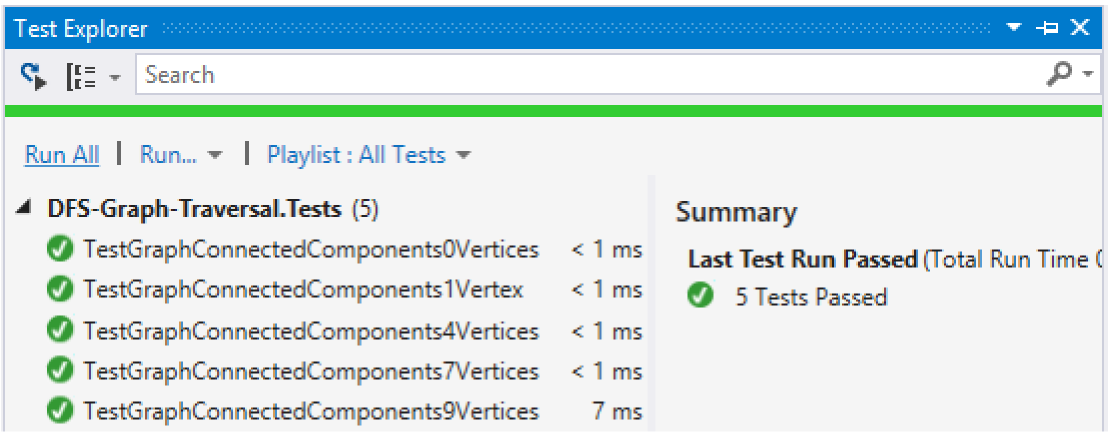


```
7


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 7. Run the Unit Tests

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


Test Explorer

Search

Run All | Run... | Playlist : All Tests

```
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.

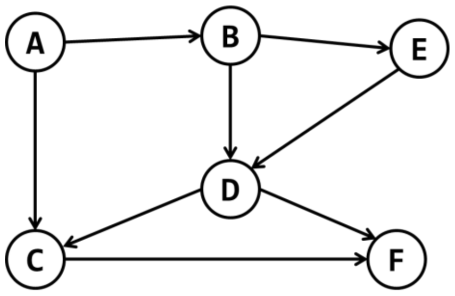
Part II – Topological Sorting

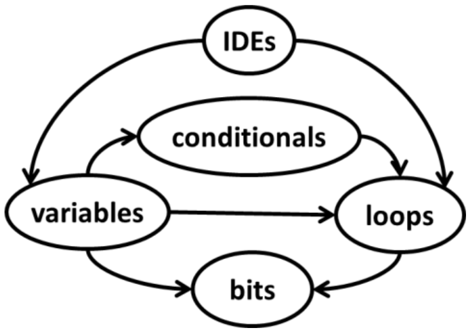
We're given a **directed graph** which means that if node A is connected to node B and the vertex is directed from A to B, we can move from A to B, but not the other way around, i.e. we have a one-way street. We'll call A "source" or "predecessor" and B – "child".

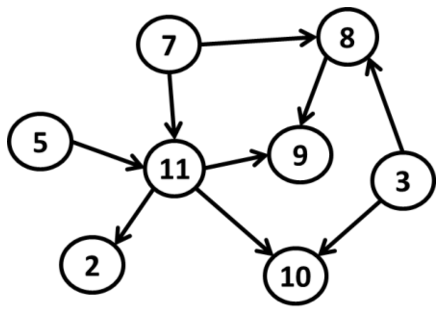
Let's consider the tasks a SoftUni student needs to perform during an exam – "Read description", "Receive input", "Print output", etc. Clearly, some of the tasks depend on other tasks – we cannot print the output of a problem before we get the input. If all such tasks are nodes in a graph, a directed vertex will represent dependency between two tasks, e.g. if A -> B (A is connected to B and the direction is from A to B), this means B cannot be performed before completing A first. Having all tasks as nodes and the relationships between them as vertices, we can order the tasks using topological sorting. After the sorting procedure we'll obtain a list showing all tasks in the order in which they should be performed. Of course, there may be more than one such order, and there usually is, but in general, the tasks which are less dependent on other tasks will appear first in the resulting list.

For this problem, you need to implement topological sorting over a directed graph of strings. Some examples of what your program should do:

Input	**Picture**	**Output**

`"A" -> "B", "C"`
`"B" -> "D", "E"`
`"C" -> "F"`
`"D" -> "C", "F"`
`"E" -> "D"`
`"F" ->`
		`Topological sorting:`
		`A, B, E, D, C, F`

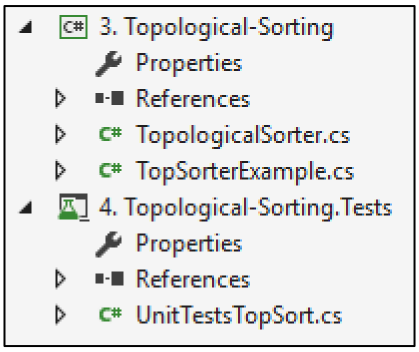
`"IDEs" -> "variables",`
`"loops"`
`"variables" ->`
`"conditionals", "loops",`
`"bits"`
`"conditionals" -> "loops"`
`"loops" -> "bits"`
`"bits"`
		`Topological sorting:`
		`IDEs, variables,`
		`conditionals, loops,`
		`bits`

`"5" -> "11"`
`"7" -> "11", "8"`
`"8" -> "9"`
`"11" -> "9", "10", "2"`
`"9" ->`
`"3" -> "8", "10"`
`"2" ->`
`"10"`
		`Topological sorting:`
		`3, 7, 8, 5, 11, 2,`
		`10, 9`

We'll solve this using two different algorithms – source removal and DFS.

Problem 1. Topological Sorting – Project Skeleton

You are given a project skeleton containing a **TopologicalSorter** class where all operations should be performed, a **TestSorterExample** class which you may use to test your program with different input values and a unit test project.

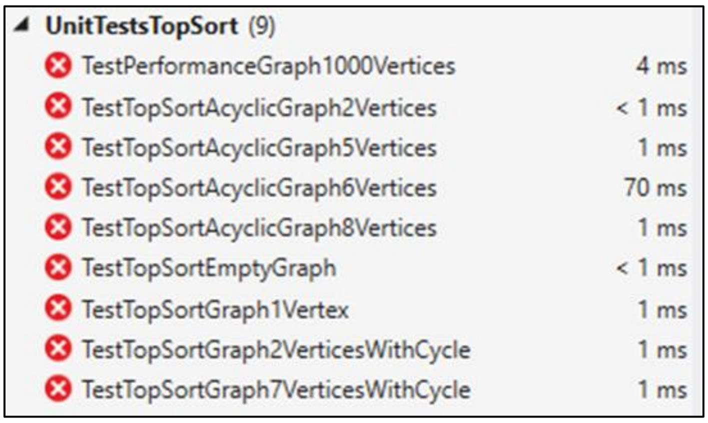


```
3. Topological-Sorting
    Properties
    References
    TopologicalSorter.cs
    TopSorterExample.cs
4. Topological-Sorting.Tests
    Properties
    References
    UnitTestsTopSort.cs
```

Complete the **TopologicalSorter** class and make sure the procedure works correctly by running the unit tests.

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

Before you start, run the unit tests. Since the sorting algorithm isn't implemented yet, they should all fail.



```
UnitTestsTopSort (9)
  TestPerformanceGraph1000Vertices         4 ms
  TestTopSortAcyclicGraph2Vertices       < 1 ms
  TestTopSortAcyclicGraph5Vertices         1 ms
  TestTopSortAcyclicGraph6Vertices        70 ms
  TestTopSortAcyclicGraph8Vertices         1 ms
  TestTopSortEmptyGraph                  < 1 ms
  TestTopSortGraph1Vertex                  1 ms
  TestTopSortGraph2VerticesWithCycle       1 ms
  TestTopSortGraph7VerticesWithCycle       1 ms
```

Problem 3. Source Removal Algorithm

The source removal algorithm is pretty simple – if finds the node which isn't dependent on any other node and removes it along with all vertices connected to it. A node that doesn't depend on other nodes can be thought of as a source, the starting point, hence the name of the algorithm – source removal.

Note that we're guaranteed to have an independent node if all prerequisites are met – we have a directed graph without any cycles. After removing the source, we obtain a new graph in which we are guaranteed to have at least one, perhaps more, independent nodes (sources). We continue removing each node recursively until we're done and all nodes have been removed.

Problem 4. Source Removal Algorithm: Calulate Predecessors

In order to efficiently remove a node at each step, we need to know the number of predecessors for each node. An easy way to do this is to calculate the number of predecessors beforehand and keep track of it when removing nodes.

An appropriate structure to use would be a dictionary where the key is the node's value and the value is the number of predecessors. Just loop through each node and its children and for each child node increment the number of predecessors:

```
// Find the predecessorsCount
var predecessorsCount = new Dictionary<string, int>();
foreach (var node in this.graph)
{
    if (! predecessorsCount.ContainsKey(node.Key))
    {
        predecessorsCount[node.Key] = 0;
    }
    foreach (var childNode in node.Value)
    {
        if (!predecessorsCount.ContainsKey(childNode))
        {
            predecessorsCount[childNode] = 0;
        }

        predecessorsCount[childNode]++;
    }
}
```

Problem 5. Source Removal Algorithm: Implementation

Now that we know how many predecessors each node has, we just need to find one without predecessors and remove it, then repeat until we're done.

We'll keep the nodes in a list and start a loop which we'll stop as soon as we're done:

```
var removedNodes = new List<string>();
while (true)
{
    // TODO
}
```

Finding a source can be simplified with LINQ. We just need to check if such a node exists; if not, we break the loop (we either have no nodes left or we encountered a cycle):

```
string nodeToRemove = graph.Keys.FirstOrDefault(n => predecessorsCount[n] == 0);

if (nodeToRemove == null)
{
    // No more nodes for removal (with 0 predecessors)
    break;
}
```

Removing a node involves several steps:

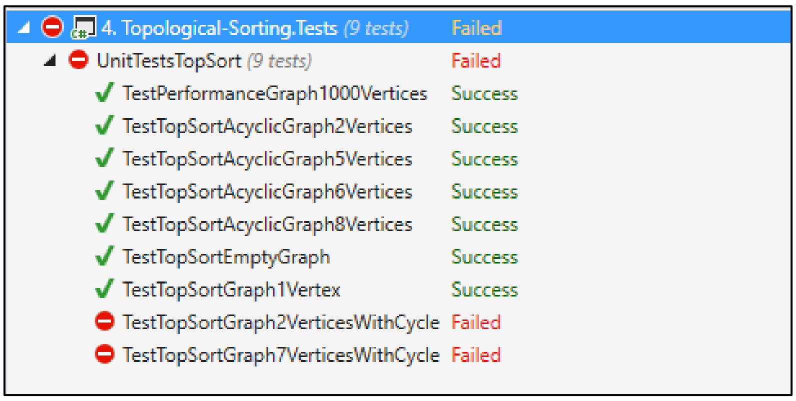
1) All its child nodes lose a predecessor -> decrement the count of predecessors for each of the children
2) Remove the node from the graph
3) Add the node to the list of removed nodes

```
// TODO: Remove vertices, i.e. decrement predecessor count

graph.Remove(nodeToRemove);
removedNodes.Add(nodeToRemove);
```

Finally, return the list of removed nodes.

Problem 6. Test the Source Removal Algorithm

Run the unit tests. It seems we have a problem:



```
4. Topological-Sorting.Tests (9 tests)      Failed
  UnitTestsTopSort (9 tests)                Failed
    TestPerformanceGraph1000Vertices        Success
    TestTopSortAcyclicGraph2Vertices        Success
    TestTopSortAcyclicGraph5Vertices        Success
    TestTopSortAcyclicGraph6Vertices        Success
    TestTopSortAcyclicGraph8Vertices        Success
    TestTopSortEmptyGraph                   Success
    TestTopSortGraph1Vertex                 Success
    TestTopSortGraph2VerticesWithCycle      Failed
    TestTopSortGraph7VerticesWithCycle      Failed
```

We can see that the last two unit tests fail. They test with invalid graphs, ones with cycles in them. We need to modify our algorithm to take care of cycles.

Problem 7. Source Removal Algorithm: Detect Cycles

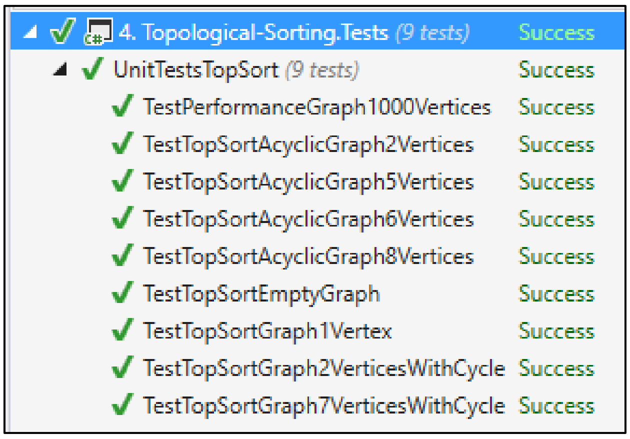
How do we know if there are cycles in the graph? Our removal algorithm looks for nodes without predecessors. If we ended the loop and the graph still has nodes, this means there is a cycle.

Just add a check after the while loop and throw the proper exception if the graph is not empty:

```
if (graph.Count > 0)
{
    throw new InvalidOperationException("A cycle detected in the graph");
}
```

Problem 8. Test the Source Removal Algorithm with Cycle Detection

Run the unit tests again. This time they should pass:



```
4. Topological-Sorting.Tests (9 tests)      Success
  UnitTestsTopSort (9 tests)                 Success
    TestPerformanceGraph1000Vertices         Success
    TestTopSortAcyclicGraph2Vertices         Success
    TestTopSortAcyclicGraph5Vertices         Success
    TestTopSortAcyclicGraph6Vertices         Success
    TestTopSortAcyclicGraph8Vertices         Success
    TestTopSortEmptyGraph                    Success
    TestTopSortGraph1Vertex                  Success
    TestTopSortGraph2VerticesWithCycle       Success
    TestTopSortGraph7VerticesWithCycle       Success
```

Problem 9. Implement the Topological Sorting with DFS Algorithm

The second algorithm we'll use is DFS. You can comment out the method you just implemented and rewrite it in order to use the same unit tests.

For this one, we'll need two collections – one to keep track of all visited nodes and one to hold the sorted nodes:

```
private Dictionary<string, List<string>> graph;
private HashSet<string> visitedNodes;
private LinkedList<string> sortedNodes;
```

Since we keep the collections as fields, we can change the **TopSort()** method's return type to **void**.

The DFS topological sort is simple – loop through each node:

```
public void TopSort()
{
    this.visitedNodes = new HashSet<string>();
    this.sortedNodes = new LinkedList<string>();

    foreach (var node in this.graph.Keys)
    {
        TopSortDFS(node);
    }
}
```

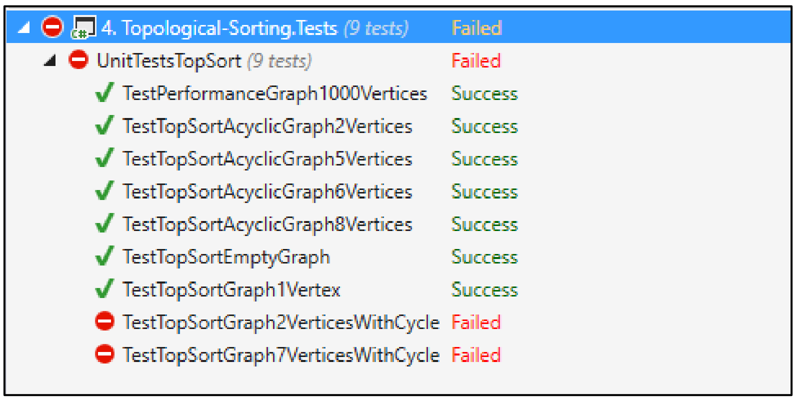
The **TopSortDFS()** method shouldn't do anything if the node is already visited; otherwise, it should mark the node as visited and add it to the list of sorted nodes. It should also do this for its children (if there are any):

```
if (!this.visitedNodes.Contains(node))
{
    this.visitedNodes.Add(node);

    // TODO: check if node has children (graph contains node as key)
    // TODO: if node has children - sort them by calling TopSortDFS

    this.sortedNodes.AddFirst(node);
}
```

Note that we add the node to the sorted list **after** we traverse its children. This guarantees that the node will be before its children in the list – this is exactly what we're trying to achieve.

Problem 10. Test the Topological Sorting with DFS Algorithm

Run the unit tests. Once again, we have problems with cycles:



```
4. Topological-Sorting.Tests (9 tests)      Failed
  UnitTestsTopSort (9 tests)                Failed
    TestPerformanceGraph1000Vertices        Success
    TestTopSortAcyclicGraph2Vertices        Success
    TestTopSortAcyclicGraph5Vertices        Success
    TestTopSortAcyclicGraph6Vertices        Success
    TestTopSortAcyclicGraph8Vertices        Success
    TestTopSortEmptyGraph                   Success
    TestTopSortGraph1Vertex                 Success
    TestTopSortGraph2VerticesWithCycle      Failed
    TestTopSortGraph7VerticesWithCycle      Failed
```

Problem 11. Add Cycle Detection

How do we know if a node forms a cycle? We can add it to a list of cycle nodes before traversing its children. If we enter a node with the same value, it will be in the **`cycleNodes`** list, so we throw an exception. If there are no descendants with the same value then there are no cycles, so once we finish traversing the children we remove the current node from **`cycleNodes`**.

Obviously, we'll need a new collection to hold the cycle nodes, e.g. a HashSet<string>. Exiting the method with an exception is easy, just check if the current node is in the list of cycle nodes at the very beginning of the **`TopSortDFS()`** method:

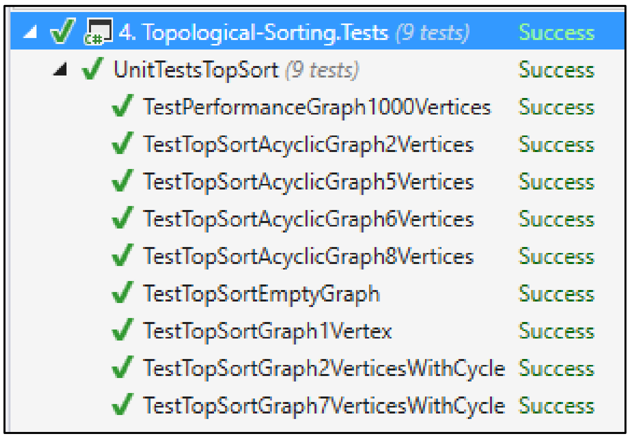
```
if (this.cycleNodes.Contains(node))
{
    throw new InvalidOperationException("A cycle detected in the graph");
}
```

Then, keep track of the cycle nodes:

```
if (!this.visitedNodes.Contains(node))
{
    this.visitedNodes.Add(node);
    this.cycleNodes.Add(node);

    // TODO: check if node has children (graph contains node as key)
    // TODO: if node has children - sort them by calling TopSortDFS

    this.cycleNodes.Remove(node);
    this.sortedNodes.AddFirst(node);
}
```

Problem 12. Test the Topological Sorting with Cycle Detection

Re-run the unit tests. This time they should all pass:



```
◢ ✓ 4. Topological-Sorting.Tests (9 tests)      Success
  ◢ ✓ UnitTestsTopSort (9 tests)                 Success
      ✓ TestPerformanceGraph1000Vertices         Success
      ✓ TestTopSortAcyclicGraph2Vertices         Success
      ✓ TestTopSortAcyclicGraph5Vertices         Success
      ✓ TestTopSortAcyclicGraph6Vertices         Success
      ✓ TestTopSortAcyclicGraph8Vertices         Success
      ✓ TestTopSortEmptyGraph                    Success
      ✓ TestTopSortGraph1Vertex                  Success
      ✓ TestTopSortGraph2VerticesWithCycle       Success
      ✓ TestTopSortGraph7VerticesWithCycle       Success
```

You have implemented topological sorting using two different algorithms!