

# Lab: Graph Theory, Traversal, and Topological Sorting

Please submit your solutions (source code) of all below-described problems in [Judge](#).

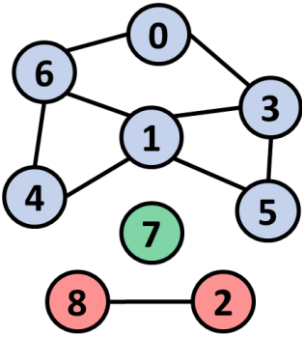
## 1. Connected Components

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 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.

### Example

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
0	(empty graph)	

### Hints

#### Read Input

Let's implement the data entry logic (read the graph from the console):

```
nodes = int(input())  
graph = []  
for _ in range(nodes):  
    graph.append([int(x) for x in input().split()])
```

#### DFS Algorithm

First, create a bool array that will be with the size of your graph:

```
visited = [False] * nodes
```

Next, implement the **DFS algorithm** (Depth-First-Search) to traverse all nodes connected to the specified start node:

```
def dfs(node, graph, visited, component):
    if visited[node]:
        return
    visited[node] = True
    for child in graph[node]:
        dfs(child, graph, visited, component)
    component.append(node)
```

## Find All Components

We want to **find all connected components**. We can just run the DFS algorithm for each node taken as a start (which was not visited already):

```
for node in range(nodes):
    if visited[node]:
        continue
    component = []
    dfs(node, graph, visited, component)
    print(f"Connected component: {' '.join([str(x) for x in component])}")
```

## Test

Now let's test the above code. The output is as expected. It prints all connected components in the graph:

```
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

Process finished with exit code 0
```

## 2. Source Removal 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.

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  $A \rightarrow B$  (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.

## Input

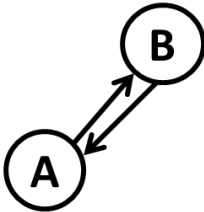
- On the first line, you will receive an integer **n** – nodes count.
- On the next **n** lines, you will receive nodes in the following format: "**{key} -> {children1}, {children2},... {childrenN}**".
  - It is possible some of the keys to not having any children.

## Output

- If the sorting is possible then print "**Topological sorting: {sortedKey1}, {sortedKey2}, ...{sortedKeyN}**".
- Otherwise, print "**Invalid topological sorting**".

## Example

Input	Picture	Output
6 A -> B, C B -> D, E C -> F D -> C, F E -> D F ->	<pre> graph TD     A((A)) --&gt; B((B))     A((A)) --&gt; C((C))     B((B)) --&gt; D((D))     B((B)) --&gt; E((E))     C((C)) --&gt; F((F))     D((D)) --&gt; C((C))     D((D)) --&gt; F((F))     E((E)) --&gt; D((D))     F((F)) --&gt; D((D))         </pre>	Topological sorting: A, B, E, D, C, F
5 IDEs -> variables, loops variables -> conditionals, loops, bits conditionals -> loops loops -> bits bits ->	<pre> graph TD     IDEs((IDEs)) --&gt; variables((variables))     IDEs((IDEs)) --&gt; loops((loops))     variables((variables)) --&gt; conditionals((conditionals))     variables((variables)) --&gt; loops((loops))     conditionals((conditionals)) --&gt; loops((loops))     loops((loops)) --&gt; bits((bits))     bits((bits)) --&gt; loops((loops))         </pre>	Topological sorting: IDEs, variables, conditionals, loops, bits

2 A -> B B -> A		Invalid topological sorting
-----------------------	---	-----------------------------

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

## Source Removal Algorithm

The source removal algorithm is pretty simple – it **finds the node which isn't dependent on any other node** and **removes it** along with all vertices connected to it.

We **continue removing** each node recursively **until we're done** and all nodes have been removed. If there are nodes in the graph after the algorithm is complete, there are circular dependencies (we will throw an exception).

## Compute Predecessors

To **efficiently** remove a node at each step, we need to **know the number of predecessors for each node**. To do this, we will calculate the number of predecessors beforehand.

Compute the predecessor count for each node:

```
def get_predecessors_count(graph):
    result = {}
    for node, children in graph.items():
        if node not in predecessors_count:
            result[node] = 0
        for child in children:
            if child not in predecessors_count:
                result[child] = 1
            else:
                result[child] += 1
    return result
```

## Remove Independent Nodes

Now that we know how many predecessors each node has, we just need to:

1. Find a node without predecessors and remove it
2. Repeat until we're done

We'll keep the result in a list and start a loop that will stop when there is no independent node:

```
def top_sort(graph, predecessors_count):
    result = []
    while predecessors_count:
        pass
    return result
```

Finding a source can be done by a custom function.

```
def find_node_without_predecessors(predecessors_count):
    for node, count in predecessors_count.items():
        if count == 0:
            return node
    return None
```

We just need to check if such a node exists; if not, we break the loop:

```
node = find_node_without_predecessors(predecessors_count)
if node is None:
    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 predecessor dictionary
3. Add the node to the list of removed nodes

```
for child in graph[node]:
    predecessors_count[child] -= 1

result.append(node)
predecessors_count.pop(node)
```

Finally, print the sorted nodes.

## Detect Cycles

If we ended the loop and the **predecessor\_count** still has nodes, this means there is a cycle. Just add a check after the while loop and print the proper message if the **predecessor\_count** is not empty:

```
sorted_nodes = top_sort(graph, predecessors_count)

if predecessors_count:
    print('Invalid topological sorting')
else:
    print(f"Topological sorting: {' '.join(sorted_nodes)}")
```

# DFS Topological Sorting

## DFS Algorithm

The second algorithm we'll use is **DFS**. For this one, we'll need the following collections:

```
graph = {}  
visited = set()  
cycles = set()
```

The DFS topological sort is simple – loop through each node. We create a deque for all sorted nodes because the DFS will find them in reverse order (we will add nodes in the beginning):

```
sorted_nodes = deque()  
for node in graph:  
    top_sort_dfs(node, sorted_nodes, graph, visited, cycles)
```

The **DFS** 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):

```
def top_sort_dfs(node, sorted_nodes, graph, visited, cycles):  
    if node in visited:  
        return  
    visited.add(node)  
    for child in graph[node]:  
        top_sort_dfs(child, sorted_nodes, graph, visited, cycles)  
    sorted_nodes.appendleft(node)
```

Note that we add the node to the result **after** we traverse its children. This guarantees that the node will be added after the nodes that depend on it.

## 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 **cycles** collection, 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 **cycles**.

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 **DFS** method then, keep track of the cycle nodes:

```
def top_sort_dfs(node, sorted_nodes, graph, visited, cycles):  
    if node in cycles:  
        raise Exception  
    if node in visited:  
        return  
  
    visited.add(node)  
    cycles.add(node)  
  
    for child in graph[node]:  
        top_sort_dfs(child, sorted_nodes, graph, visited, cycles)  
  
    sorted_nodes.appendleft(node)  
    cycles.remove(node)
```