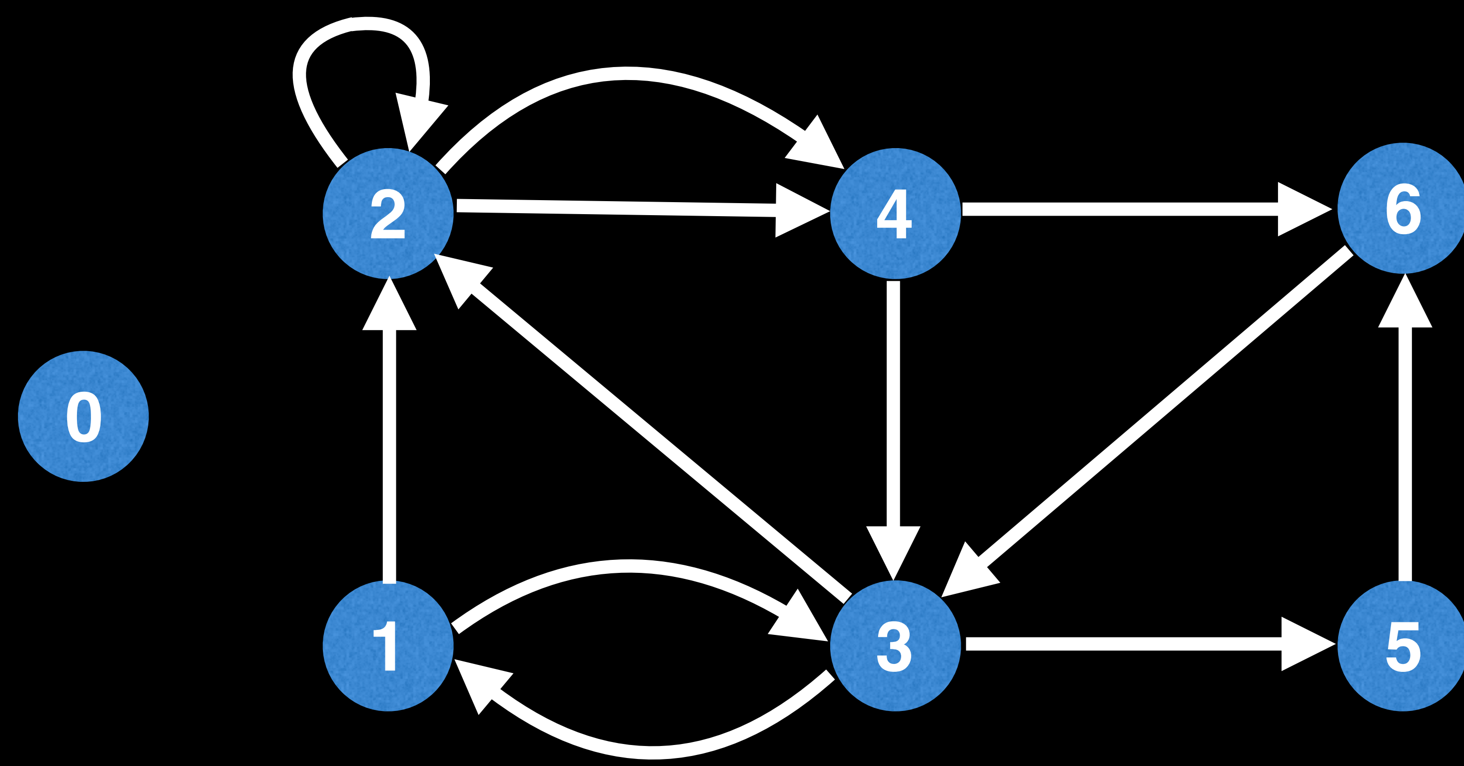
Finding Eulerian Paths and Circuits

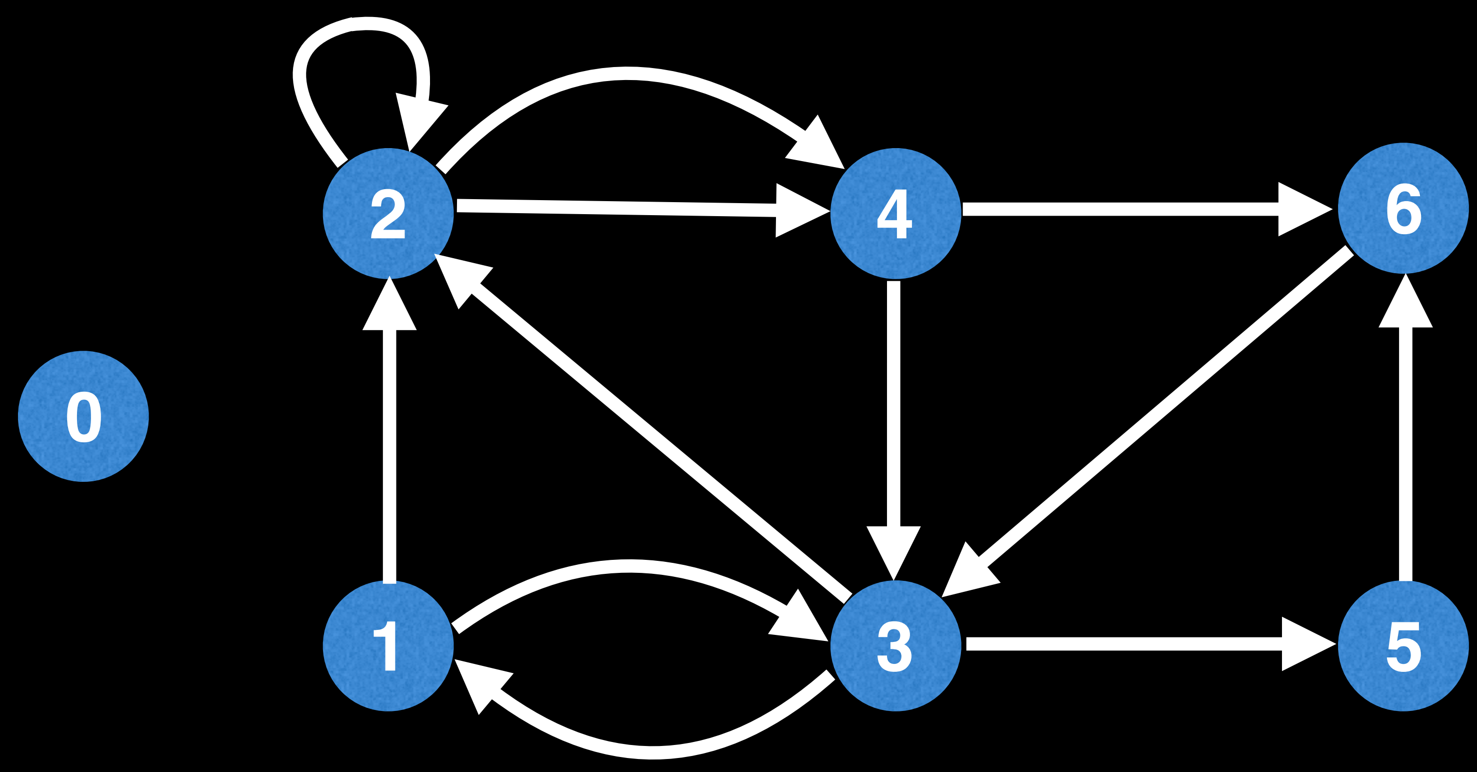
William Fiset

Previous video:

Finding an Eulerian path (directed graph)

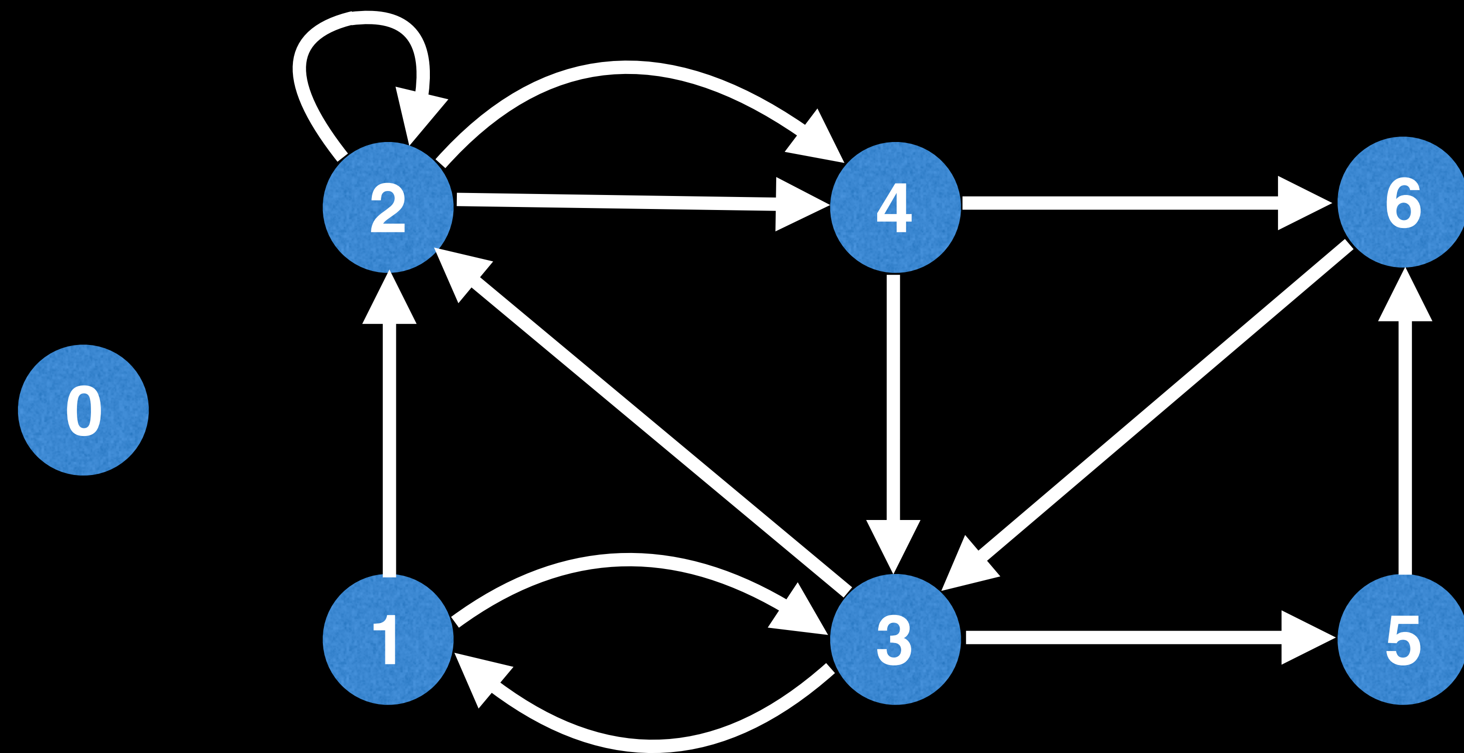


Finding an Eulerian path (directed graph)



Step 1 to finding an Eulerian path is determining if there even exists an Eulerian path.

Finding an Eulerian path (directed graph)

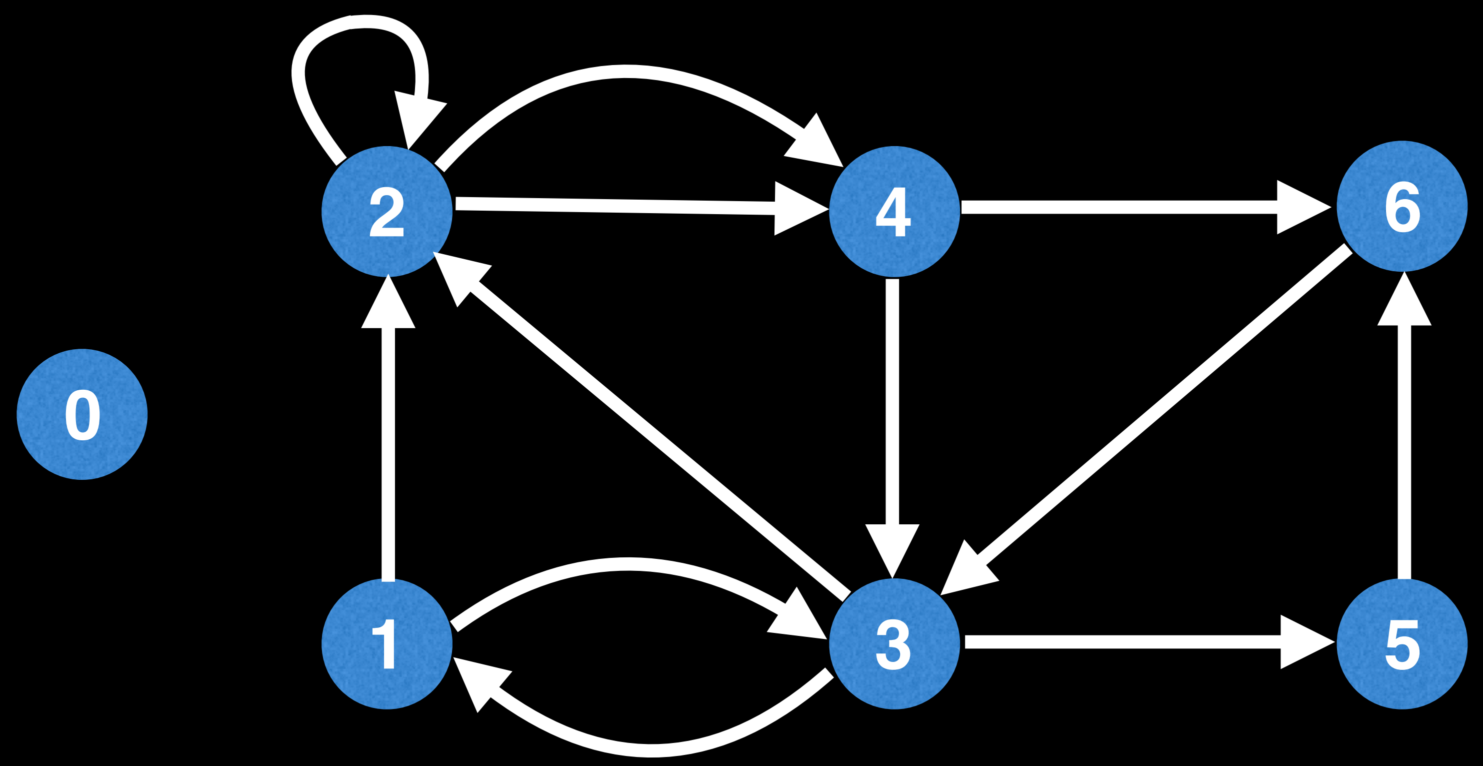


Step 1 to finding an Eulerian path is determining if there even exists an Eulerian path.

Recall that for an Eulerian path to exist at most one one vertex has (outdegree) - (indegree) = 1 and at most one vertex has (indegree) - (outdegree) = 1 and all other vertices have equal in and out degrees.

Finding an Eulerian path (directed graph)

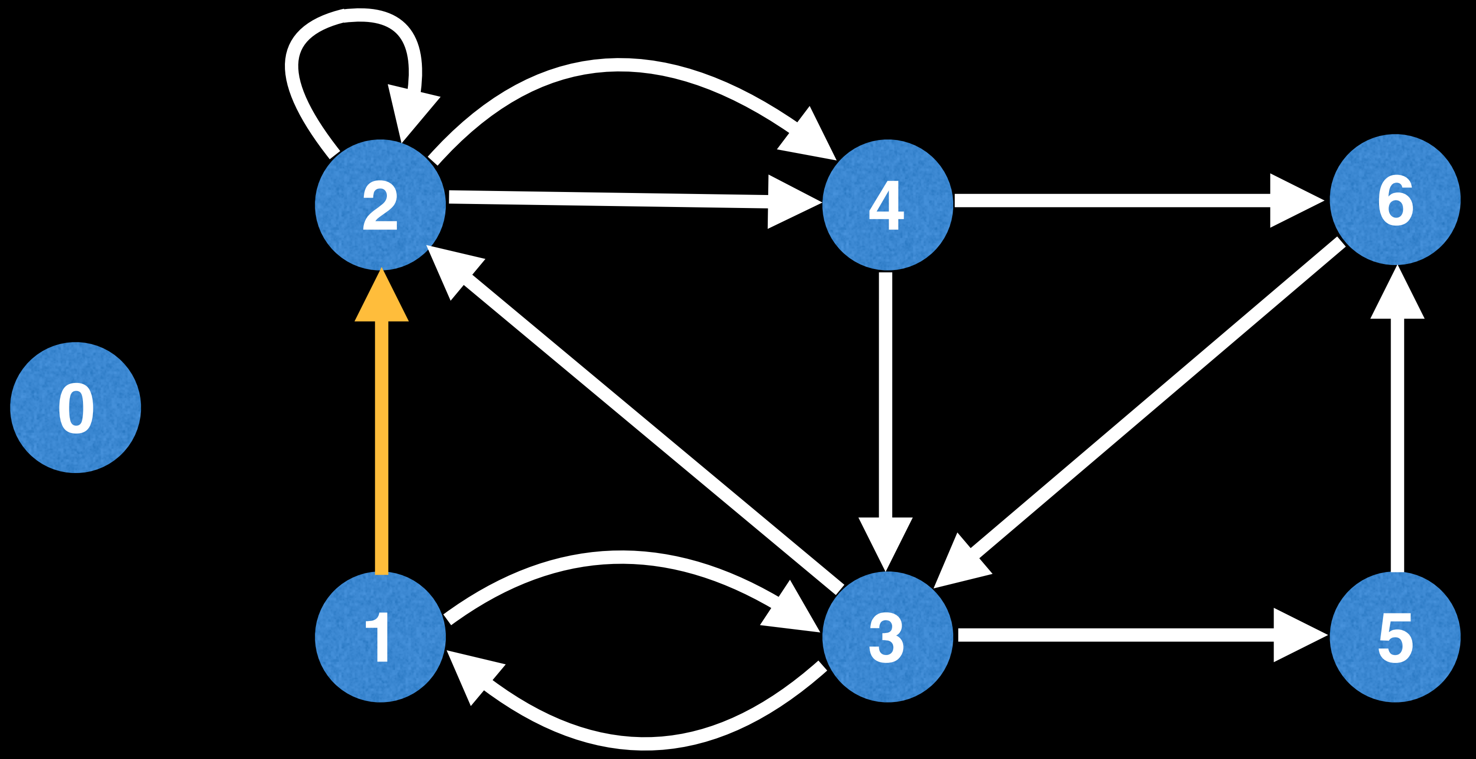
Node	In	Out
0		
1		
2		
3		
4		
5		
6		



Count the in/out degrees of each node by looping through all the edges.

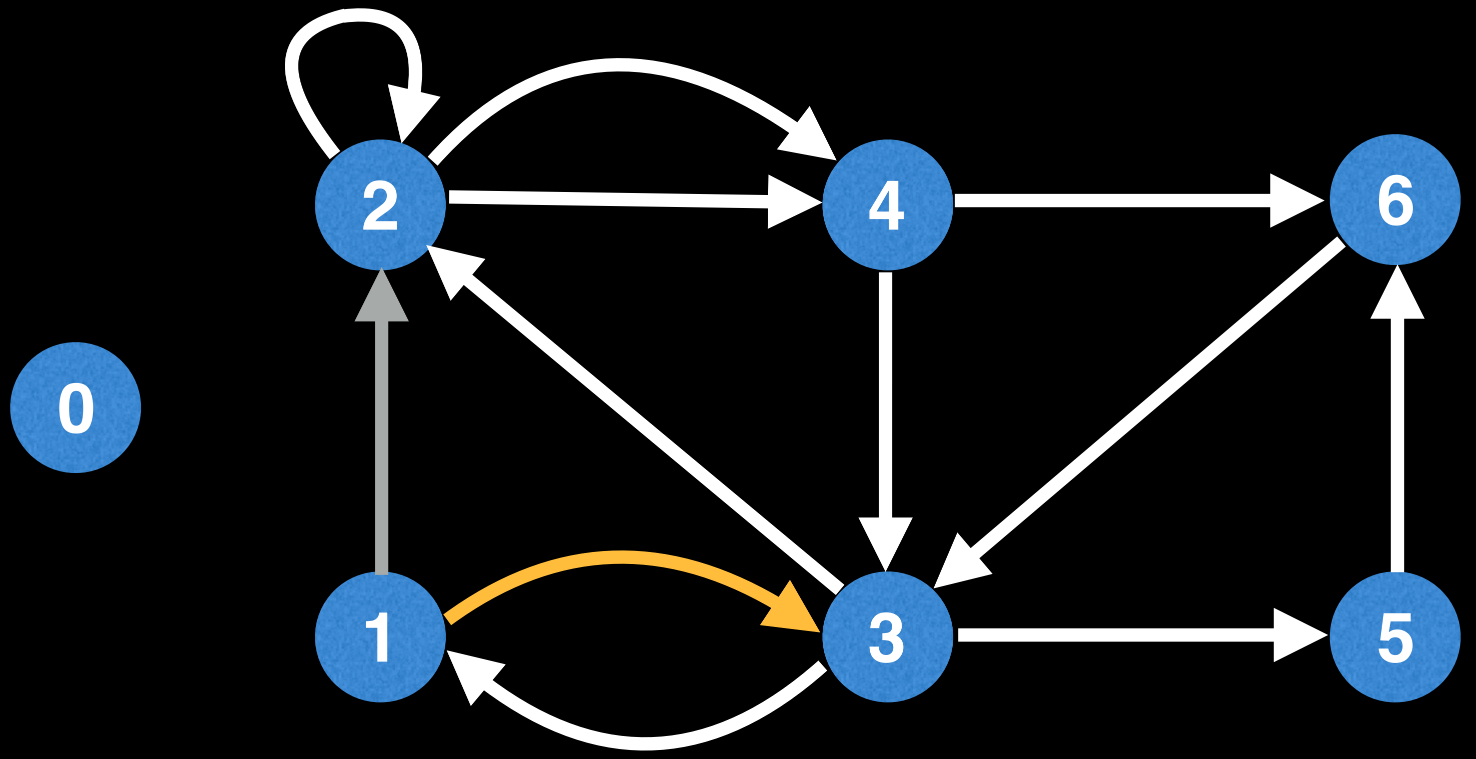
Finding an Eulerian path (directed graph)

Node	In	Out
0		
1		1
2	1	
3		
4		
5		
6		



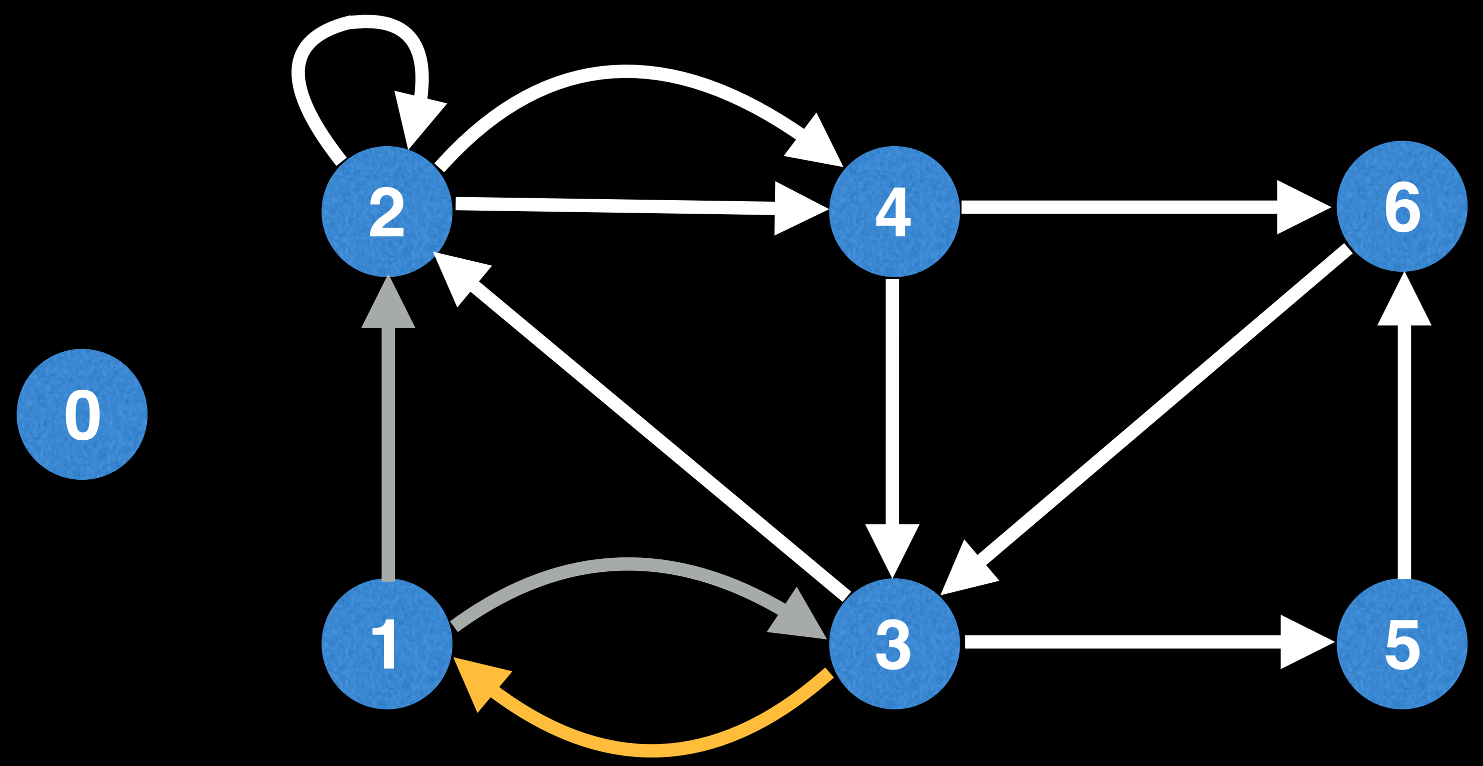
Finding an Eulerian path (directed graph)

Node	In	Out
0		
1		2
2	1	
3	1	
4		
5		
6		



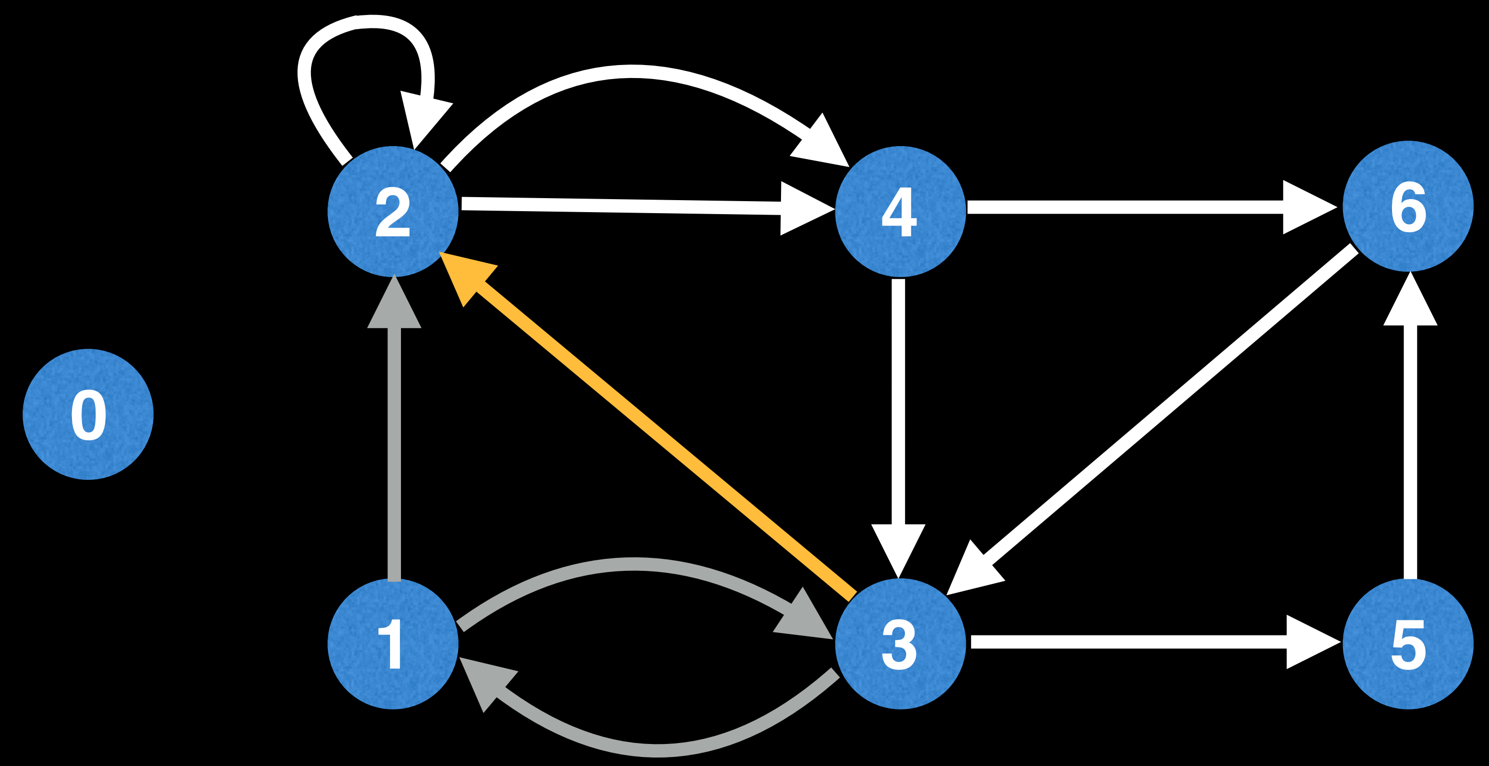
Finding an Eulerian path (directed graph)

Node	In	Out
0		
1	1	2
2	1	
3	1	1
4		
5		
6		



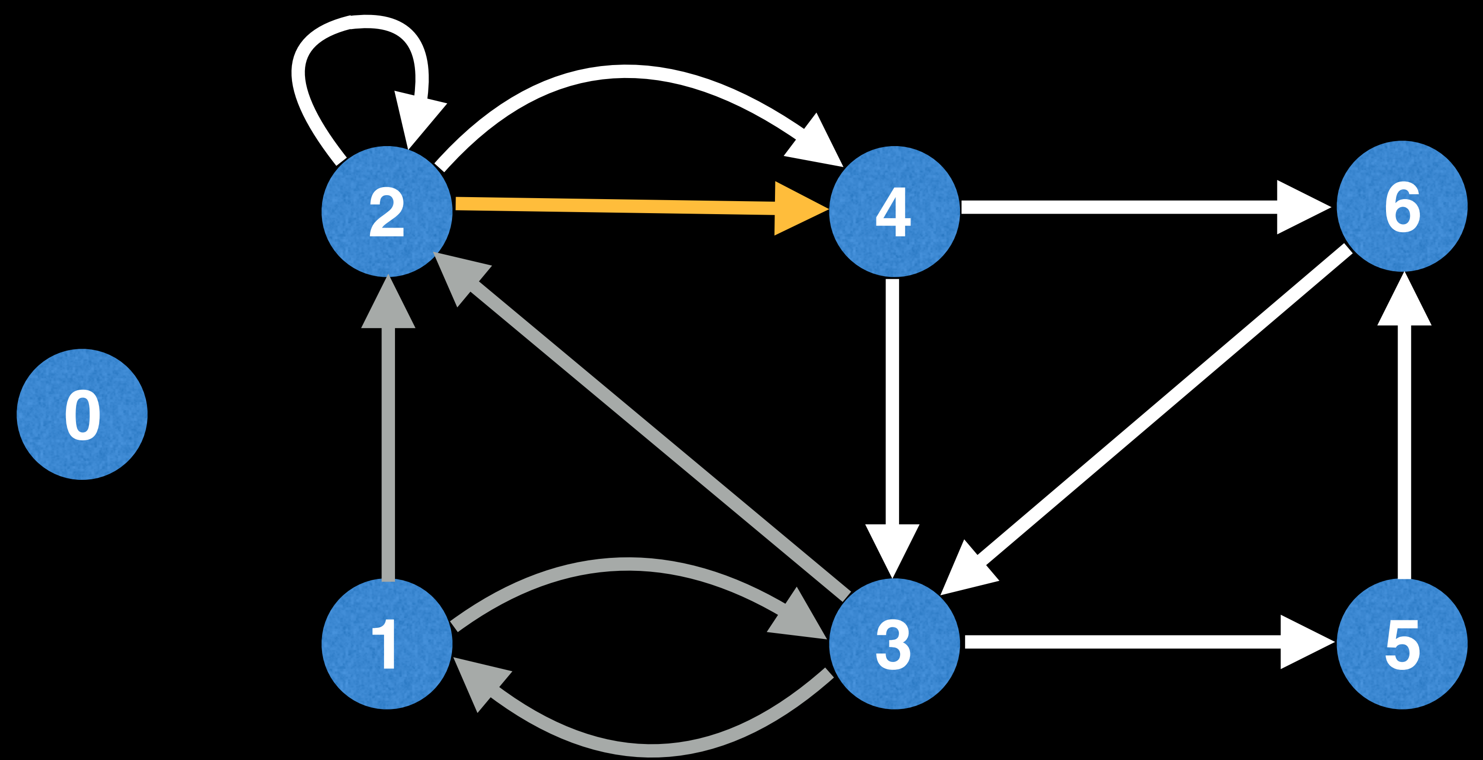
Finding an Eulerian path (directed graph)

Node	In	Out
0		
1	1	2
2	2	
3	1	2
4		
5		
6		



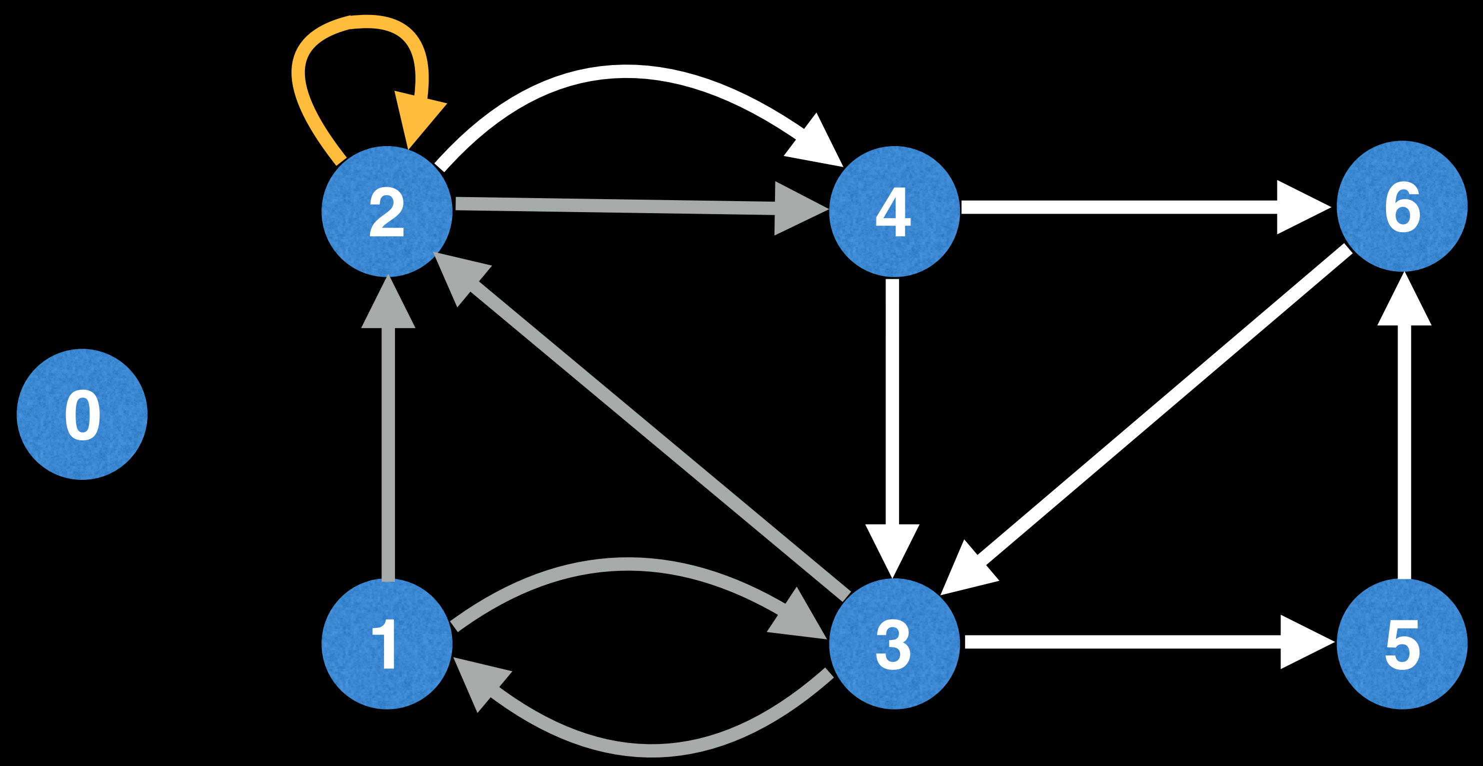
Finding an Eulerian path (directed graph)

Node	In	Out
0		
1	1	2
2	2	1
3	1	2
4	1	
5		
6		



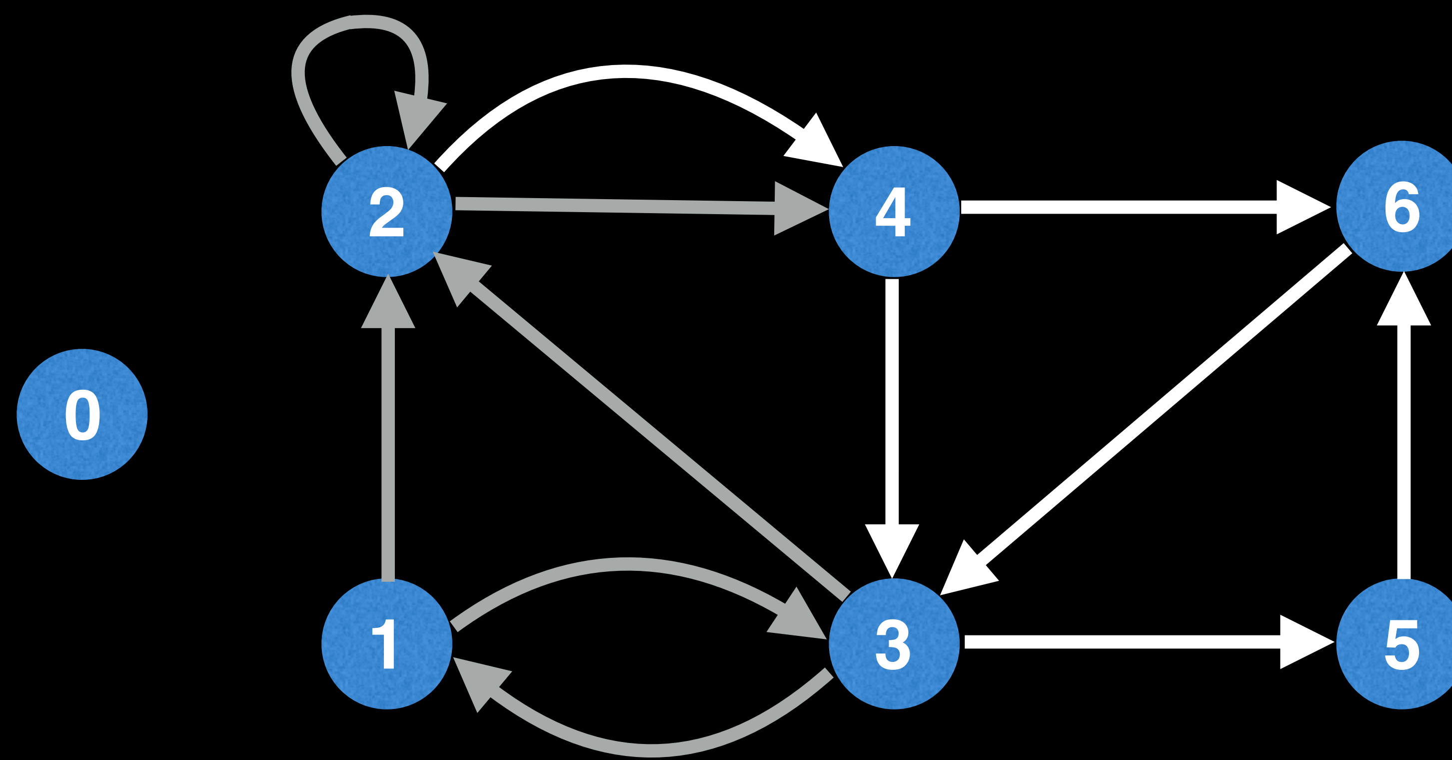
Finding an Eulerian path (directed graph)

Node	In	Out
0		
1	1	2
2	3	2
3	1	2
4	1	
5		
6		



Finding an Eulerian path (directed graph)

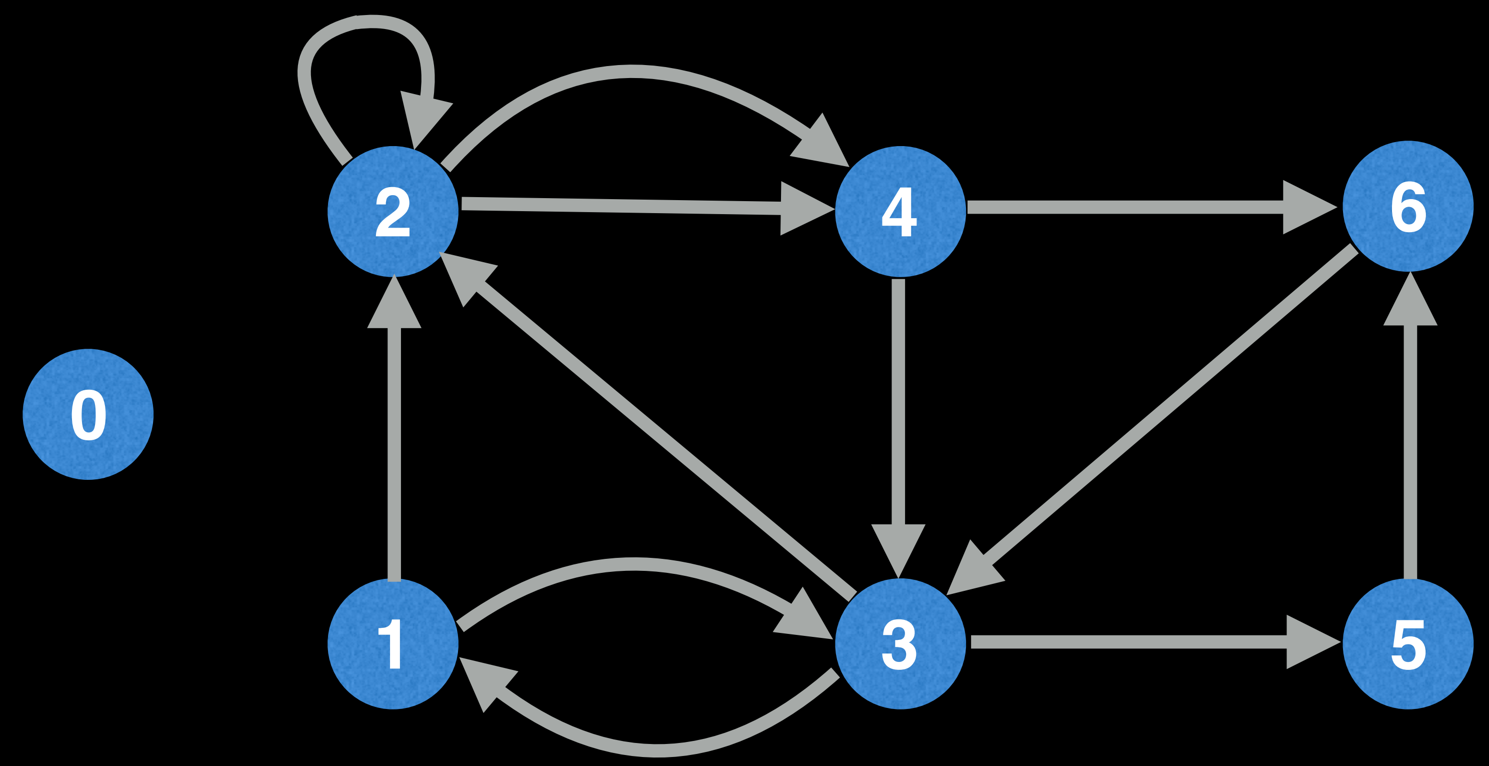
Node	In	Out
0		
1	1	2
2	3	2
3	1	2
4	1	
5		
6		



And so on for all other edges…

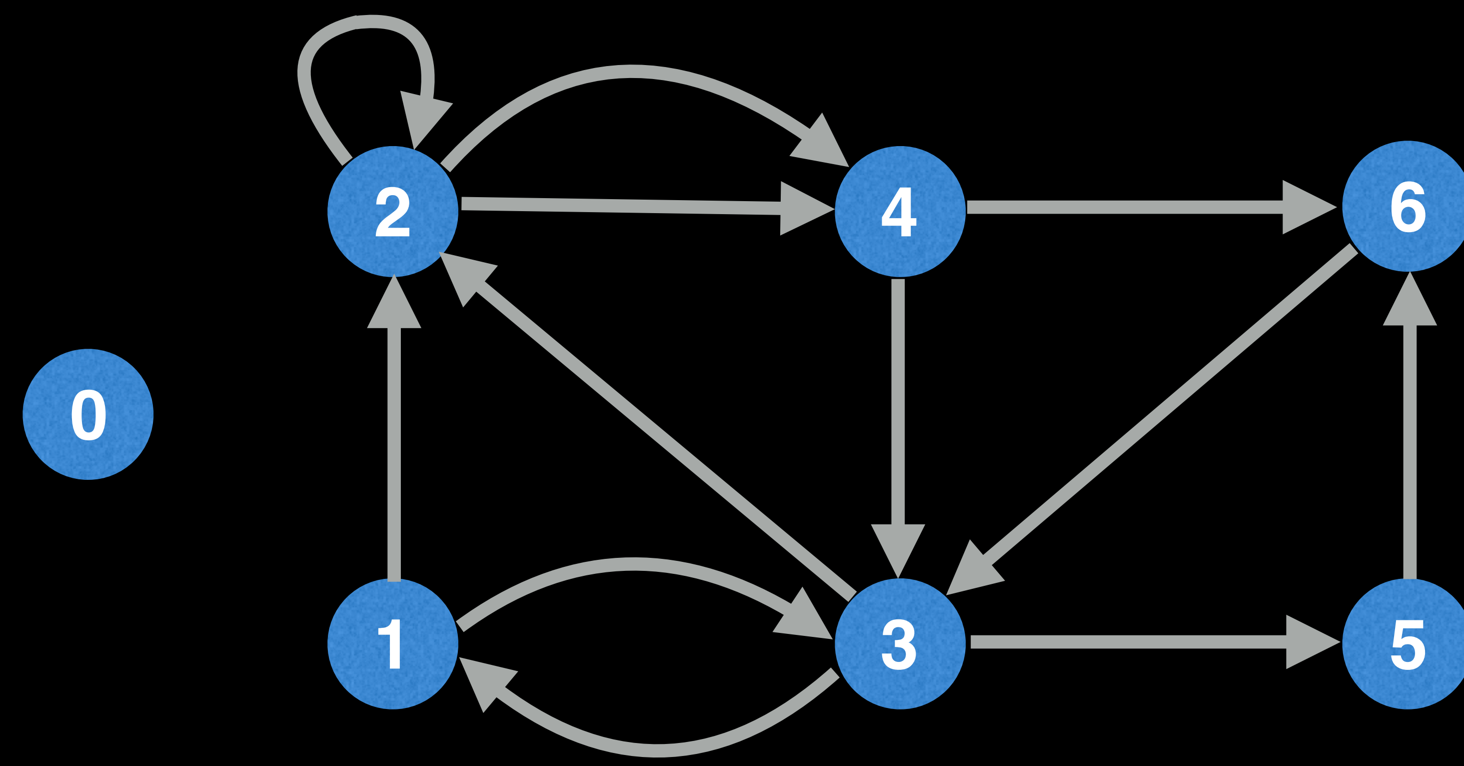
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

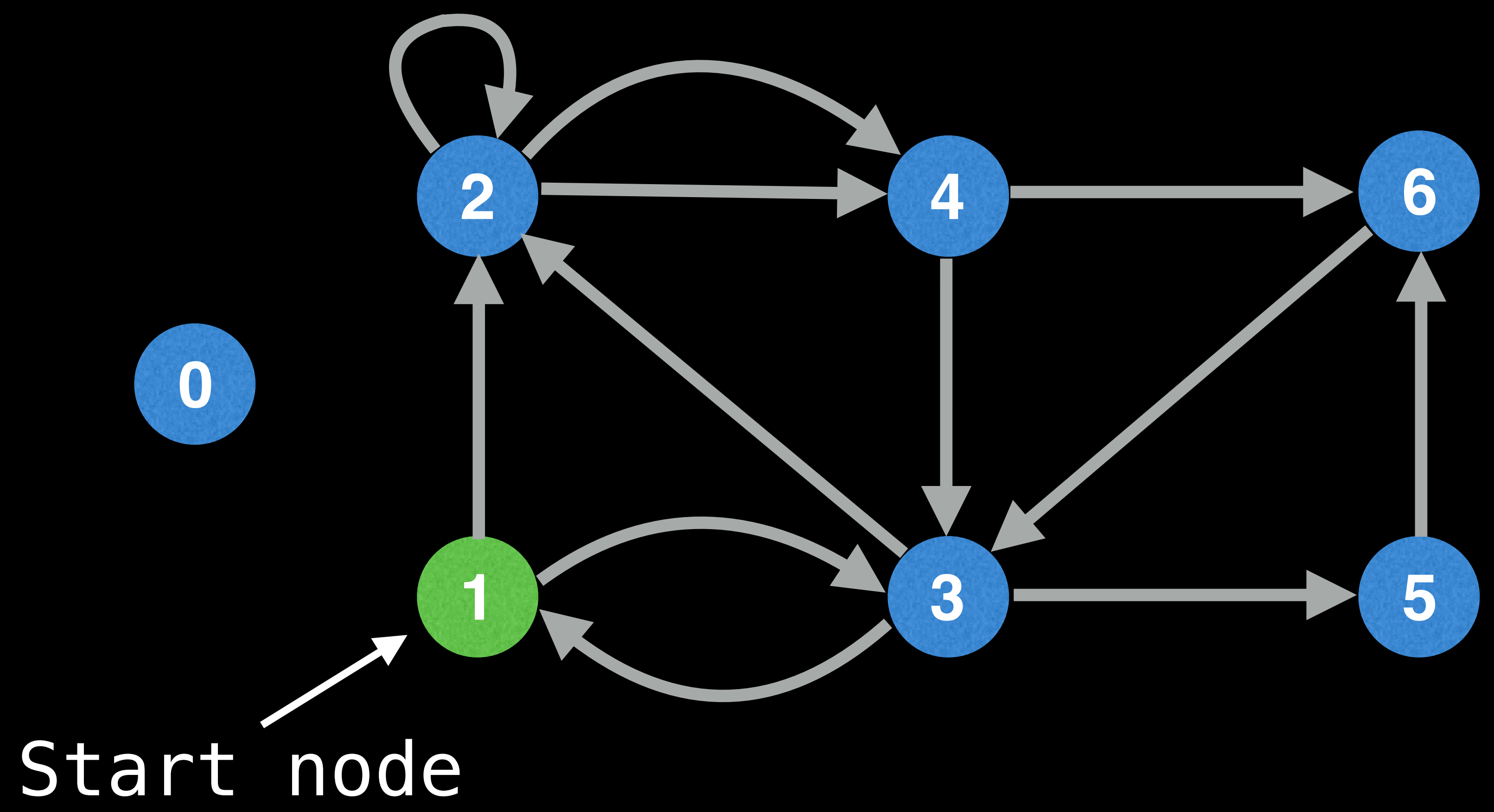


Once we've verified that no node has too many outgoing edges (out[i] – in[i] > 1) or incoming edges (in[i] – out[i] > 1) and there are just the right amount of start/end nodes we can be certain that an Eulerian path exists.

The next step is to find a valid starting node.

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

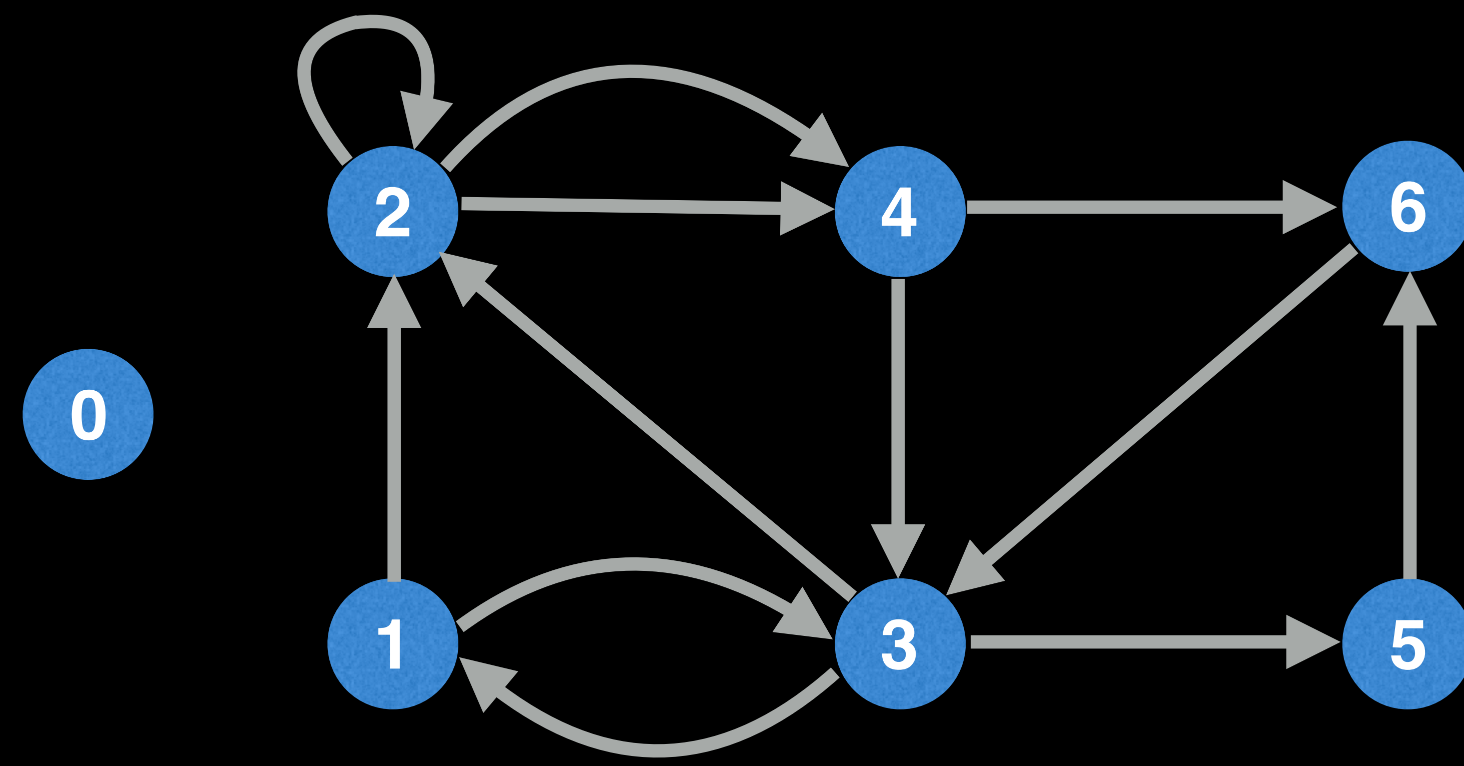


Node 1 is the only node with exactly one extra outgoing edge, so it's our only valid start node. Similarly, node 6 is the only node with exactly one extra incoming edge, so it will end up being the end node.

$$\text{out}[1] - \text{in}[1] = 2 - 1 = 1$$
$$\text{in}[6] - \text{out}[6] = 2 - 1 = 1$$

Finding an Eulerian path (directed graph)

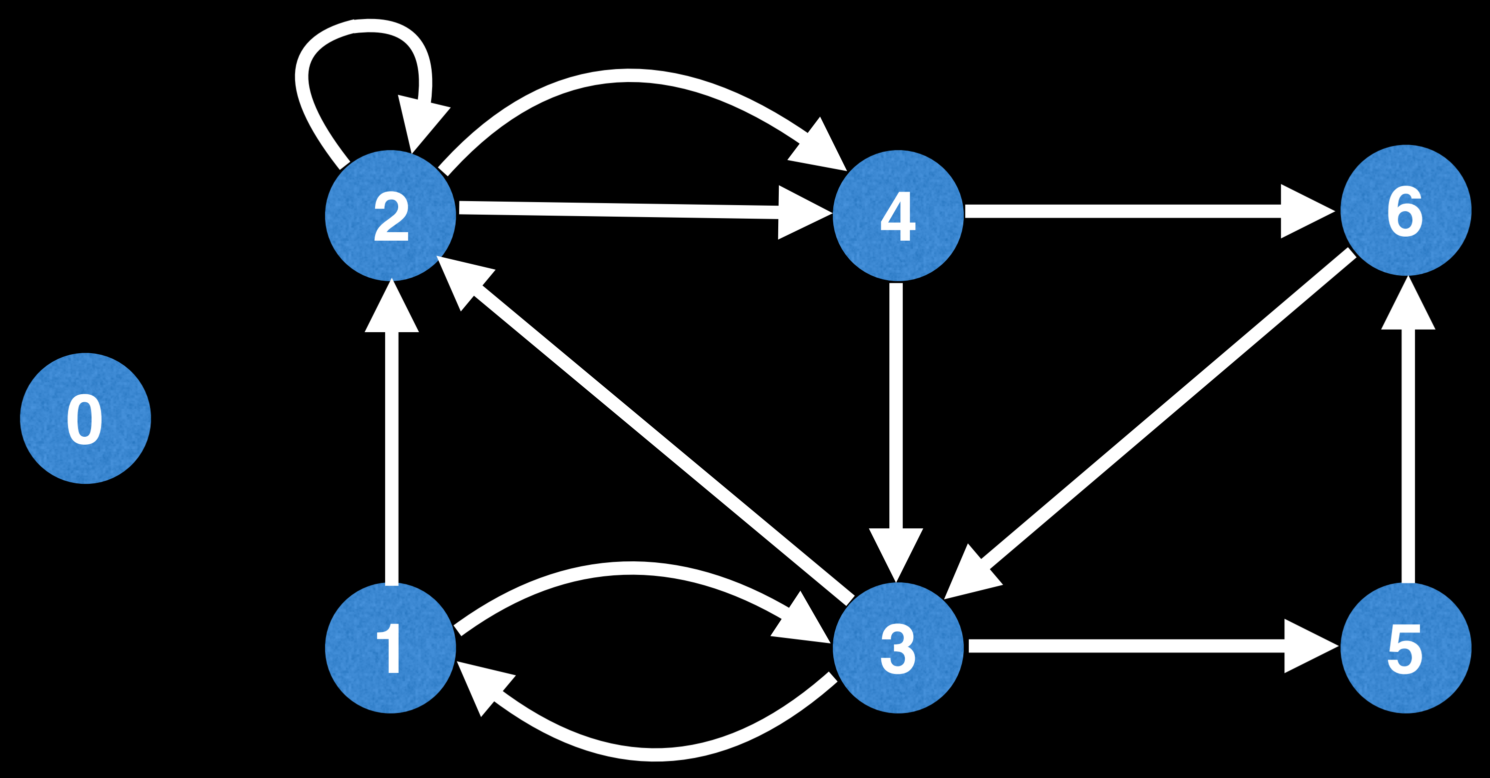
Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



NOTE: If all in and out degrees are equal (Eulerian circuit case) then any node with non-zero degree would serve as a suitable starting node.

Finding an Eulerian path (directed graph)

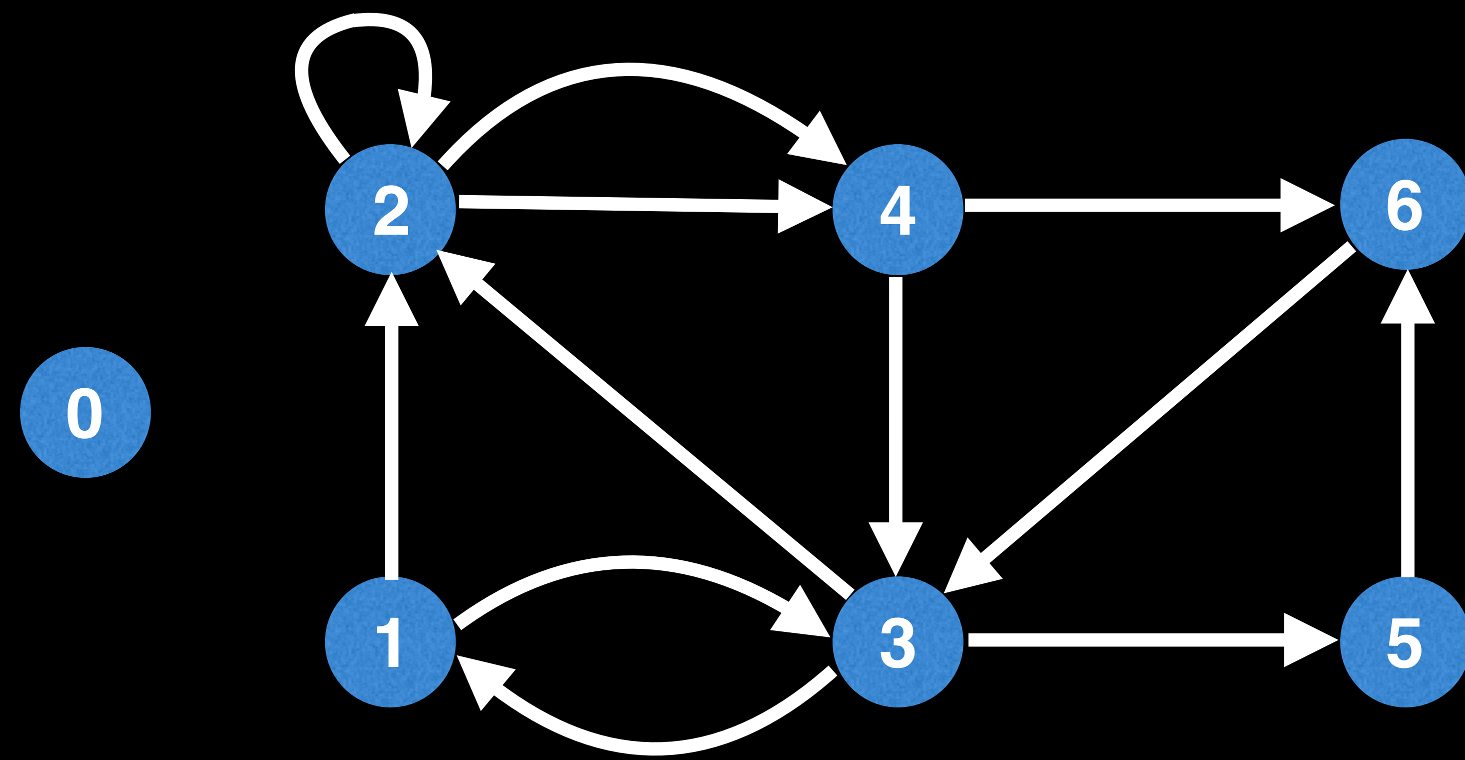
Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



Now that we know the starting node, let's find an Eulerian path!

Finding an Eulerian path (directed graph)

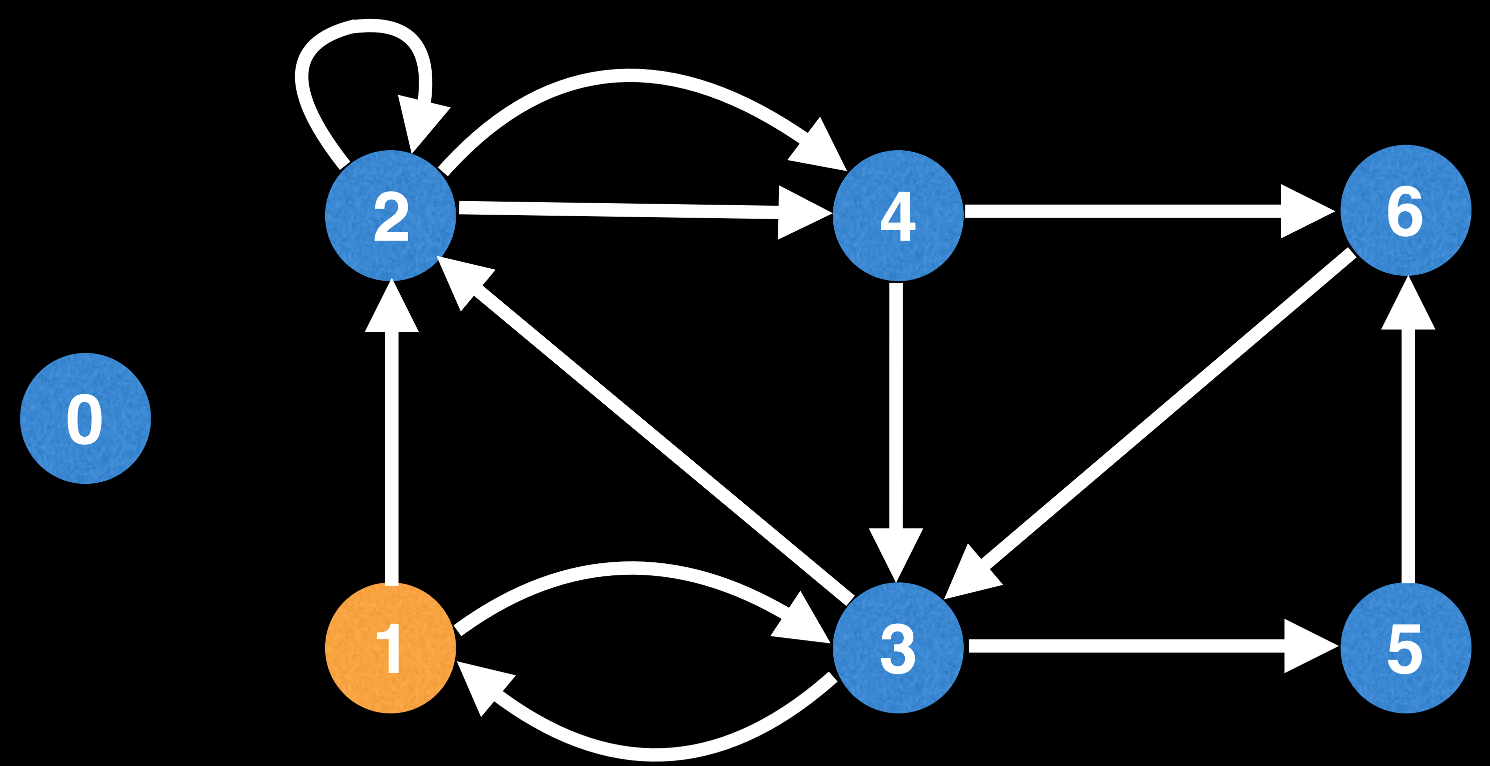
Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



Let's see what happens if we do a naive DFS, trying to traverse as many edges as possible until we get stuck.

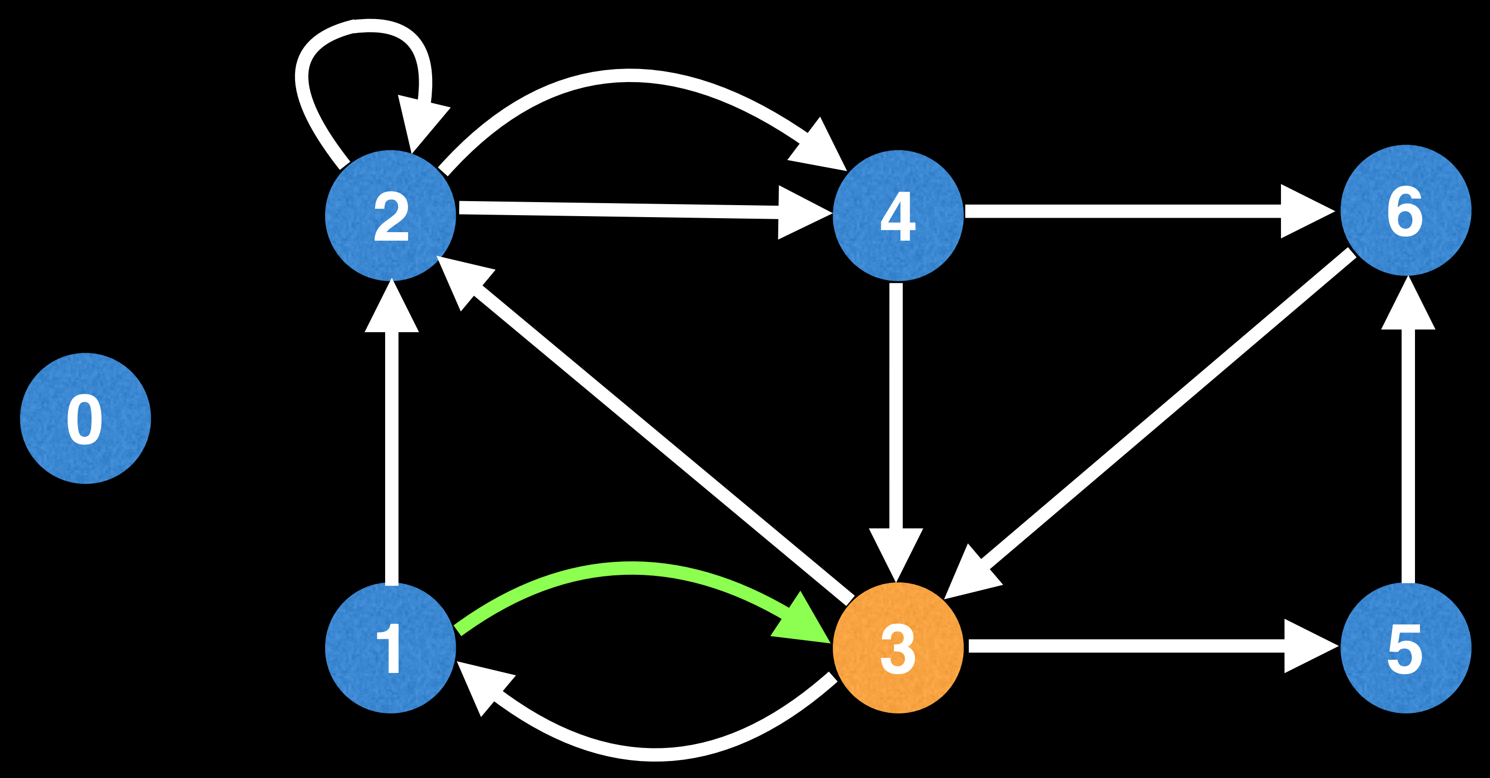
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



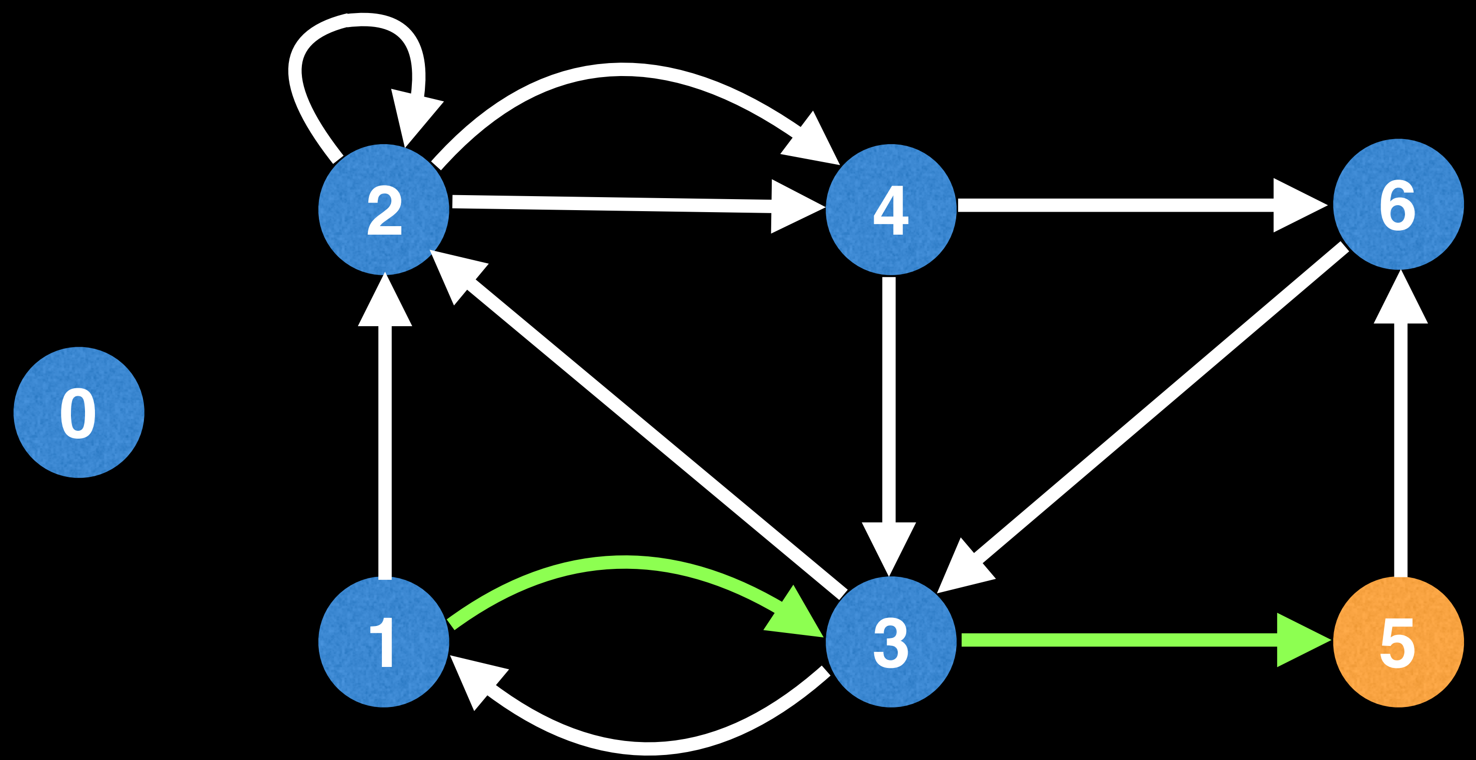
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



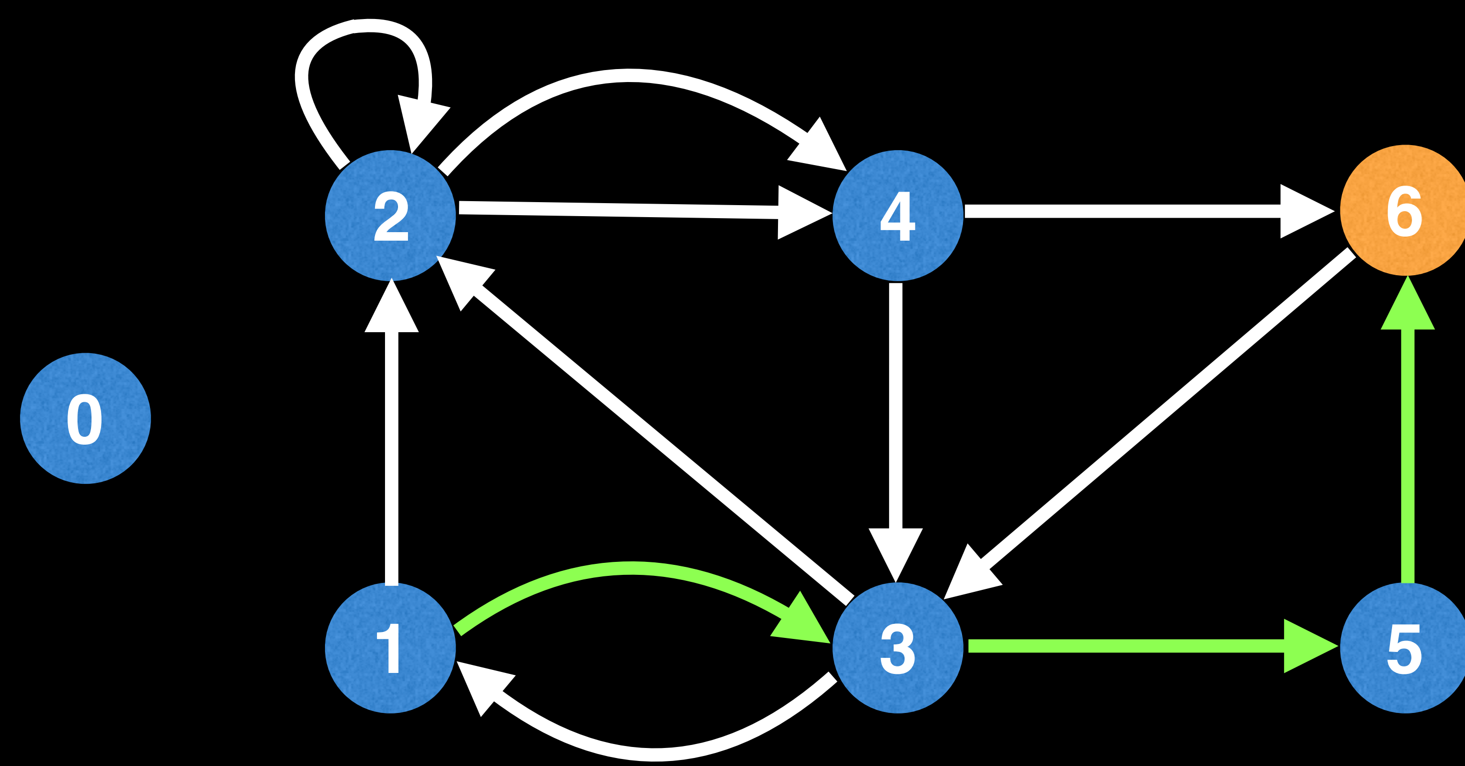
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



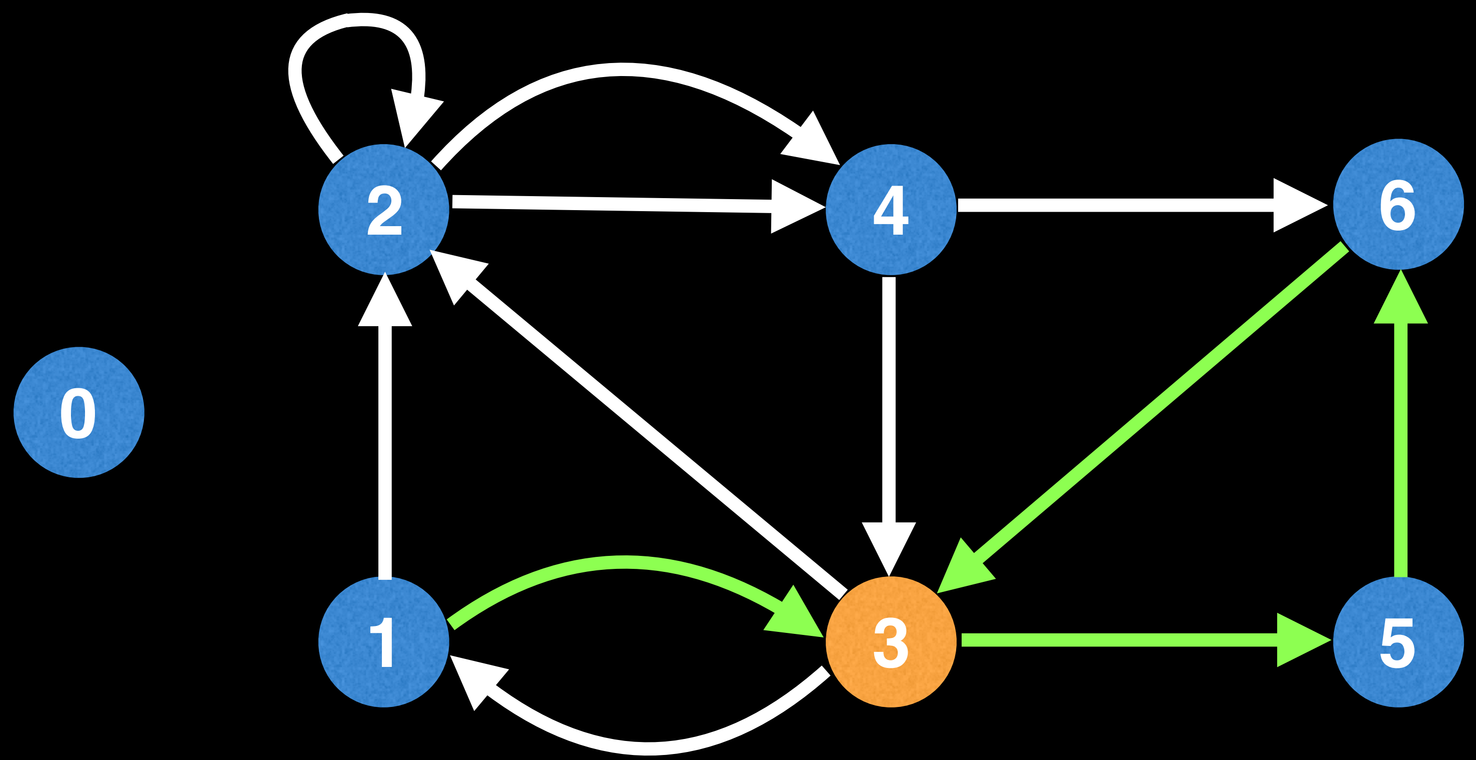
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



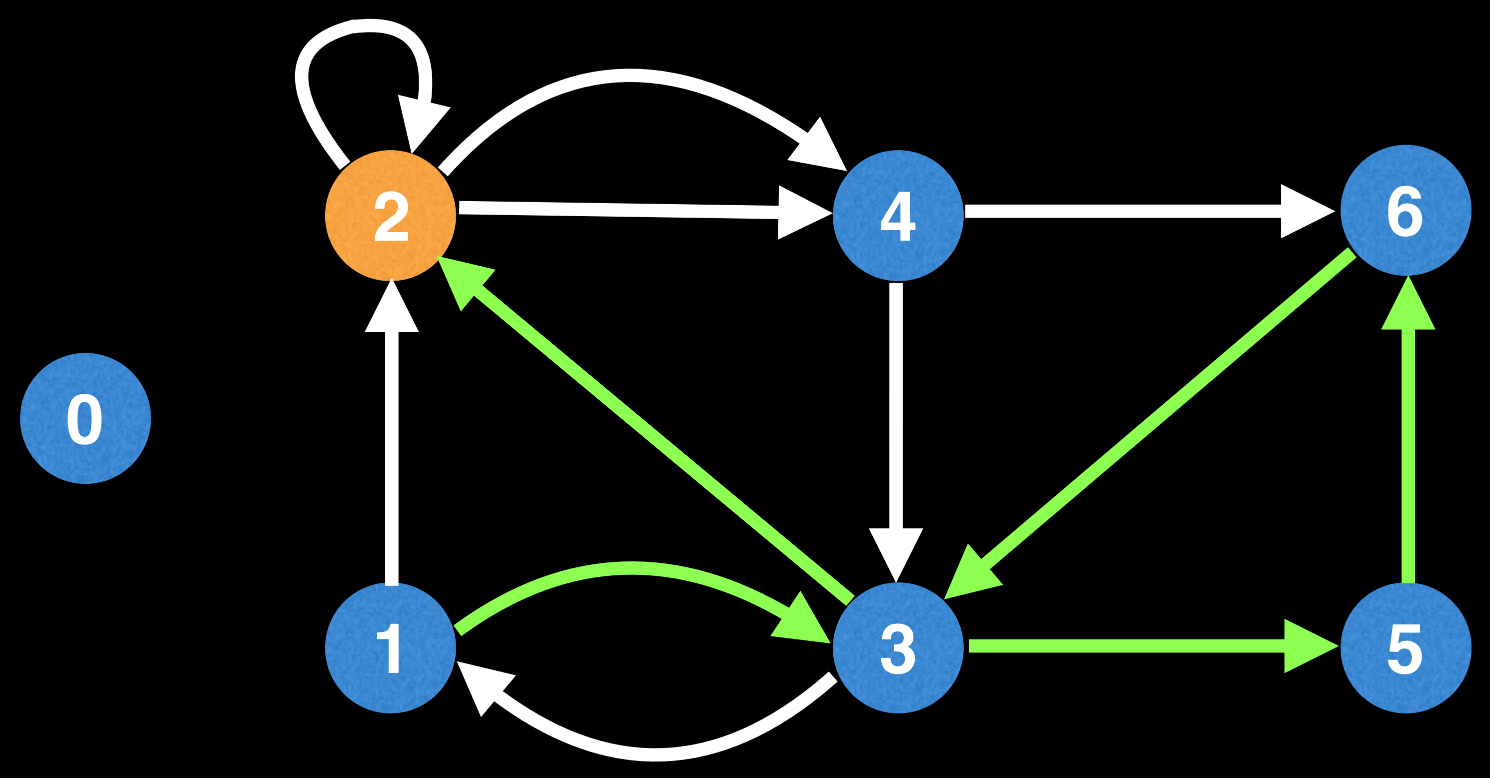
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



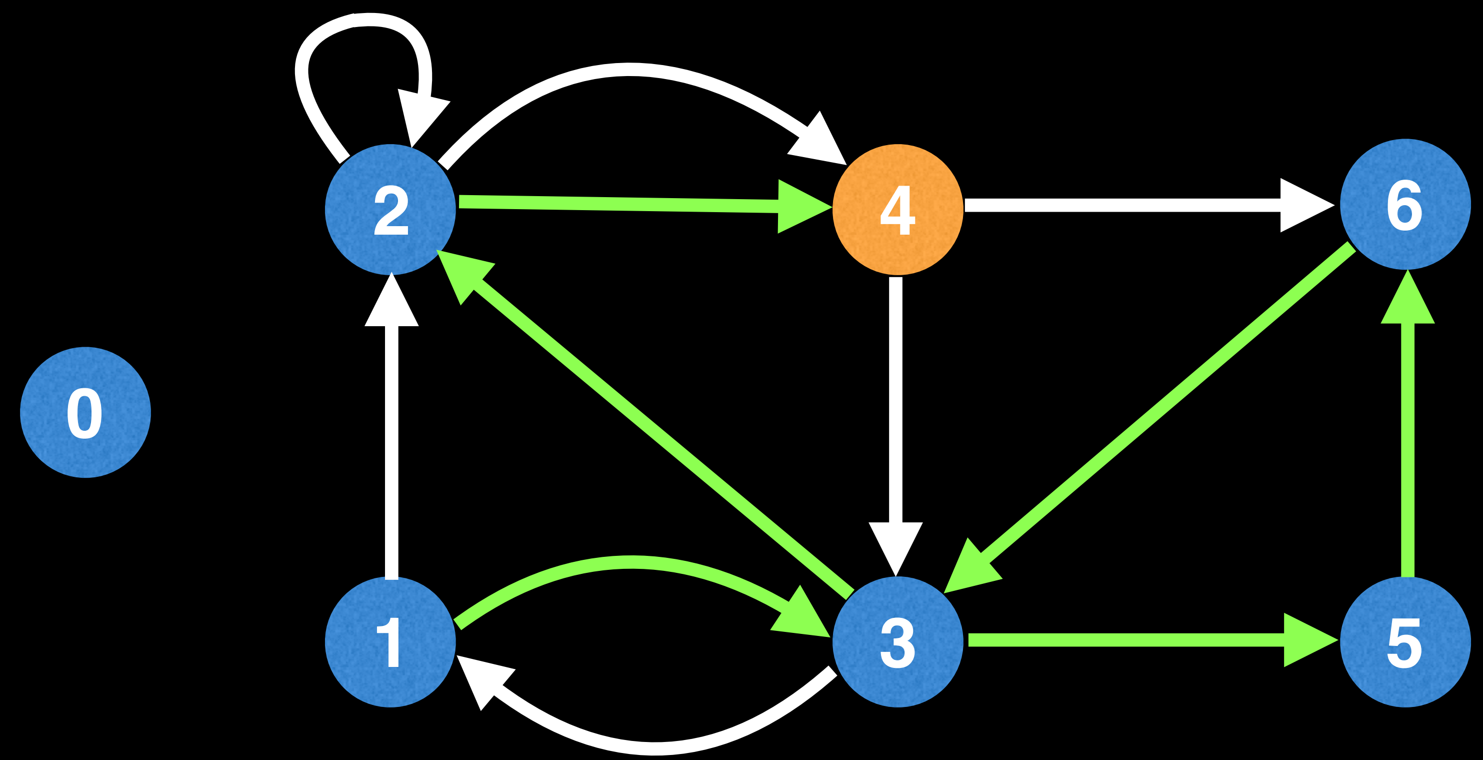
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



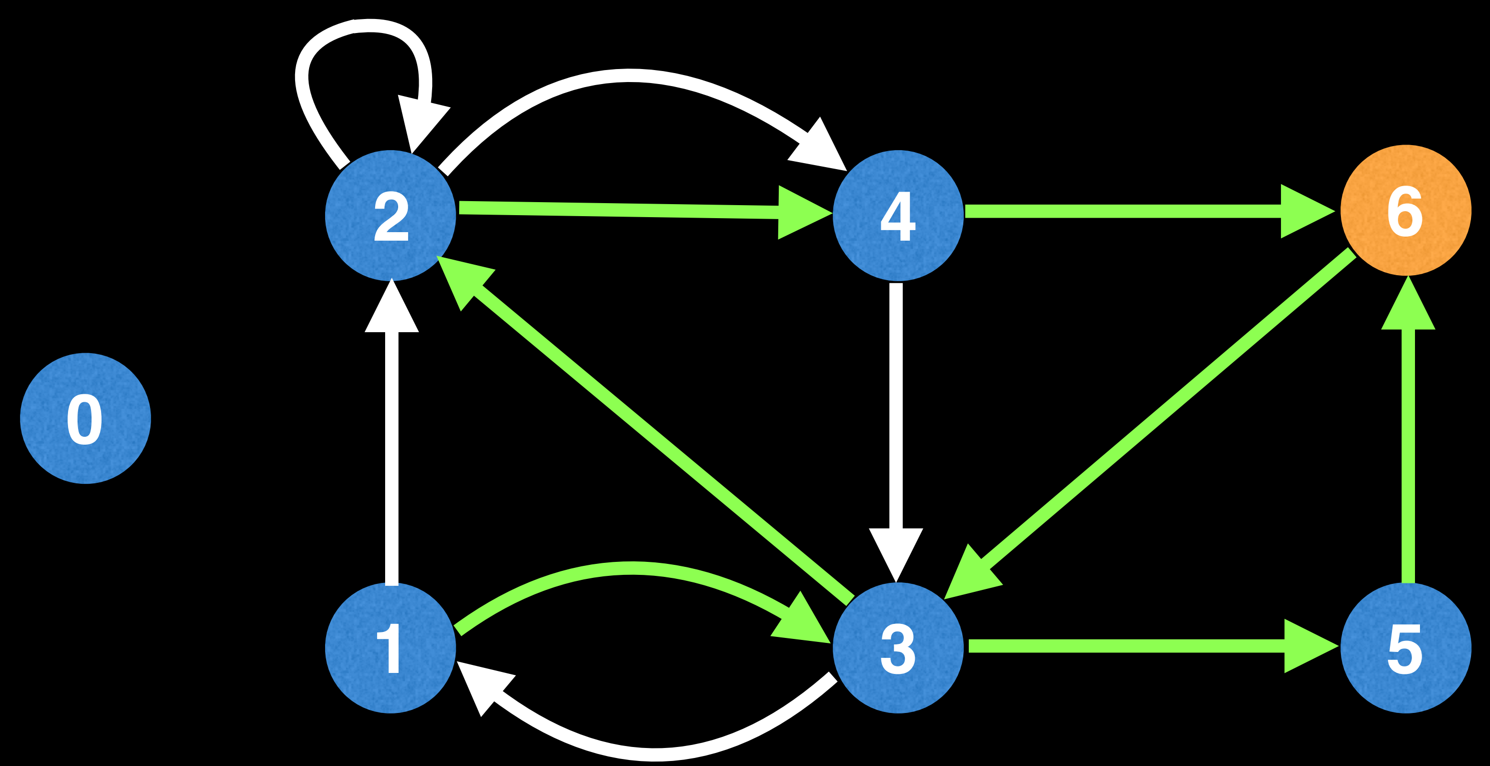
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



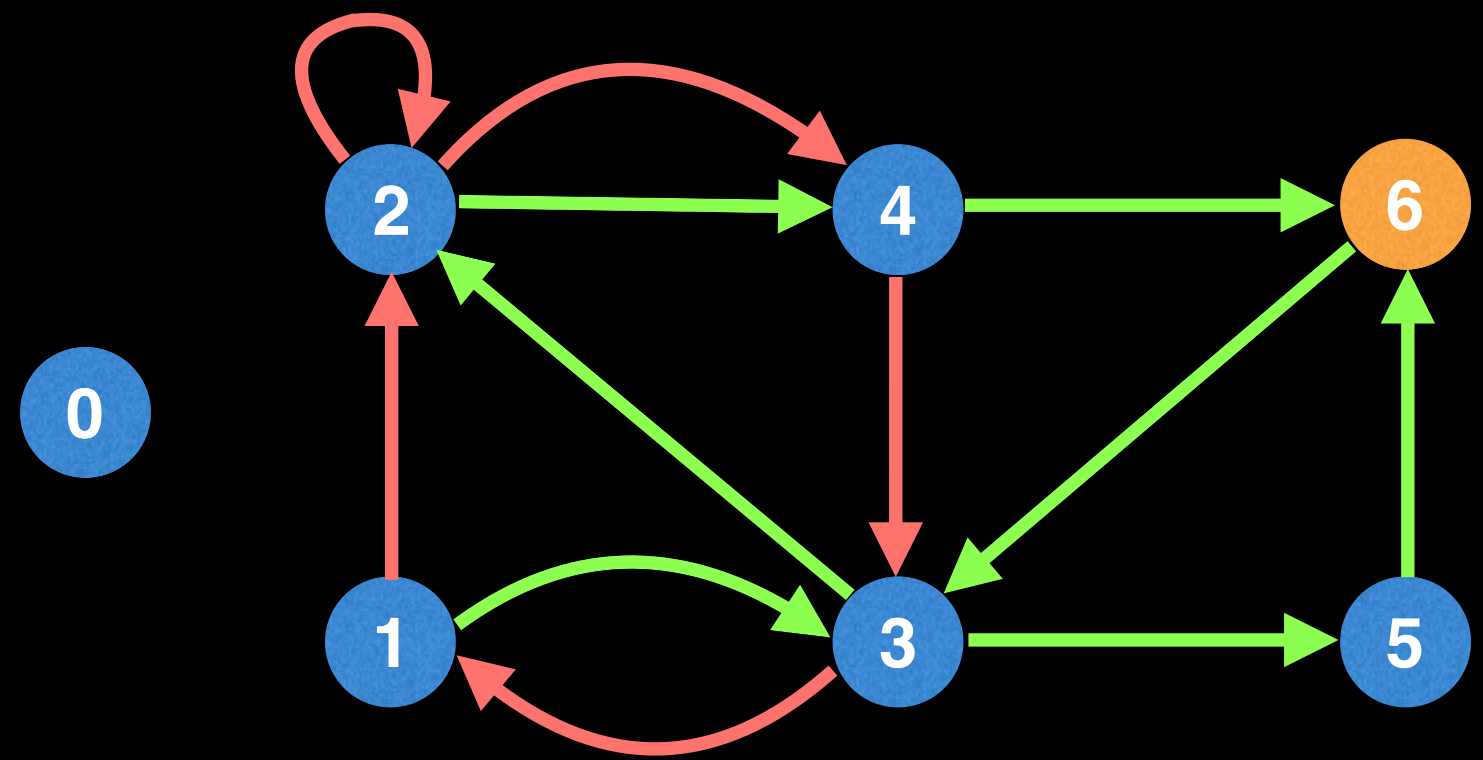
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

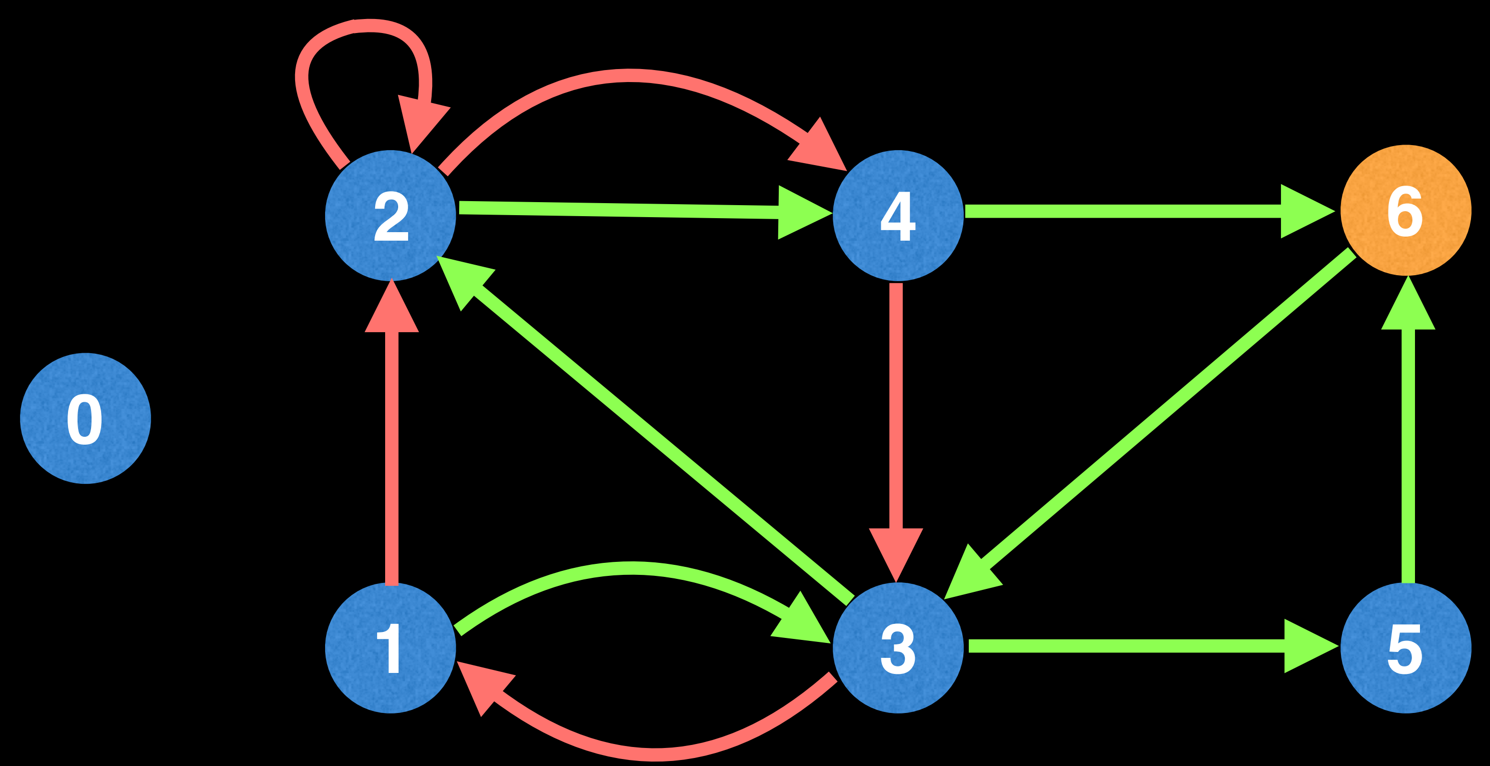


By randomly selecting edges during the DFS we made it from the start node to the end node.

However, we did not find an Eulerian path because we didn't traverse all the edges in our graph!

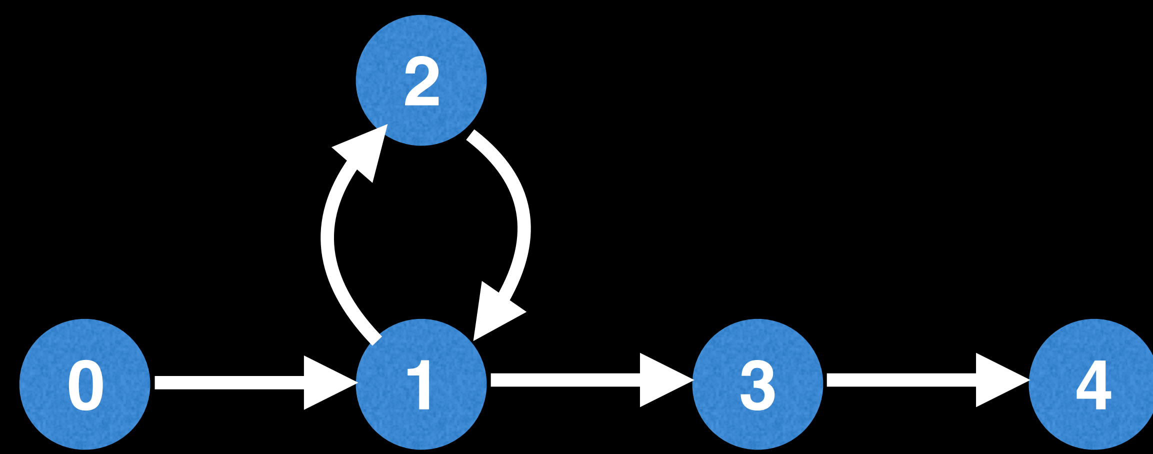
Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



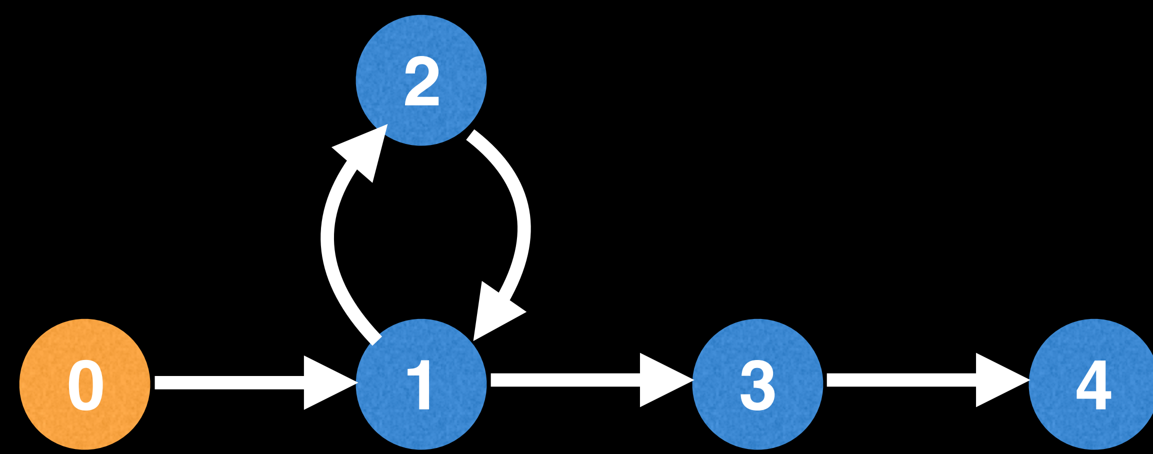
The good news is we can modify our DFS to handle forcing the traversal of all edges :)

Finding an Eulerian path (directed graph)



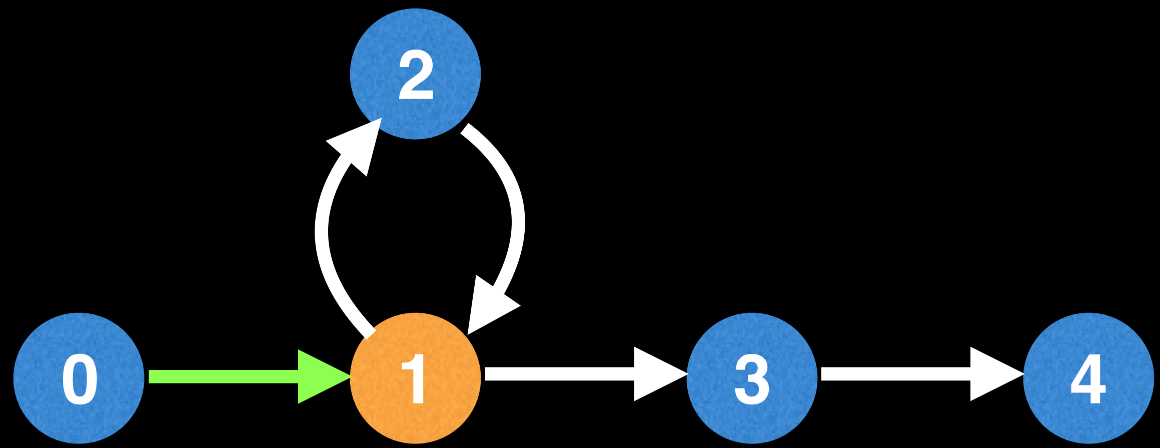
To illustrate this, consider starting at node 0 and trying to find an Eulerian path.

Finding an Eulerian path (directed graph)



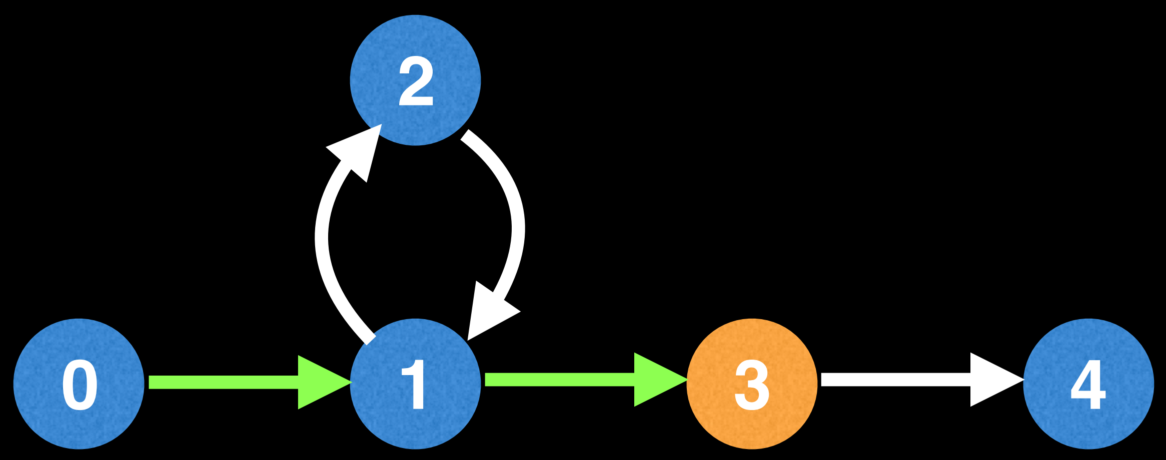
To illustrate this, consider starting at node 0 and trying to find an Eulerian path.

Finding an Eulerian path (directed graph)



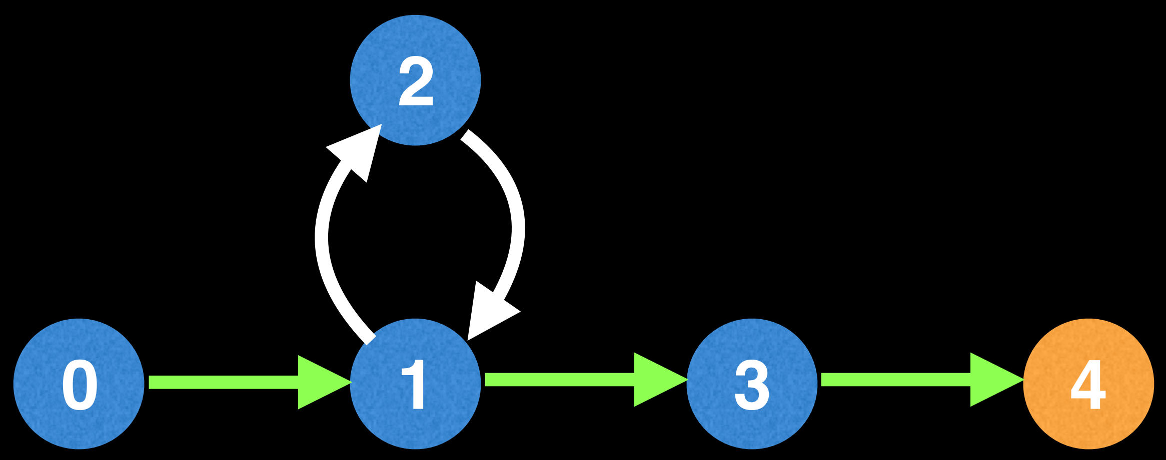
To illustrate this, consider starting at node 0 and trying to find an Eulerian path.

Finding an Eulerian path (directed graph)



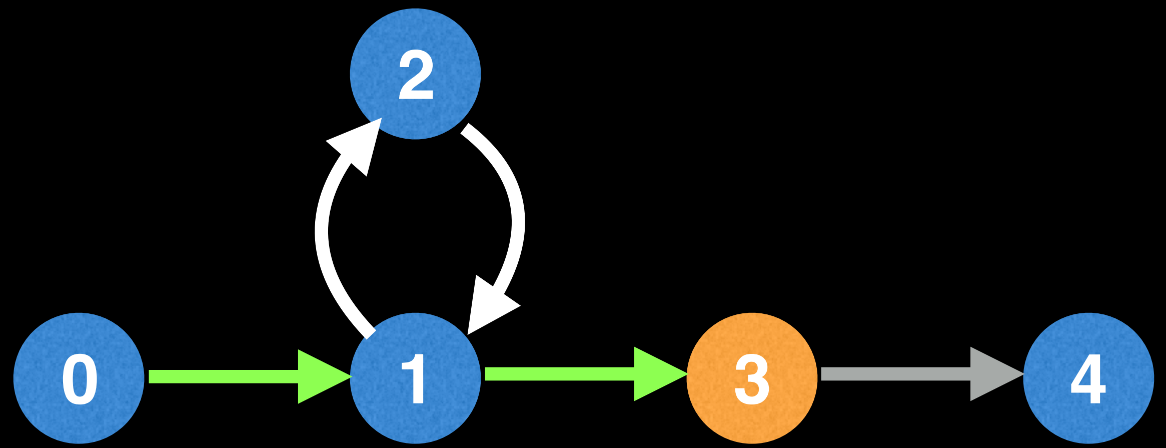
Whoops… we skipped the edges going to node 2 and back which need to be part of the solution.

Finding an Eulerian path (directed graph)



Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

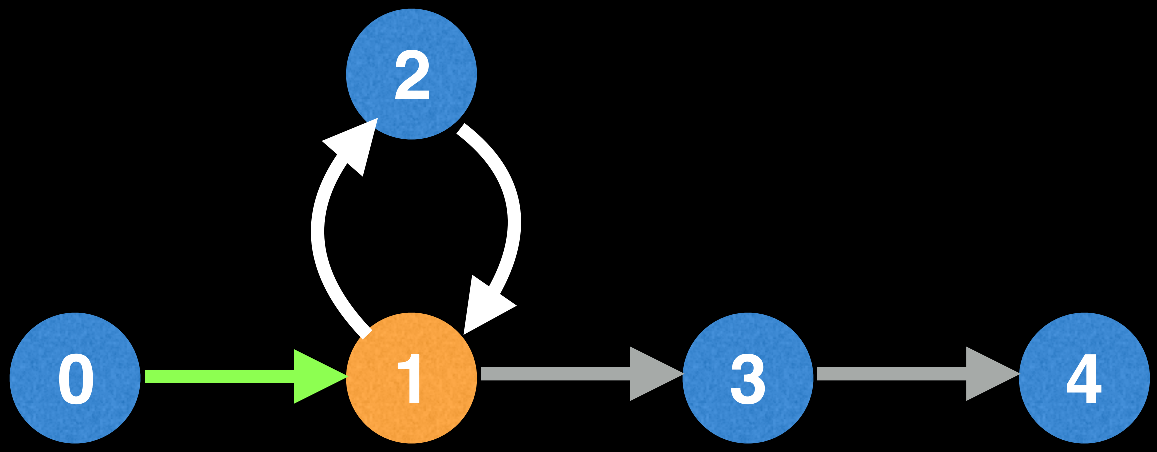
Finding an Eulerian path (directed graph)



Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [4]

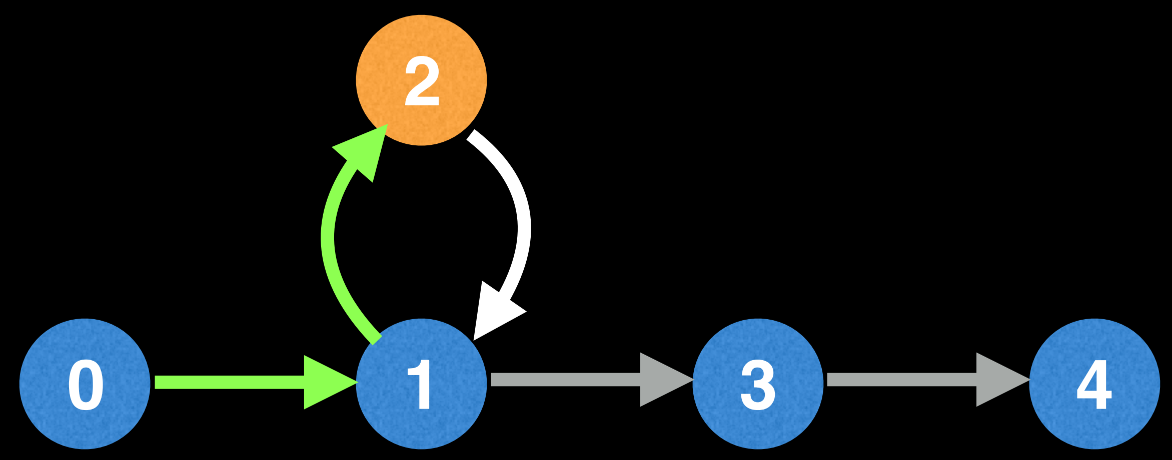
Finding an Eulerian path (directed graph)



When backtracking, if the current node has any remaining unvisited edges (white edges) we follow any of them calling our DFS method recursively to extend the Eulerian path.

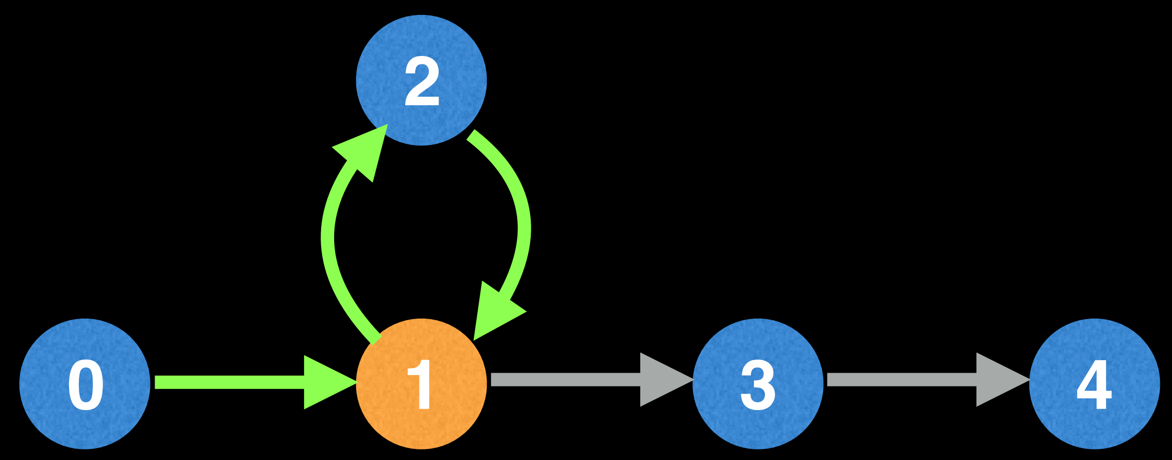
Solution: [3, 4]

Finding an Eulerian path (directed graph)



Solution: [3, 4]

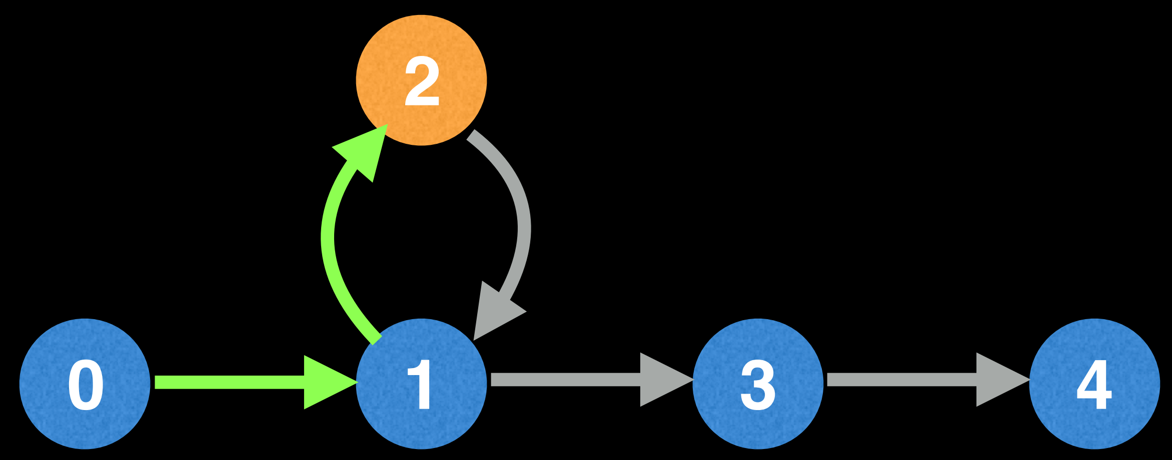
Finding an Eulerian path (directed graph)



Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [3, 4]

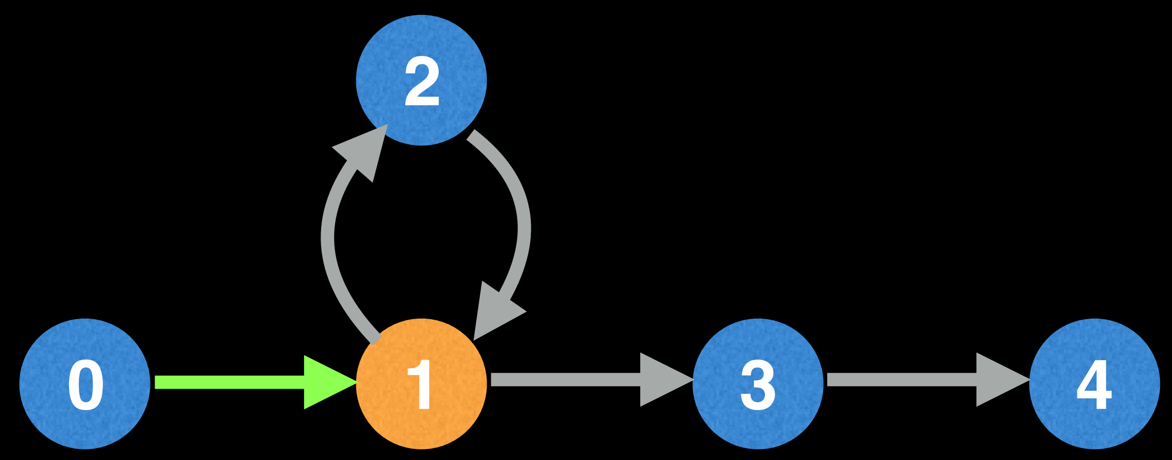
Finding an Eulerian path (directed graph)



Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [1, 3, 4]

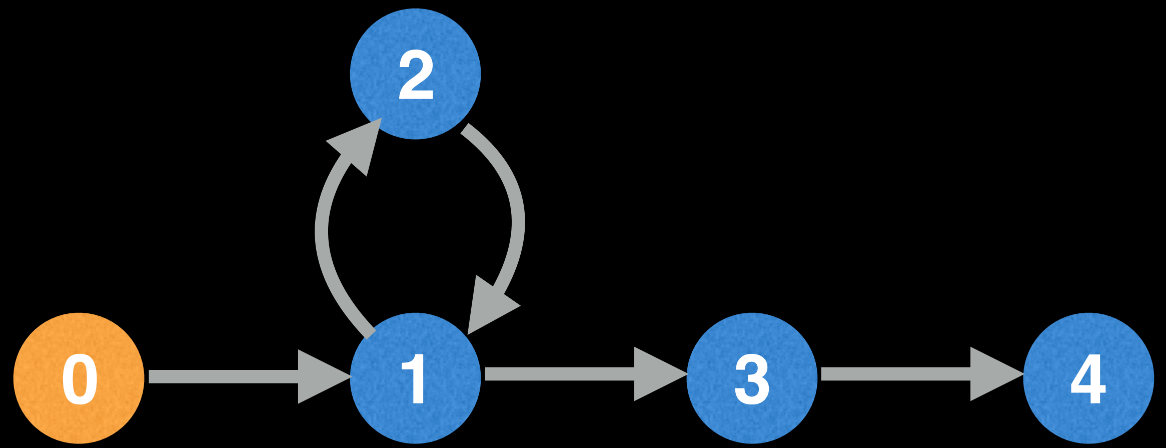
Finding an Eulerian path (directed graph)



Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [2, 1, 3, 4]

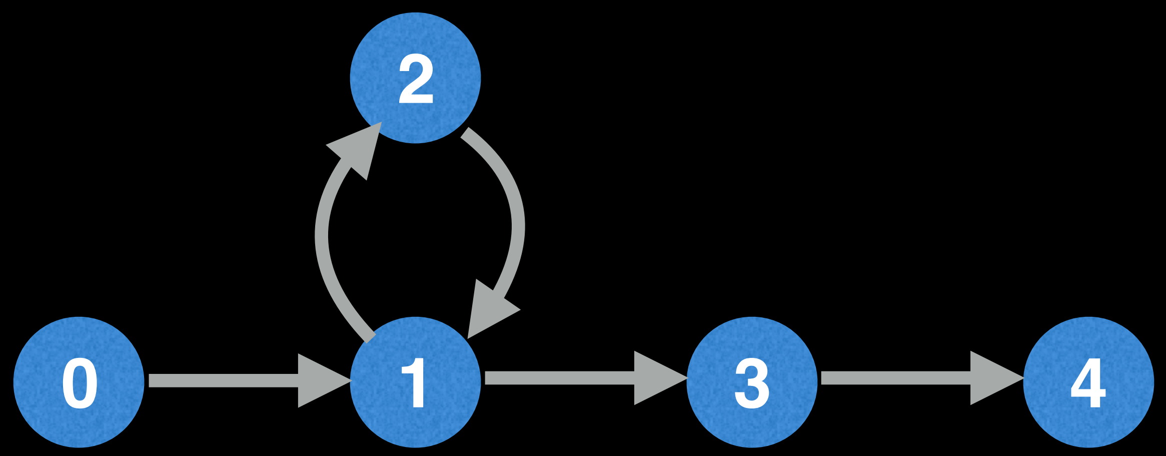
Finding an Eulerian path (directed graph)



Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [1, 2, 1, 3, 4]

Finding an Eulerian path (directed graph)

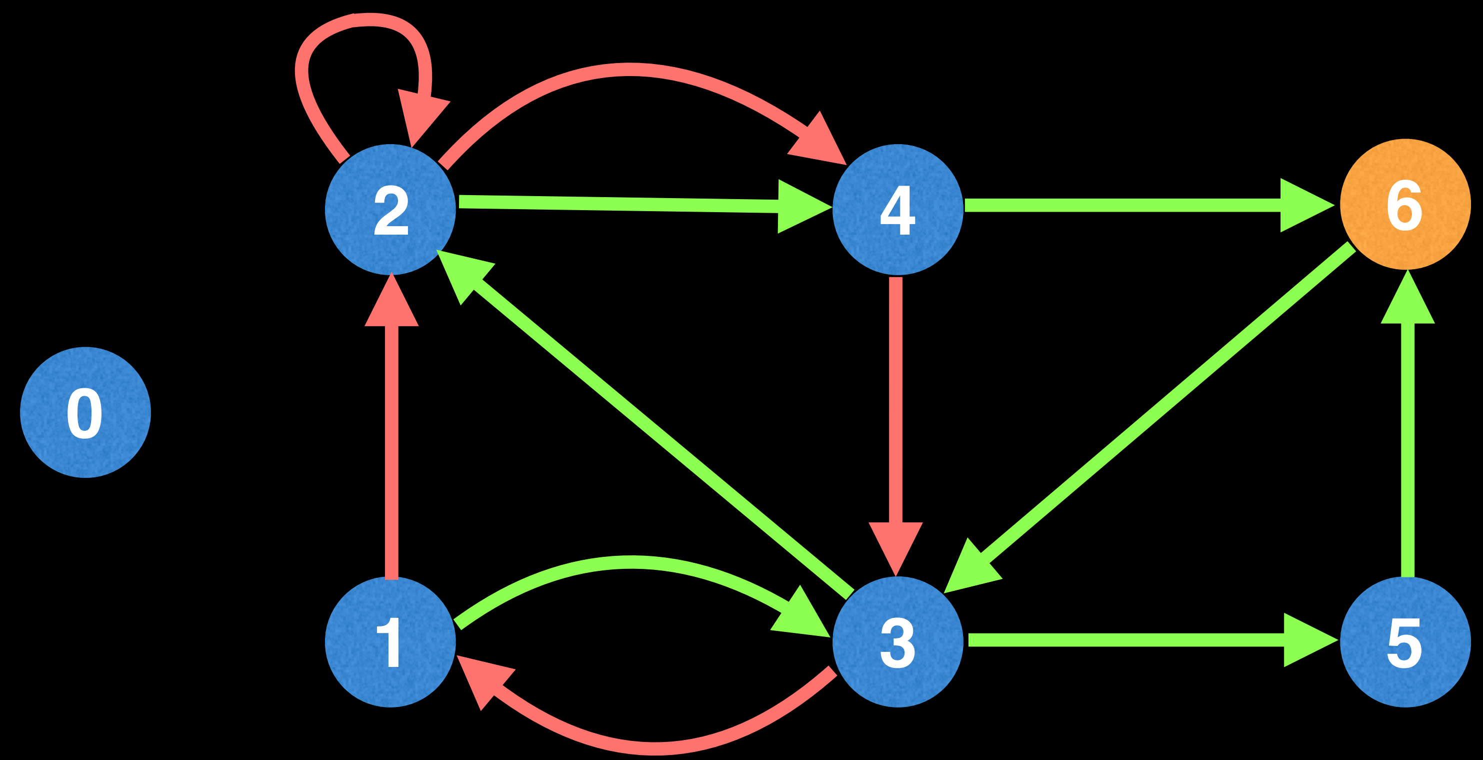


Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [0, 1, 2, 1, 3, 4]

Finding an Eulerian path (directed graph)

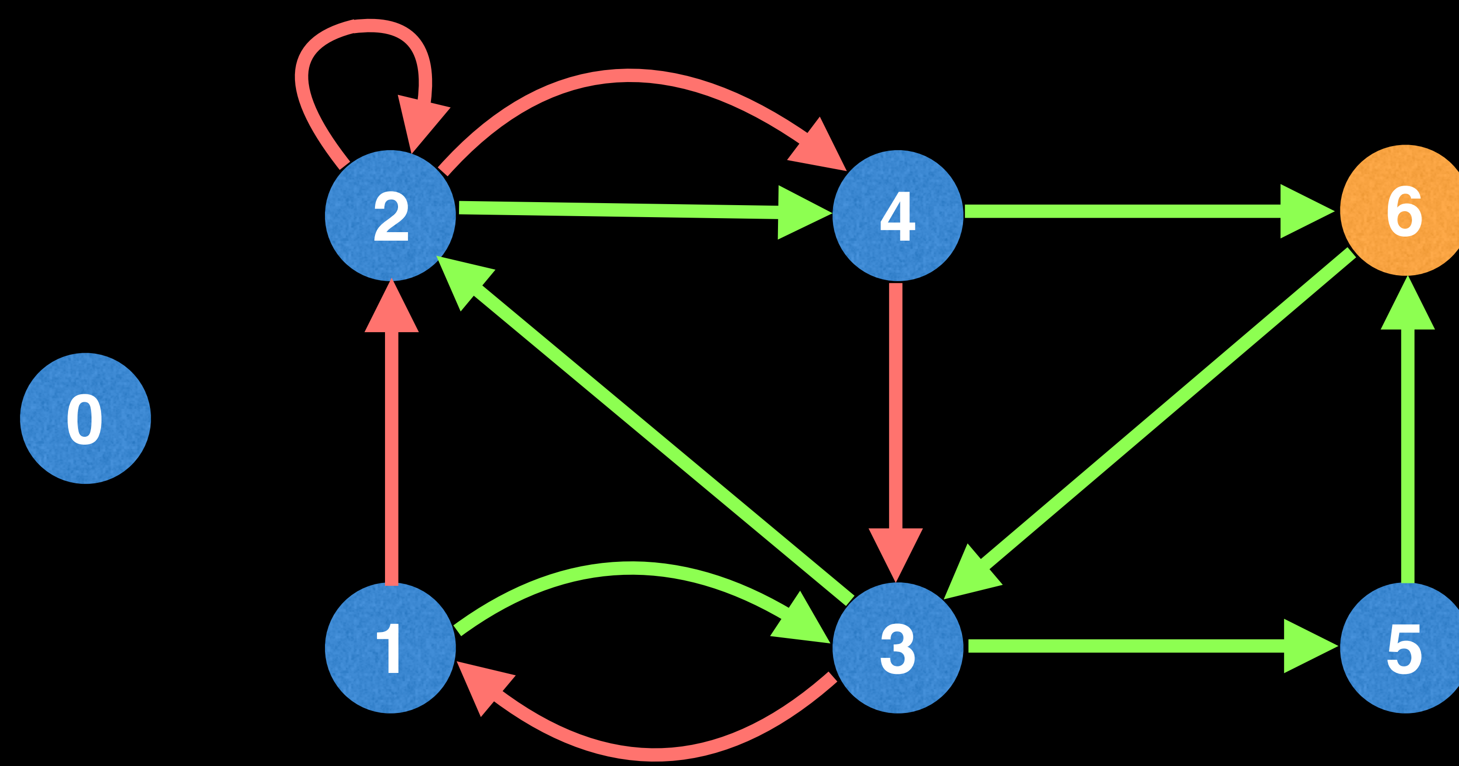
Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1



Coming back to the previous example, let's restart the algorithm, but this time track the number of unvisited edges we have left to take for each node.

Finding an Eulerian path (directed graph)

Node	[illegible]	Out
0	[illegible]	0
1	[illegible]	2
2	[illegible]	3
3	[illegible]	3
4	[illegible]	2
5	[illegible]	1
6	[illegible]	1

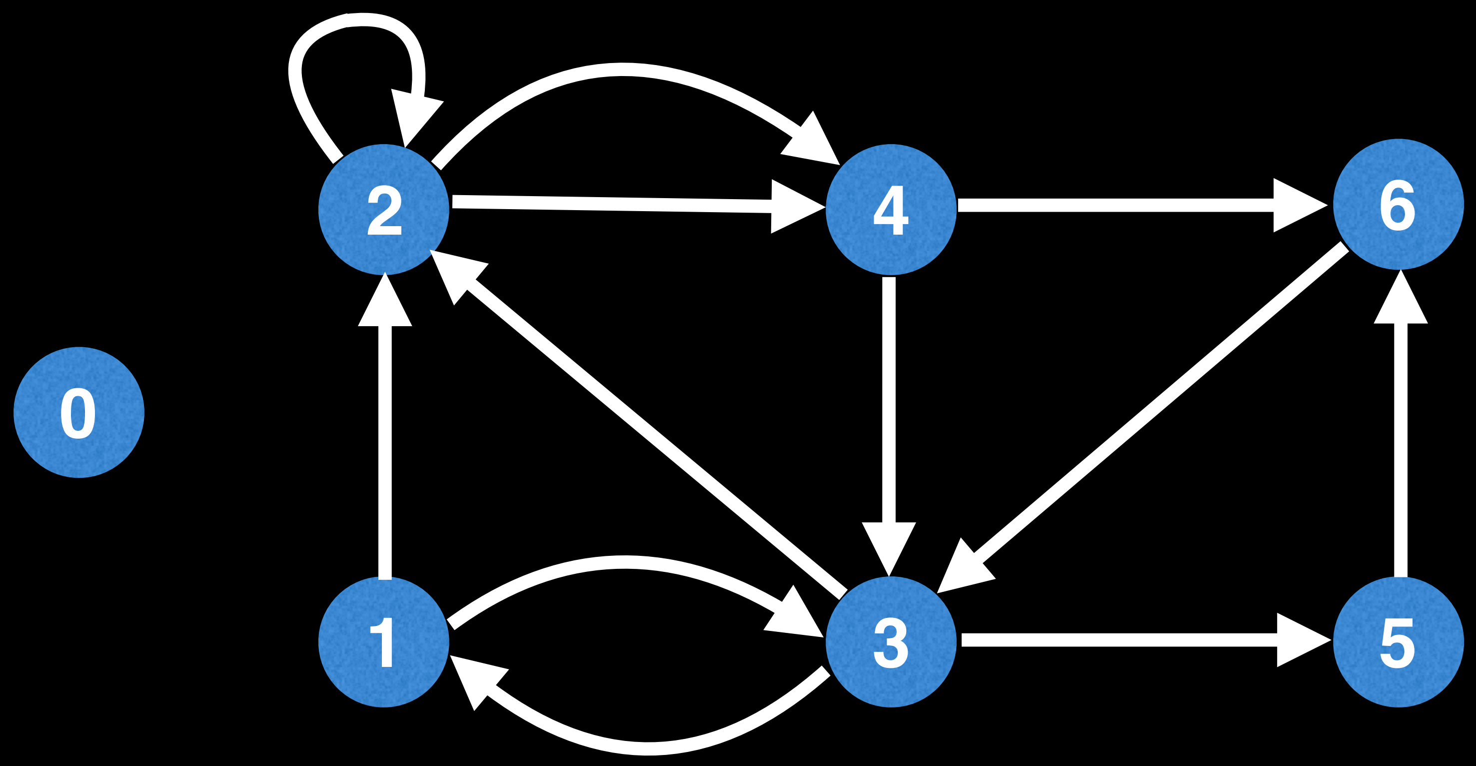


In fact, we have already computed the number of outgoing edges for each edge in the “out” array which we can reuse.

We won’t be needing the “in” array after we’ve validated that an Eulerian path exists.

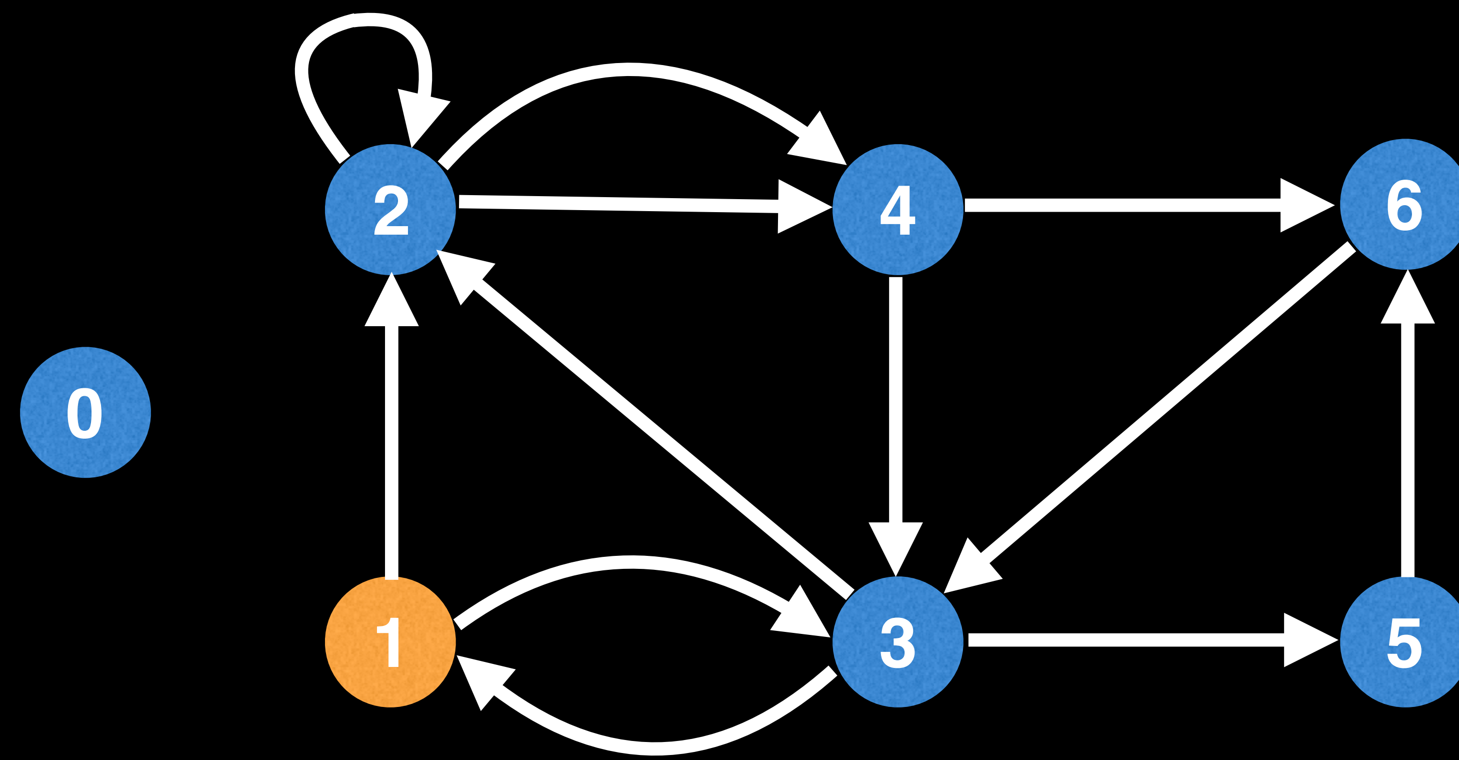
Finding an Eulerian path (directed graph)

Node	Out
0	0
1	2
2	3
3	3
4	2
5	1
6	1



Finding an Eulerian path (directed graph)

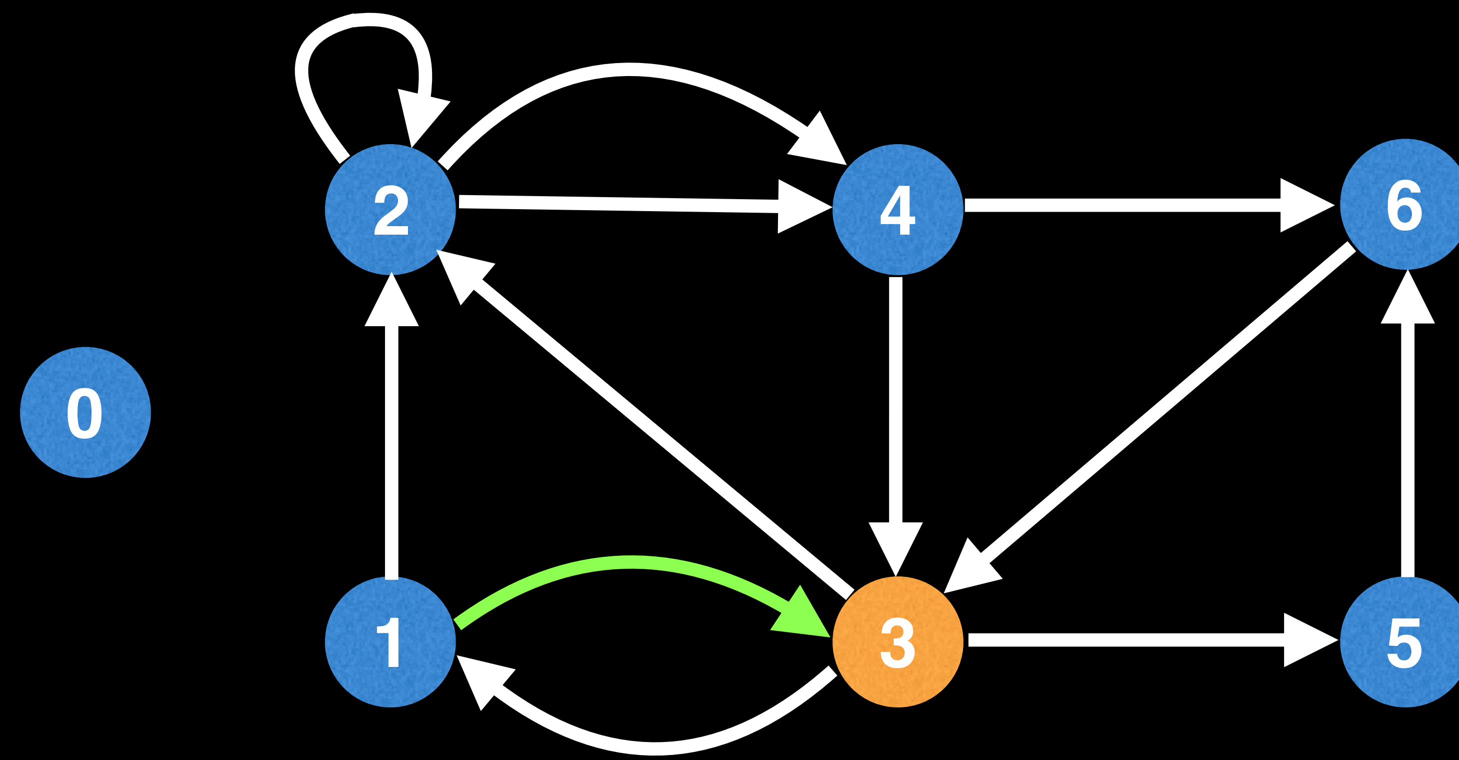
Node	Out
0	0
1	2
2	3
3	3
4	2
5	1
6	1



Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	3
3	3
4	2
5	1
6	1

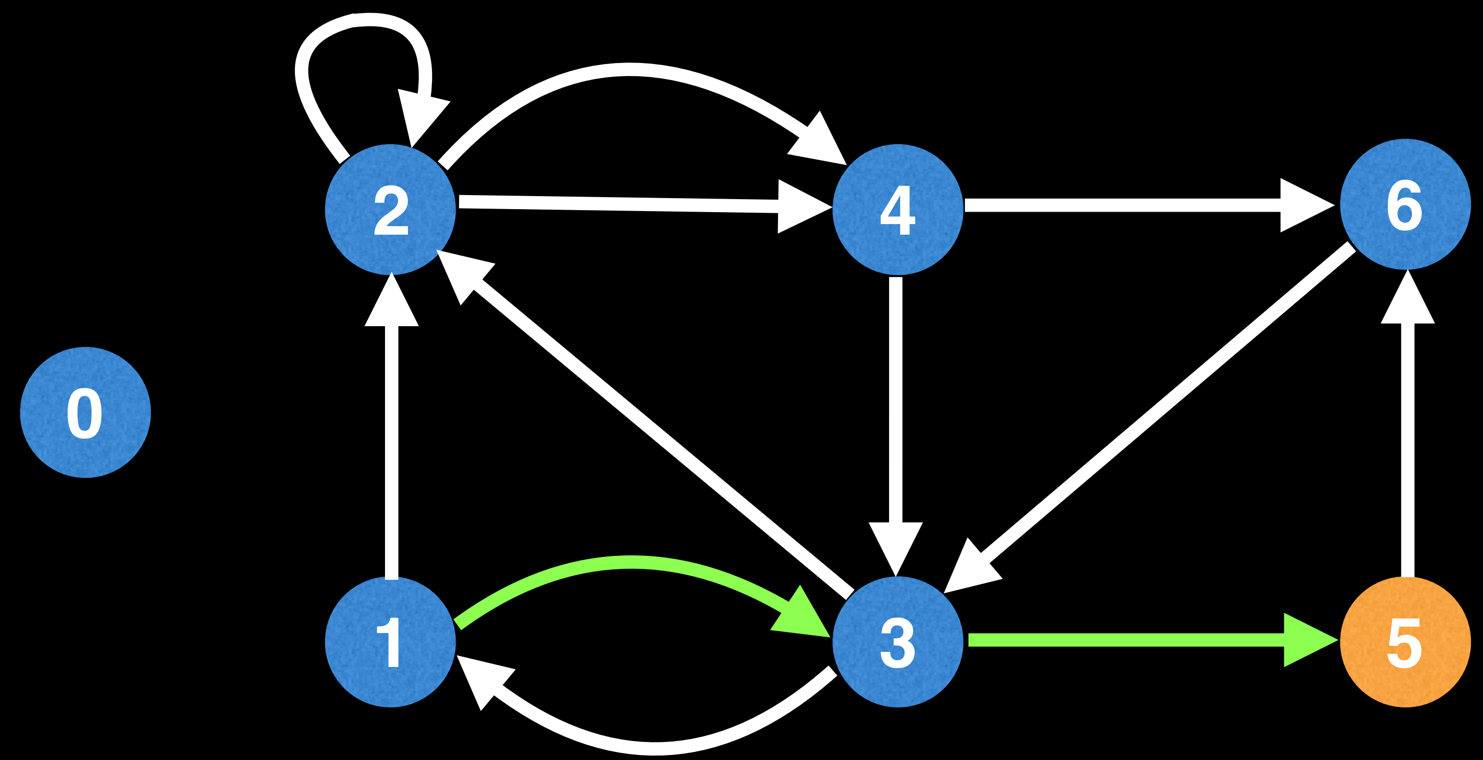


Every time an edge is taken, reduce the outgoing edge count in the out array.

Solution = []

Finding an Eulerian path (directed graph)

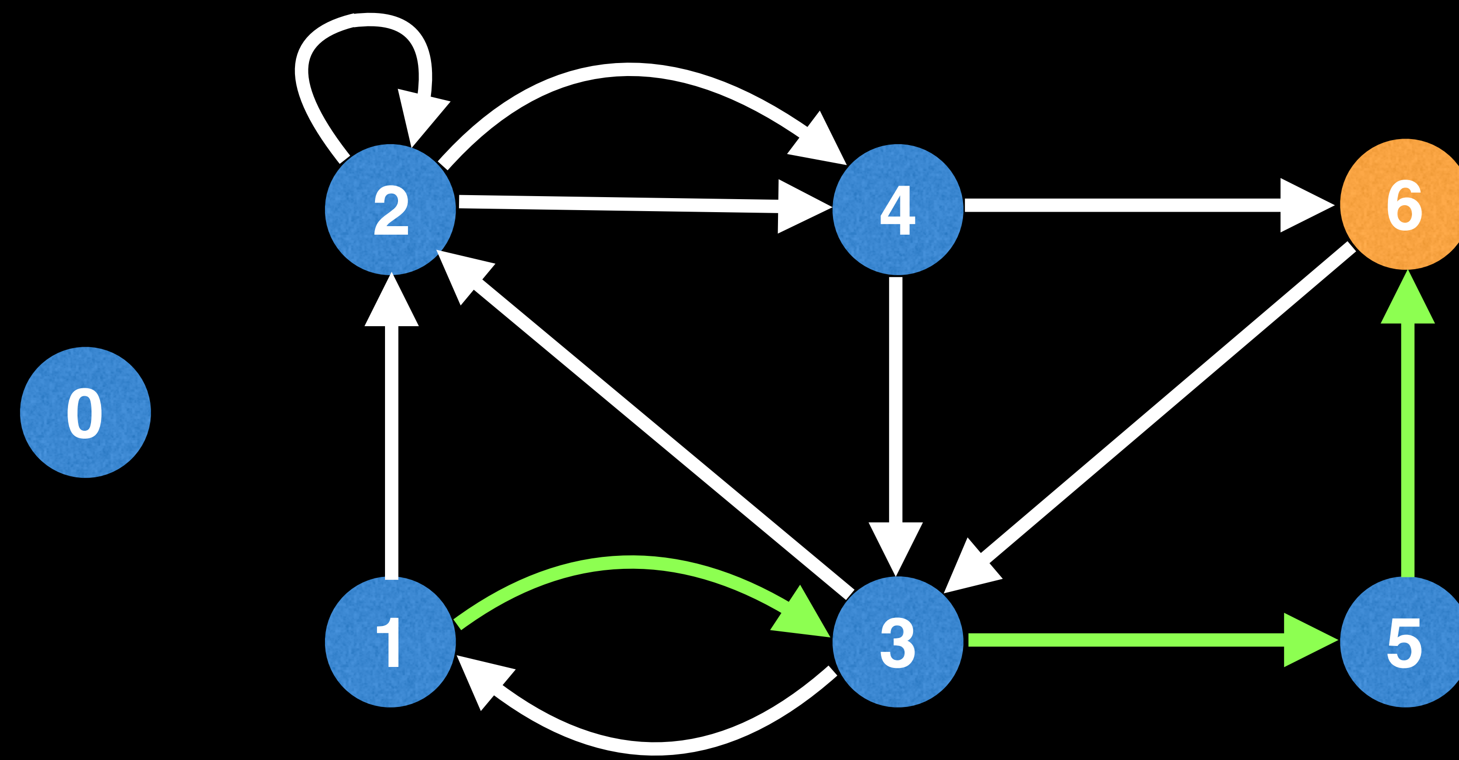
Node	Out
0	0
1	1
2	3
3	2
4	2
5	1
6	1



Solution = []

Finding an Eulerian path (directed graph)

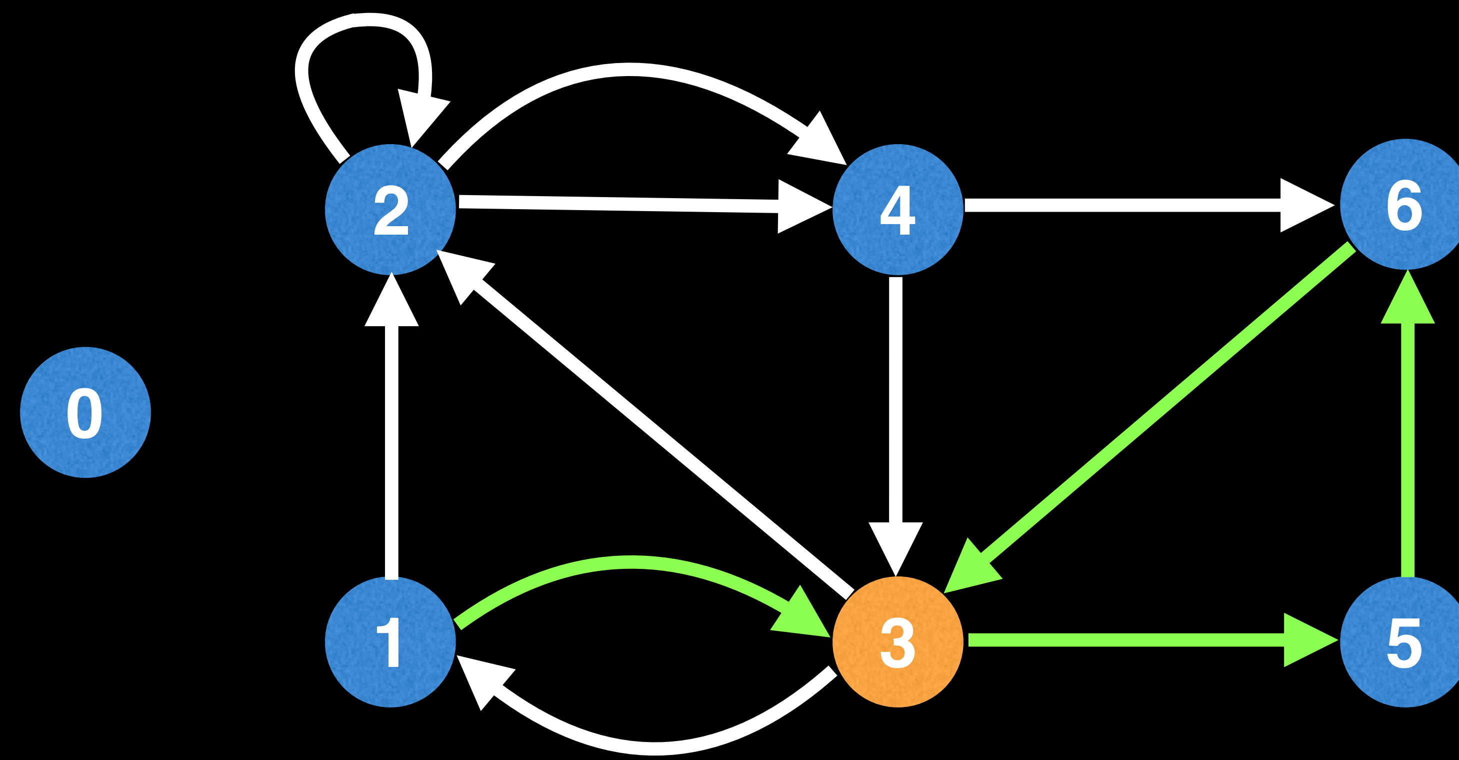
Node	Out
0	0
1	1
2	3
3	2
4	2
5	0
6	1



Solution = []

Finding an Eulerian path (directed graph)

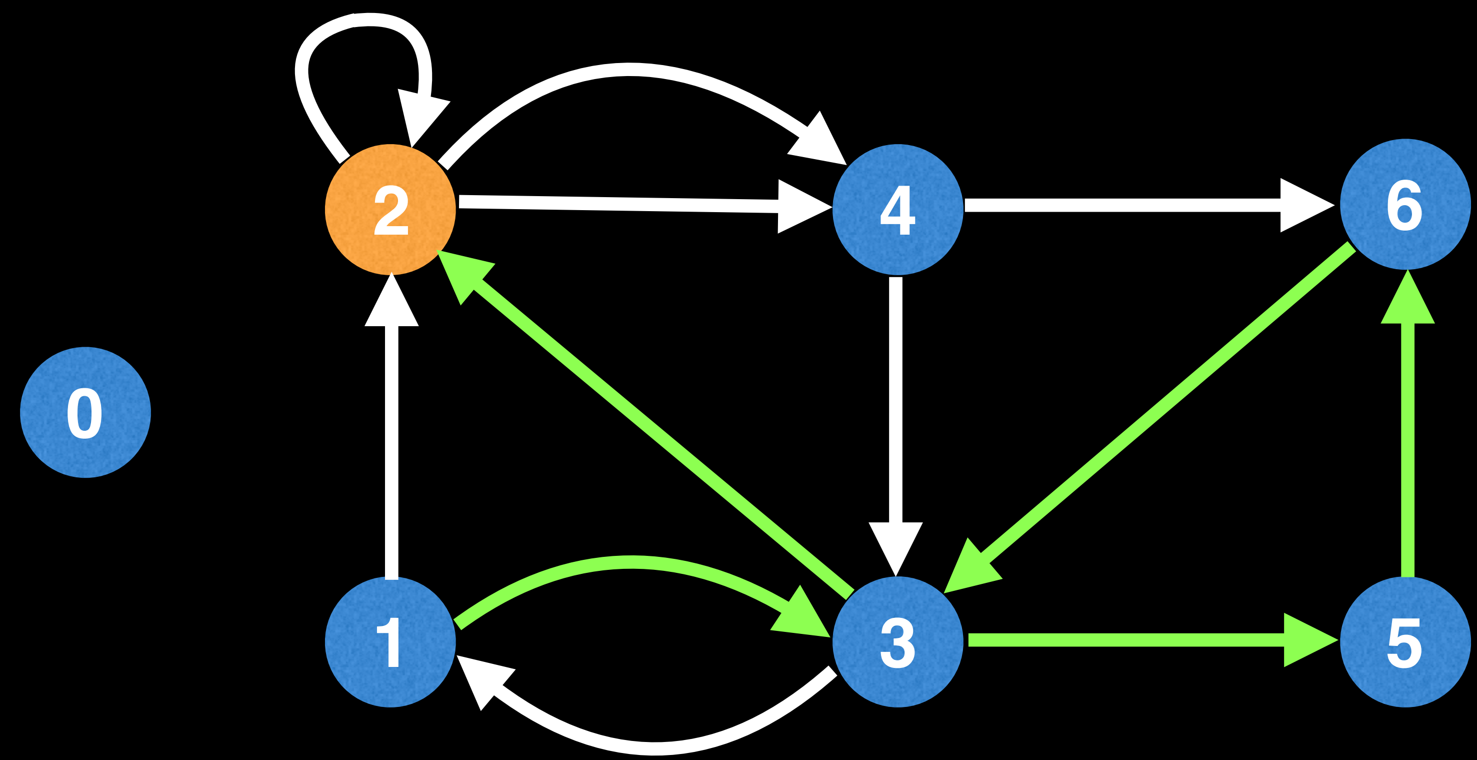
Node	Out
0	0
1	1
2	3
3	2
4	2
5	0
6	0



Solution = []

Finding an Eulerian path (directed graph)

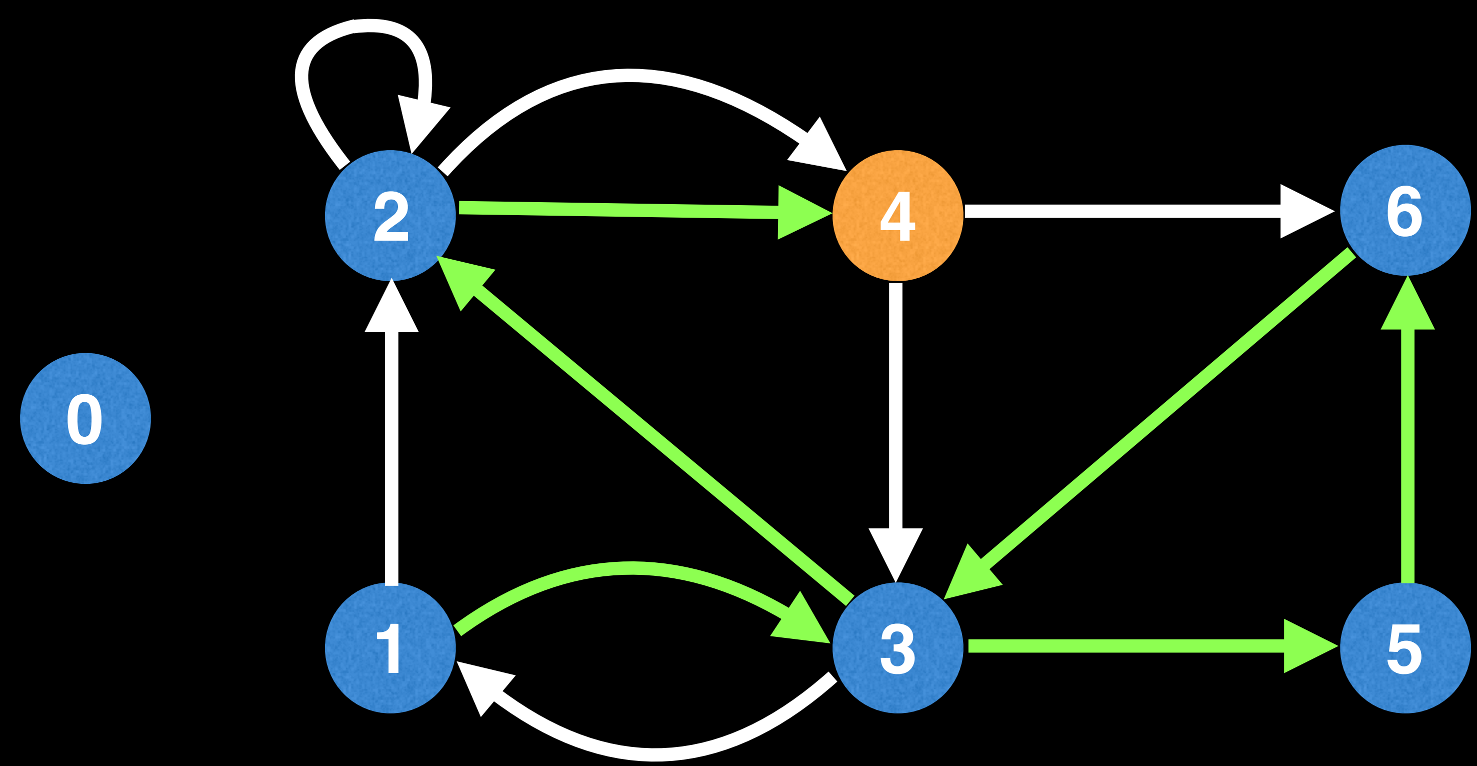
Node	Out
0	0
1	1
2	3
3	1
4	2
5	0
6	0



Solution = []

Finding an Eulerian path (directed graph)

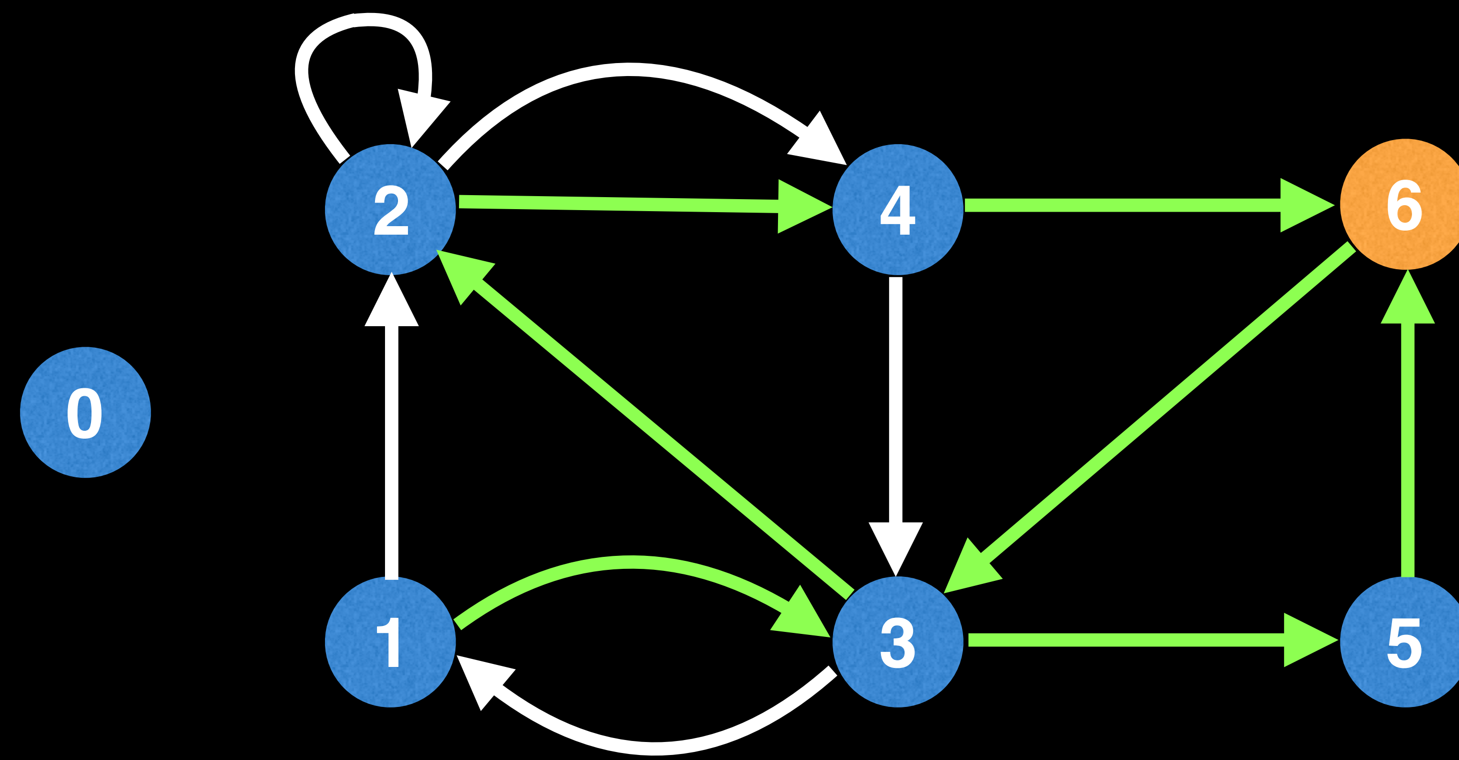
Node	Out
0	0
1	1
2	2
3	1
4	2
5	0
6	0



Solution = []

Finding an Eulerian path (directed graph)

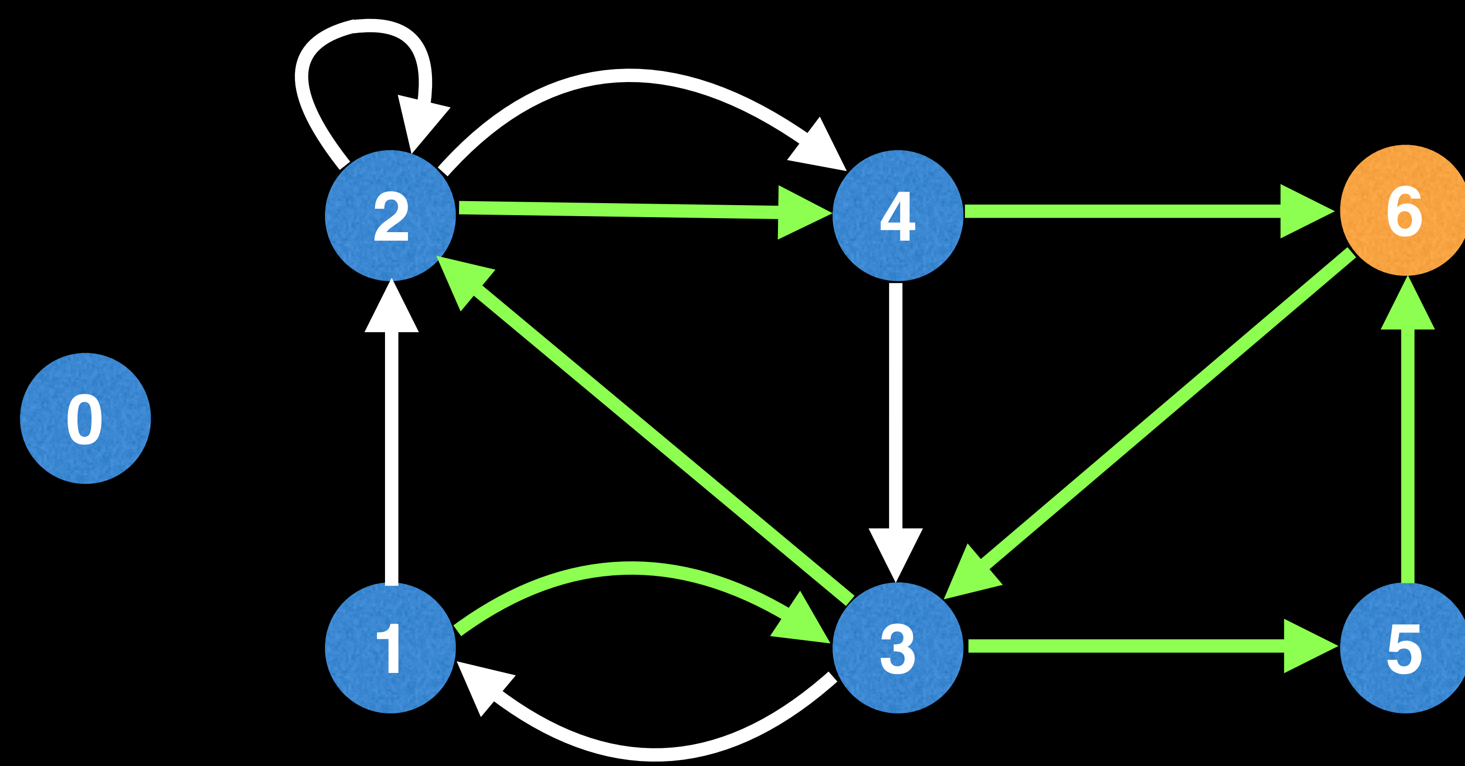
Node	Out
0	0
1	1
2	2
3	1
4	1
5	0
6	0



Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	2
3	1
4	1
5	0
6	0

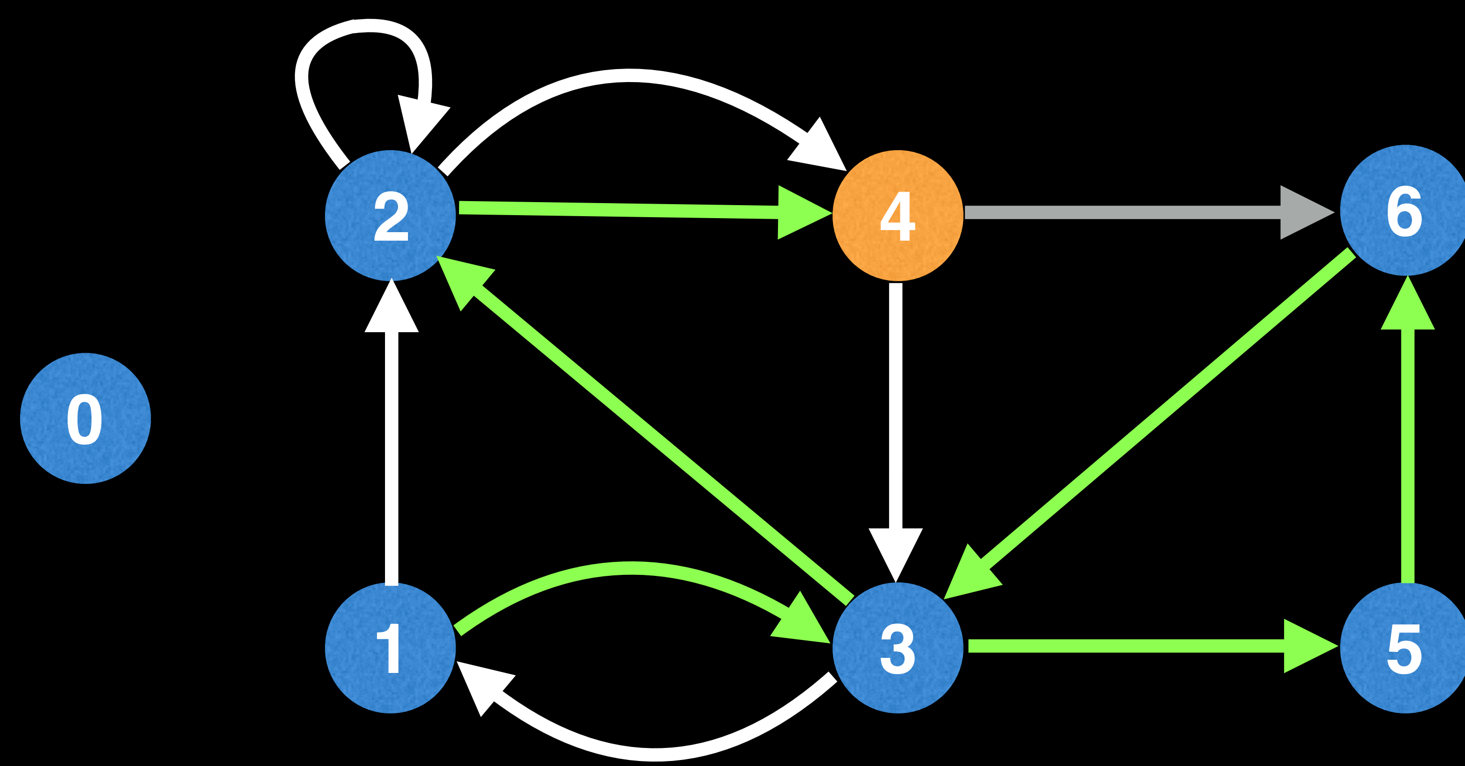


When the DFS is stuck, meaning there are no more outgoing edges (i.e out[i] = 0), then we know to backtrack and add the current node to the solution.

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	2
3	1
4	1
5	0
6	0

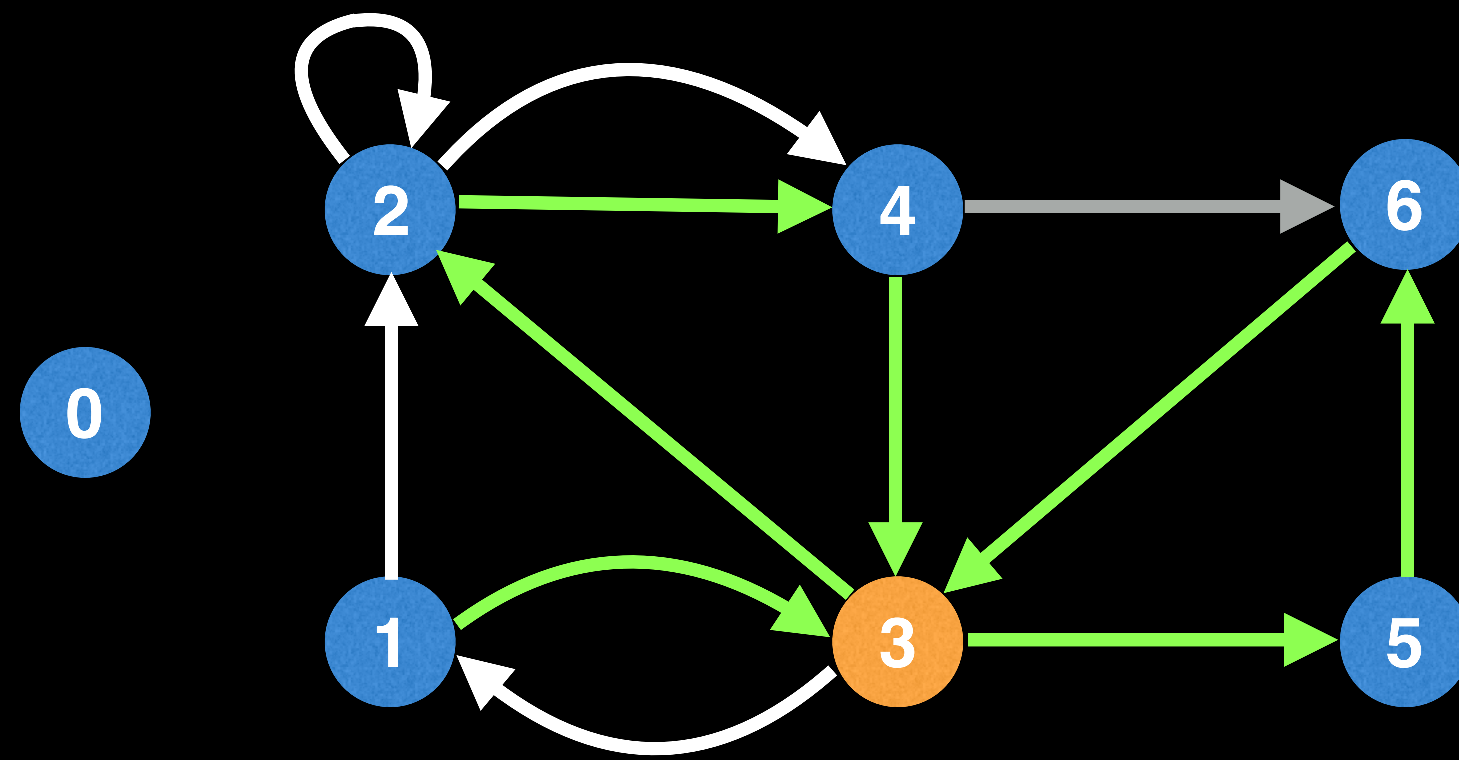


When backtracking, if the current node has any remaining unvisited edges (white edges), we follow any of them, calling our DFS method recursively to extend the Eulerian path. We can verify there still are outgoing edges by checking if out[i] != 0.

Solution = [6]

Finding an Eulerian path (directed graph)

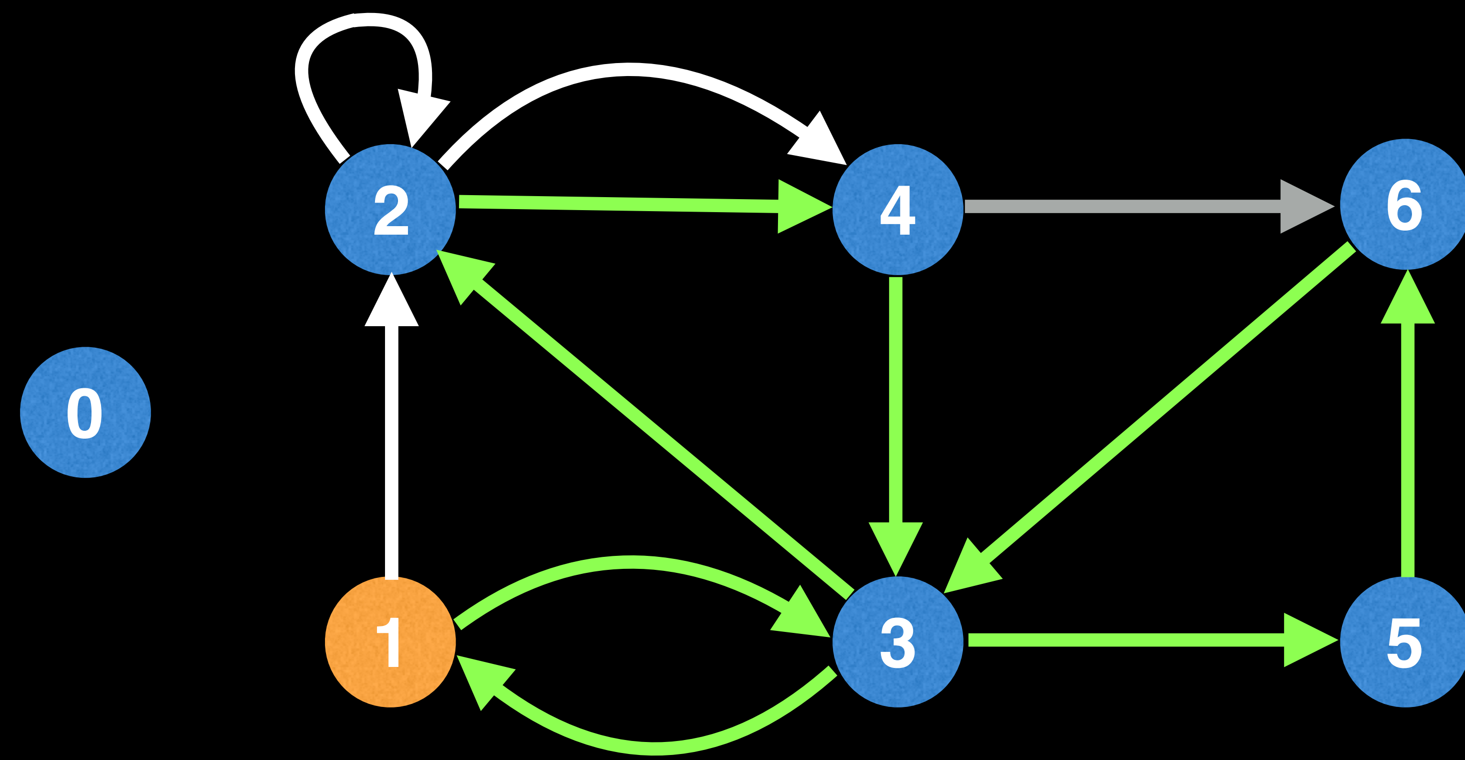
Node	Out
0	0
1	1
2	2
3	1
4	0
5	0
6	0



Solution = [6]

Finding an Eulerian path (directed graph)

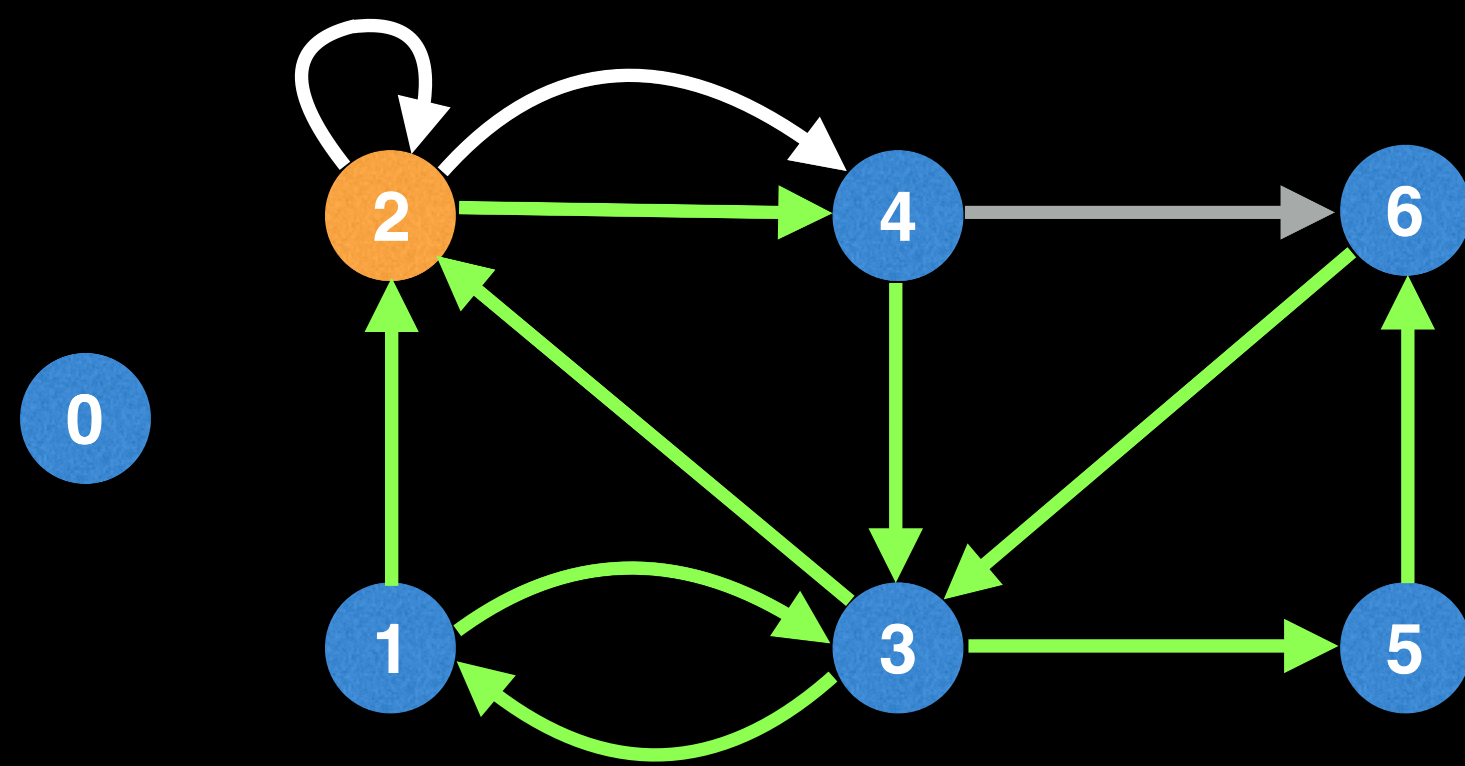
Node	Out
0	0
1	1
2	2
3	0
4	0
5	0
6	0



Solution = [6]

Finding an Eulerian path (directed graph)

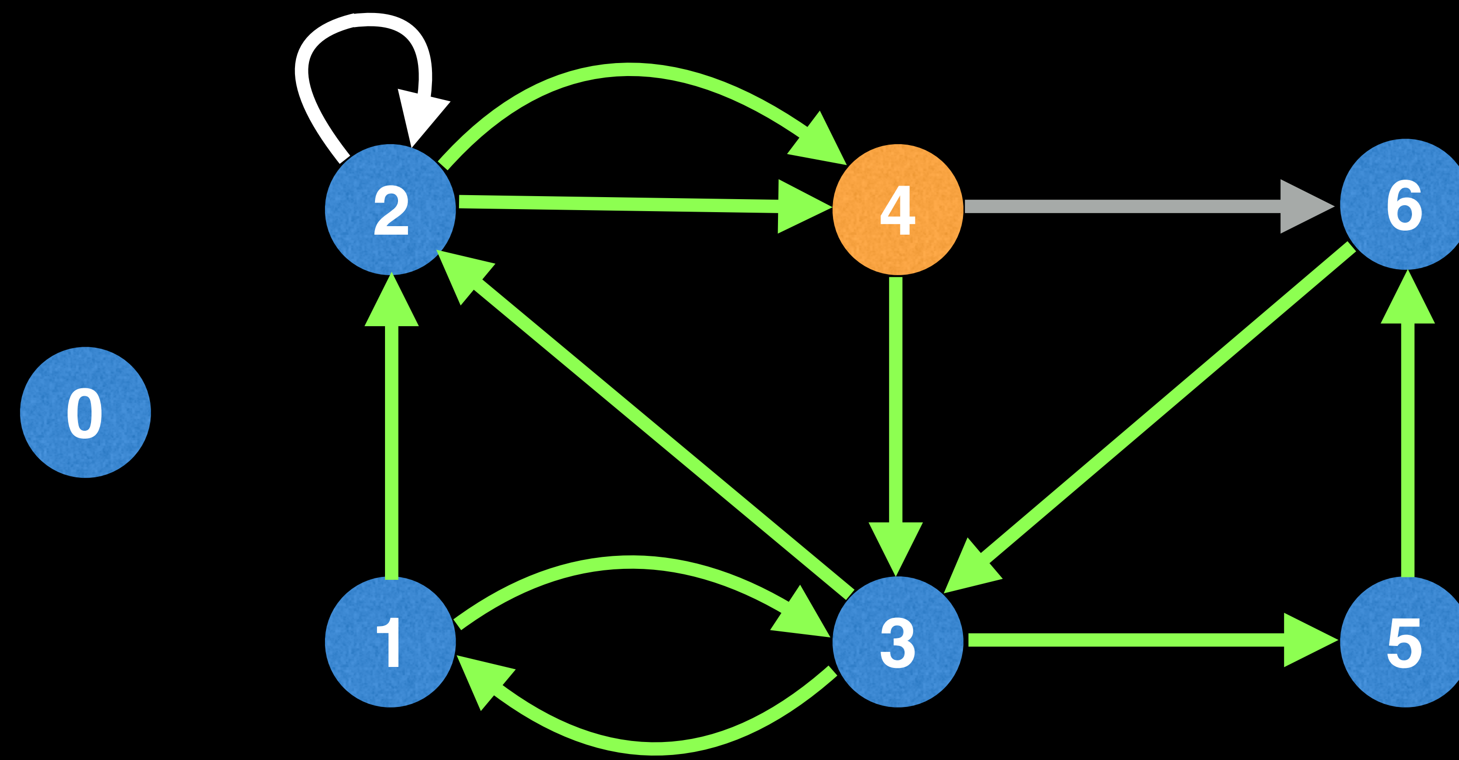
Node	Out
0	0
1	0
2	2
3	0
4	0
5	0
6	0



Solution = [6]

Finding an Eulerian path (directed graph)

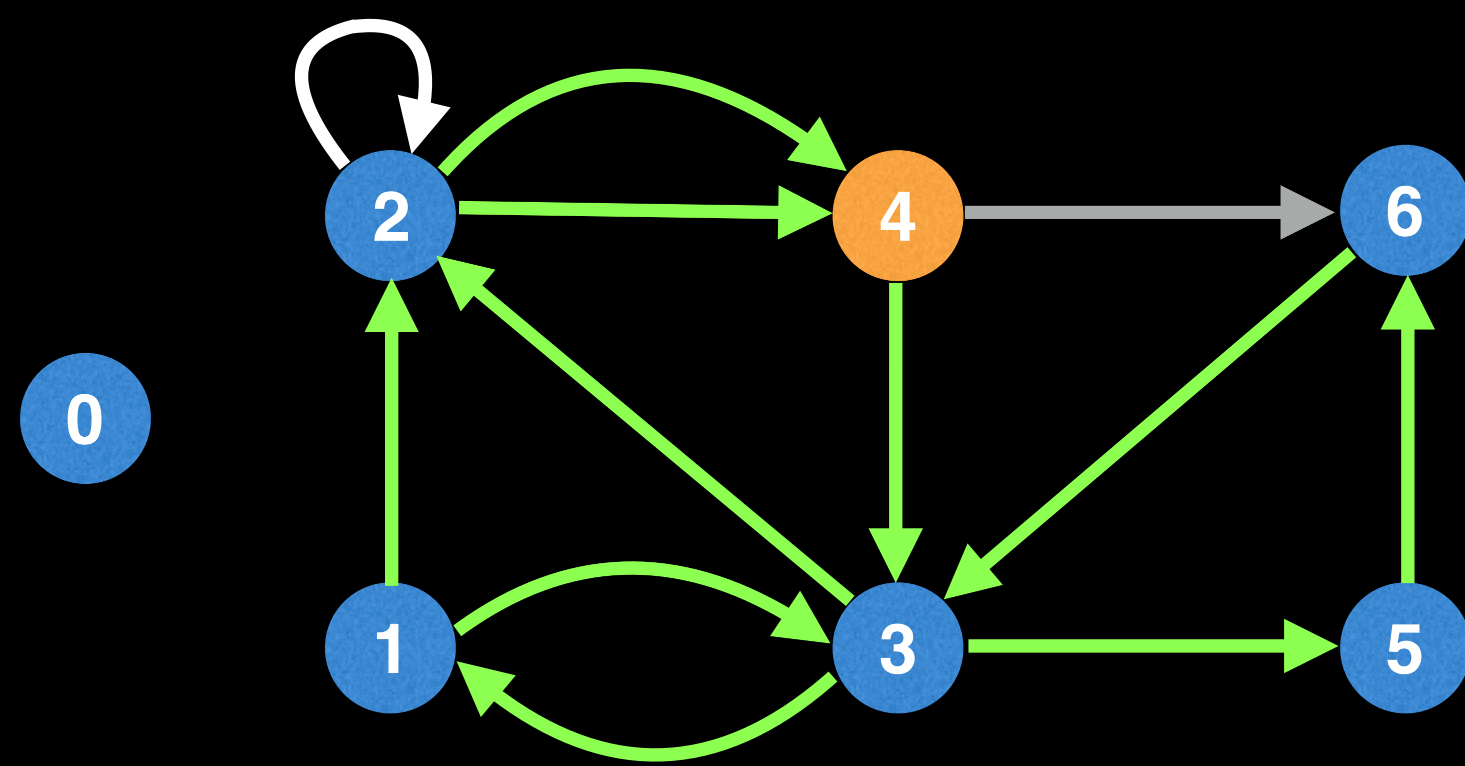
Node	Out
0	0
1	0
2	1
3	0
4	0
5	0
6	0



Solution = [6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	1
3	0
4	0
5	0
6	0

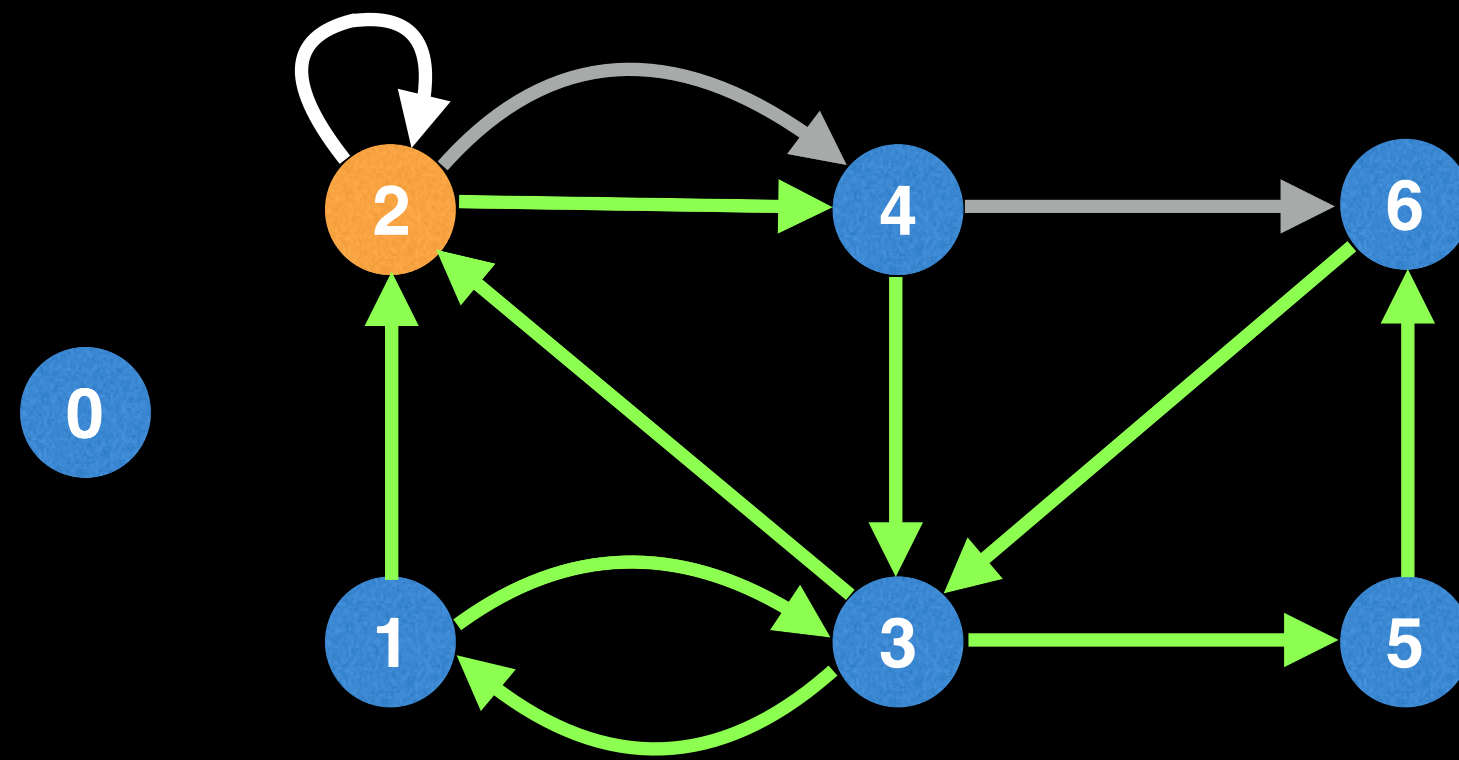


When the DFS is stuck, meaning there are no more outgoing edges (i.e out[i] = 0), then we know to backtrack and add the current node to the solution.

Solution = [6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	1
3	0
4	0
5	0
6	0

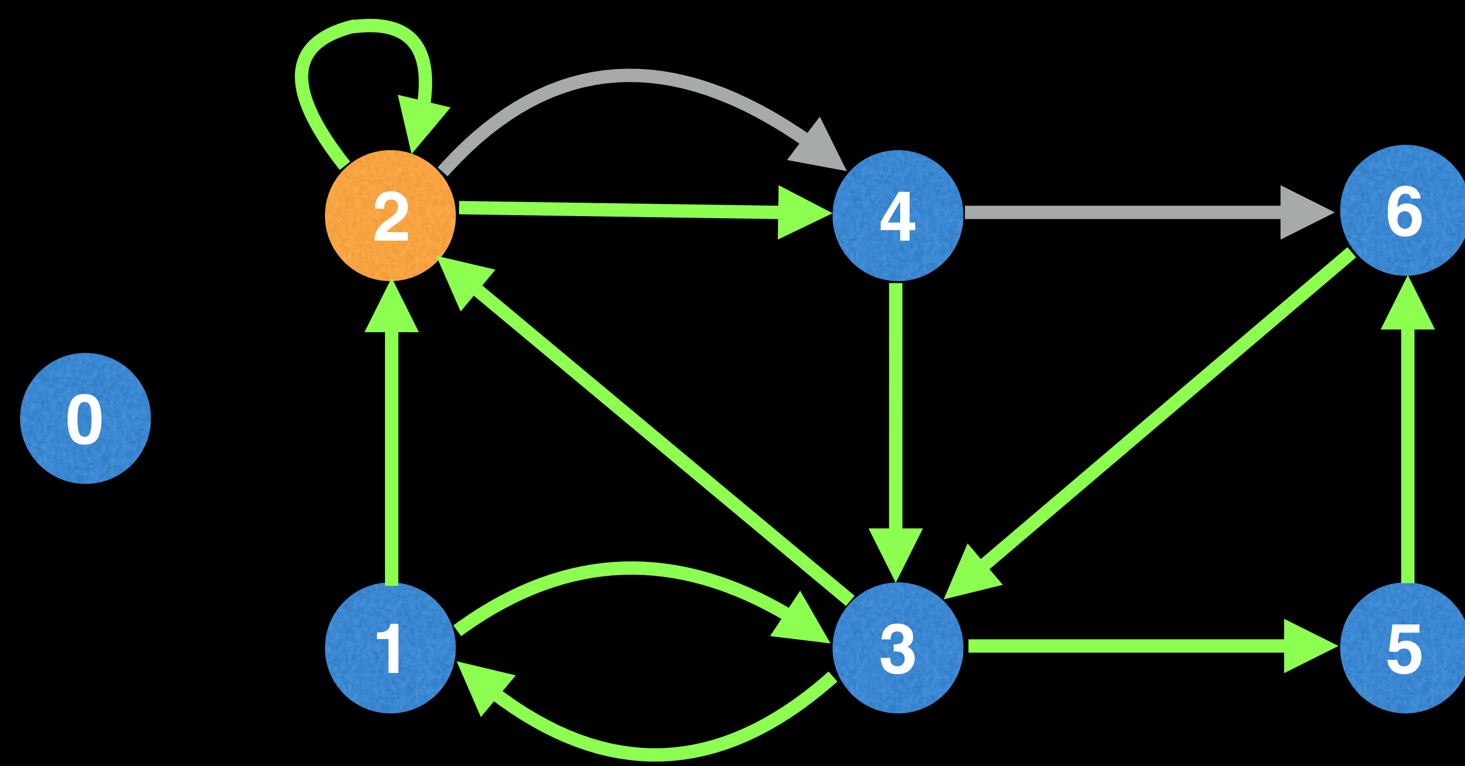


Node 2 still has an unvisited edge (since out[i] != 0) so we need to follow that edge.

Solution = [4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

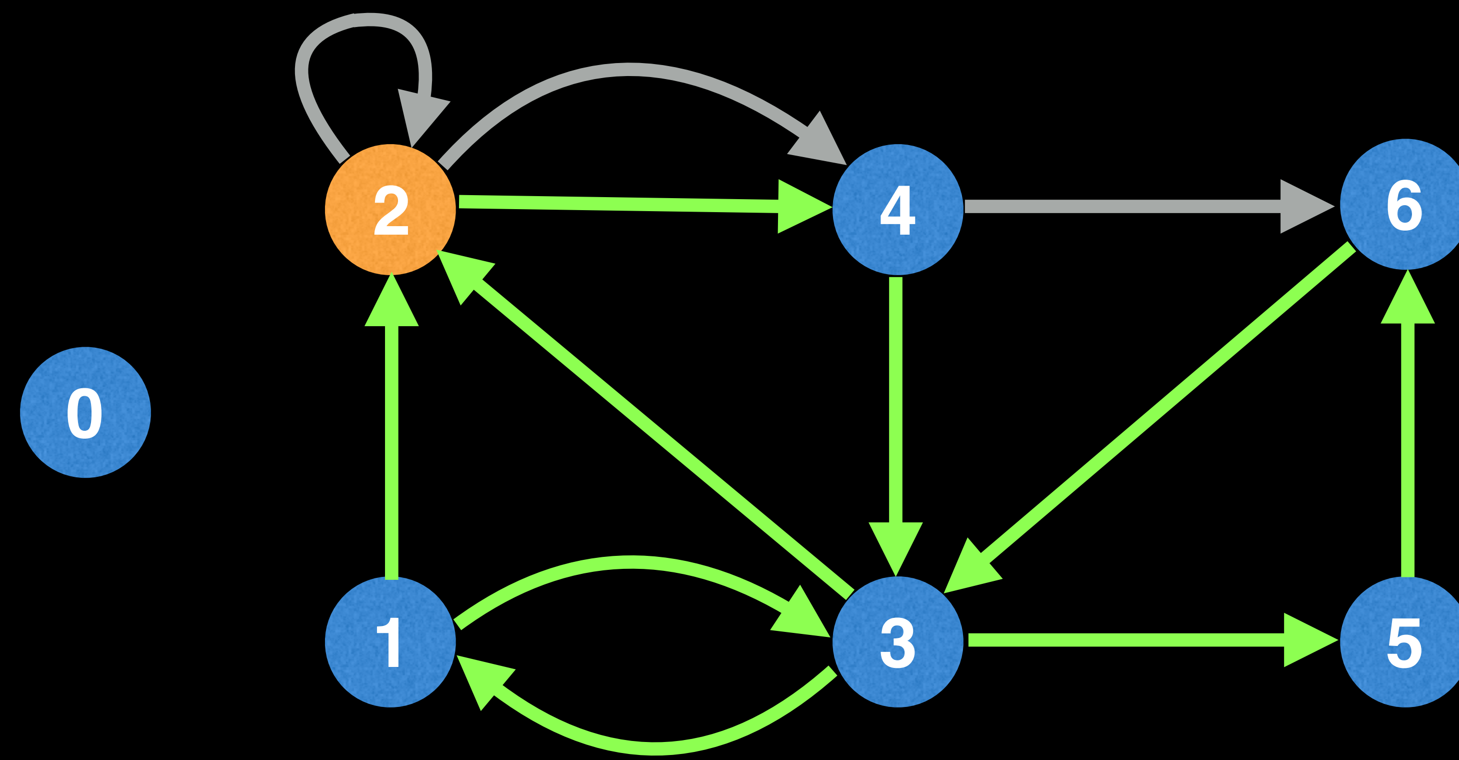


When the DFS is stuck, meaning there are no more outgoing edges (i.e out[i] = 0), then we know to backtrack and add the current node to the solution.

Solution = [4,6]

Finding an Eulerian path (directed graph)

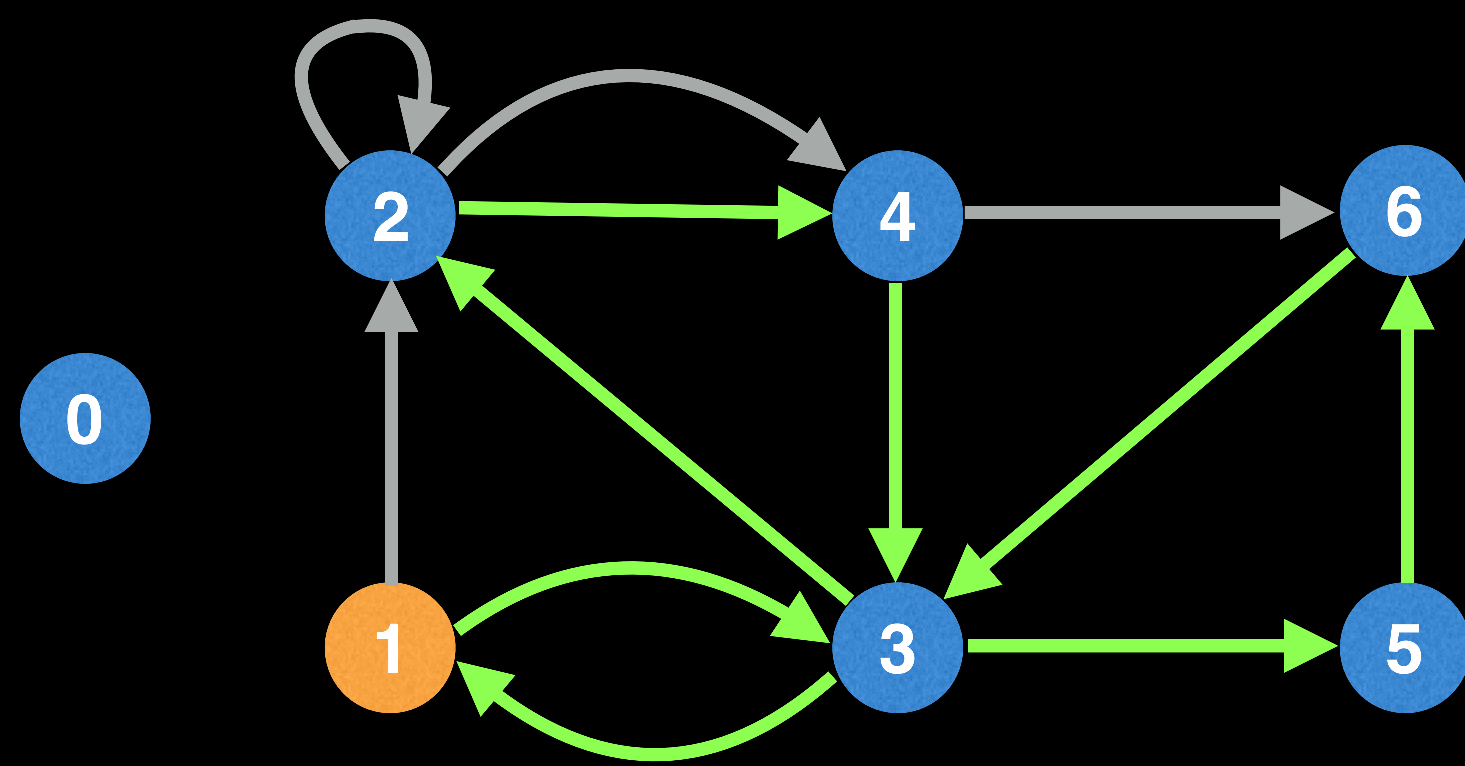
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [2,4,6]

Finding an Eulerian path (directed graph)

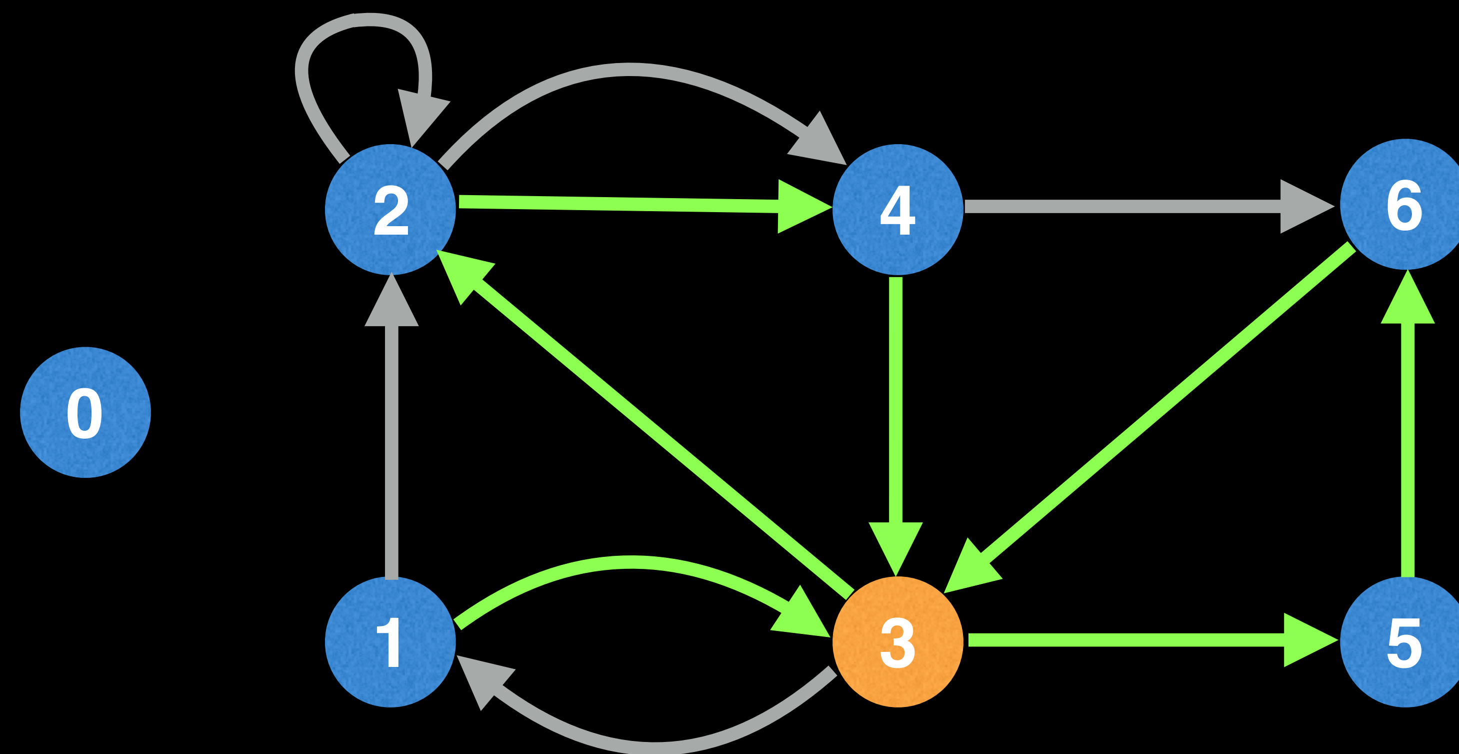
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [2,2,4,6]

Finding an Eulerian path (directed graph)

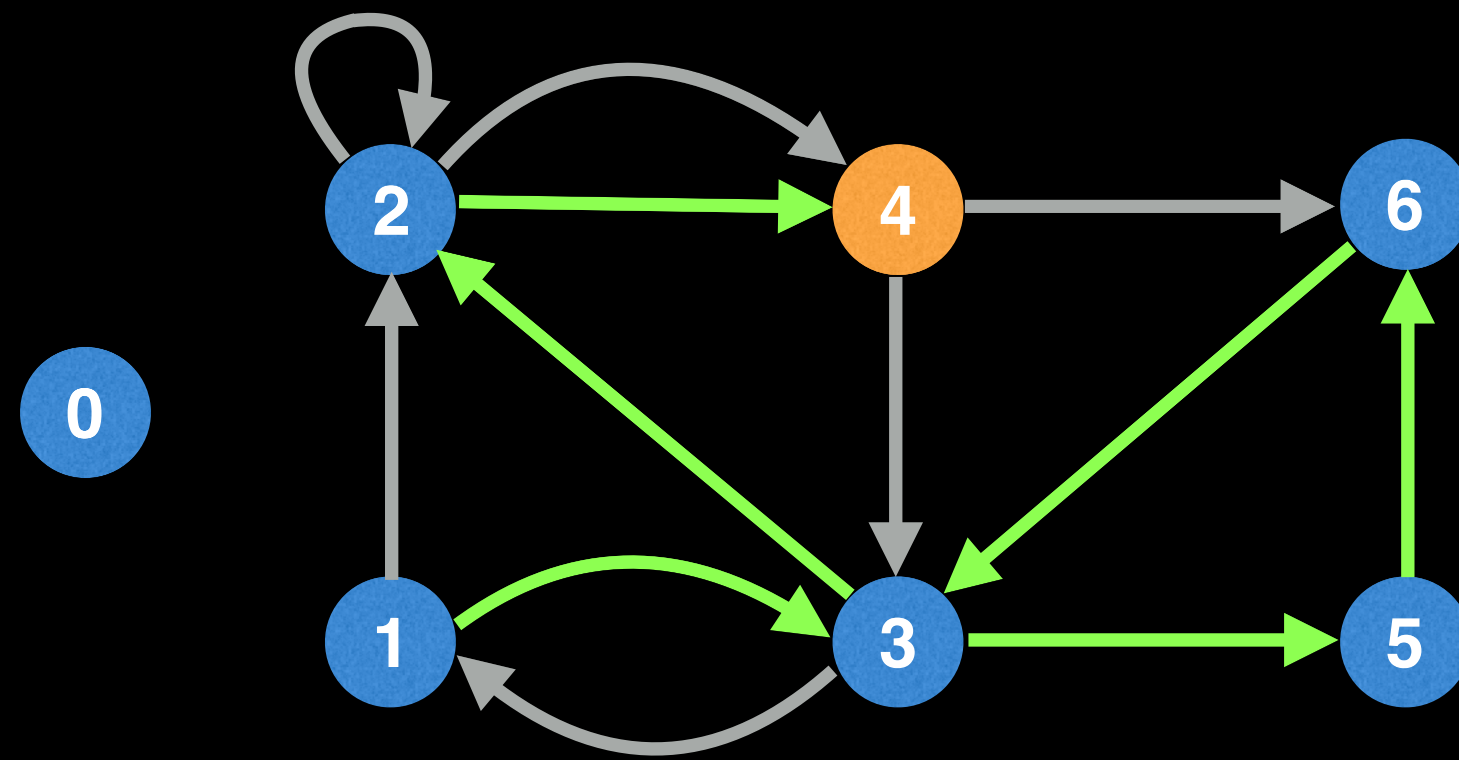
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [1,2,2,4,6]

Finding an Eulerian path (directed graph)

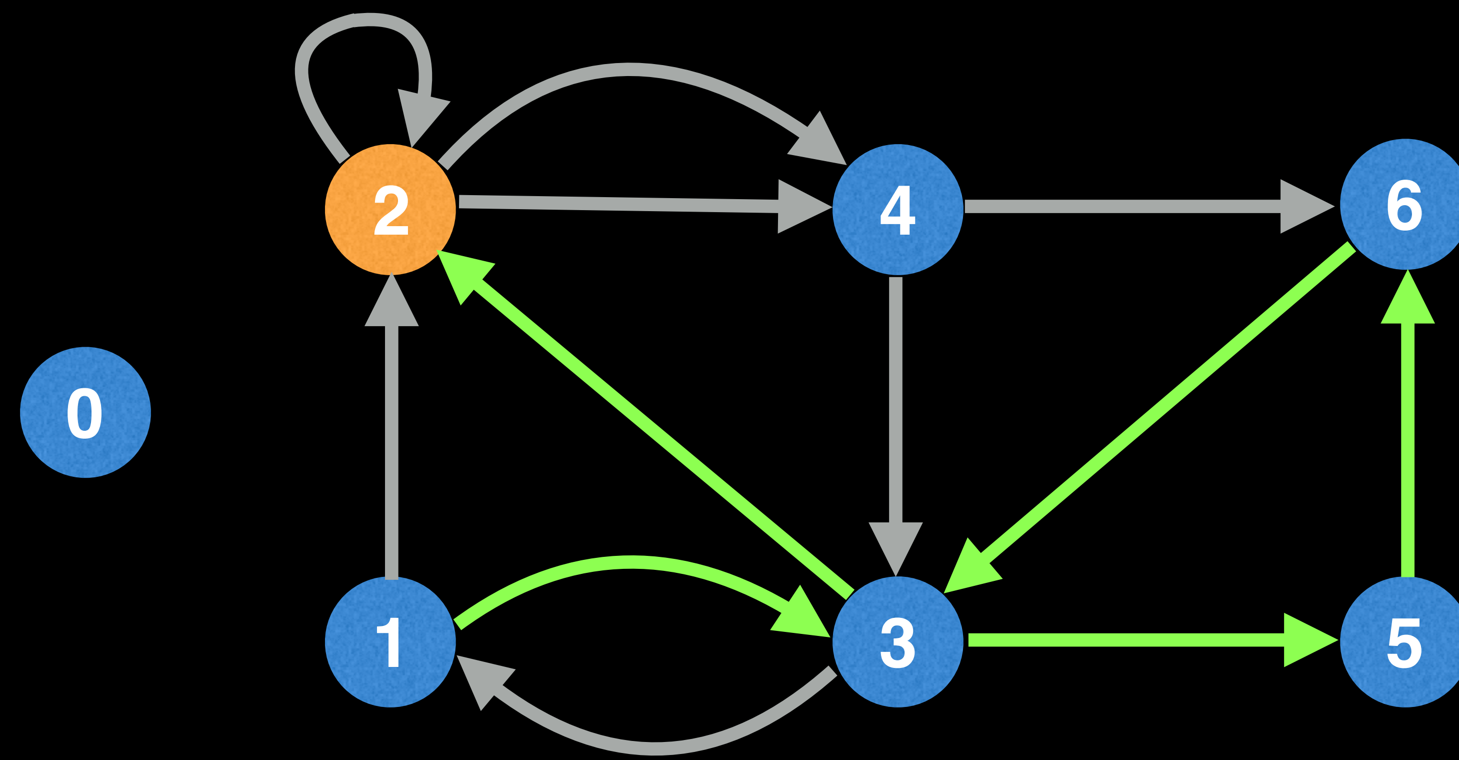
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

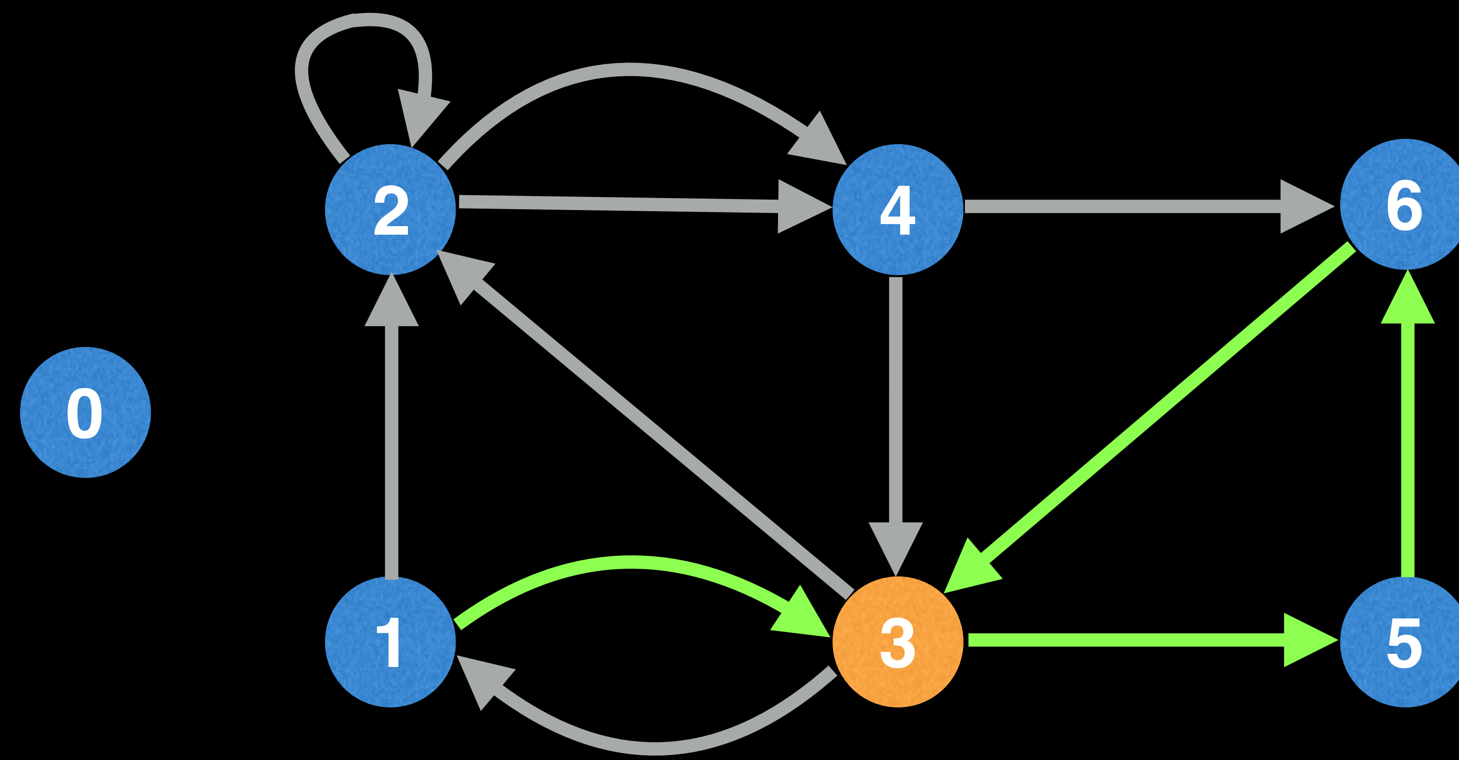
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

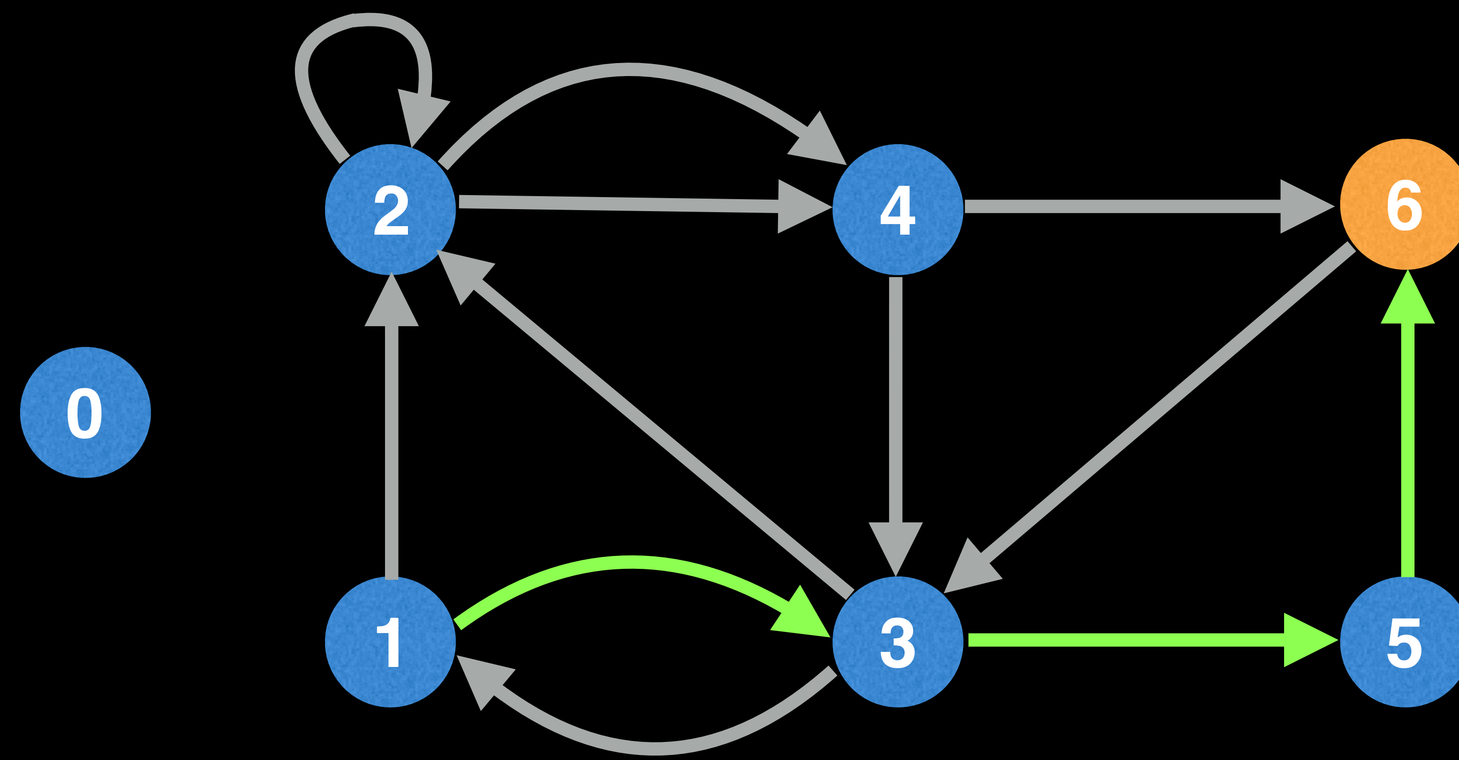
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

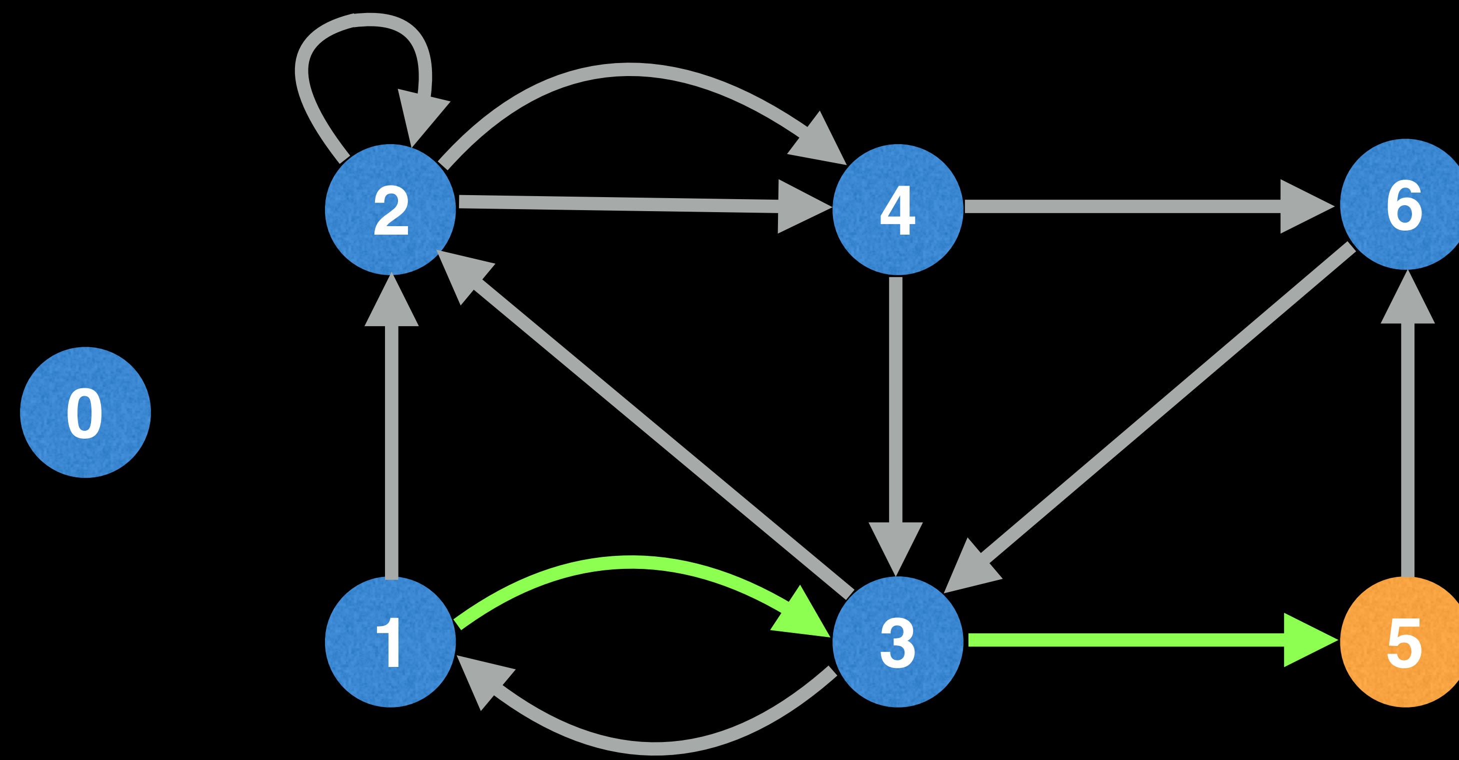
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

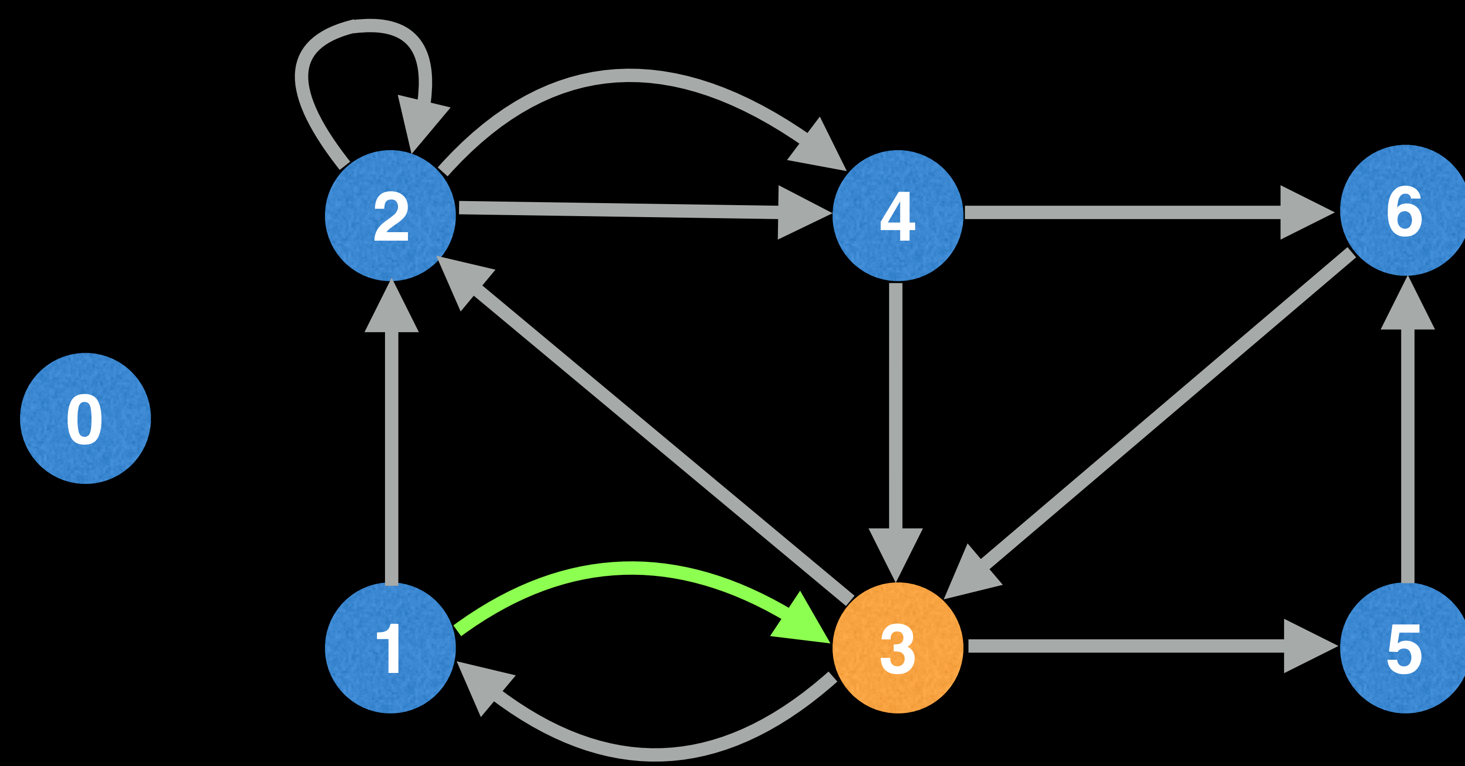
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [6,3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

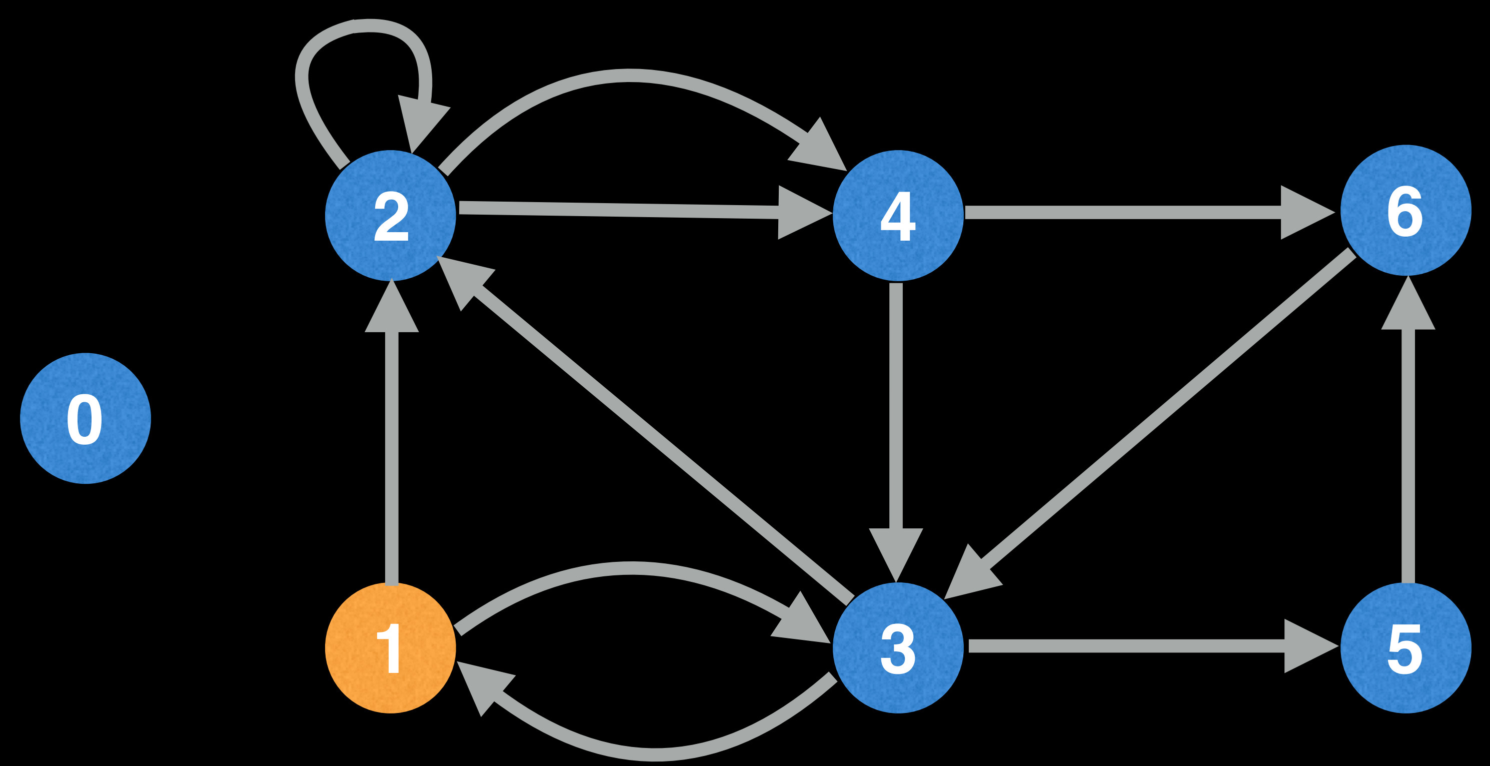
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [5,6,3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

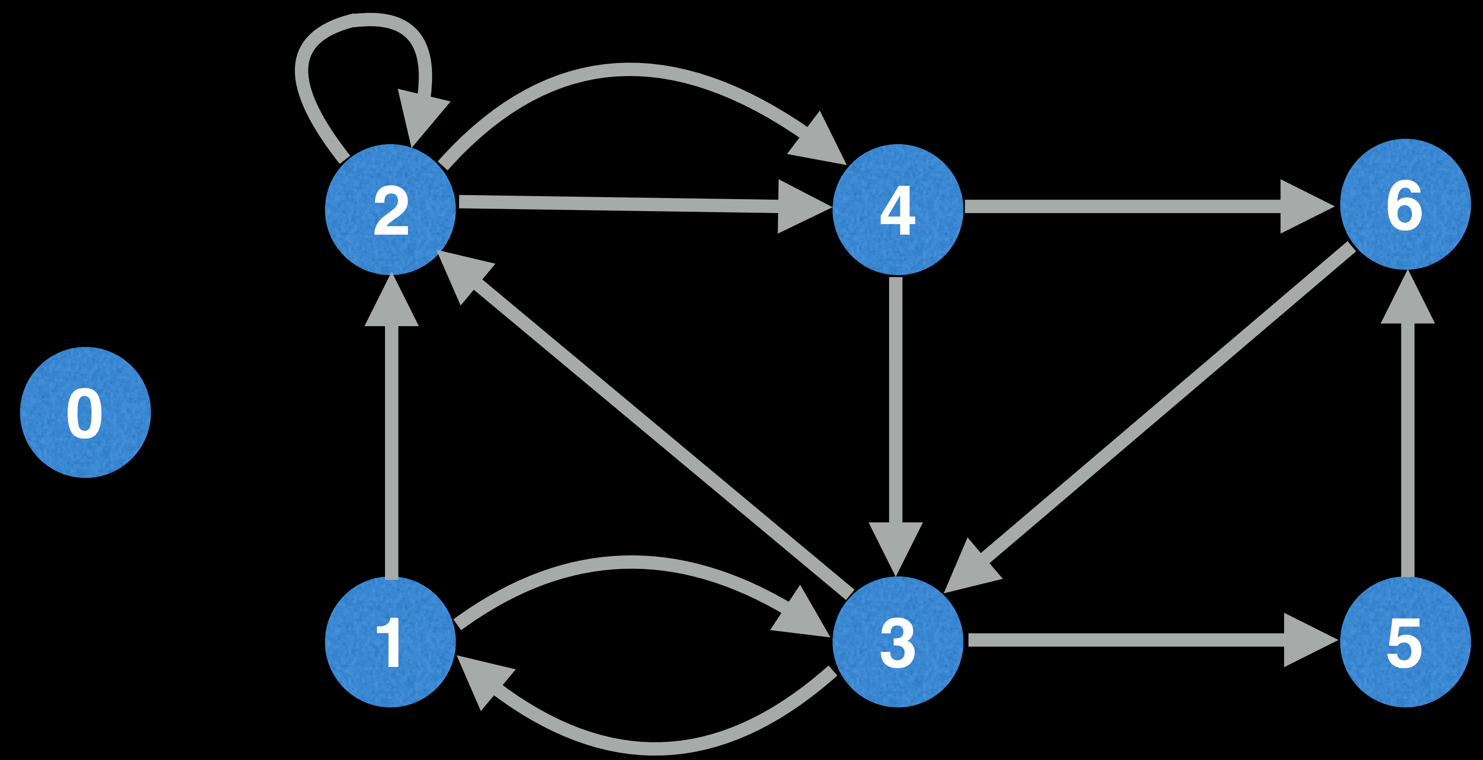
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [3,5,6,3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

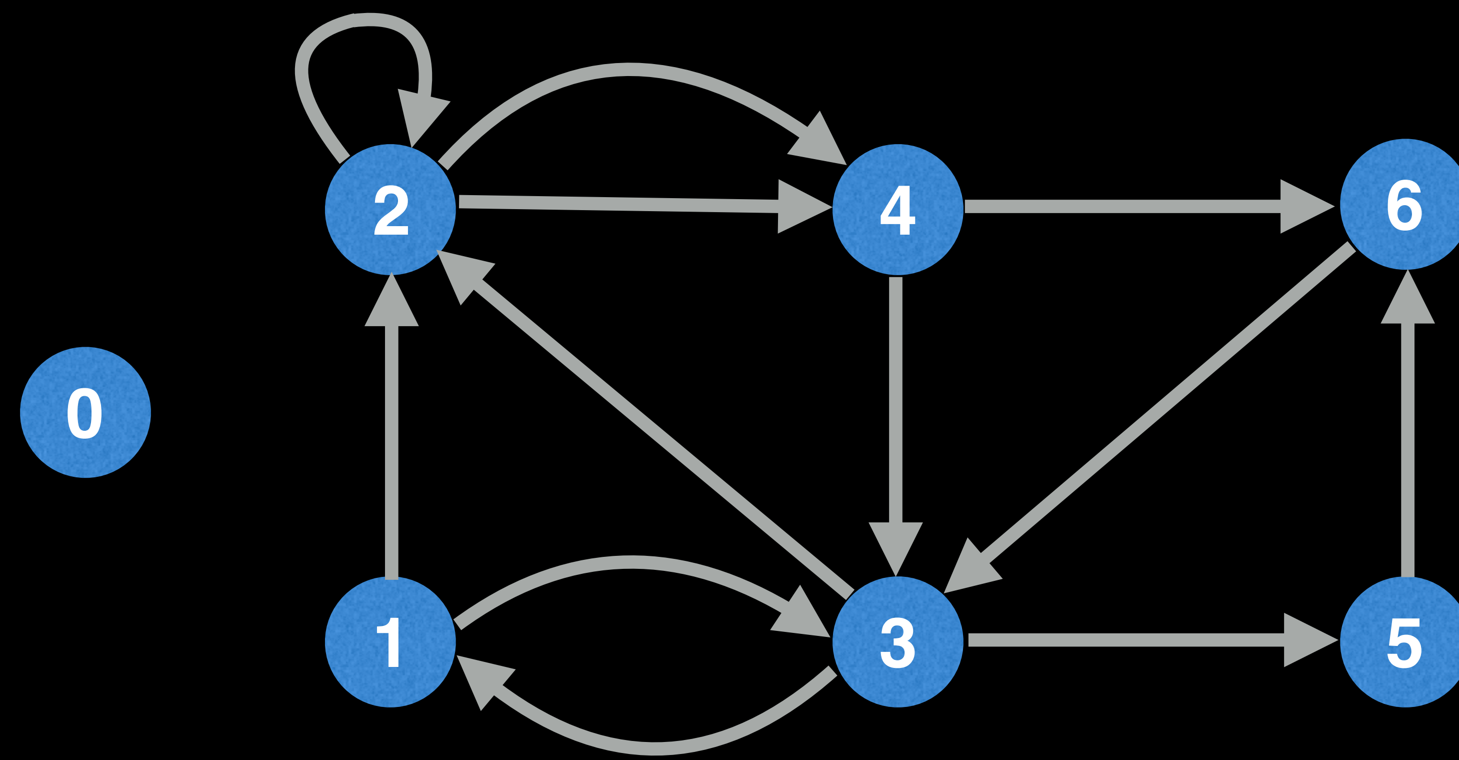
Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



Solution = [1,3,5,6,3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0



The time complexity to find an eulerian path with this algorithm is **O(E)**. The two calculations we're doing: computing in/out degrees + DFS are both linear in the number of edges.

Solution = [1,3,5,6,3,2,4,3,1,2,2,4,6]

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it’s impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it's impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it’s impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it’s impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it’s impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it’s impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
  for edges in g:
    for edge in edges:
      out[edge.from]++
      in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it’s impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it’s impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

Avoids starting DFS at a singleton

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

The out array is currently serving two purposes. One purpose is to track whether or not there are still outgoing edges, and the other is to index into the adjacency list to select the next outgoing edge.

This assumes the adjacency list stores edges in a data structure that is indexable in **O(1)** (e.g stored in an array). If not (e.g a linked-list/stack/etc…), you can use an iterator to iterate over the edges.

```
function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it’s impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

  countInOutDegrees()
  if not graphHasEulerianPath(): return null

  dfs(findStartNode())

  # Return eulerian path if we traversed all the
  # edges. The graph might be disconnected, in which
  # case it's impossible to have an euler path.
  if path.size() == m+1: return path
  return null
```