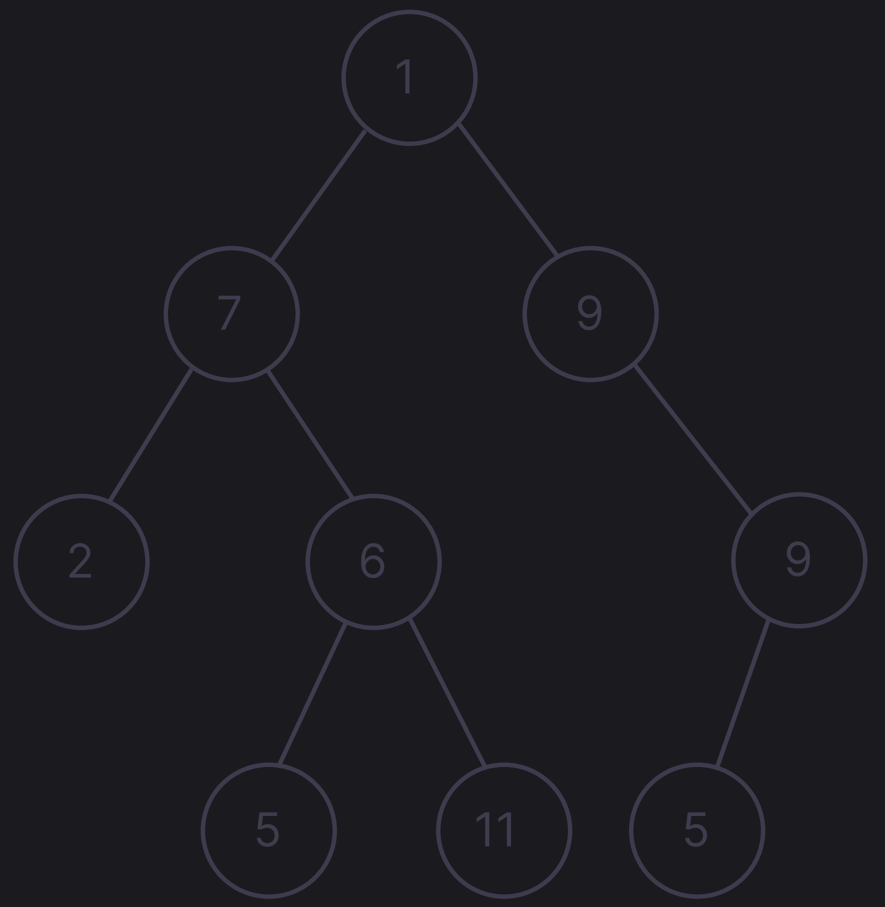
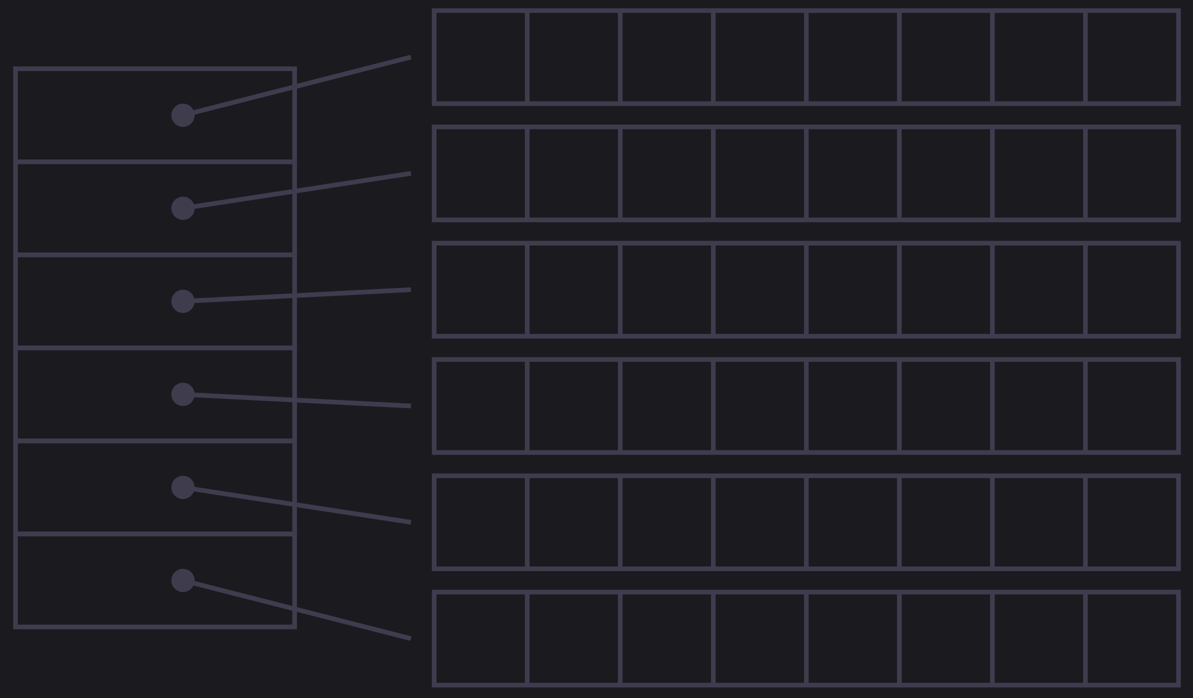
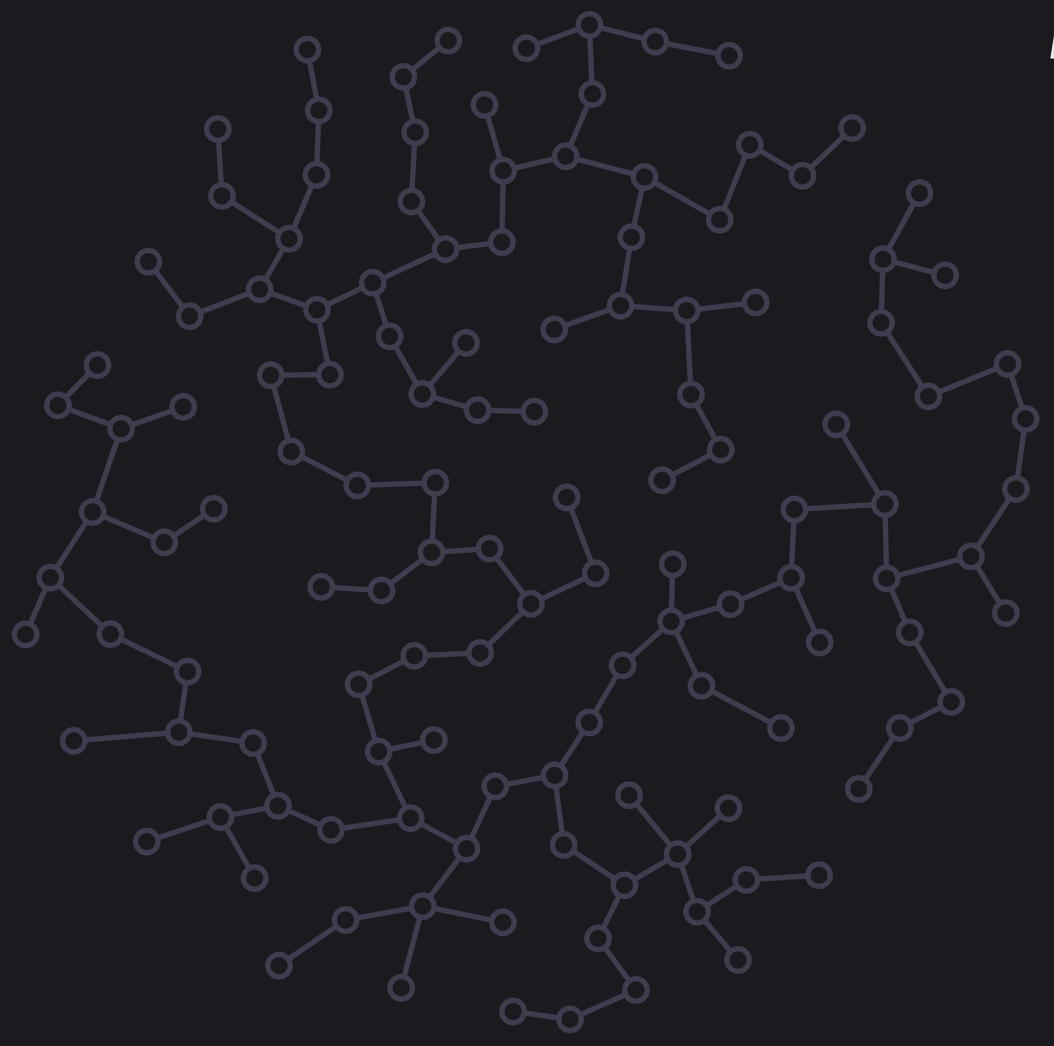


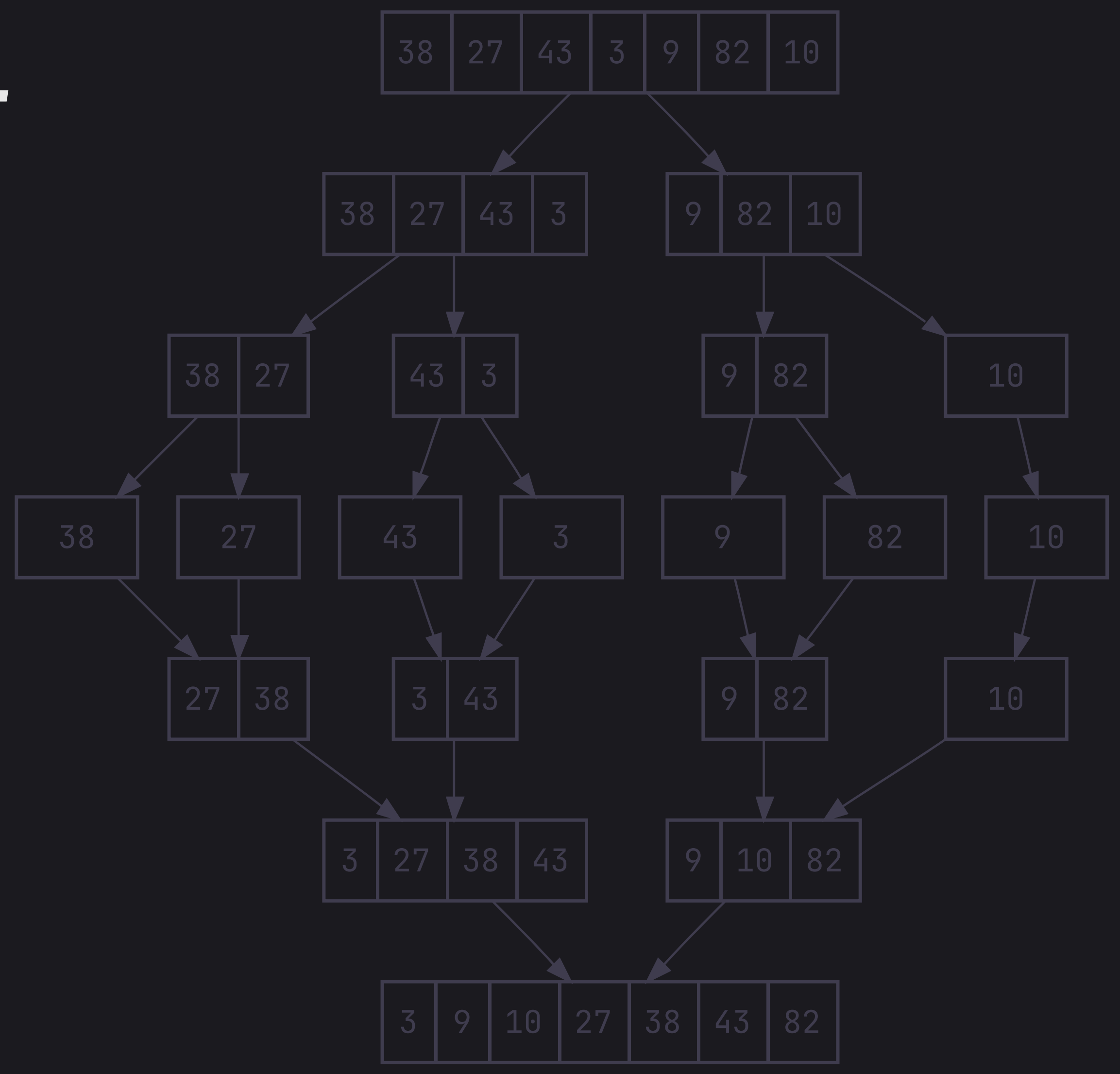


data structures & algorithms

Tutorial 11

(Week 12)





Overview

- 🔥 Burning questions from last week?
- This week's Lab
  - Detecting Cycles
  - DAGs
  - Topological Sort

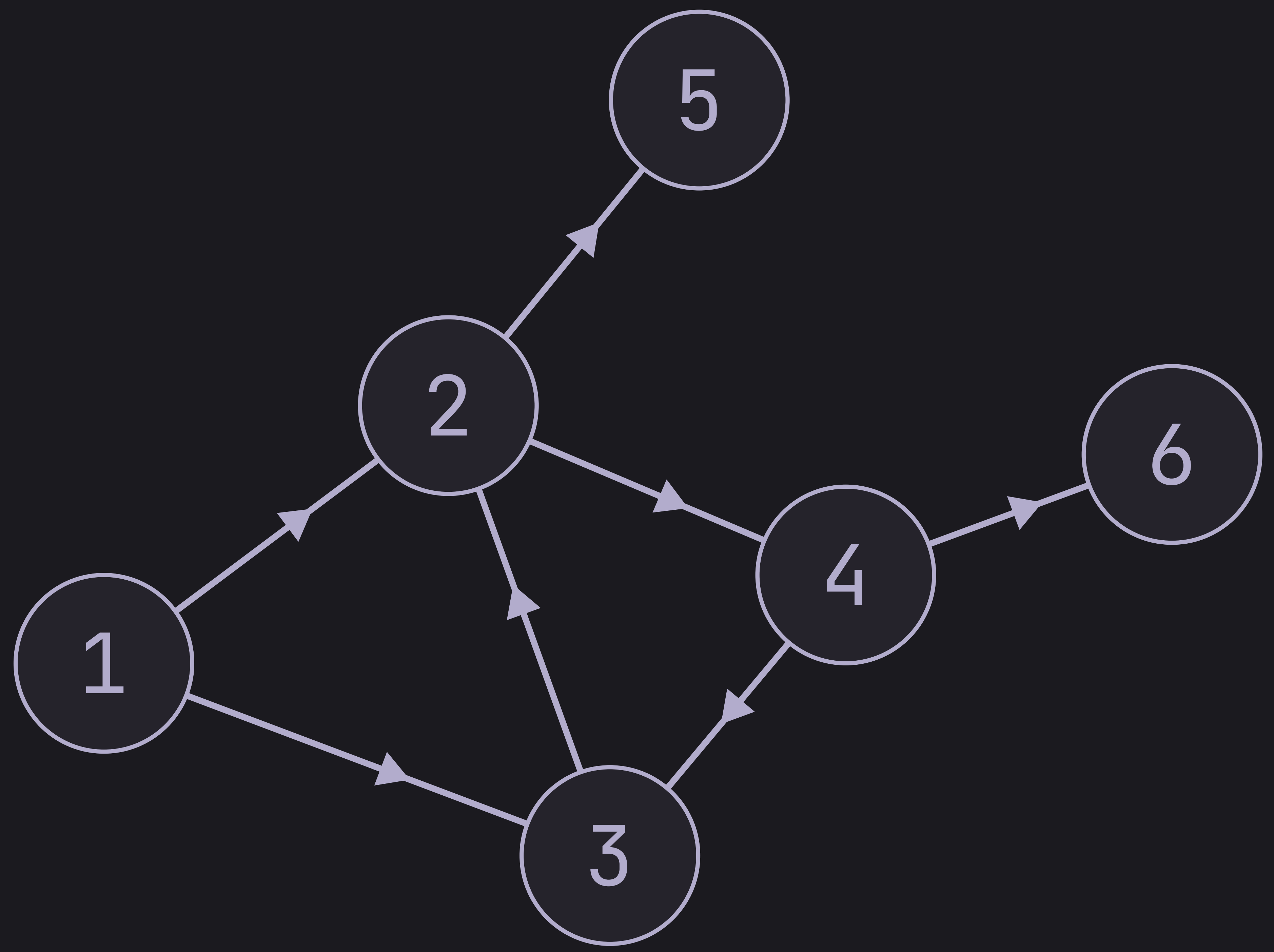
This week's Lab

Cycles

Today we are learning about some more useful types of graphs and graph algorithms.

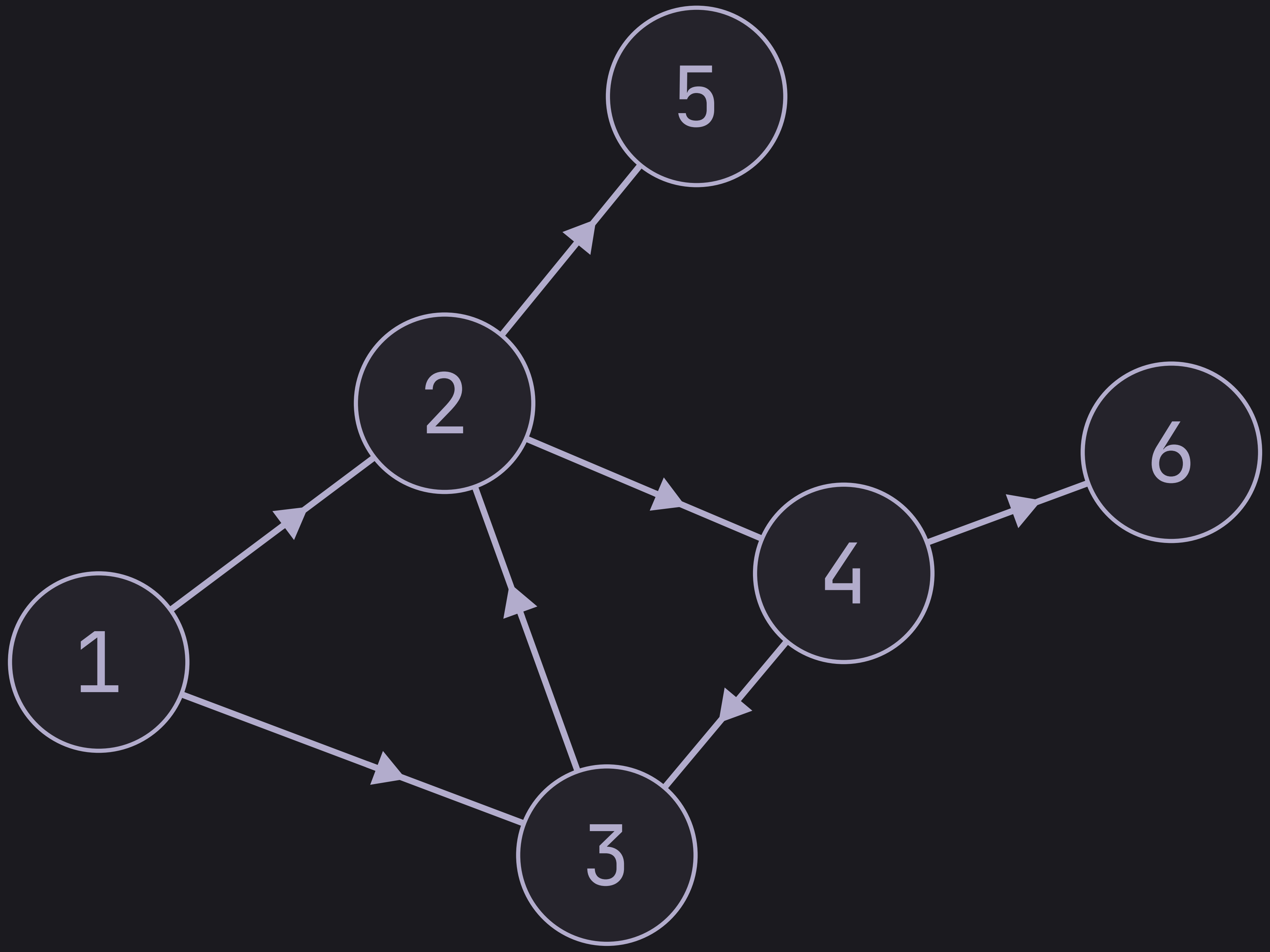
- Directed Acyclic Graph (DAG)
- Detecting Cycles
- Topological Sort

Detecting Cycles



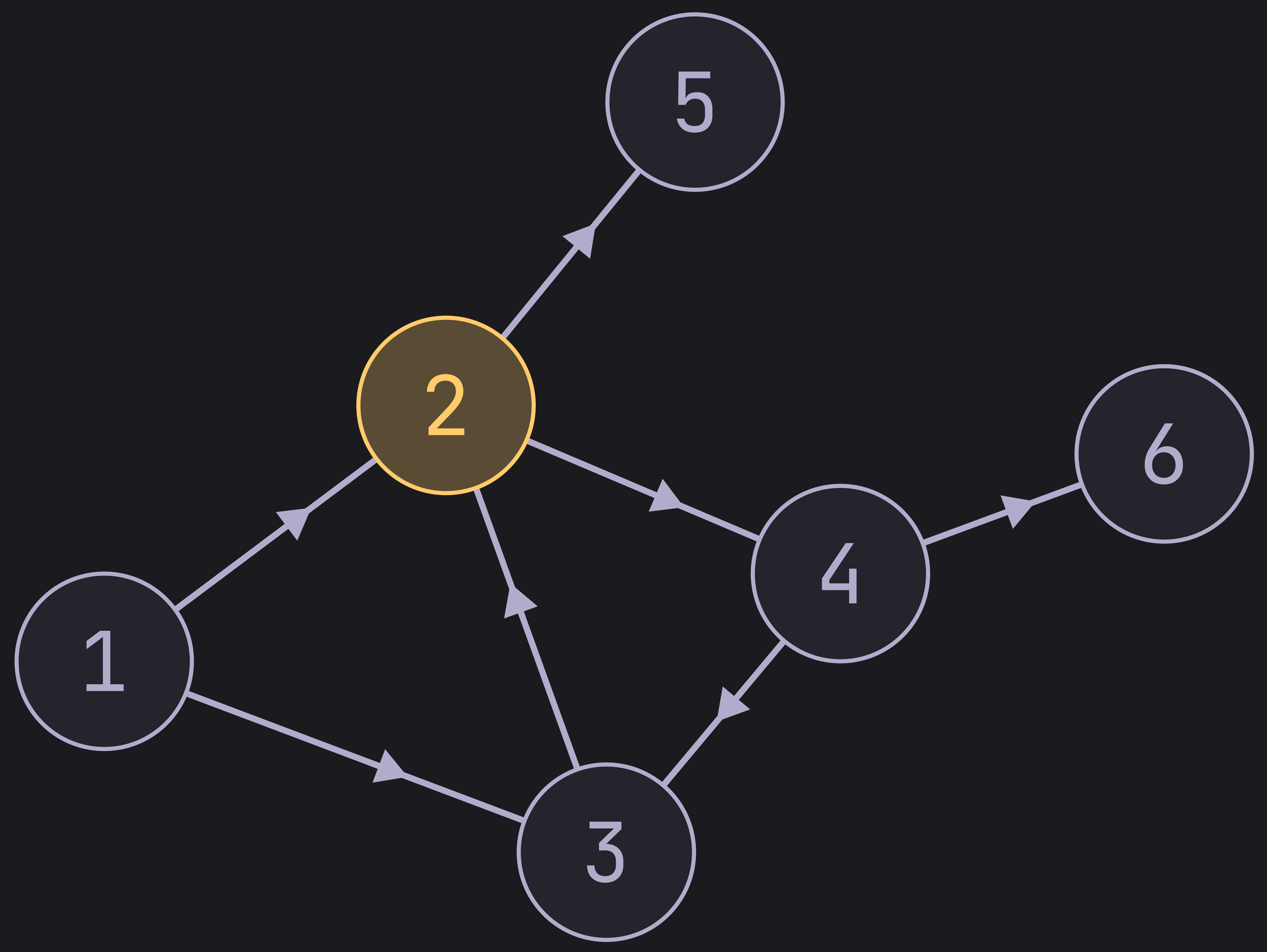
Here we have a **directed** graph

Detecting Cycles



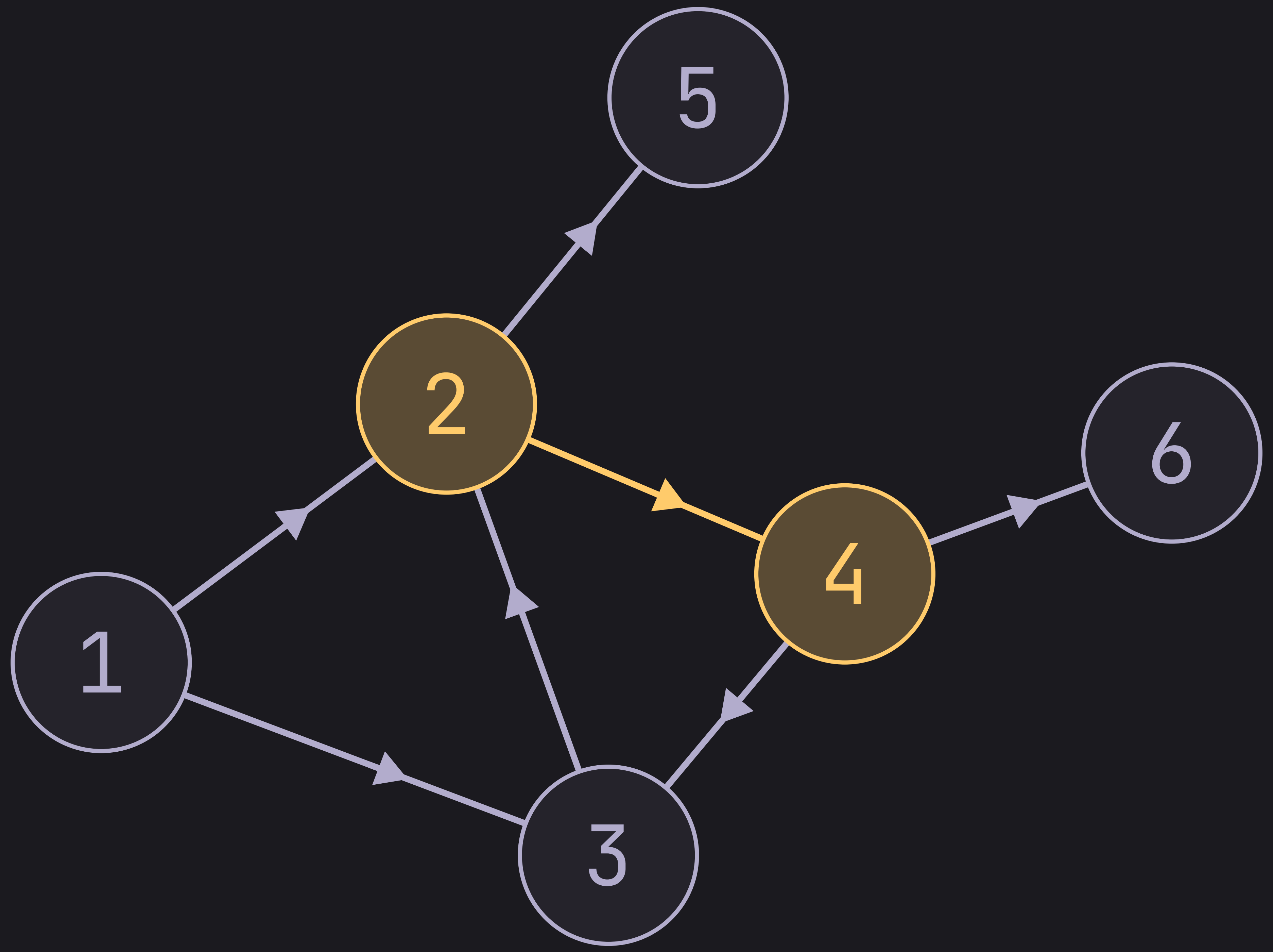
Can anyone see any **cycles** in this graph?

Detecting Cycles



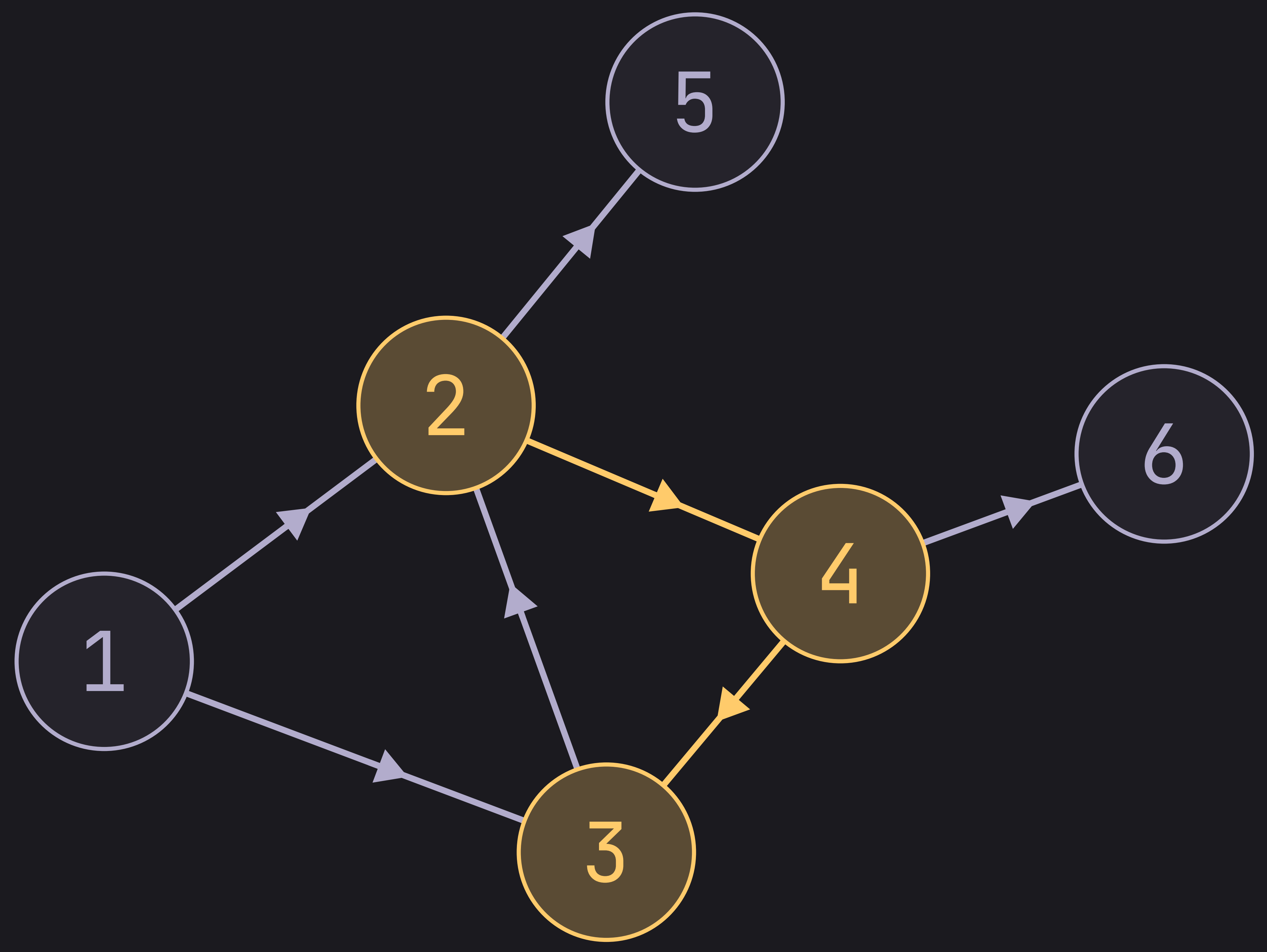
If we start at **vertex 2**, can we travel along the edges and get back to vertex 2?

Detecting Cycles



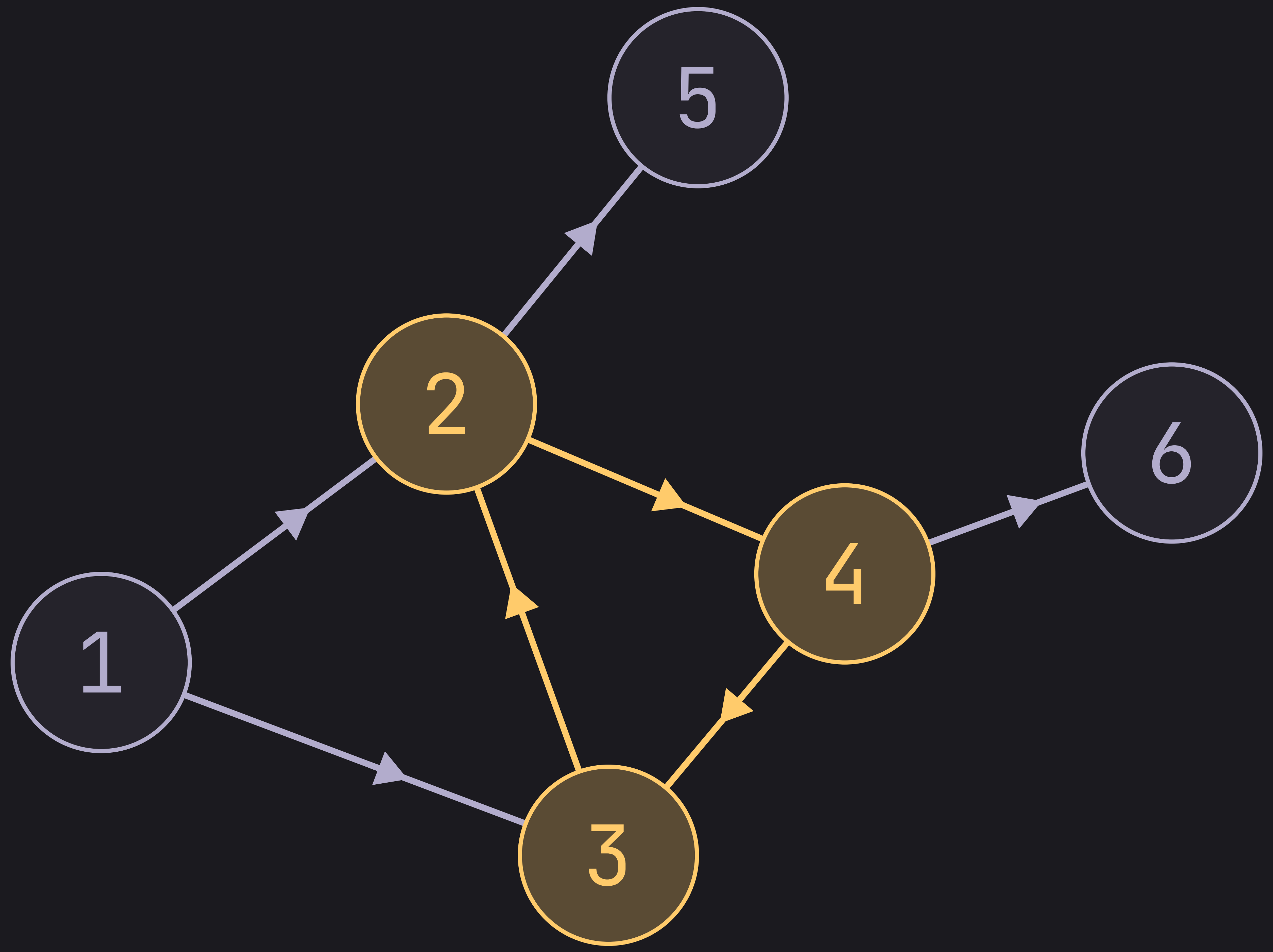
If we start at **vertex 2**, can we travel along the edges and get back to vertex 2?

Detecting Cycles



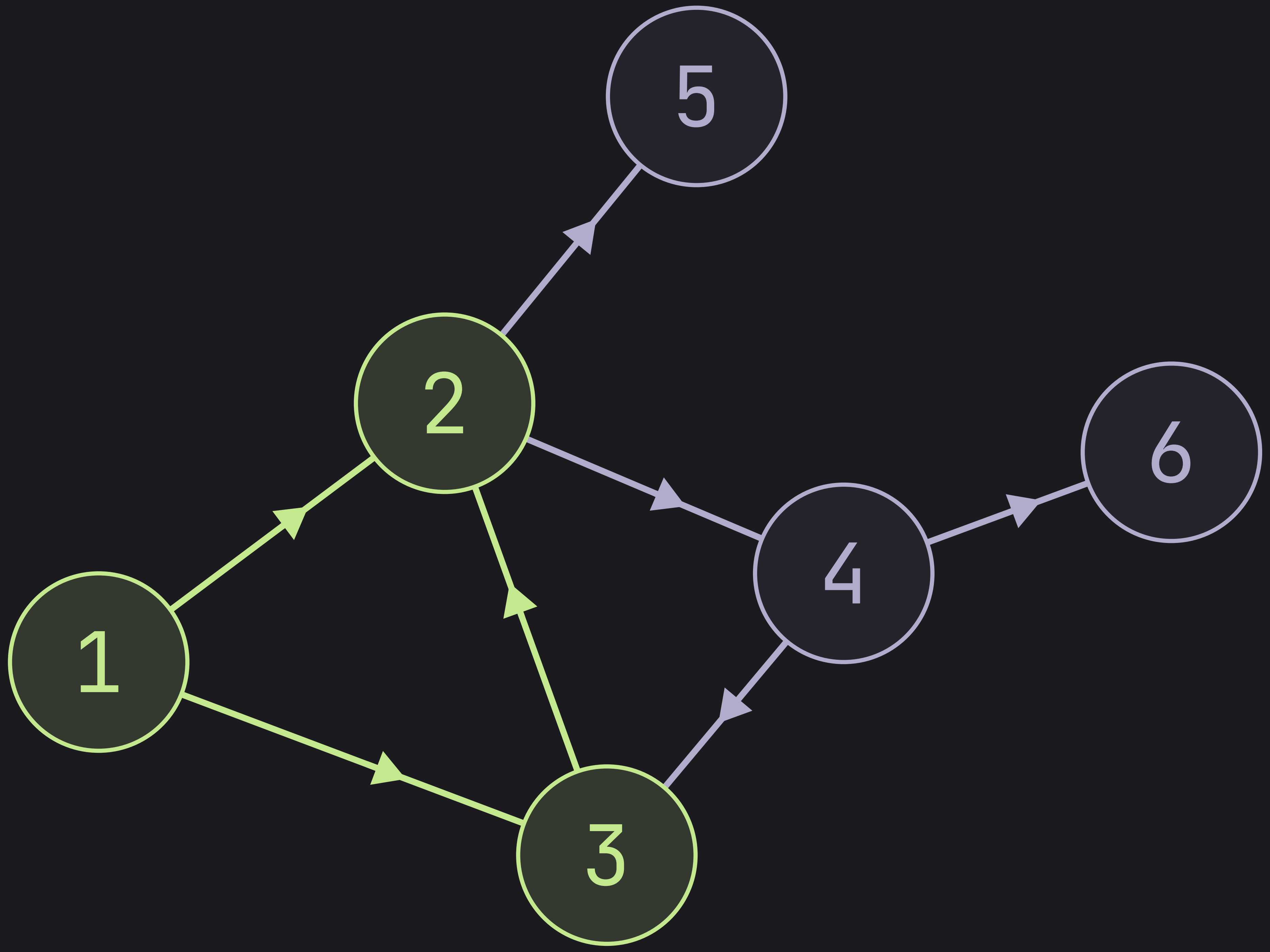
If we start at **vertex 2**, can we travel along the edges and get back to vertex 2?

Detecting Cycles



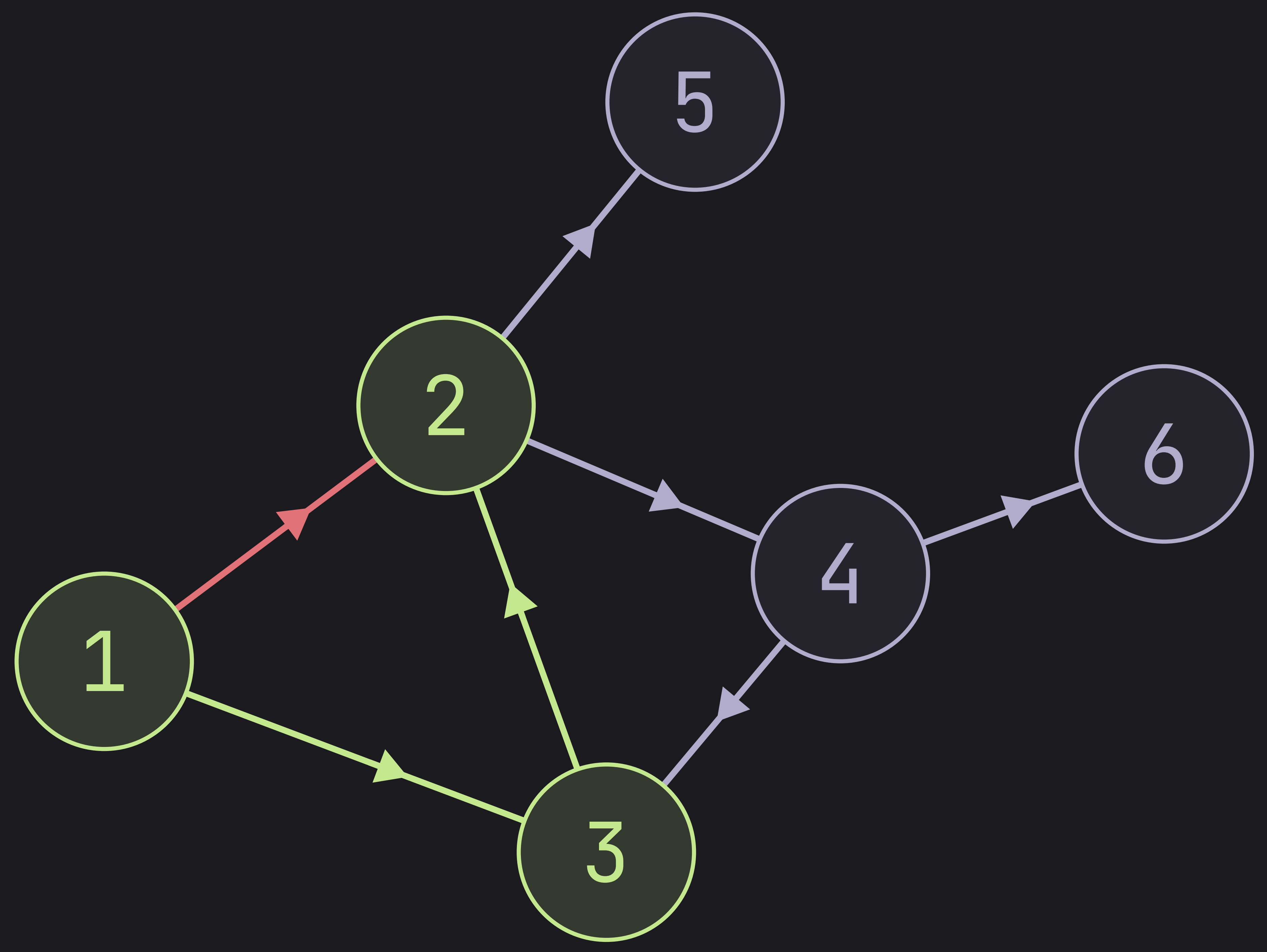
Yes! We have found a cycle

Detecting Cycles



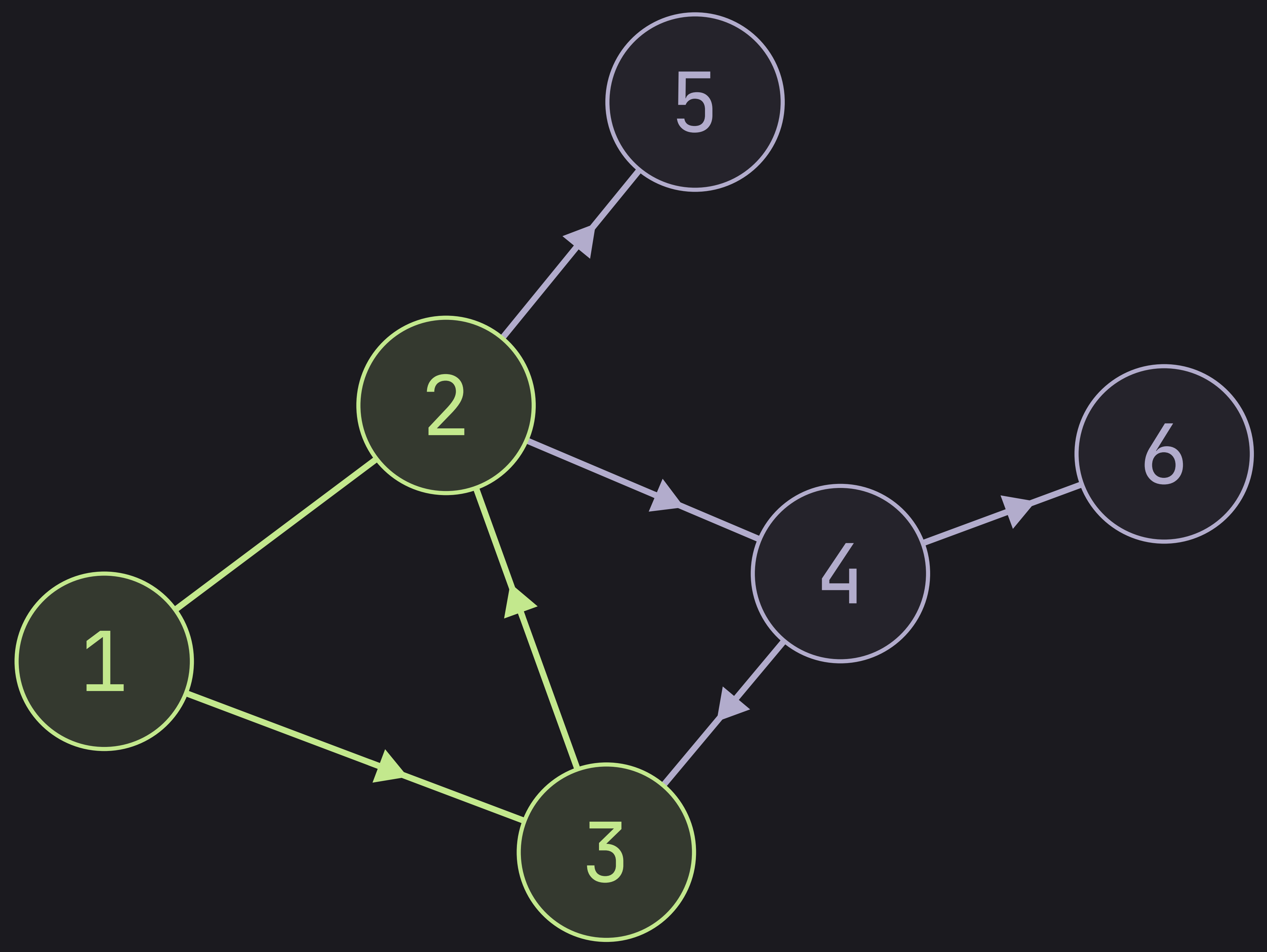
Does 1, 2, 3 form a cycle?

Detecting Cycles



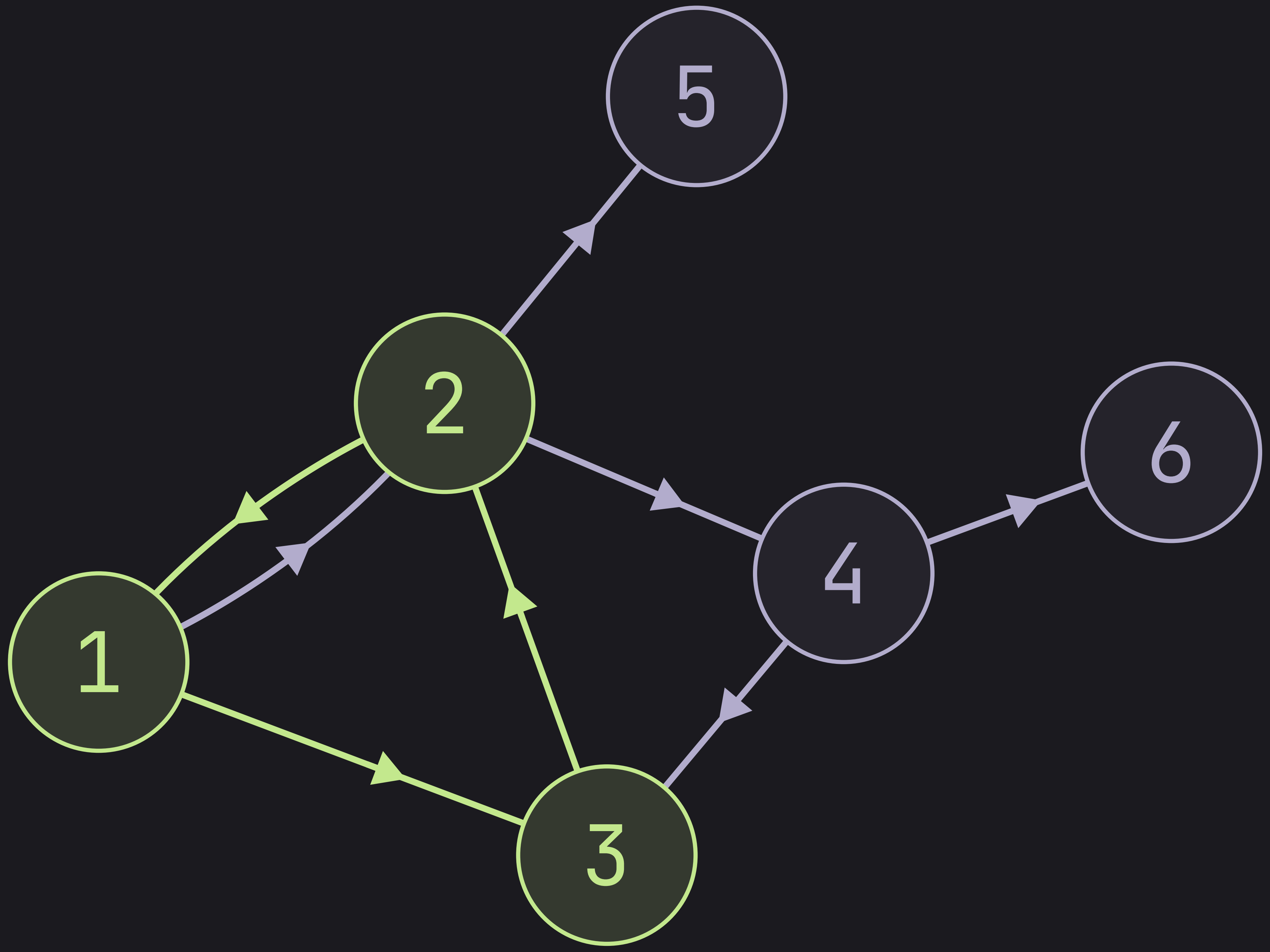
No! because we cannot travel from 2 $\longrightarrow$ 1. The edge is pointing the **wrong way**

Detecting Cycles



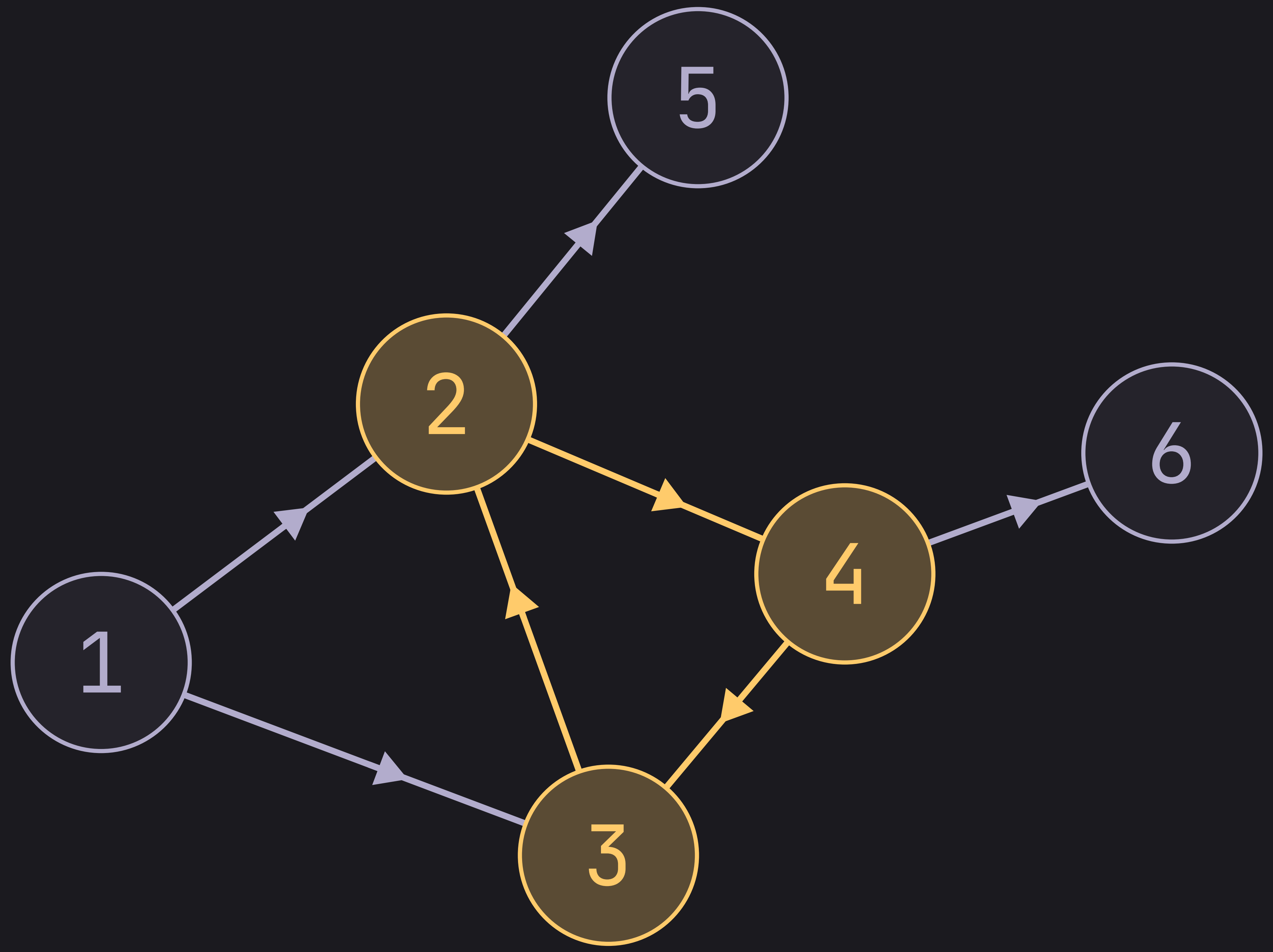
What about if the edge is **bidirectional**? Is this a cycle?

Detecting Cycles



Yes! because the edge is bidirectional so we can travel from 2 ⟶ 1, which completes the cycle

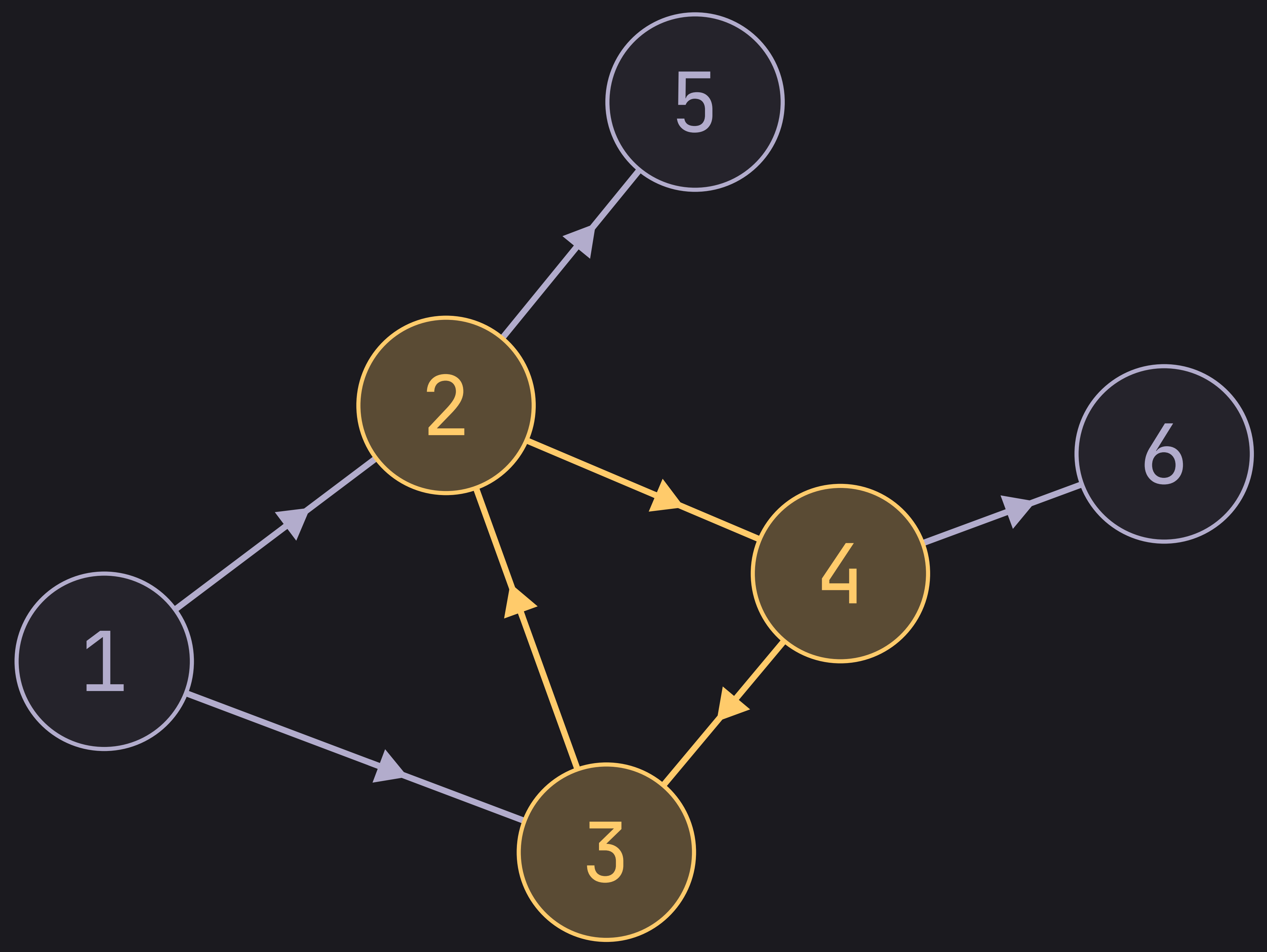
Detecting Cycles



Does this process of following edges deeper and deeper feel familiar?

What **algorithm** have we seen the past two weeks that does this?

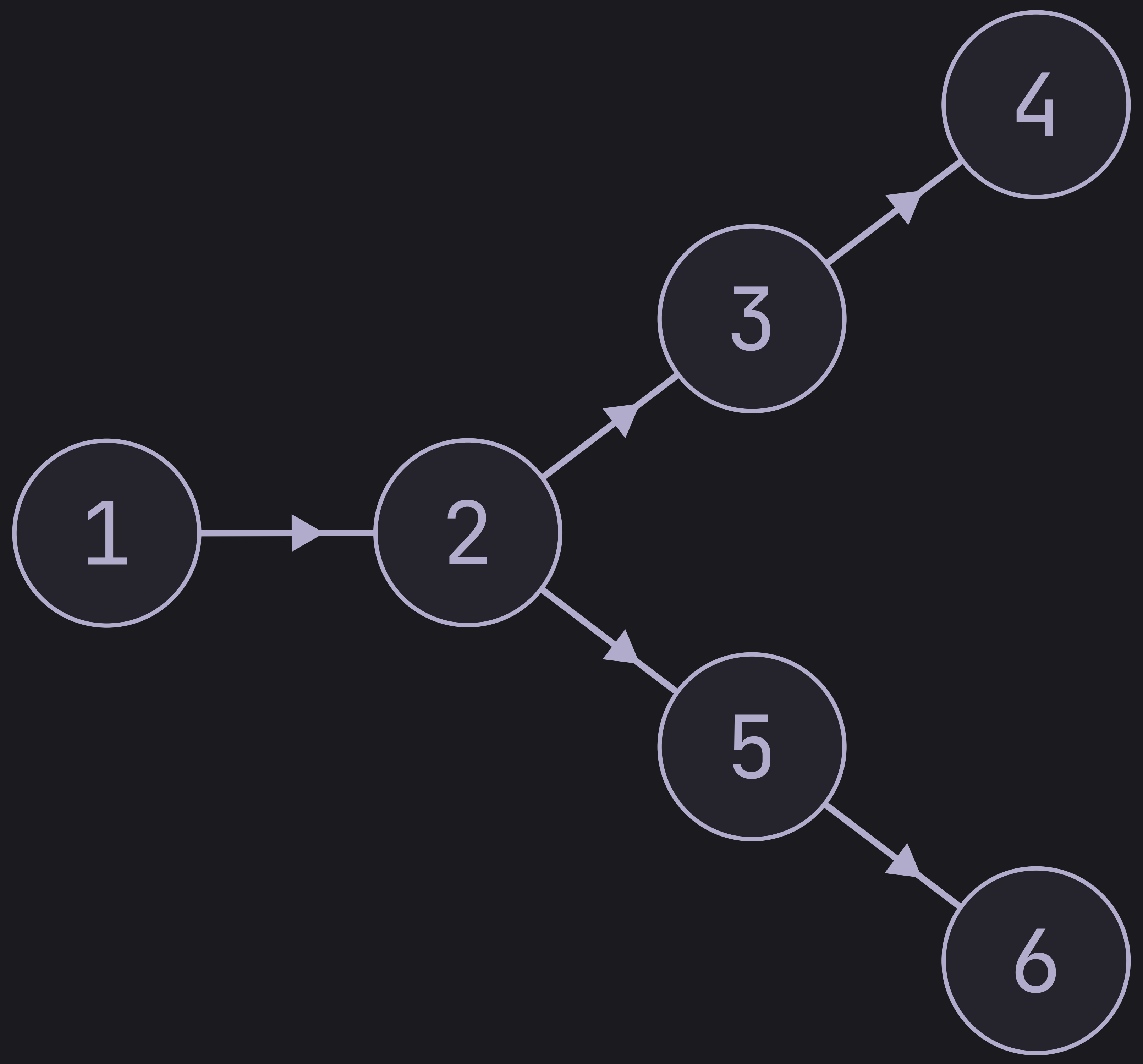
Detecting Cycles



This is very similar to DFS!

So maybe we can use DFS to detect a cycle?

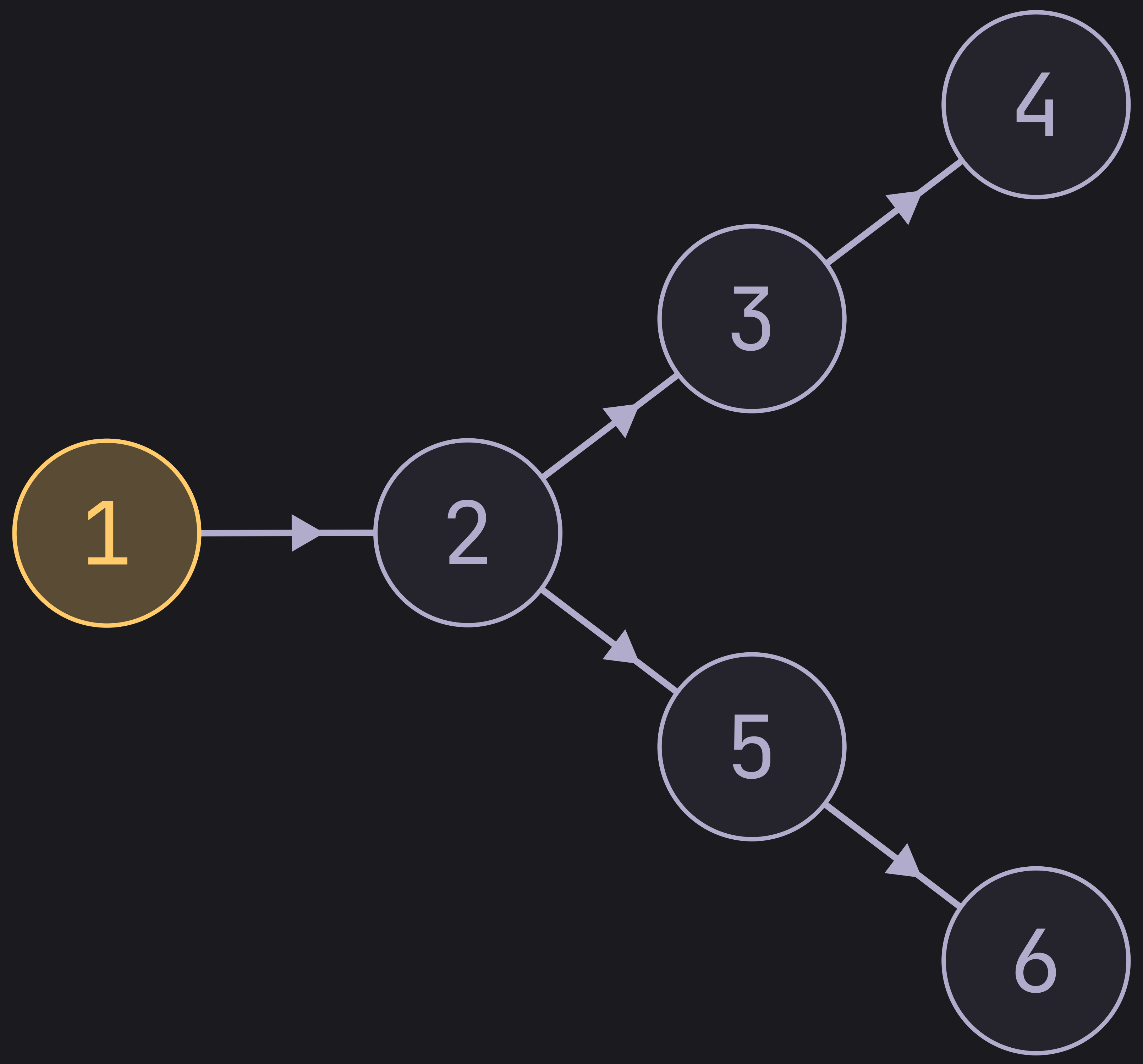
Detecting Cycles



Let’s have at look at DFS on a simple graph

Remember DFS is like exploring a maze by going down one corridor until we can’t go any further and then turning back

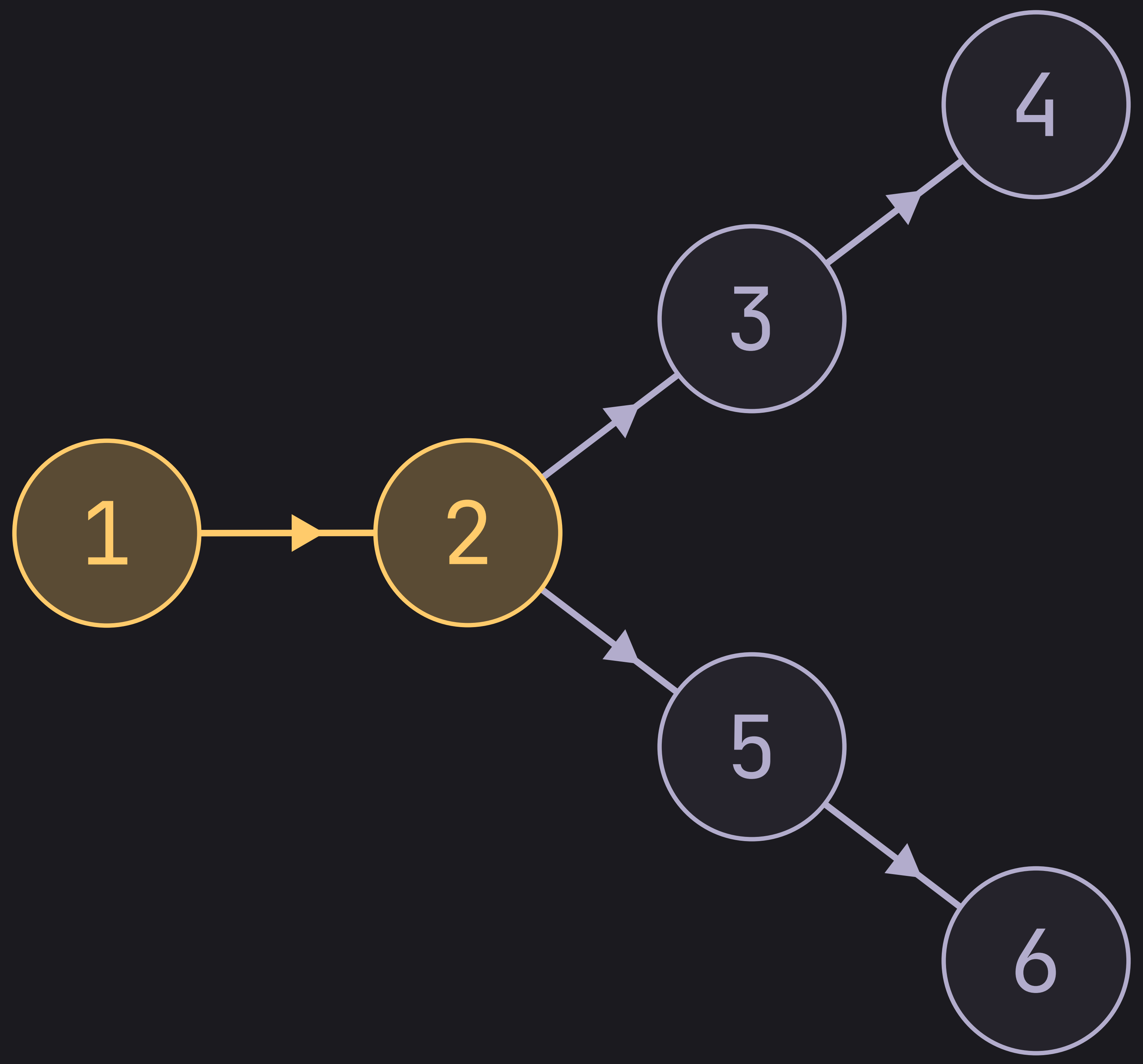
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **“corridor”** we are exploring

[1]

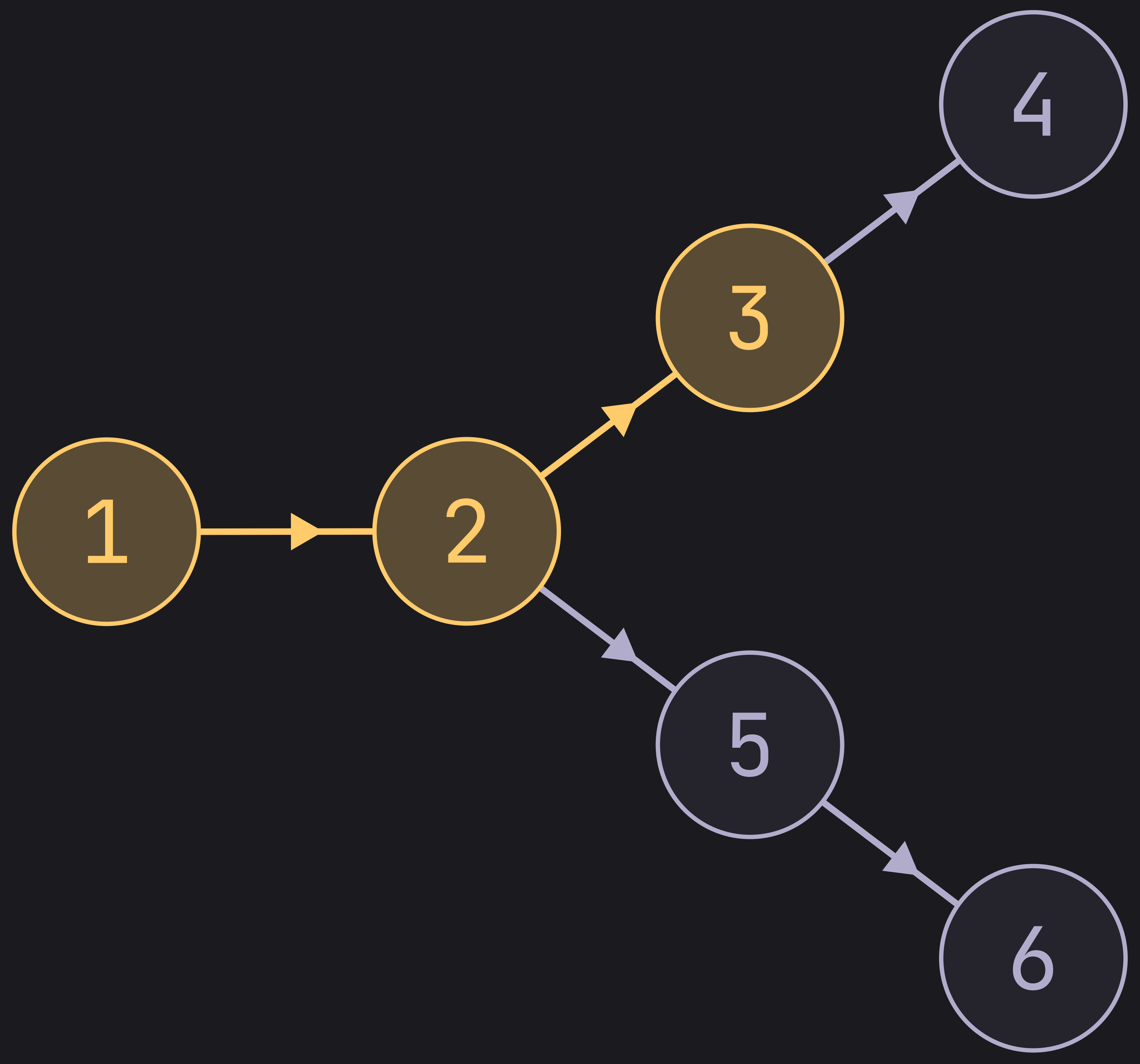
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **“corridor”** we are exploring

[1, 2]

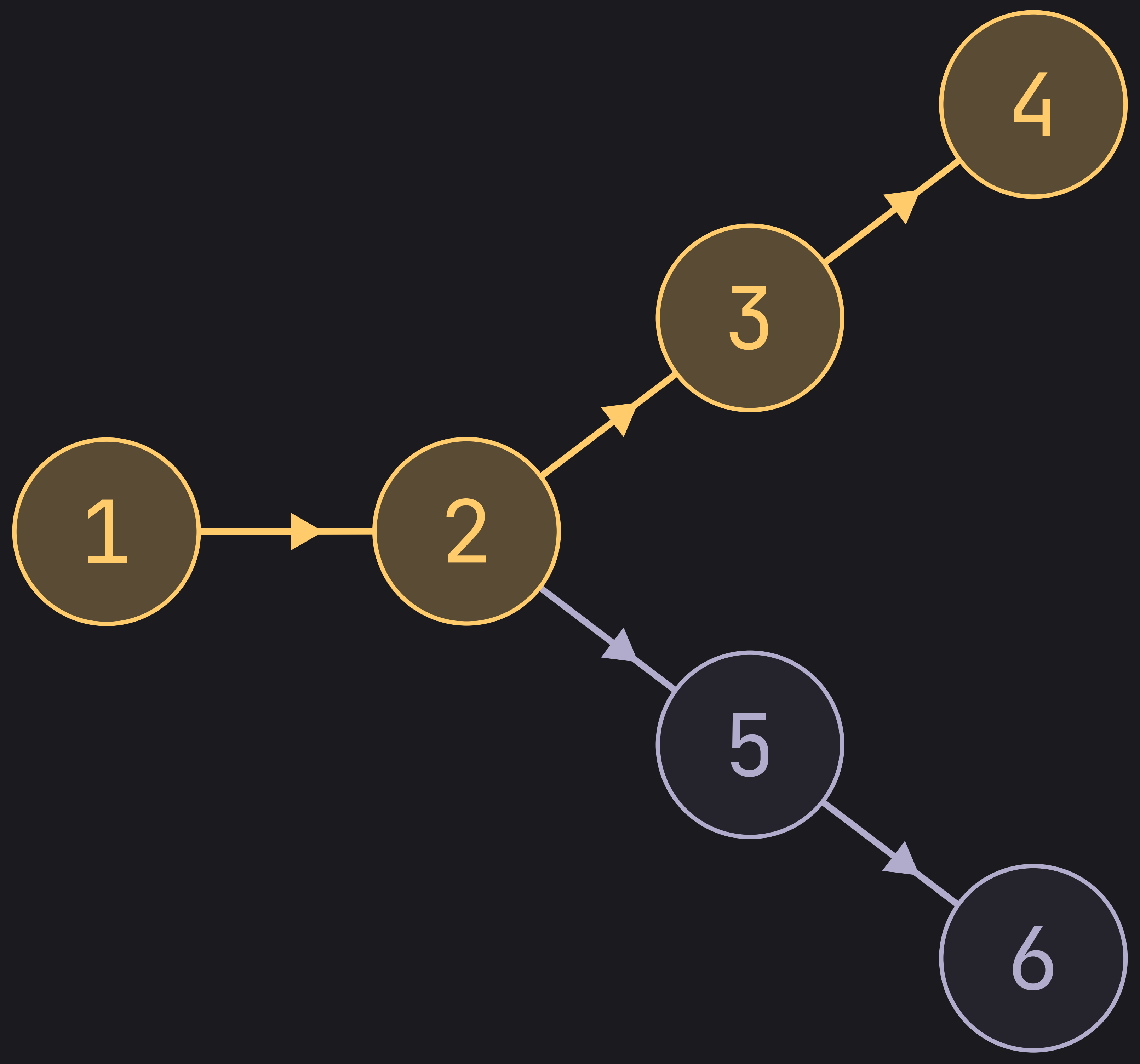
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **"corridor"** we are exploring

[1, 2, 3]

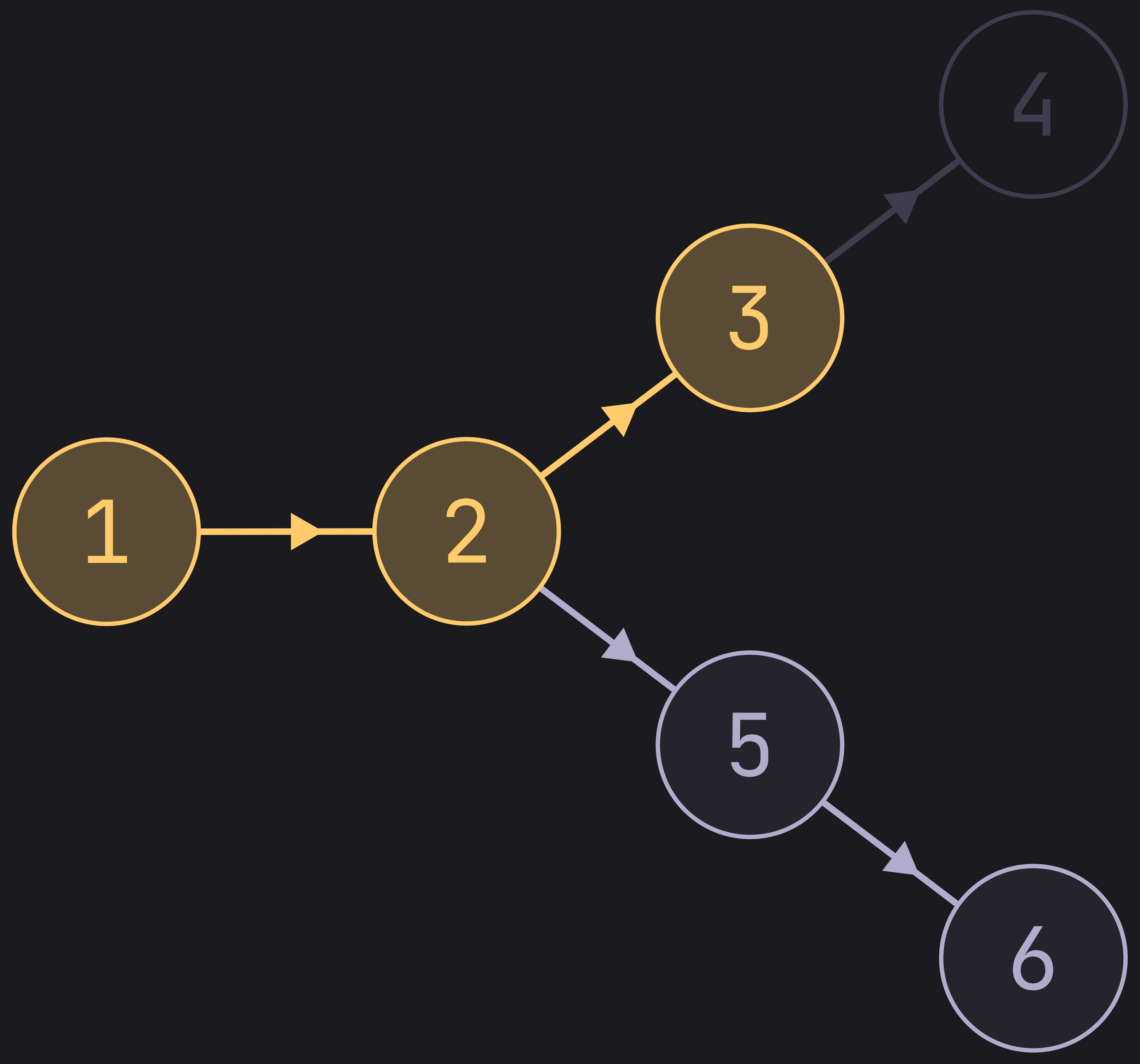
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **“corridor”** we are exploring

[1, 2, 3, 4]

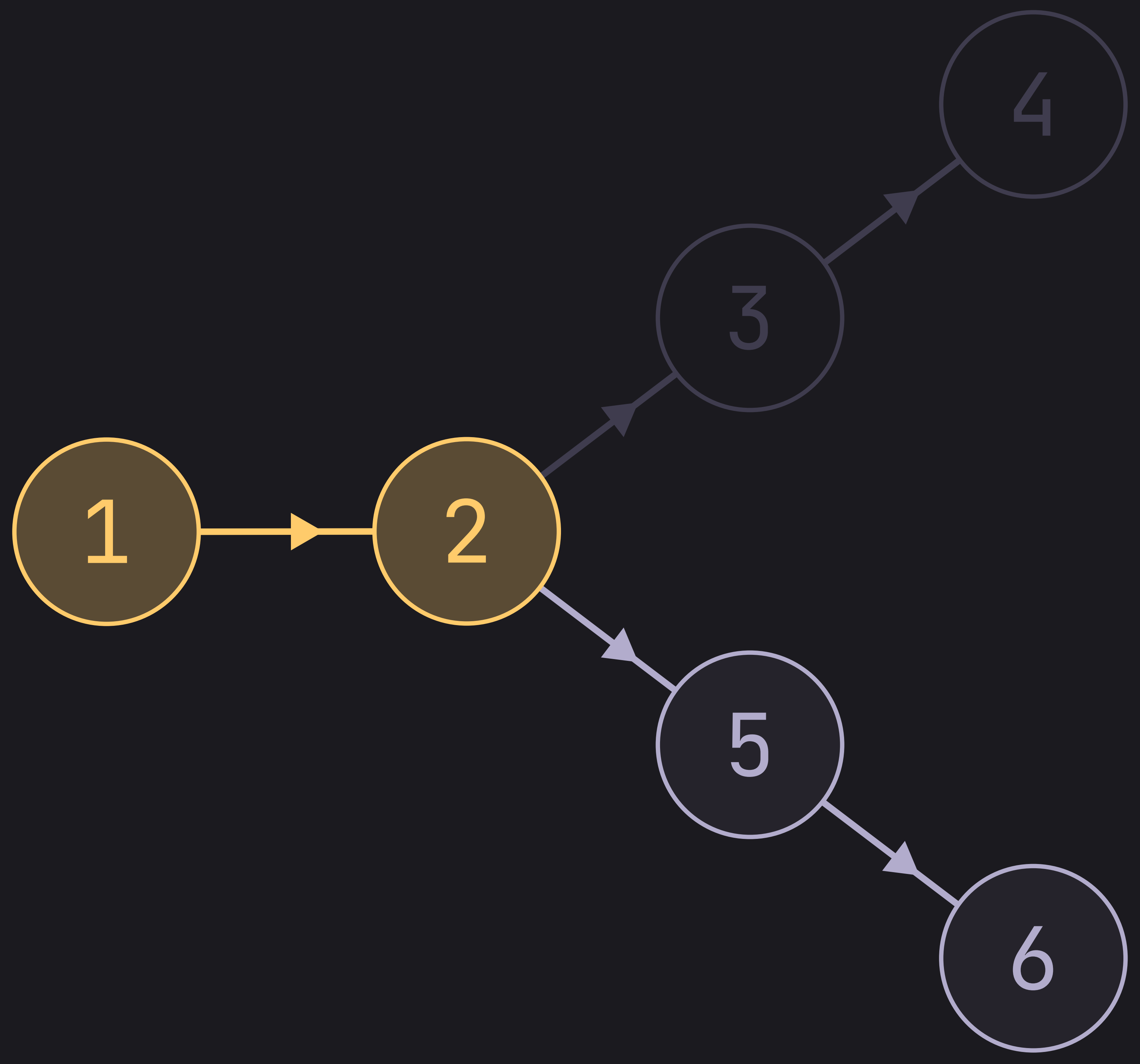
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **"corridor"** we are exploring

[1, 2, 3]

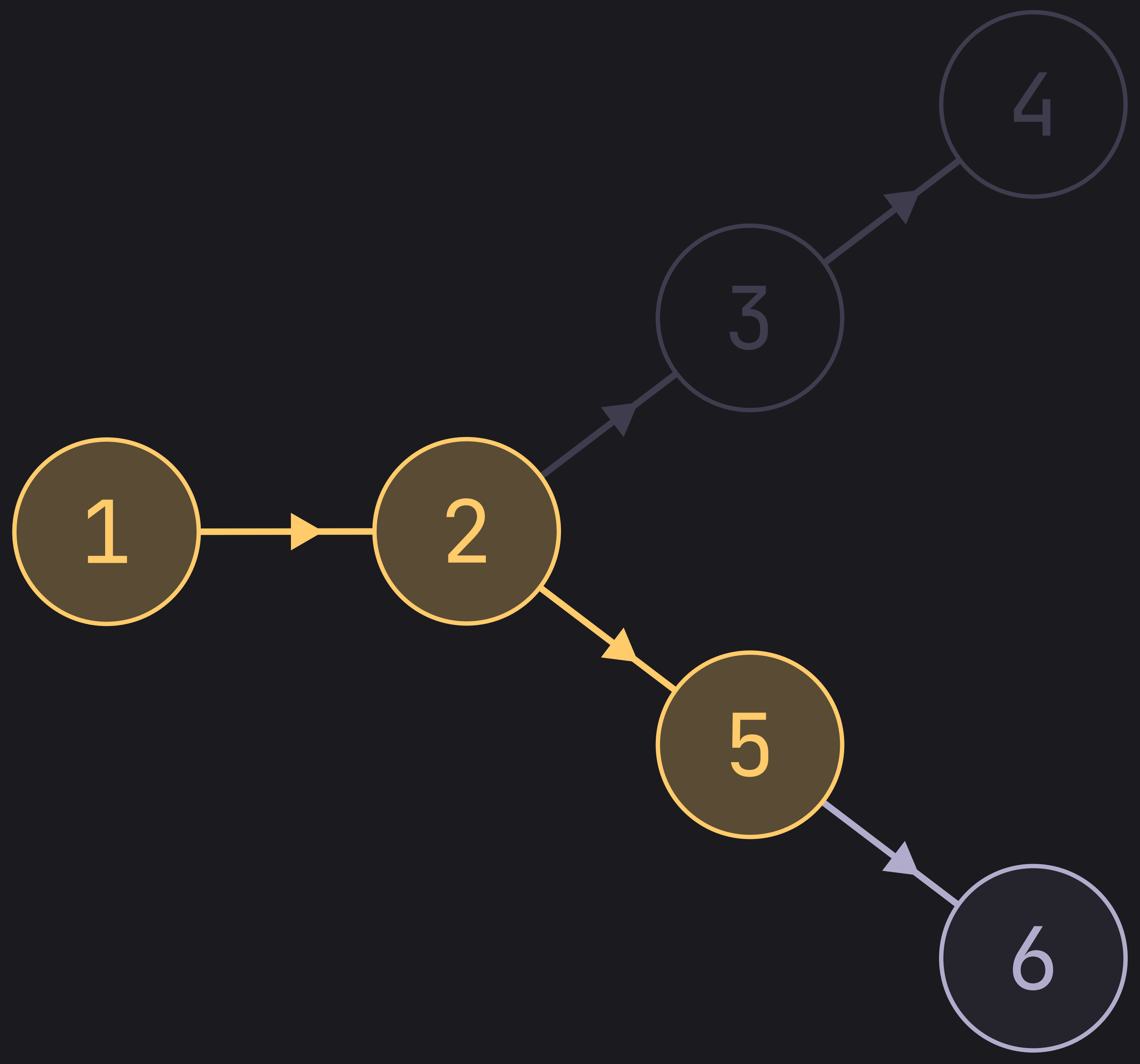
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **“corridor”** we are exploring

[1, 2]

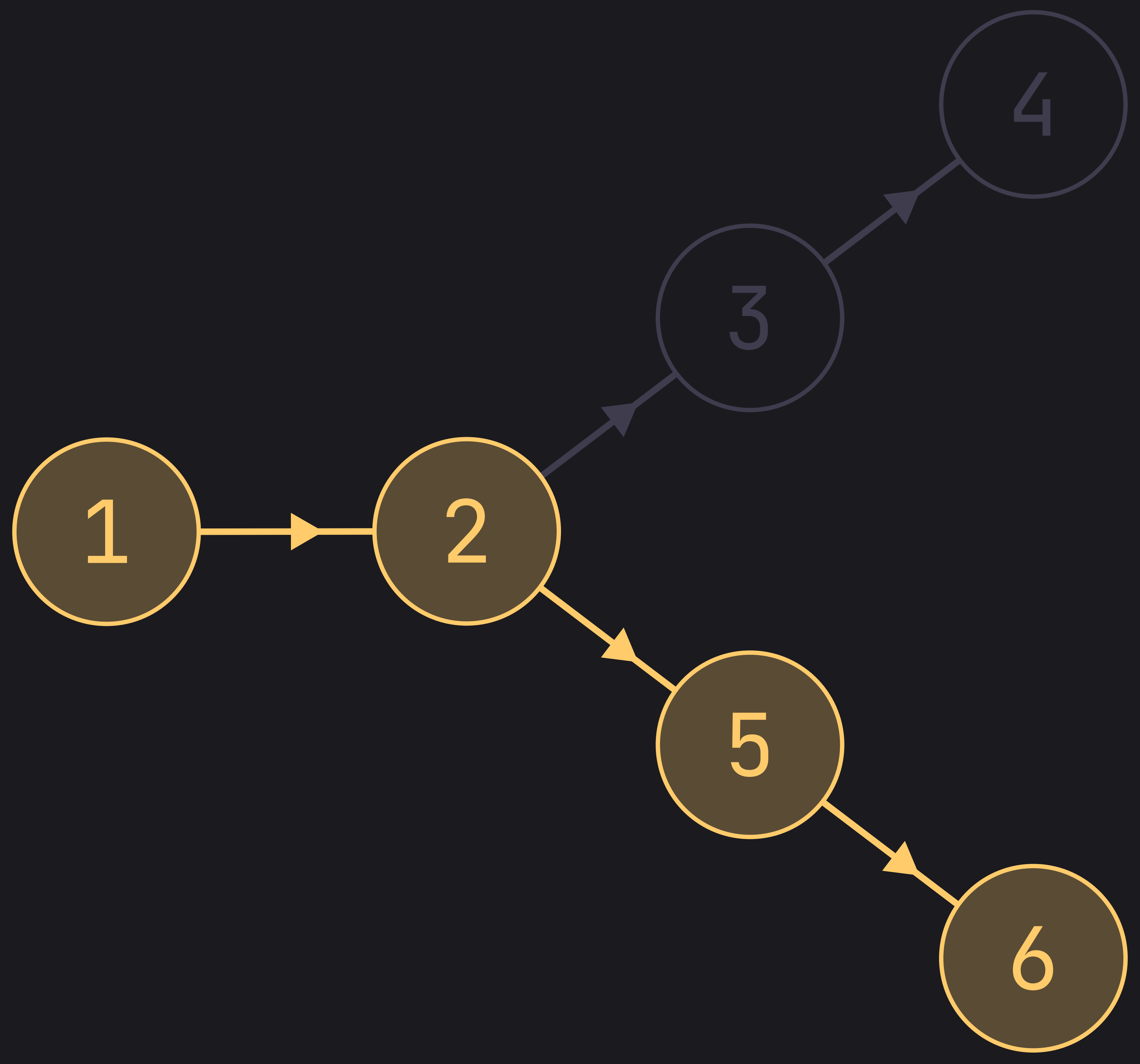
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **"corridor"** we are exploring

[1, 2, 5]

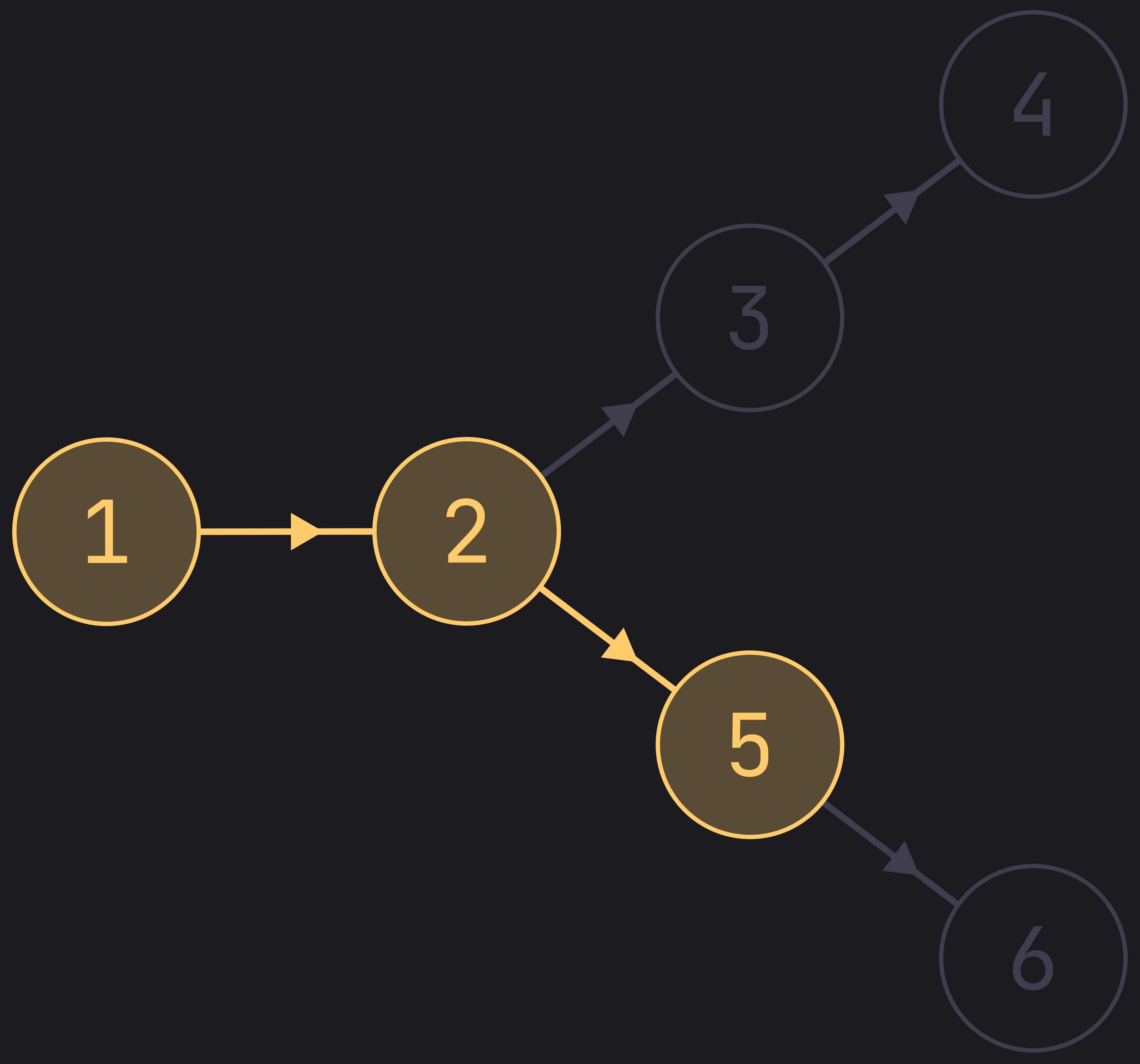
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **"corridor"** we are exploring

[1, 2, 5, 6]

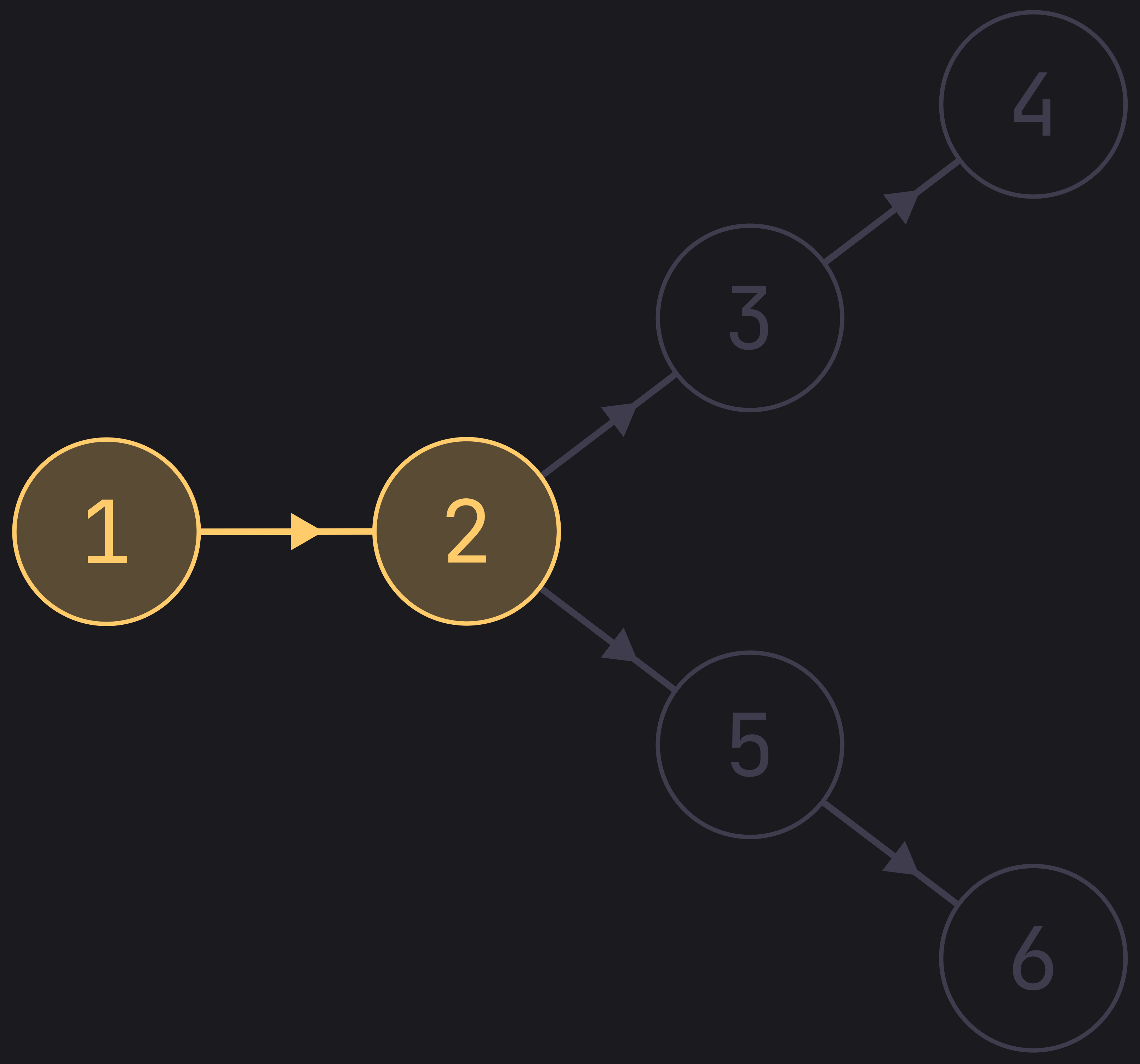
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **"corridor"** we are exploring

[1, 2, 5]

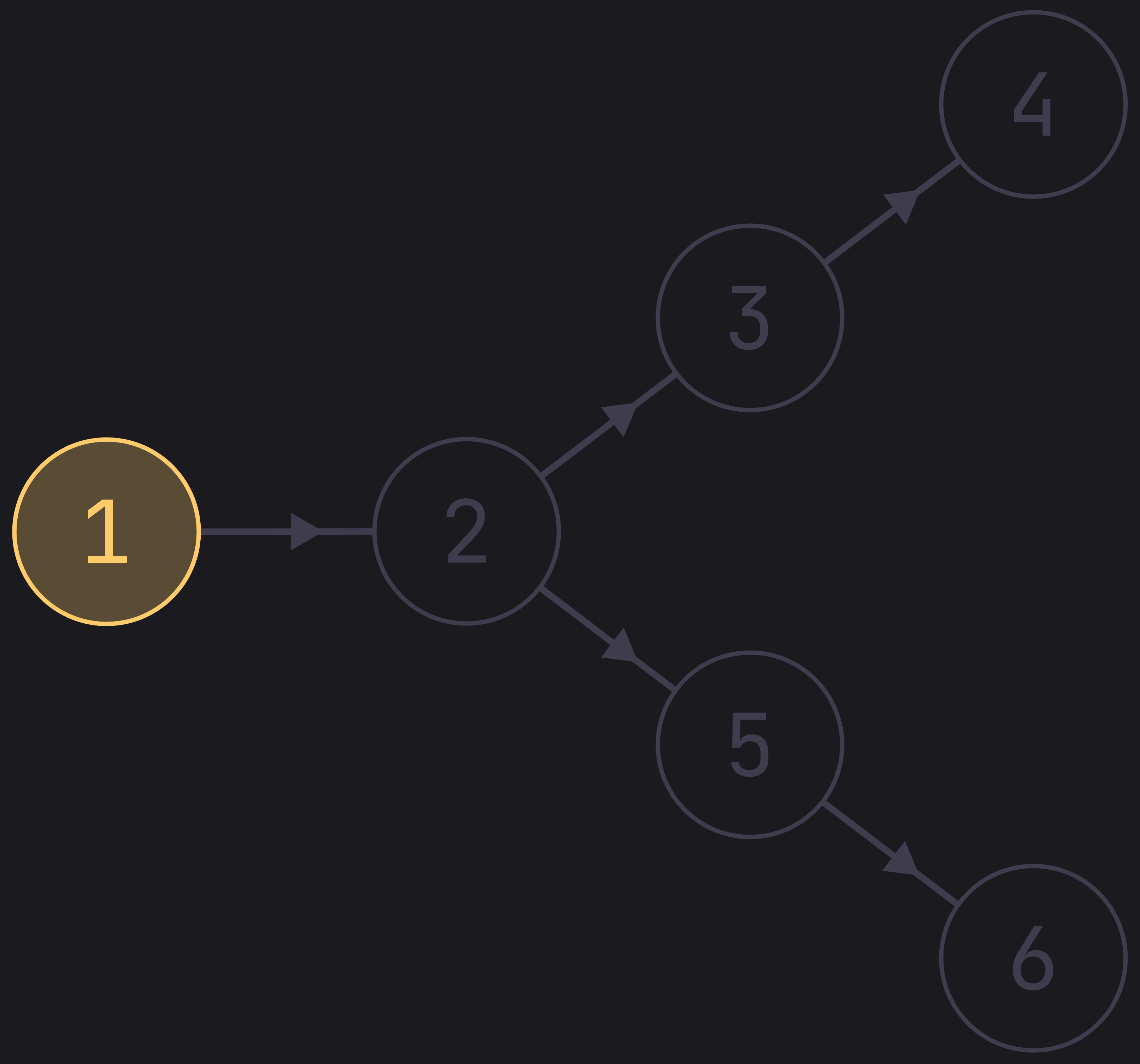
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **“corridor”** we are exploring

[1, 2]

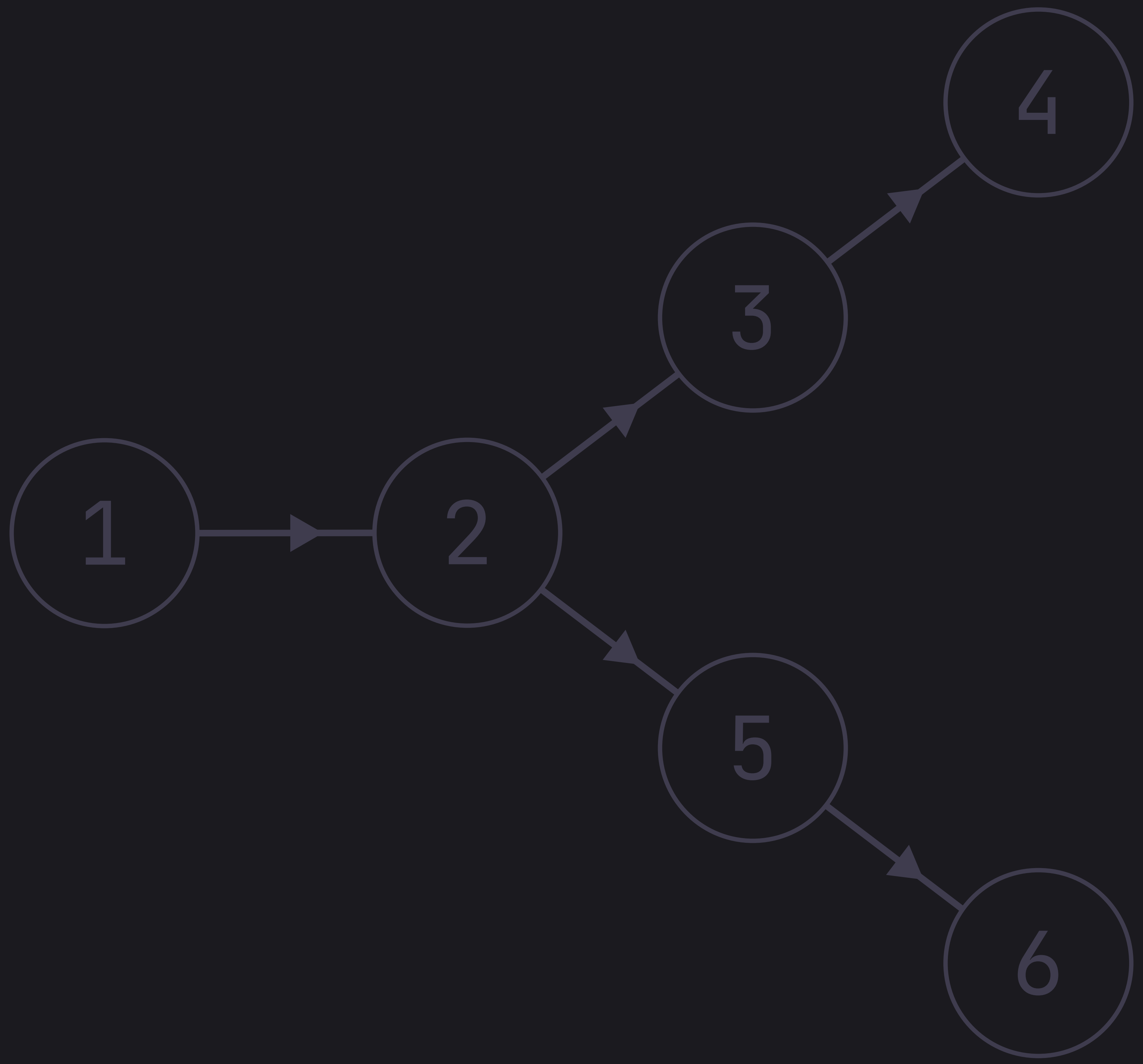
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **"corridor"** we are exploring

[1]

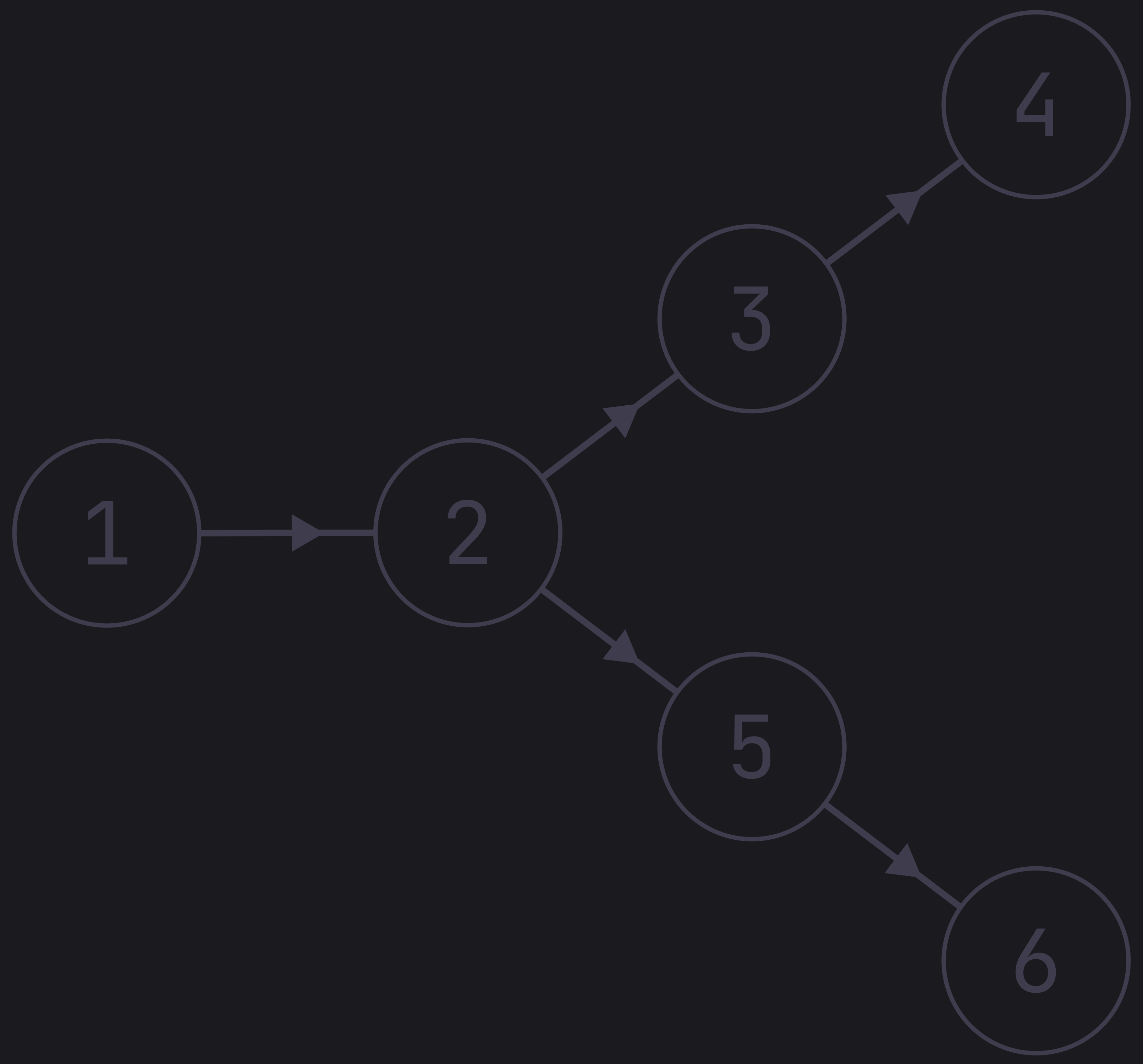
Detecting Cycles



We are going to keep track of the vertices that are currently part of the **“corridor”** we are exploring

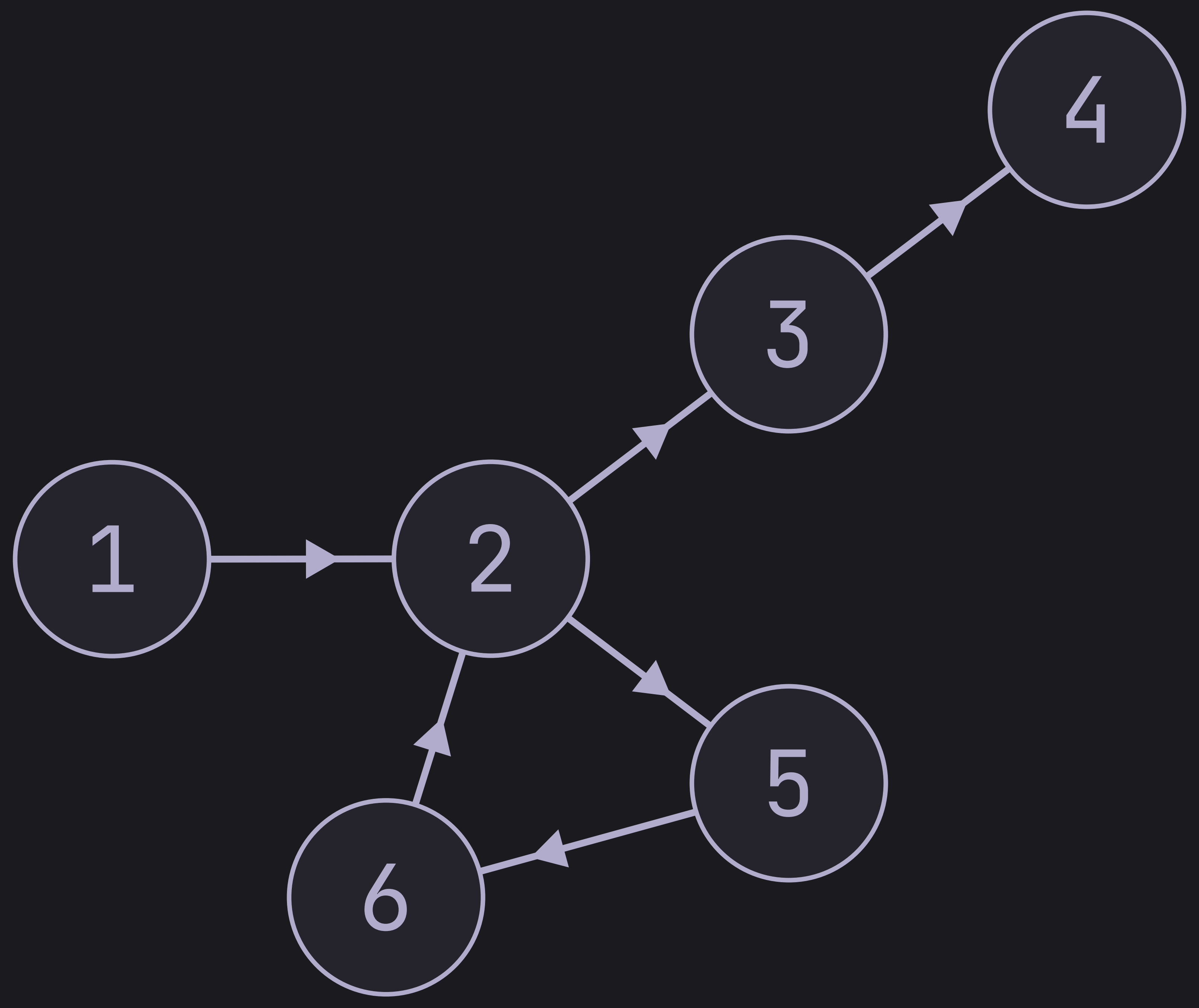
[ ]

Detecting Cycles



So what would happen if we encounter a cycle along our journey?

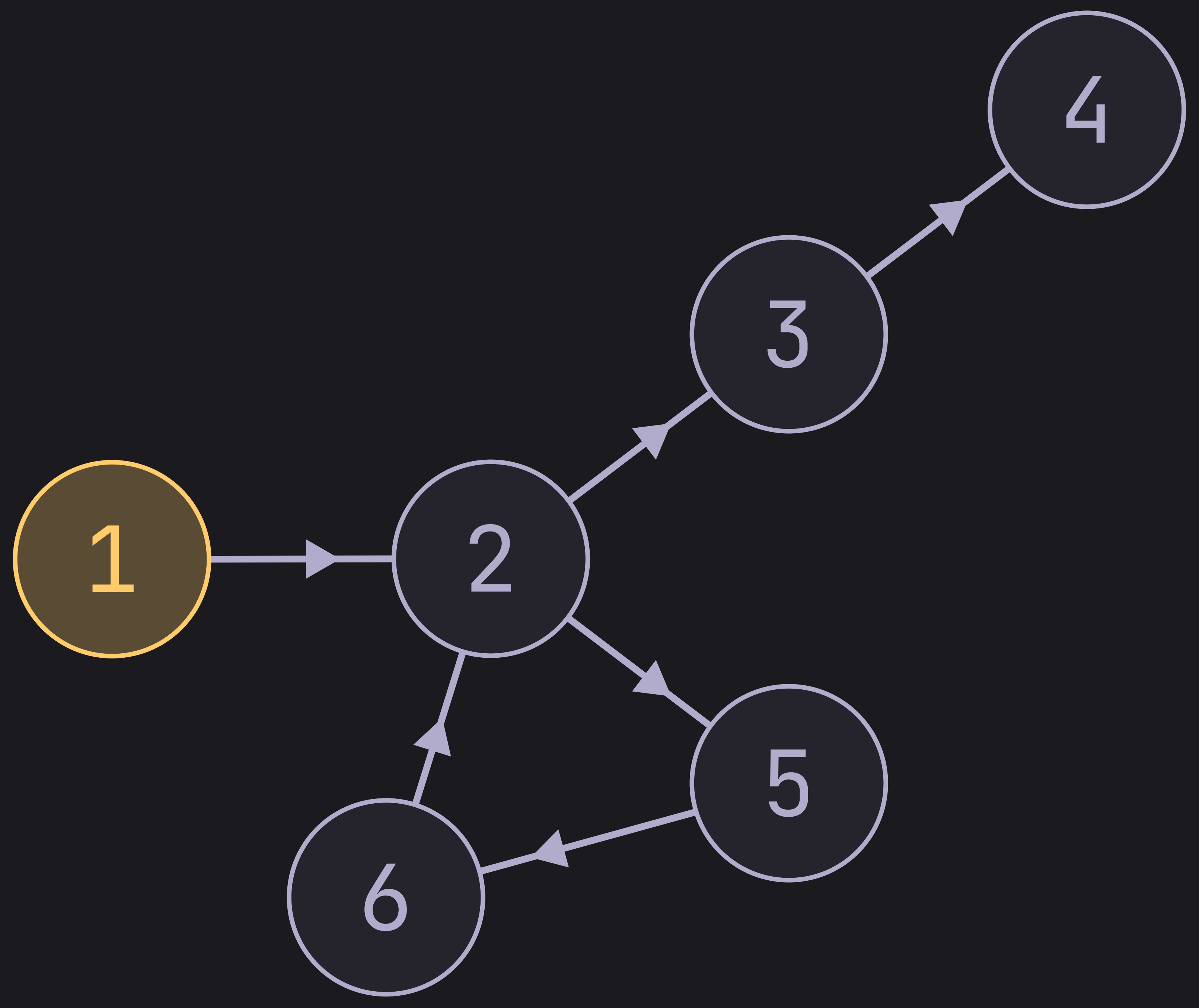
Detecting Cycles



So what would happen if we encounter a cycle along our journey?

[ ]

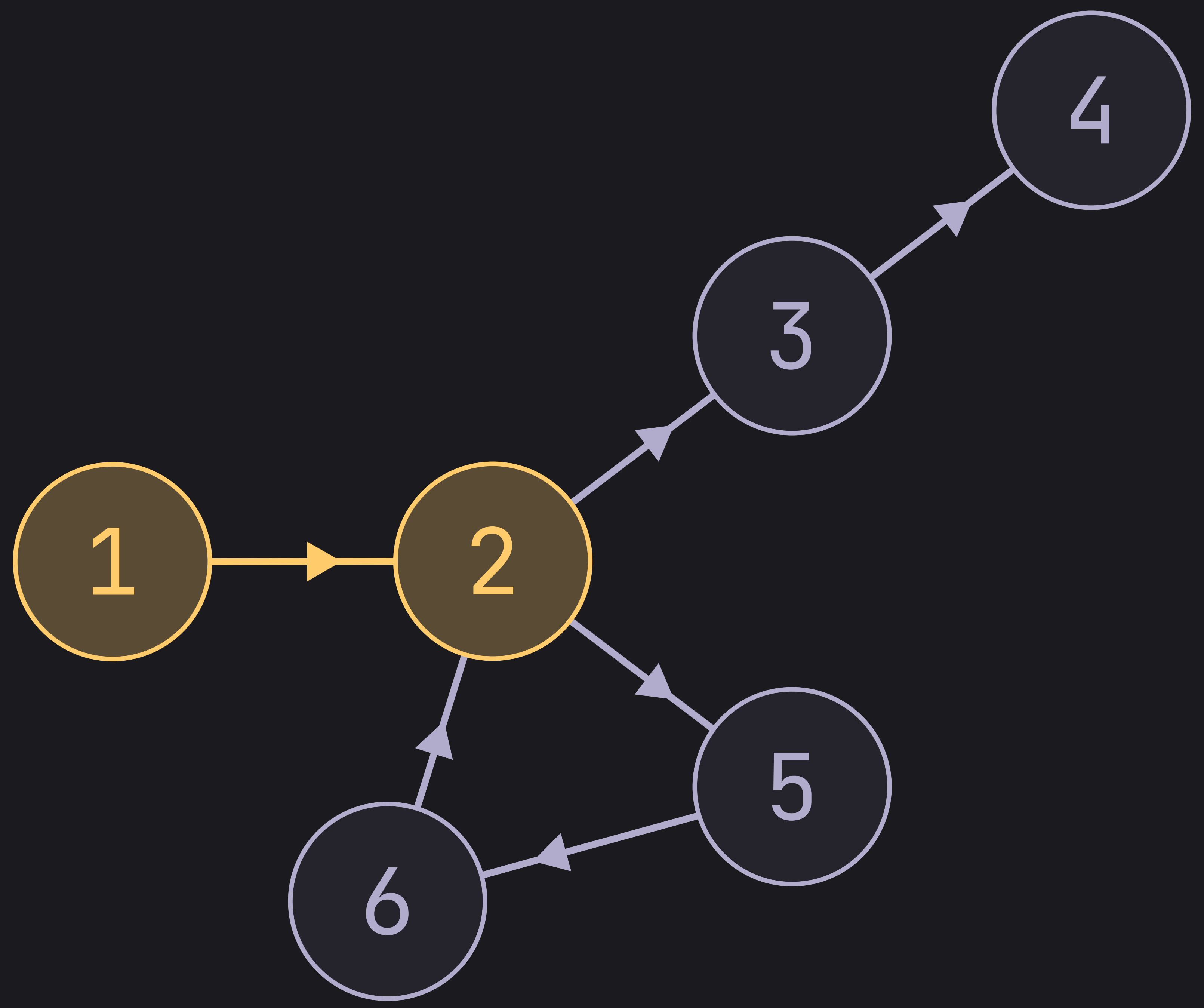
Detecting Cycles



So what would happen if we encounter a cycle along our journey?

[1]

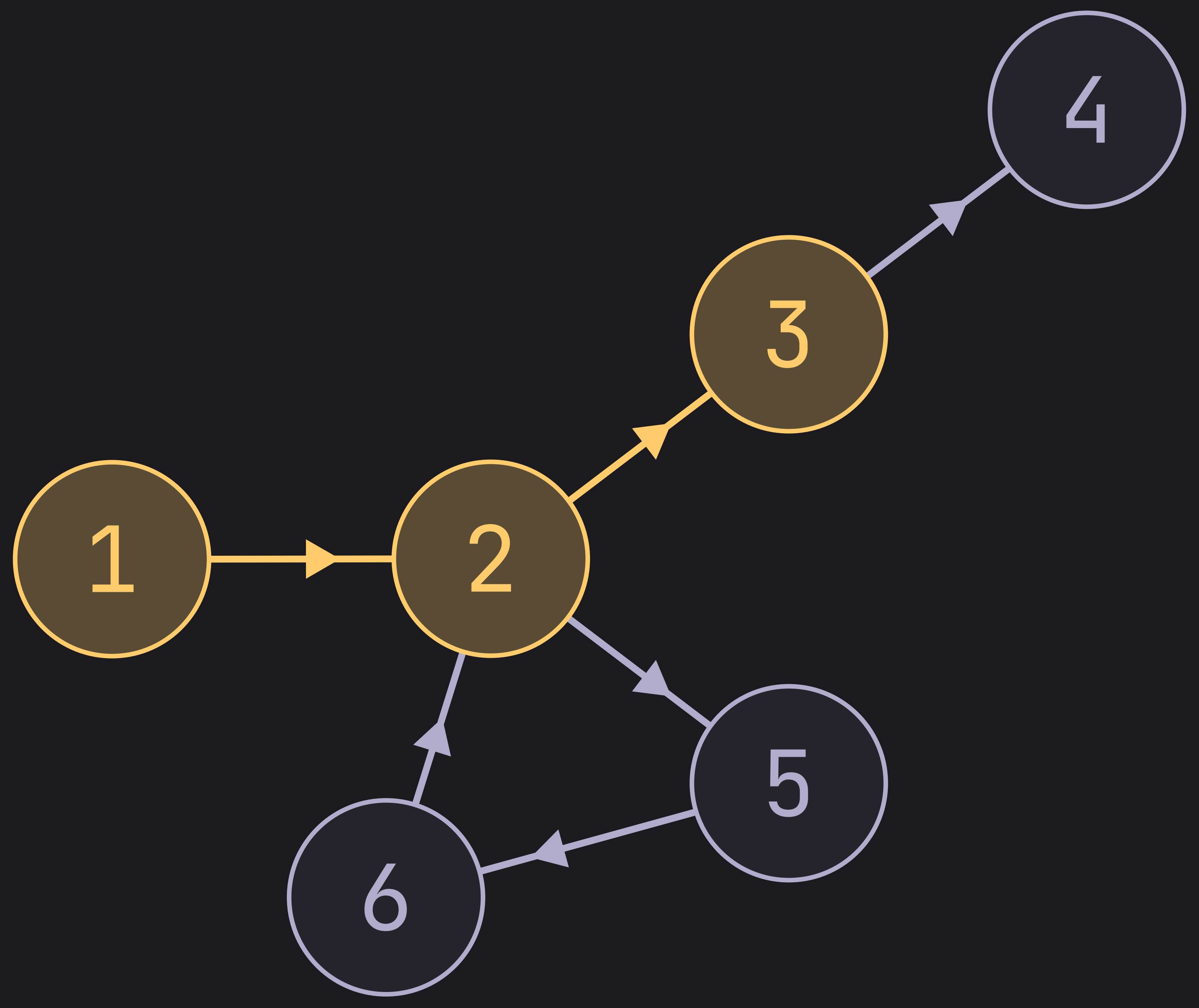
Detecting Cycles



So what would happen if we encounter a cycle along our journey?

[1, 2]

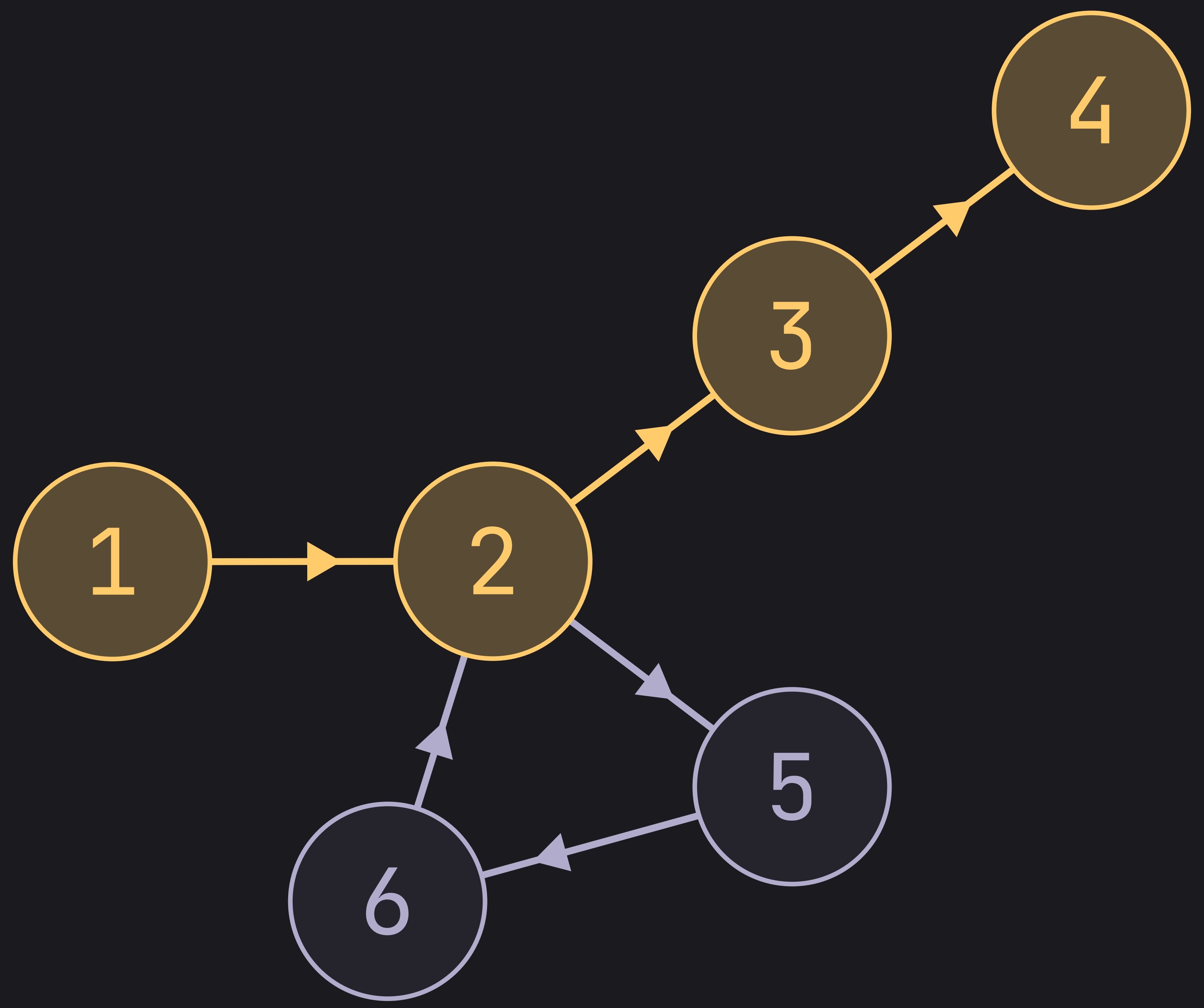
Detecting Cycles



So what would happen if we encounter a cycle along our journey?

[1, 2, 3]

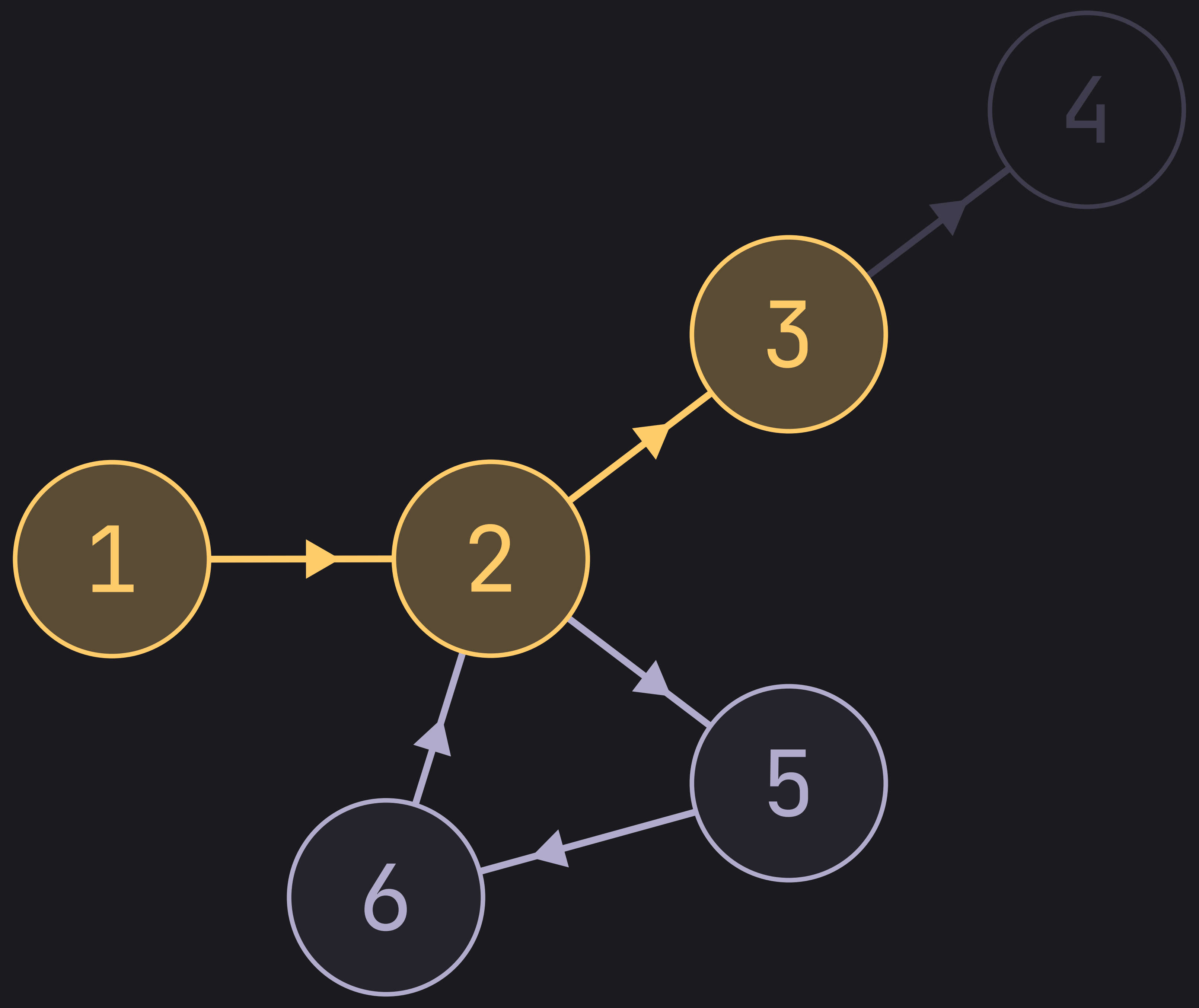
Detecting Cycles



So what would happen if we encounter a cycle along our journey?

[1, 2, 3, 4]

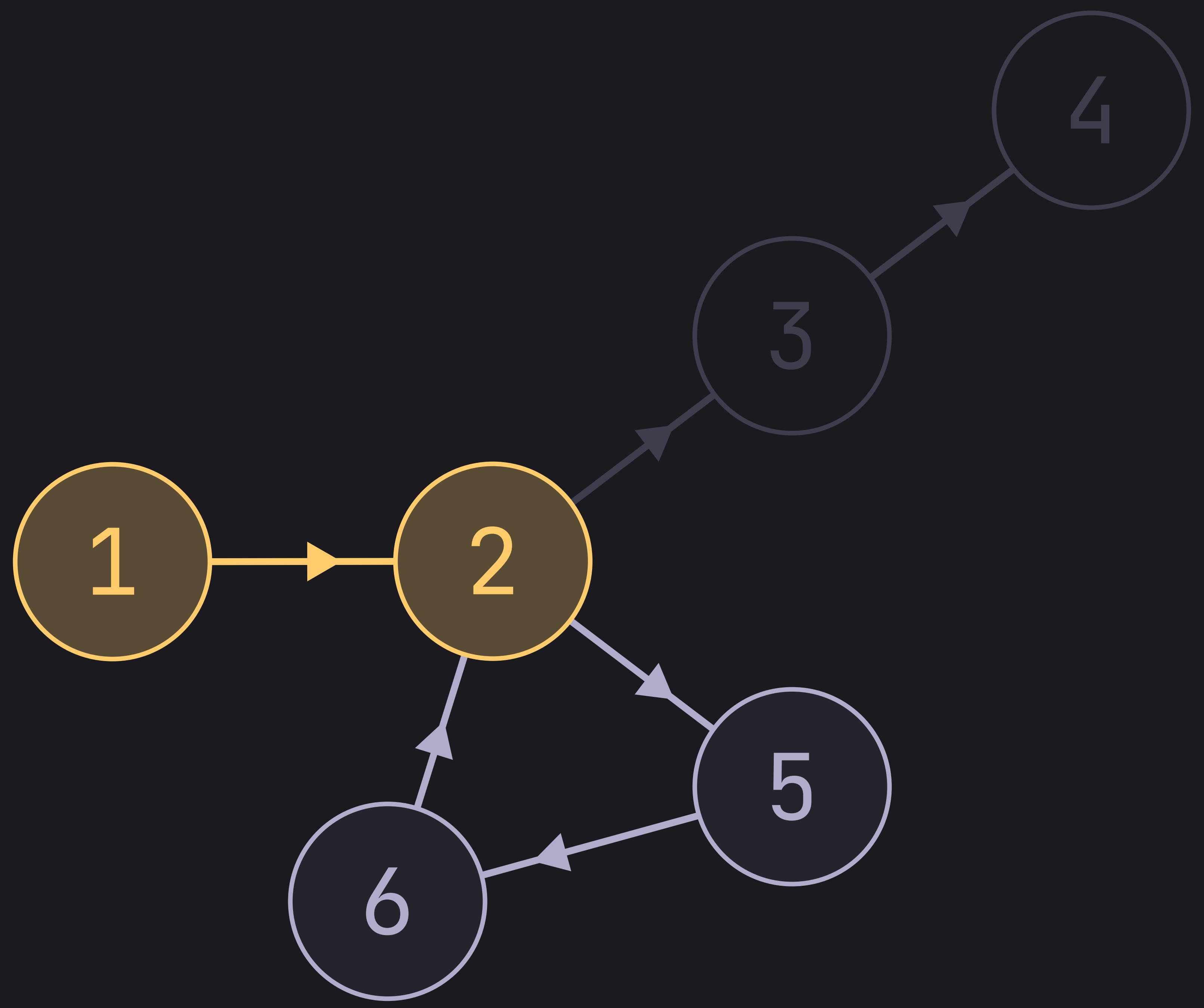
Detecting Cycles



So what would happen if we encounter a cycle along our journey?

[1, 2, 3]

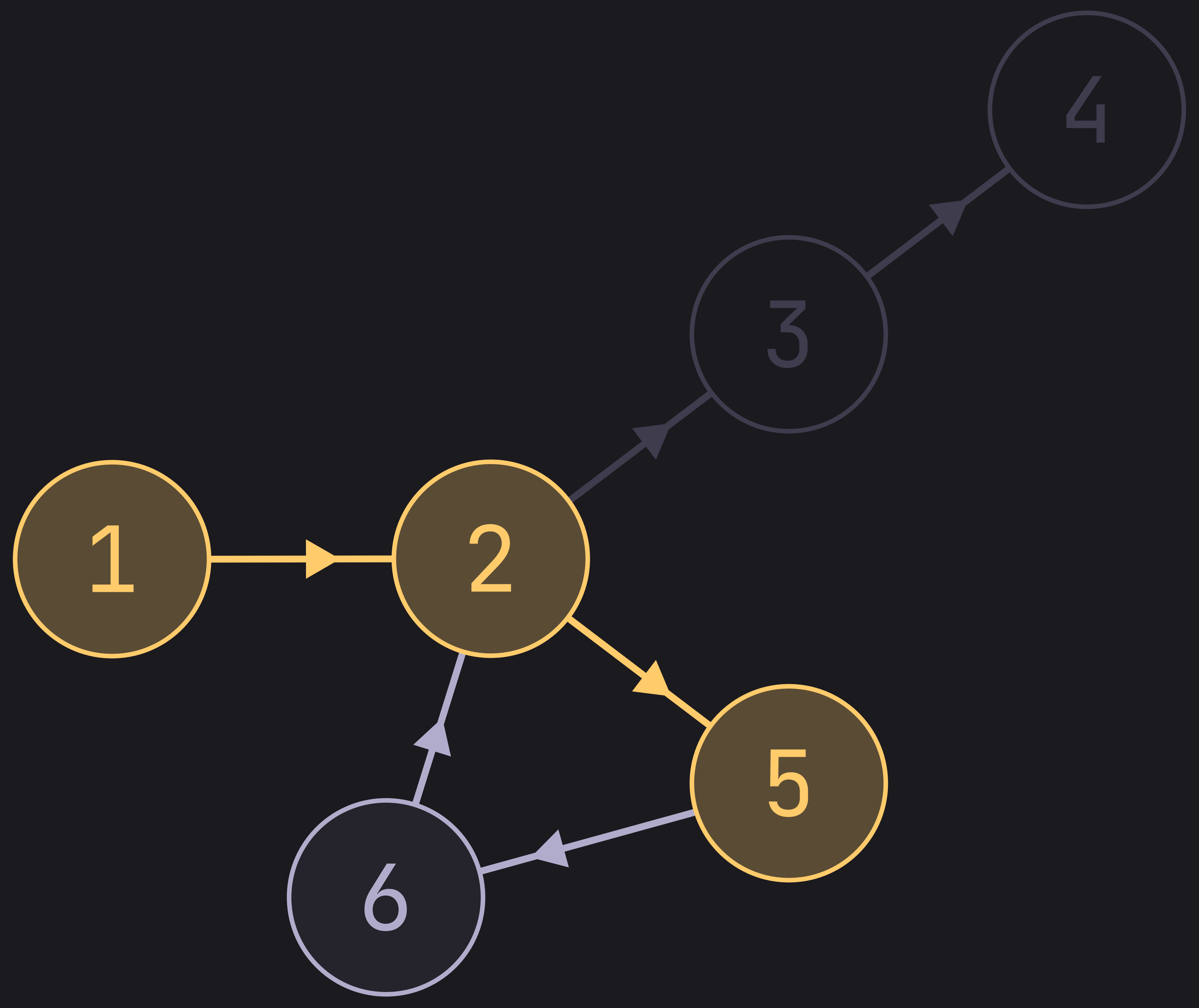
Detecting Cycles



So what would happen if we encounter a cycle along our journey?

[1, 2]

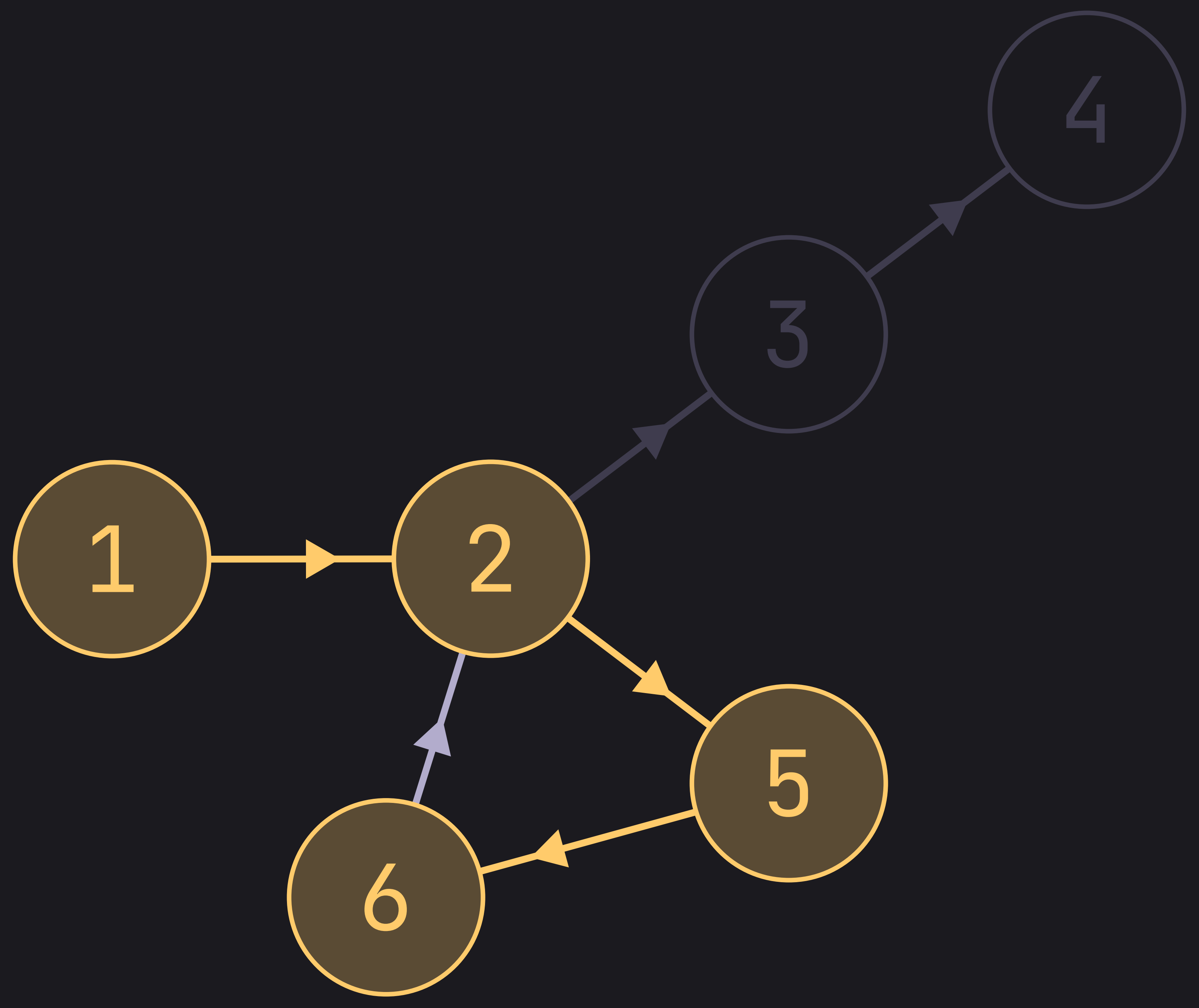
Detecting Cycles



So what would happen if we encounter a cycle along our journey?

[1, 2, 5]

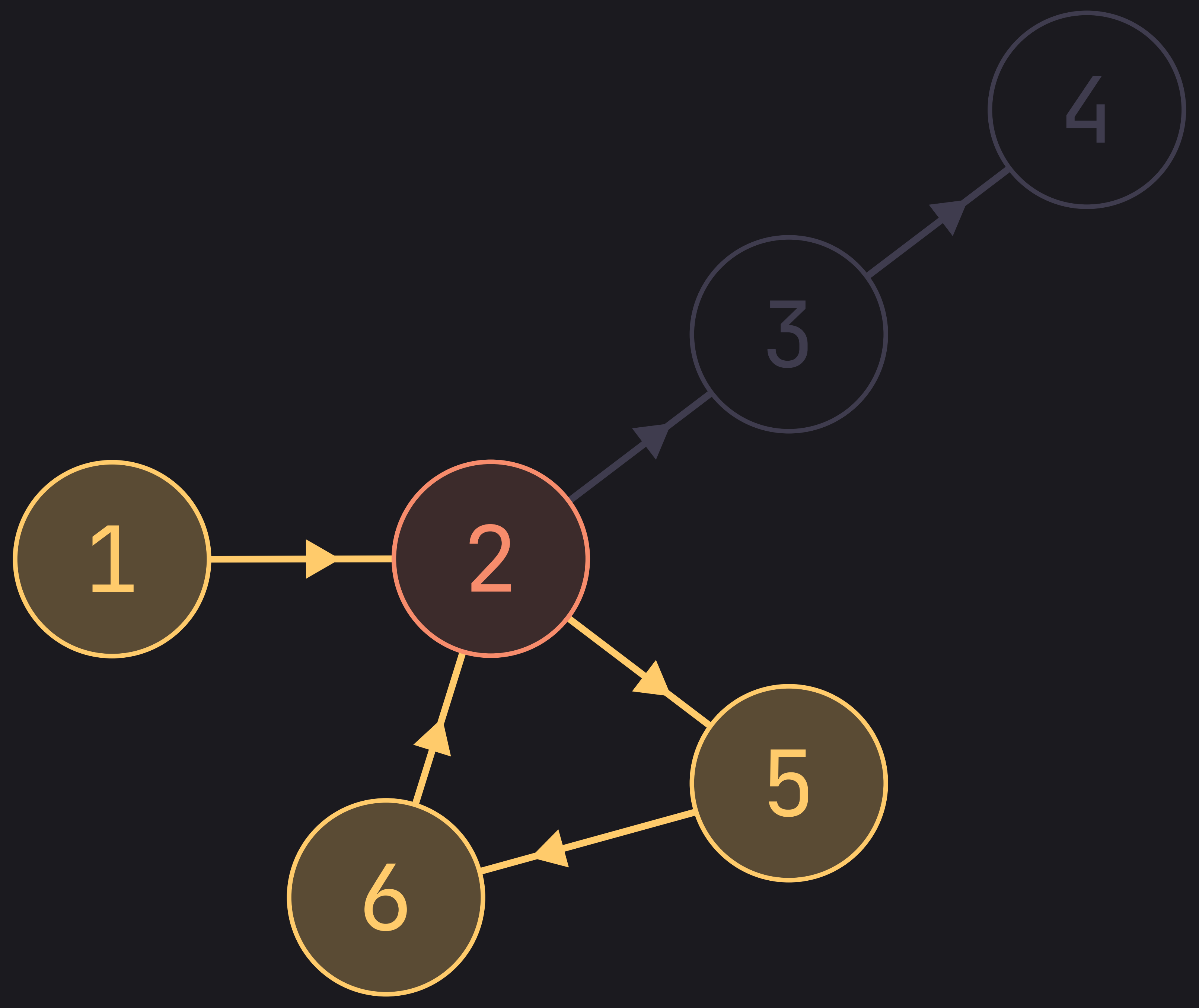
Detecting Cycles



So what would happen if we encounter a cycle along our journey?

[1, 2, 5, 6]

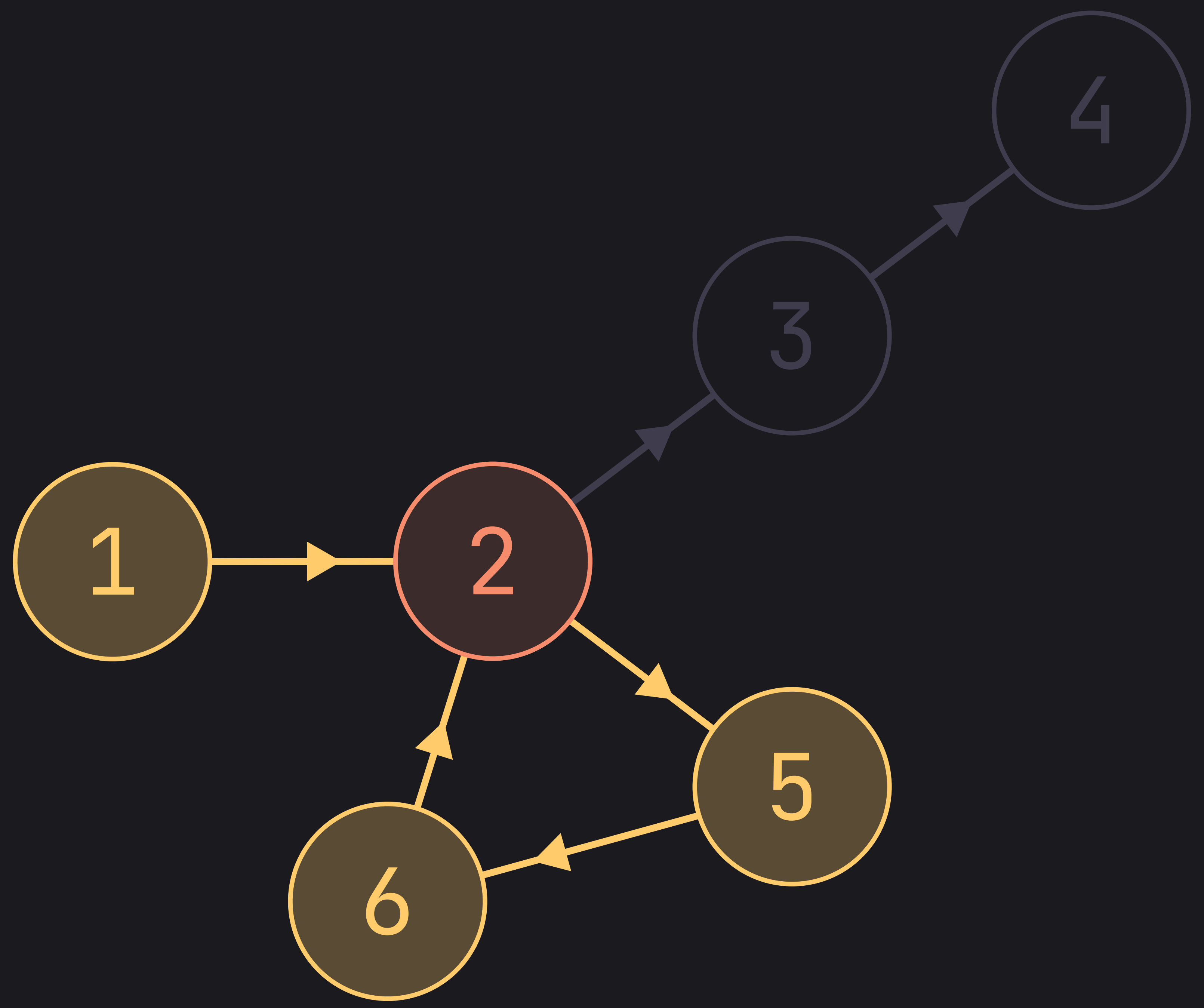
Detecting Cycles



Its only during a **cycle** that we reencounter one of the vertices of a "corridor" that we are currently exploring

[1, 2, 5, 6, 2]

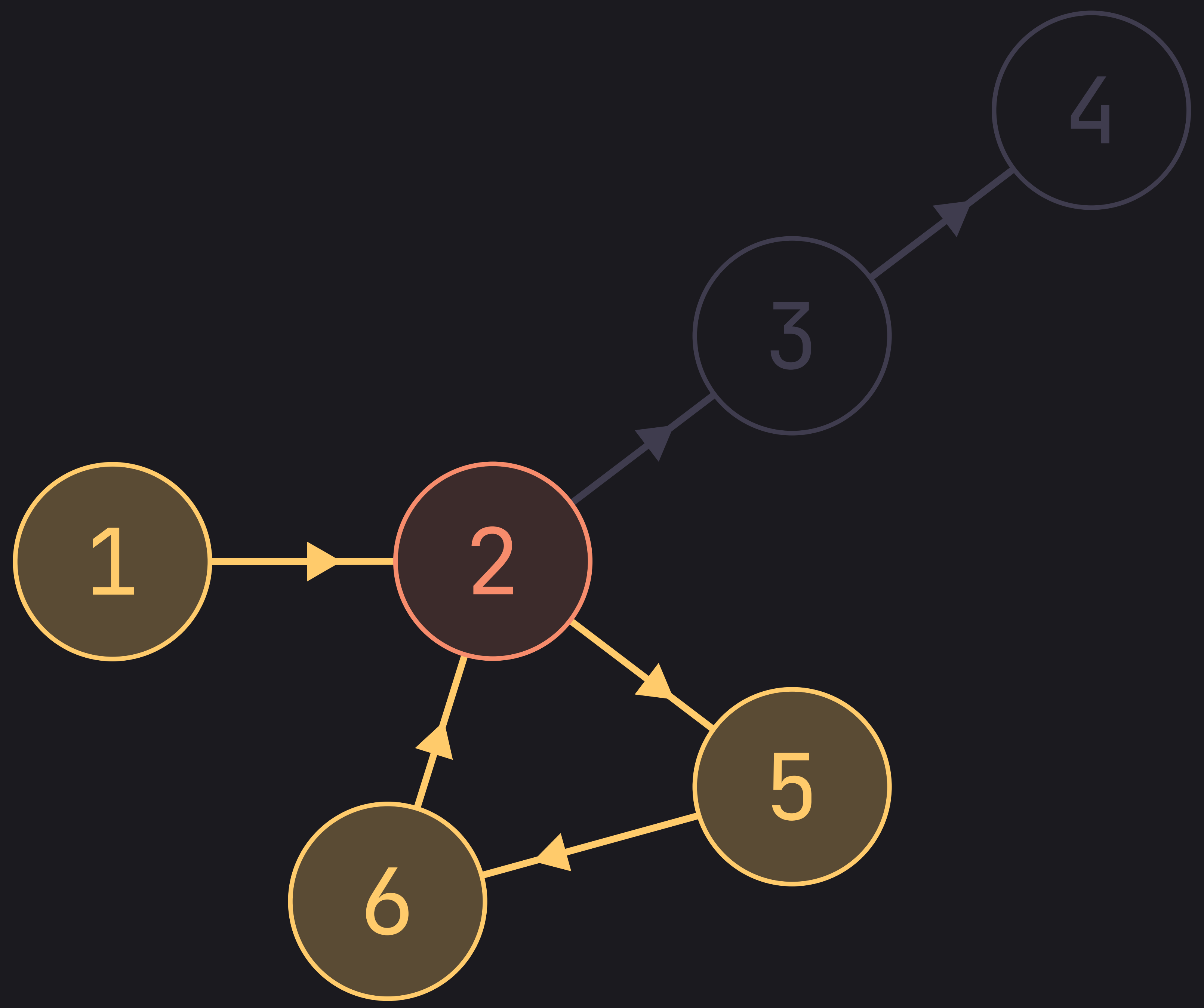
Detecting Cycles



So we are going to use a slightly modified version of DFS to check our graph for cycles

[1, 2, 5, 6, 2]

Detecting Cycles



So we are going to use a slightly modified version of DFS to check our graph for cycles

[1, 2, 5, 6, 2]

Detecting Cycles

```
class DirectedCycle:
    visited = []
    onStack = []

    bool containsCycle(Graph graph):
    bool dfs(int start, Graph graph)
```

So we are going to create a simple class that is going to help us find cycles in a graph

Detecting Cycles

```
class DirectedCycle:
    visited = []
    onStack = []

    bool containsCycle(Graph graph):
    bool dfs(int start, Graph graph)
```

```
void dfs(int v, Graph graph):
    visited[v] = true
    for neighbour in graph.getNeighbours(v):
        if not visited[neighbour]:
            dfs(neighbour, graph)
```

Let’s start with a our basic implementation of DFS

Detecting Cycles

```
class DirectedCycle:
    visited = []
    onStack = []

    bool containsCycle(Graph graph):
    bool dfs(int start, Graph graph)
```

```
void dfs(int v, Graph graph):
    onStack[v] = true
    visited[v] = true
    for neighbour in graph.getNeighbours(v):
        if not visited[neighbour]:
            dfs(neighbour, graph)
```

We need to keep track of the current “corridor” we are exploring. So we use add the onStack vector.

Detecting Cycles

```
class DirectedCycle:
    visited = []
    onStack = []

    bool containsCycle(Graph graph):
    bool dfs(int start, Graph graph)
```

```
bool dfs(int v, Graph graph):
    onStack[v] = true
    visited[v] = true
    for neighbour in graph.getNeighbours(v):
        if onStack[neighbour]:
            return true
        if not visited[neighbour]:
            dfs(neighbour, graph)
```

Now we also want to check if we reencounter one of the "corridor's" vertices. If we do we have found a cycle

Detecting Cycles

```
class DirectedCycle:
    visited = []
    onStack = []

    bool containsCycle(Graph graph):
    bool dfs(int start, Graph graph)
```

```
bool dfs(int v, Graph graph):
    onStack[v] = true
    visited[v] = true
    for neighbour in graph.getNeighbours(v):
        if onStack[neighbour]:
            return true
        if not visited[neighbour]:
            existsCycle = dfs(neighbour, graph)
            if existsCycle:
                return true
```

So we now also need to update the recursive call. This call will also return true if it finds a cycle

Detecting Cycles

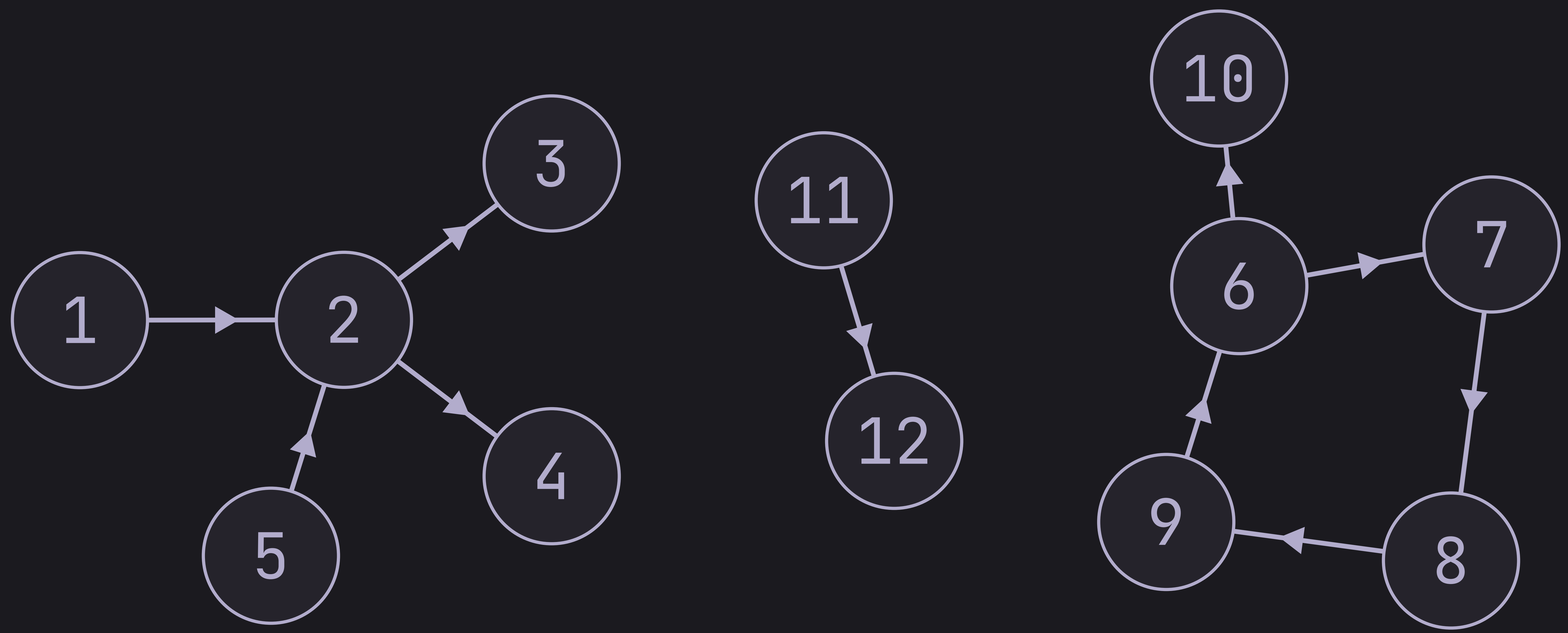
```
class DirectedCycle:
    visited = []
    onStack = []

    bool containsCycle(Graph graph):
    bool dfs(int start, Graph graph)
```

```
bool dfs(int v, Graph graph):
    onStack[v] = true
    visited[v] = true
    for neighbour in graph.getNeighbours(v):
        if onStack[neighbour]:
            return true
        if not visited[neighbour]:
            existsCycle = dfs(neighbour, graph)
            if existsCycle:
                return true
    onStack[v] = false
    return false
```

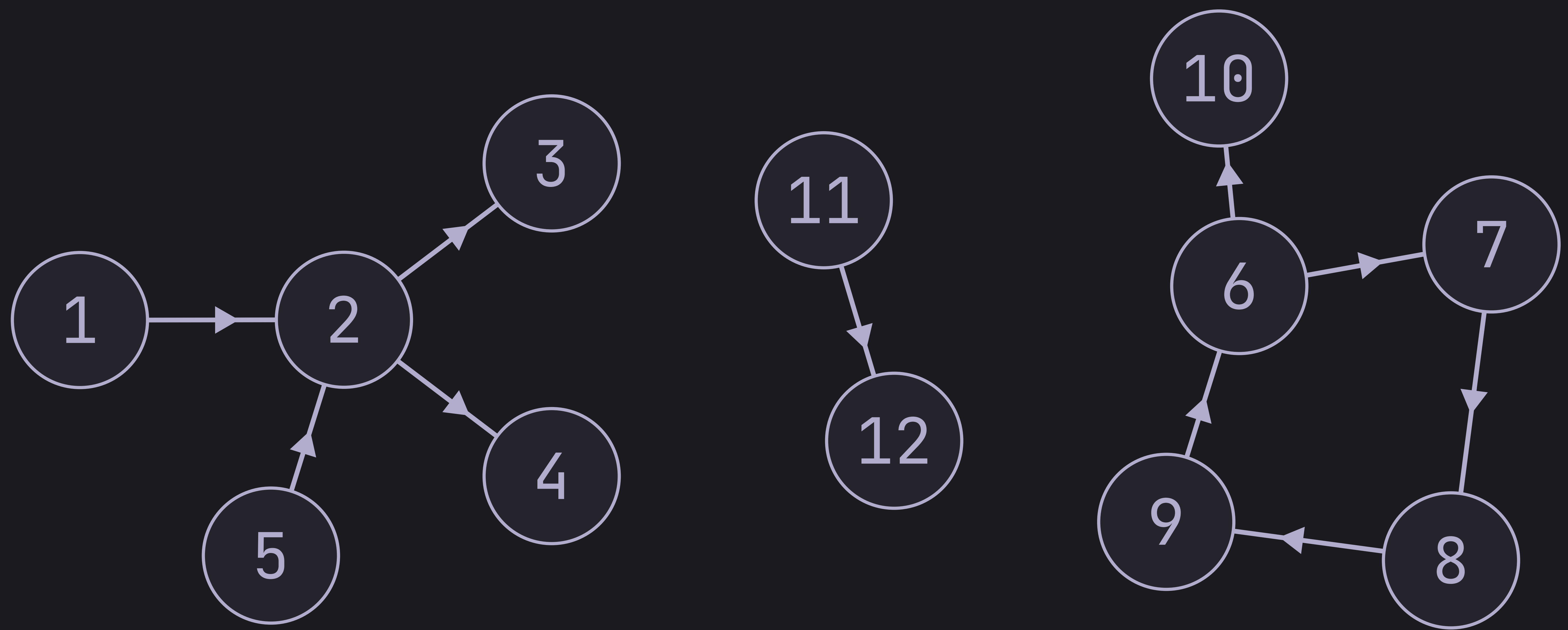
Finally we need to handle the case if we don’t find a cycle

Detecting Cycles



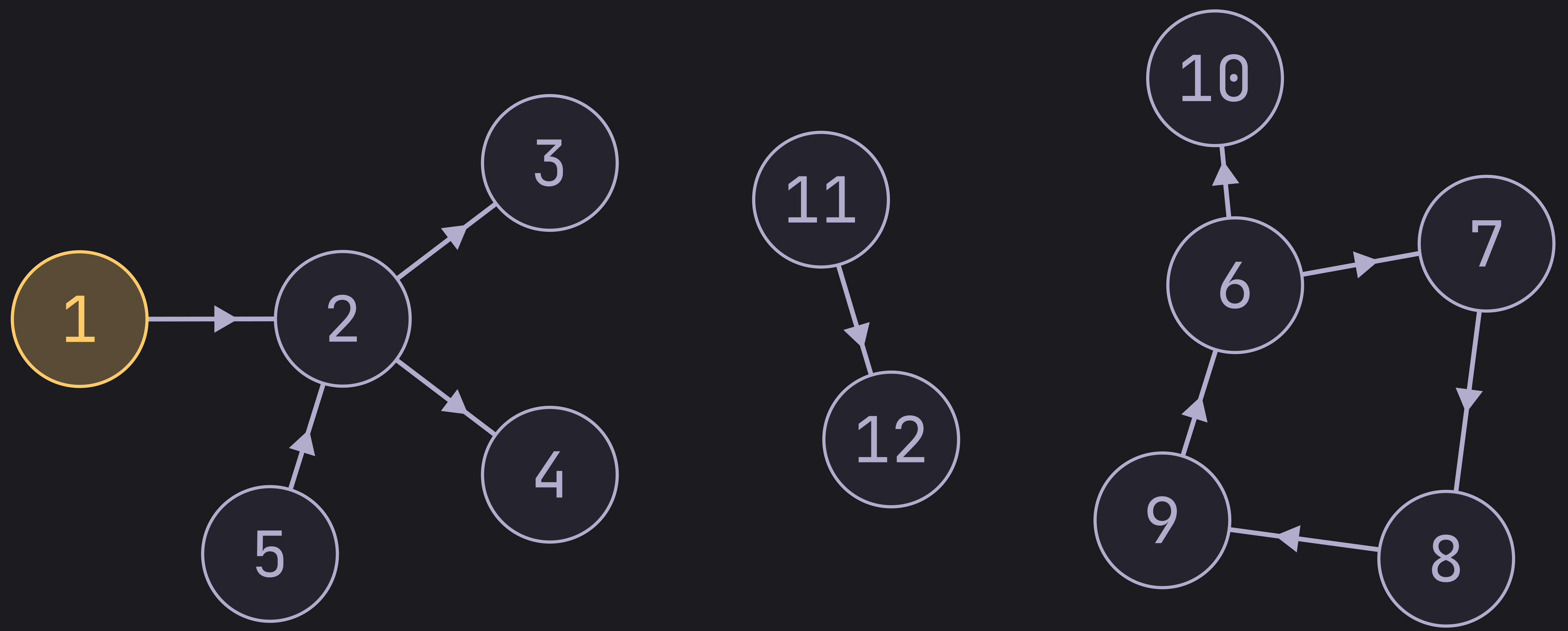
So far this code *kinda* works but we have some problems
Can anyone see the two problems that we will run into?

Detecting Cycles



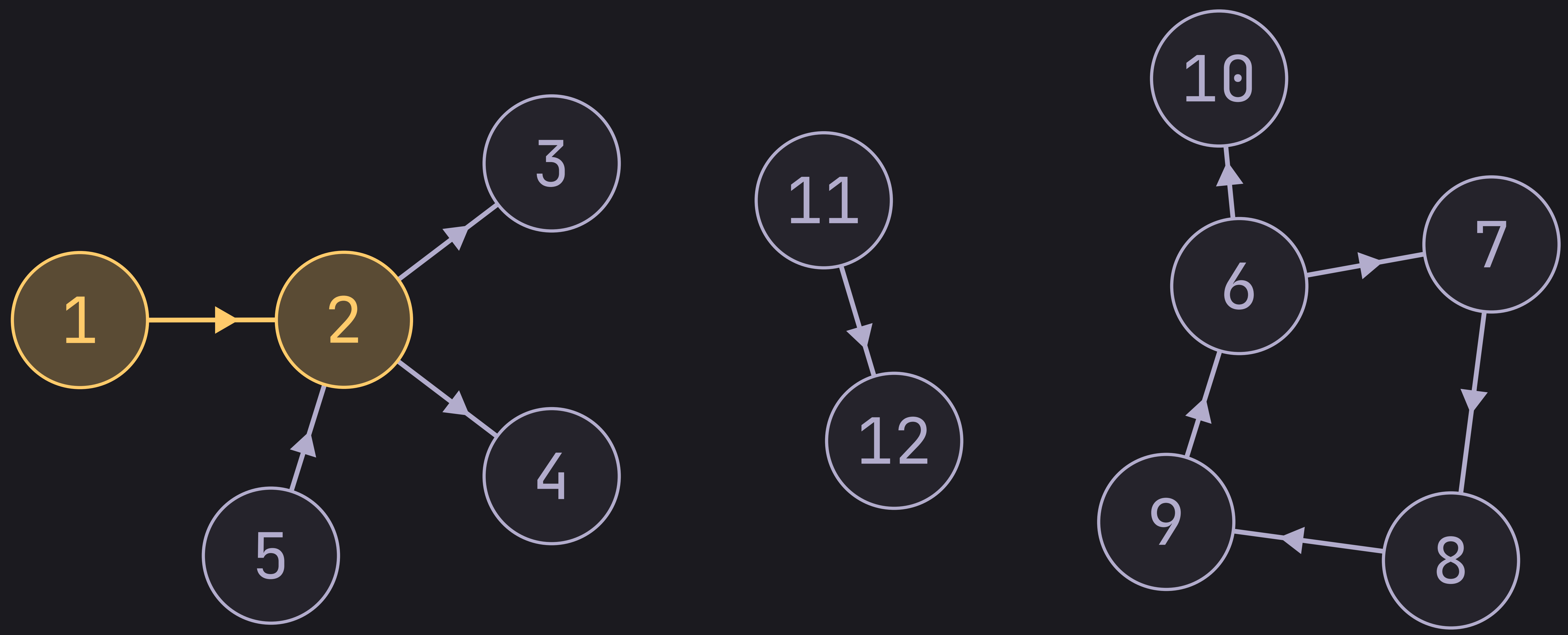
Suppose we start with dfs(1)

Detecting Cycles



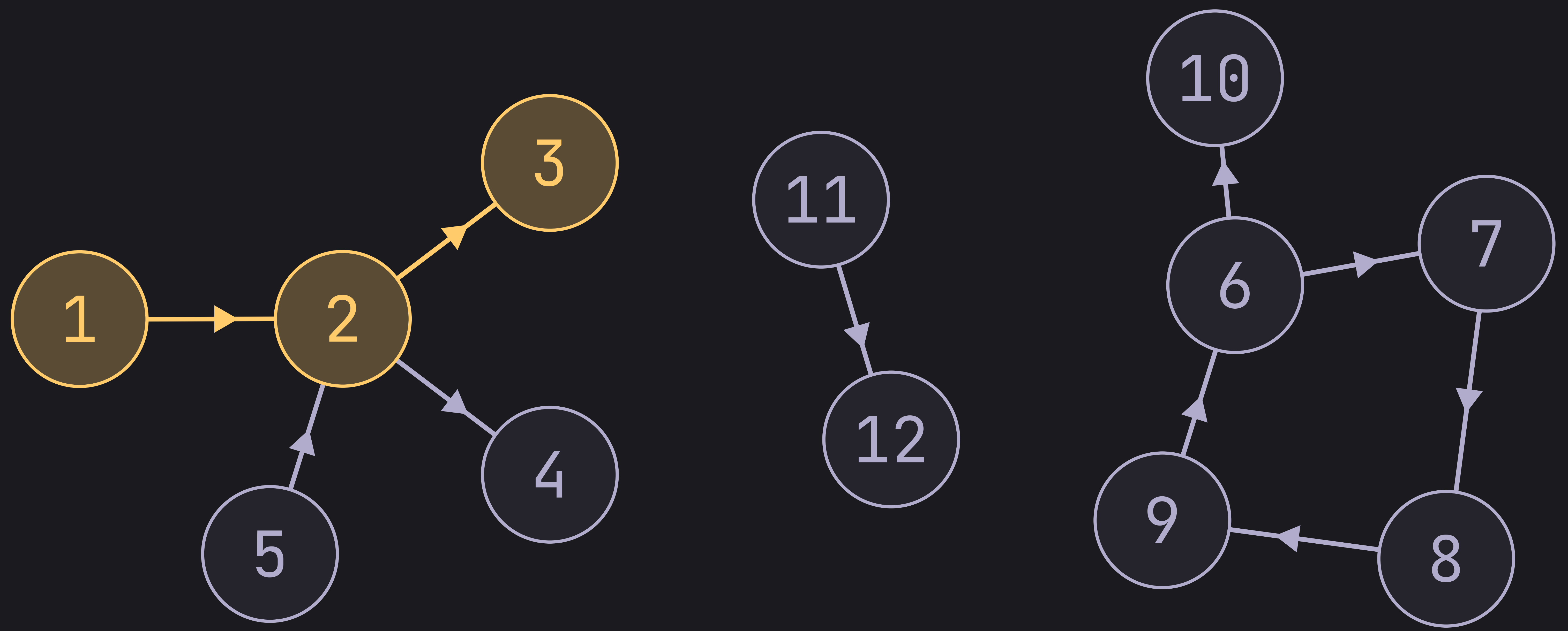
Suppose we start with `dfs(1)`

Detecting Cycles



Suppose we start with dfs(1)

Detecting Cycles

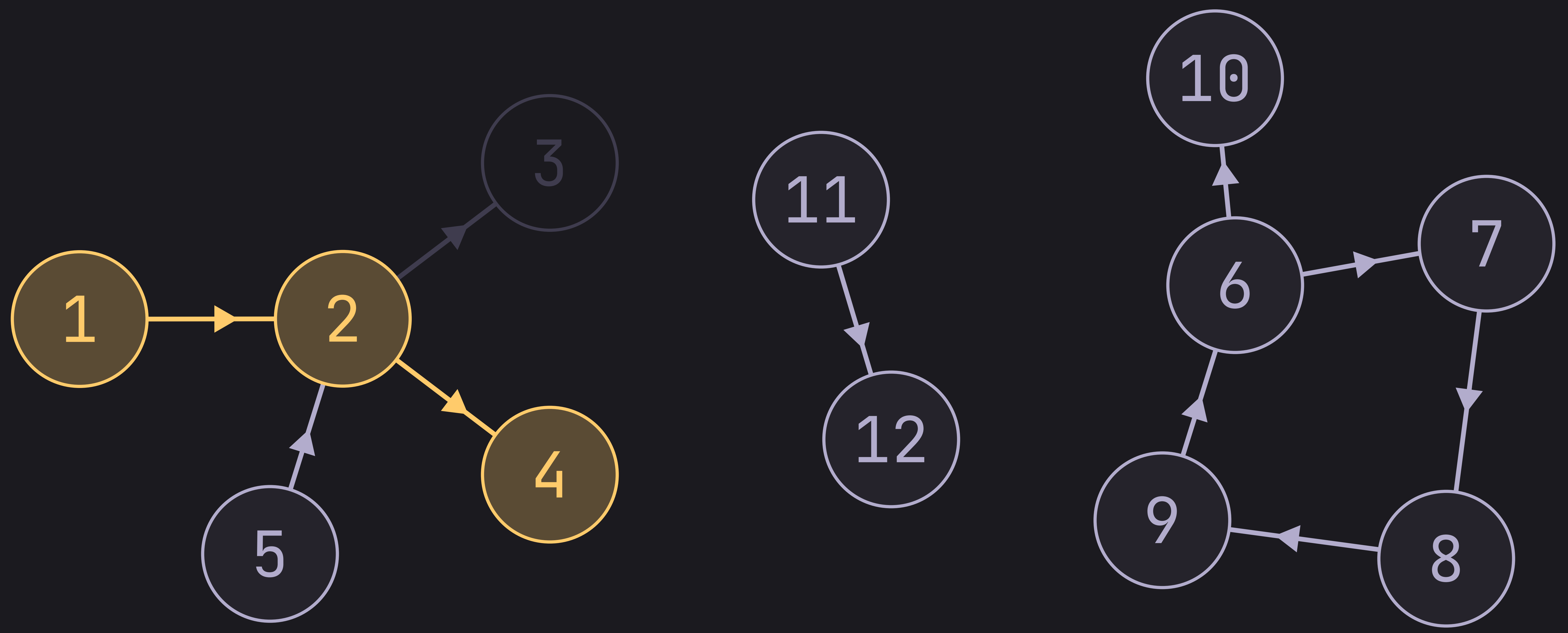


Suppose we start with dfs(1)

Detecting Cycles

Suppose we start with `dfs(1)`

Detecting Cycles



Suppose we start with `dfs(1)`

Detecting Cycles

Suppose we start with `dfs(1)`

Detecting Cycles

Suppose we start with `dfs(1)`

Detecting Cycles

Well... the problem is there are still a lot of unexplored vertices ☹️

Detecting Cycles

Luckily the solution is simple, we just pick another unvisited vertex, and run the dfs algorithm on that vertex

Detecting Cycles

So lets try `dfs(5)` next

Detecting Cycles

So lets try `dfs(5)` next

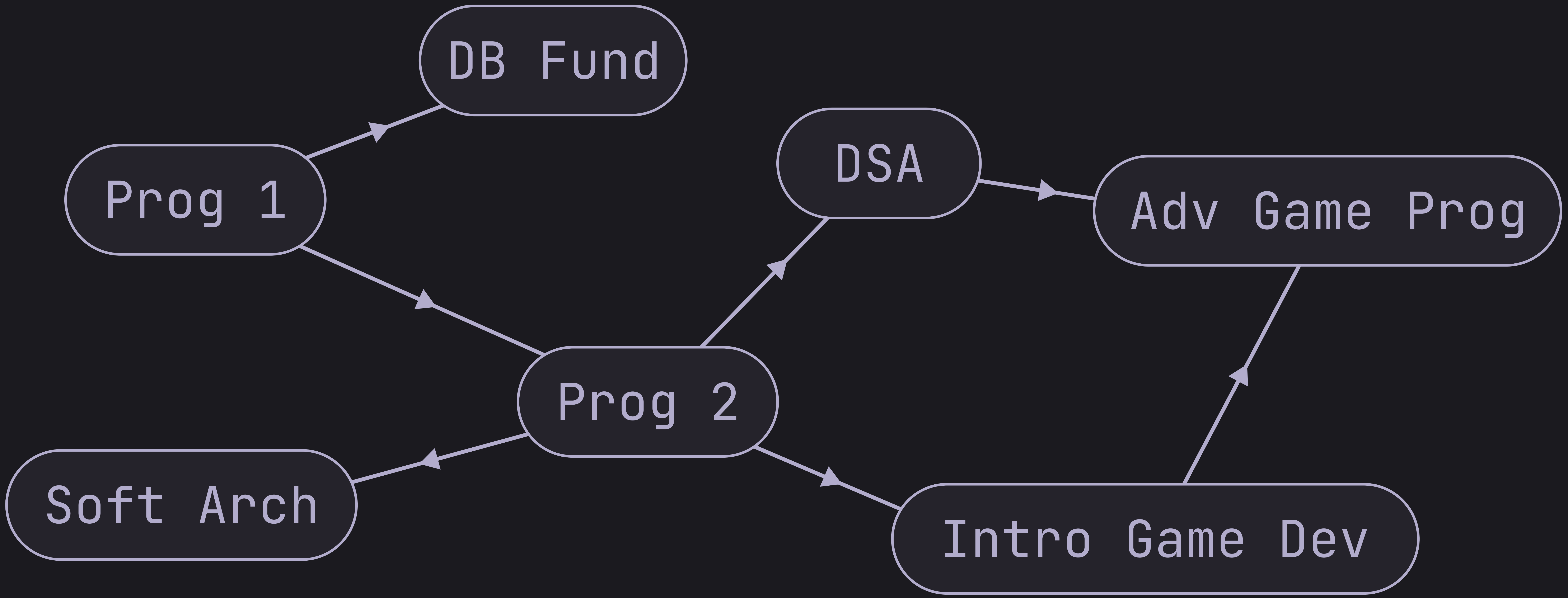
Detecting Cycles

And then we could do `dfs(11)` next which will visit all the vertices in that connected component

Detecting Cycles

And then we could do dfs(9) where we will detect a cycle

Topological Sort

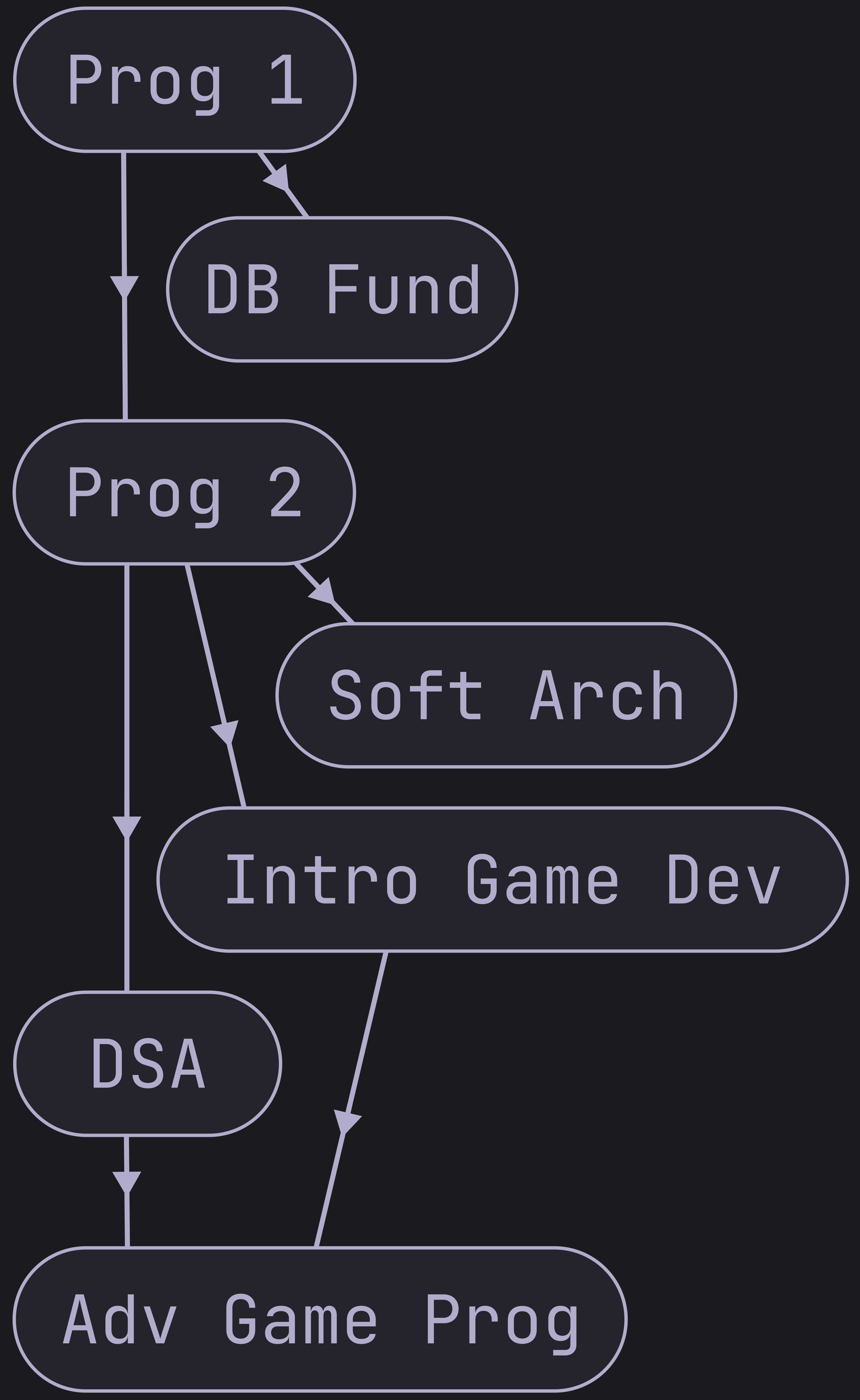


Suppose that you are trying to create a learning plan for your degree so that you know what order to take your subjects in

Topological Sort

You want to find some kind of **ordering** of your subjects so that you have done all the prerequisites by the time you take that class

Notice that all of the edges point in the same direction. This is called a topological sort

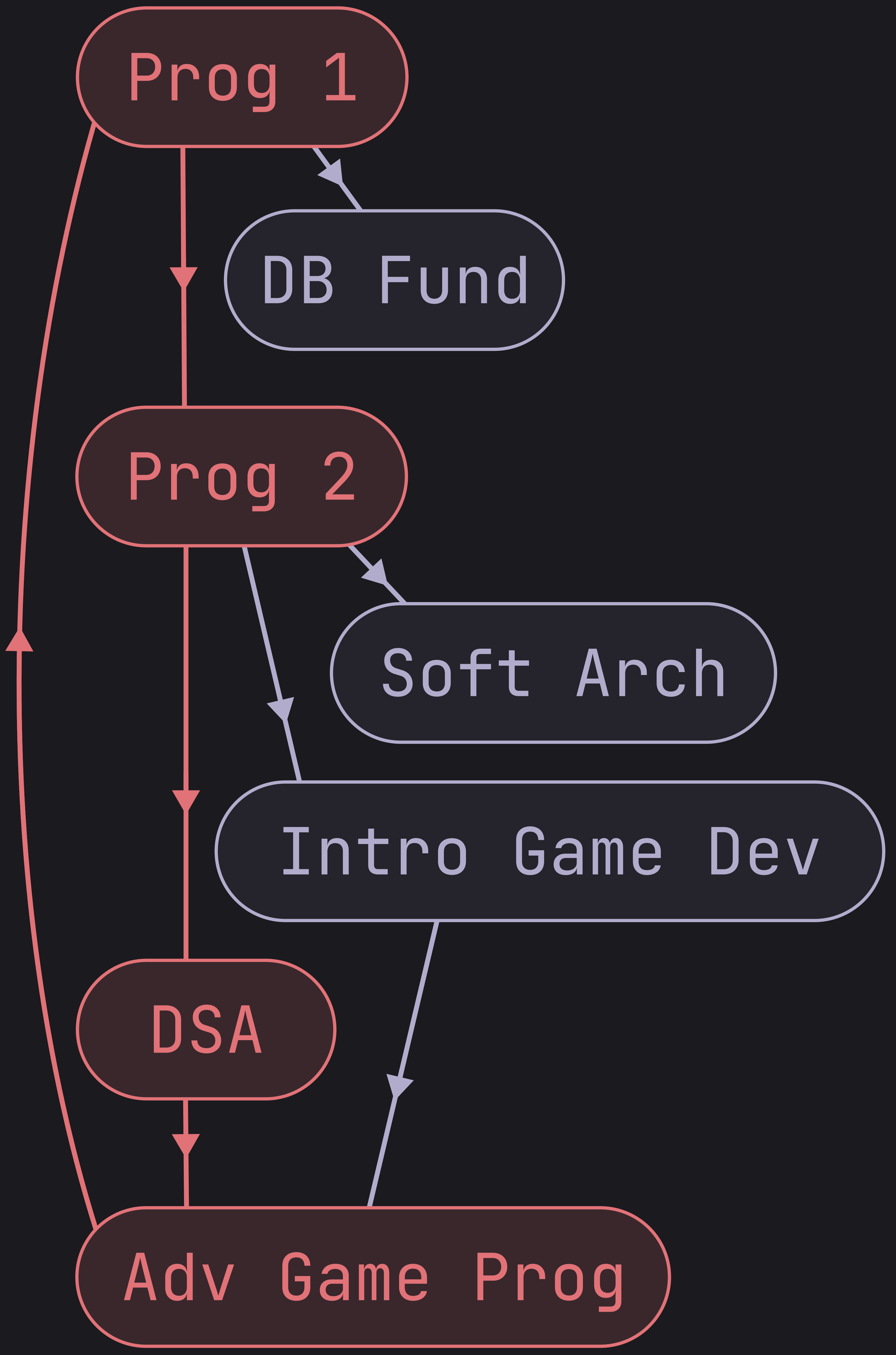


Topological Sort

Also notice that we want cannot create a topological ordering if there is a cycle in our graph.

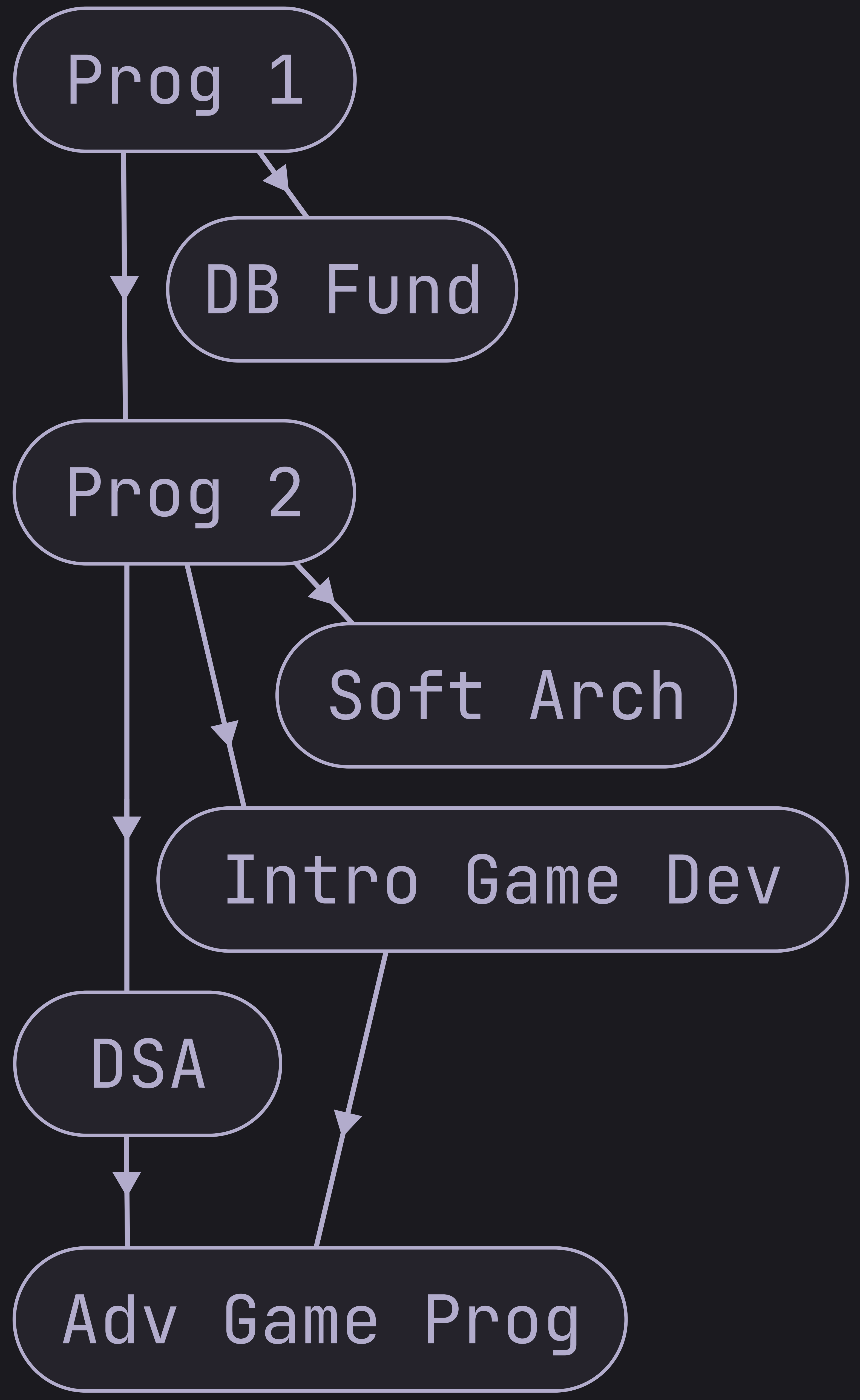
At least one edge will always point in the **wrong direction**

But the good news is, we just learned how to detect cycles!

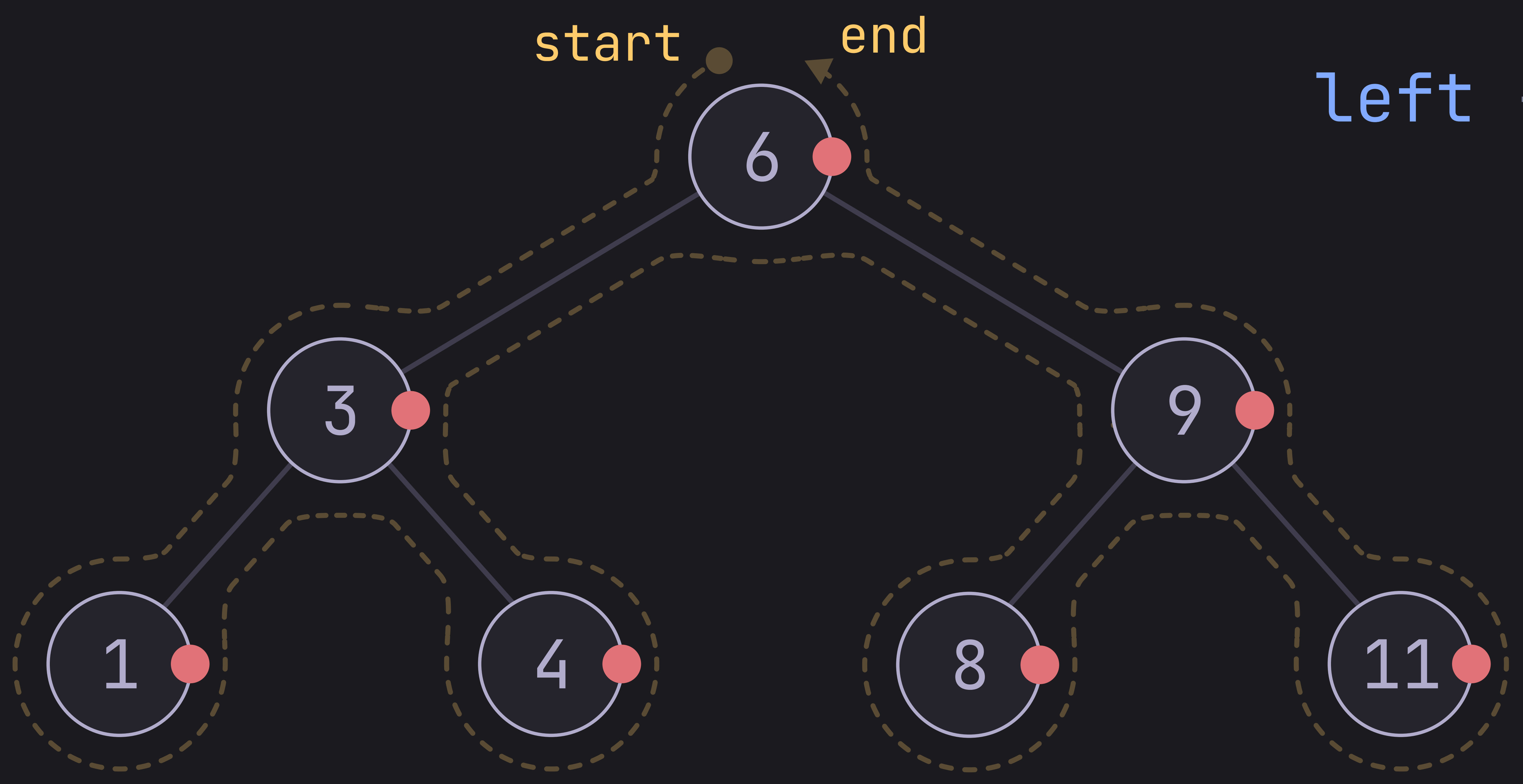


Topological Sort

Our task is to create an algorithm that can solve this problem efficiently for us!



Topological Sort



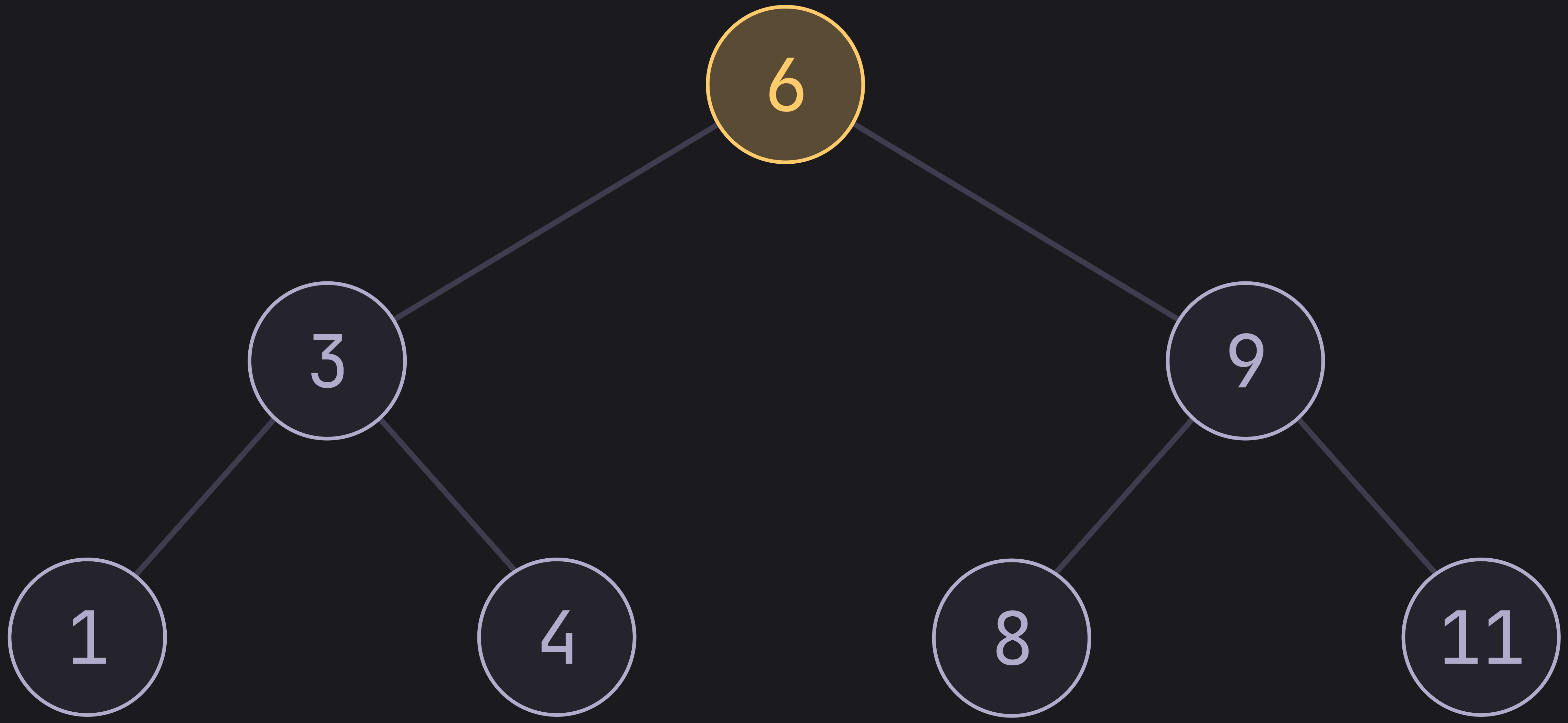
left → right → root

1, 4, 3, 8, 11, 9, 6

Postorder

Do you all remember postorder traversal?

Topological Sort



[6]

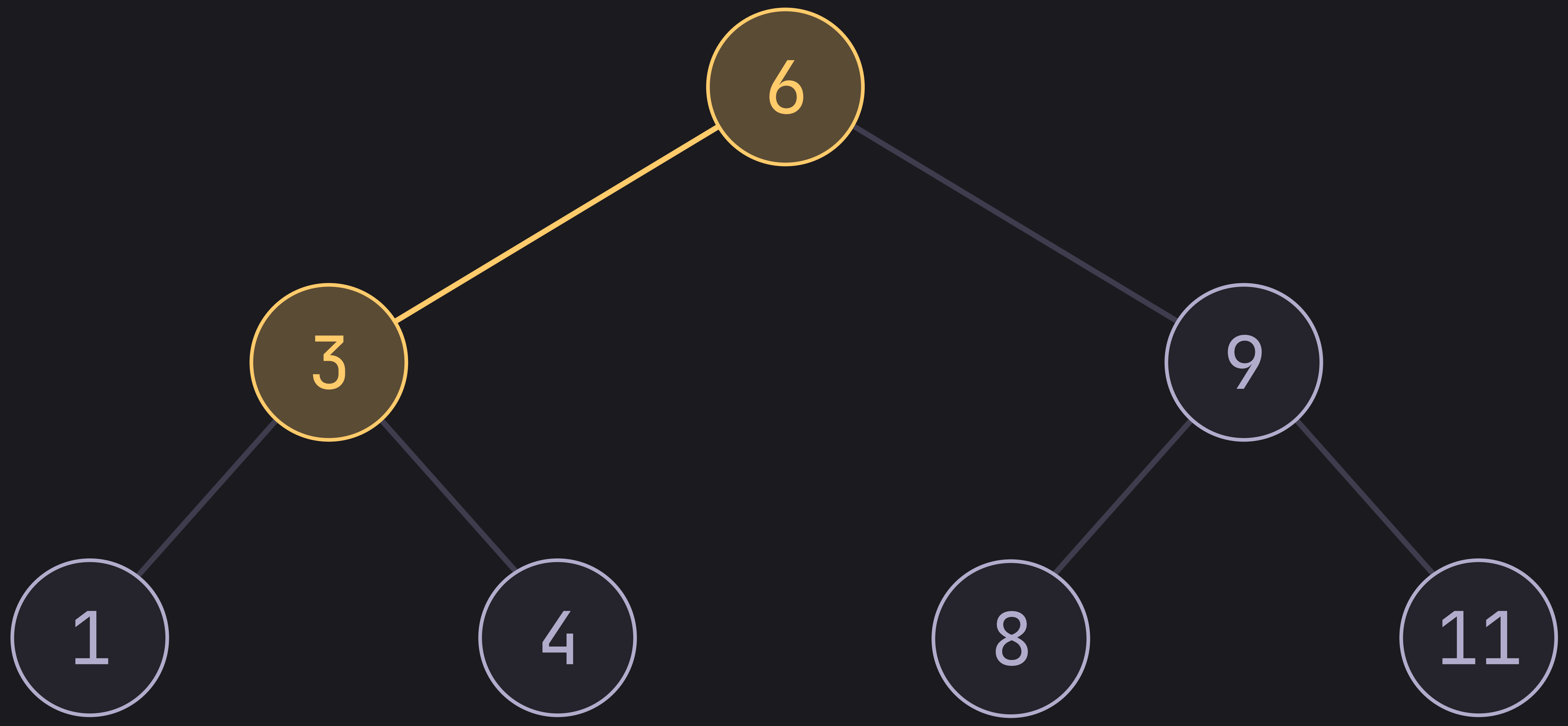
Stack

1, 4, 3, 8, 11, 9, 6

Post order

Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



[6, 3]

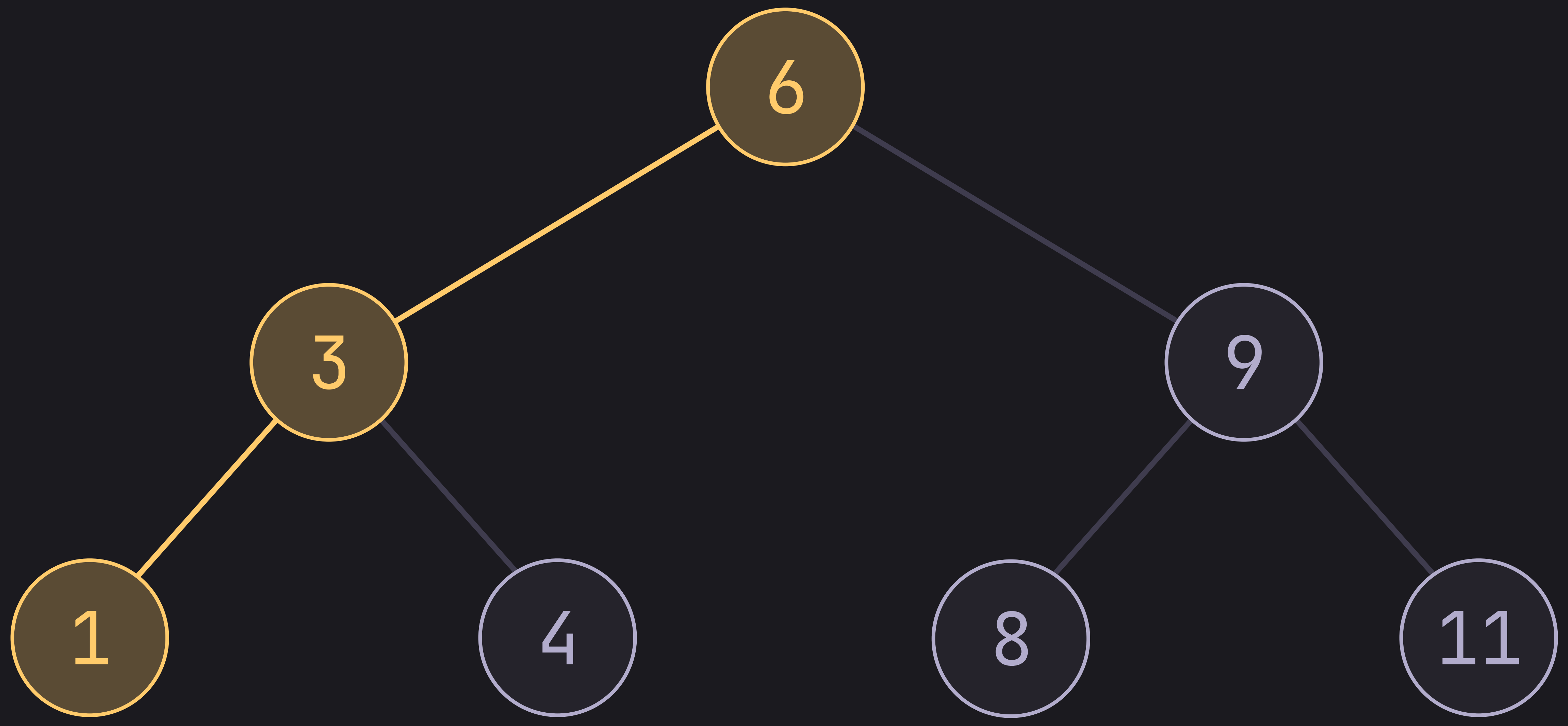
Stack

1, 4, 3, 8, 11, 9, 6

Post order

Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



[6, 3, 1]

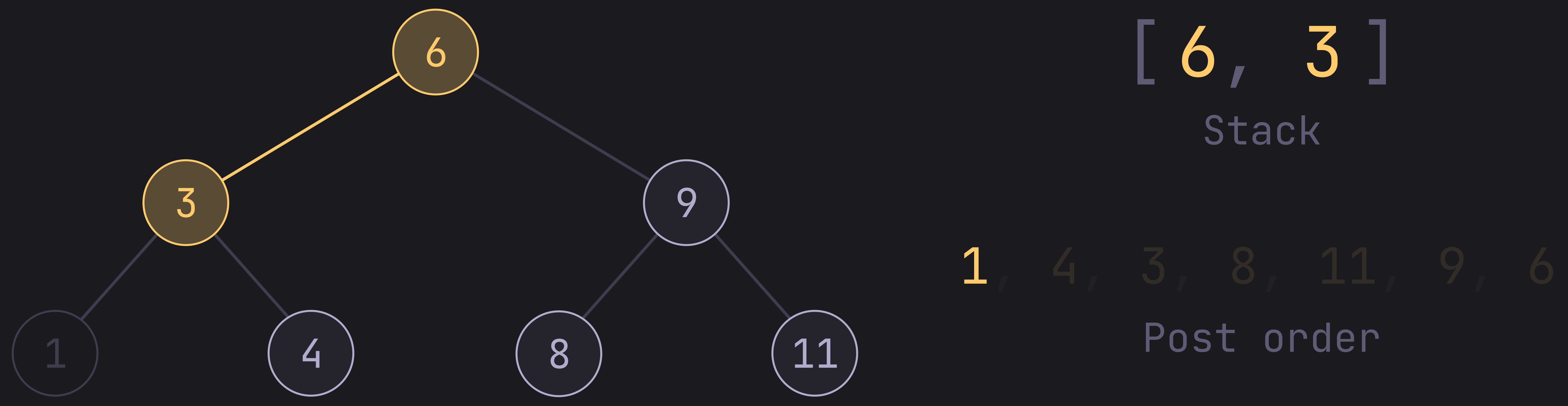
Stack

1, 4, 3, 8, 11, 9, 6

Post order

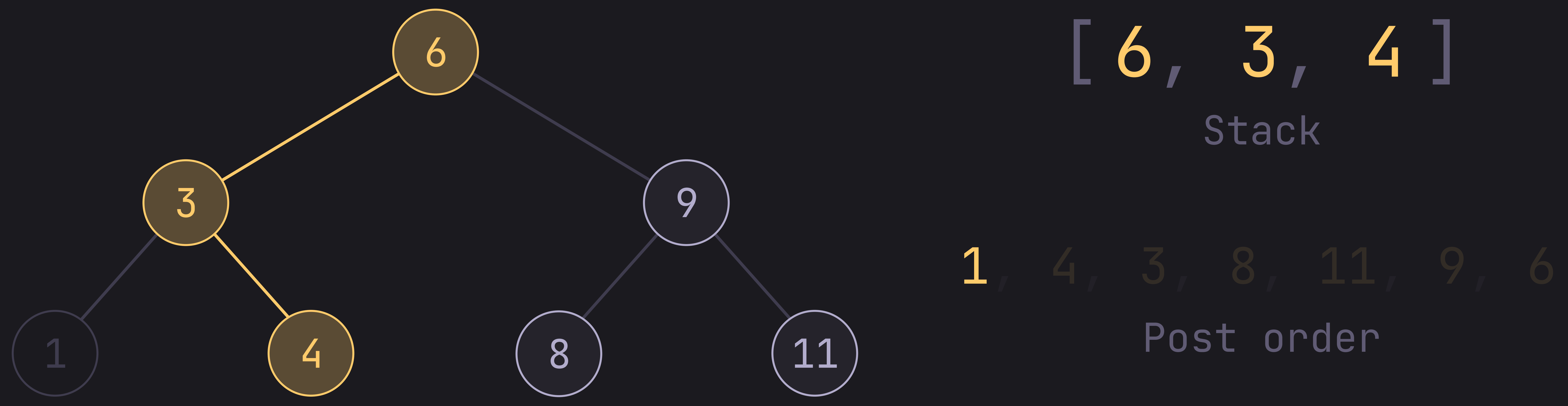
Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



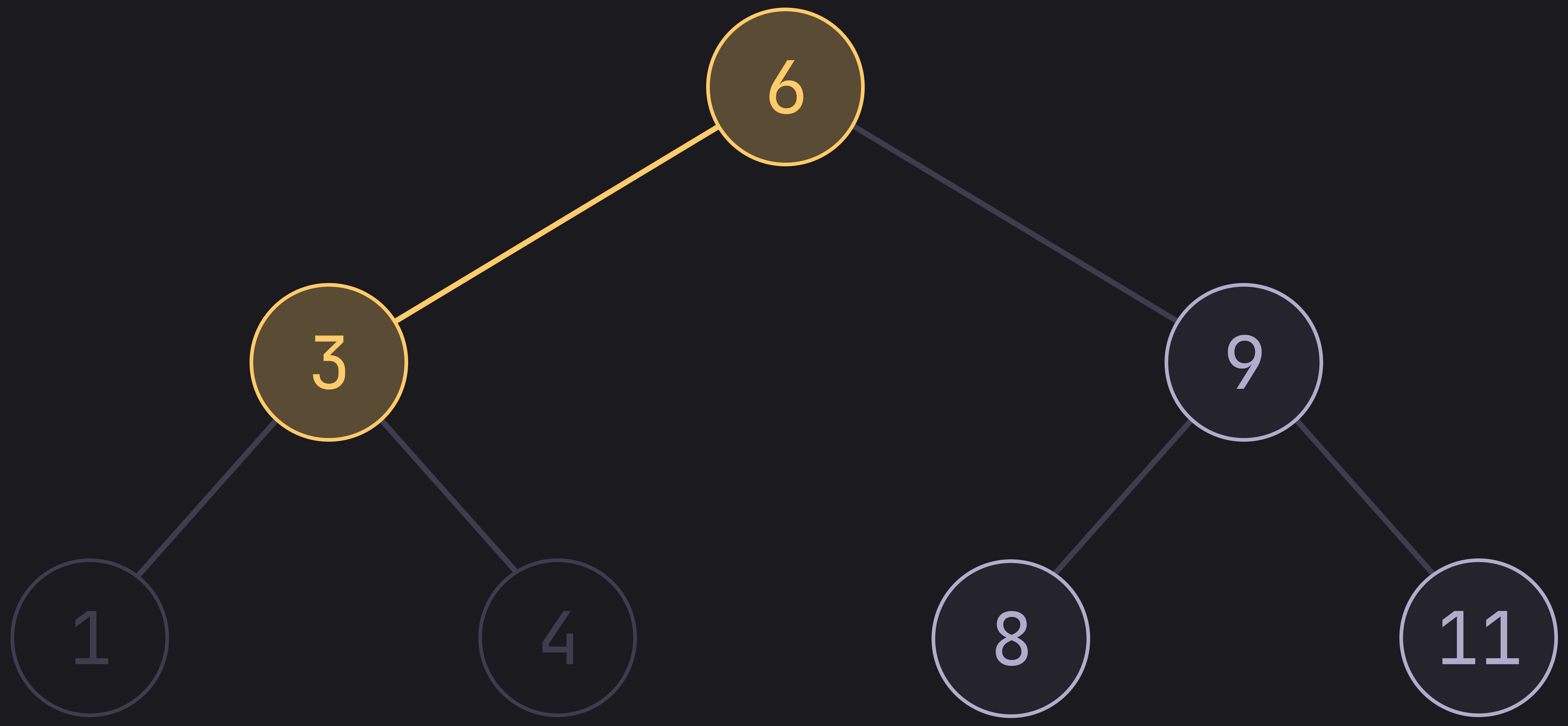
Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



[6, 3]

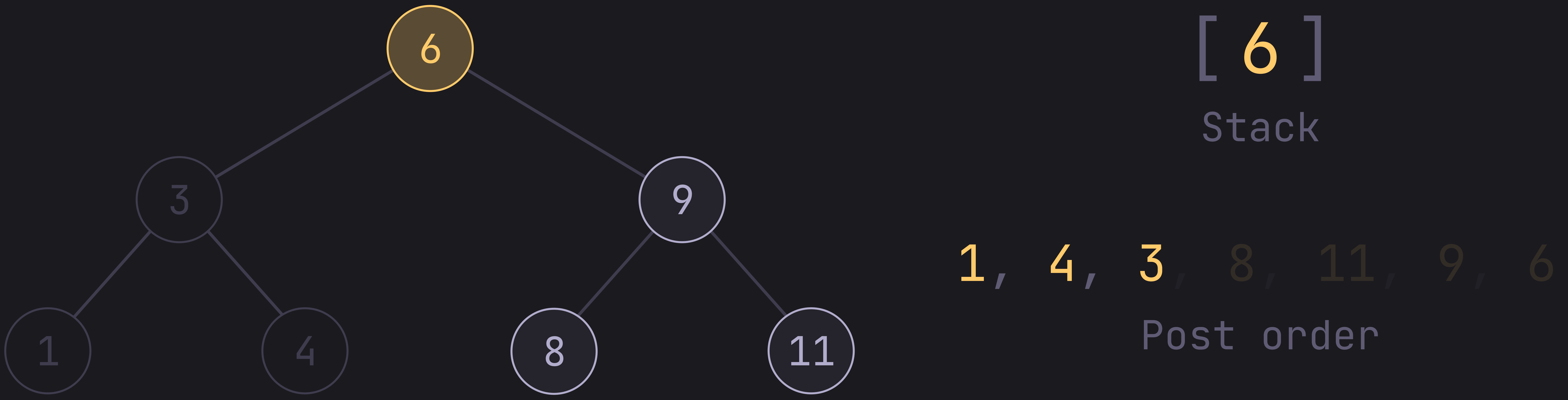
Stack

1, 4, 3, 8, 11, 9, 6

Post order

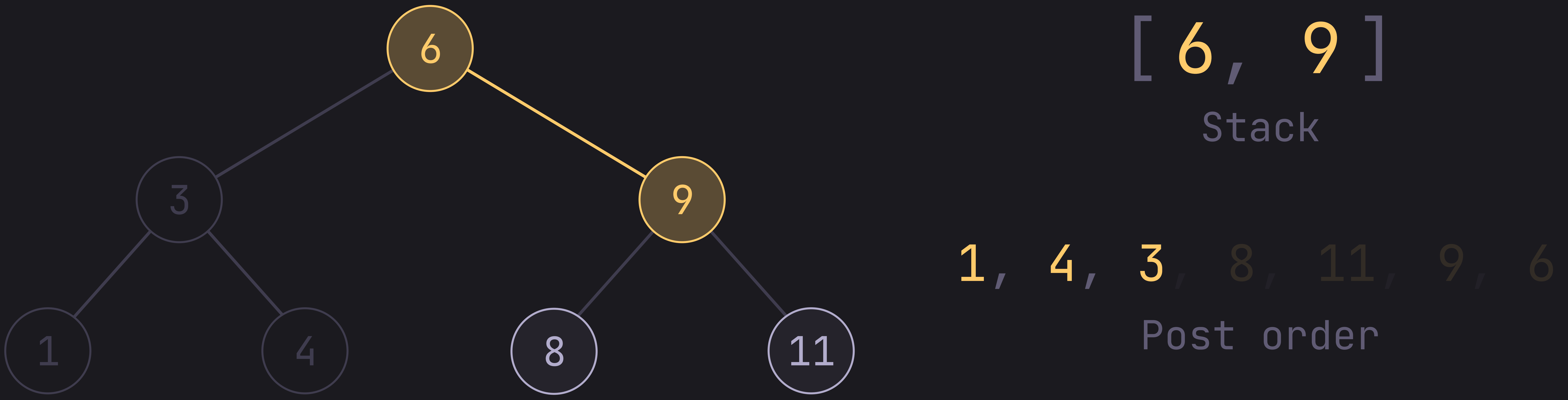
Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



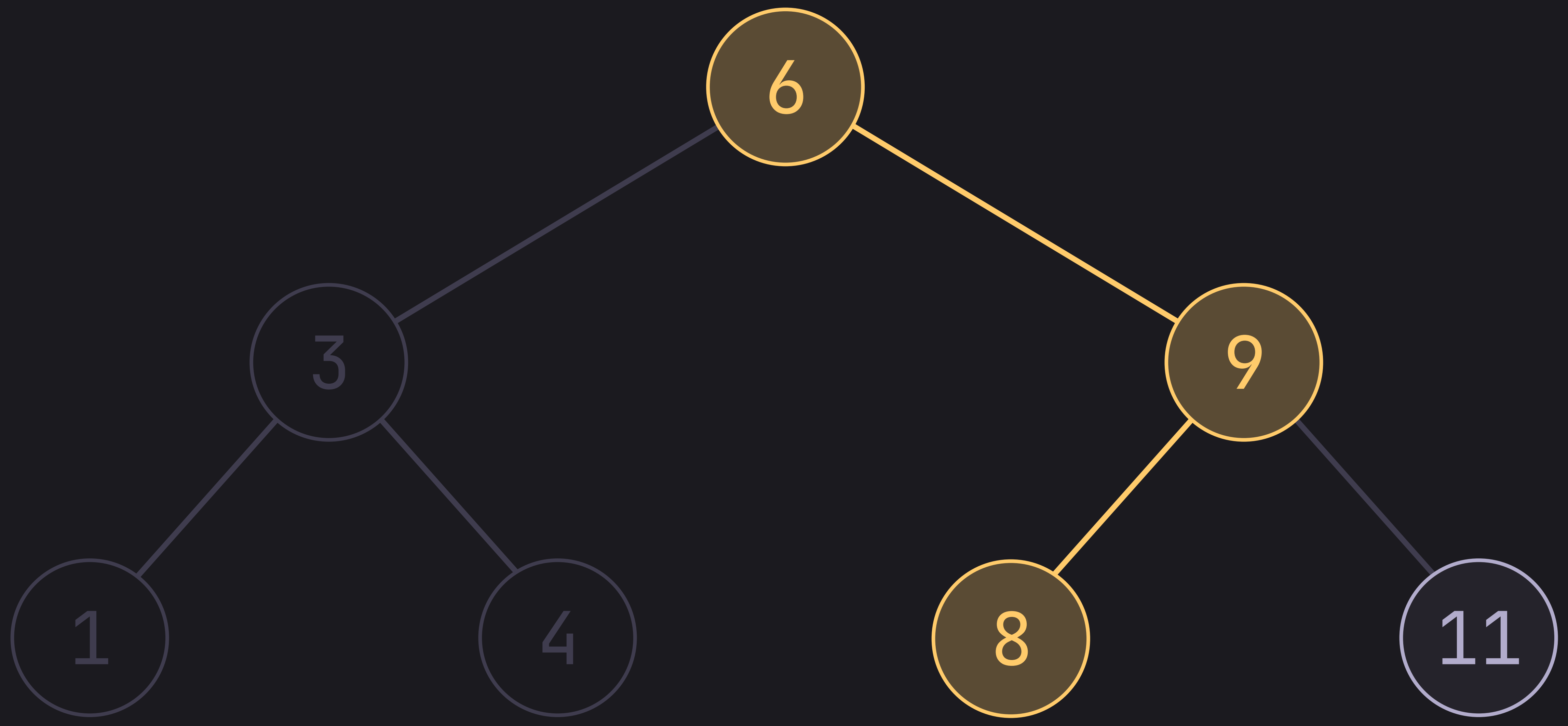
Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



[6, 9, 8]

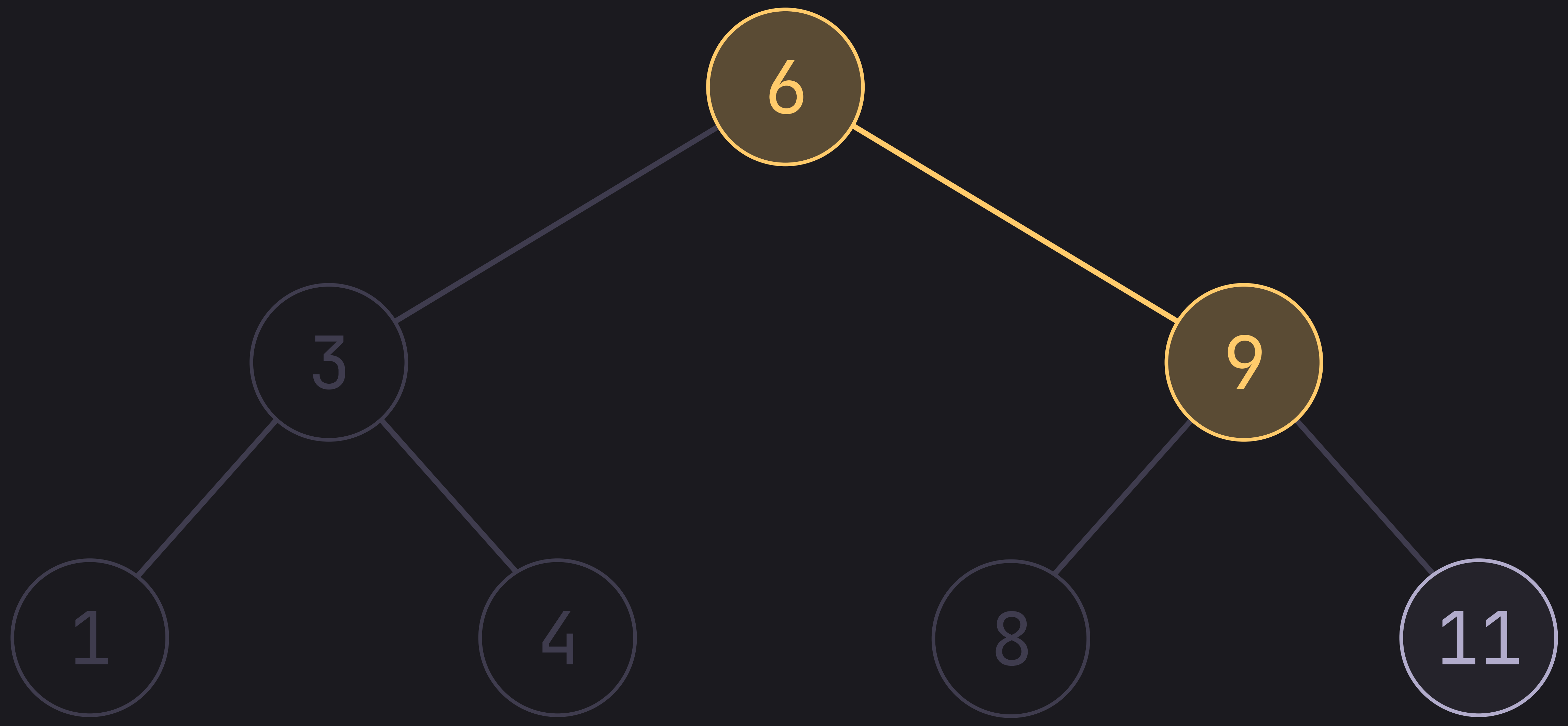
Stack

1, 4, 3, 8, 11, 9, 6

Post order

Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



[6, 9]

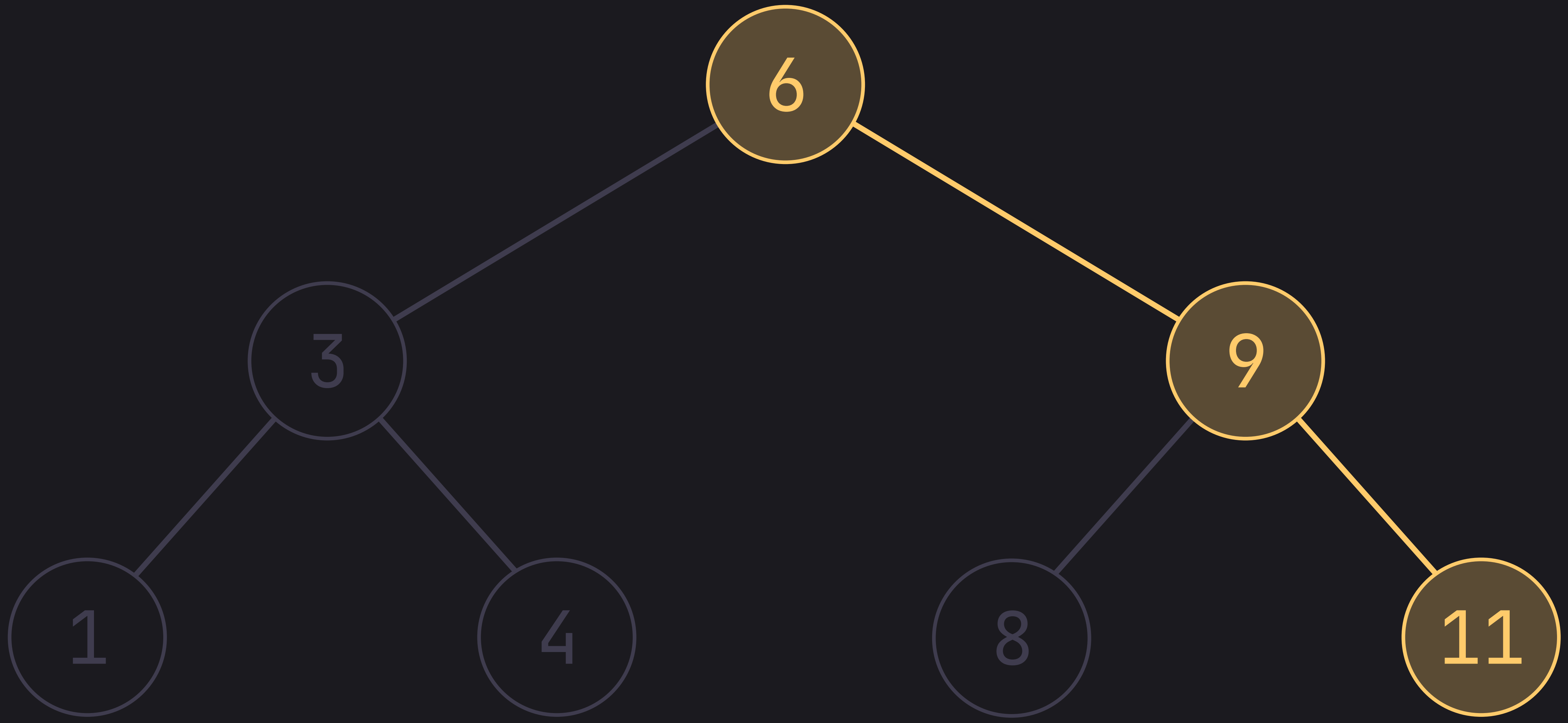
Stack

1, 4, 3, 8, 11, 9, 6

Post order

Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



[6, 9, 11]

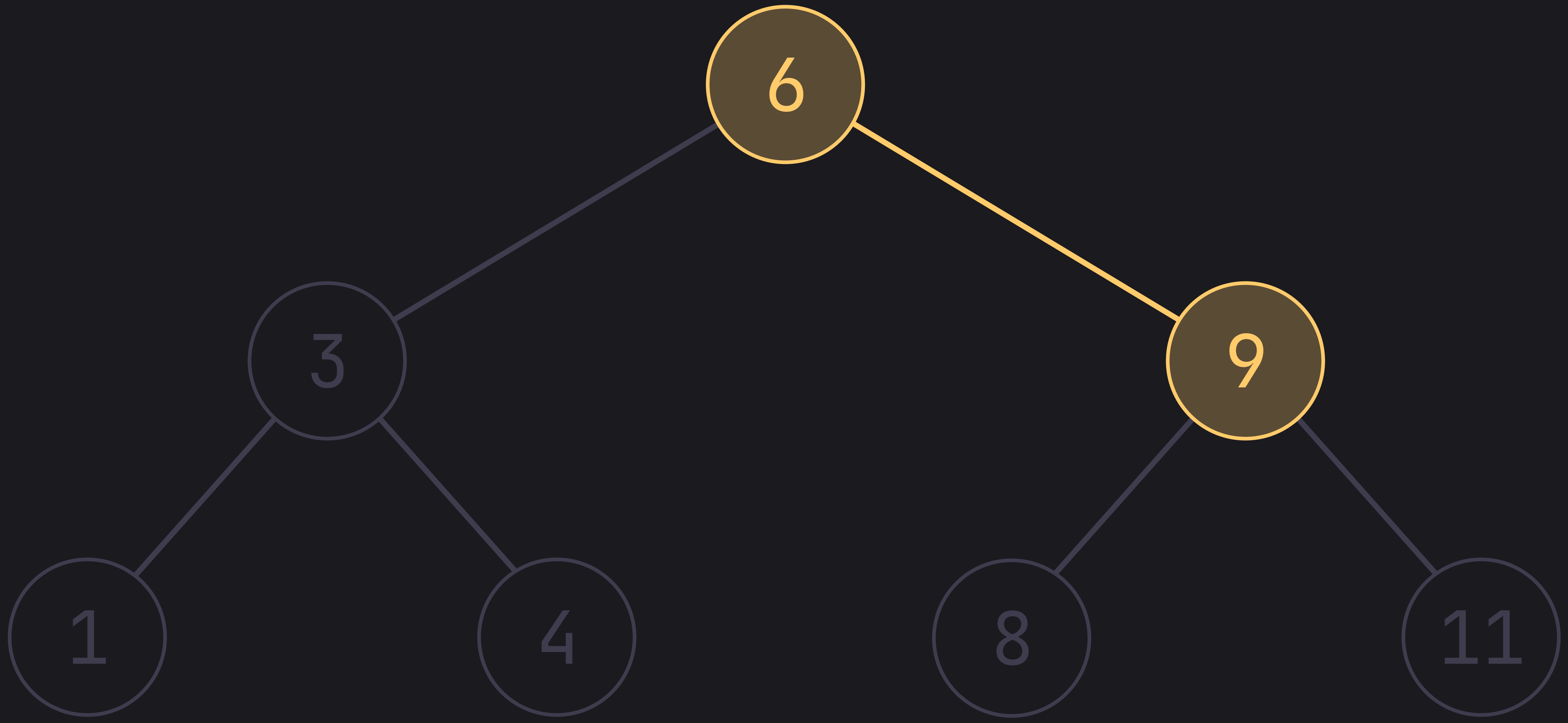
Stack

1, 4, 3, 8, 11, 9, 6

Post order

Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



[6, 9]

Stack

1, 4, 3, 8, 11, 9, 6

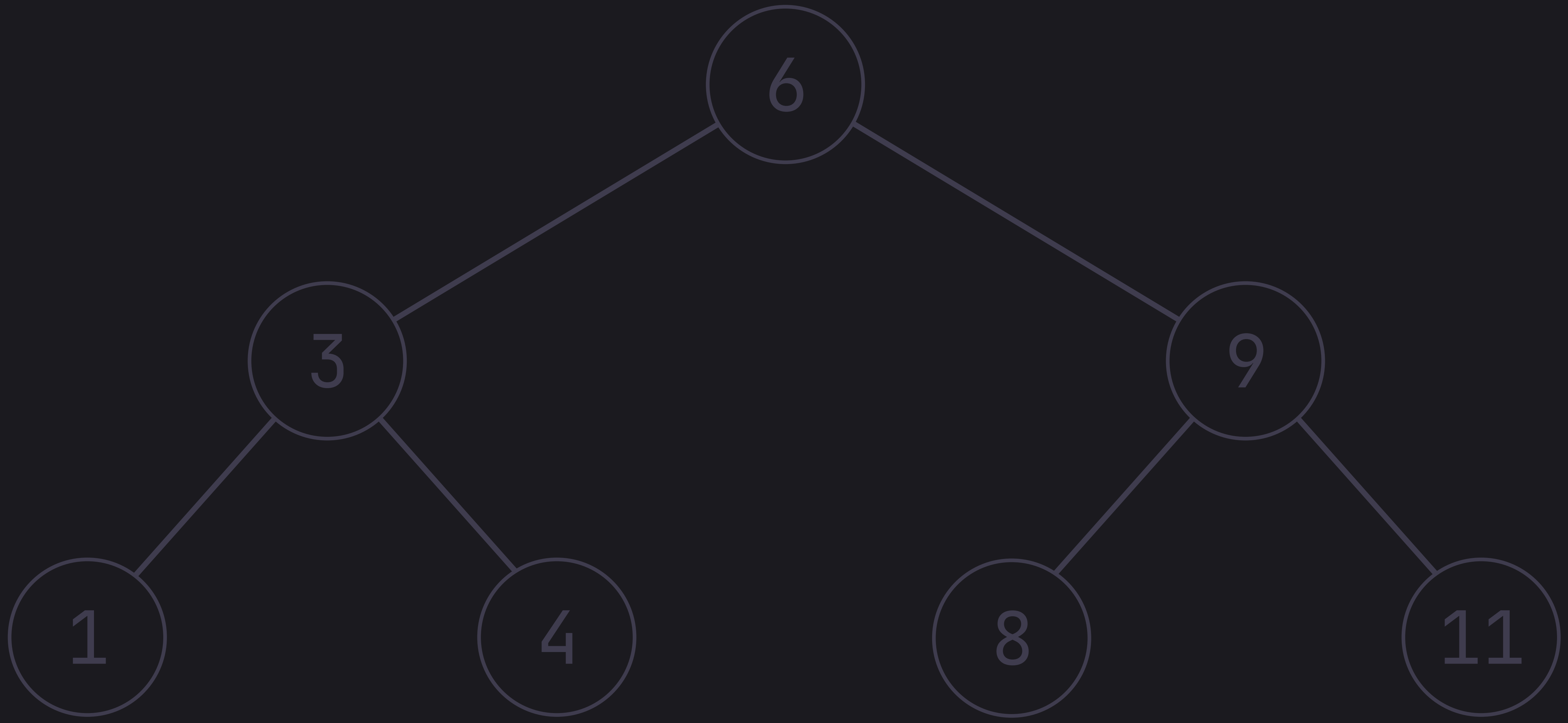
Post order

Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort

Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort



[ ]

Stack

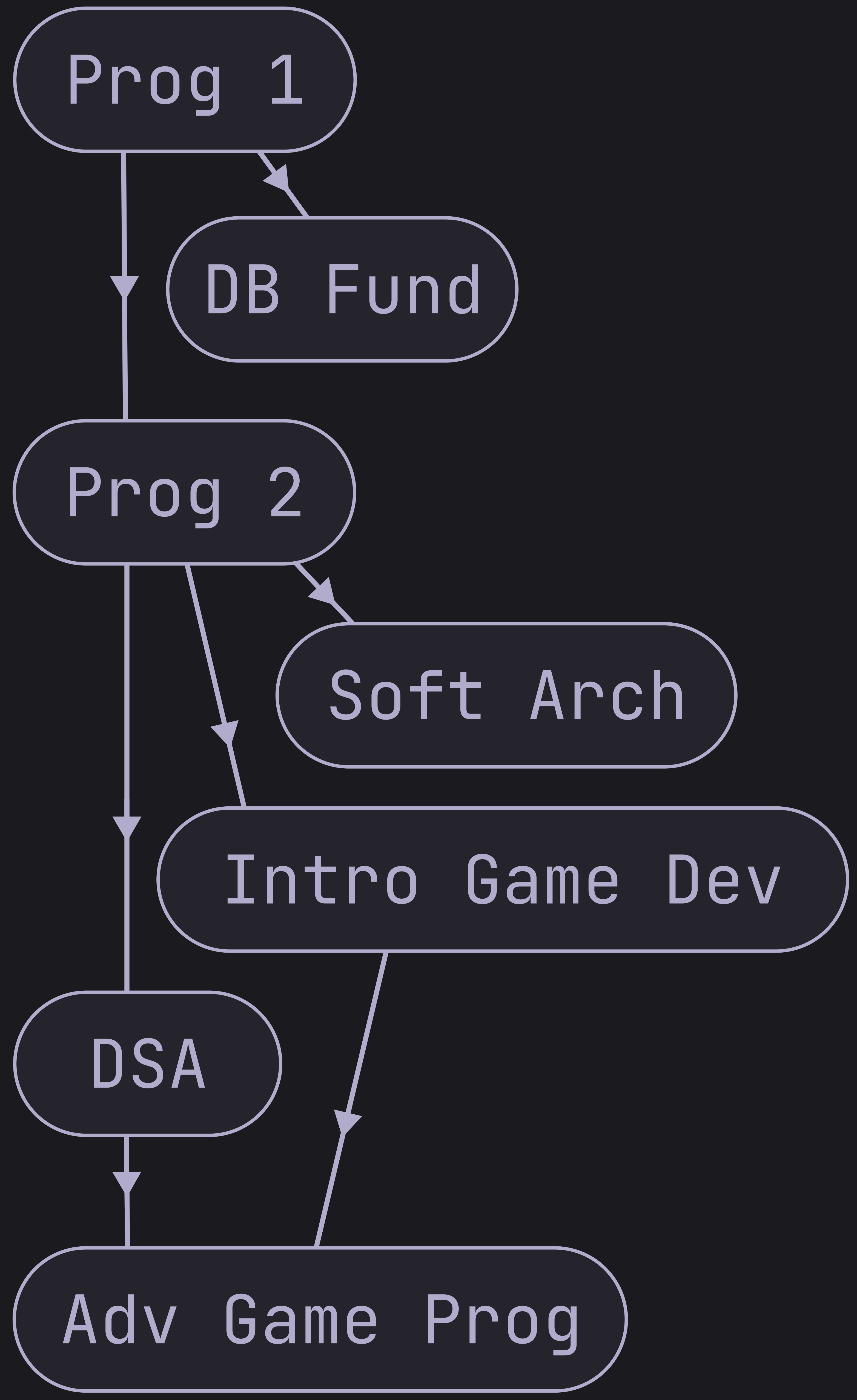
1, 4, 3, 8, 11, 9, 6

Post order

Well here is another way to think of it. If we perform a DFS, and we write down the vertex only when we remove it from our stack

Topological Sort

So let’s try apply a postorder traversal to our graph of subjects



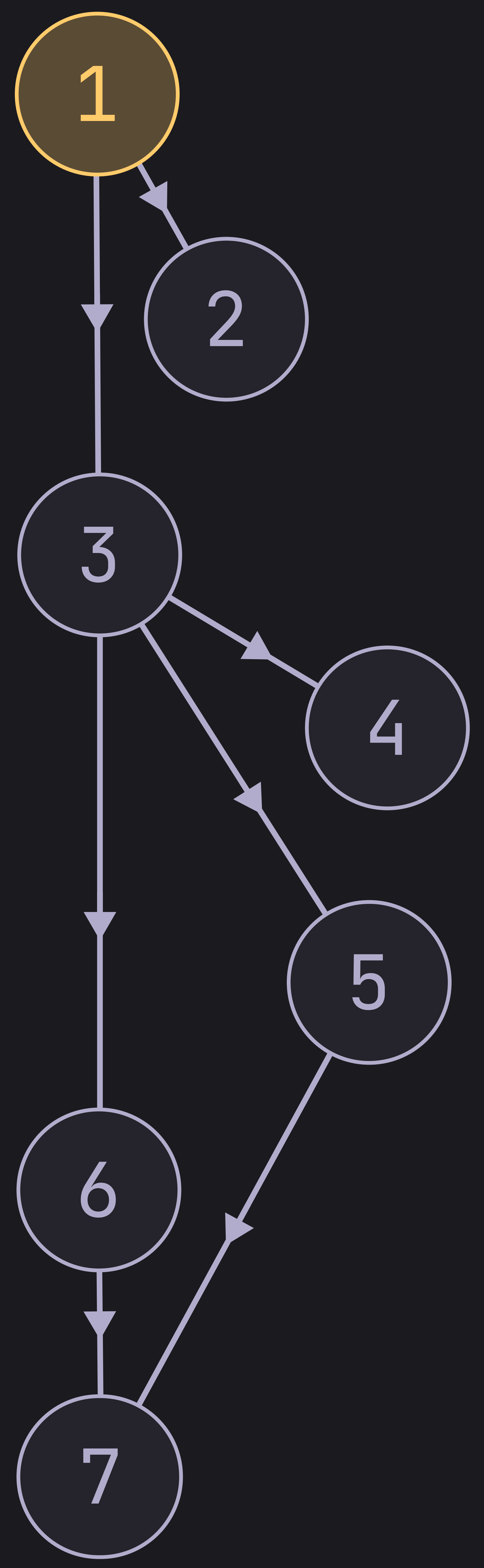
Topological Sort

So let's try apply a postorder traversal to our graph of subjects

[1]

Stack

Post order



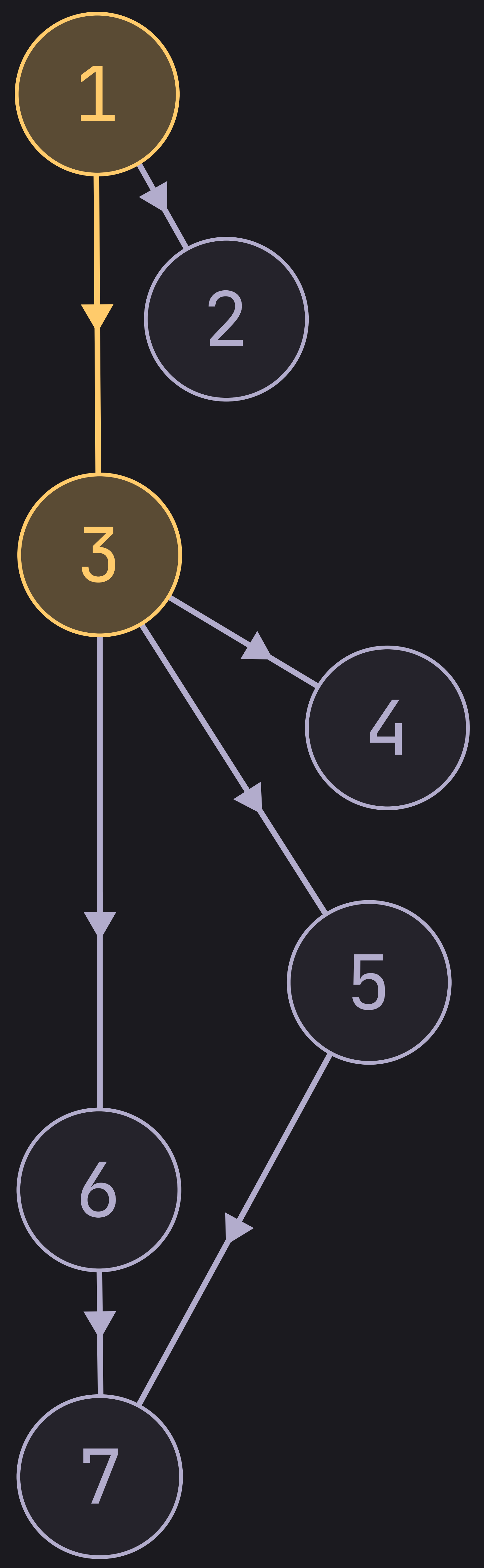
Topological Sort

So let's try apply a postorder traversal to our graph of subjects

[1, 3]

Stack

Post order



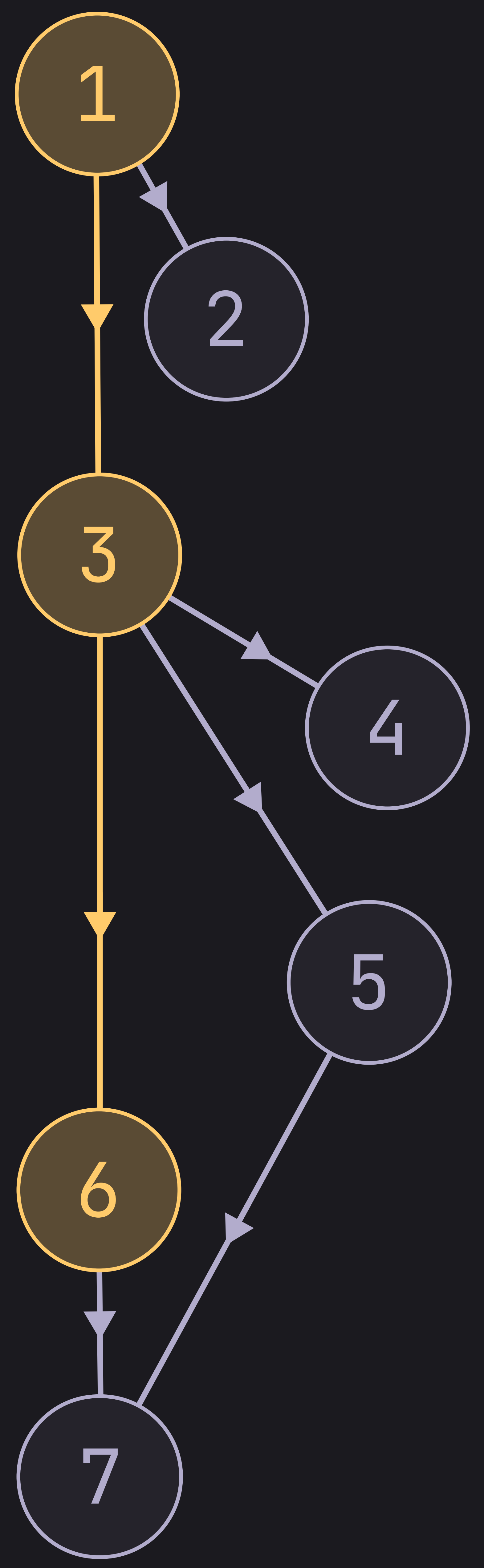
Topological Sort

So let's try apply a postorder traversal to our graph of subjects

[1, 3, 6]

Stack

Post order



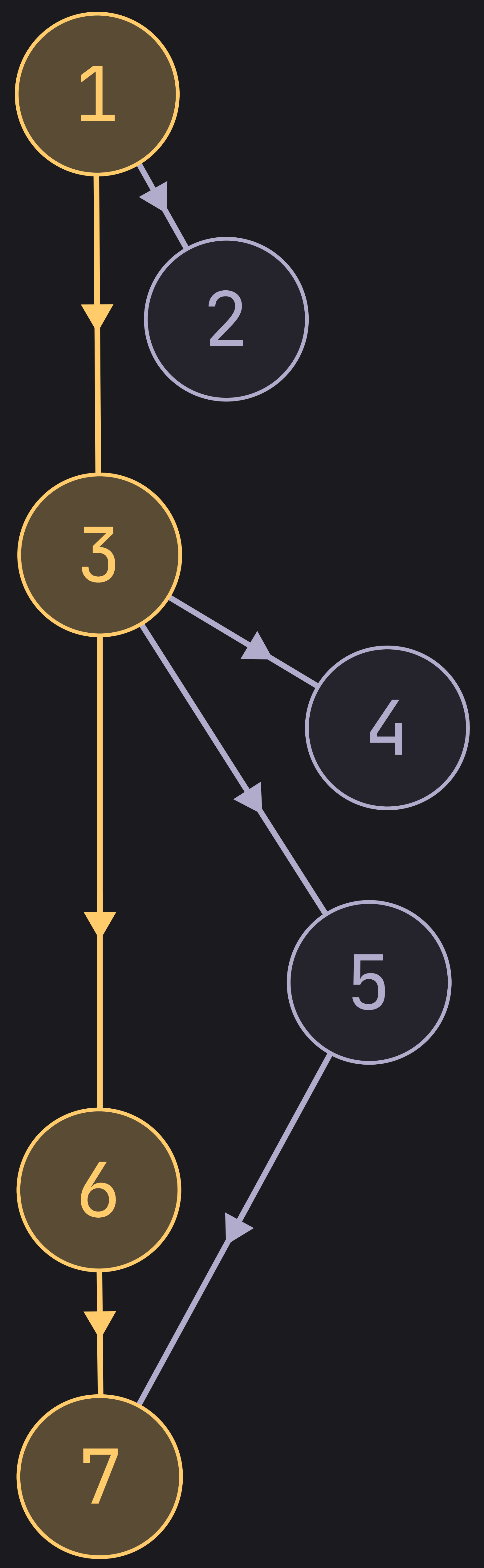
Topological Sort

So let's try apply a postorder traversal to our graph of subjects

[1, 3, 6, 7]

Stack

Post order



Topological Sort

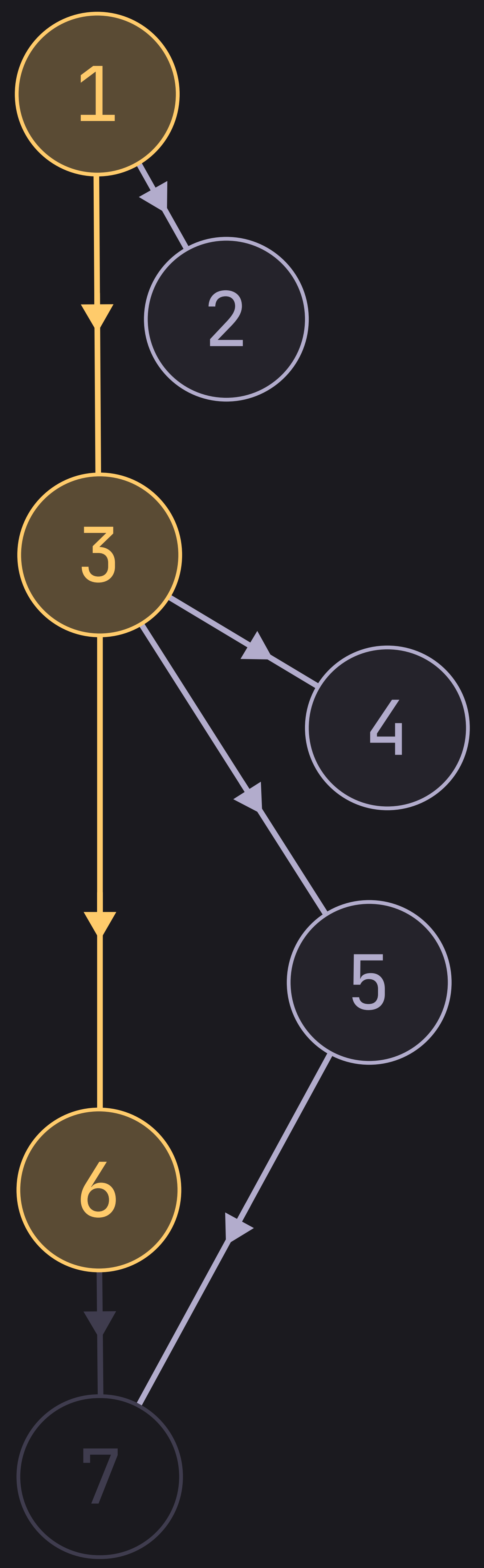
So let’s try apply a postorder traversal to our graph of subjects

[1, 3, 6]

Stack

7

Post order



Topological Sort

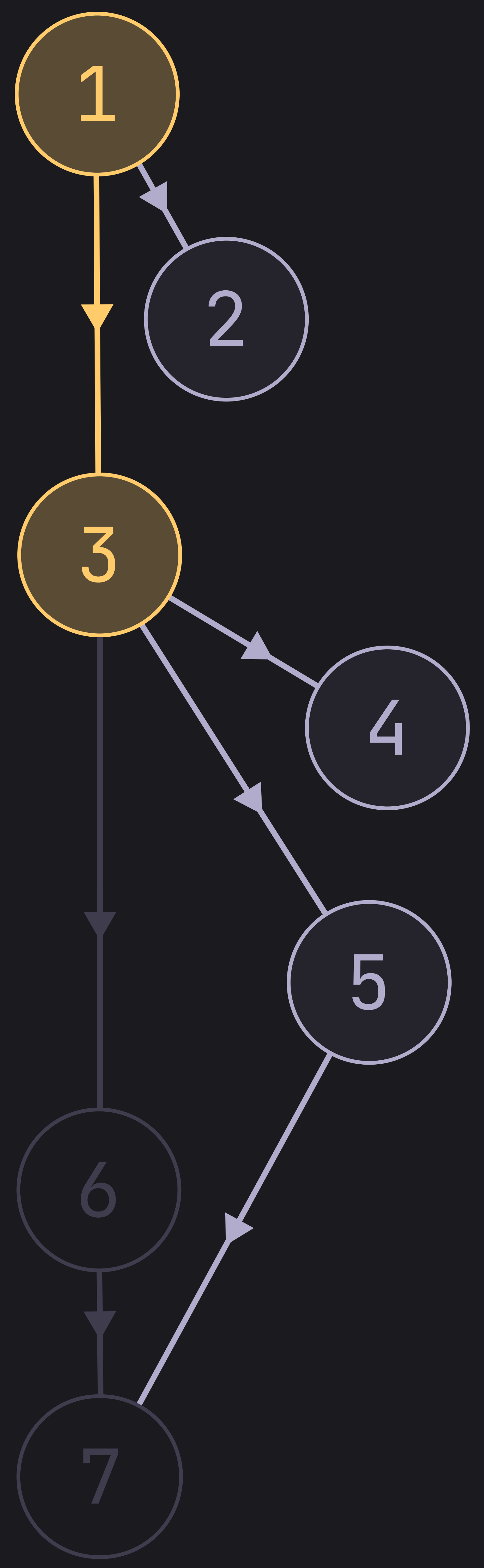
So let's try apply a postorder traversal to our graph of subjects

[1, 3]

Stack

7, 6

Post order



Topological Sort

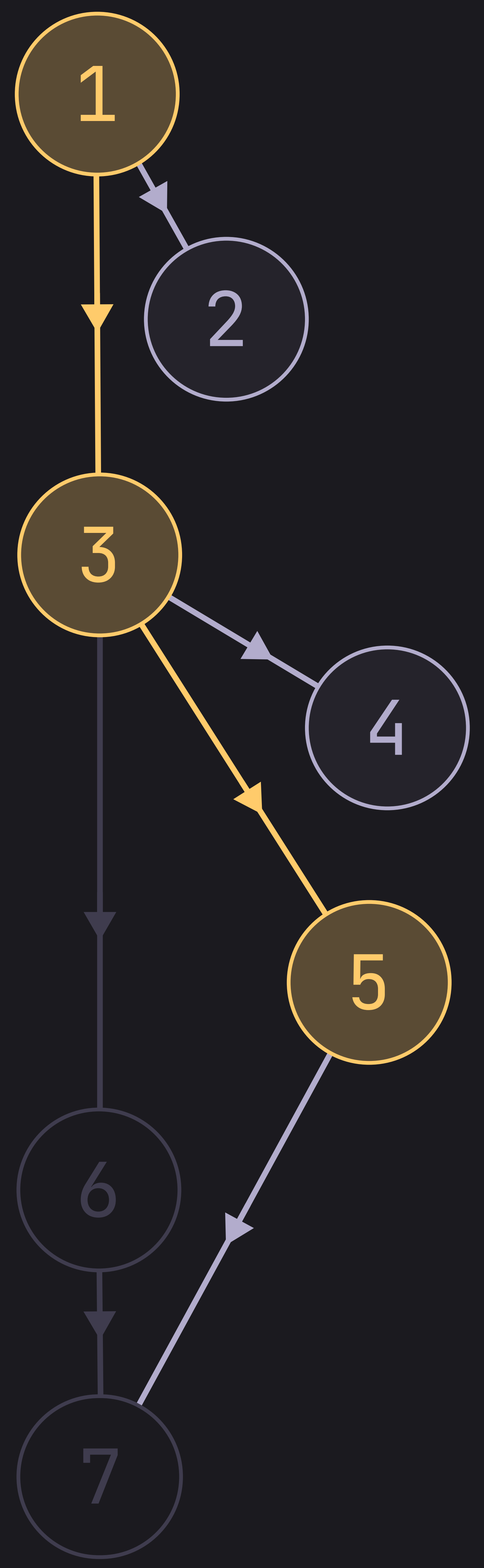
So let's try apply a postorder traversal to our graph of subjects

[1, 3, 5]

Stack

7, 6

Post order



Topological Sort

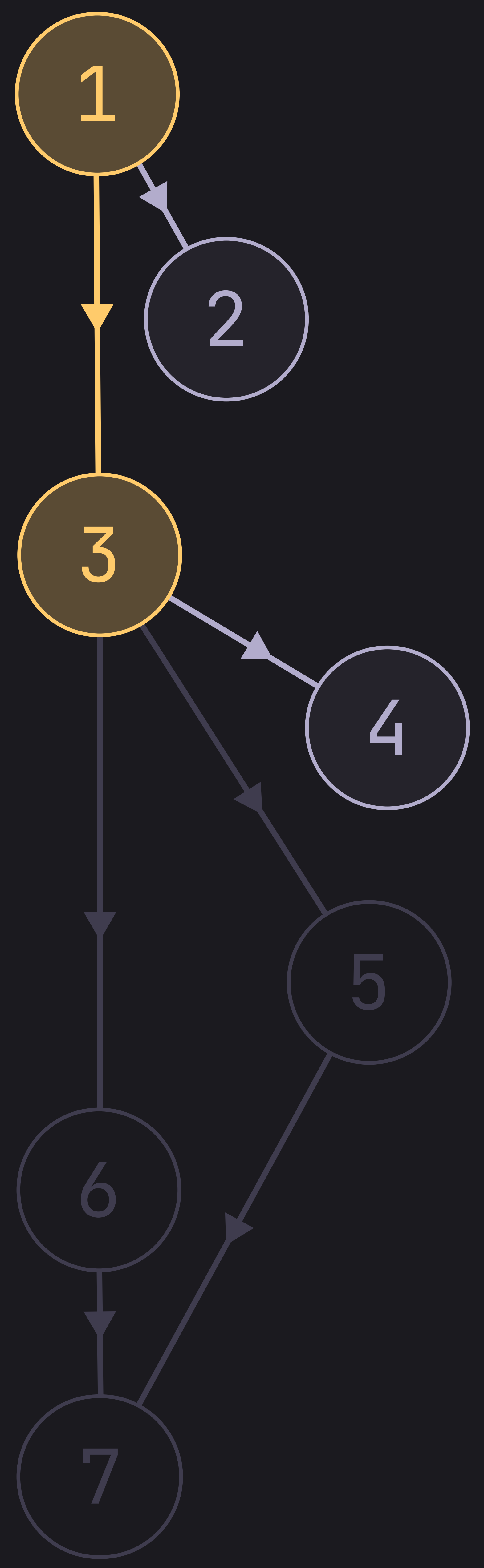
So let's try apply a postorder traversal to our graph of subjects

[1, 3]

Stack

7, 6, 5

Post order



Topological Sort

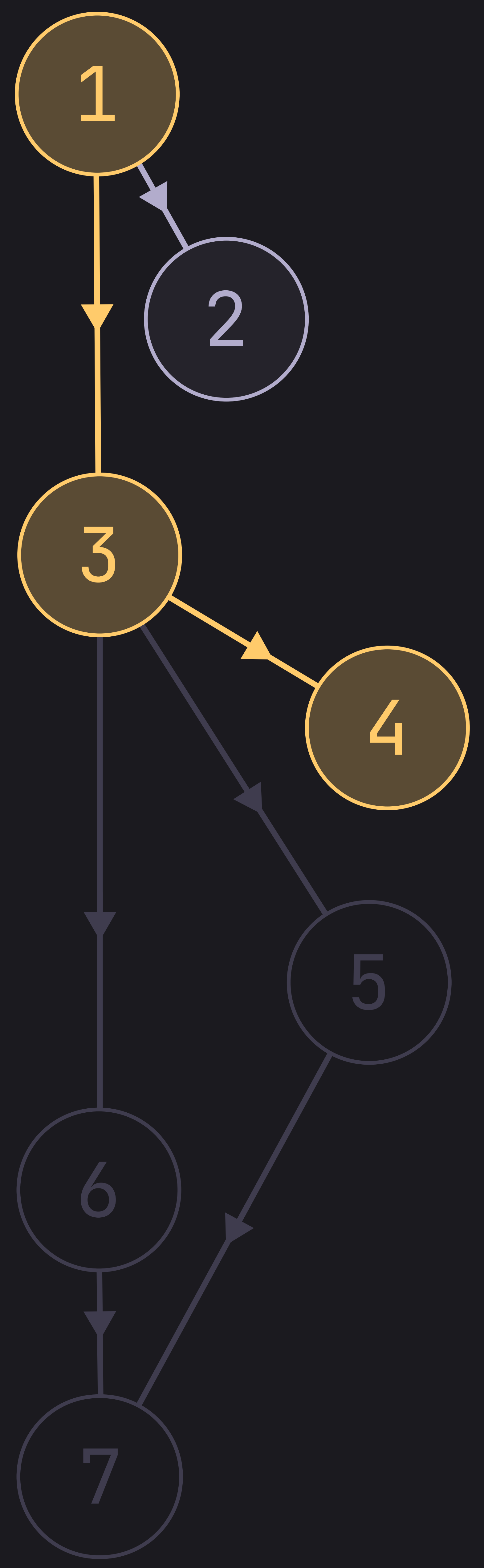
So let's try apply a postorder traversal to our graph of subjects

[1, 3, 4]

Stack

7, 6, 5

Post order



Topological Sort

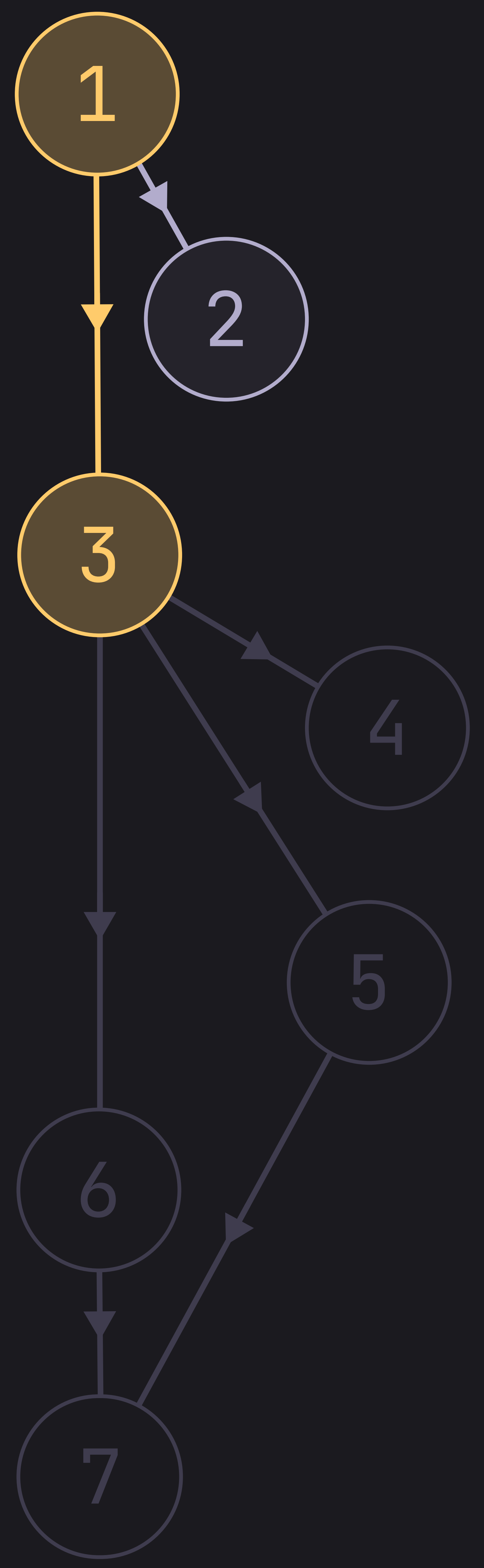
So let's try apply a
postorder traversal to our
graph of subjects

[1, 3]

Stack

7, 6, 5, 4

Post order



Topological Sort

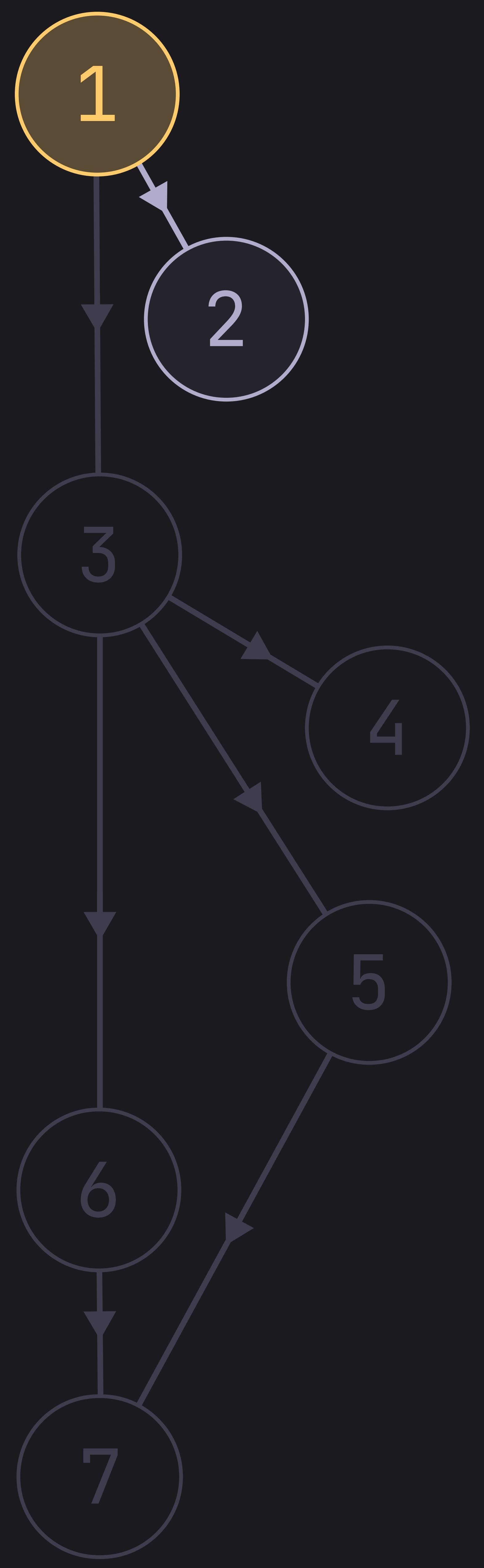
So let's try apply a postorder traversal to our graph of subjects

[1]

Stack

7, 6, 5, 4, 3

Post order



Topological Sort

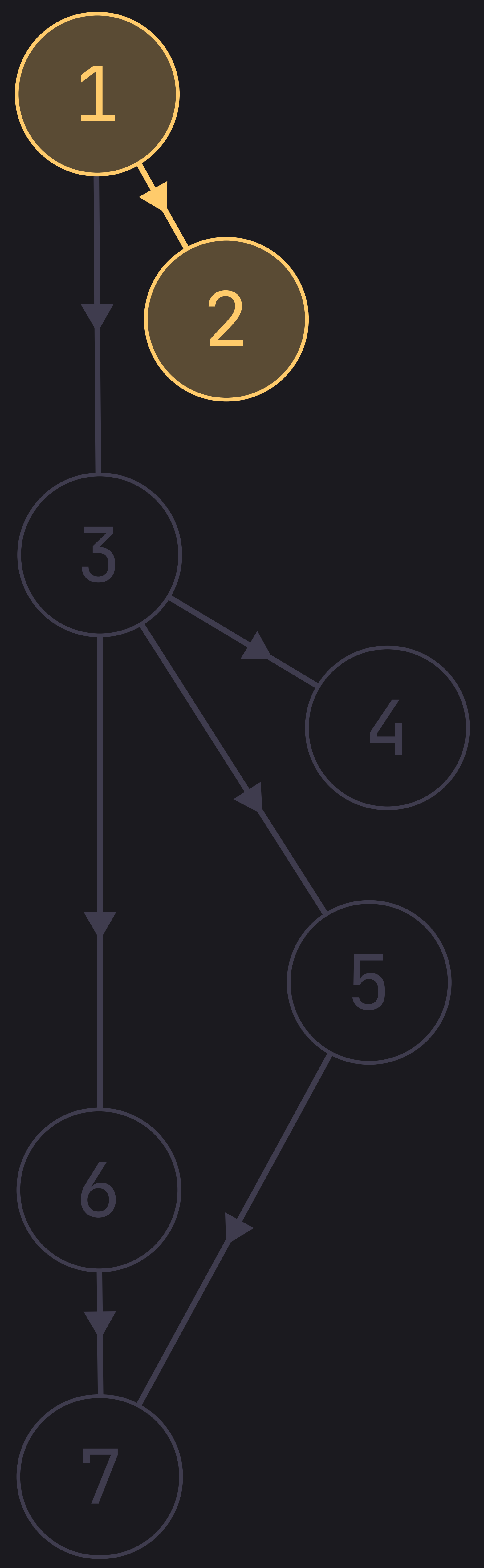
So let's try apply a postorder traversal to our graph of subjects

[1, 2]

Stack

7, 6, 5, 4, 3

Post order



Topological Sort

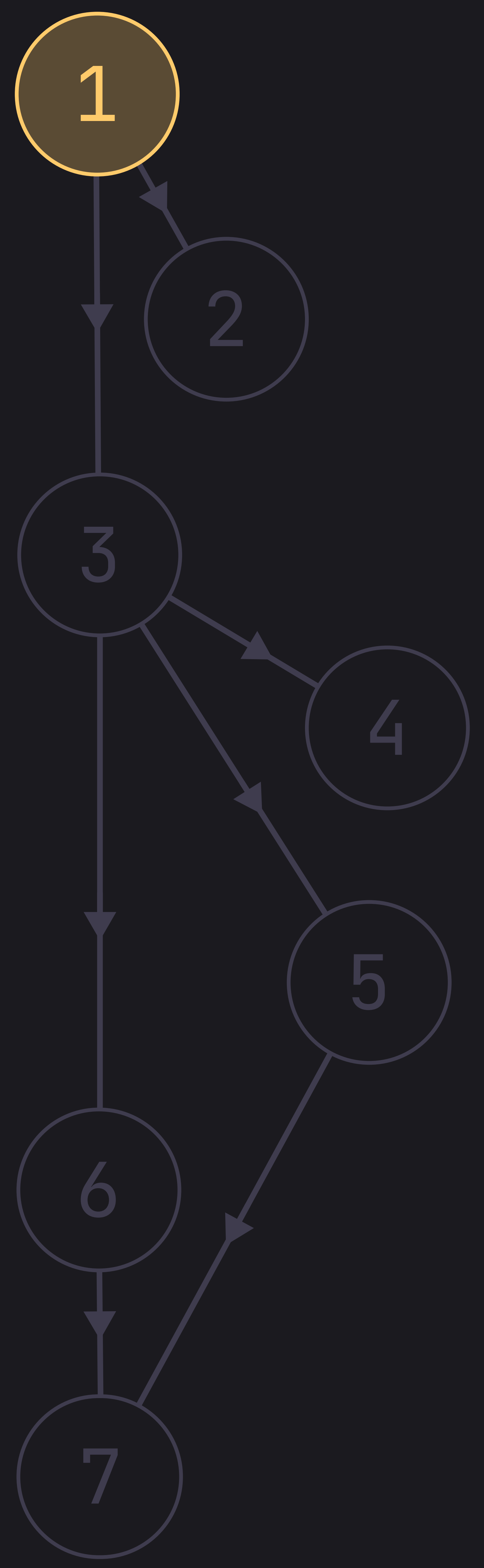
So let's try apply a postorder traversal to our graph of subjects

[1]

Stack

7, 6, 5, 4, 3, 2

Post order



Topological Sort

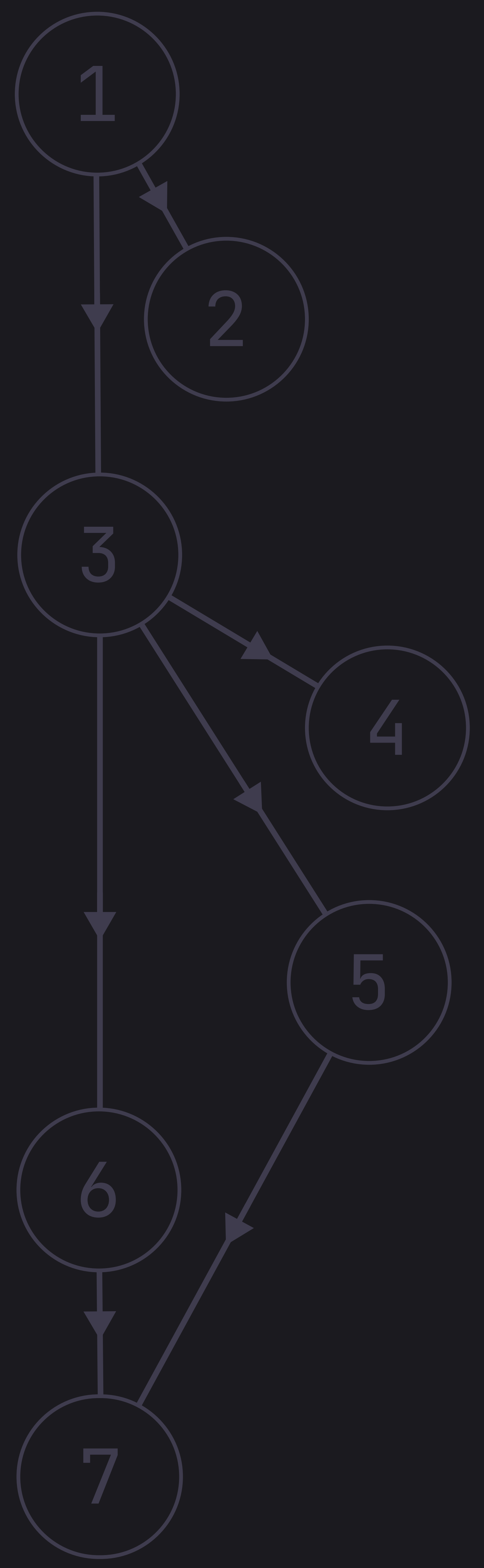
So let's try apply a postorder traversal to our graph of subjects

[ ]

Stack

7, 6, 5, 4, 3, 2, 1

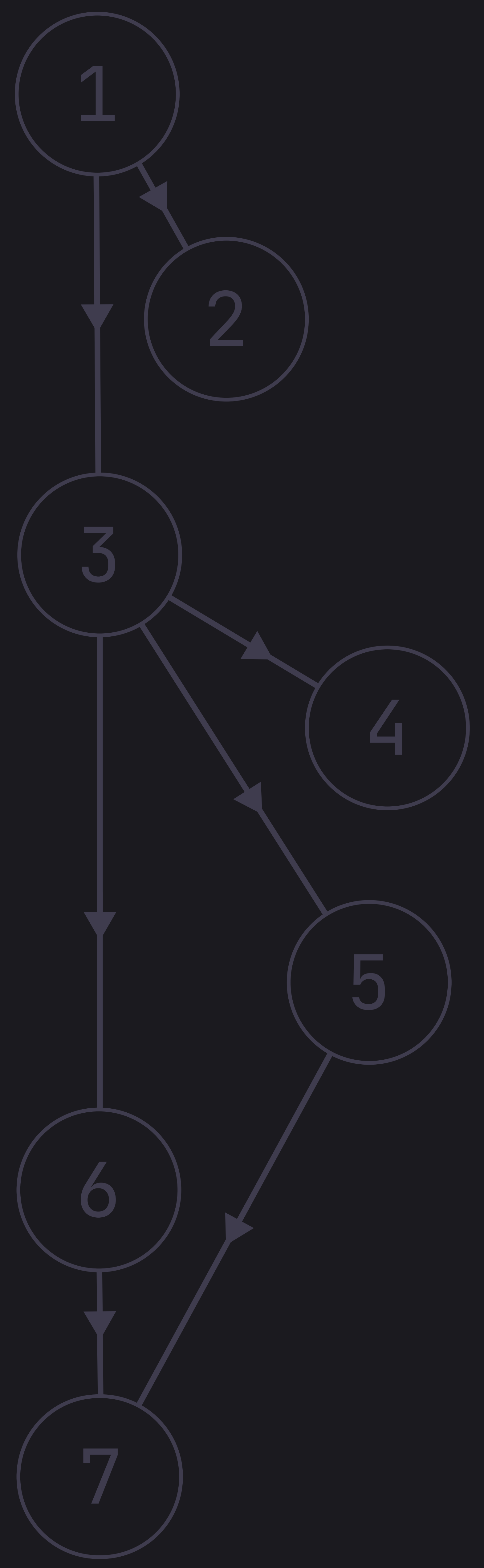
Post order



Topological Sort

So a postorder traversal gives us a reverse topological sort.

So all we need to do is **reverse** the result

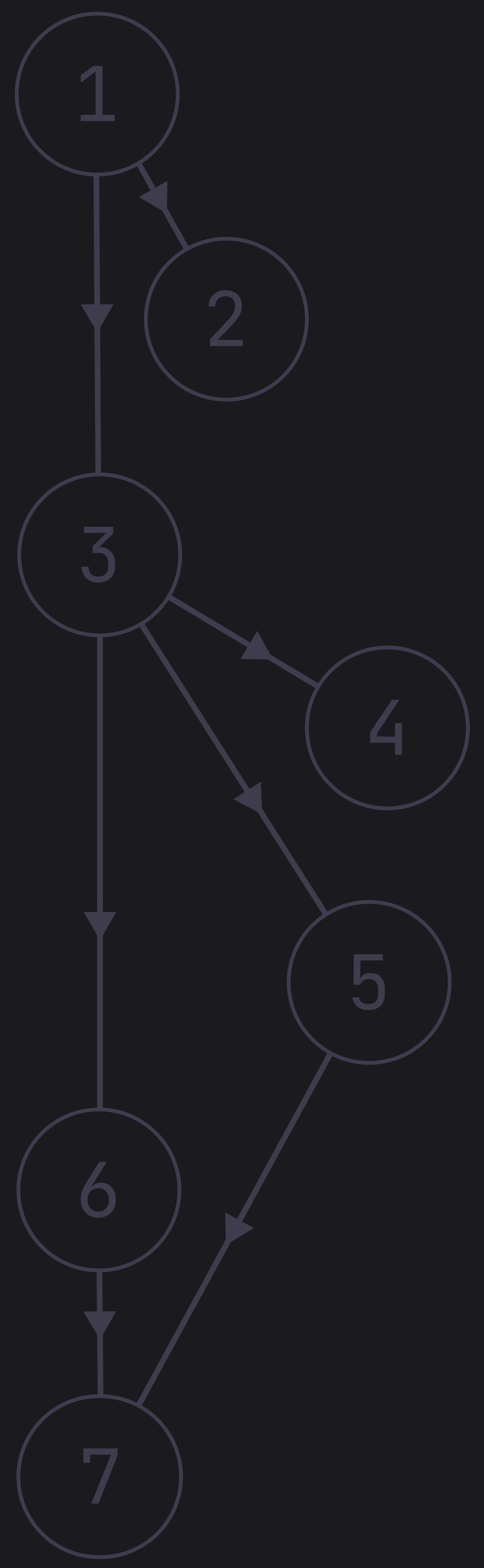


1, 2, 3, 4, 5, 6, 7

Reverse post order

Topological Sort

The only problem we have is we don’t know what the starting vertex is.

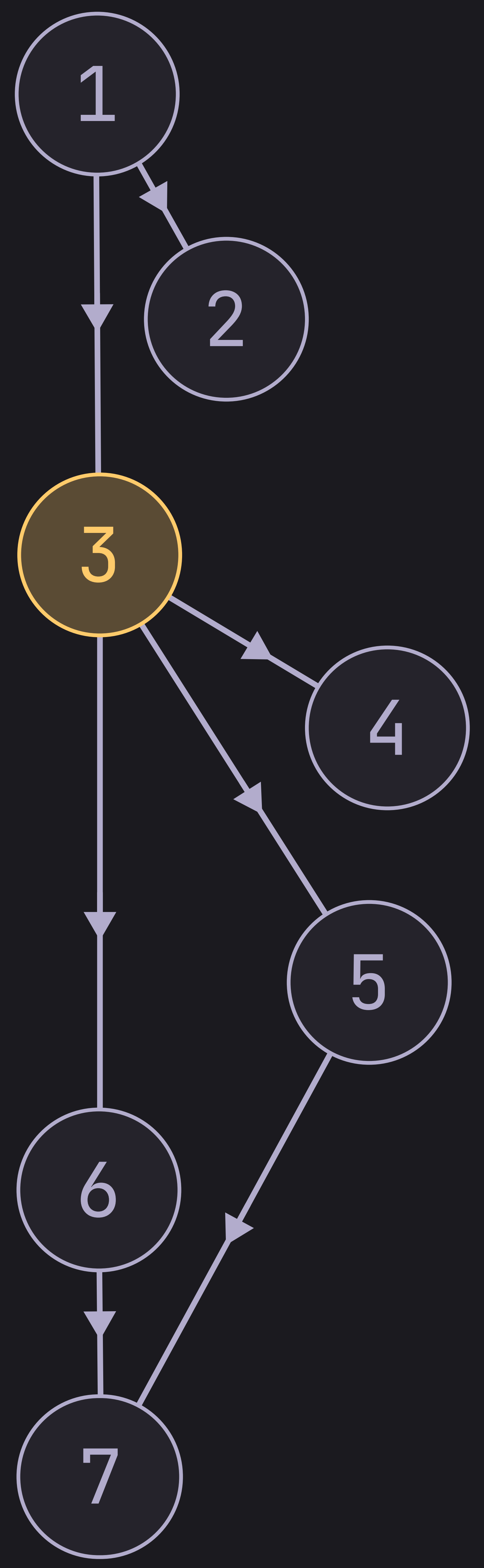


Topological Sort

The only problem we have is we don't know what the starting vertex is.

Thankfully this is really easy to deal with

Suppose we start at **vertex 3**

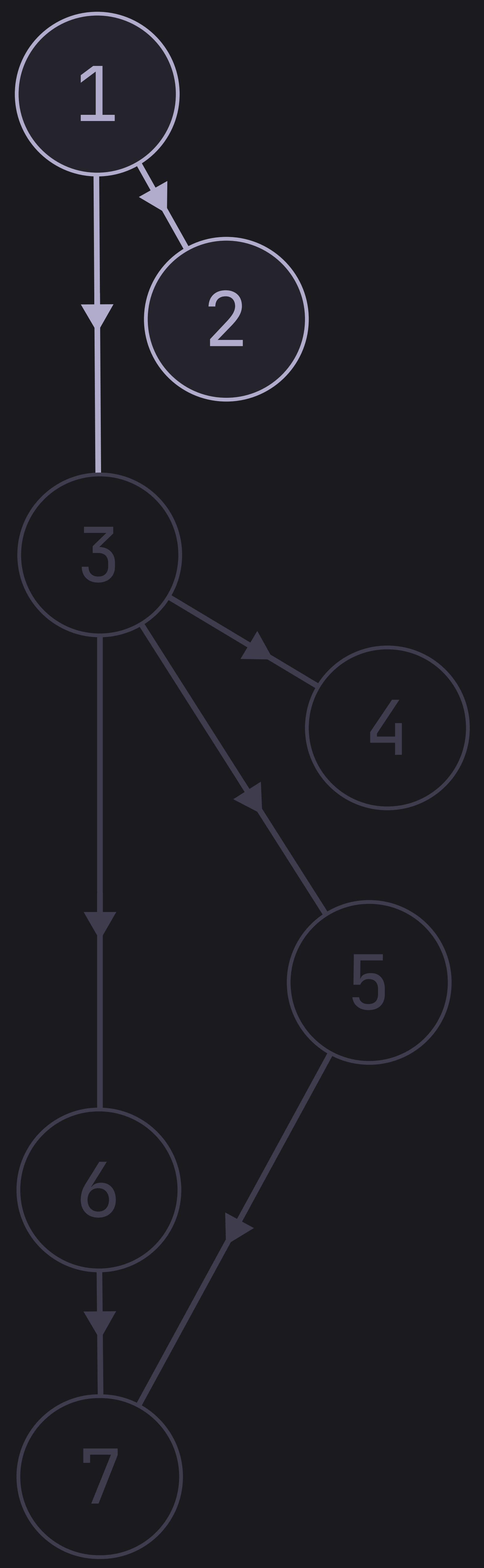


Topological Sort

After performing a post order traversal starting from **vertex 3** we should have the following

7, 6, 5, 4, 3

Post order

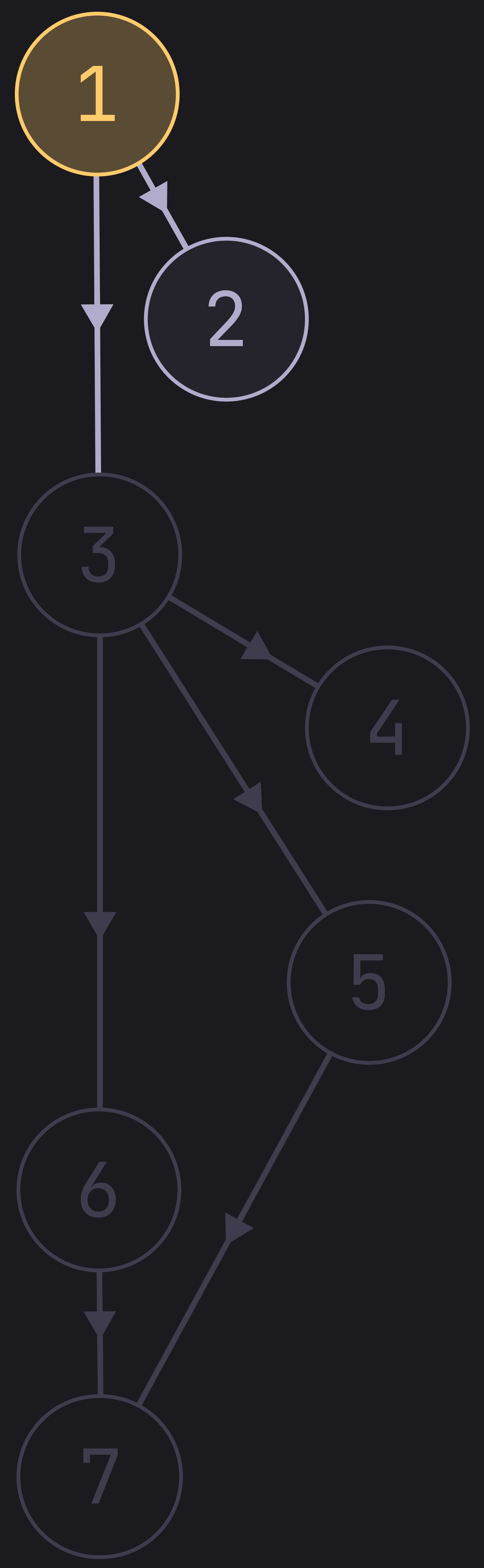


Topological Sort

Now we just pick another unvisited vertex. Then we perform another post order traversal and we just **append** to the current result

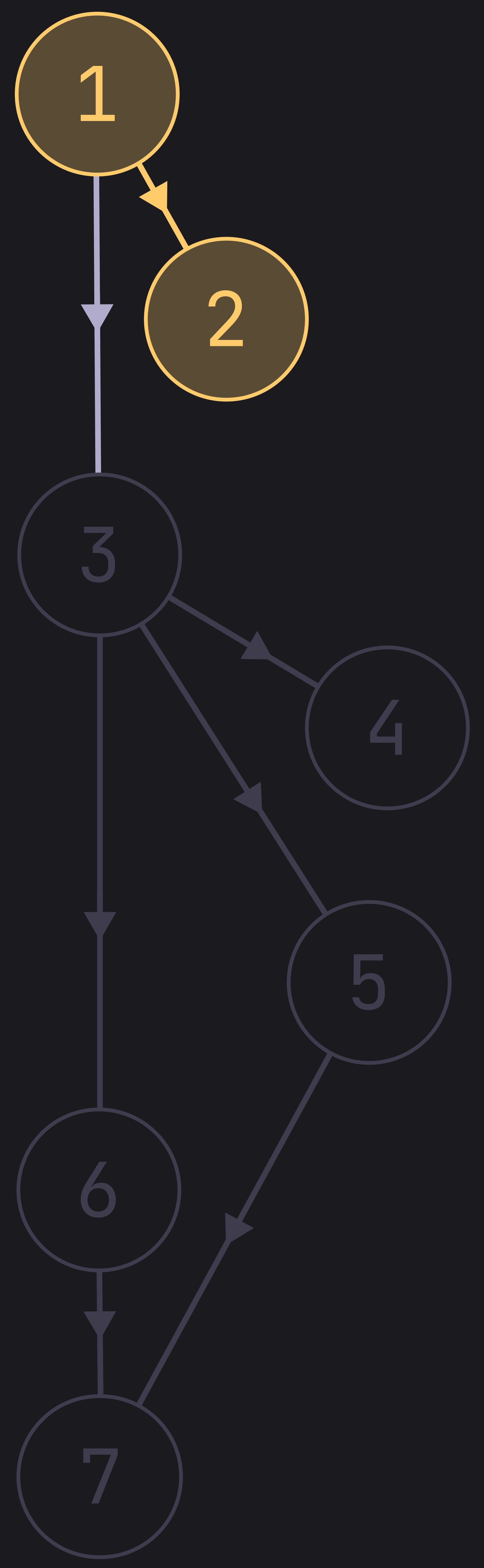
7, 6, 5, 4, 3

Post order



Topological Sort

Now we just pick another unvisited vertex. Then we perform another post order traversal and we just **append** to the current result



7, 6, 5, 4, 3

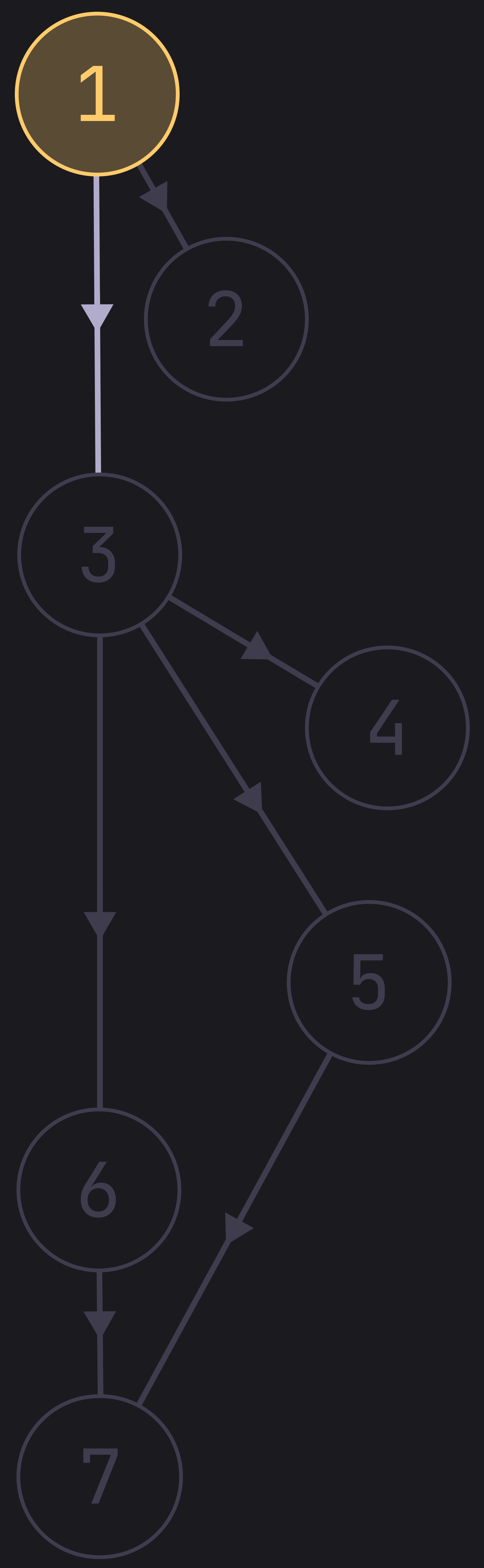
Post order

Topological Sort

Now we just pick another unvisited vertex. Then we perform another post order traversal and we just **append** to the current result

7, 6, 5, 4, 3, 2

Post order

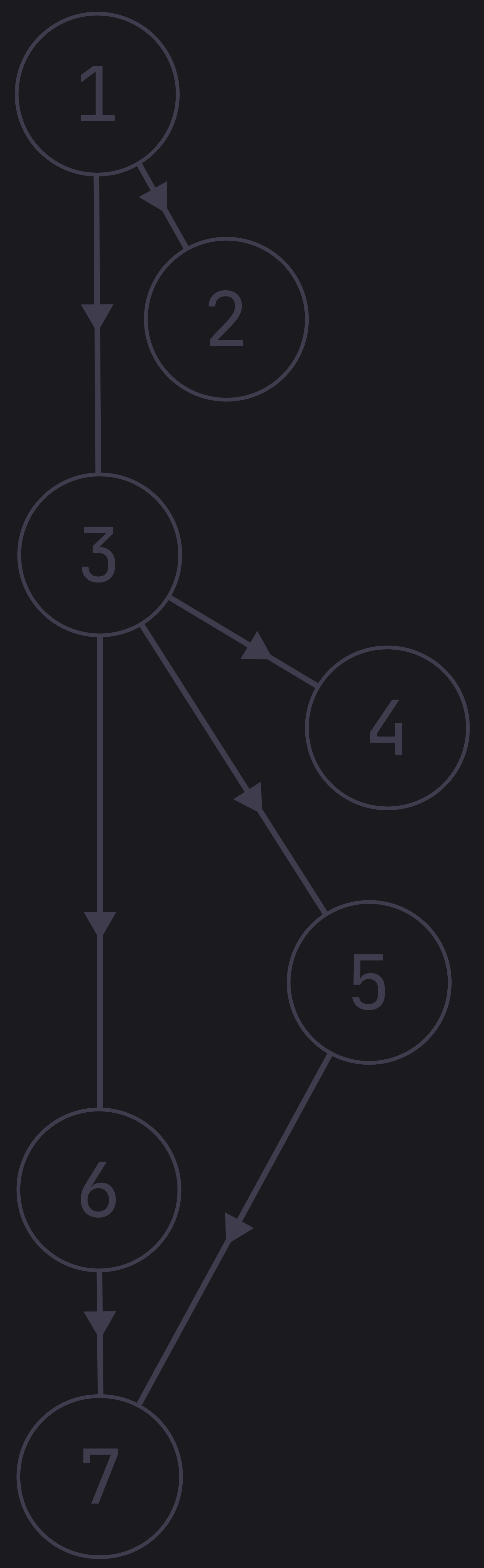


Topological Sort

Now we just pick another unvisited vertex. Then we perform another post order traversal and we just **append** to the current result

7, 6, 5, 4, 3, 2, 1

Post order



Topological Sort

```
bool dfs(int v, Graph graph):
    onStack.insert(v)
    visited.insert(v)
    for neighbour in graph.getNeighbours(start):
        if onStack.contains(neighbour):
            # we have detected a cycle
            return true
        if not visited.contains(neighbour):
            existsCycle = dfs(neighbour, graph)
            if existsCycle:
                return true
    onStack.erase(v)
    order.push_back(v)
    return false
```

```
bool computeTopoOrder(Graph graph):
    visited.clear()
    for vertex in graph:
        if not visited.contains(v):
            # clear onStack for each running of dfsVisit
            onStack.clear()
            onStack.add(v)
            existsCycle = dfs(v, graph)
            if existsCycle:
                return True
    order.reverse()
    return False
```

Student Feedback Survey

The SFS is open as of now! You can now write tutor specific feedback. Please give me some feedback. It would really mean a lot 🥰

https://www.sfs.uts.edu.au/