DSC 40B

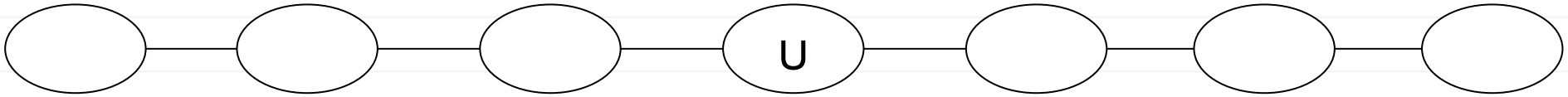
Lecture 22 : Depth First Search

Depth First Search

Visiting the Next Node

- Which node do we process next in a search?
- **BFS**: the **oldest** pending node.
- **DFS** (today): the **newest** pending node.
  - Naturally recursive.
  - Useful for solving different problems.

Example (BFS)



Example (BFS)

Example (BFS)

Example (BFS)

Example (BFS)

Example (DFS)

Example (DFS)

Example (DFS)

Example (DFS)

Example (DFS)

Example (DFS)

Example (DFS)

Example (DFS)

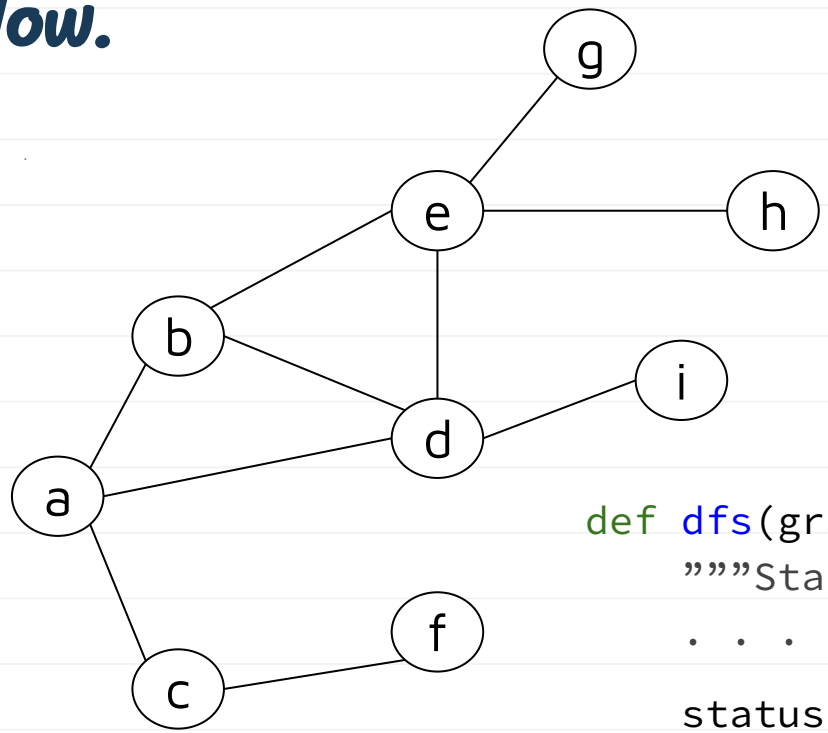
Example (DFS)

Example (DFS)

Example (DFS)

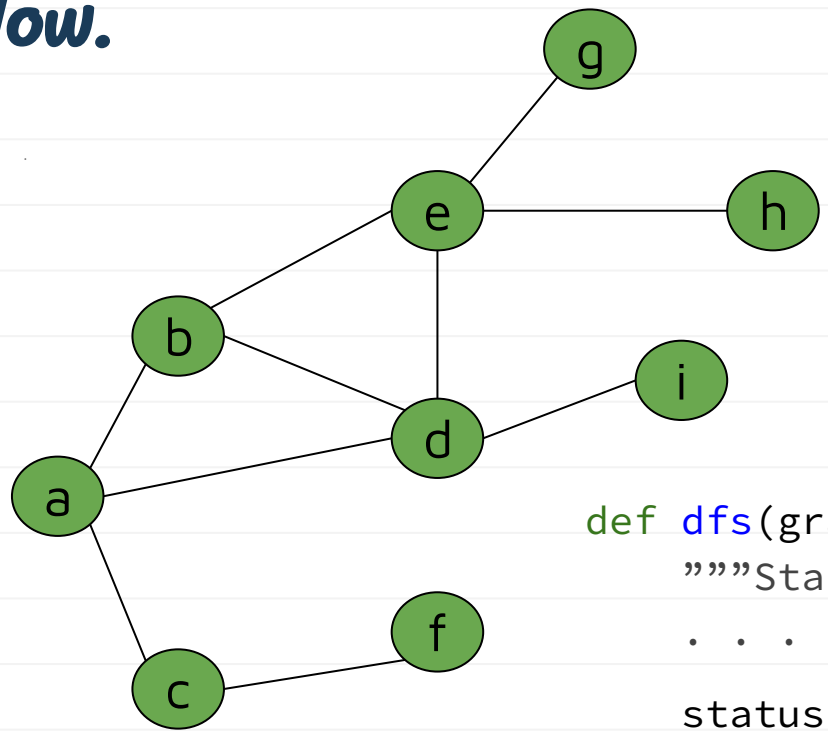
```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Write the nested function calls for a DFS on the graph below.



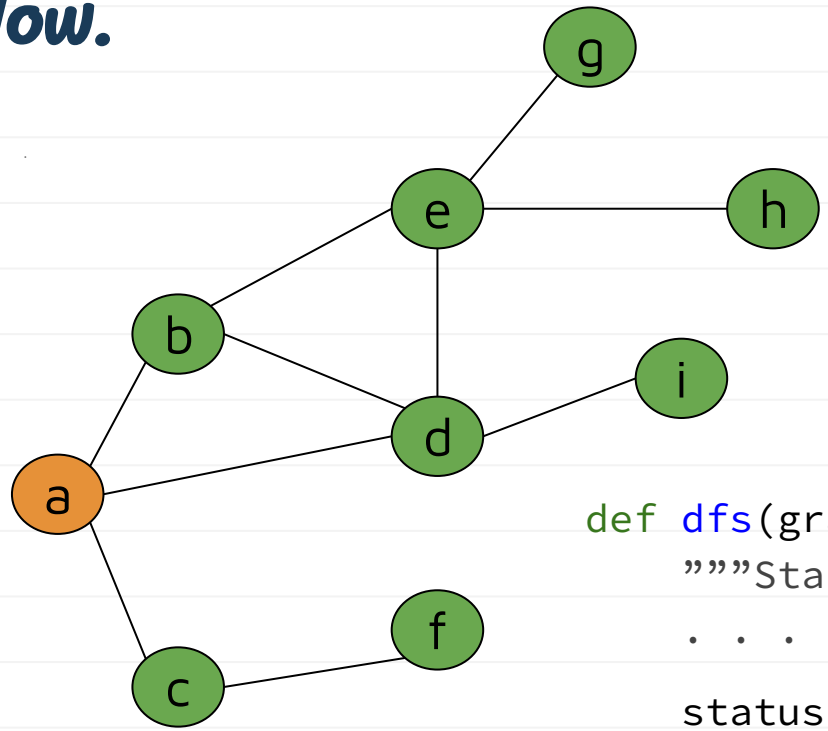

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Write the nested function calls for a DFS on the graph below.




```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

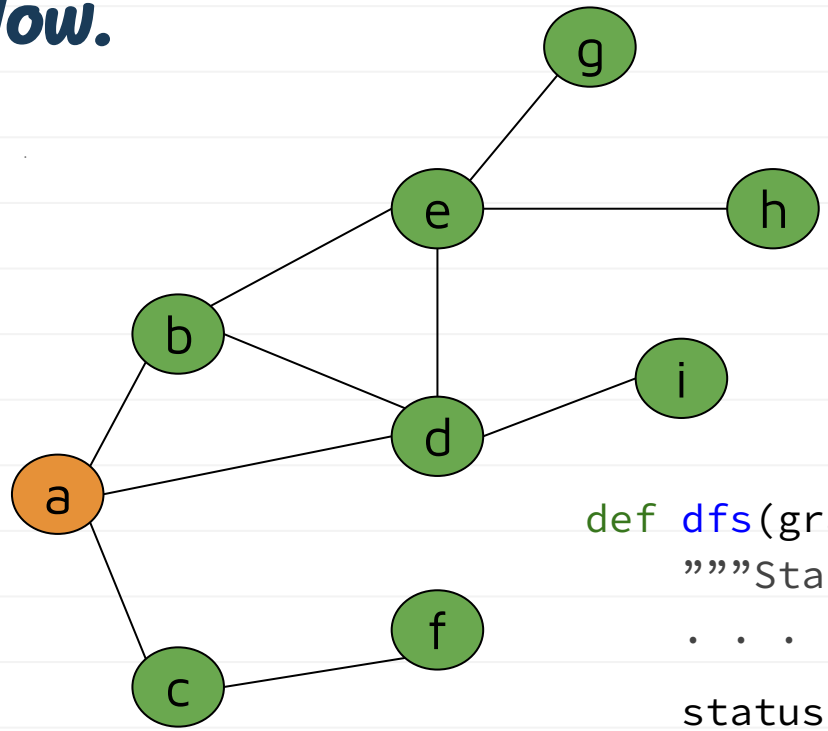
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

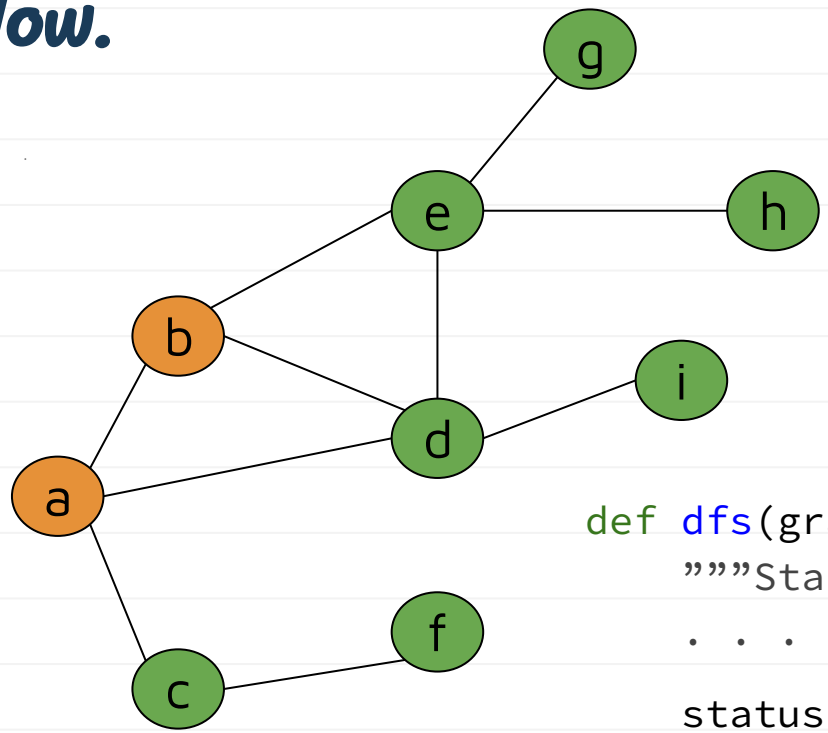
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

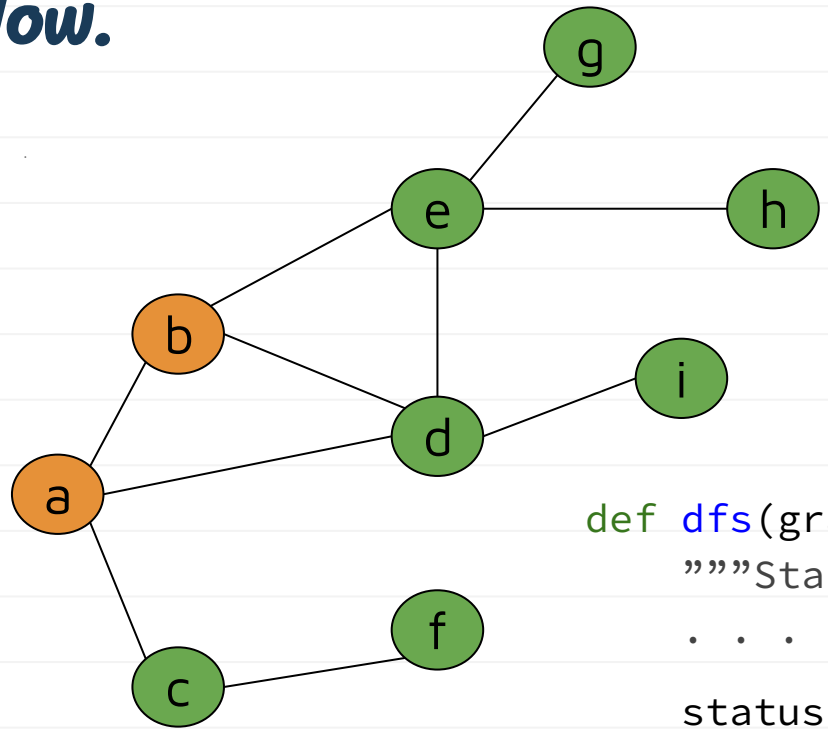
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

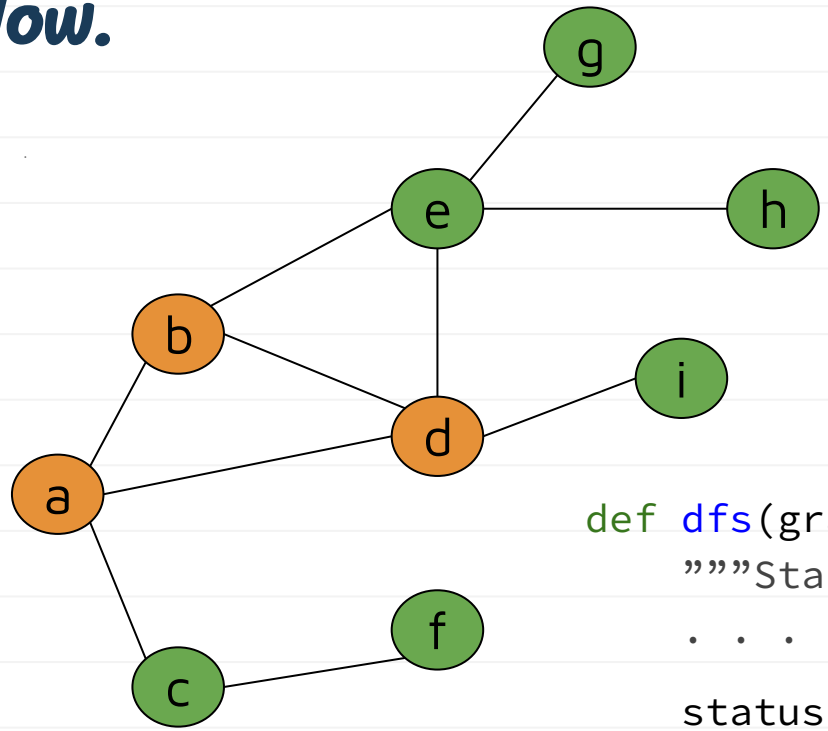
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

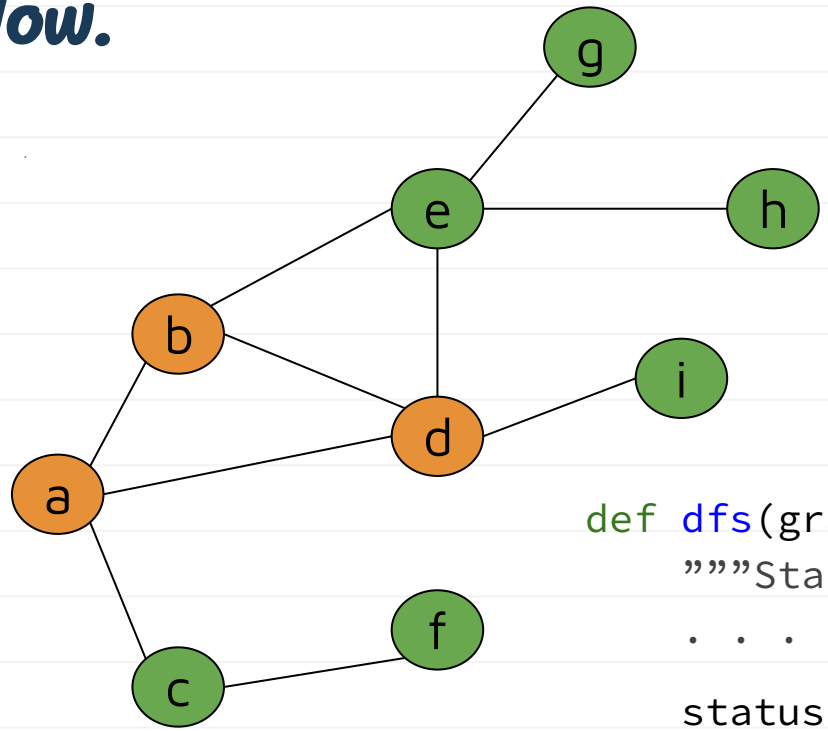
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

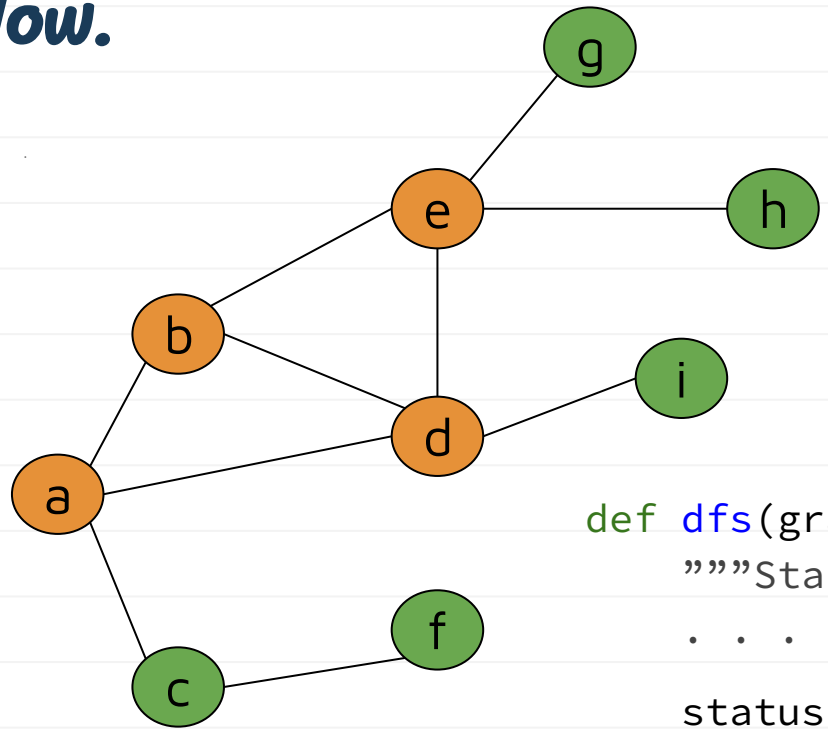
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

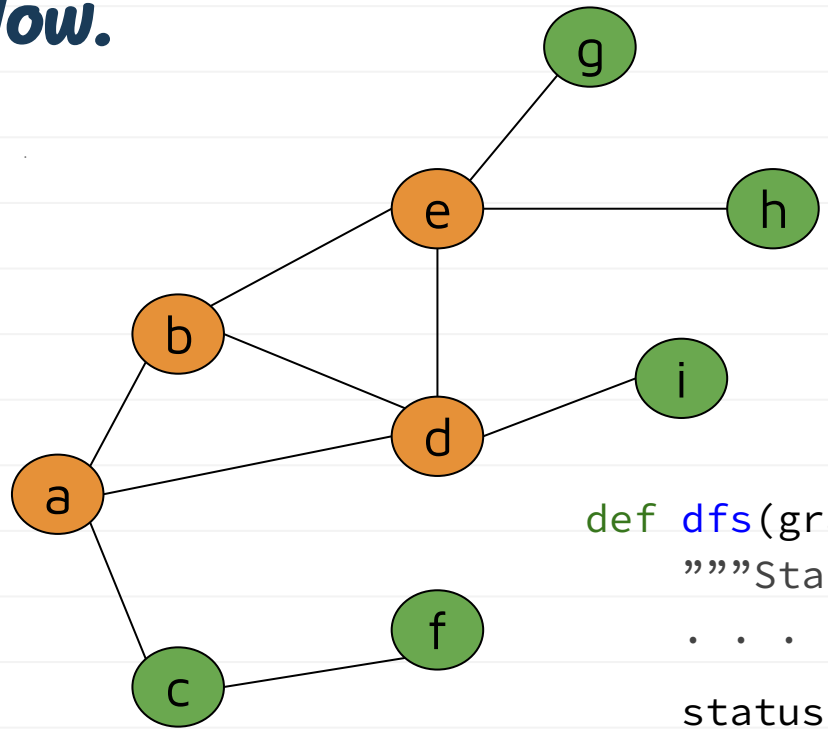
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

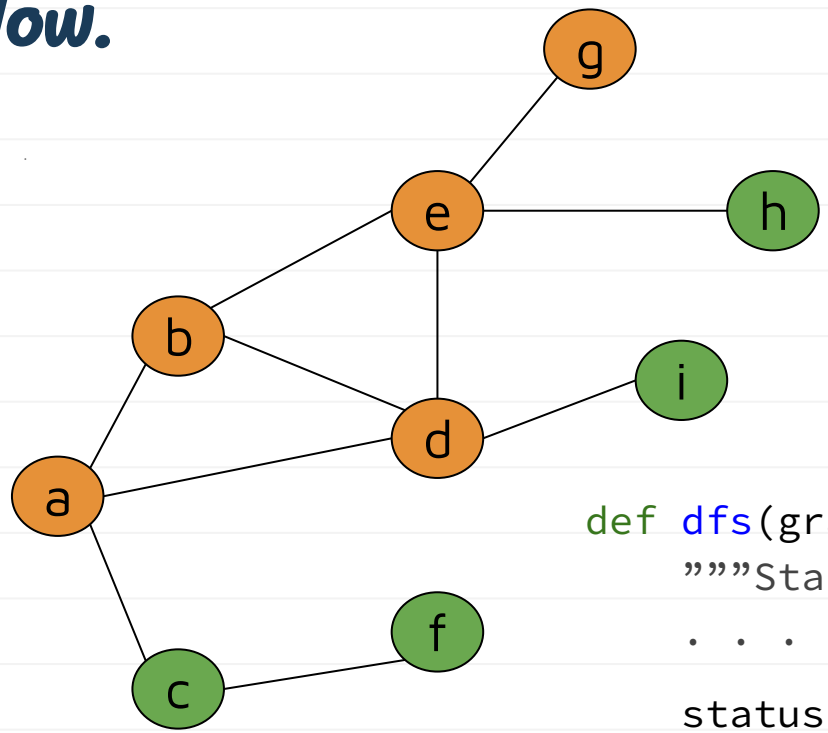
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

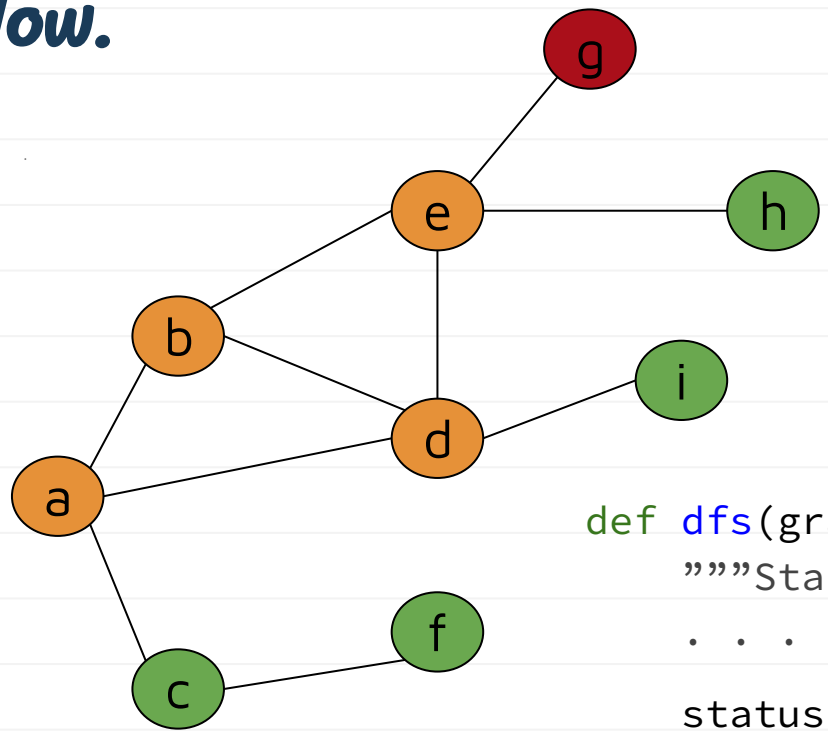
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

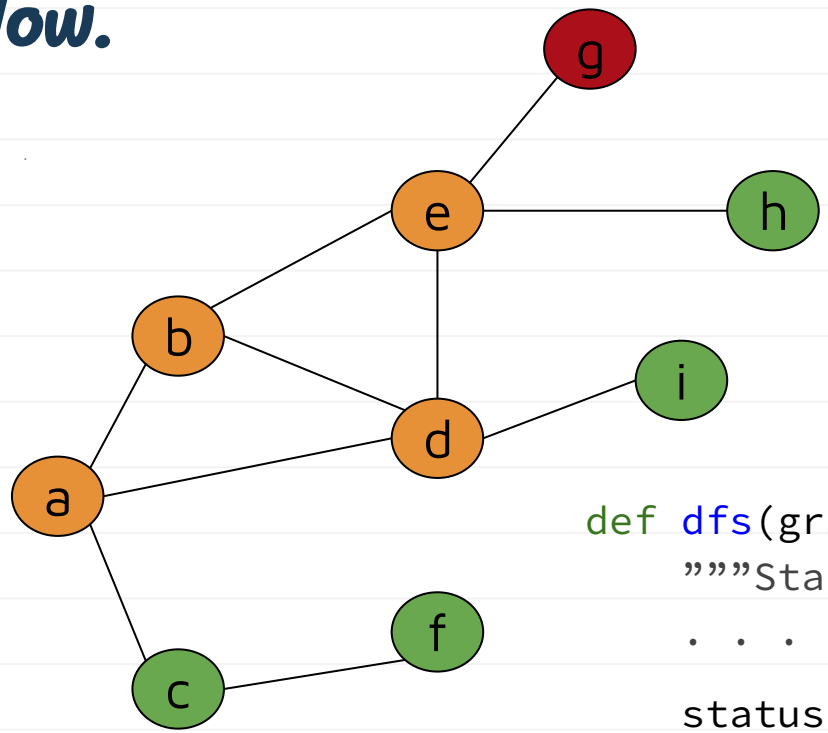
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Write the nested function calls for a DFS on the graph below.



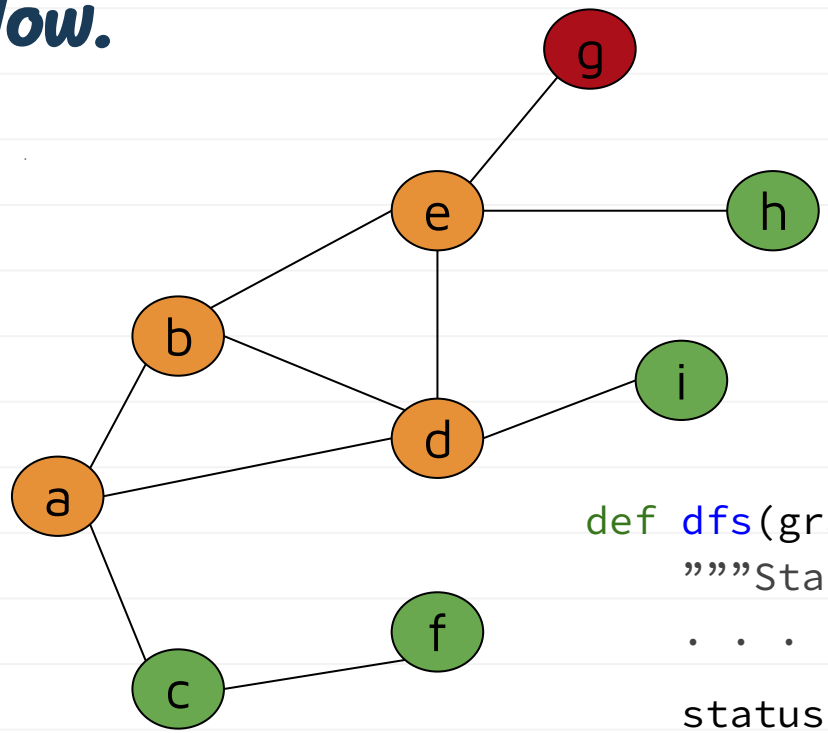

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        ~~dfs(g)~~
```

What node becomes v?

A: e
B: h
C: i

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

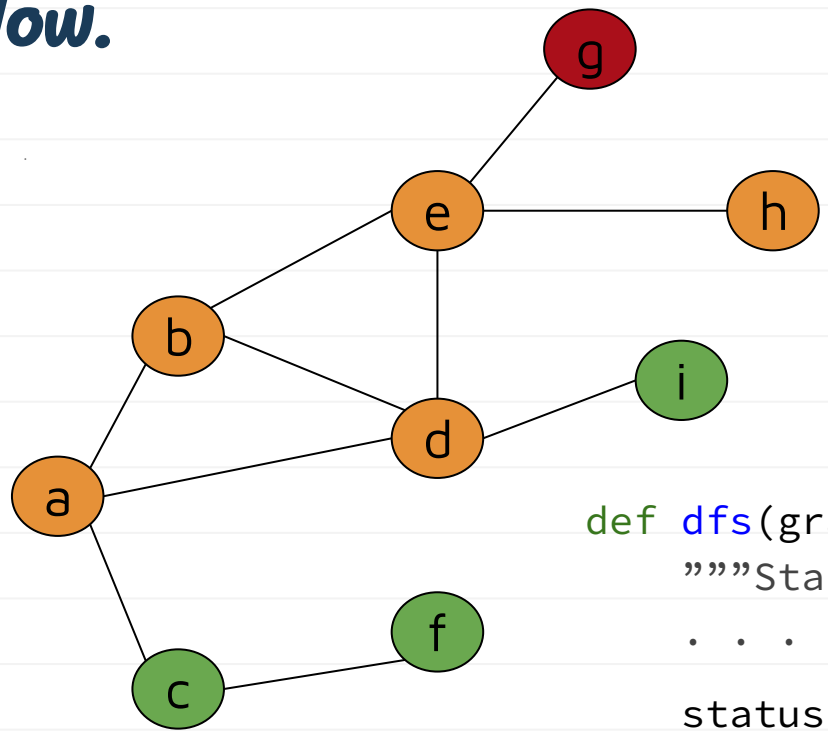
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        ~~dfs(g)~~
        dfs(h)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

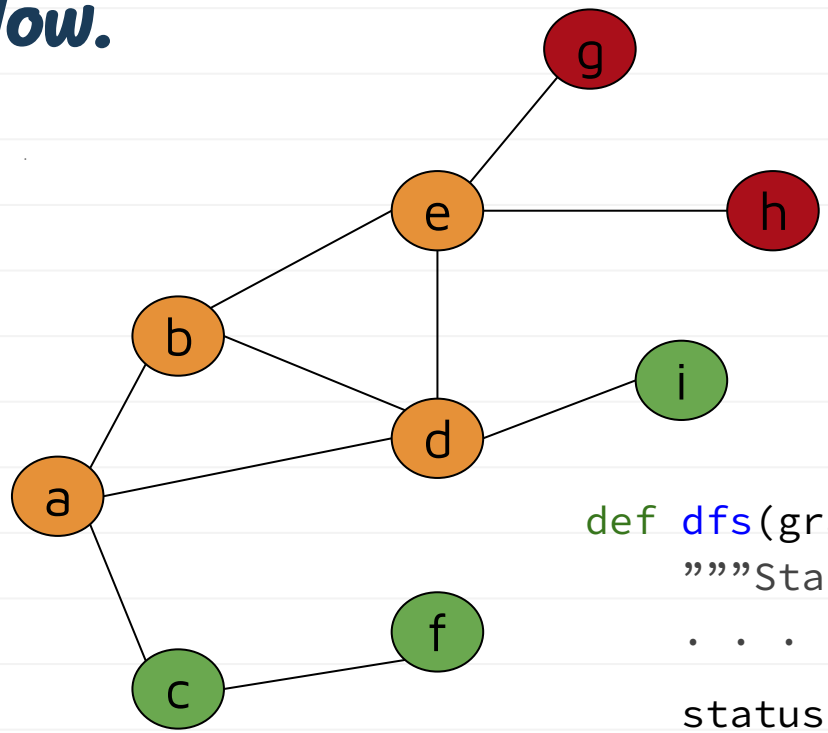
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        ~~dfs(g)~~
        dfs(h)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

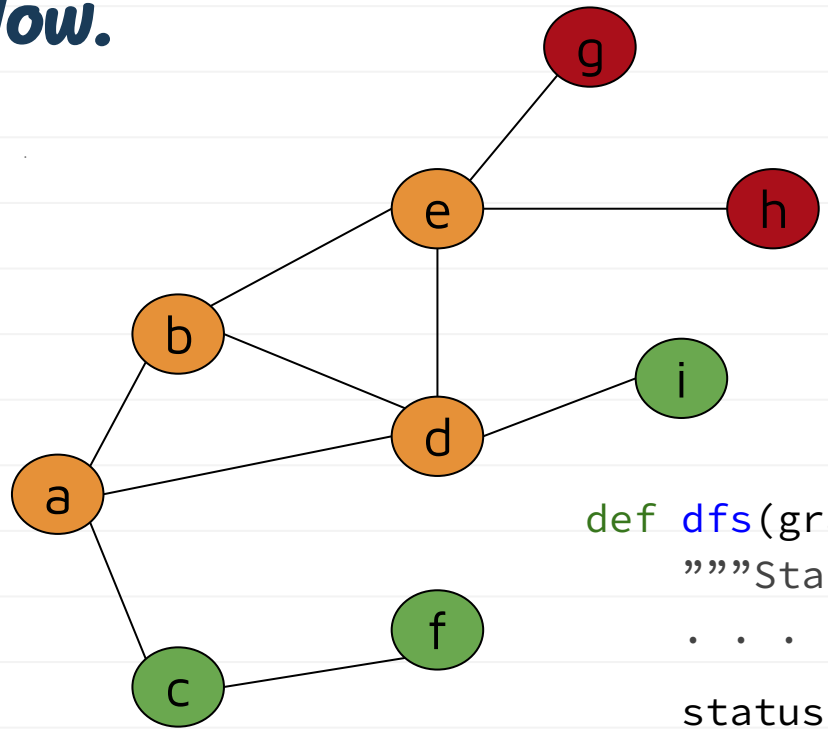
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        ~~dfs(g)~~
        dfs(h)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

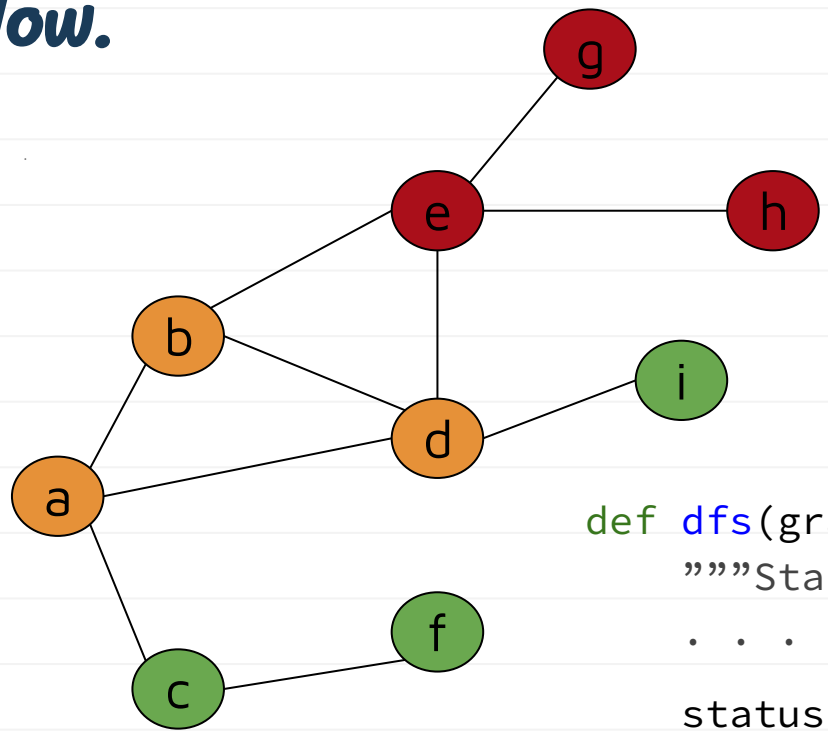
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        ~~dfs(g)~~
        ~~dfs(h)~~
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

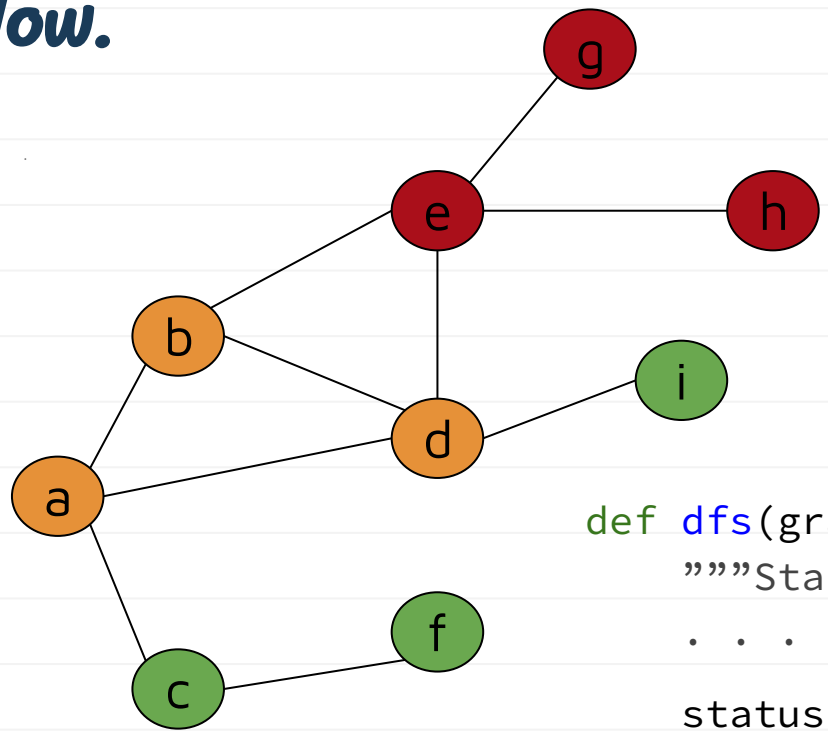
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        ~~dfs(g)~~
        ~~dfs(h)~~
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

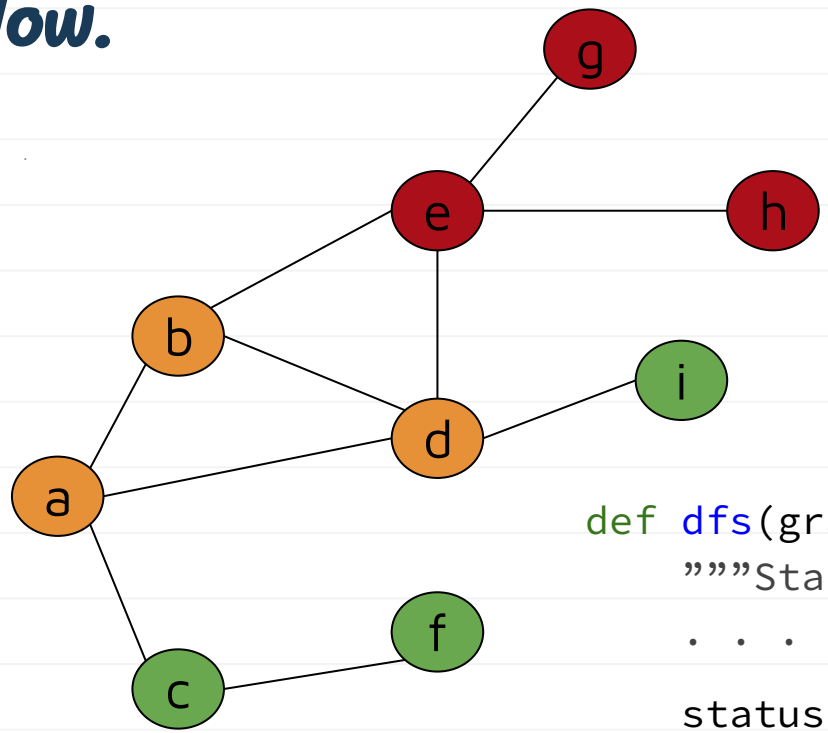
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      ~~dfs(e)~~
        ~~dfs(g)~~
        ~~dfs(h)~~
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

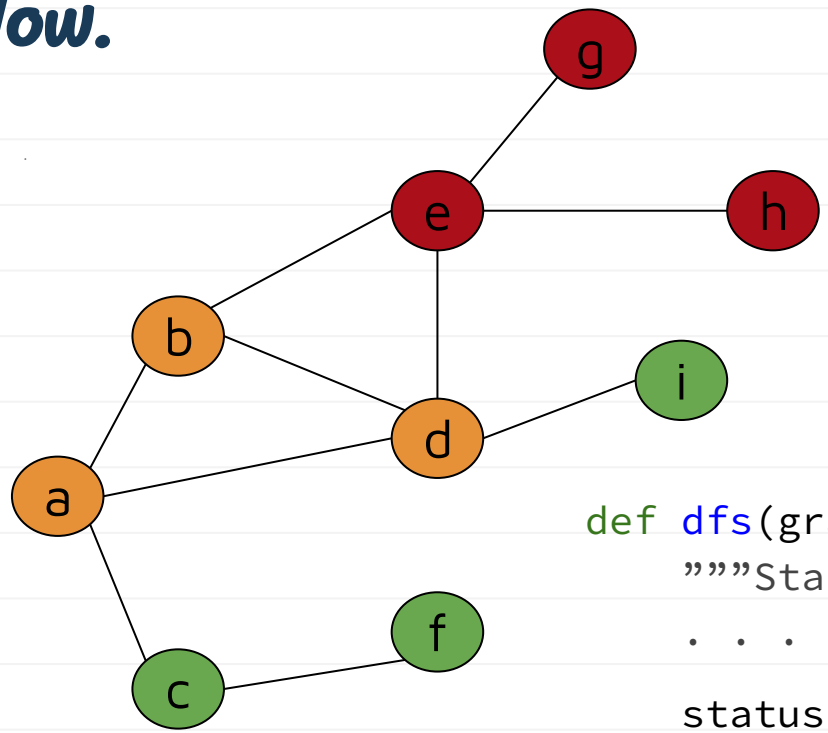
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      ~~dfs(e)~~
        ~~dfs(g)~~
        ~~dfs(h)~~
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

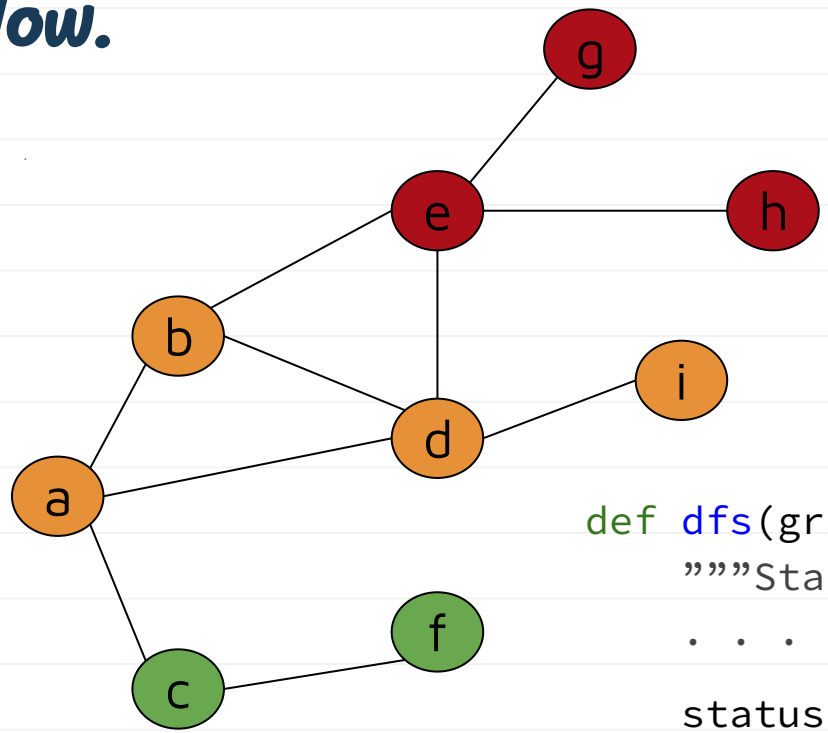
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      ~~dfs(e)~~
        ~~dfs(g)~~
        ~~dfs(h)~~
      dfs(i)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

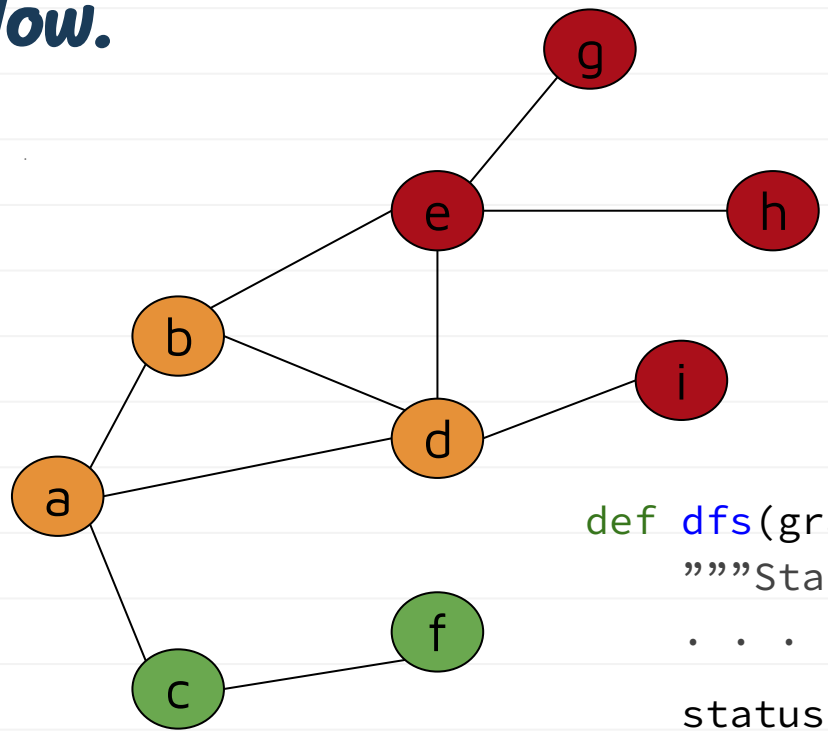
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      ~~dfs(e)~~
        ~~dfs(g)~~
        ~~dfs(h)~~
      dfs(i)
```

```python
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

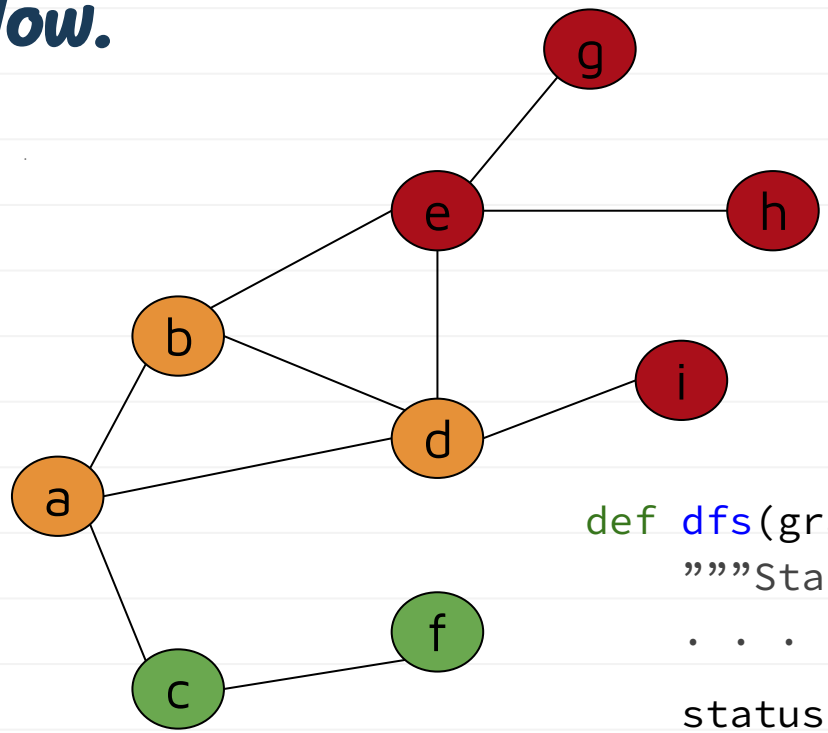
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      ~~dfs(e)~~
        ~~dfs(g)~~
        ~~dfs(h)~~
      dfs(i)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

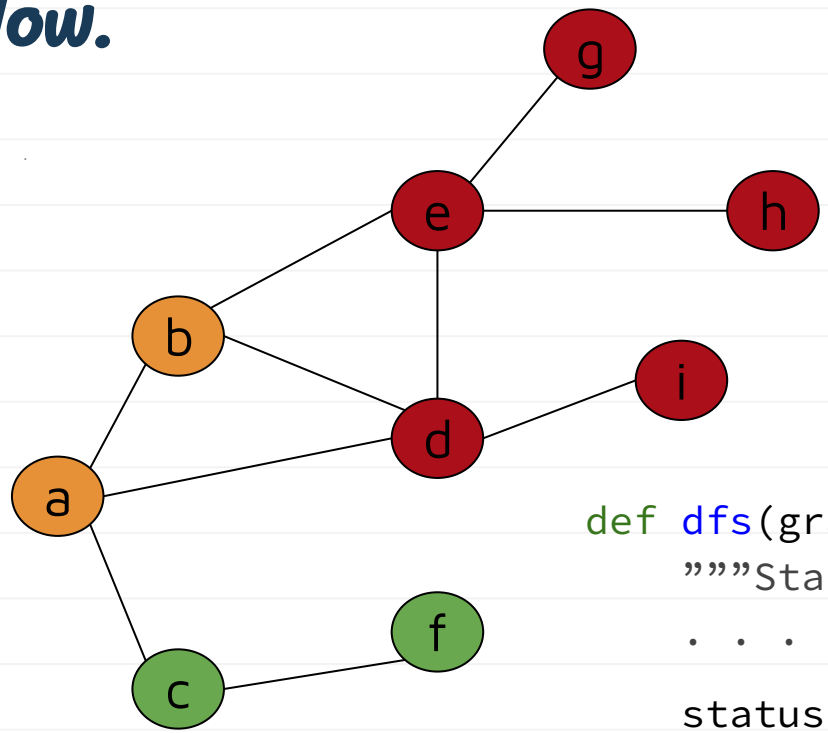
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      ~~dfs(e)~~
        ~~dfs(g)~~
        ~~dfs(h)~~
      ~~dfs(i)~~
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

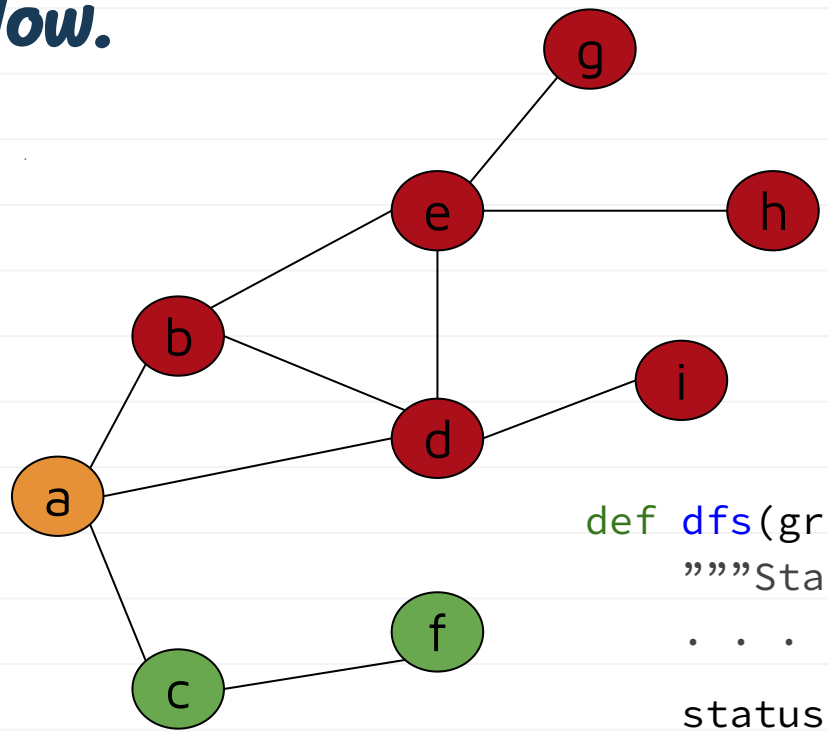
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    ~~dfs(d)~~
      ~~dfs(e)~~
        ~~dfs(g)~~
        ~~dfs(h)~~
      ~~dfs(i)~~
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

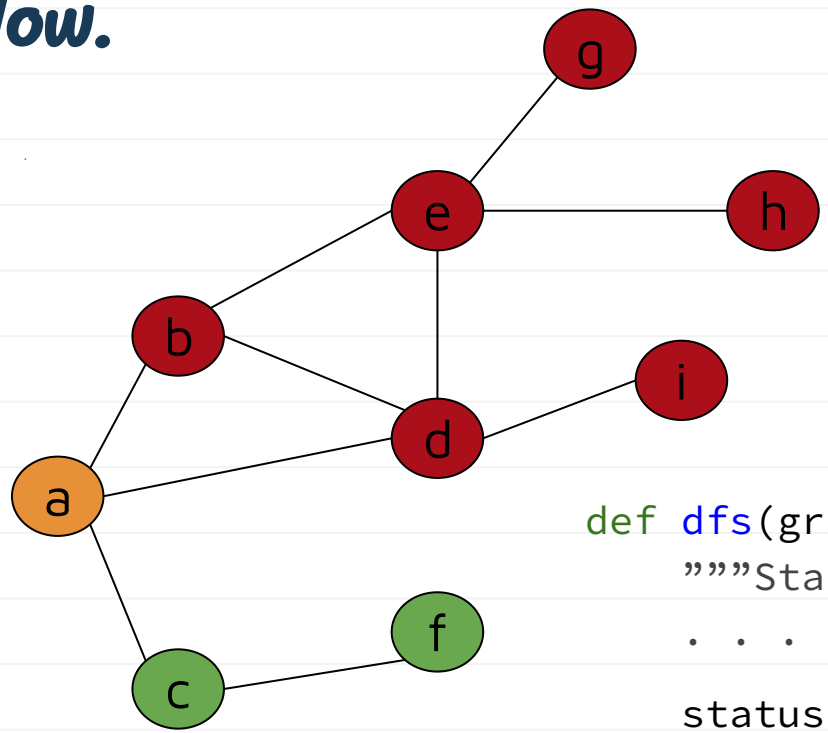
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    ~~dfs(d)~~
      ~~dfs(e)~~
        ~~dfs(g)~~
        ~~dfs(h)~~
      ~~dfs(i)~~
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

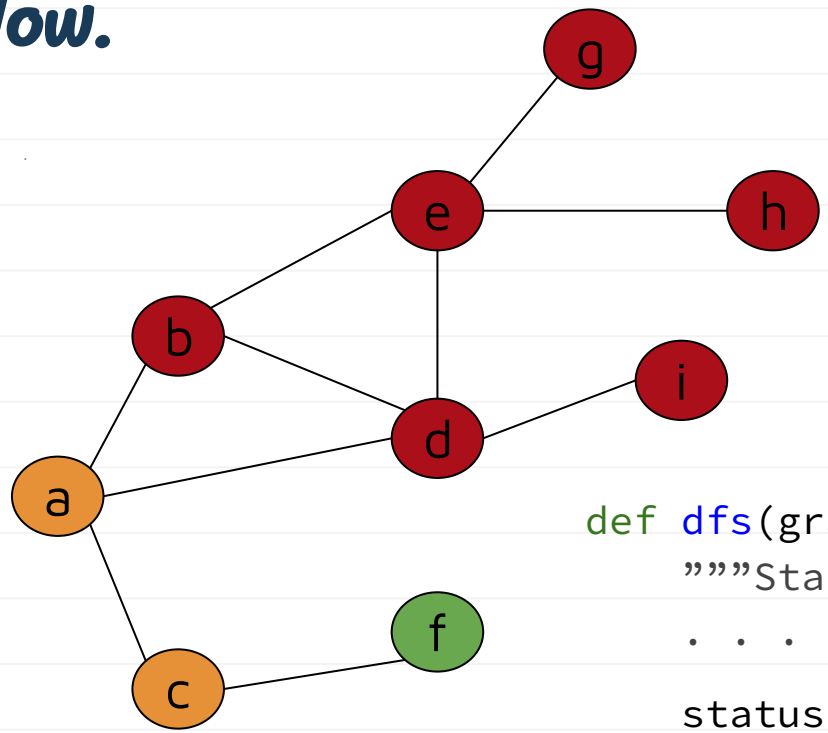
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

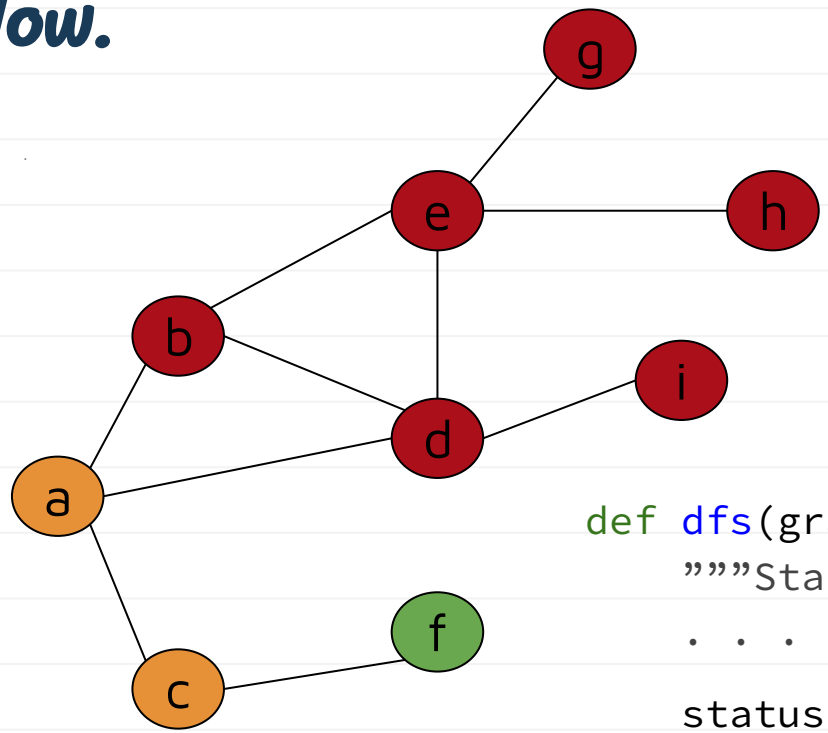
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

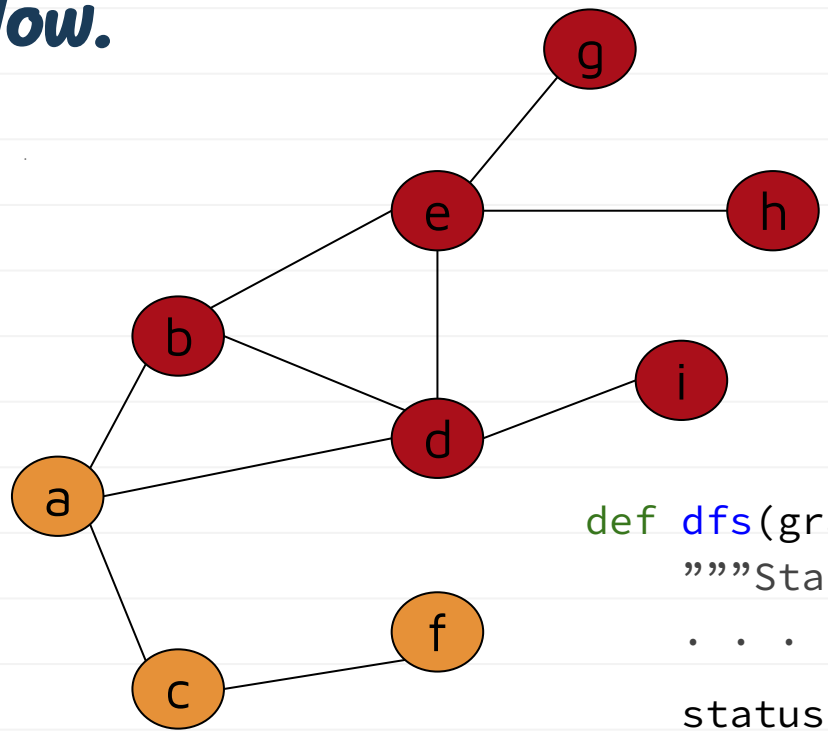
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
  dfs(f)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

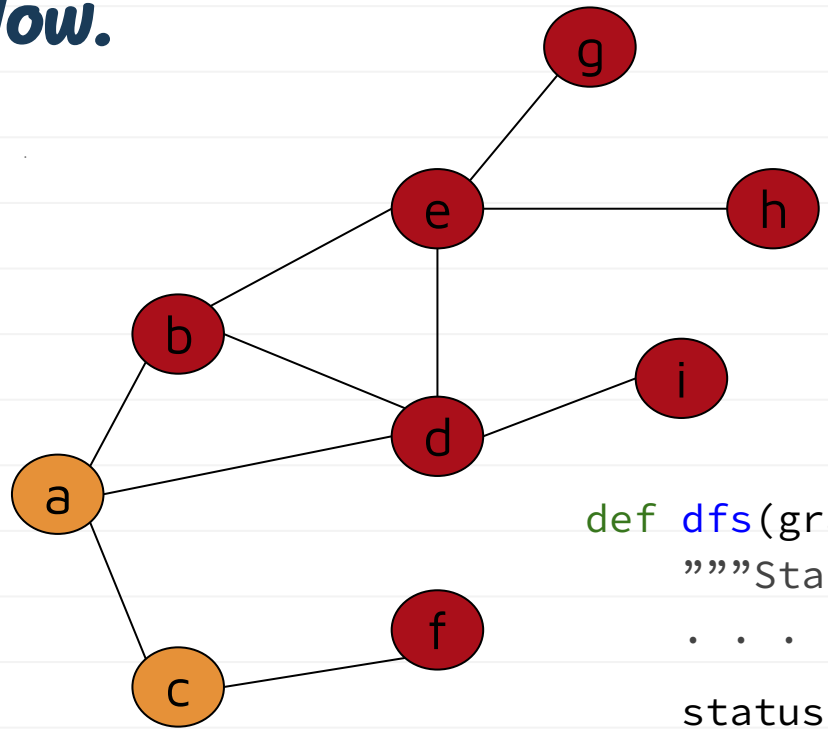
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
  dfs(f)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

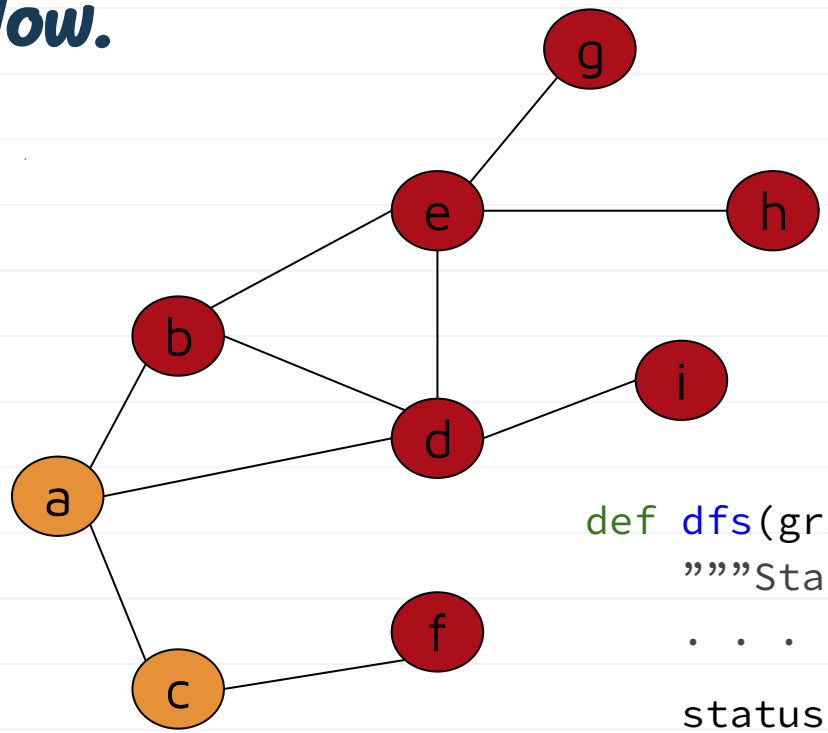
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
  dfs(f)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

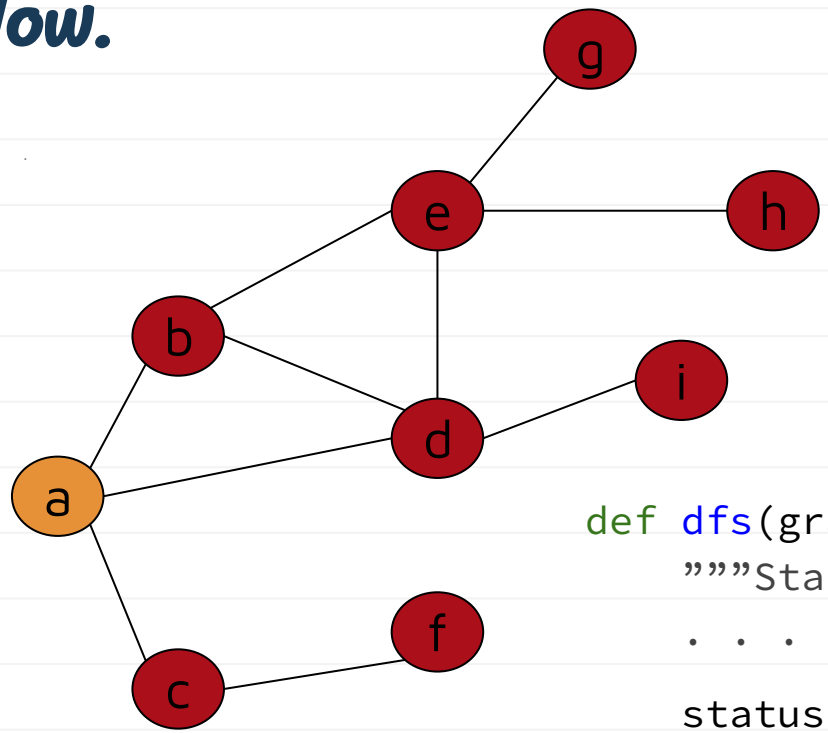
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
  dfs(f)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

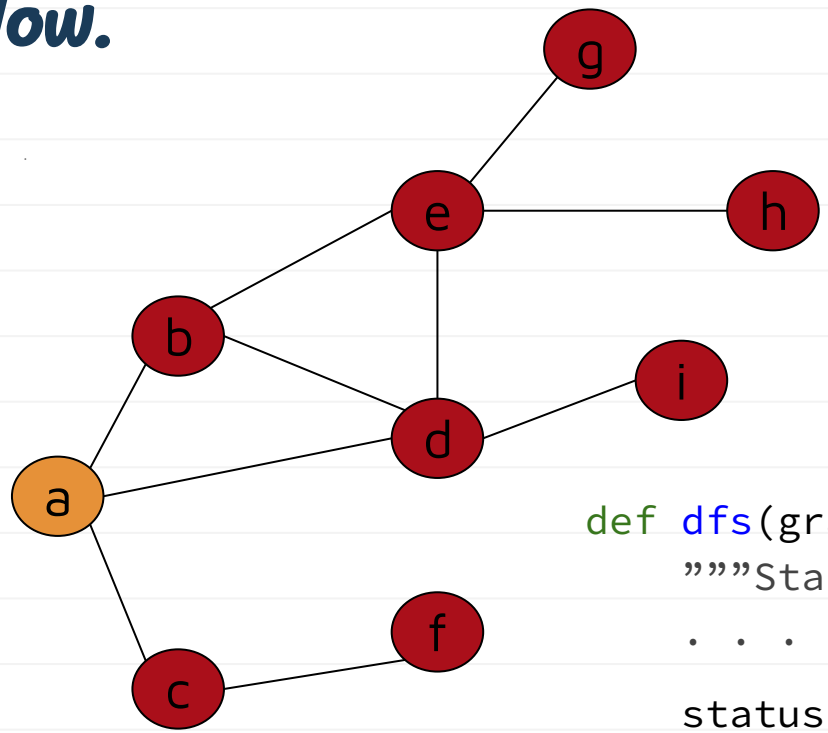
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
  dfs(f)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

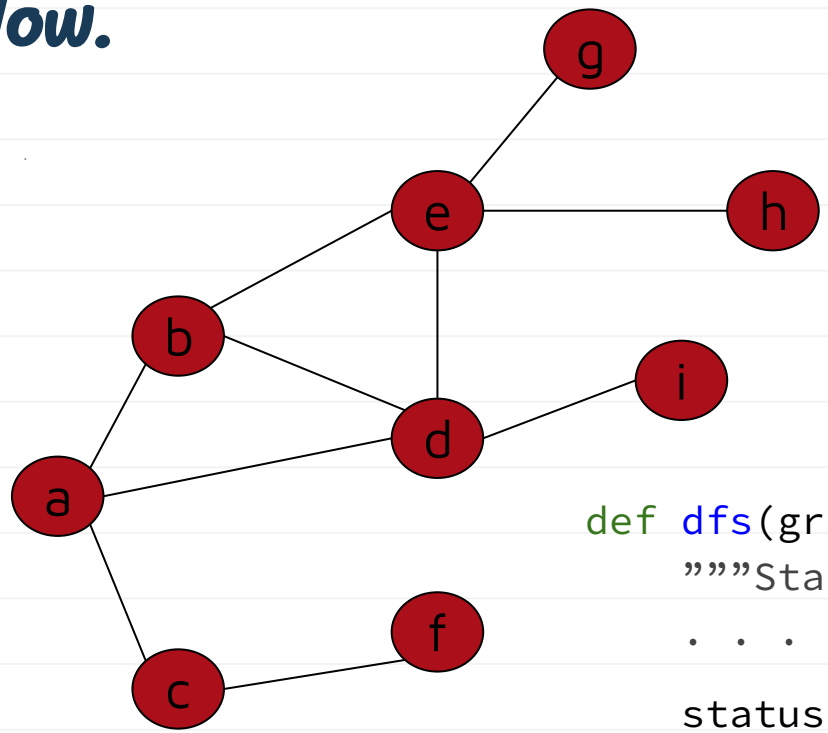
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
  dfs(c)
    dfs(f)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

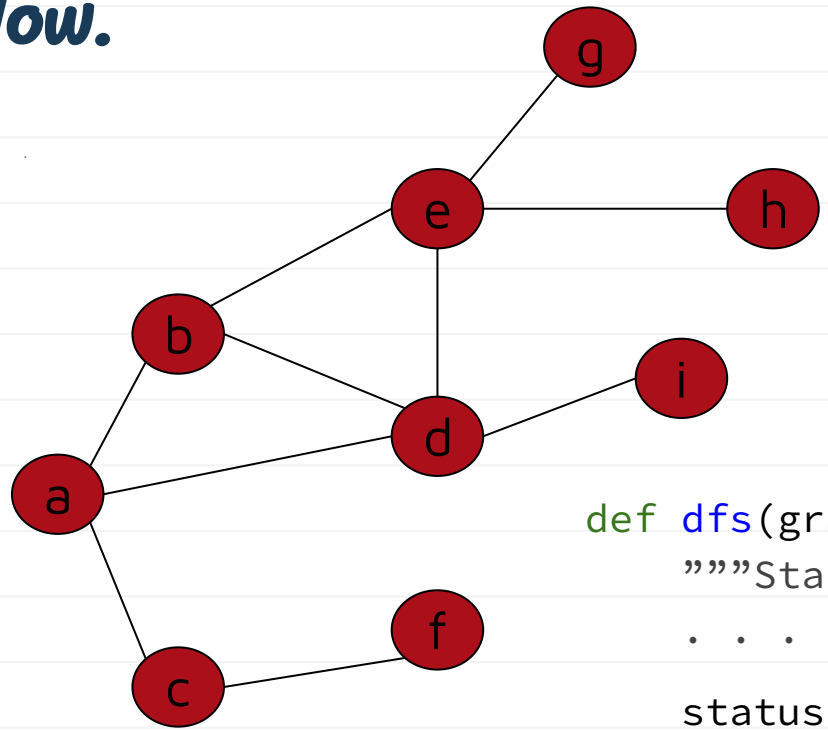
Write the nested function calls for a DFS on the graph below.




```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
  dfs(c)
    dfs(f)
```

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Write the nested function calls for a DFS on the graph below.



~~dfs(a)~~
~~dfs(b)~~
~~dfs(d)~~
~~dfs(e)~~
~~dfs(g)~~
~~dfs(h)~~
~~dfs(i)~~
~~dfs(e)~~
~~dfs(f)~~

```
def dfs(graph, u, status = None):
    """Start a DFS at `u`."""
    . . .

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Differences

- **In BFS**, we "finish" a node u before moving on to the next.
- **In DFS**, we go to many other nodes, but "come back" to u.

- We'll see that the nested structure of the **recursive function calls** gives us useful new information about the graph's structure.

Full DFS

- dfs(u) will visit all nodes **reachable** from u.
  - But not all nodes may be reachable from u!
- To visit **all** nodes in graph, need full DFS.

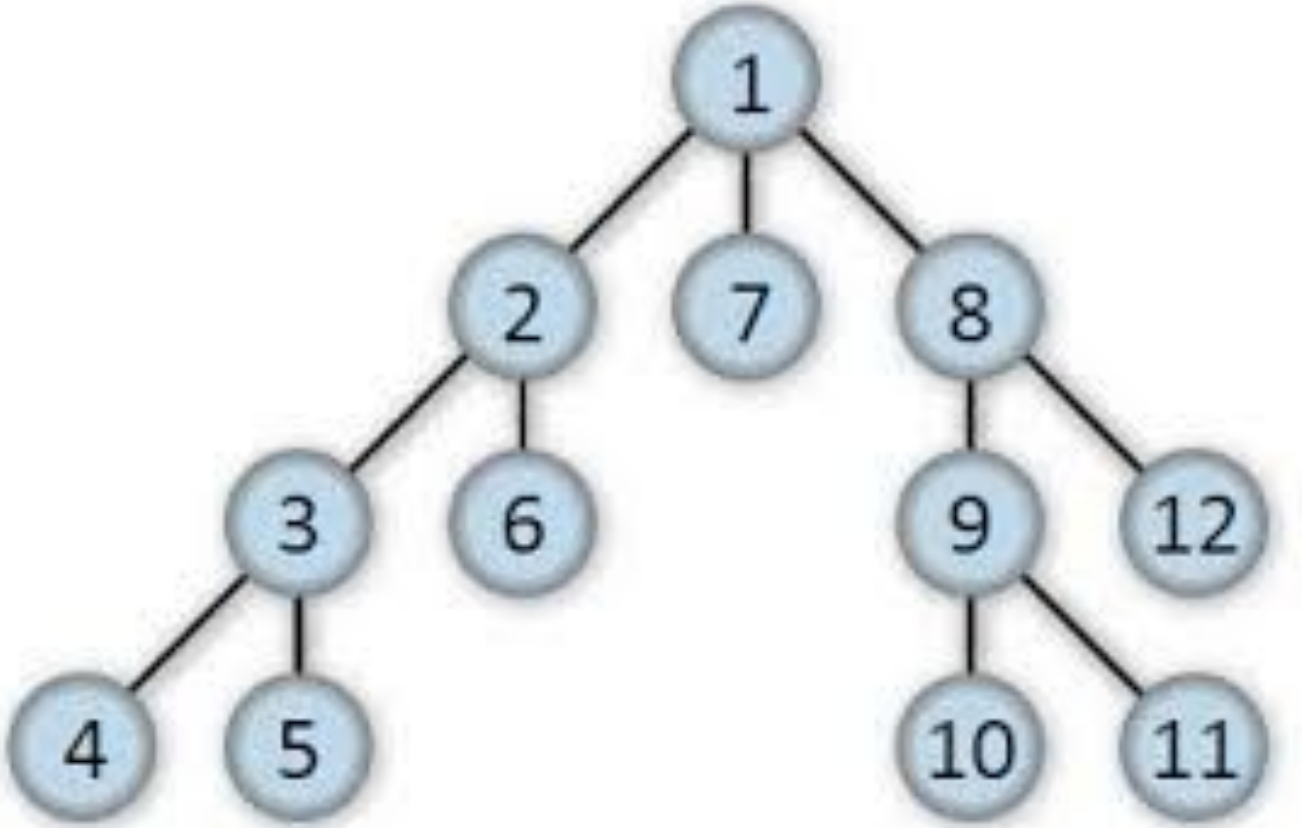
```
def full_dfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
    if status[node] == 'undiscovered'
        dfs(graph, node, status)
```

```python
def full_dfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            dfs(graph, node, status)


def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Time Complexity

- In a full DFS:
  - dfs called on each node exactly once.
  - Similar to BFS, each edge is explored exactly:
    - once if directed
    - twice if undirected
- Time: $\Theta(V + E)$, **just like BFS.**

BFS vs DFS Check



BFS (1): What do you expect to see first?

A: Level (2, 7, 8)

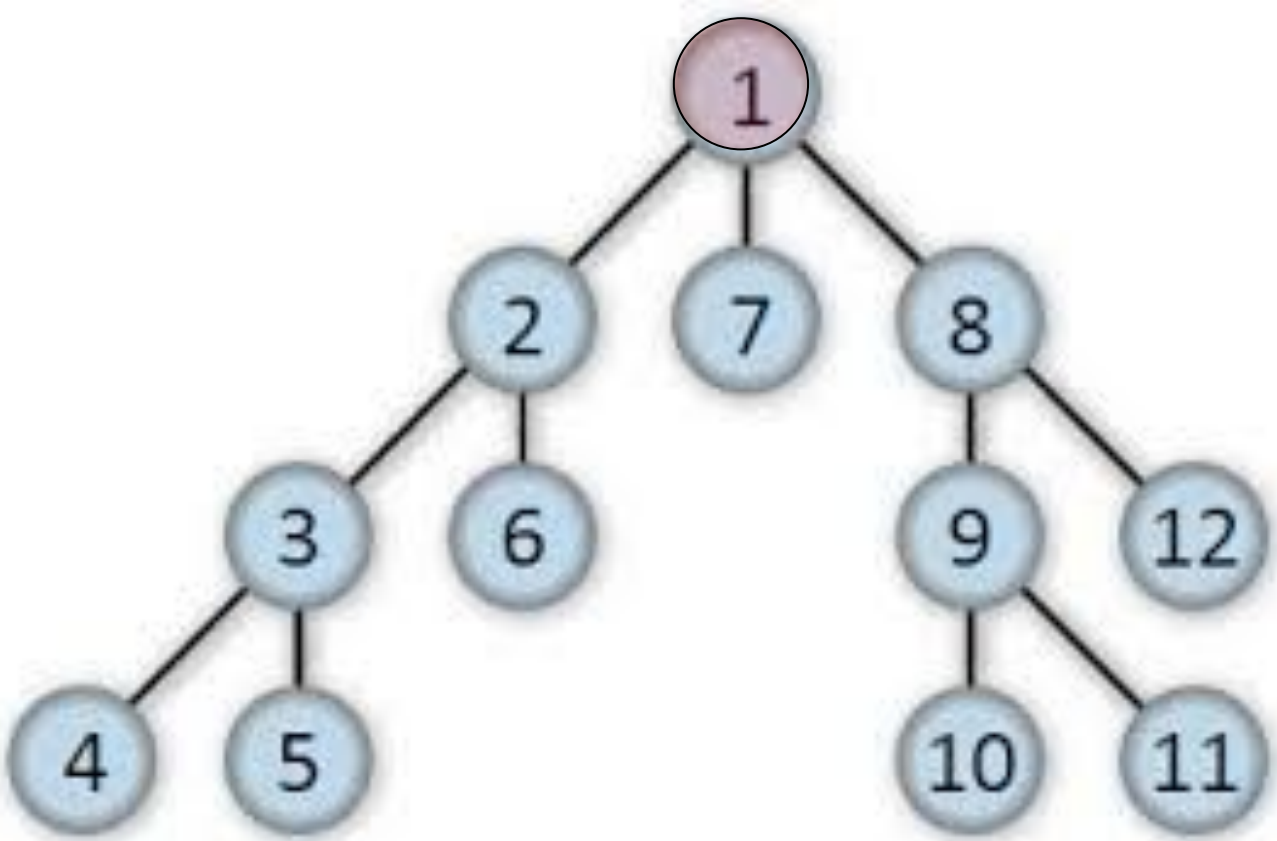
B: Level (3, 6, 9, 12)

C: Branch (4, 3, 2, 1)

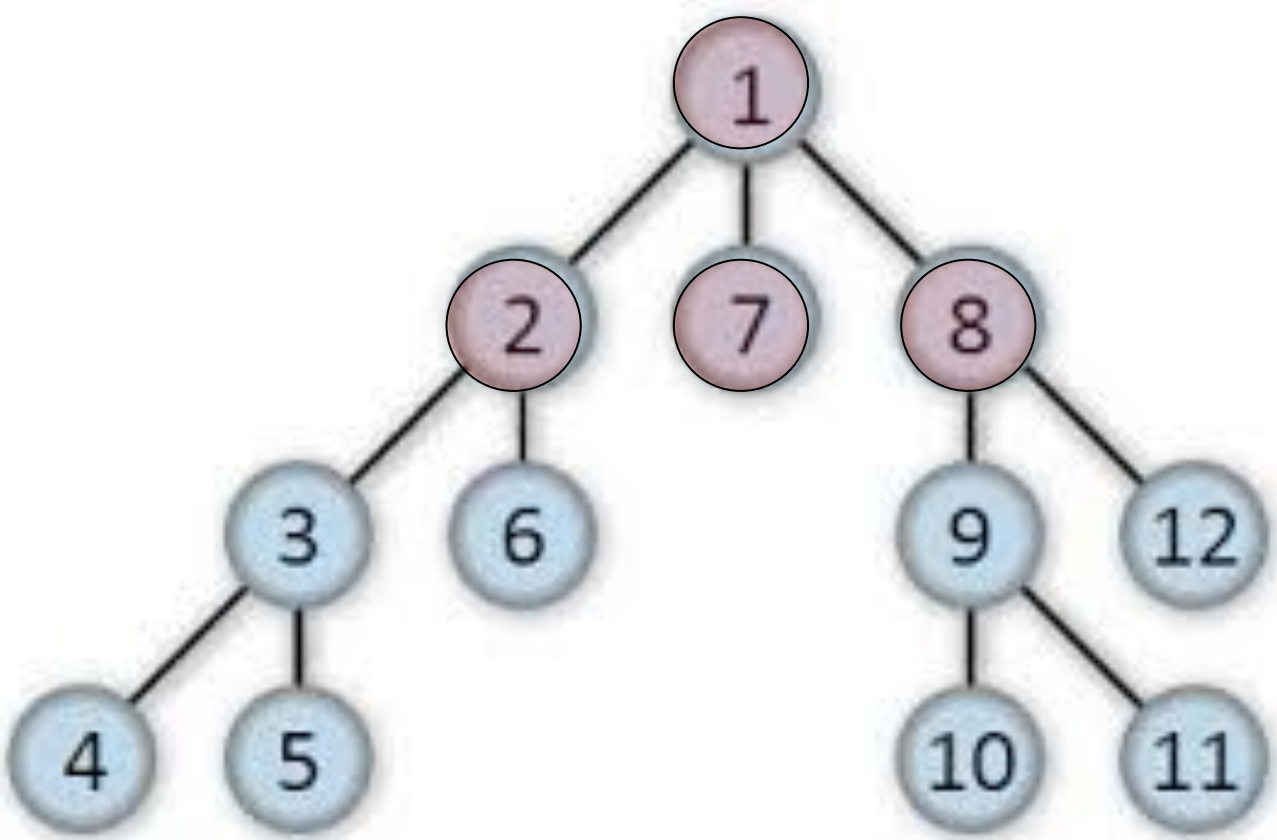
D: Branch (12, 8 , 1)

E: Something else

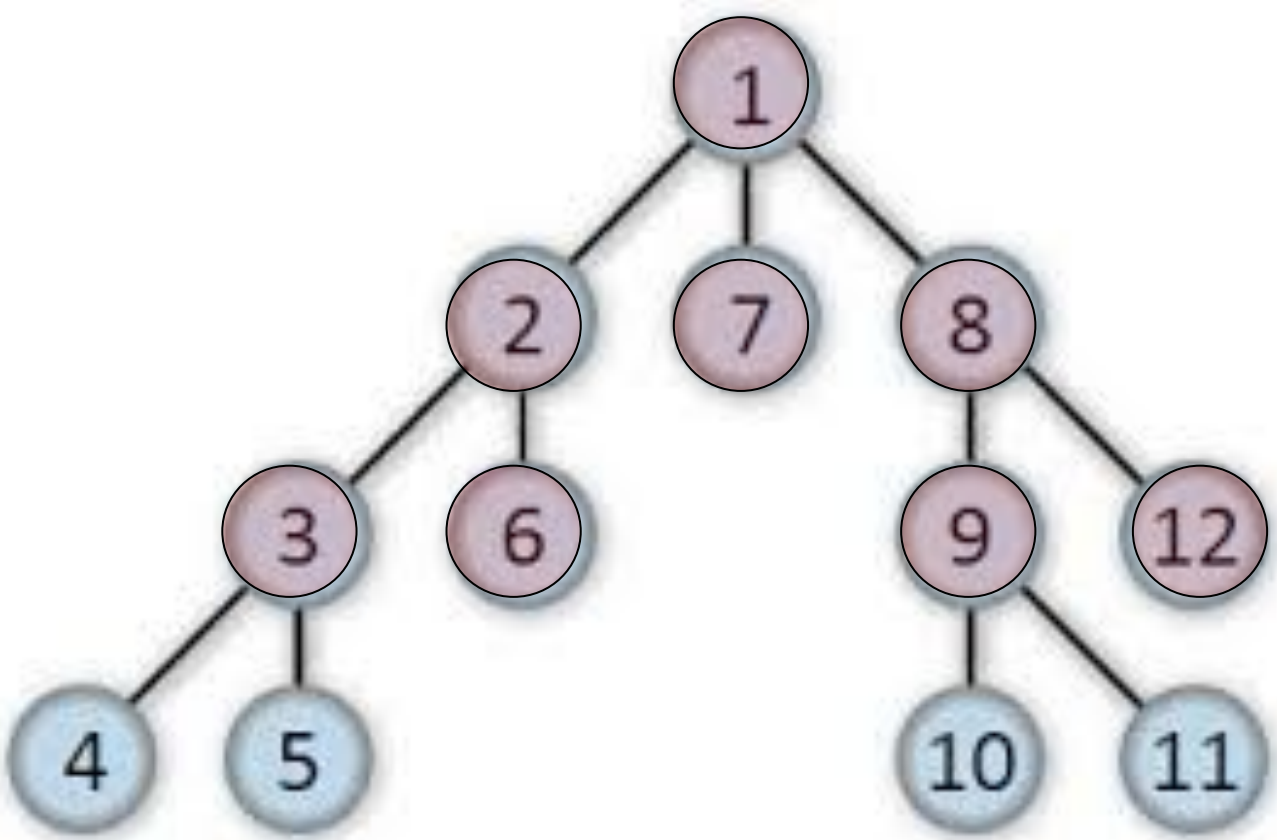
BFS(1)



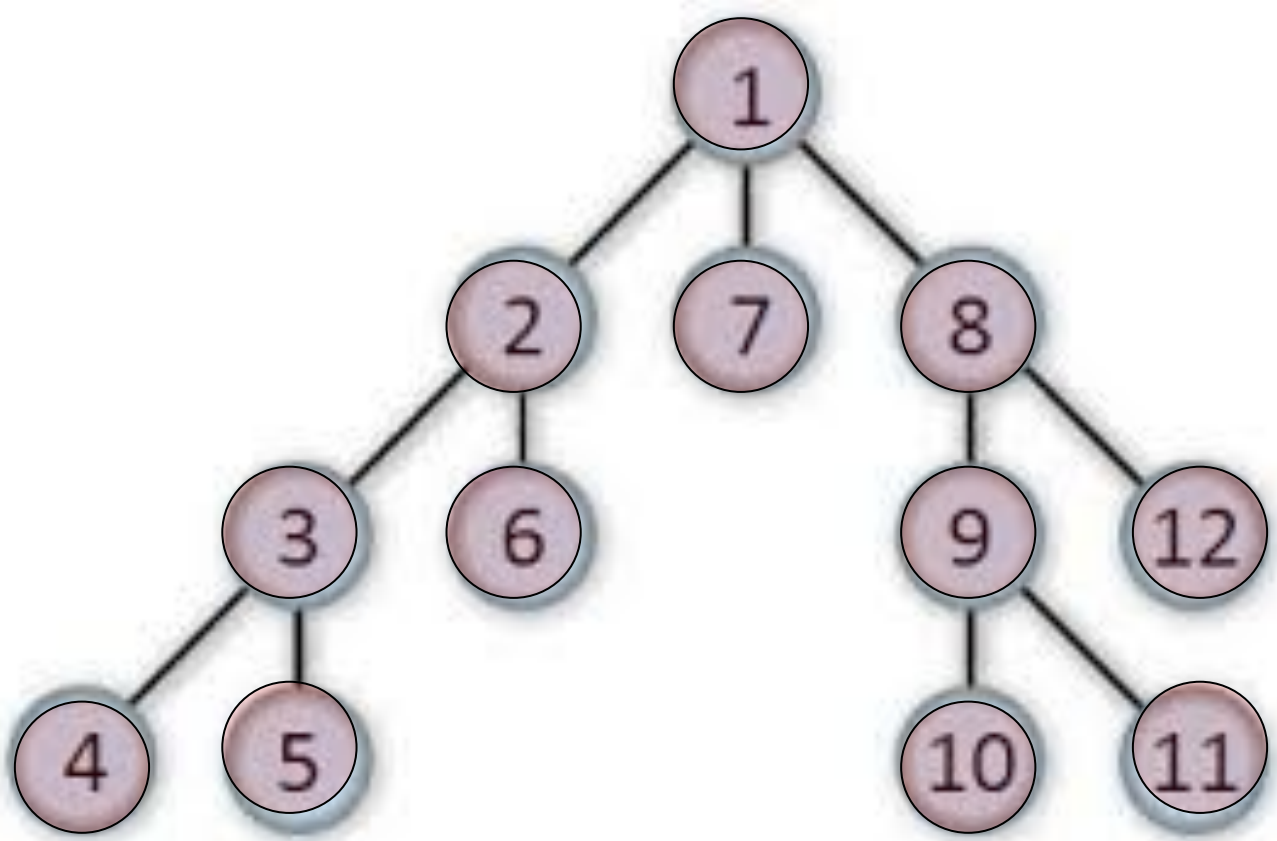
BFS(1)



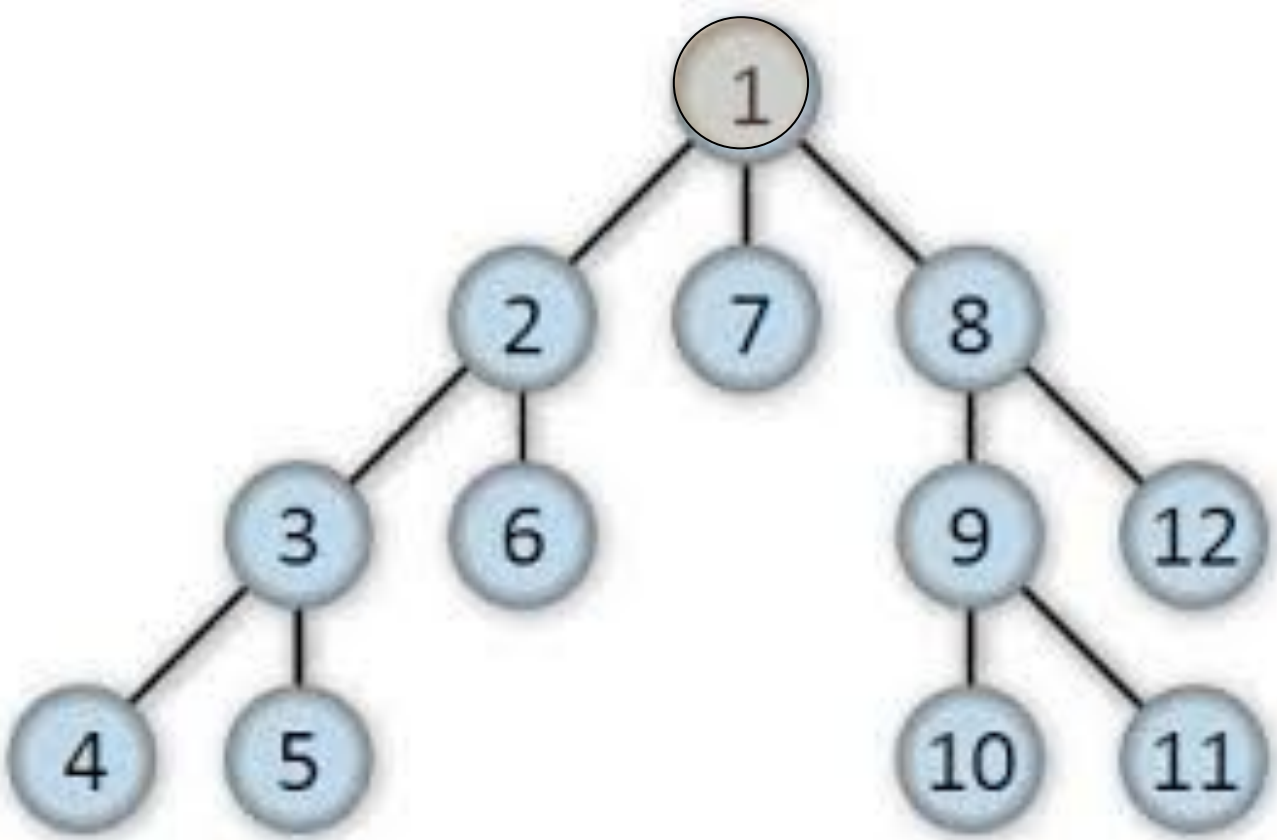
BFS(1)



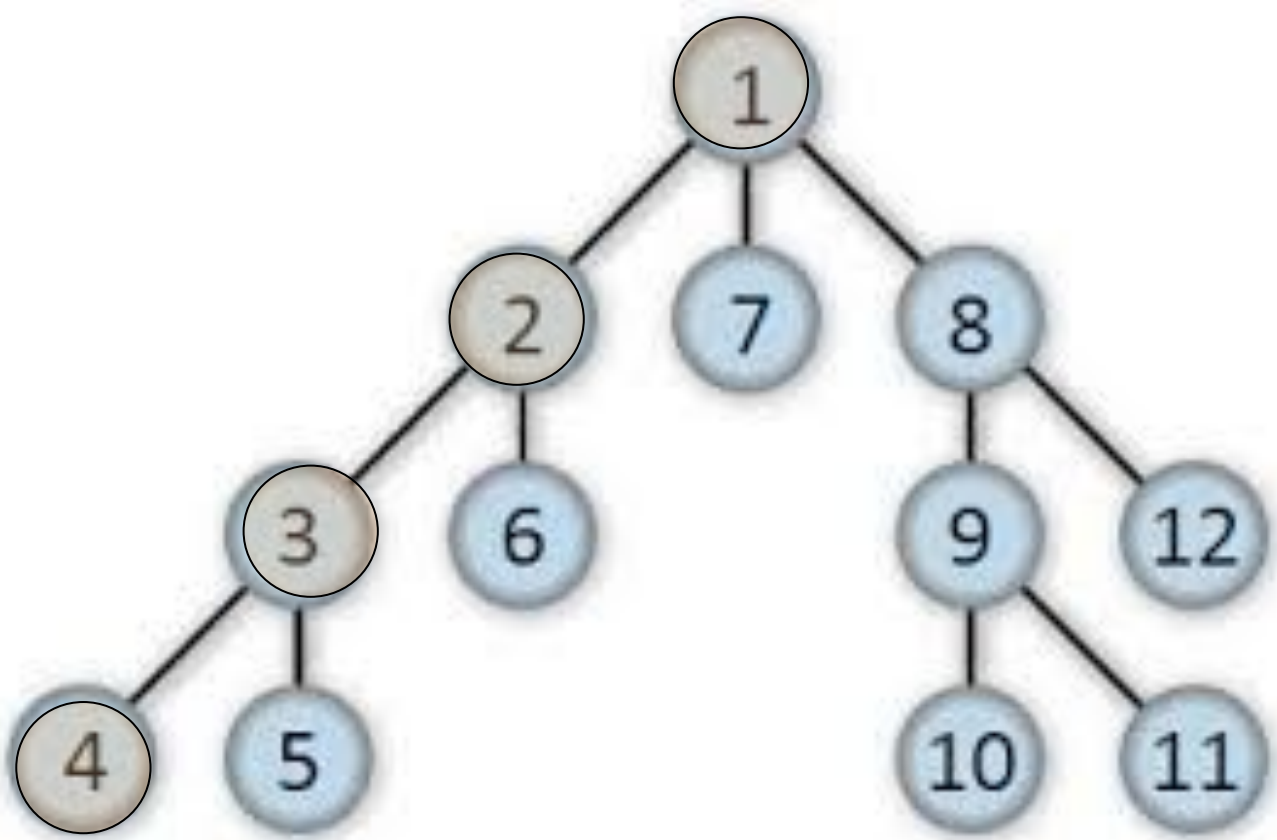
BFS(1)



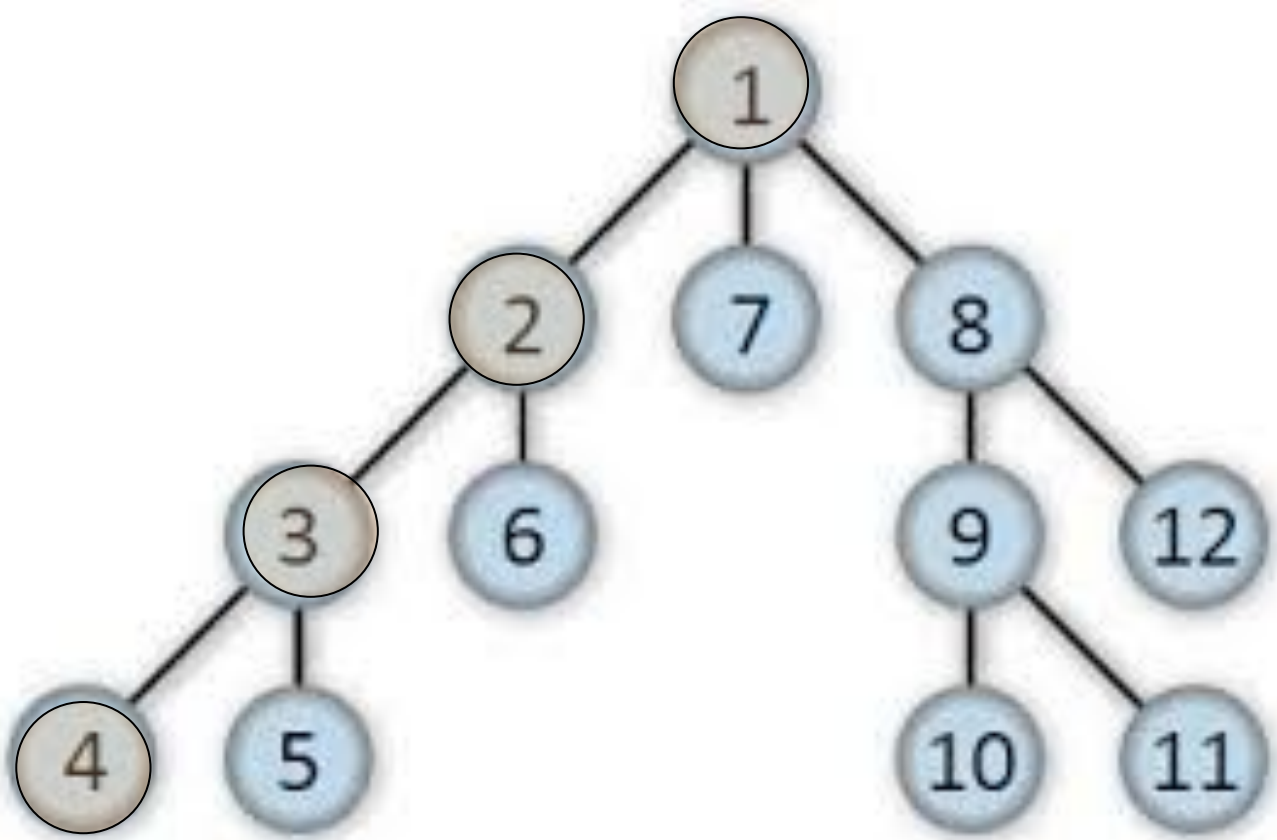
DFS(1)



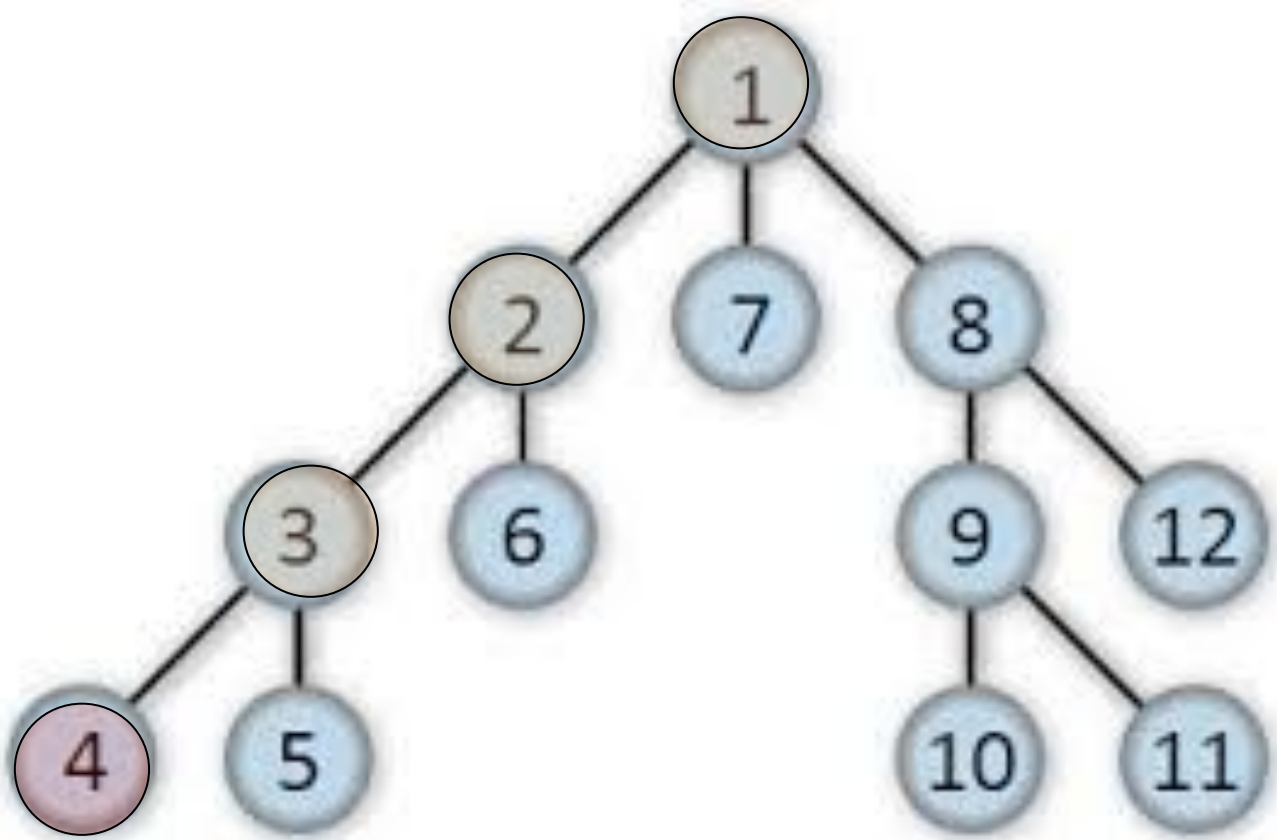
DFS(1)



DFS(1)

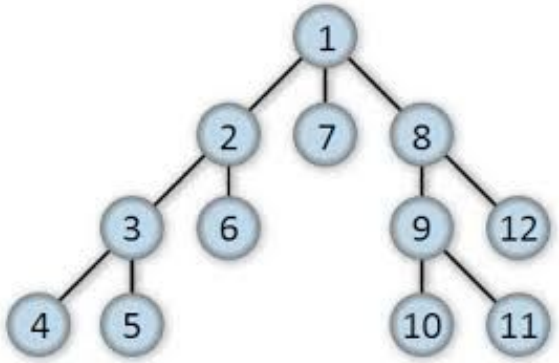


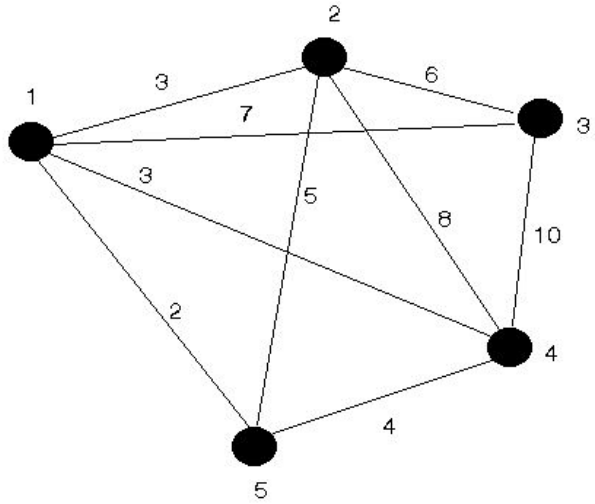
DFS(1)

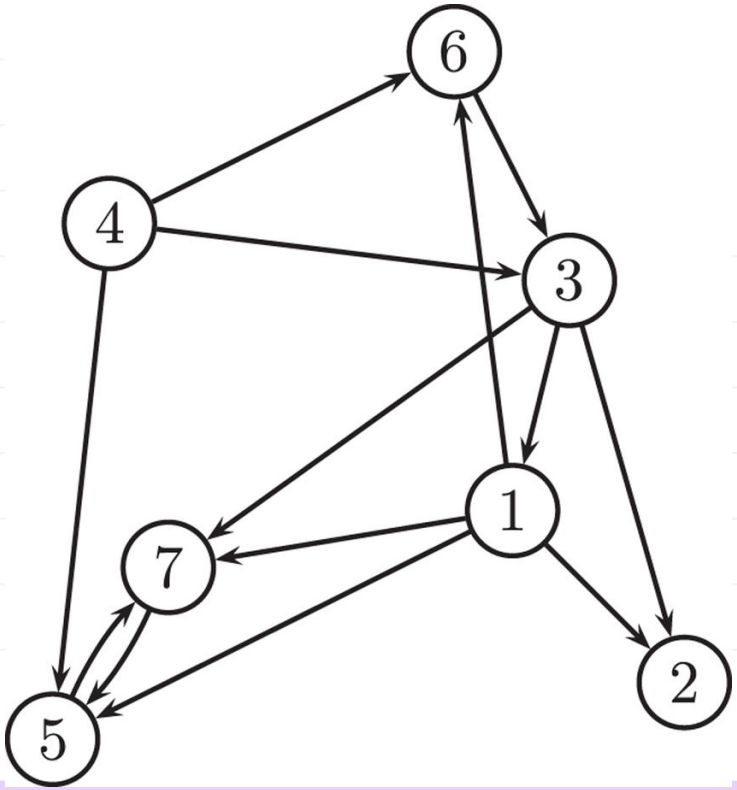


Nesting Properties of DFS

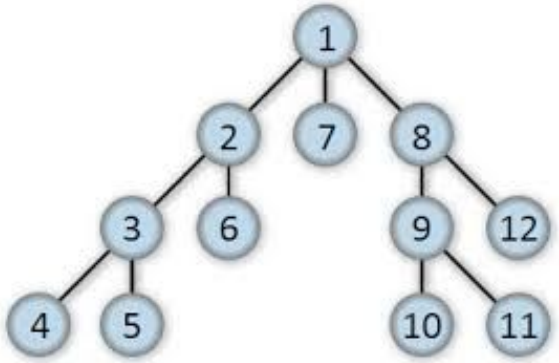
Motivation

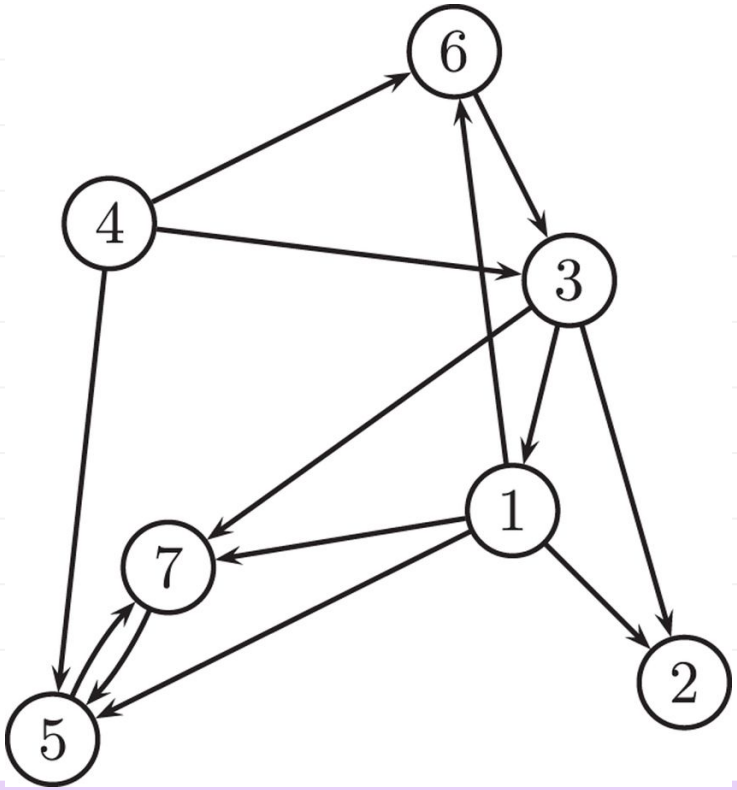


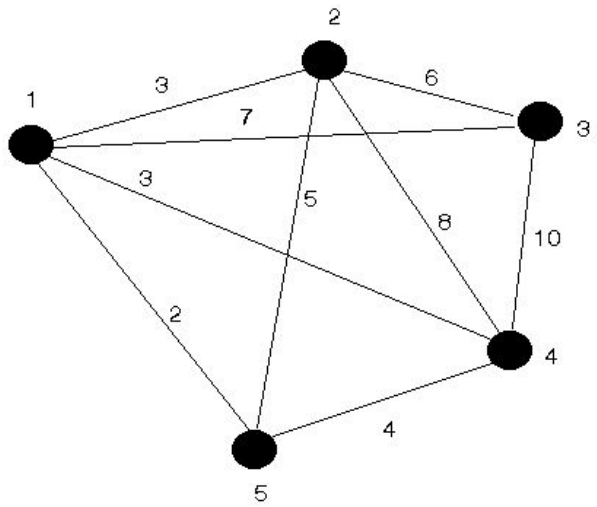




Is there a cycle?





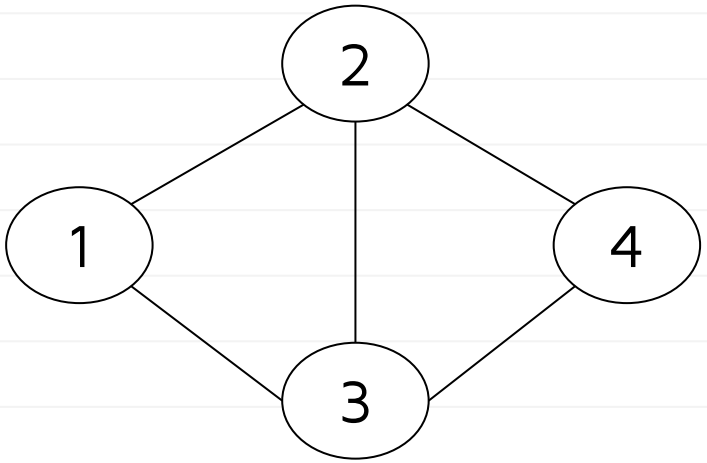


Why do we care?

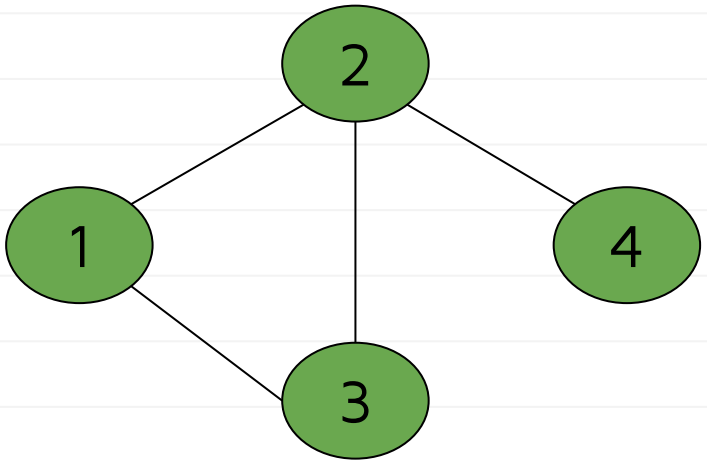
- **In computer programs (dependency graphs):**
  - If you have tasks that depend on each other, a cycle means a circular dependency — *task A depends on B, and B depends on A* — so you can't complete any of them. Detecting the cycle helps prevent deadlocks or infinite loops.
- **In scheduling or planning:**
  - When building a project schedule or course prerequisite list, a cycle means it's impossible to order the tasks — e.g., "CS101 requires CS201, and CS201 requires CS101." You can't take either first.

Last time: DFS

mic

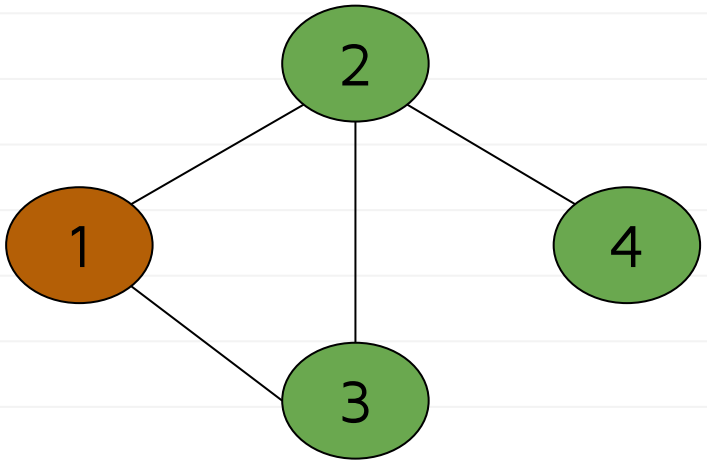


Last time: DFS



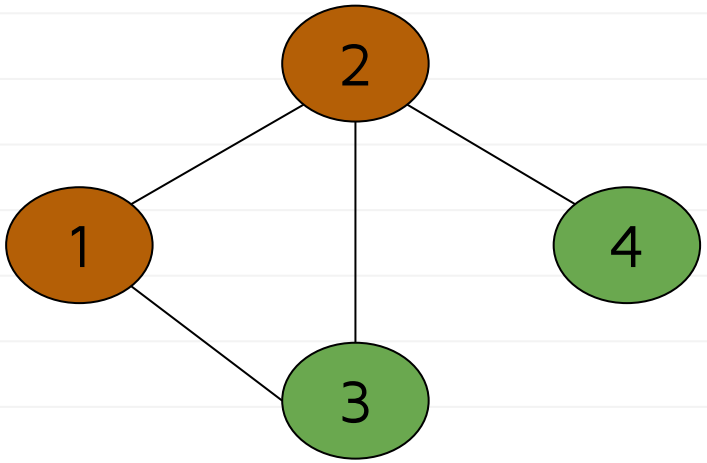

```
dfs(1)
```

Last time: DFS



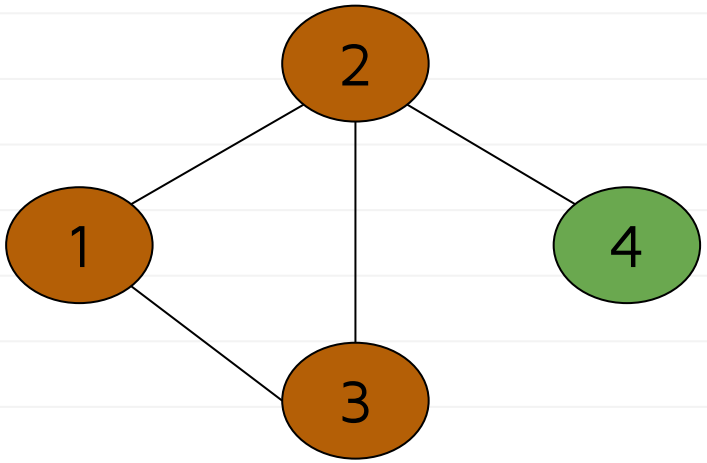

```
dfs(1)
```

Last time: DFS



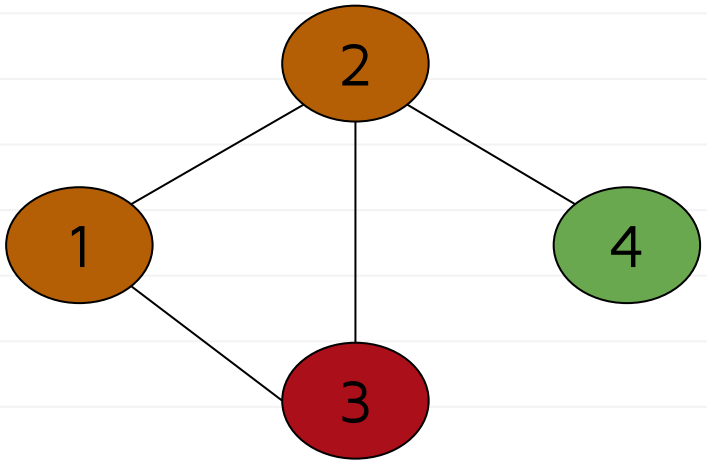

```
dfs(1)

    dfs(2)
```

Last time: DFS



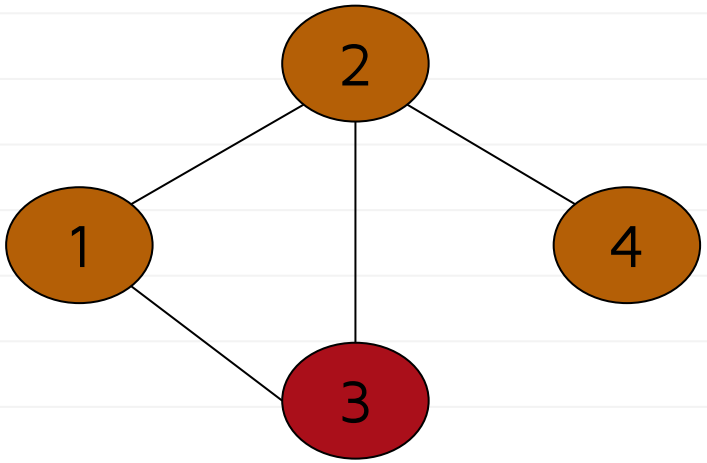

```
dfs(1)
    dfs(2)
        dfs(3)
```

Last time: DFS



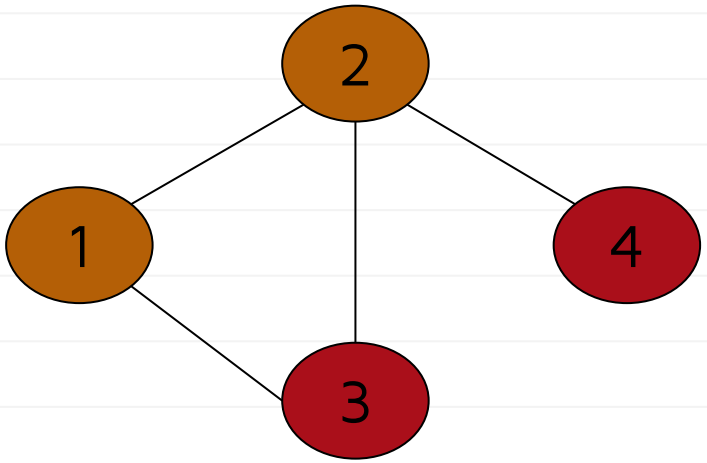

```
dfs(1)
    dfs(2)
        dfs(3)
```

Last time: DFS



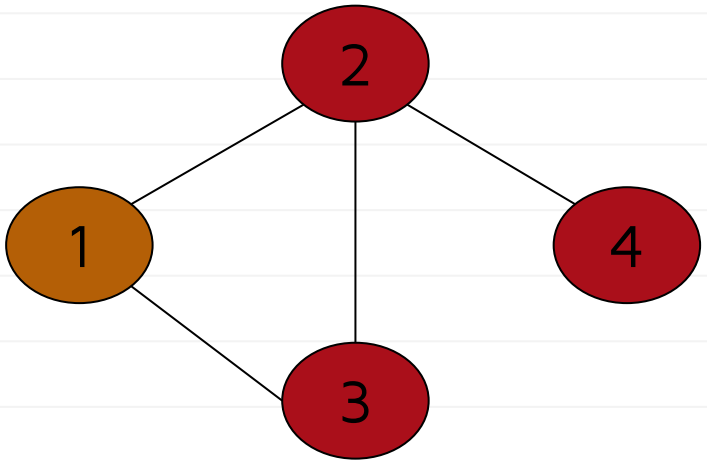

```
dfs(1)
    dfs(2)
        dfs(3)
        dfs(4)
```

Last time: DFS



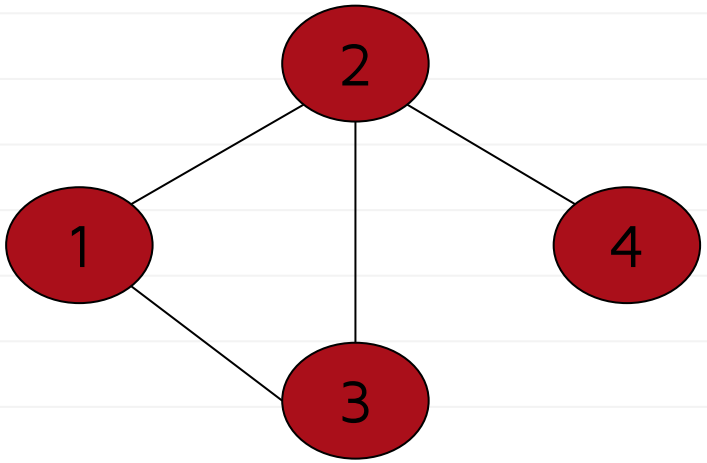

```
dfs(1)
    dfs(2)
        dfs(3)
        dfs(4)
```

Last time: DFS




```
dfs(1)
    dfs(2)
        dfs(3)
        dfs(4)
```

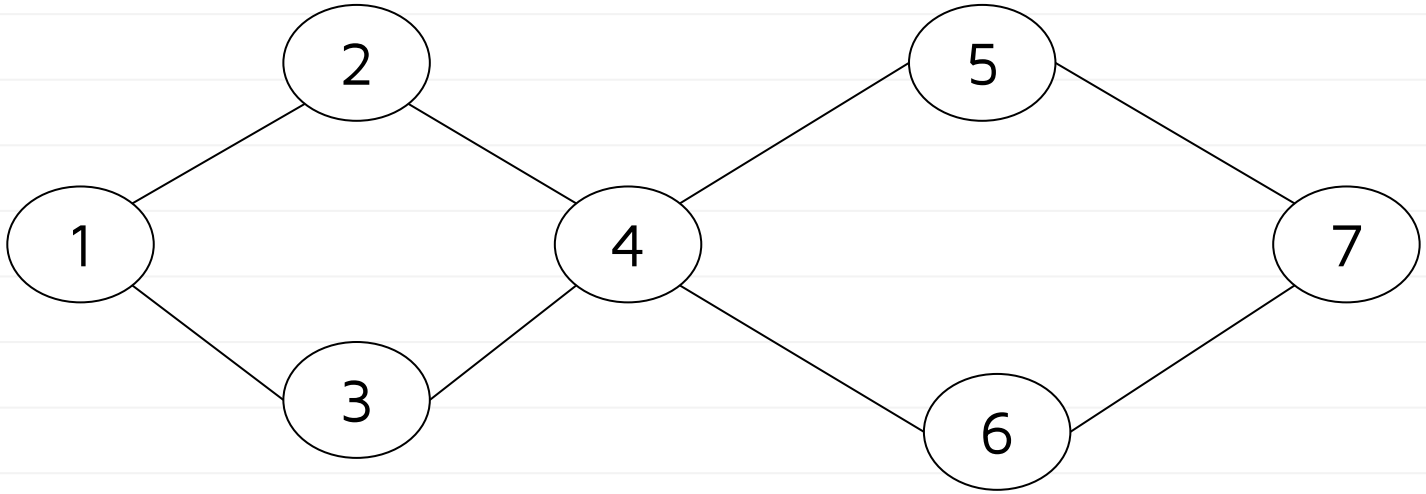
Last time: DFS




```
dfs(1)
    dfs(2)
        dfs(3)
        dfs(4)
```

Exercise #1

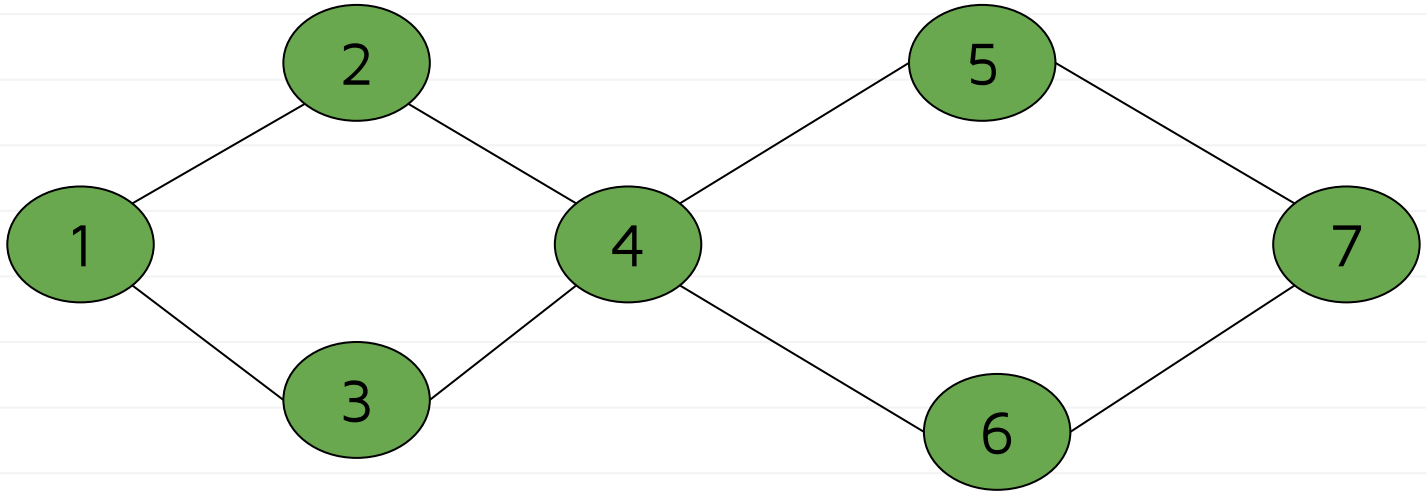
True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.




```
dfs(u)
 ....
   dfs(v)
```

Exercise

True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called *during* `dfs(u)`.

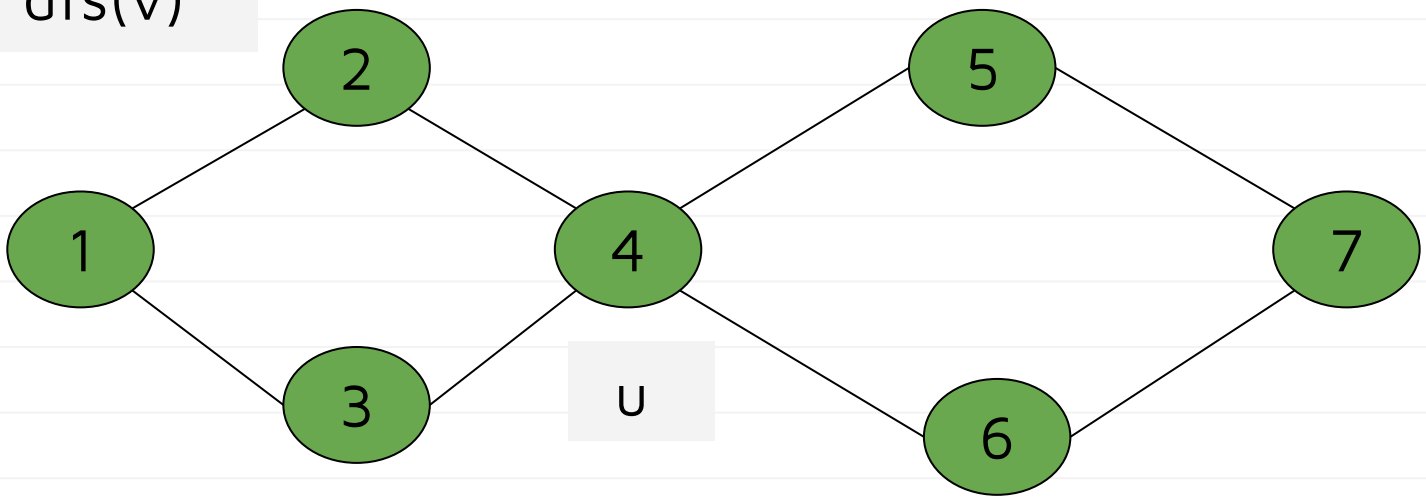



```
dfs(u)
....
    dfs(v)
```

True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

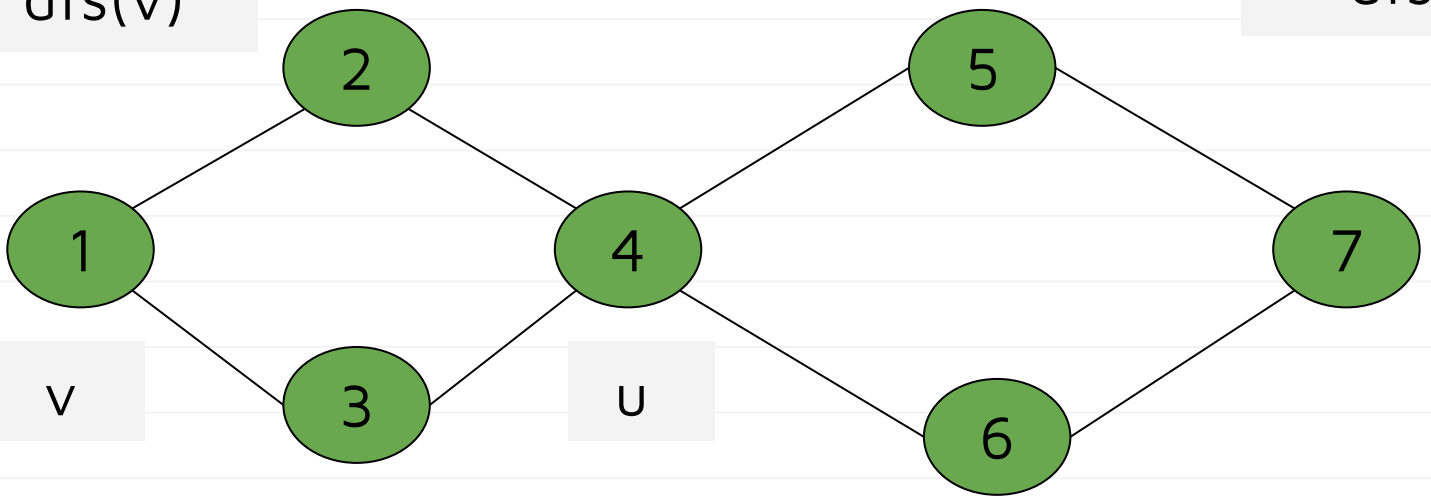
dfs(4)



True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
 ....
    dfs(1)
```

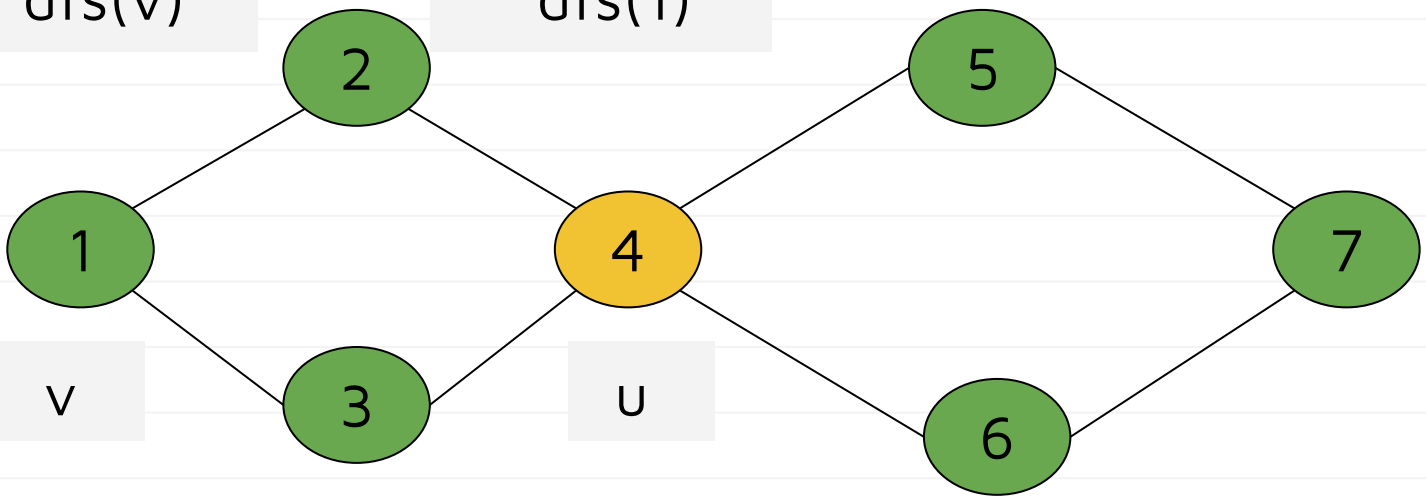


True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
 ....
    dfs(1)
```

```
dfs(4)
```

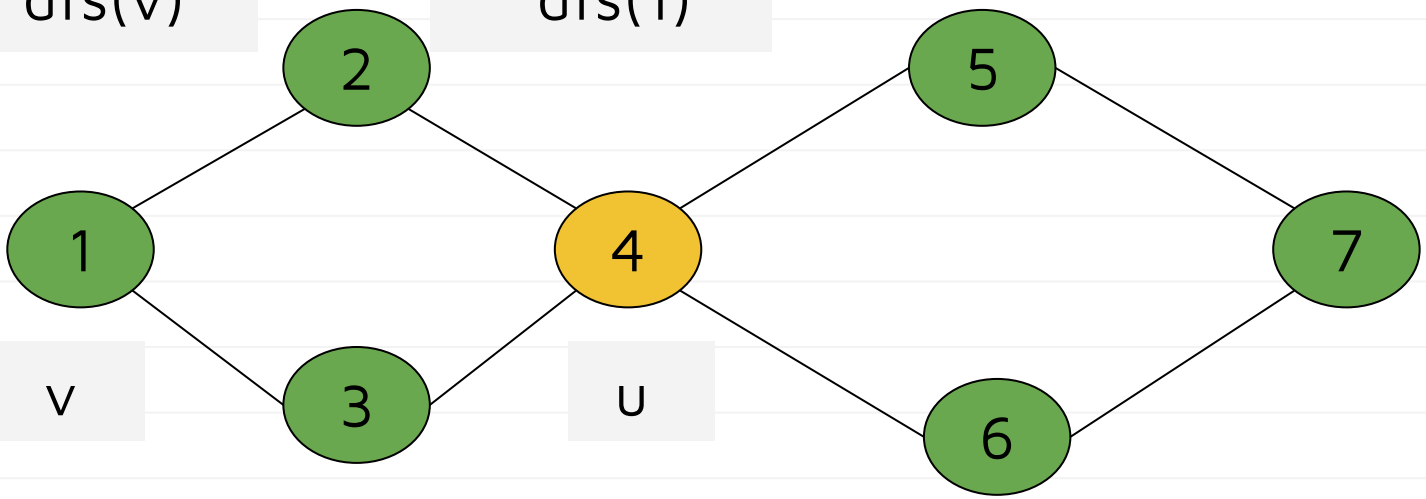


True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
 ....
    dfs(1)
```

```
dfs(4)
    dfs(2)
```

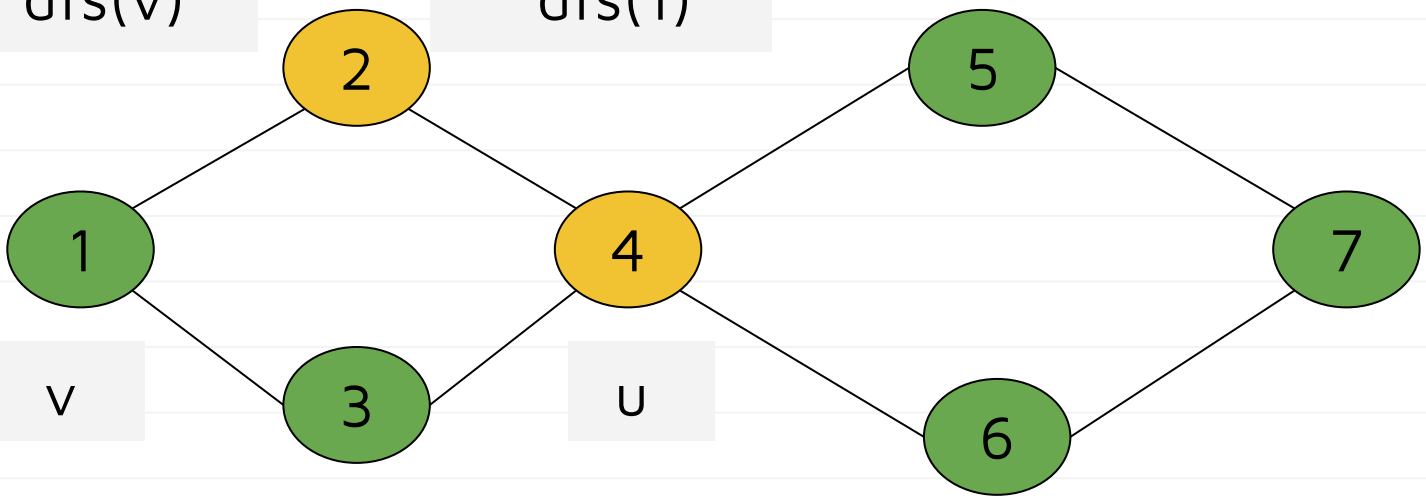


True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
 ....
    dfs(1)
```

```
dfs(4)
    dfs(2)
```

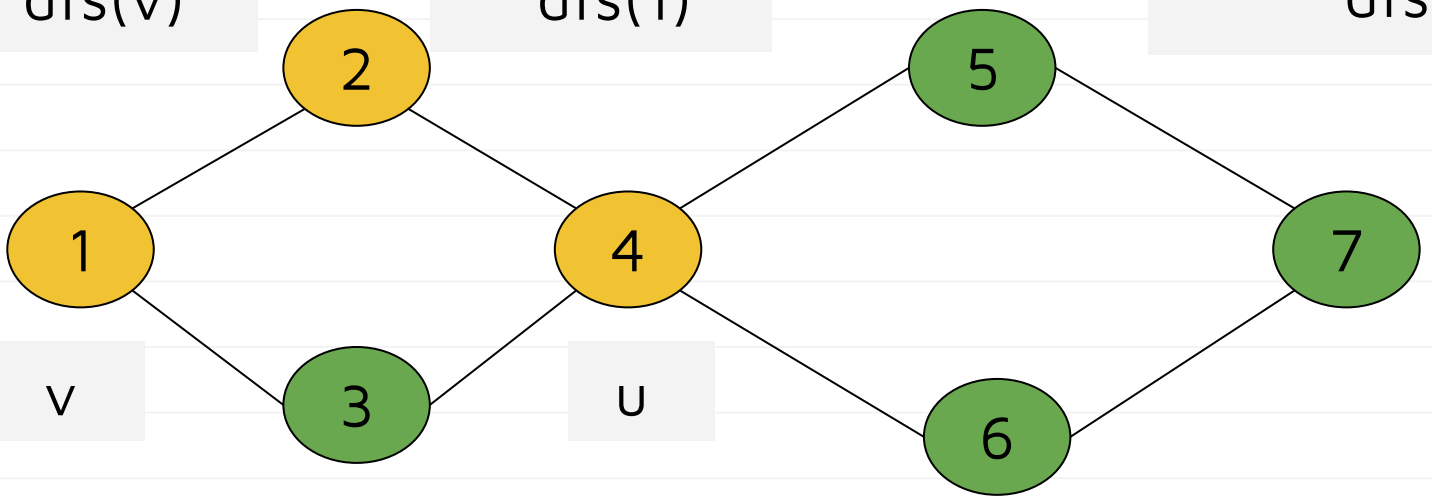


True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
 ....
    dfs(1)
```

```
dfs(4)
    dfs(2)
        dfs(1)
```

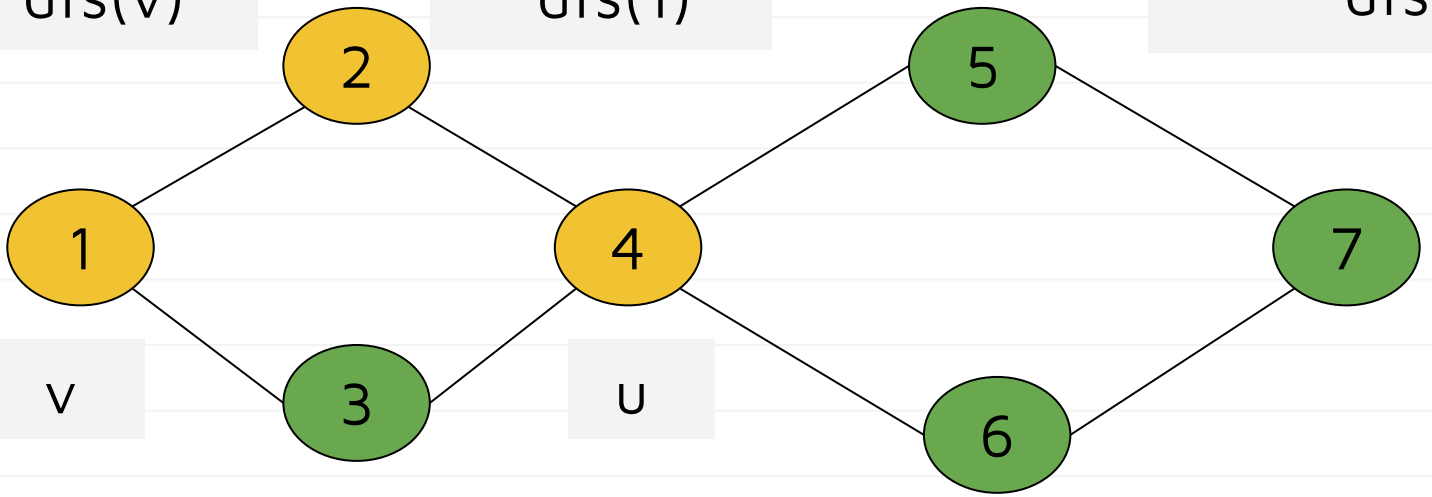


True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
  ....
     dfs(1)
```

```
dfs(4)
    dfs(2)
        dfs(1)
```



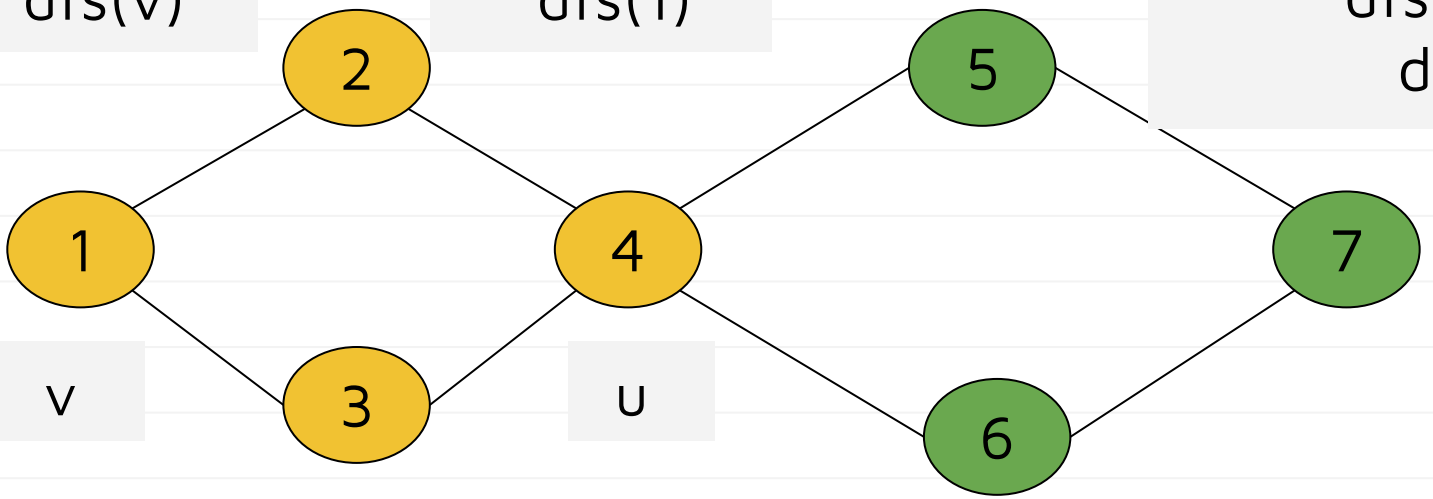
So far it works out

True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
  ....
     dfs(1)
```

```
dfs(4)
    dfs(2)
        dfs(1)
            dfs(3)
```

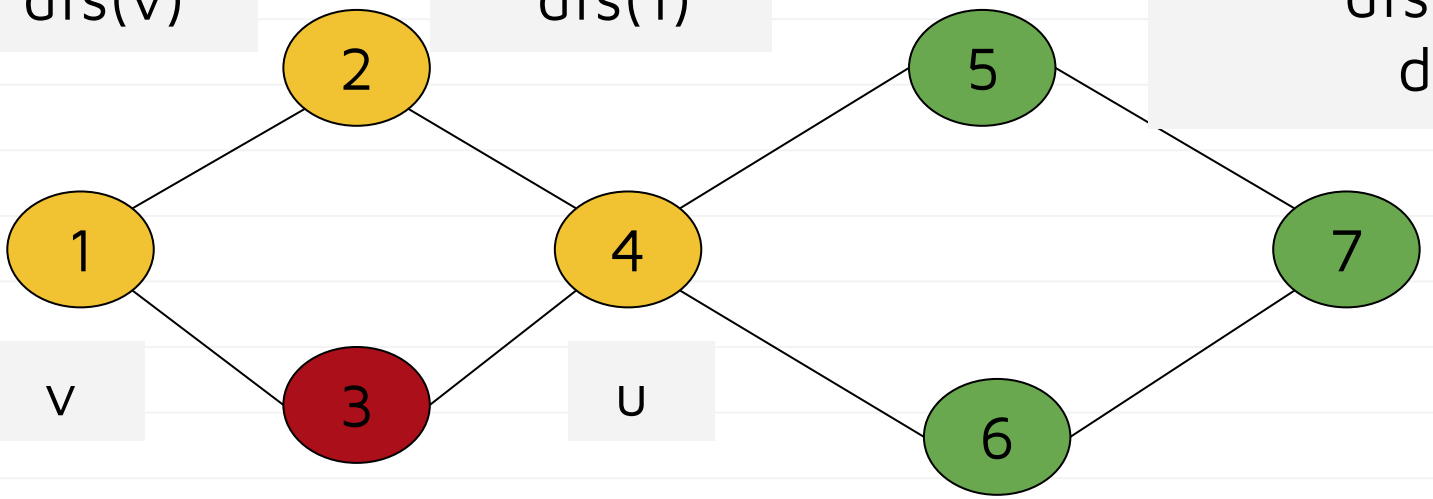


True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
  ....
     dfs(1)
```

```
dfs(4)
    dfs(2)
        dfs(1)
            dfs(3)
```

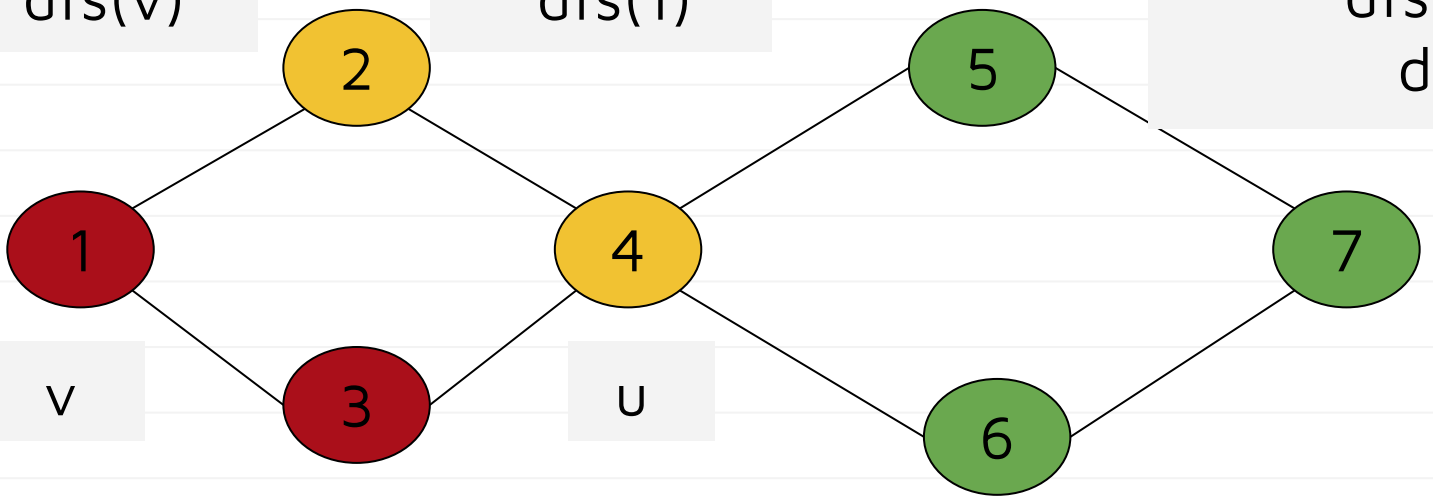


True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
  ....
     dfs(1)
```

```
dfs(4)
    dfs(2)
        dfs(1)
            dfs(3)
```

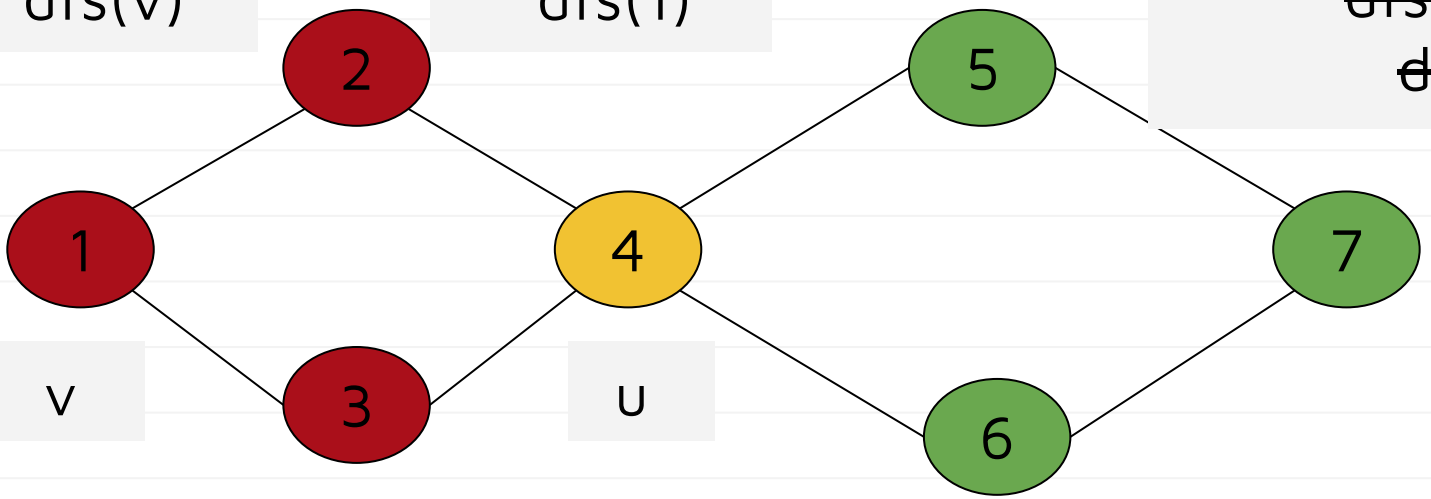


True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

dfs(u)
....
dfs(v)

dfs(4)
....
dfs(1)

dfs(4)
~~dfs(2)~~
~~dfs(1)~~
~~dfs(3)~~

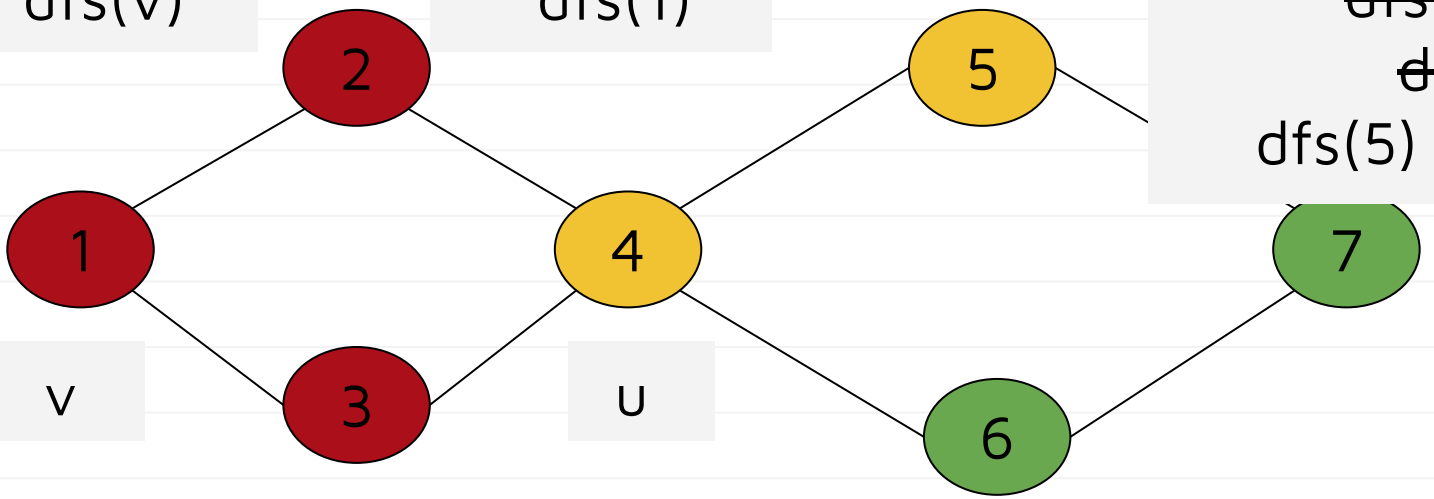


True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

```
dfs(u)
 ....
    dfs(v)
```

```
dfs(4)
 ....
    dfs(1)
```

```
dfs(4)
    dfs(2)
        dfs(1)
            dfs(3)
    dfs(5)
```



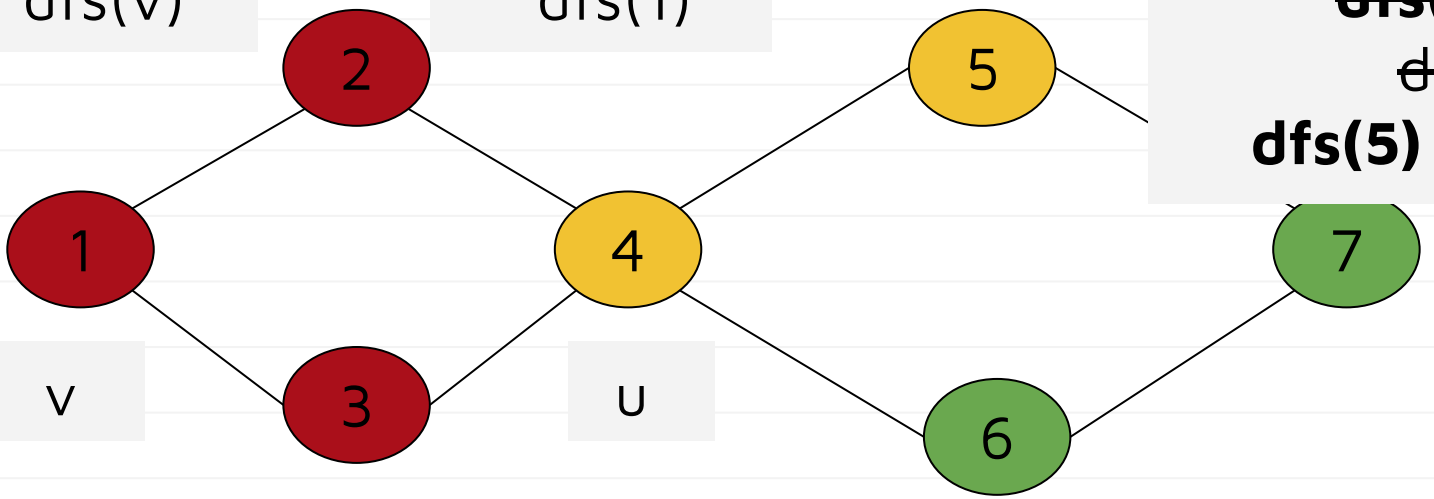
And now we can see that it is **False**.

True or **False**: if v is *reachable* from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.

dfs(u)
....
dfs(v)

dfs(4)
....
dfs(1)

dfs(4)
~~dfs(2)~~
~~**dfs(1)**~~
~~dfs(3)~~
dfs(5)



And now we can see that it is False.

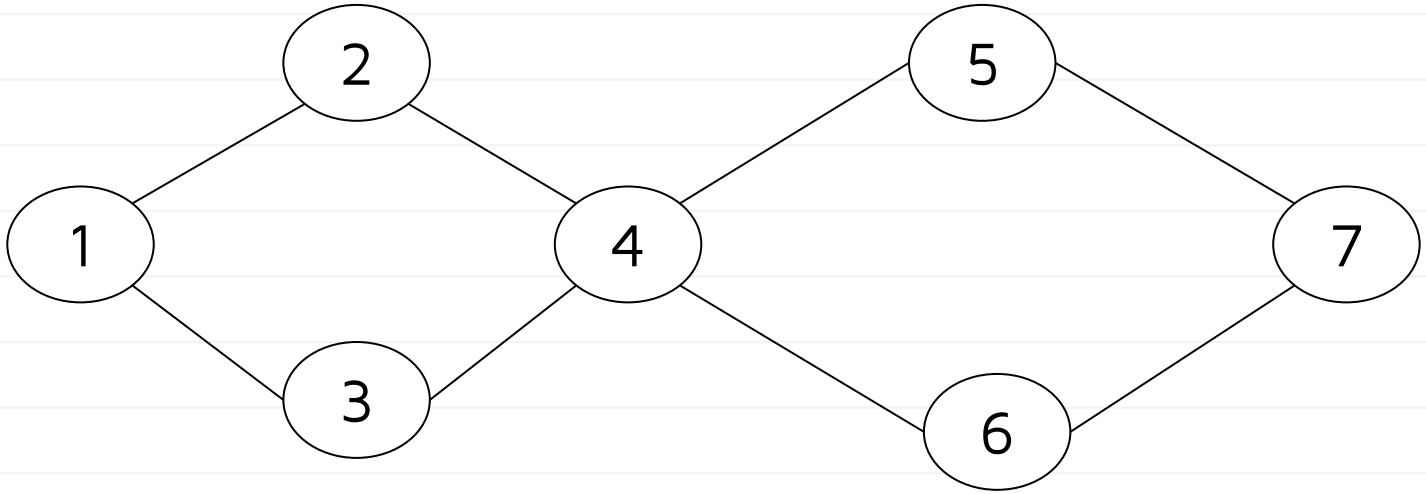
False!

- Suppose `dfs(4)` is the root call.
- When `dfs(1)` is called, node 5 is **undiscovered**
- But `dfs(5)` is **not** called during `dfs(1)`.

However..

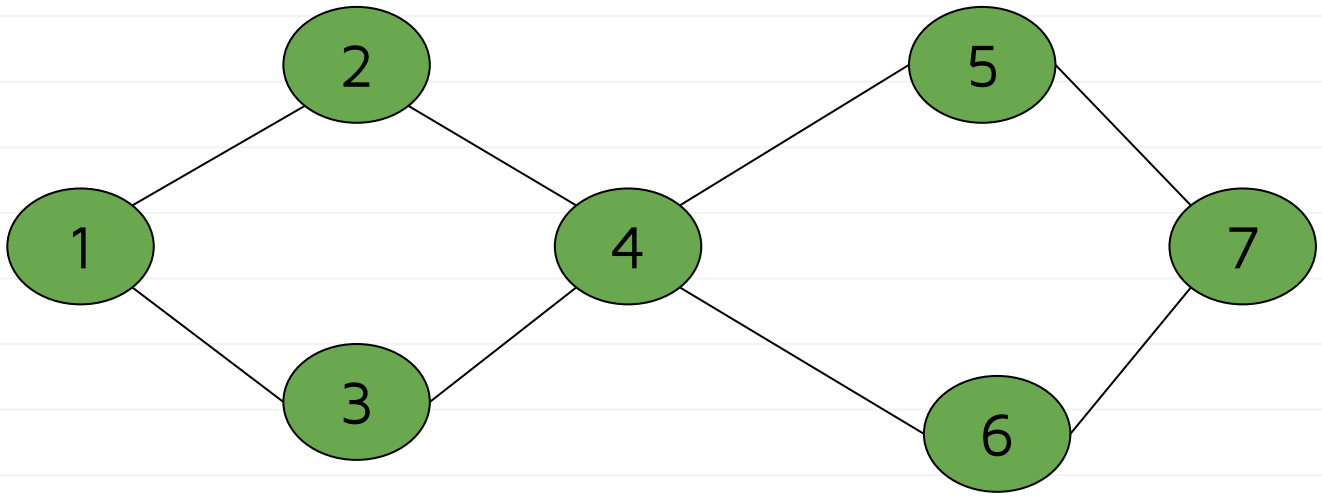
This intuition is correct if there is a path of **undiscovered** nodes from u to v when dfs(u) is called.

```
dfs(u)
 ....
     dfs(v)
```



However..

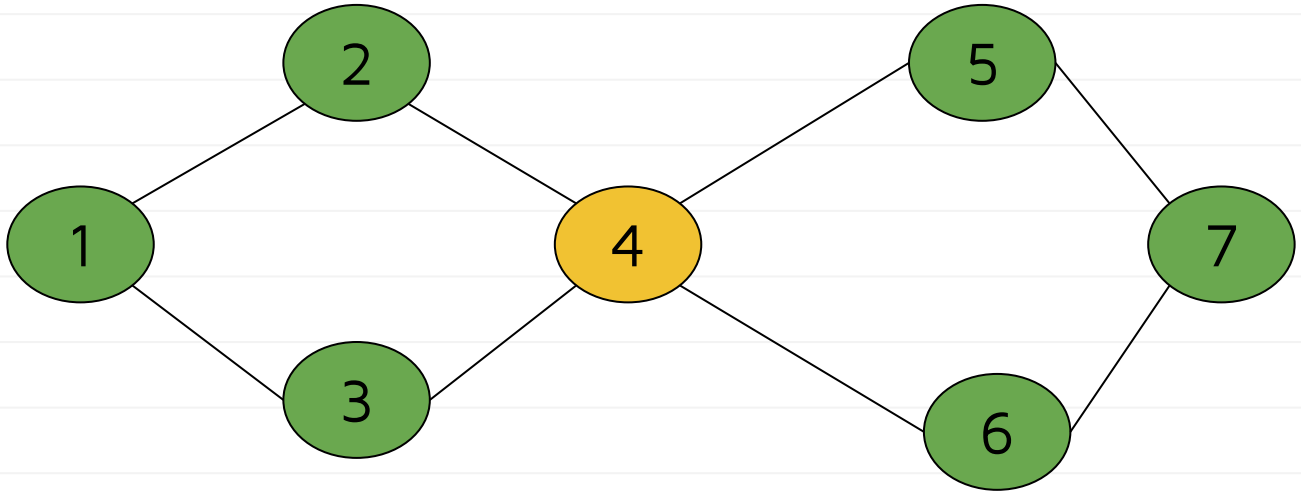
This intuition is correct if there is a path of **undiscovered** nodes from u to v when dfs(u) is called.



However..

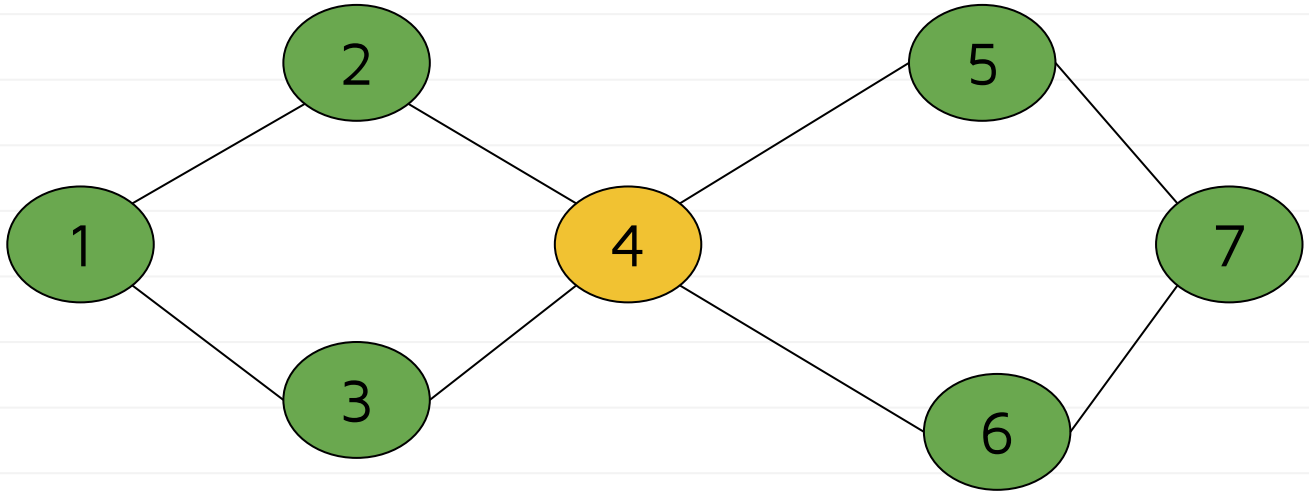
This intuition is correct if there is a path of **undiscovered** nodes from u to v when dfs(u) is called.

dfs(4)



However..

This intuition is correct if there is a path of **undiscovered** nodes from u to v when dfs(u) is called.

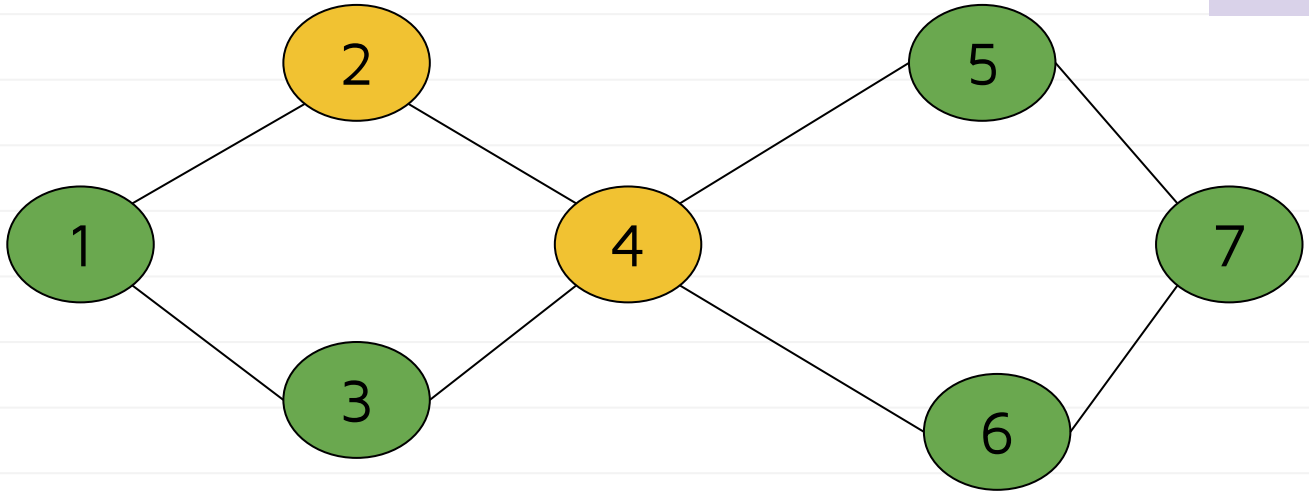


dfs(4)
- dfs(2)
  - dfs(1)
    - dfs(3)
- dfs(5)
  - dfs(7)
    - dfs(6)

However..

This intuition is correct if there is a path of **undiscovered** nodes from u to v when dfs(u) is called.

What calls are nested within the dfs(2)?

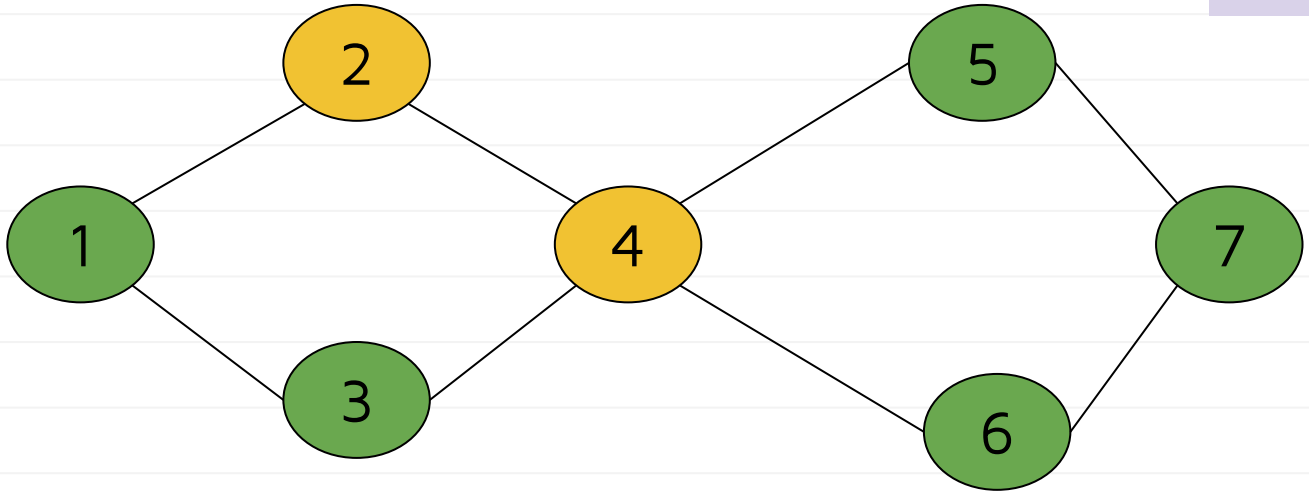


dfs(4)
dfs(2)
dfs(1)
dfs(3)
dfs(5)
dfs(7)
dfs(6)

However..

This intuition is correct if there is a path of **undiscovered** nodes from u to v when dfs(u) is called.

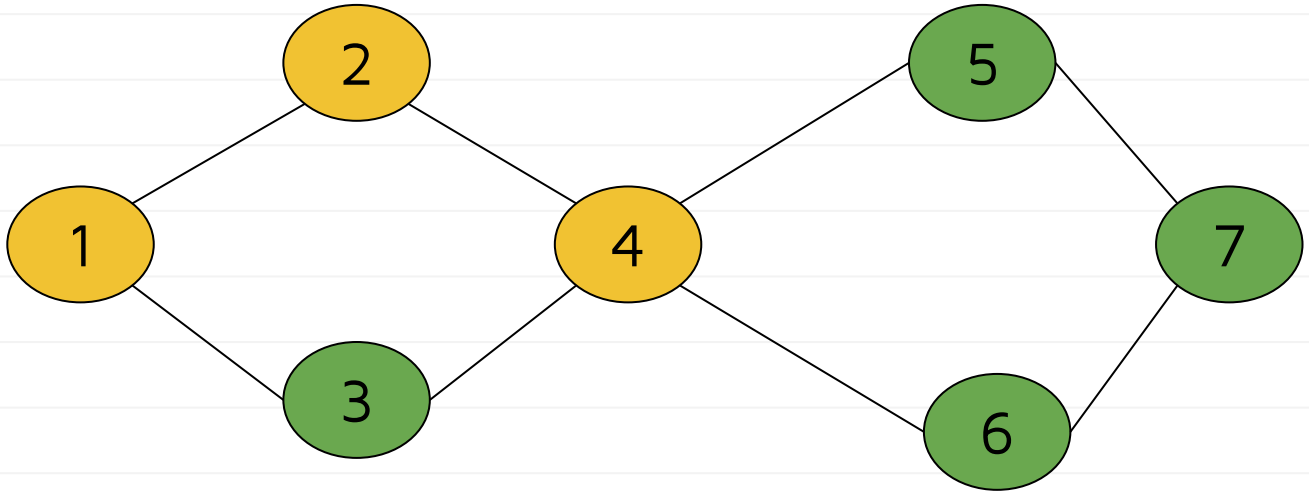
What calls are nested within the dfs(2)?



dfs(4)
dfs(2)
dfs(1)
dfs(3)
dfs(5)
dfs(7)
dfs(6)

However..

This intuition is correct if there is a path of **undiscovered** nodes from u to v when dfs(u) is called.




```
dfs(4)
    dfs(2)
        dfs(1)
            dfs(3)
    dfs(5)
        dfs(7)
            dfs(6)
```

Key Property of DFS (Informal)

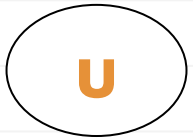
- If at the time `dfs(u)` is called…
  1. v is **undiscovered**; and
  2. there is a path of **undiscovered** nodes from u to v,
- …then `dfs(v)` will **start and finish** during the call to `dfs(u)`.

Exercise #2

Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a path from v to u.

Exercise

Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a path from v to u.

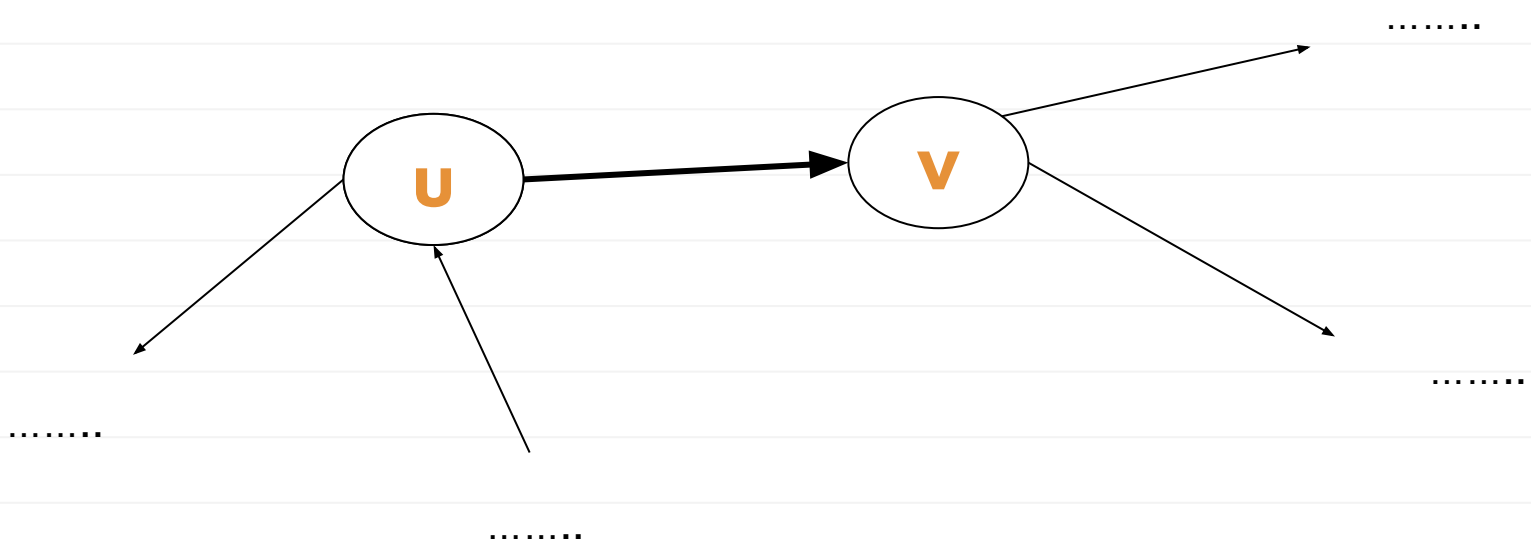


Exercise

Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a path from v to u.

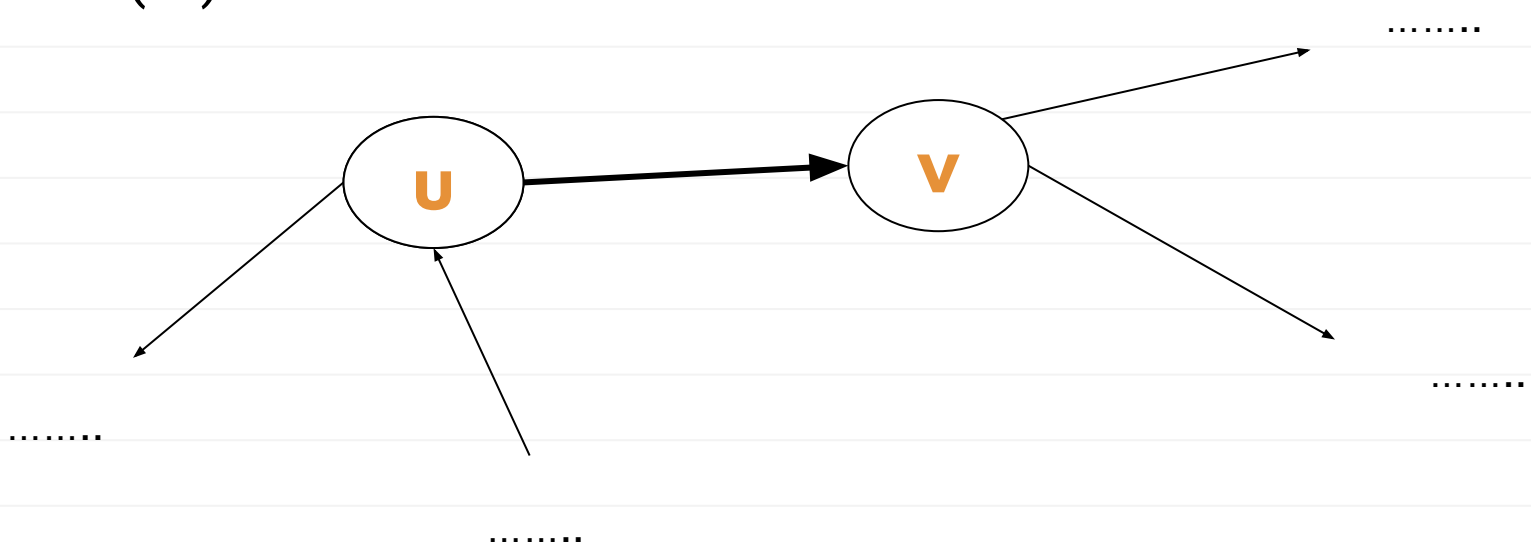
Exercise

Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a path from v to u.



Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a *path* from v to u.

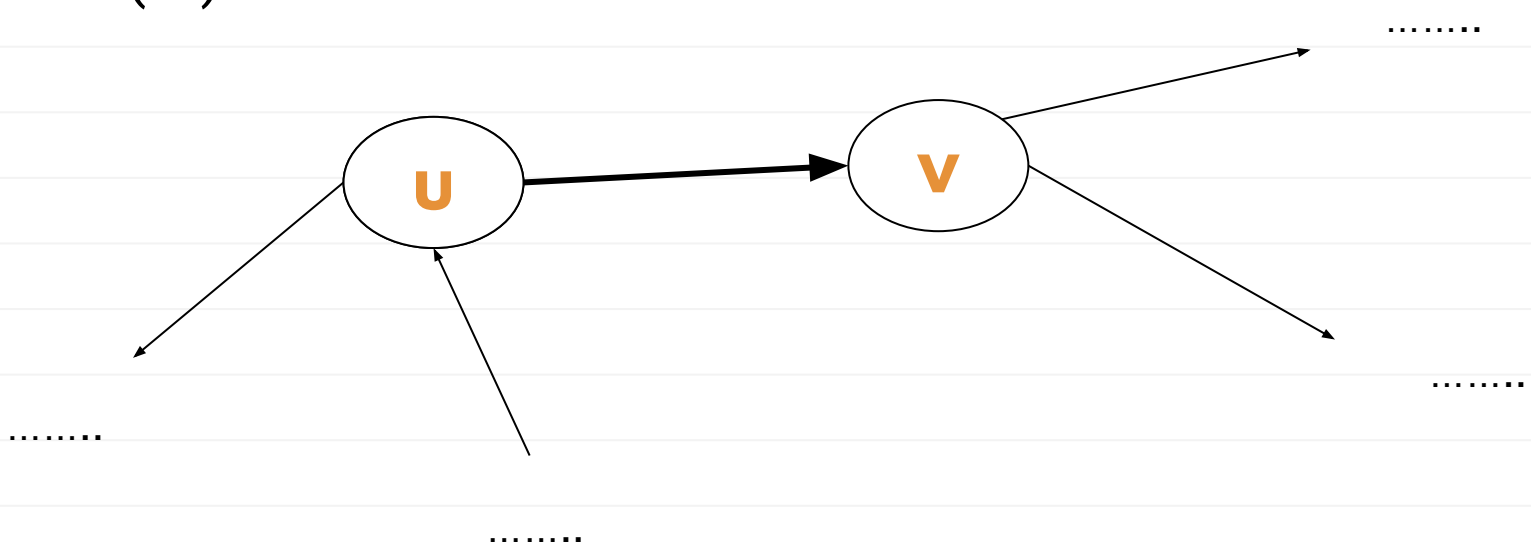
```
dfs(u)
```



Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a *path* from v to u.

V is **pending**. What does it mean?

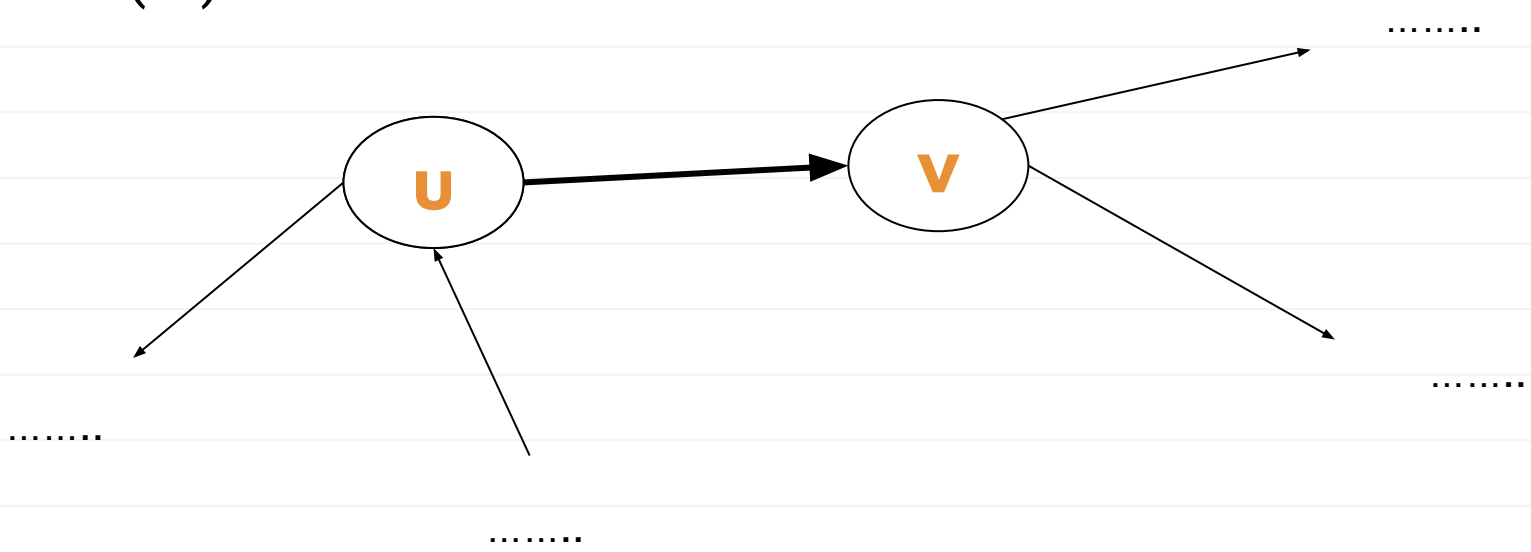
```
dfs(u)
```



Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a *path* from v to u.

`dfs(u)`

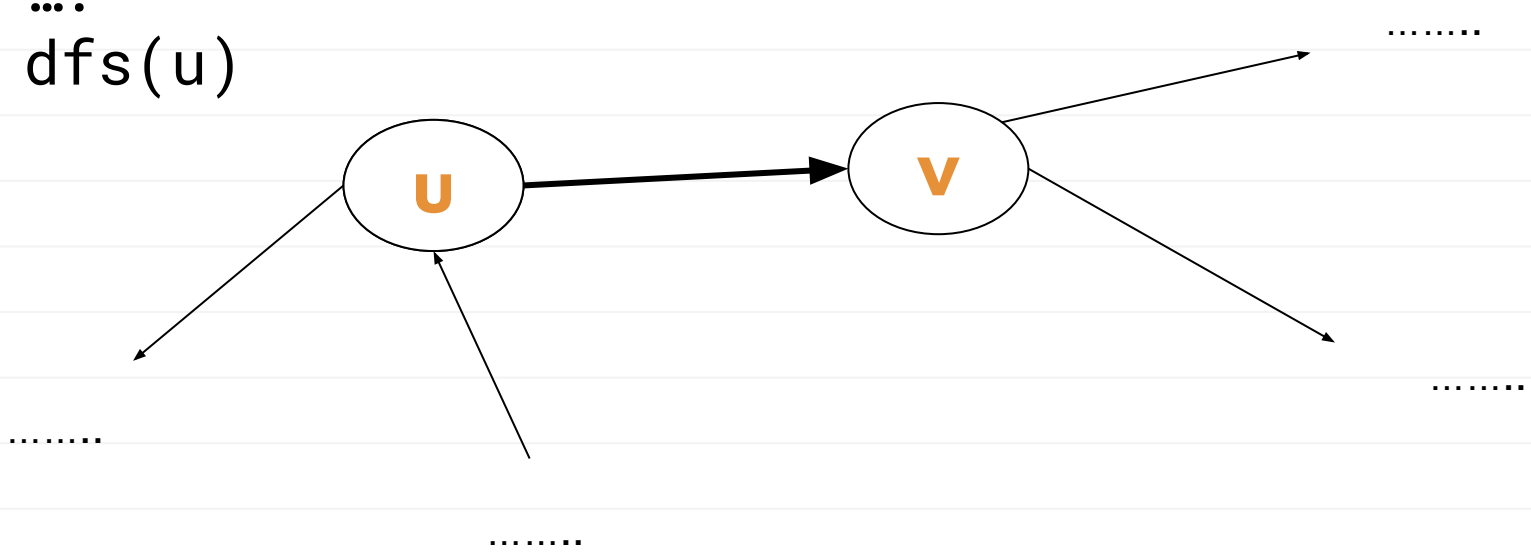
There was a call to `dfs(v)` already, **before** `dfs(u)`



Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a *path* from v to u.

```
dfs(v)
   ….
   dfs(u)
```

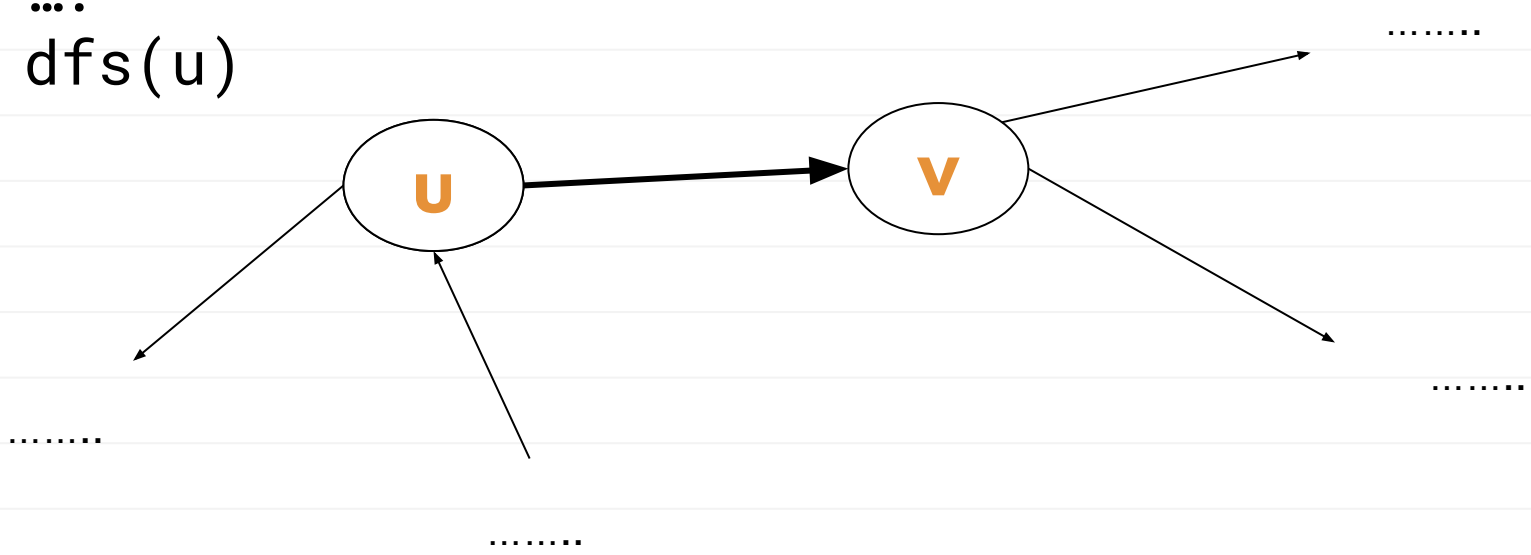
Are they nested?



Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a *path* from v to u.

```
dfs(v)
    ….
    dfs(u)
```

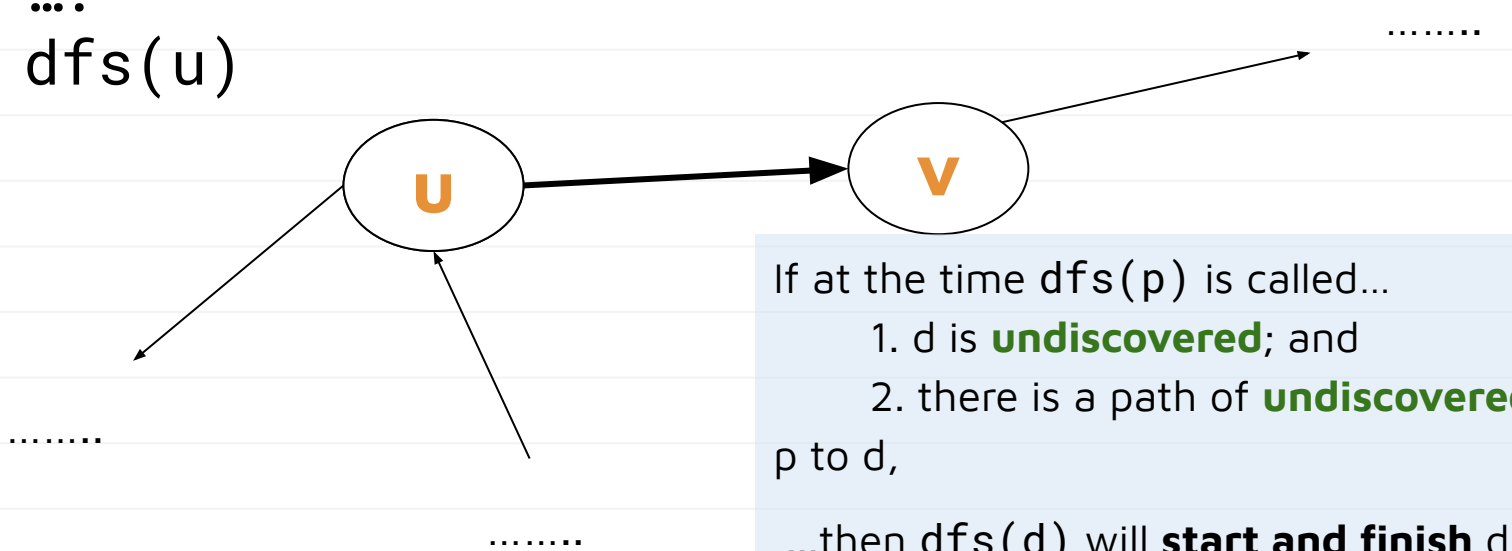
Yes! Because v is not finished yet, it is not red.



Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a *path* from v to u.

```
dfs(v)
    ….
    dfs(u)
```

There is a path from v to u, from our key property.



If at the time `dfs(p)` is called…

1. d is **undiscovered**; and
2. there is a path of **undiscovered** nodes from p to d,

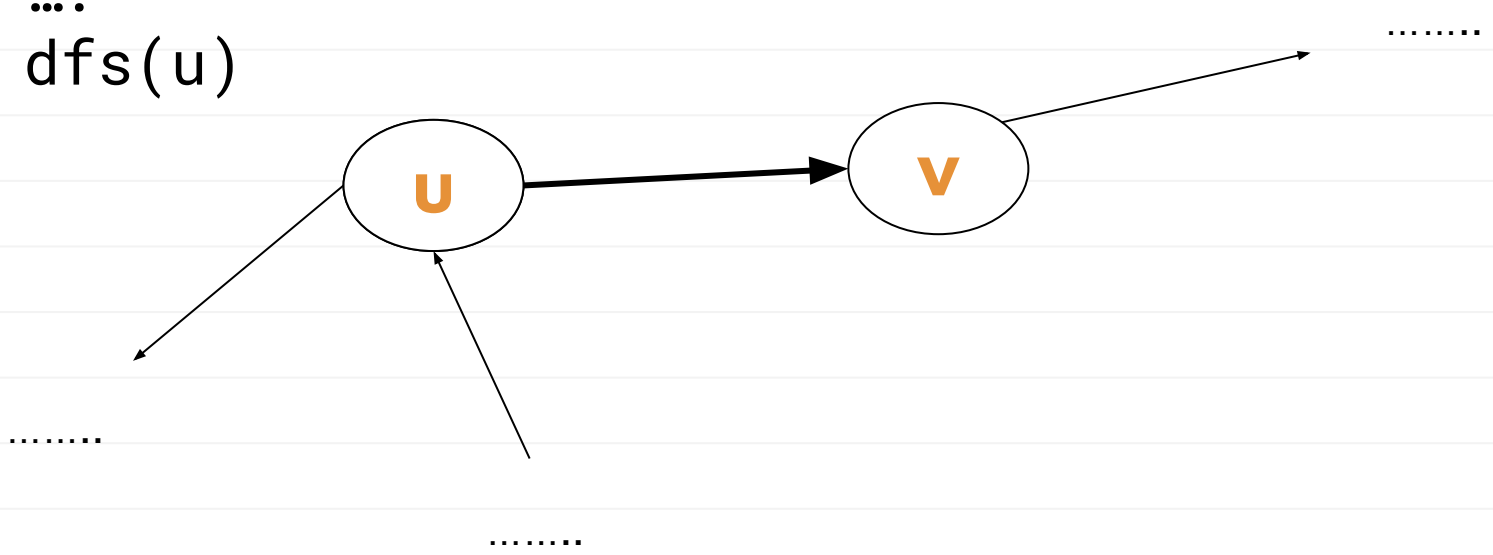
…then `dfs(d)` will **start and finish** during the call to `dfs(p)`.

back

Suppose while calling dfs on node u, we see that neighbor v is **pending**. **True** or **False**: there is a *path* from v to u.

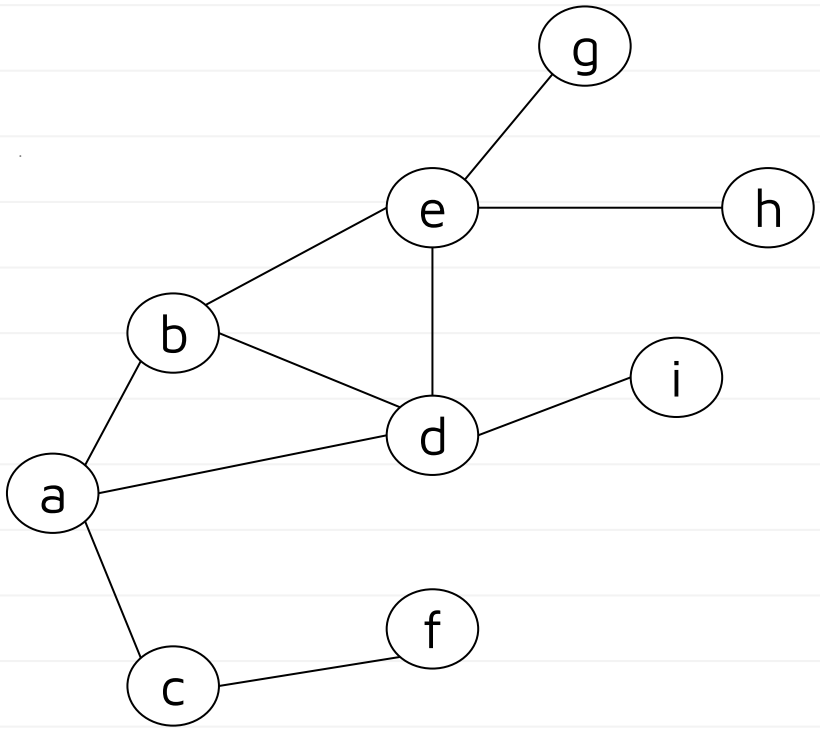
```
dfs(v)
   ….
   dfs(u)
```

There is a path from u to v, (v is a neighbor).



Start and Finish Times

- Keep a running clock (an integer).
- For each node, record
  - **Start time:** time when marked **pending**
  - **Finish time**: time when marked **visited**
- Increment clock whenever node is marked **pending**/**visited**.

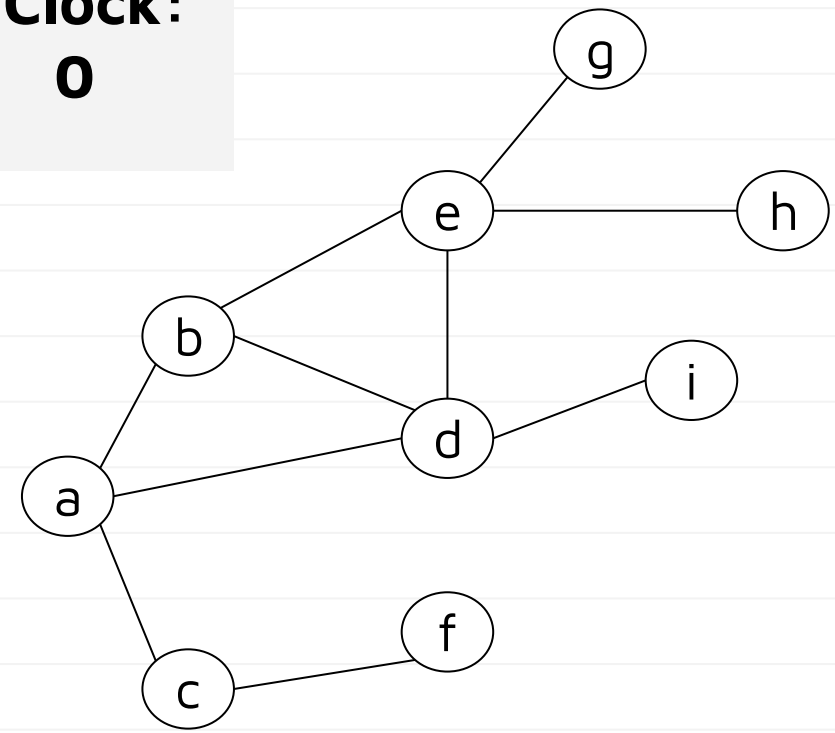


For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
0

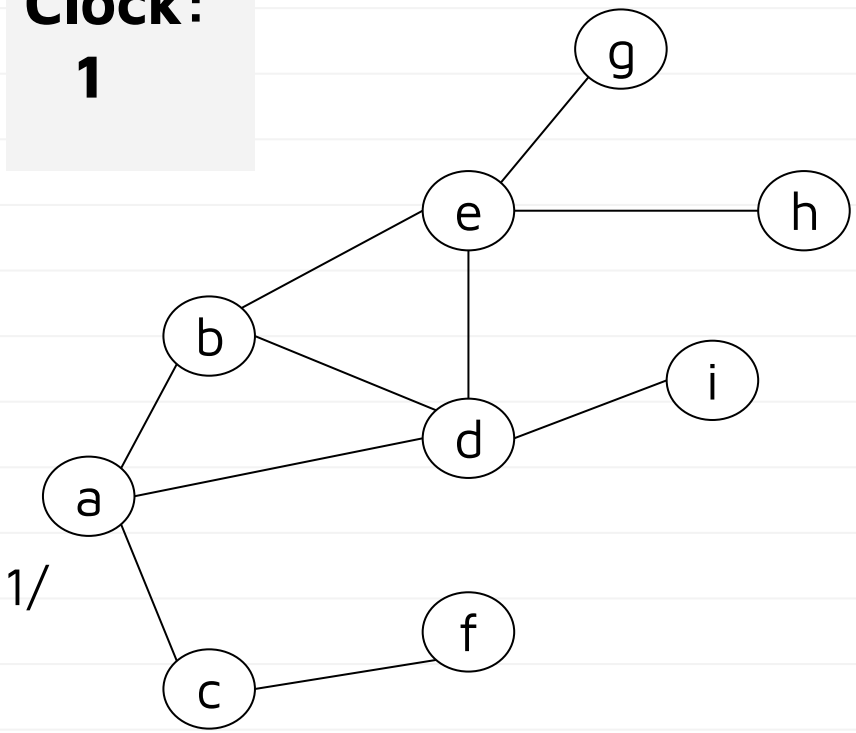


For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
1



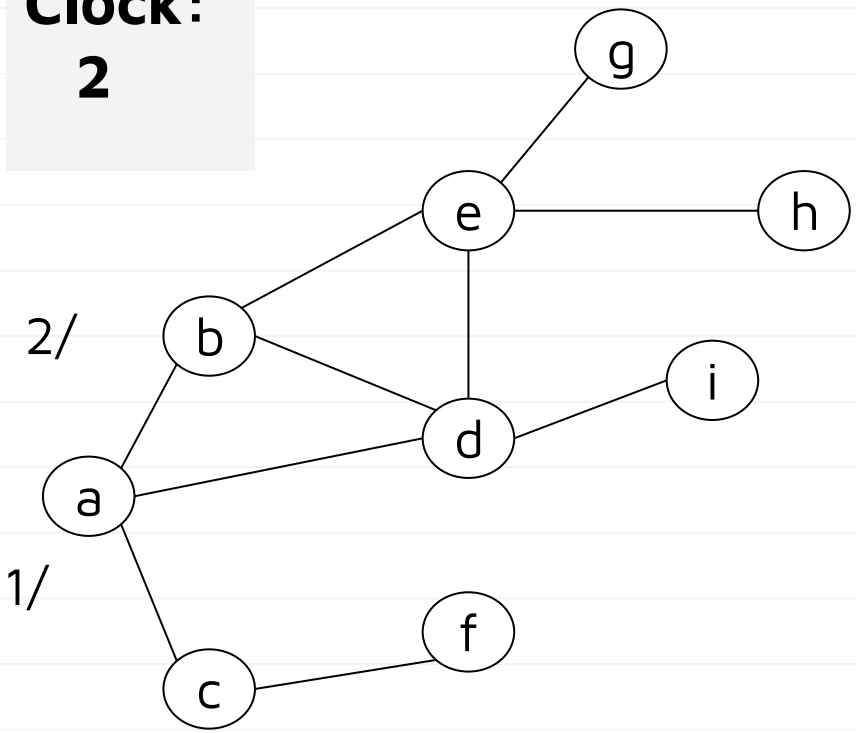
`dfs(a)`

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
2



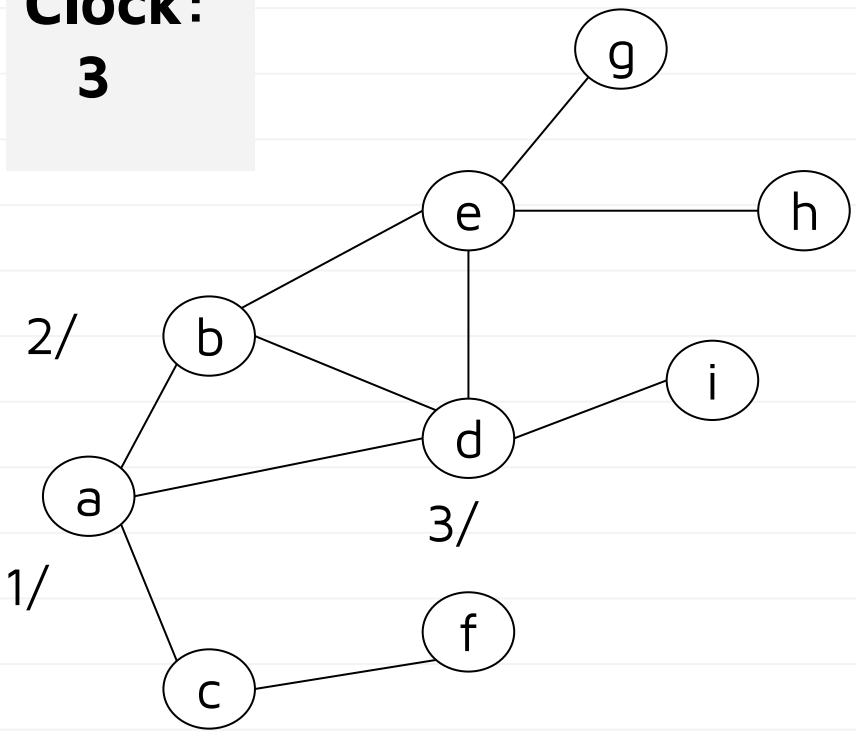

```
dfs(a)
  dfs(b)
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
3



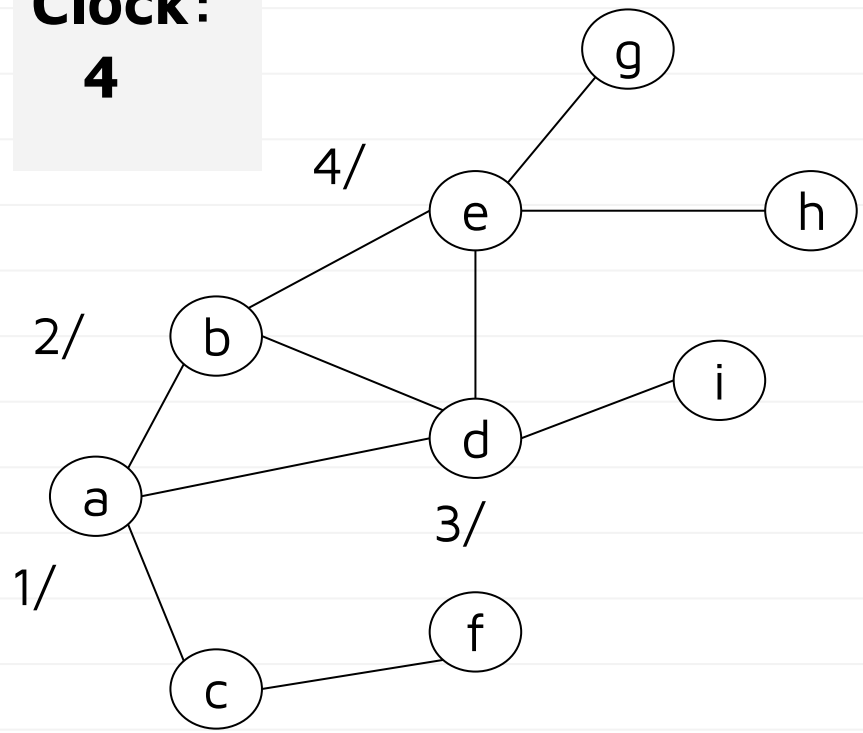

```
dfs(a)
  dfs(b)
    dfs(d)
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
4



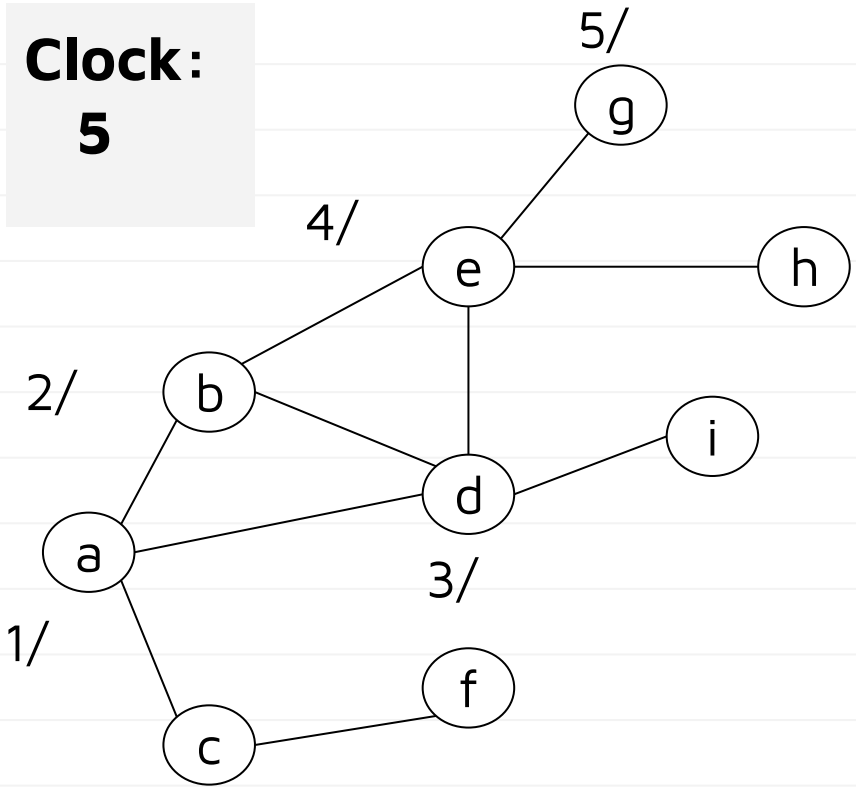

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
5



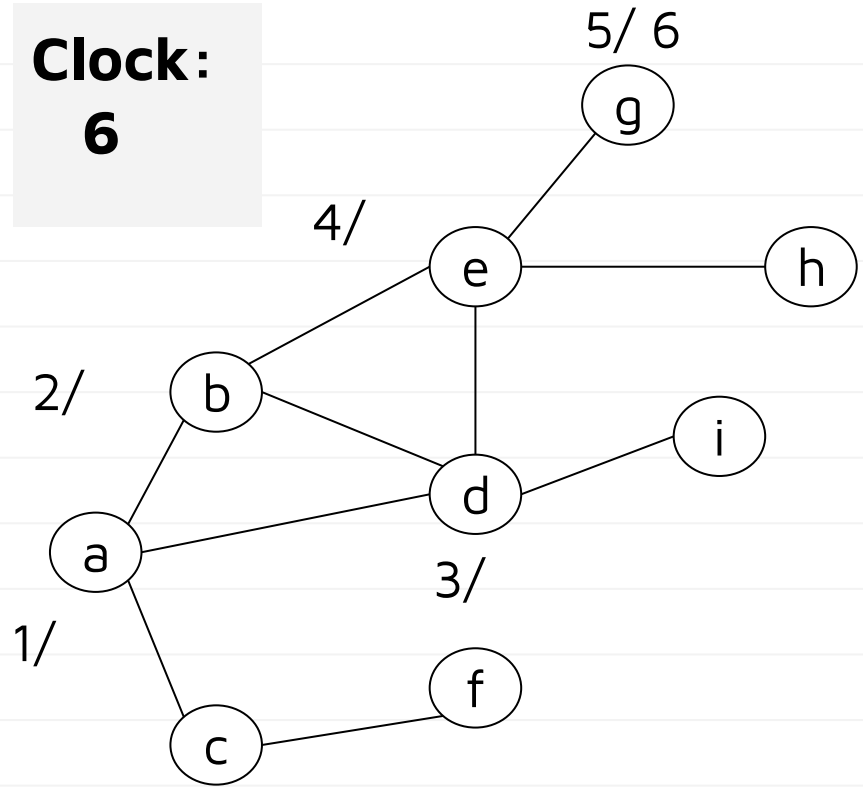

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
6



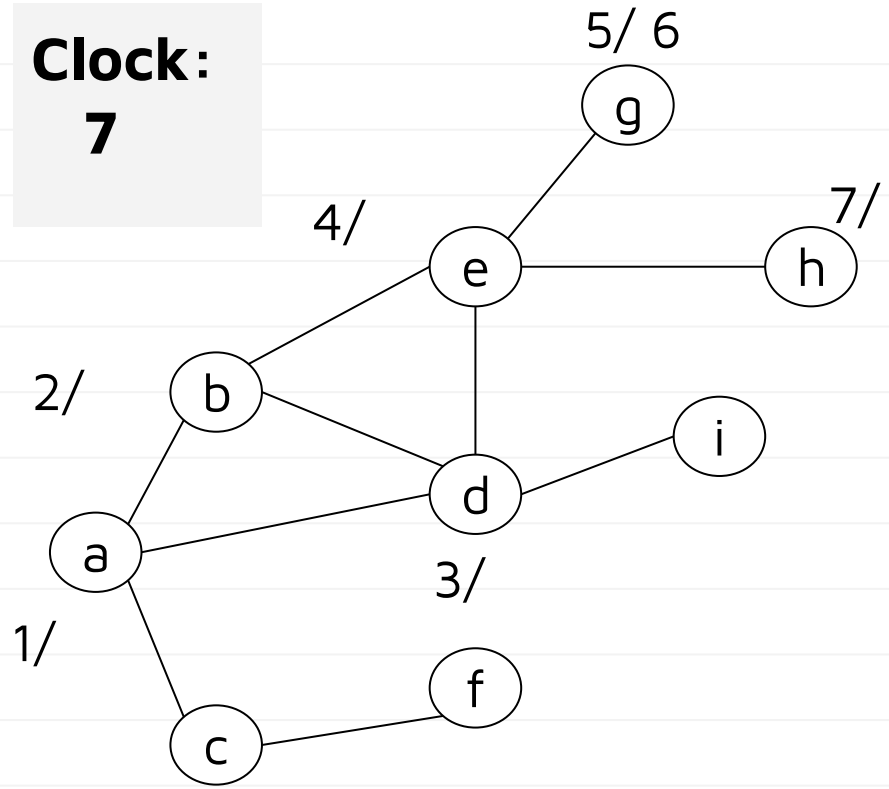

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 7



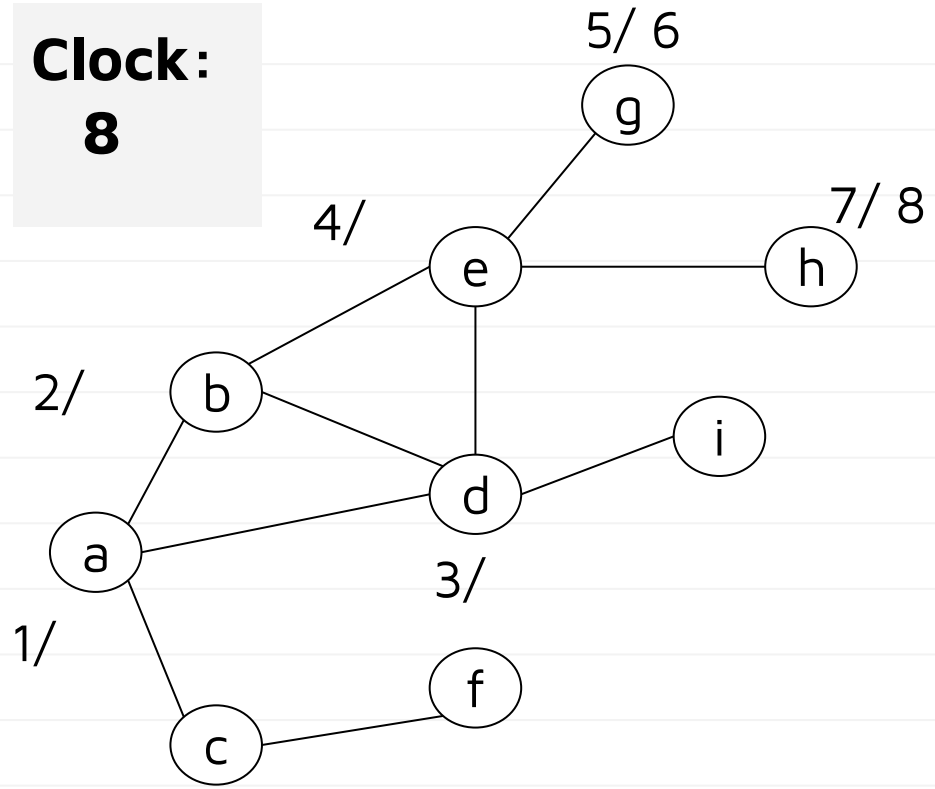

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
8



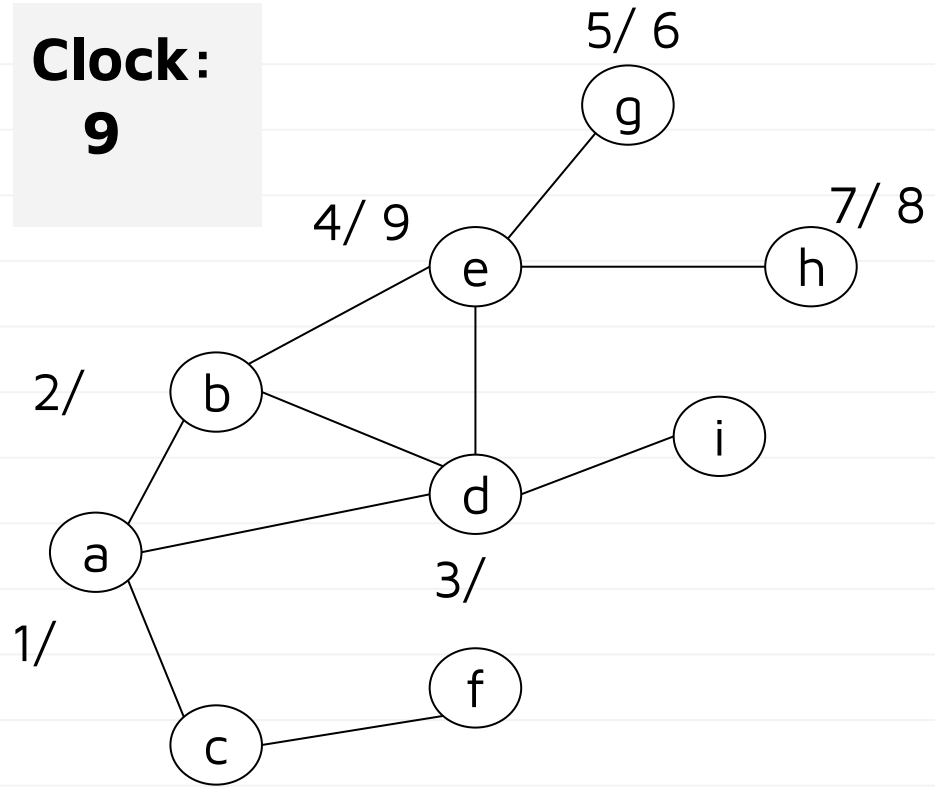

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        ~~dfs(g)~~
        ~~dfs(h)~~
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 9



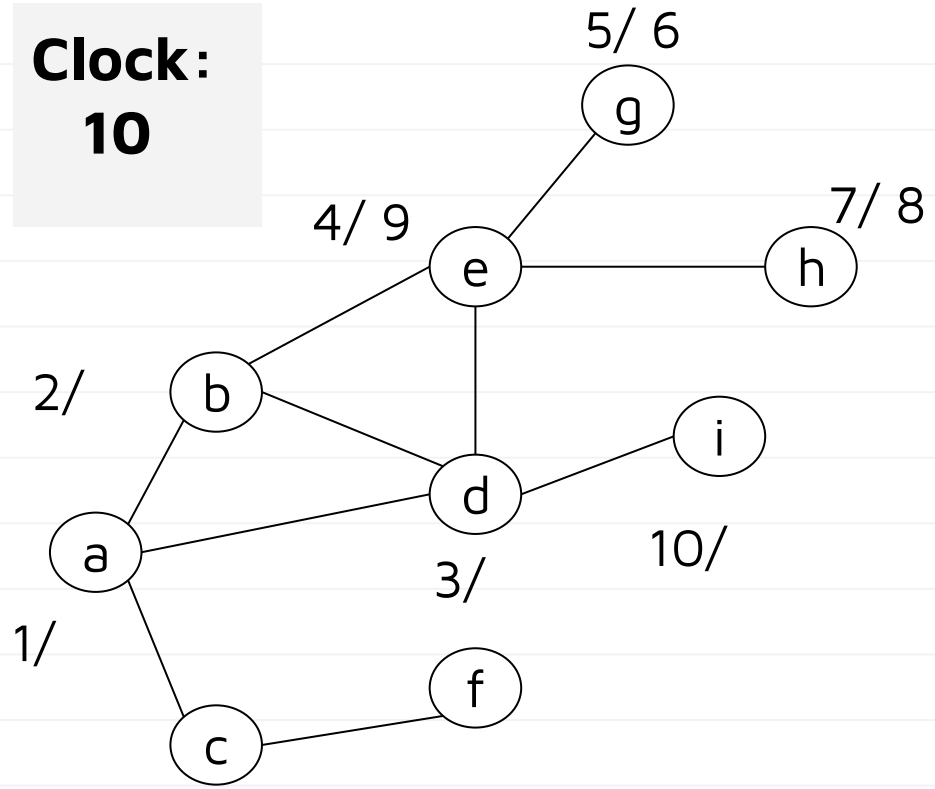

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
```

For each node, record

- **Start time**: time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 10



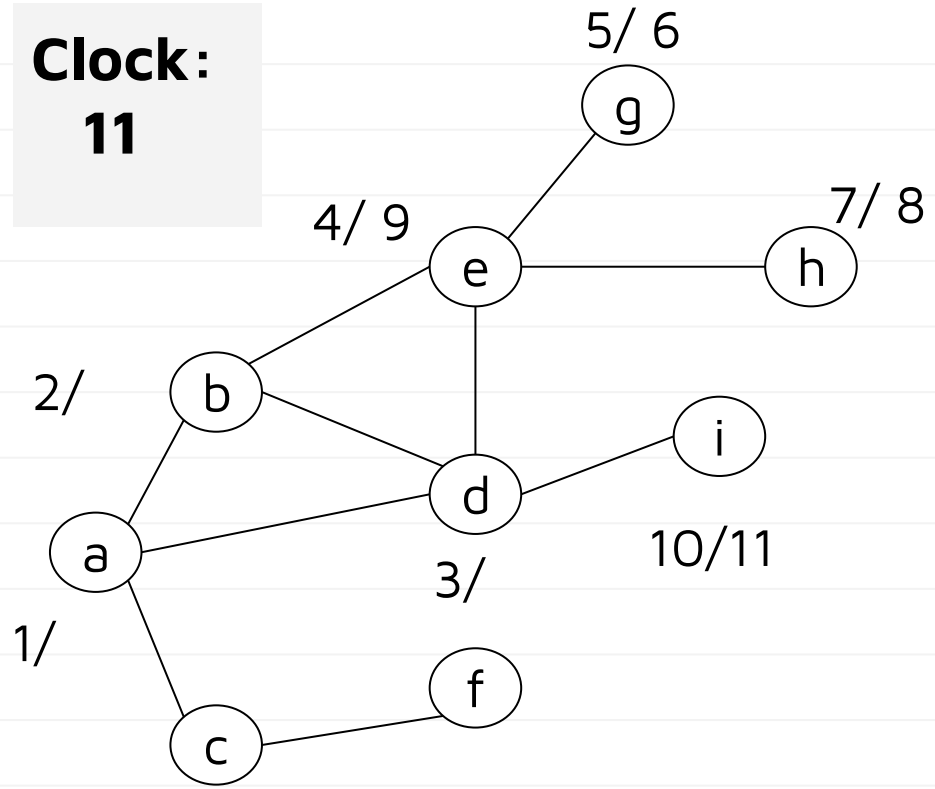

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
```

For each node, record

- **Start time**: time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 11



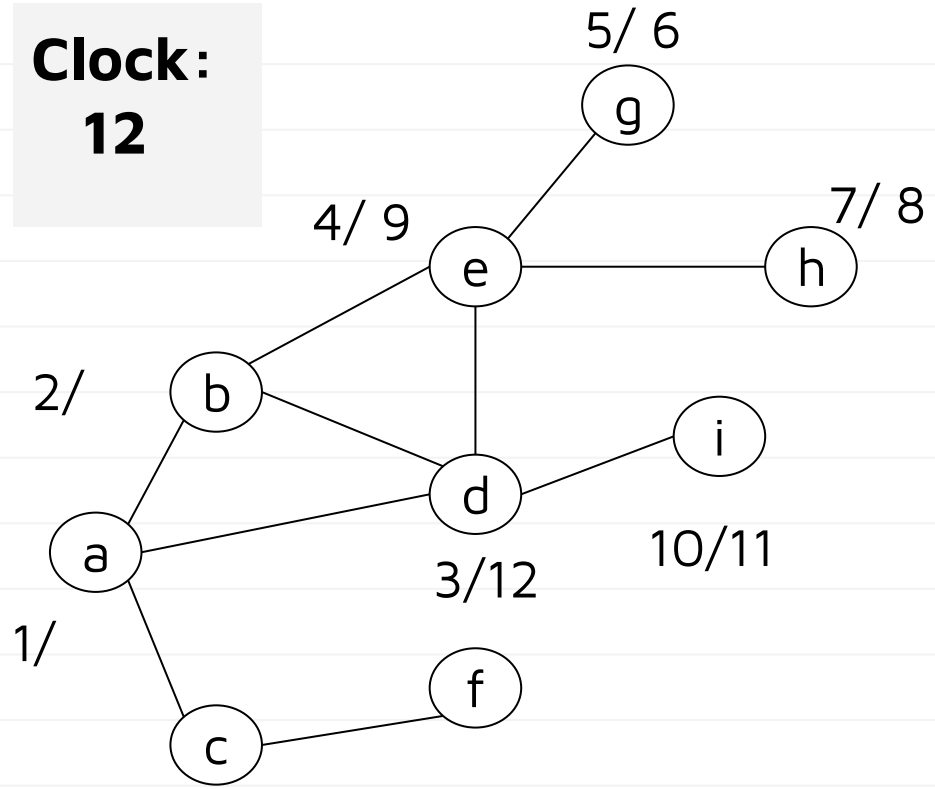

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
```

For each node, record

- **Start time**: time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
12



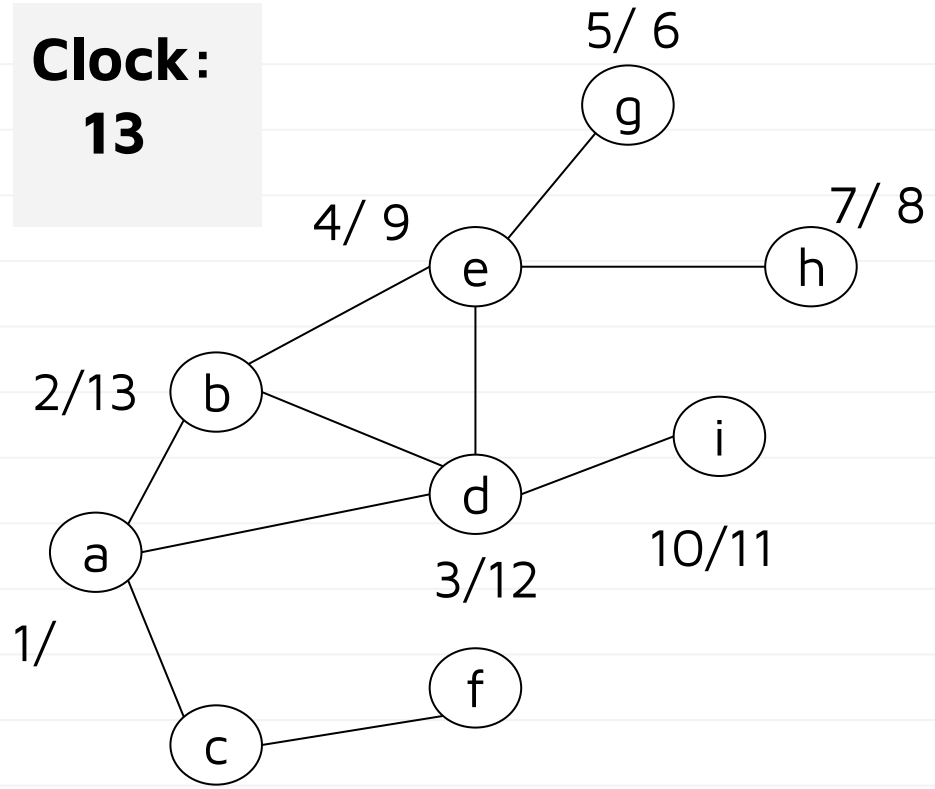

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 13



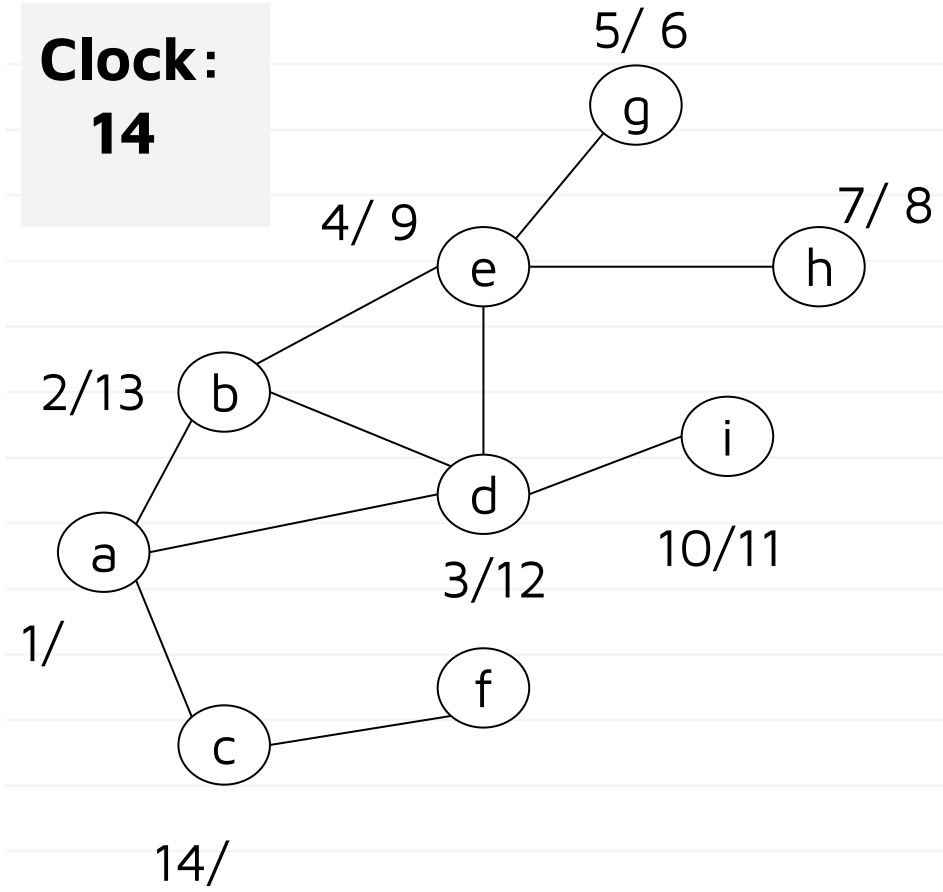

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
```

For each node, record

- **Start time**: time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 14



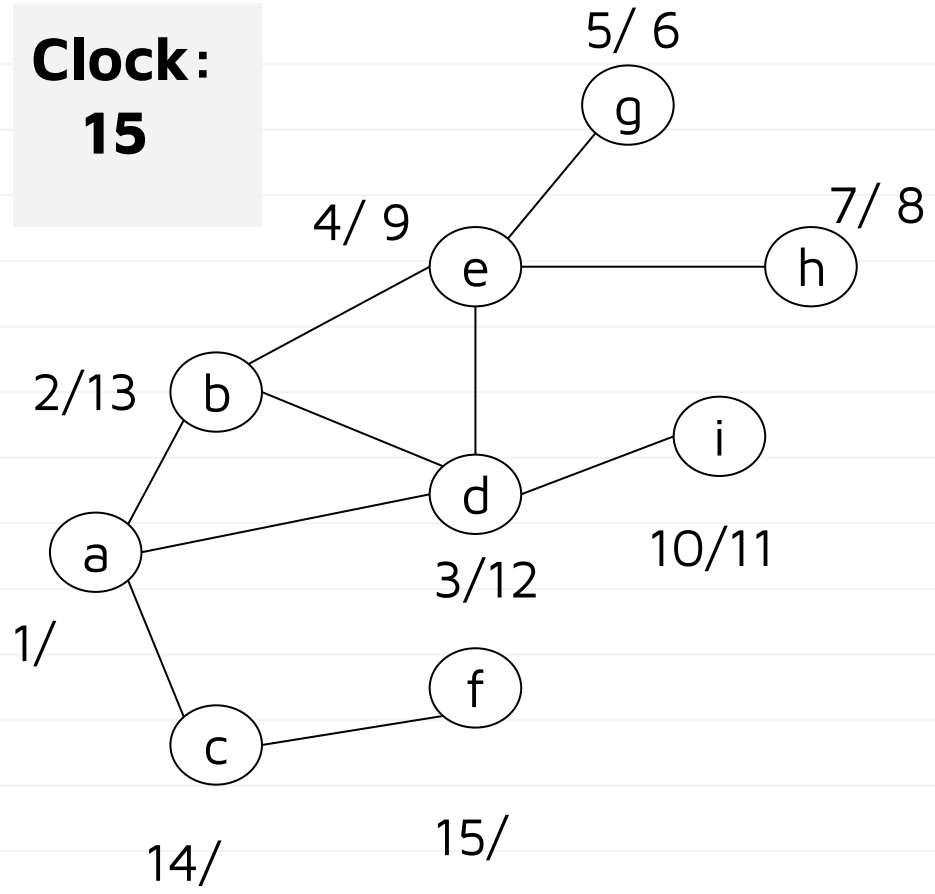

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
```

For each node, record

- **Start time**: time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 15



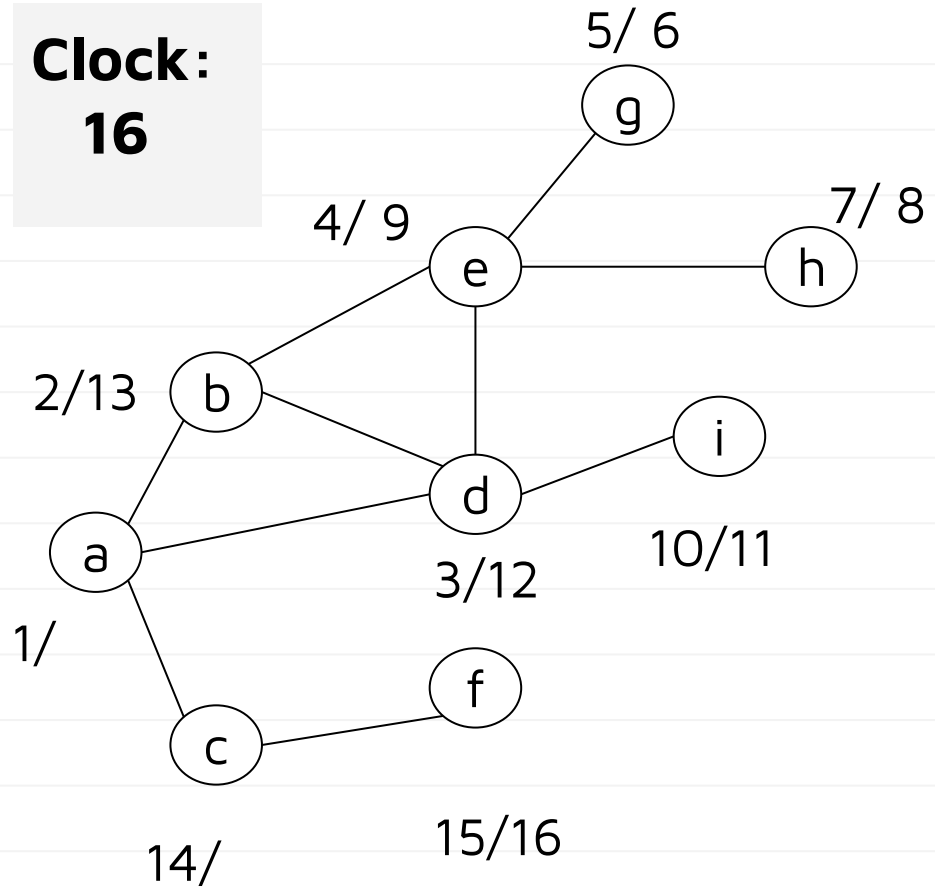

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
 dfs(f)
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 16



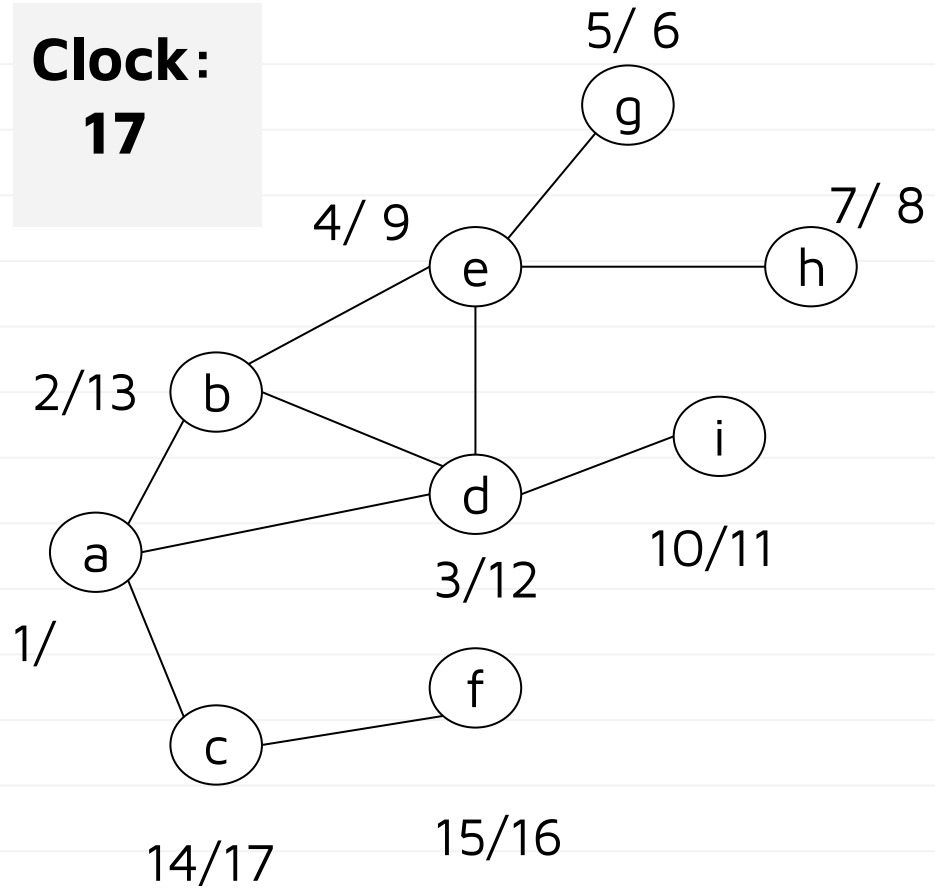

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
dfs(c)
  dfs(f)
```

For each node, record

- **Start time**: time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 17



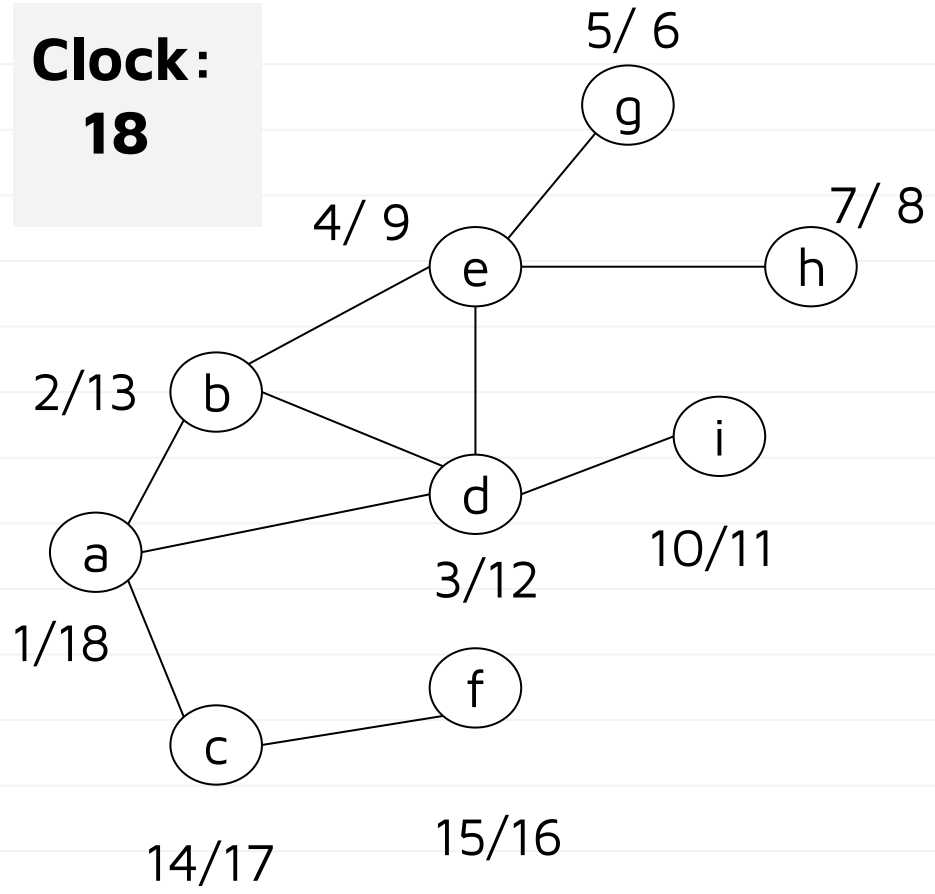

```
dfs(a)
  dfs(b)
    dfs(d)
      dfs(e)
        dfs(g)
        dfs(h)
      dfs(i)
  dfs(c)
    dfs(f)
```

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock: 18



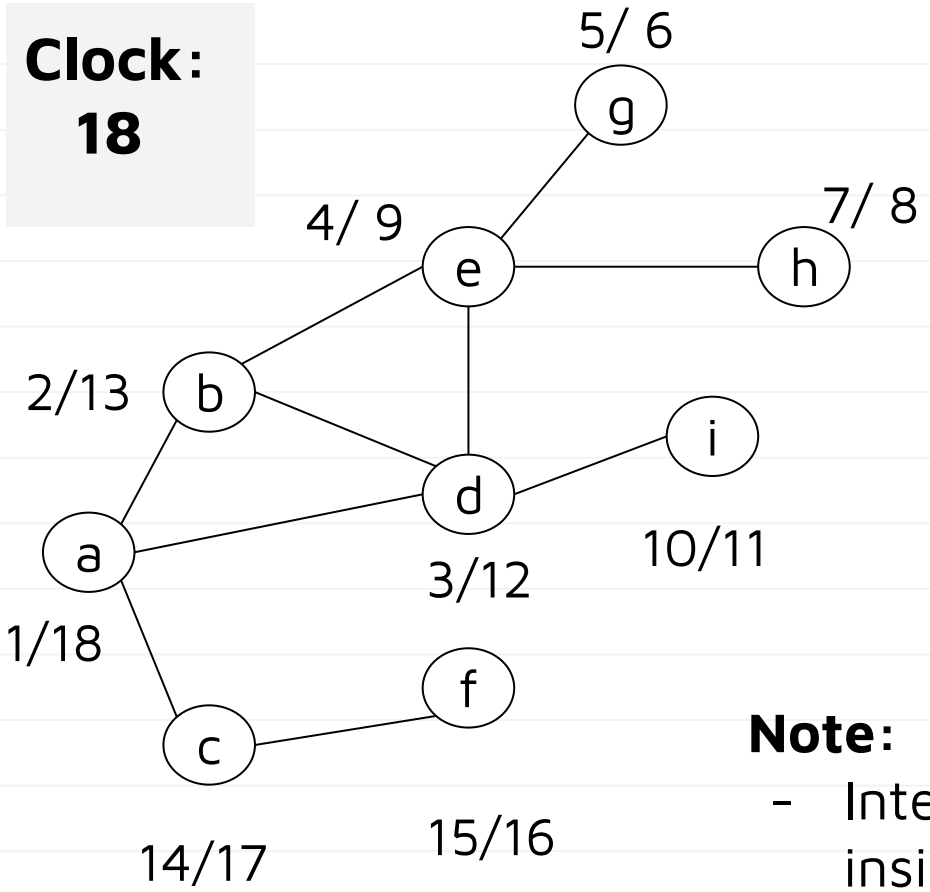
~~dfs(a)~~
~~dfs(b)~~
~~dfs(d)~~
~~dfs(e)~~
~~dfs(g)~~
~~dfs(h)~~
~~dfs(i)~~
~~dfs(c)~~
~~dfs(f)~~

For each node, record

- **Start time:** time when marked **pending**
- **Finish time**: time when marked **visited**

Increment clock whenever node is marked **pending**/**visited**.

Clock:
18



~~dfs(a)~~
~~dfs(b)~~
~~dfs(d)~~
~~dfs(e)~~
~~dfs(g)~~
~~dfs(h)~~
~~dfs(i)~~
~~dfs(c)~~
~~dfs(f)~~

Note:

- Intervals are either fully contained inside each other (i.e. d and e)
- Or completely separate. (i.e. i and f)
- No overlaps

```python
from dataclasses import dataclass

@dataclass
class Times:
    clock: int
    start: dict
    finish: dict
```

```python
from dataclasses import dataclass

@dataclass
class Times:
    clock: int
    start: dict
    finish: dict


def full_dfs_times(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}
    times = Times(clock=0, start={}, finish={})
    for u in graph.nodes:
        if status[u] == 'undiscovered':
            dfs_times(graph, u, status, times)
    return times, predecessor
```

```python
from dataclasses import dataclass

@dataclass
class Times:
    clock: int
    start: dict
    finish: dict

def full_dfs_times(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}
    times = Times(clock=0, start={}, finish={})
    for u in graph.nodes:
        if status[u] == 'undiscovered':
            dfs_times(graph, u, status, times)
    return times, predecessor

def dfs_times(graph, u, status, predecessor, times):
    times.clock += 1
    times.start[u] = times.clock
    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            predecessor[v] = u
            dfs_times(graph, v, status, times)
    status[u] = 'visited'
    times.clock += 1
    times.finish[u] = times.clock
```

Key Property of DFS

- Suppose dfs(u) is called before dfs(v).
- If when dfs(u) is called there is a path of **undiscovered** nodes from u to v, then:

start[u] < start[v] < finish[v] < finish[u].

- Otherwise:

start[u] < finish[u] < start[v] < finish[v]

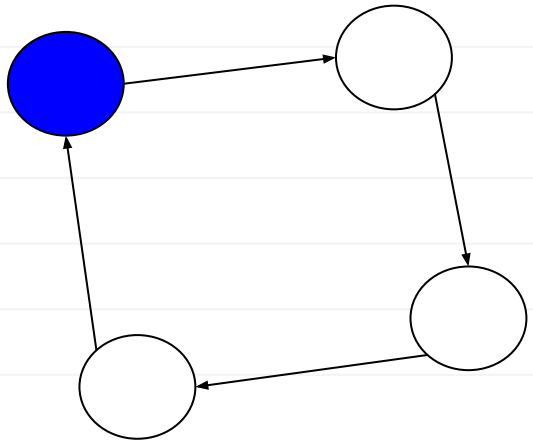
Key Property

- Take any two nodes u and v ($u \neq v$).
- Assume for simplicity that start[u] < start[v].
- Exactly one of these is true:
  - `start[u] < start[v] < finish[v] < finish[u]`
  - `start[u] < finish[u] < start[v] < finish[v]`

Cycles

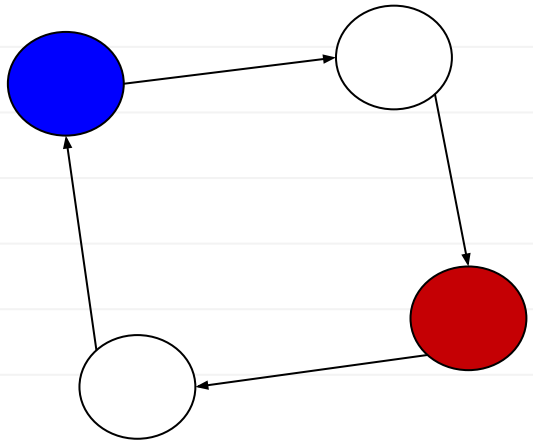
Cycles

- A **cycle** in a directed graph is a path that starts and ends at the same node.



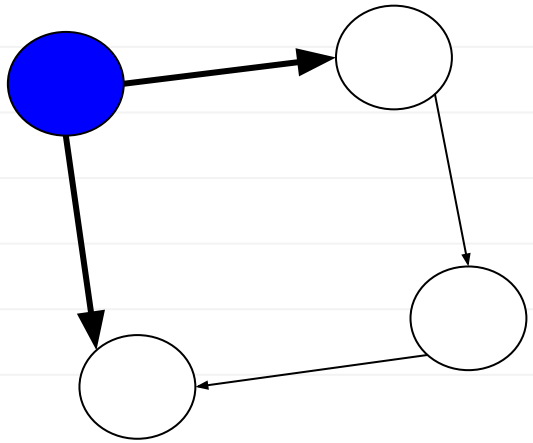
Cycle

- Alternatively: there is a **cycle** if u is reachable from v and v is reachable from u, for some $u \neq v$.



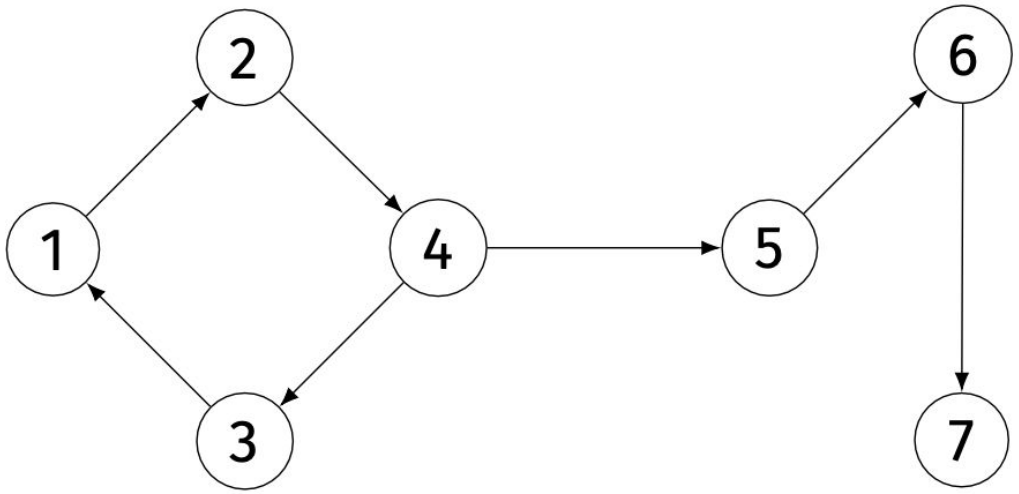
DAG

- A **directed acyclic graph** (DAG) is a directed graph with **no cycles.**



Cyclic Graphs

- A graph is **cyclic** even if it has only one cycle.
  - It doesn't have to be the whole graph.



Detecting Cycles

Link

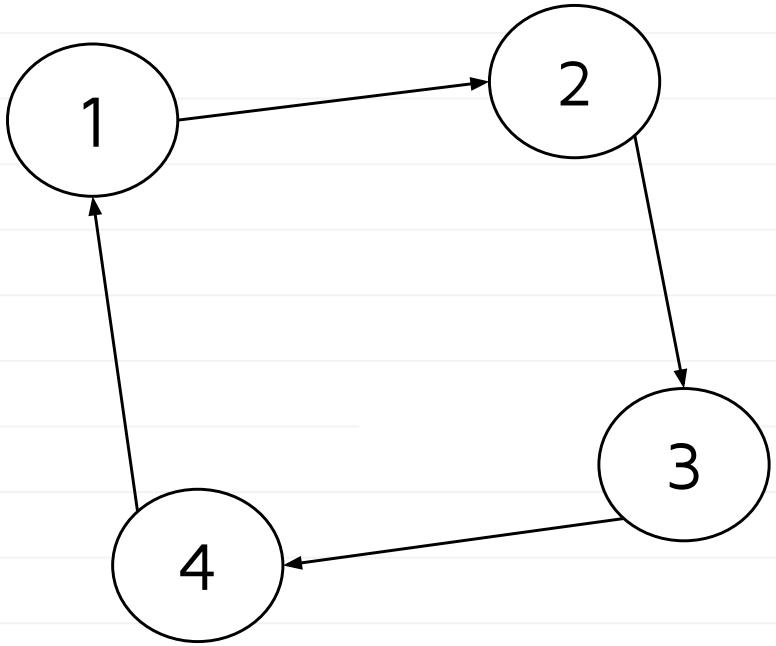
- We check for cycles by looking for **back edges** in a full DFS.
- (u, v) is a **back edg**e if while visiting node u, we see that v is **pending**.

Detecting Cycles

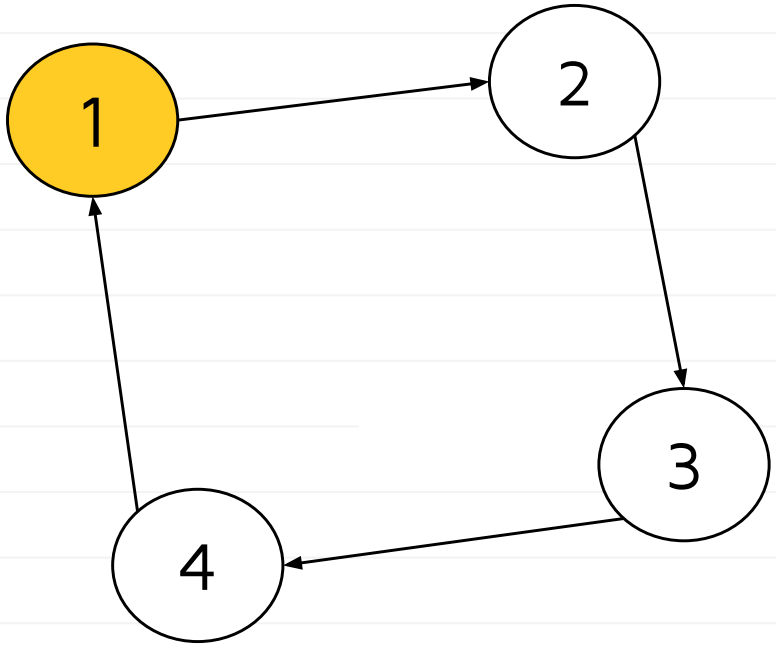
- We check for cycles by looking for **back edges** in a full DFS.
- (u, v) is a **back edge** if while visiting node u, we see that v is **pending**.

```
...
for v in graph.neighbors(u): # explore edge (u, v)
    if status[v] == 'undiscovered':
        dfs(graph, v, status)
    elif status[v] == 'pending':
        # back edge (u, v) found!
...
```

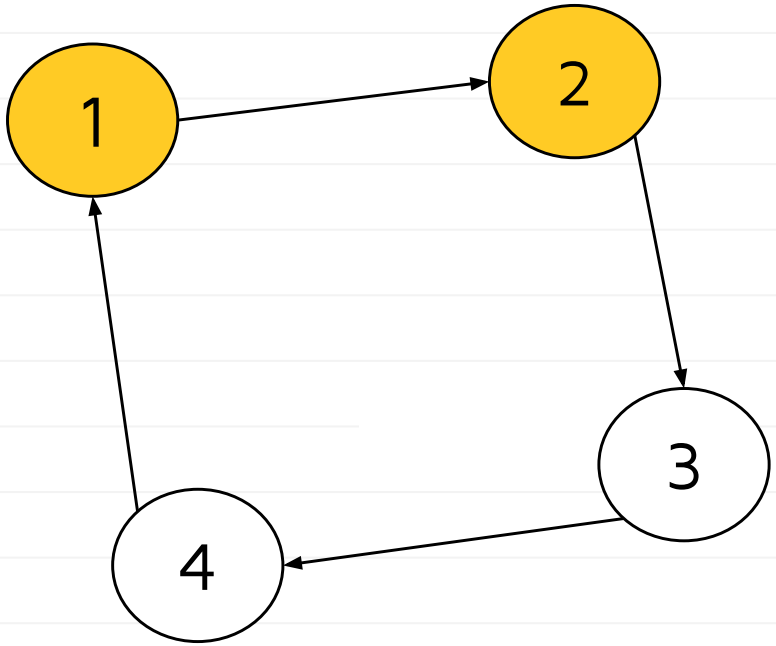
Example



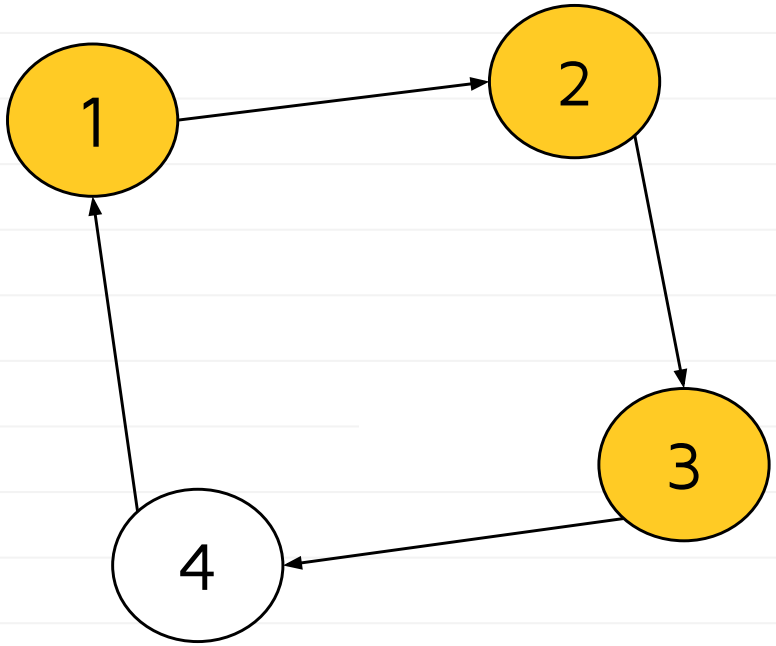
Example



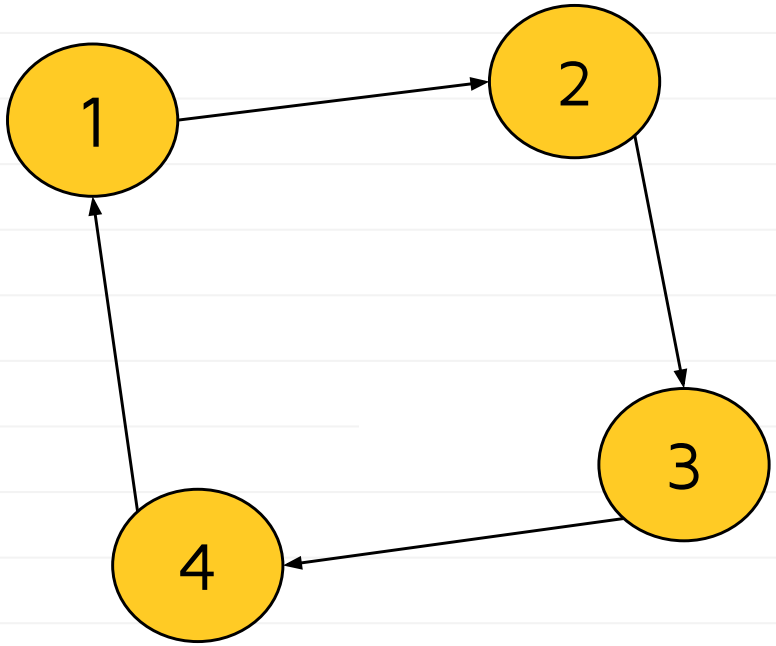
Example



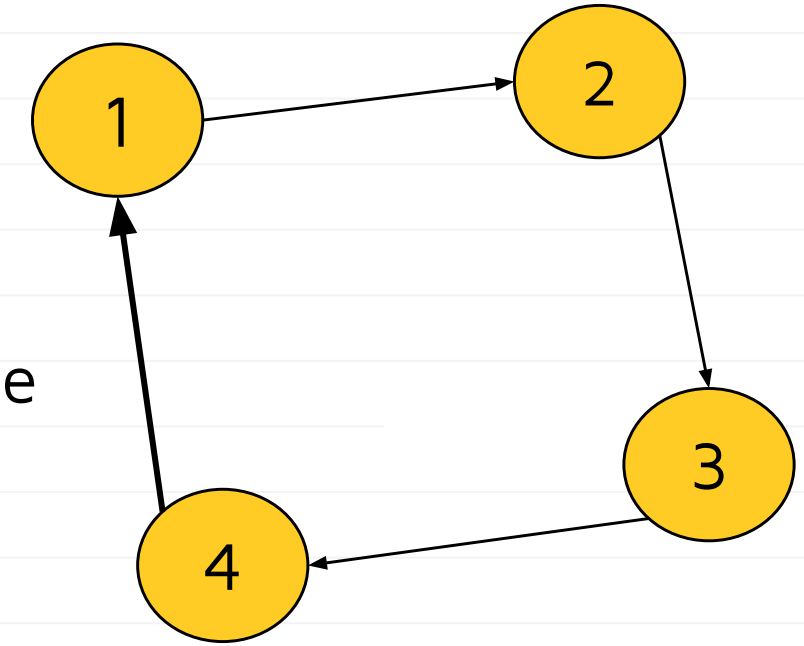
Example



Example



Example



Back edge

Theorem

A directed graph has a cycle **if (and only if)** a full DFS finds a back edge.

Why?

- **If a back edge (u, v) is found, then a cycle exists.**
  - Suppose v is pending when we visit u.
  - This means that there is a path from v to u.
  - There is also a path from u to v.
  - So there is a cycle.

Why?

- **If a cycle exists, then there is a back edge.**
  - Suppose there is a cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$.
  - Without loss of generality, assume v_1 is the first node in the cycle that is visited by the full DFS.
  - At the moment of dfs(v_1), there is a path of **undiscovered** nodes between v_1 and v_k.
  - Therefore dfs(v_k) will be called during dfs(v_1).
  - During dfs(v_k), we'll see the back edge.
    - v_1 will be **pending**.

Question

- Suppose v is reachable from u in a DAG.

True or False: after a full DFS, `finish[v] < finish[u]`.

Question

- Suppose v is reachable from u in a DAG.

True or False: after a full DFS, `finish[v] < finish[u]`.

Question

A: True

B: False

C: Not sure, need time to process the algorithm.

- Suppose v is reachable from u in a DAG.

True or **False**: after a full DFS, `finish[v] < finish[u]`.

Question

Case 1: Start DFS(u) -> True

Case 2: Start somewhere else, say, DFS(v), then restart the DFS. -> True

- Suppose v is reachable from u in a DAG.

True or **False**: after a full DFS, `finish[v] < finish[u]`.

Claim

- If v is *reachable* from u in a DAG, then:

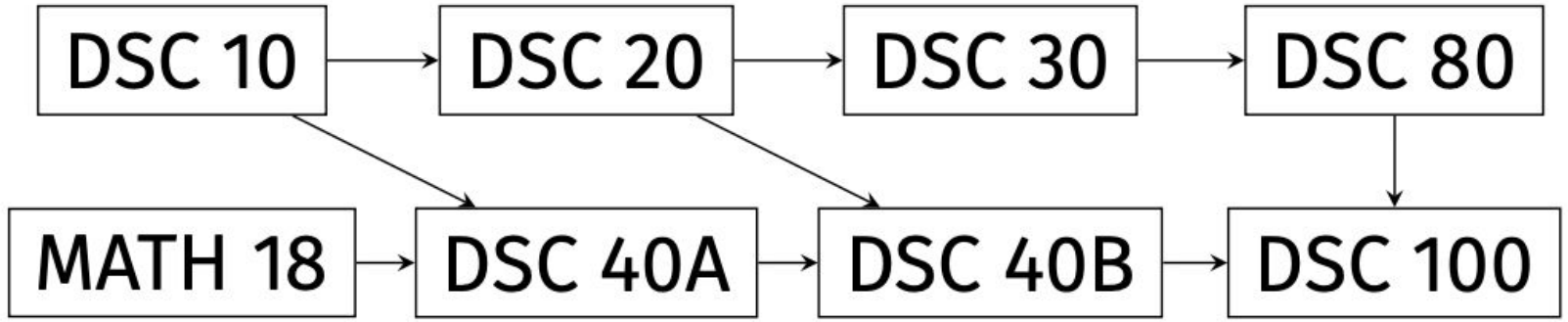
```
finish[v] < finish[u]
```

Topological Sort

Applications of DFS

- Is node v reachable from node u? **DFS, BFS**
- Is the graph connected? **DFS, BFS**
- How many connected components? **DFS, BFS**
- Find the shortest path between u and v. **DFS, BFS**
- Does the graph have a cycle? **DFS, BFS**

Prerequisite Graphs



Goal: find order in which classes should be taken in order to satisfy the prerequisites of DSC 100.

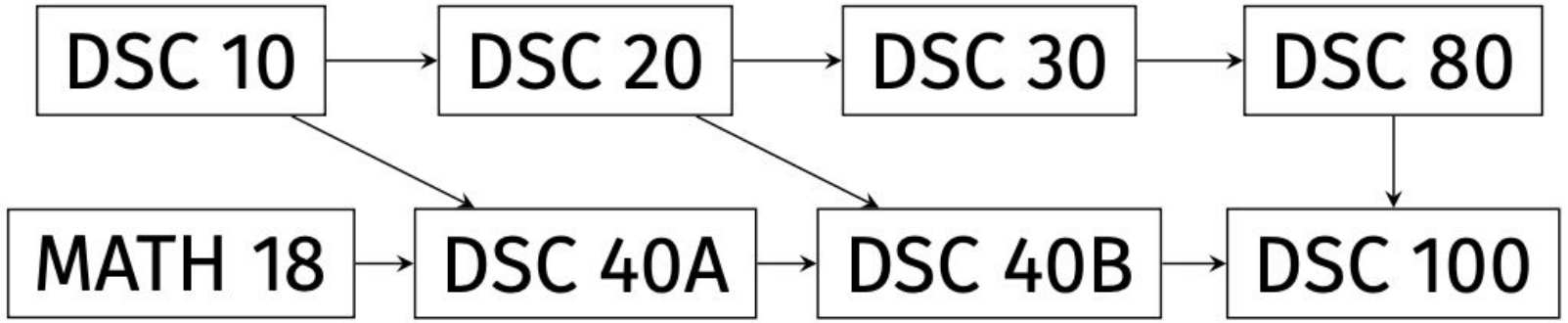
Note

- Prerequisite graphs are (Or they should be, at least!) DAGs.

Topological Sorts

- **Given**: a DAG, $G = (V, E)$.
- **Compute**: an *ordering* of V such that if $(u, v) \in E$, then u comes **before** v in the ordering.
- This is called a **topological sort** of G.

Example



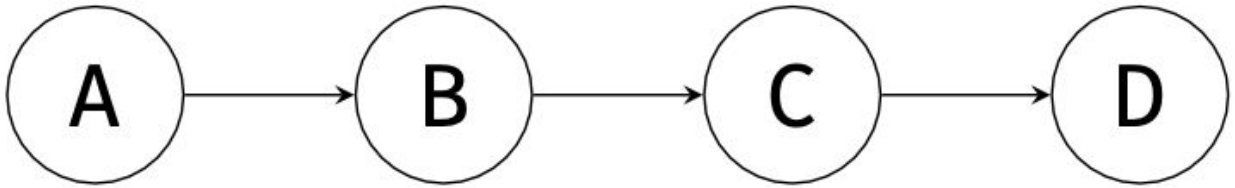
MATH 18, DSC 10, DSC 40A, DSC 20, DSC 40B, DSC 30, DSC 80, DSC 100

Note

- There can be ***many*** valid topological sorts!

Computing a Topological Sort

- How do we compute a topological sort, algorithmically?
- **Observation**: if v is reachable from u, v **must** come **after** u in the topological sort.



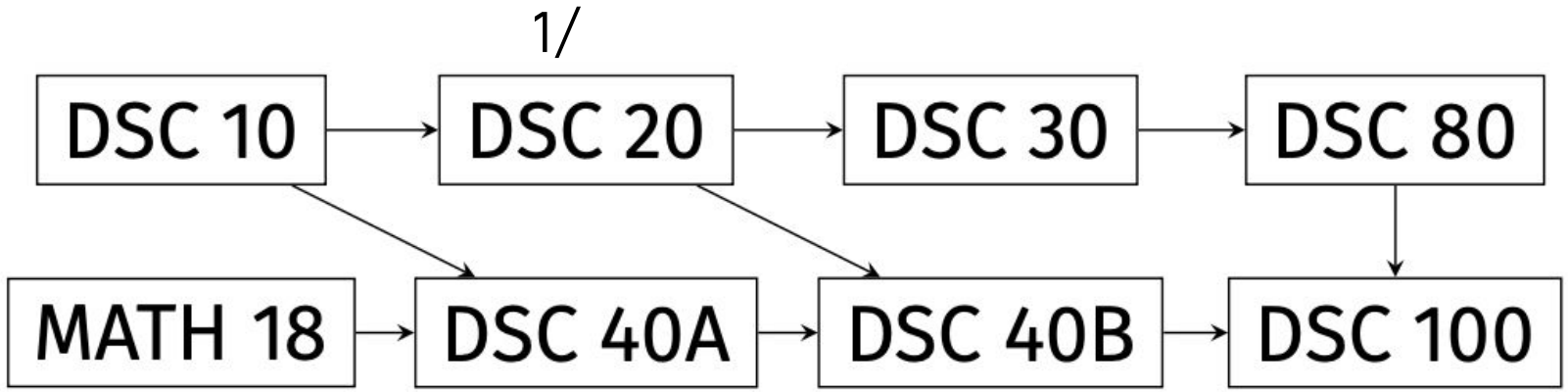
Recall

- Take any two nodes u and v ($u \neq v$).
- Assume the graph is a DAG, run DFS.
- If v is reachable from u, then `finish[v] < finish[u]`.

Putting it together...

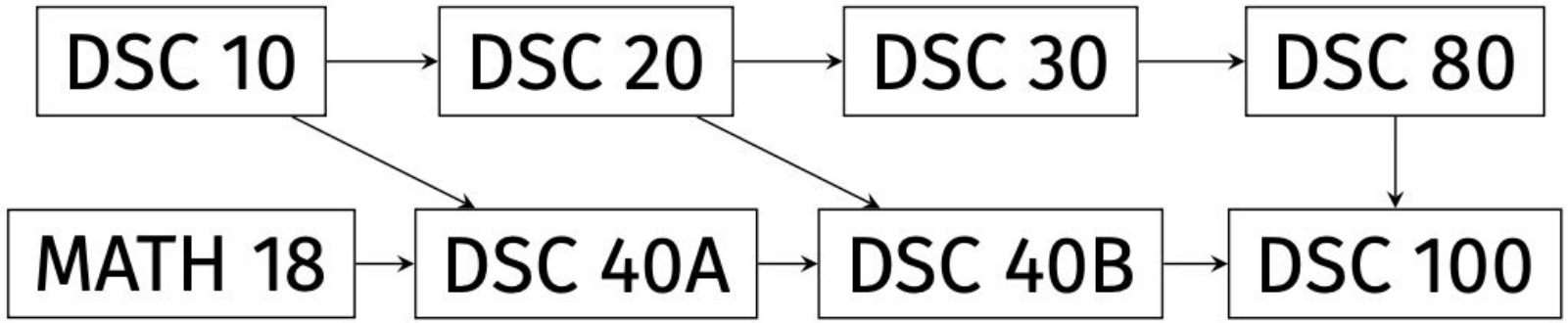
- **Observation**: If v is reachable from u, then v must come after u in the topological sort.
- **Recall**: If v is reachable from u, then `finish[v] < finish[u]`.

Compute start and finish times using DSC 10 as the source.



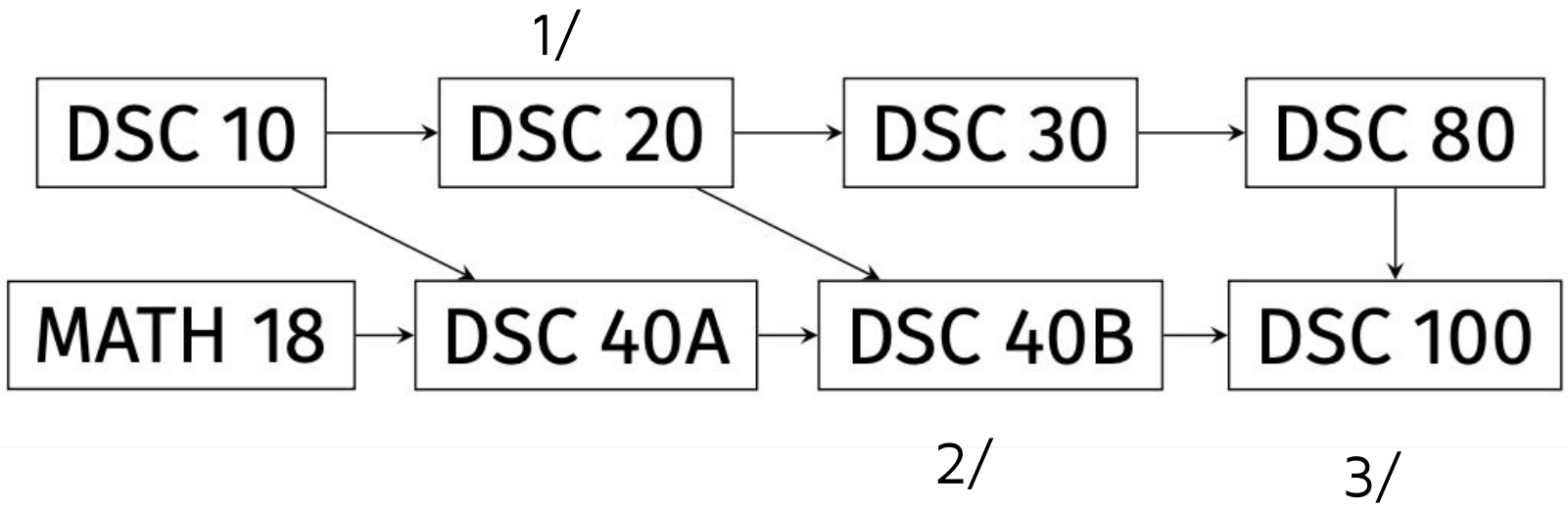
Compute start and finish times using DSC 10 as the source.

1/

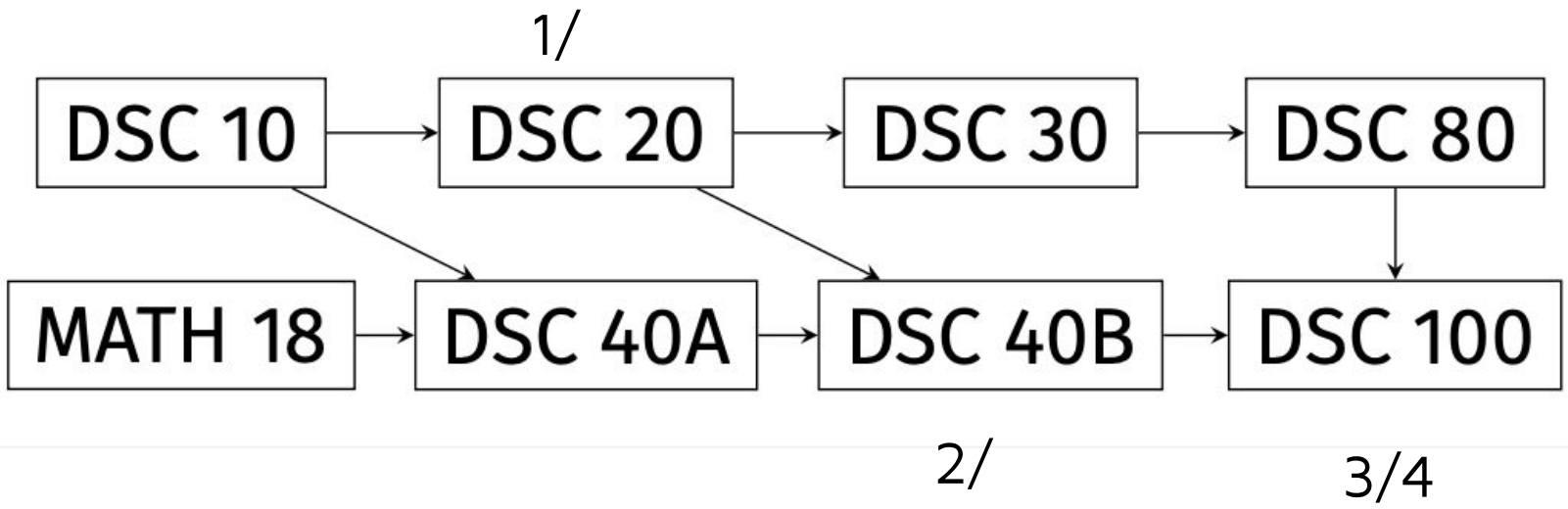


2/

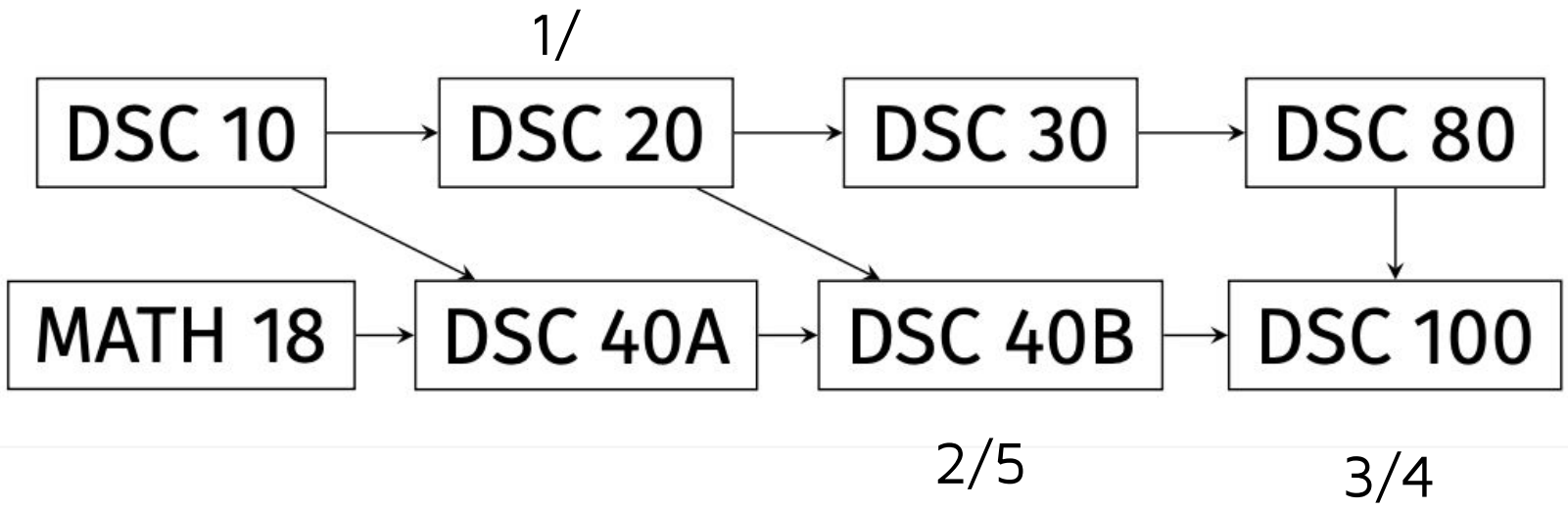
Compute start and finish times using DSC 10 as the source.



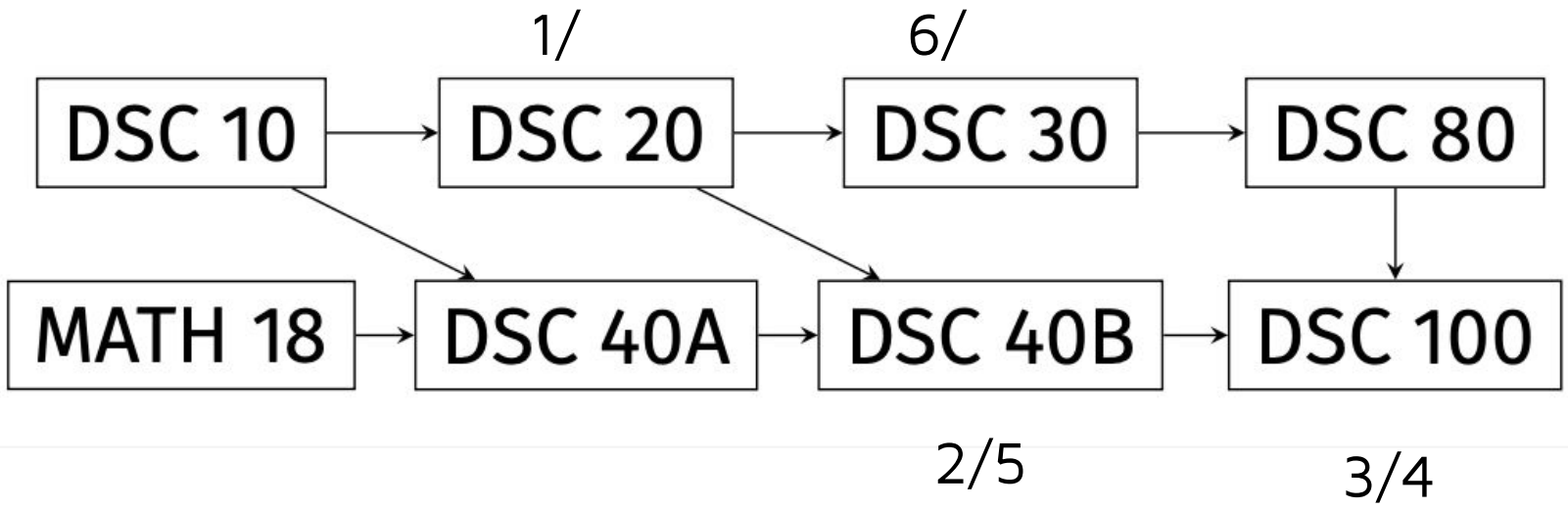
Compute start and finish times using DSC 10 as the source.



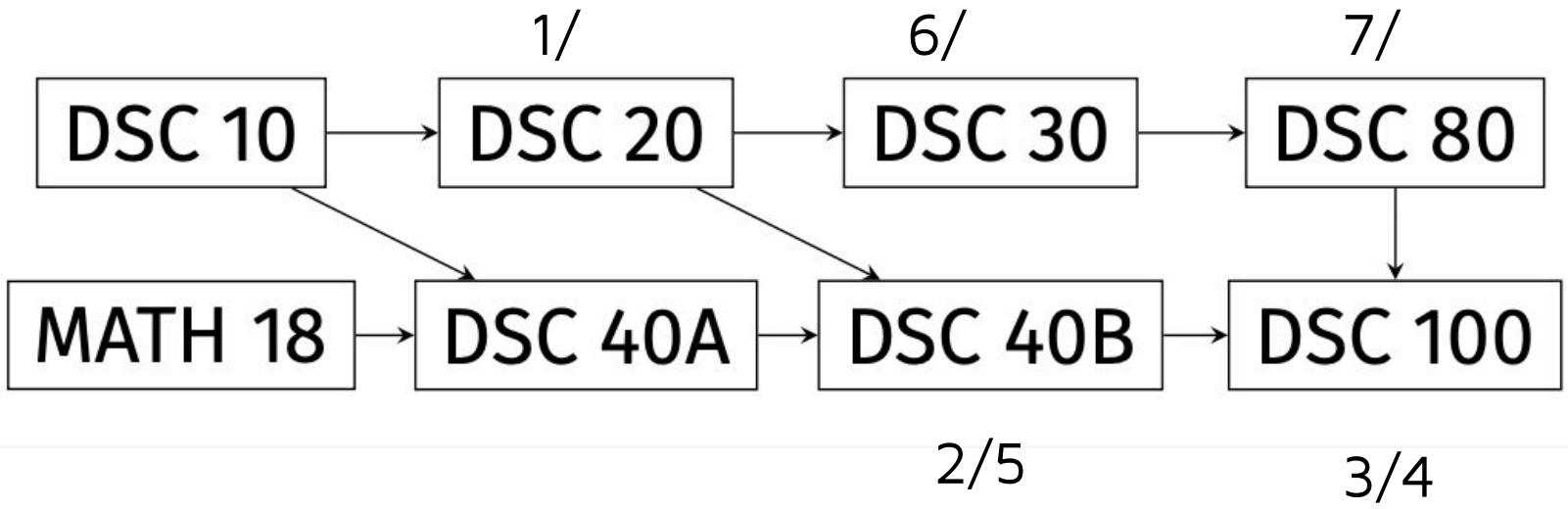
Compute start and finish times using DSC 10 as the source.



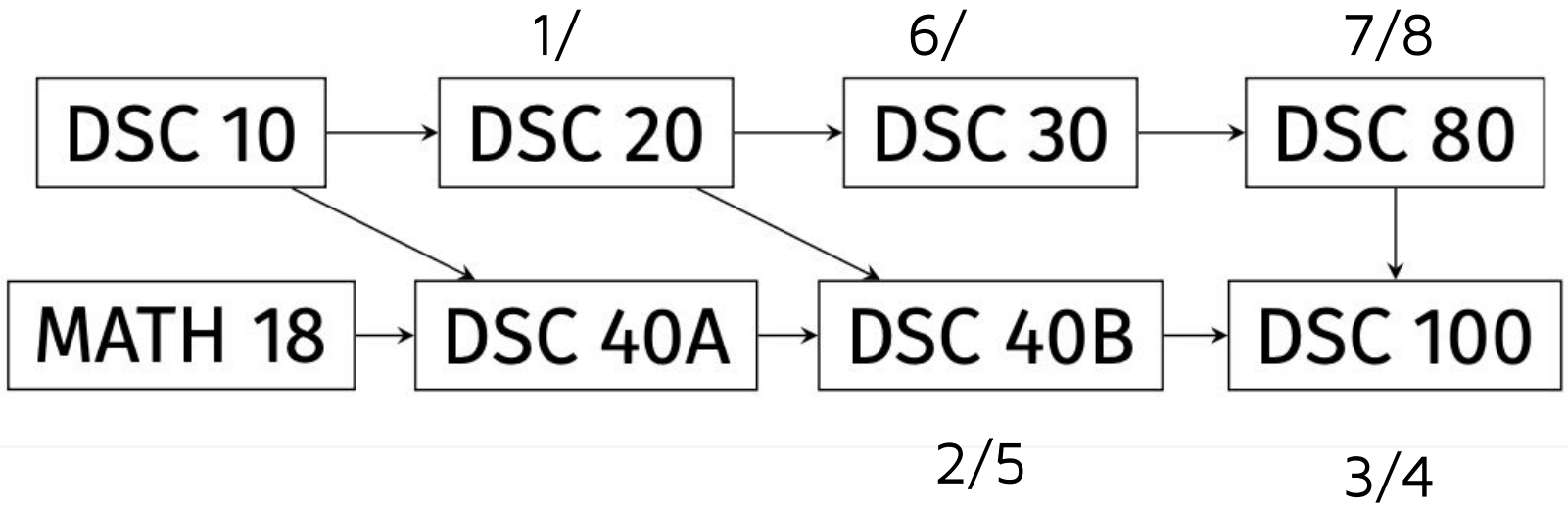
Compute start and finish times using DSC 10 as the source.



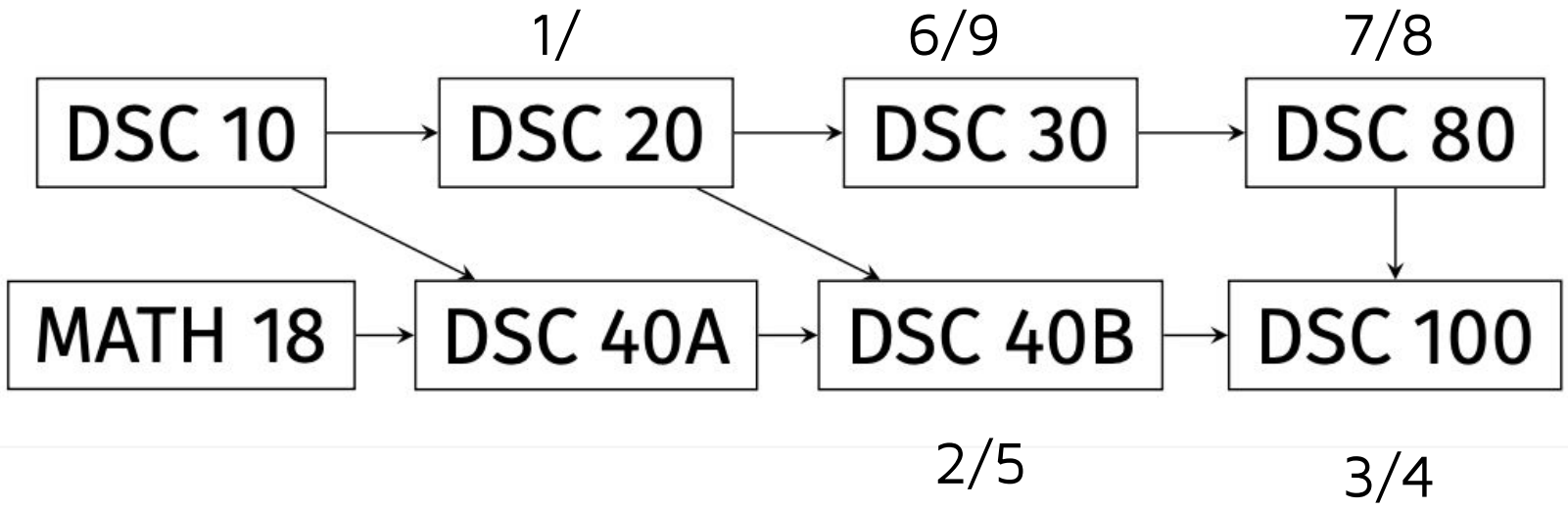
Compute start and finish times using DSC 10 as the source.



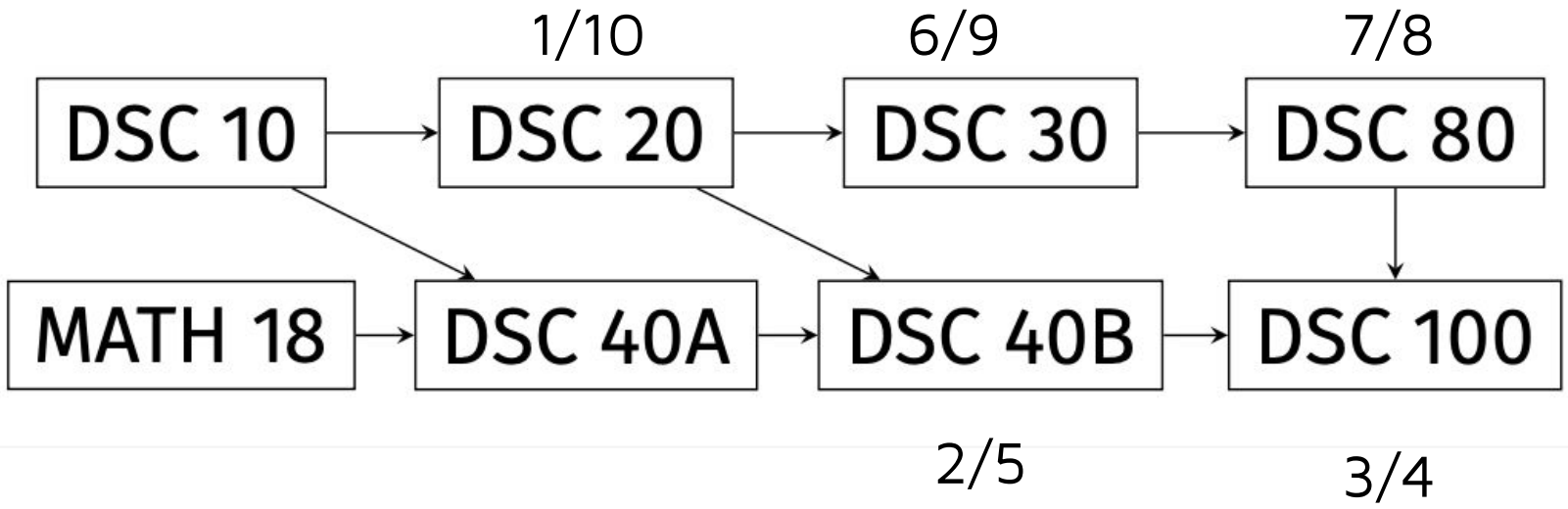
Compute start and finish times using DSC 10 as the source.



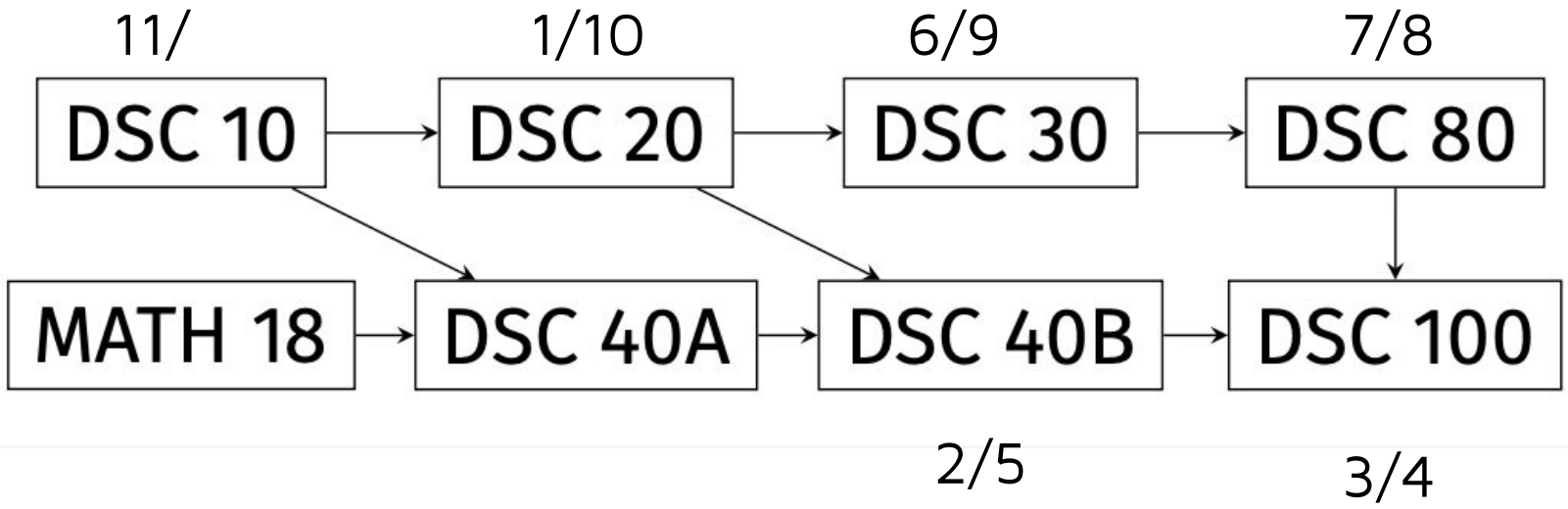
Compute start and finish times using DSC 10 as the source.



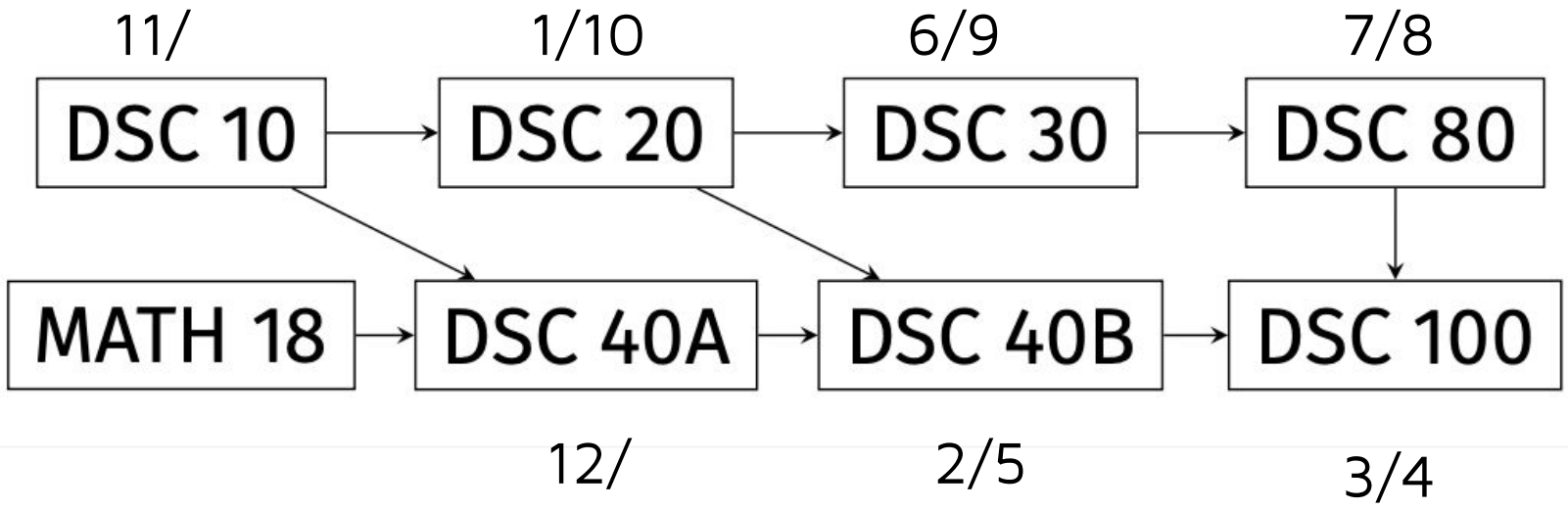
Compute start and finish times using DSC 10 as the source.



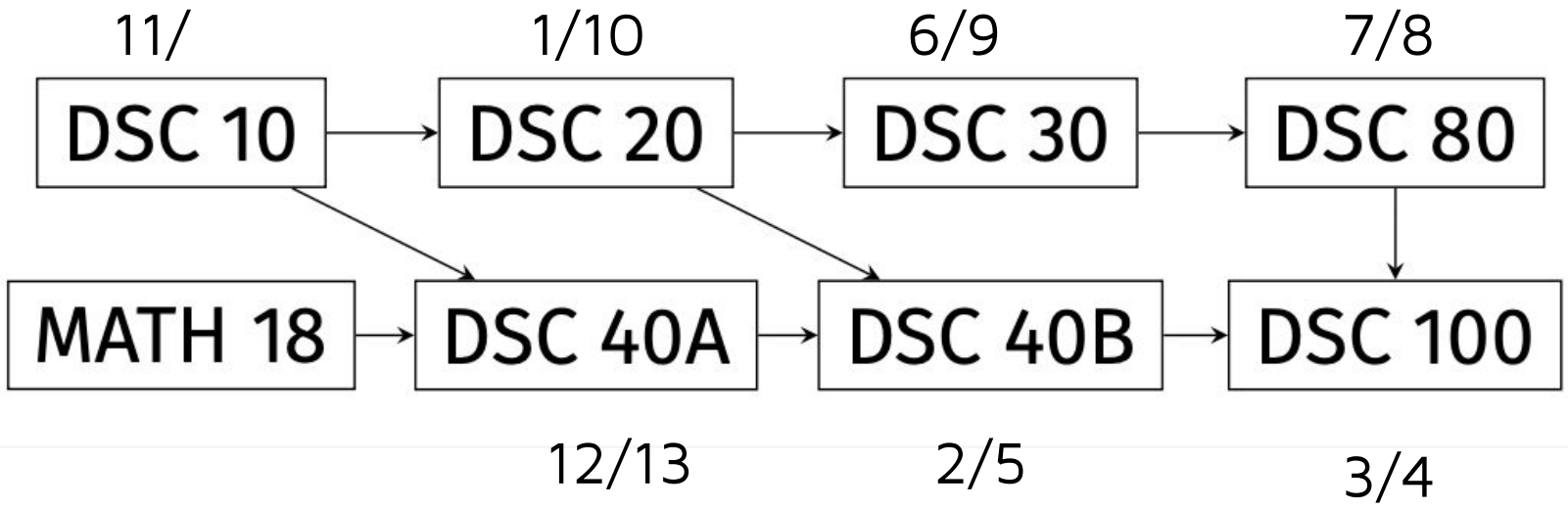
Compute start and finish times using DSC 10 as the source.



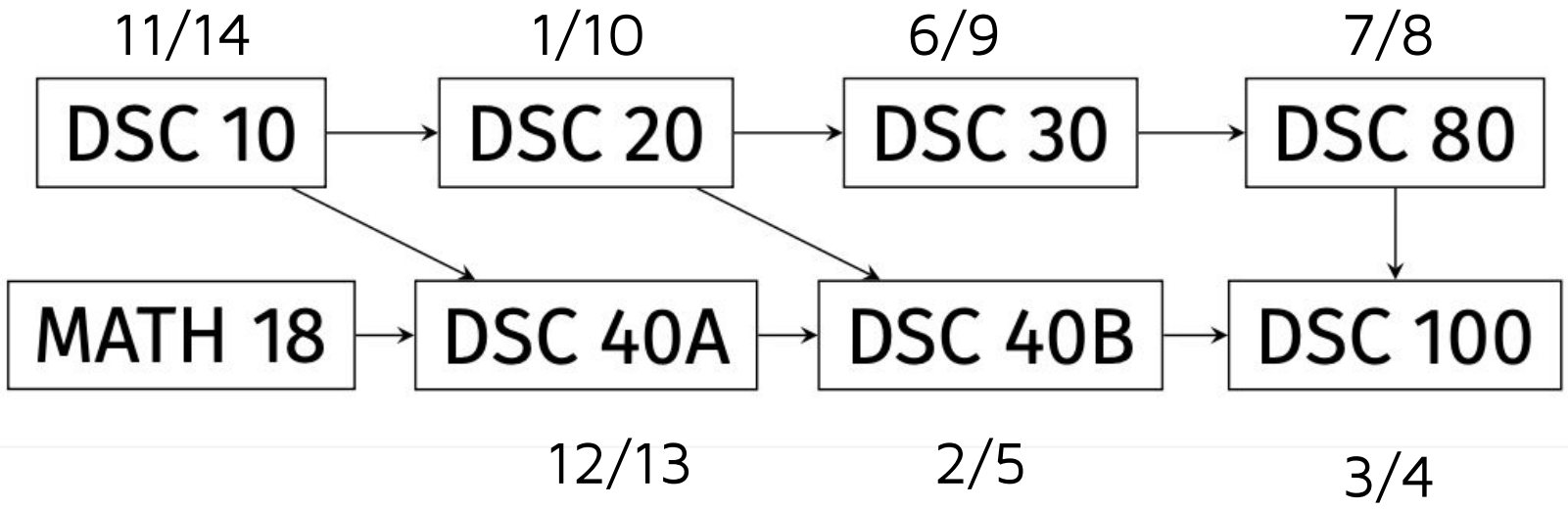
Compute start and finish times using DSC 10 as the source.



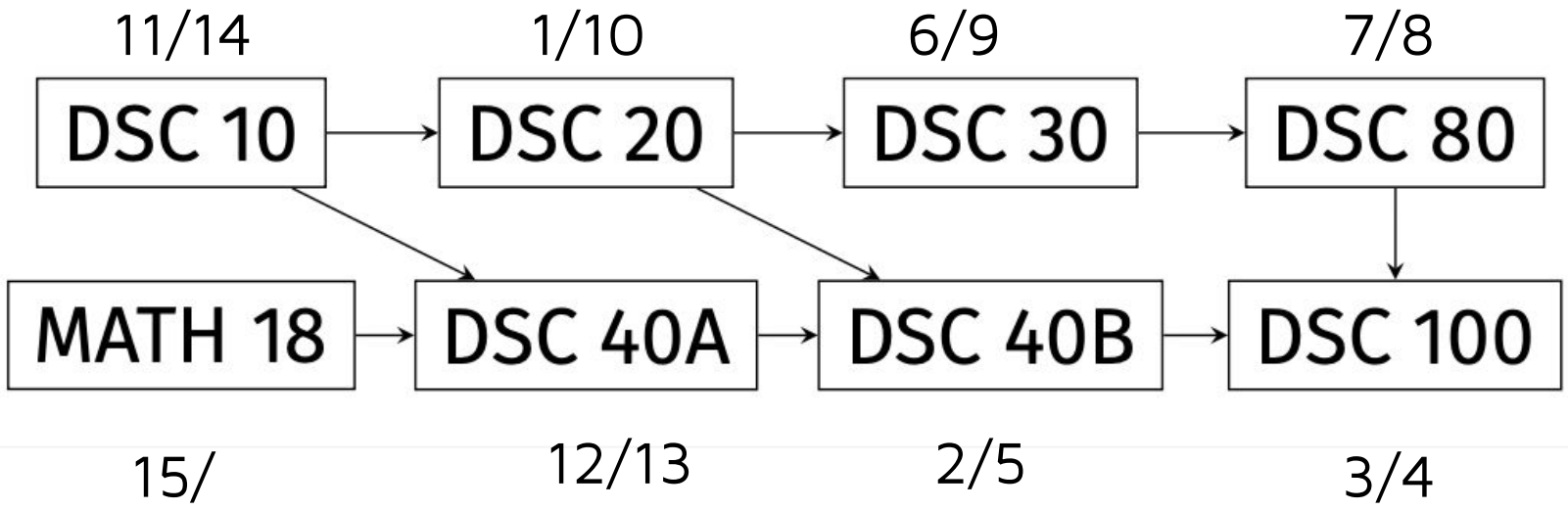
Compute start and finish times using DSC 10 as the source.



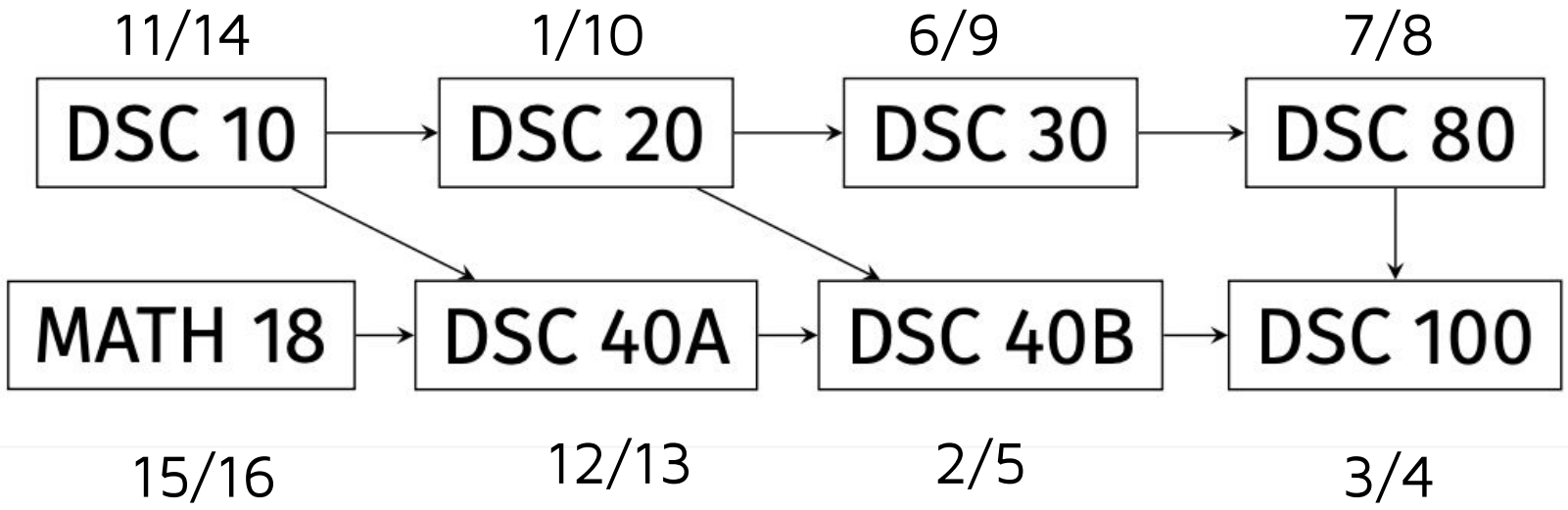
Compute start and finish times using DSC 10 as the source.



Compute start and finish times using DSC 10 as the source.

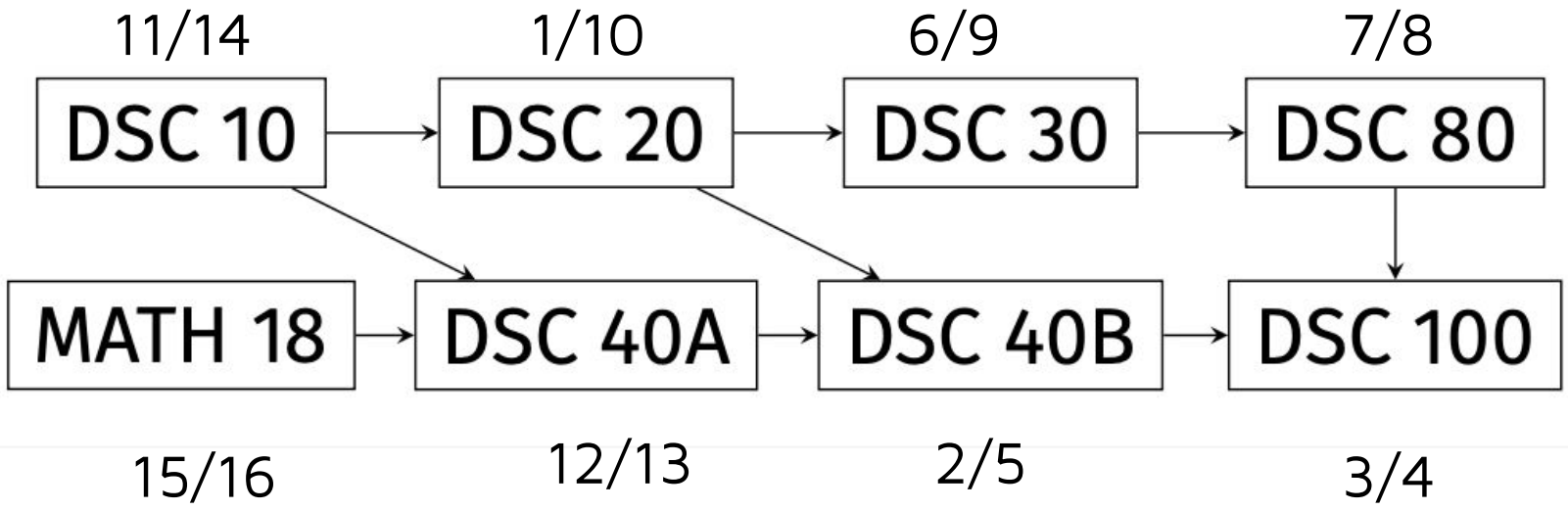


Compute start and finish times using DSC 10 as the source.



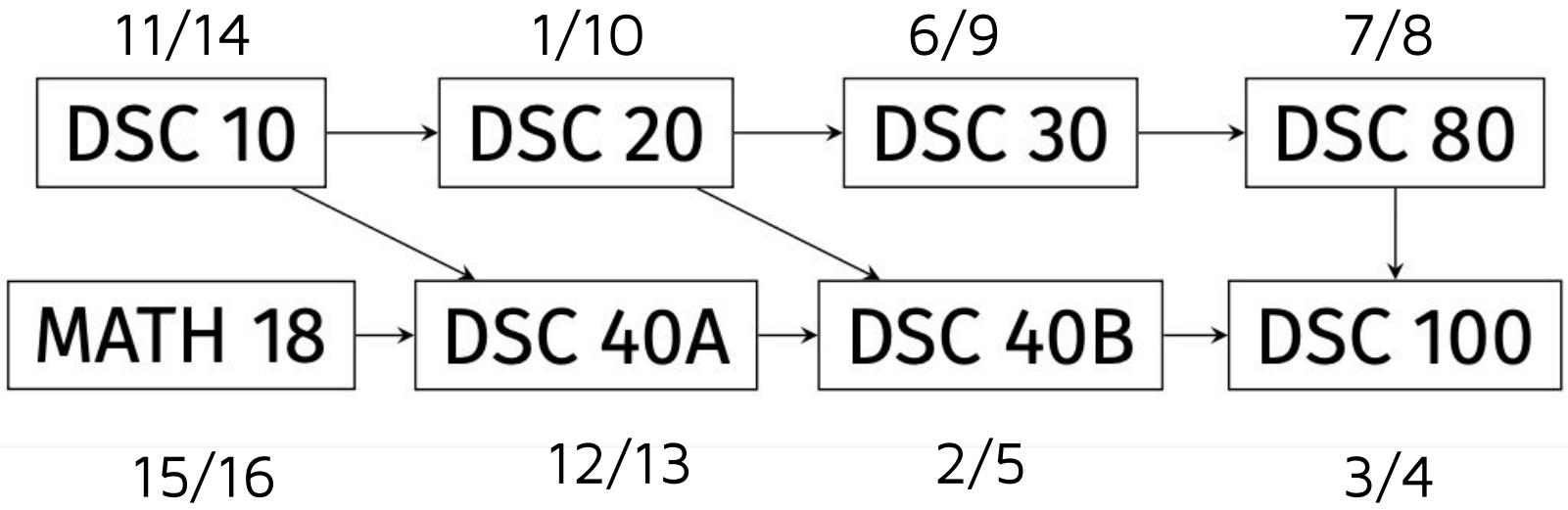
What is next?

Compute start and finish times using DSC 10 as the source.



16, 14, 13, 10, 9, 8, 5, 4

Compute start and finish times using DSC 10 as the source.



M18, D10, D40A, D20, D30, D80, D40B, D100.

Idea

- **Observation**: If v is reachable from u, then v must come after u in the topological sort.
- **Recall**: If v is reachable from u, then `finish[v] < finish[u]`.
- **Therefore**: if `finish[v] < finish[u]`, then v must come after u in the topological sort.
- **Idea**: sort nodes in descending order by finish time

Algorithm

- To find a topological sort (if it exists):
  - Compute times with Full DFS.
  - Sort in **descending** order by finish time.
- **Time complexity**:

Algorithm

- To find a topological sort (if it exists):
  - Compute times with Full DFS.
  - Sort in **descending** order by finish time.
- **Time complexity**: V + E (what else?)

Algorithm

- To find a topological sort (if it exists):
  - Compute times with Full DFS.
  - Sort in **descending** order by finish time.
- **Time complexity**: $V + E + V \log V$

Algorithm

- To find a topological sort (if it exists):
  - Compute times with Full DFS.
  - Sort in **descending** order by finish time.
- **Time complexity**: $\Theta(E + V \log V)$

Thank you!

Do you have any questions?

CampusWire!