DSC 40B

Lecture 19 : BFS

Graph Search Strategies

How do we:

- determine if there is a path between two nodes?
- check if graph is connected?
- count connected components?

Search Strategies

- A **search strategy** is a procedure for *exploring* a graph.
- Different strategies are useful in different situations.

Node Statuses

- At any point during a search, a node is in exactly one of three states:
  - **visited**
  - **pending** (discovered, but not *yet* visited)
  - **undiscovered**

Rules

- At every step, next **visited** node chosen from among **pending** nodes.
- When a node is marked as **visited**, all of its neighbors have been marked as **pending**.

Choosing the next Node

- How to choose among **pending** nodes?
  - **Idea 1**: Visit *newest* pending (**depth-first search**).
  - **Idea 2**: Visit *oldest* pending (**breadth-first search**).

Main Idea

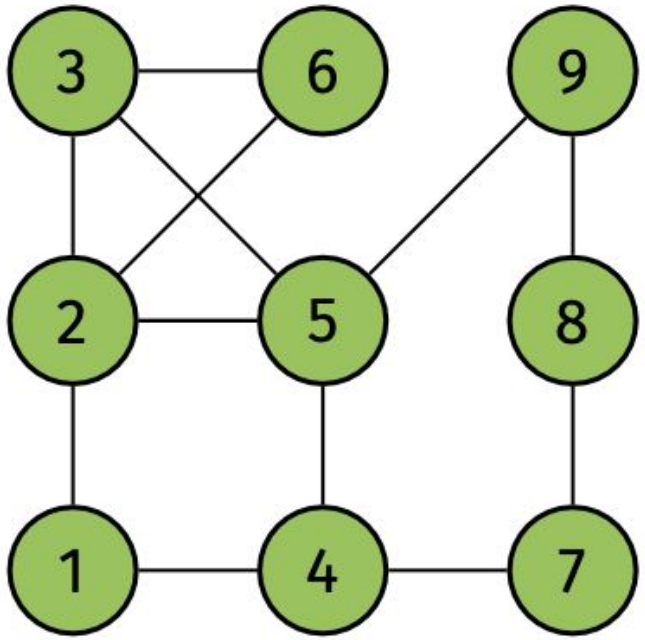
- **Depth-first search** (DFS) and **breadth-first search** (BFS) each discover *different* properties of the graph.
- For example, we'll see that BFS is useful for finding **shortest paths** (DFS in general is not).

Breadth-First Search

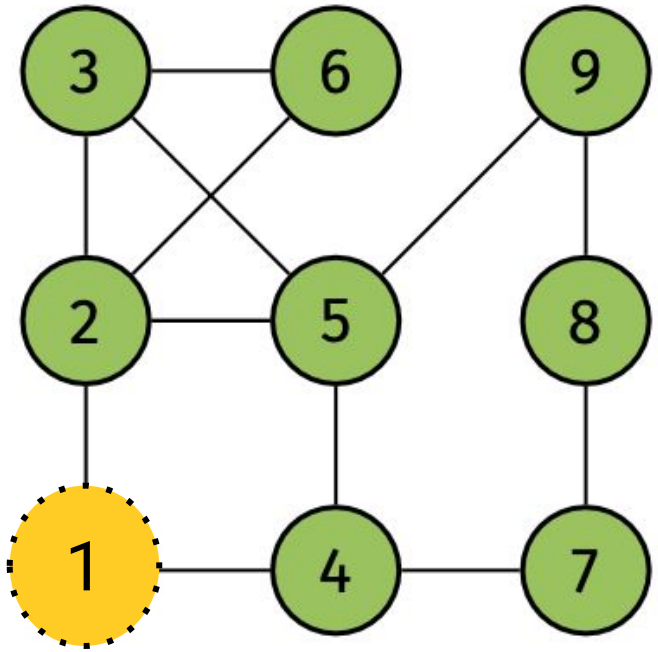
Breadth-First Search

- At every step:
  - Visit **oldest** **pending** node.
  - Mark its **undiscovered** neighbors as **pending**.
- **Convention**: in this class, neighbors produced in ***sorted*** order.
  - In general, the order in which a node's neighbors produced is arbitrary.

Example



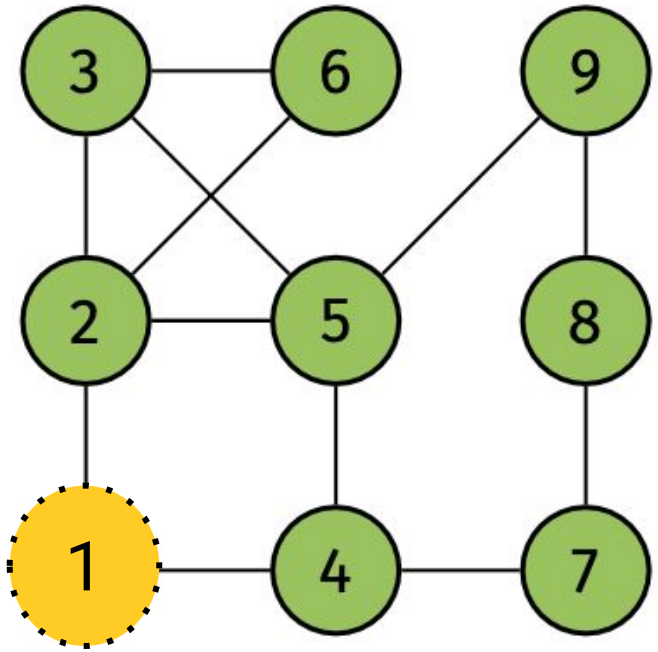
Example




```
pending = [ 1 ]
```

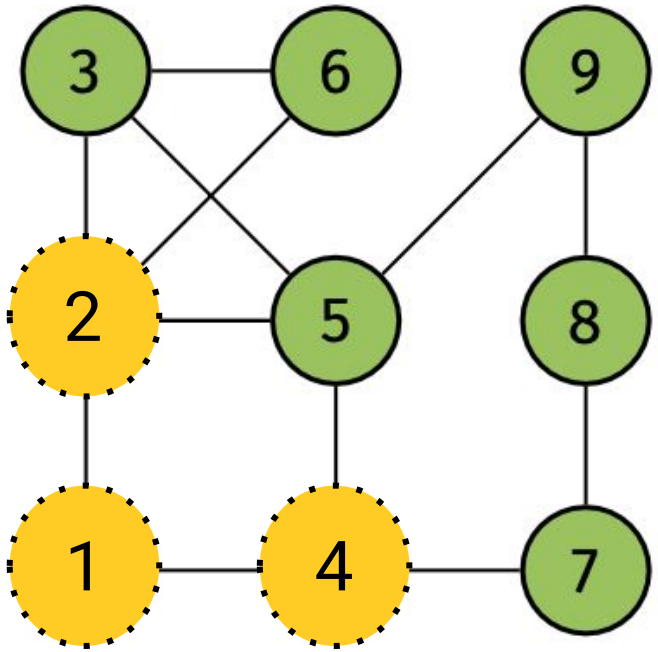
Before iterating.

Example



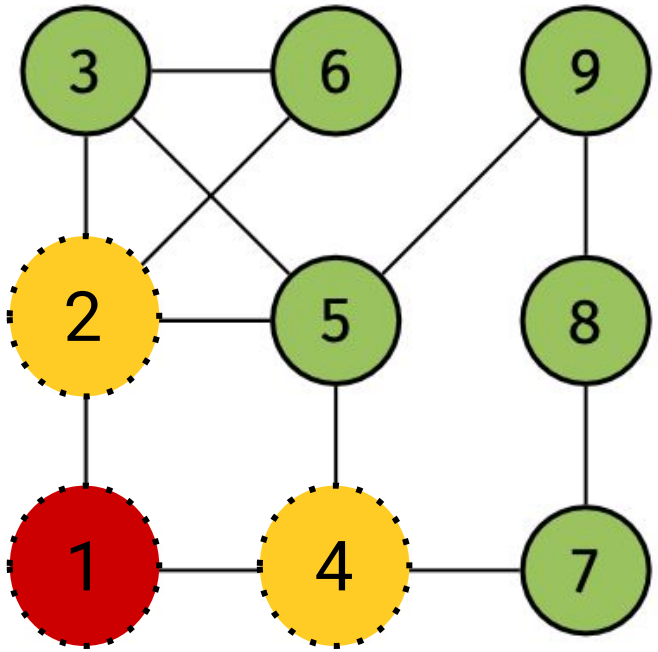

```
pending = [ 1, 2, 4 ]
```

Example




```
pending = [ 1, 2, 4 ]
```

Example

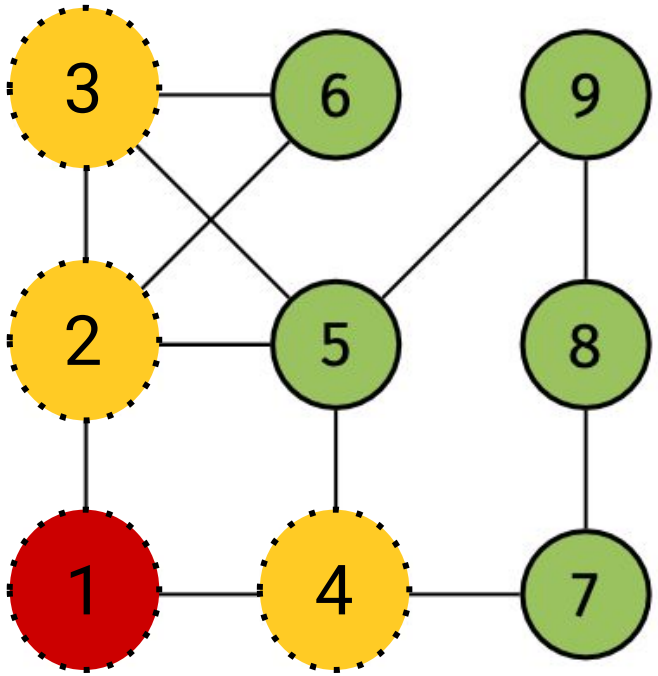


After 1st iteration.

```
pending = [ 2, 4 ]
```

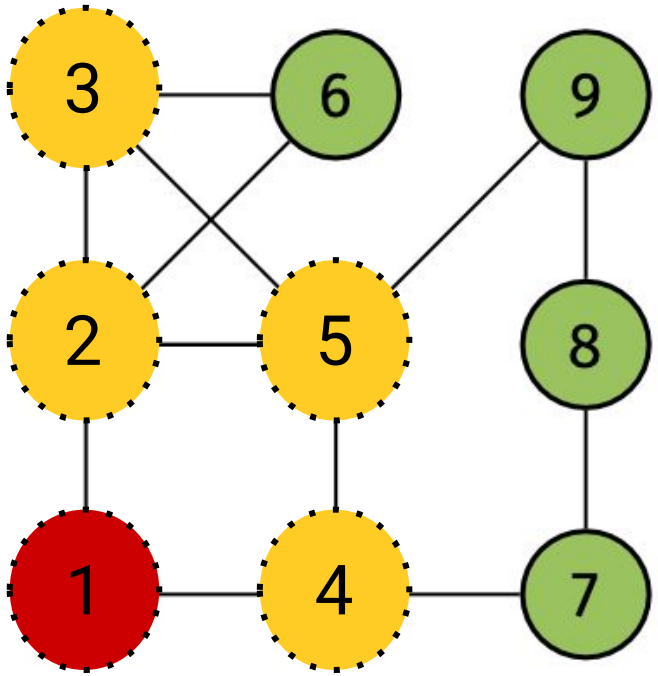
- **visited**
- **pending** (discovered, but not *yet* visited)
- **undiscovered**

Example



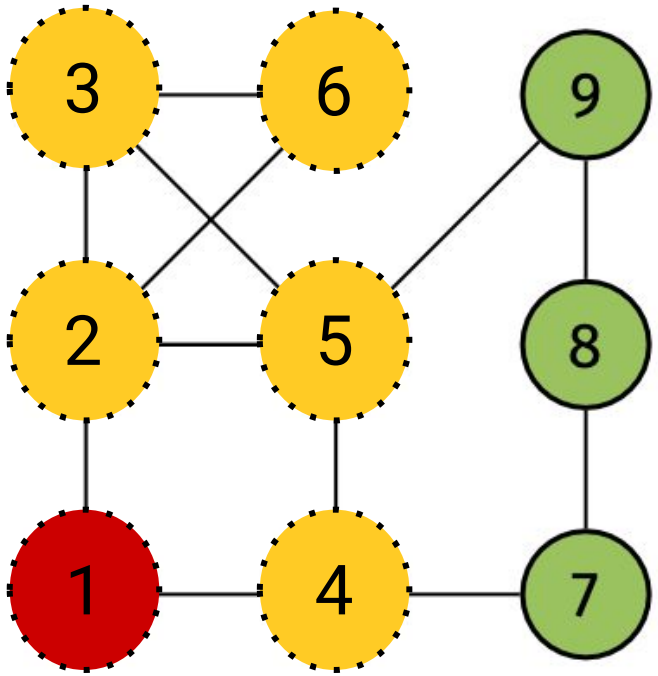

```
pending = [ 2, 4, 3 ]
```

Example



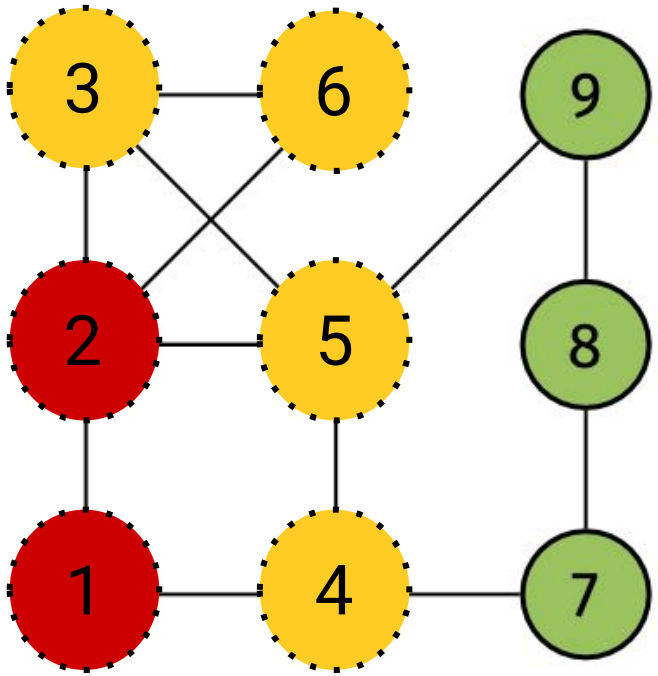

```
pending = [ 2, 4, 3, 5 ]
```

Example




```
pending = [ 2, 4, 3, 5, 6 ]
```

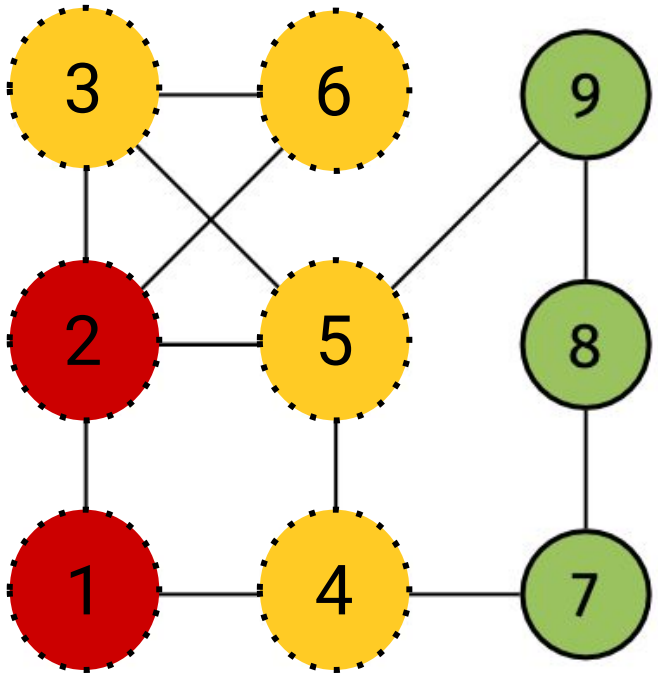
Example




```
pending = [ 4, 3, 5, 6 ]
```

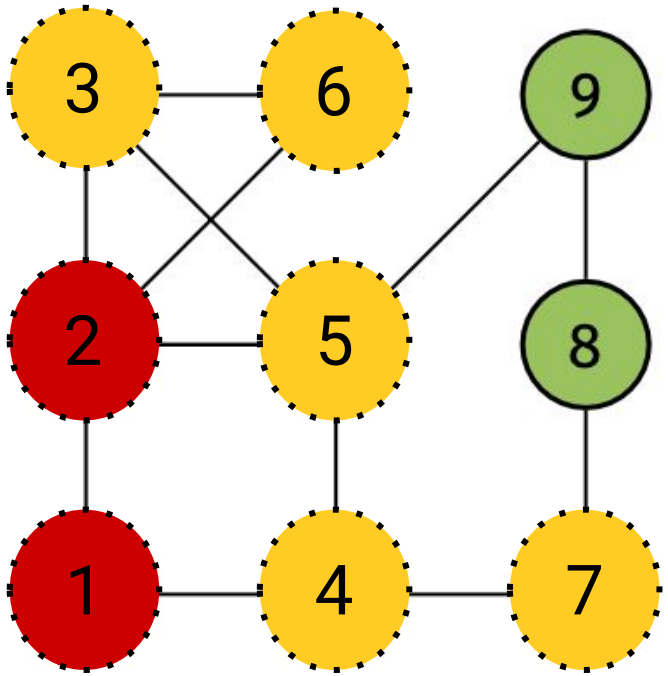
After 2nd iteration.

Example



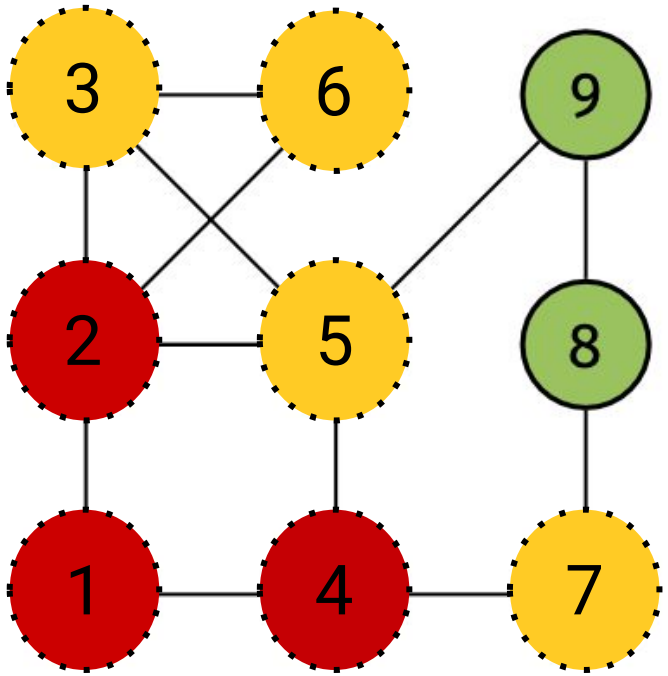

```
pending = [ 4, 3, 5, 6, ? ]
```

Example




```
pending = [ 4, 3, 5, 6, 7 ]
```

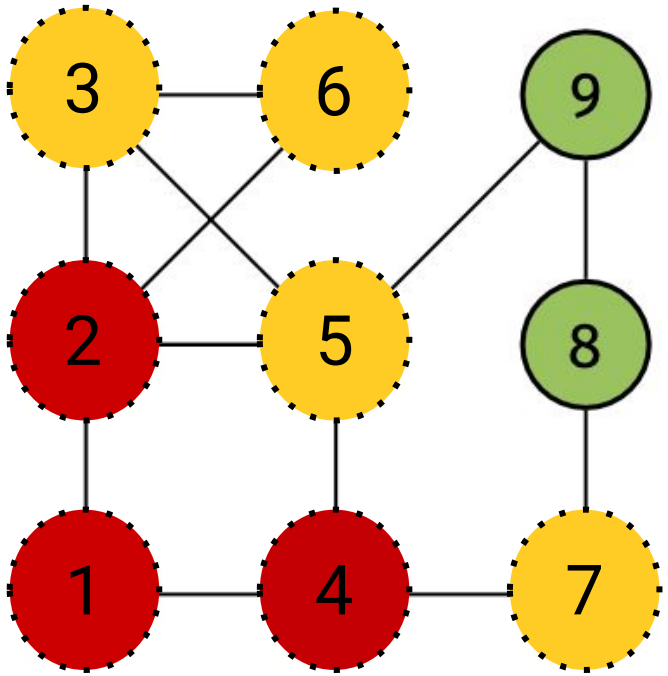
Example




```
pending = [ 3, 5, 6, 7 ]
```

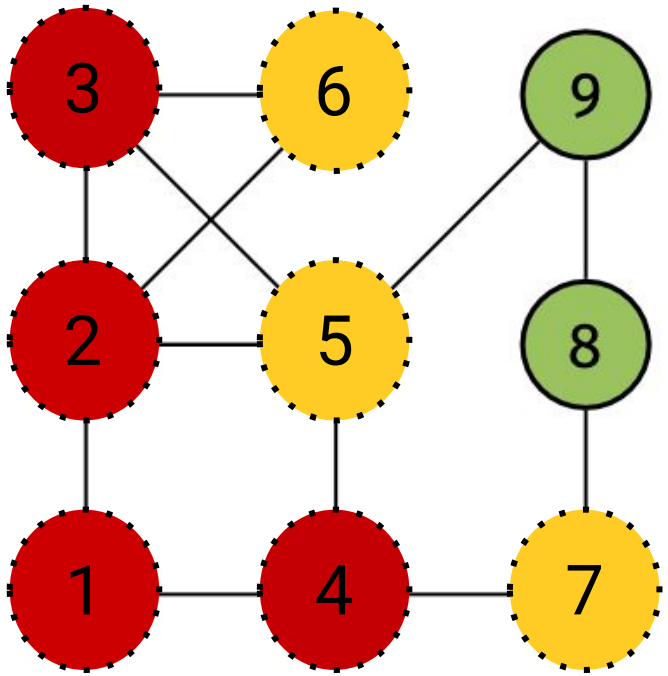
After 3rd iteration

Example




```
pending = [ 3, 5, 6, 7, ? ]
```

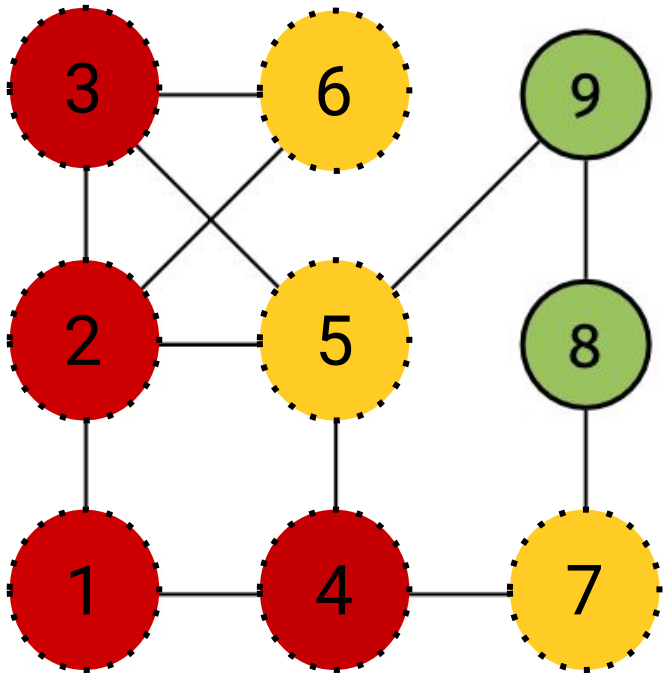
Example



pending = [5, 6, 7]

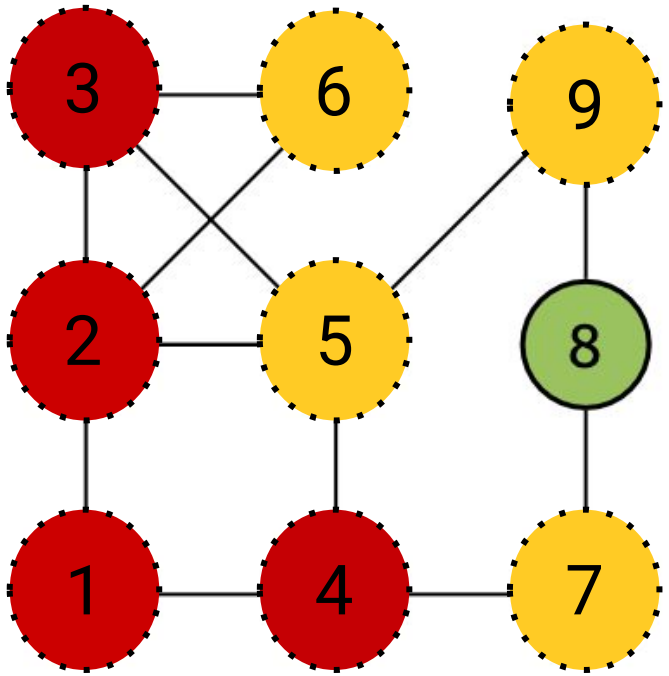
After 4th iteration

Example



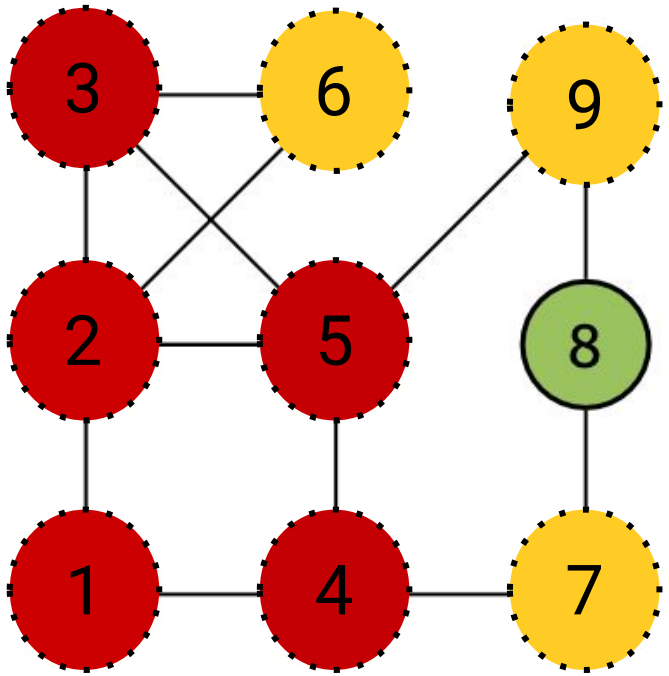

```
pending = [ 5, 6, 7, ?]
```

Example




```
pending = [ 5, 6, 7, 9]
```

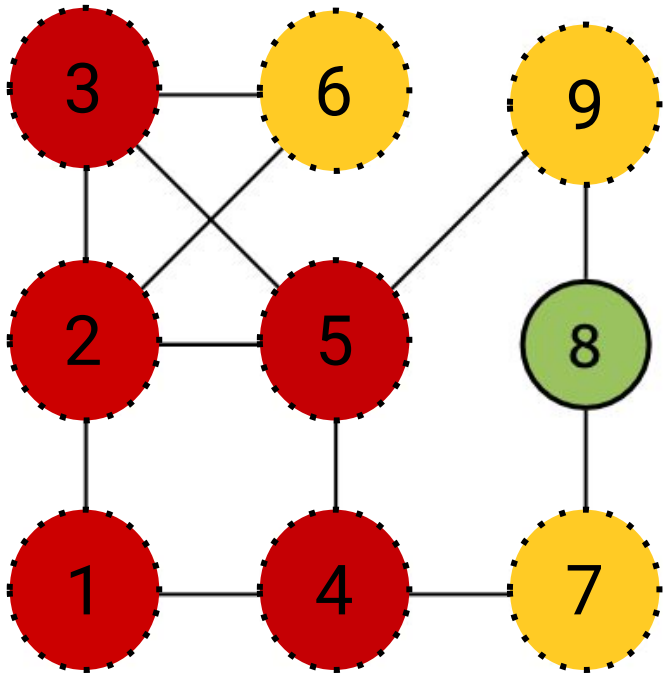
Example



pending = [6, 7, 9]

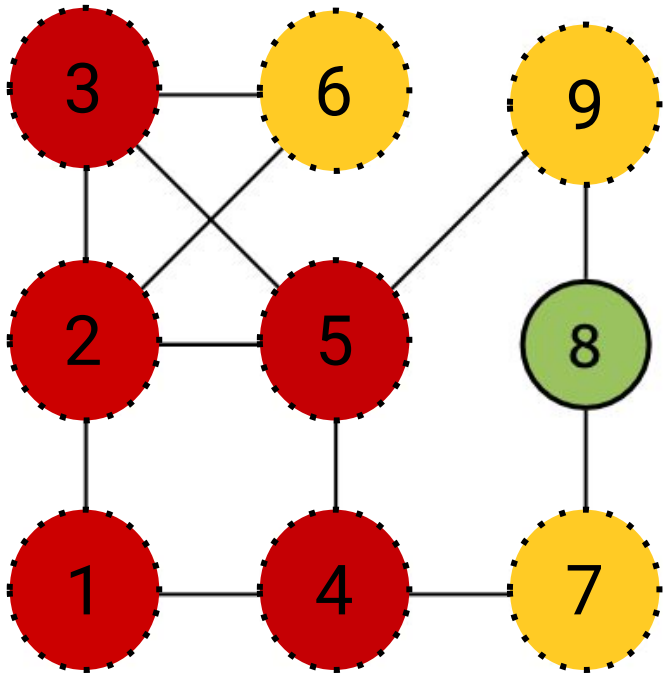
After 5th iteration

Example



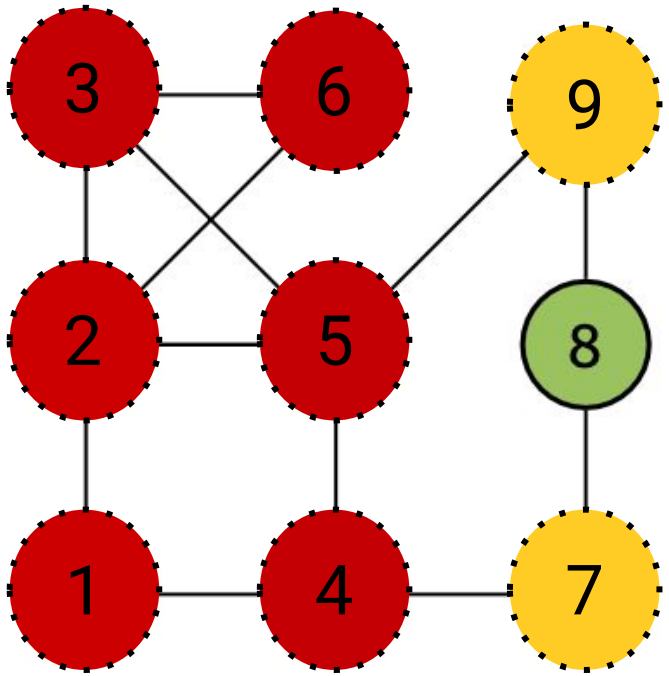

```
pending = [ 6, 7, 9, ?]
```

Example




```
pending = [ 6, 7, 9]
```

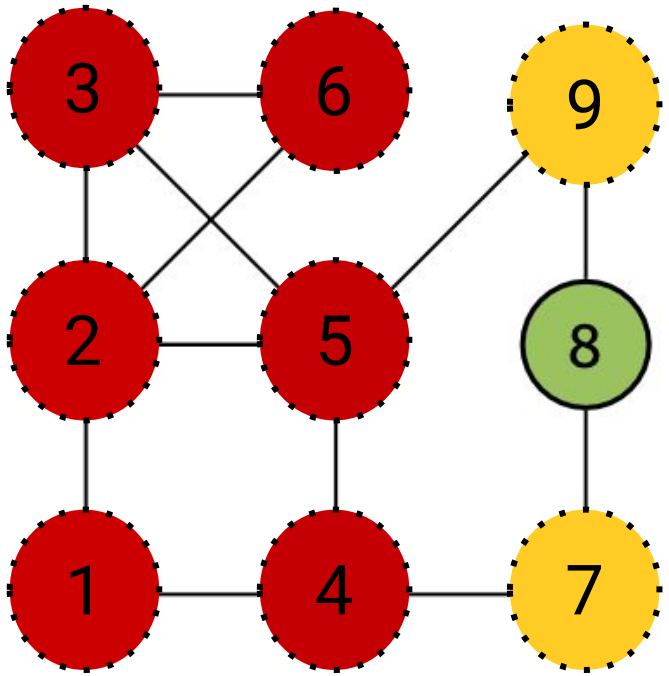
Example



pending = [7, 9]

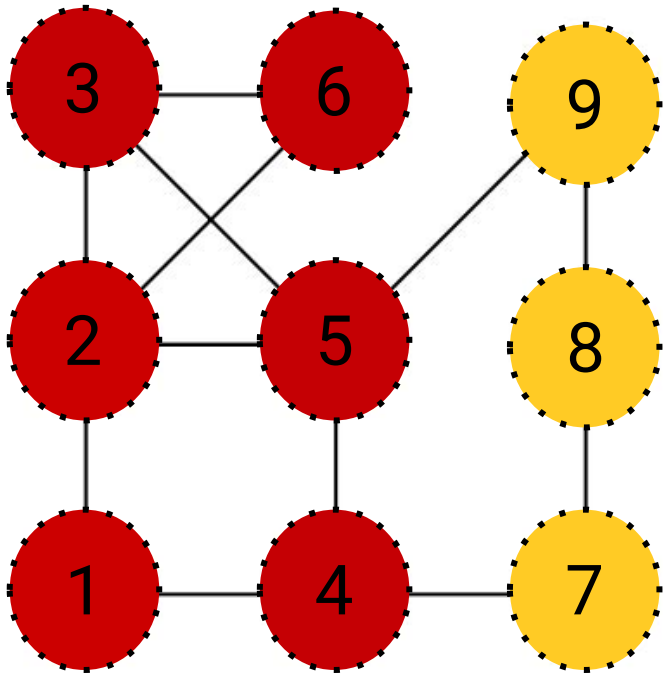
After 6th iteration

Example



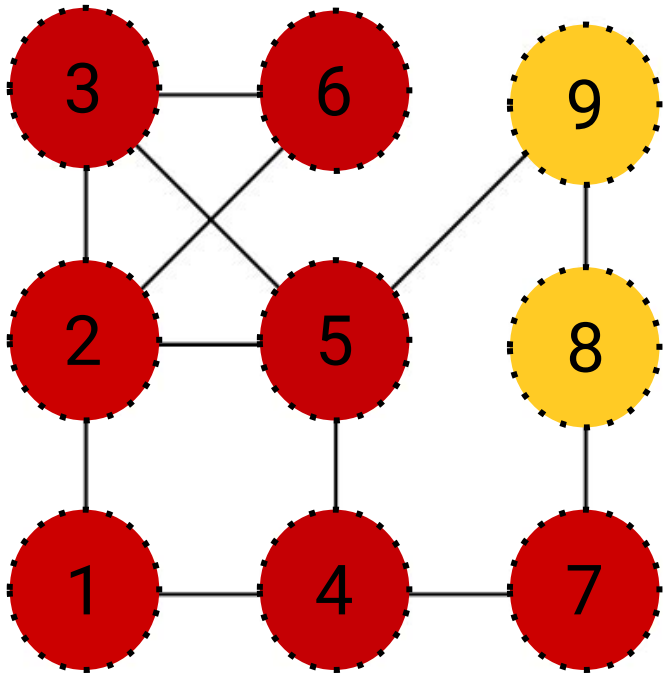

```
pending = [ 7, 9, ?]
```

Example




```
pending = [ 7, 9, 8]
```

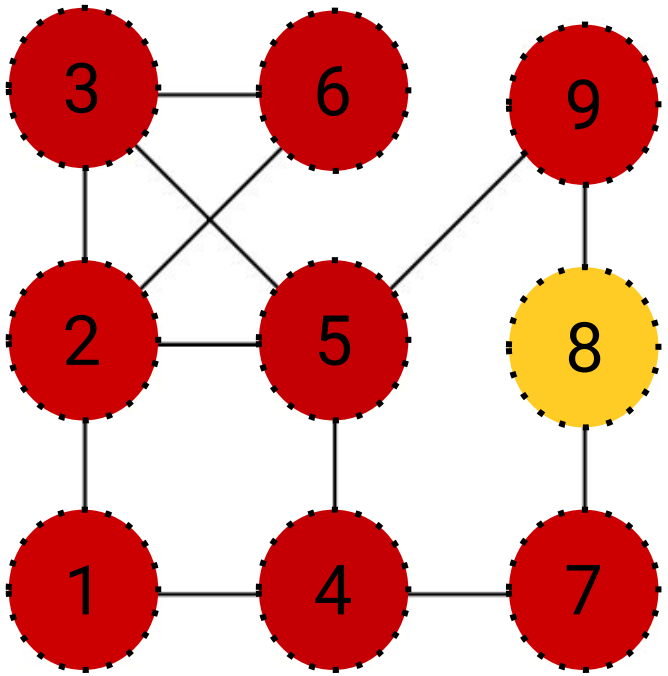
Example



pending = [9, 8]

After 7th iteration

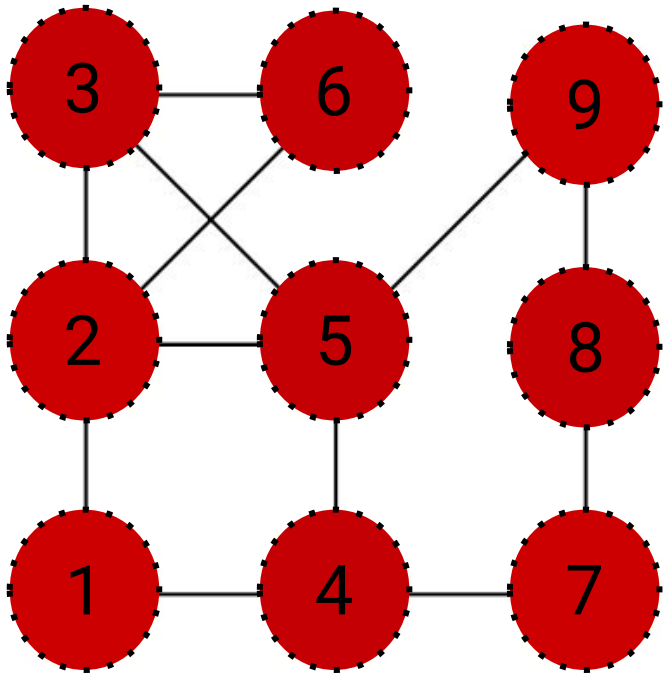
Example



`pending = [8]`

After 8th iteration

Example



pending = []

After 9th iteration

Implementation

- To store **pending** nodes, use a FIFO **queue**.
  - FIFO = “First in, first out”.
- While queue is not empty:
  - Pop a node, u. (remove from the front)
  - Add **undiscovered** neighbors to queue. (to the back)

Queues in Python

- Want Θ(1) time pops/appends on either side.
- `from collections import deque` (“deck”).
  - .popleft() and.pop()
  - `list` doesn’t have right time complexity!
  - `import queue` **isn’t** what you want!
- Keep track of node status attribute using dictionary.

```python
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        # EXERCISE: fill this in...
```

```python
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        # EXERCISE: fill this in...
```

At every step:

- Visit **oldest** pending node.
- Mark its **undiscovered** neighbors as **pending**.

```python
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft() #remove the first elem
```

At every step:

- Visit **oldest** **pending** node.
- Mark its **undiscovered** neighbors as **pending**.

```python
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
       u = pending.popleft() #remove the first elem
      for v in graph.neighbors(u):
```

At every step:

- Visit **oldest** **pending** node.
- Mark its **undiscovered** neighbors as **pending**.

```
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
       u = pending.popleft() #remove the first elem
      for v in graph.neighbors(u):
          if status[v] == 'undiscovered':
```

At every step:

- Visit **oldest** **pending** node.
- Mark its **undiscovered** neighbors as **pending**.

```
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
       u = pending.popleft() #remove the first elem
      for v in graph.neighbors(u):
          if status[v] == 'undiscovered':
              status[v] = 'pending'
```

At every step:

- Visit **oldest** **pending** node.
- Mark its **undiscovered** neighbors as **pending**.

```
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
       u = pending.popleft() #remove the first elem
      for v in graph.neighbors(u):
          if status[v] == 'undiscovered':
              status[v] = 'pending'
              pending.append(v)
```

At every step:

- Visit **oldest** **pending** node.
- Mark its **undiscovered** neighbors as **pending**.

```
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
       u = pending.popleft() #remove the first elem
      for v in graph.neighbors(u):
          if status[v] == 'undiscovered':
              status[v] = 'pending'
              pending.append(v)
      status[u] == 'visited'
```

At every step:

- Visit **oldest** **pending** node.
- Mark its **undiscovered** neighbors as **pending**.

BFS

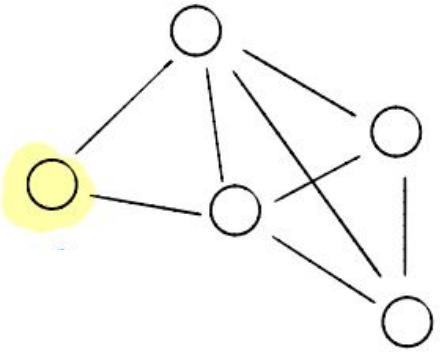
```
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

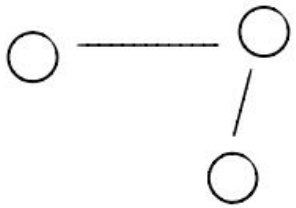
Notes

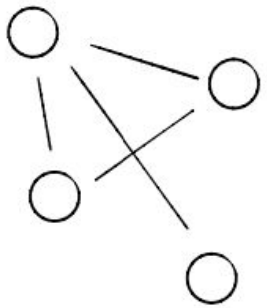
- What does this code actually return?
- Nothing, yet. It is a **foundation**.
- BFS works just as well for directed graphs.

Analysis of BFS

Exercise: What will bfs do when run on a disconnected graph?







A: Discover all nodes

B: Inf. loop

C: Discover only some nodes.

Claim

- `bfs` with source u will visit all nodes **reachable** from u (*and only those nodes*).
- **Useful!**
  - Is there a path between u and v?
  - Is graph connected?

Exploring with BFS

- BFS will visit all nodes **reachable** from source.
- If **disconnected**, BFS will **not visit** all nodes.
- We can do so with a **full BFS**.
  - **Idea**: "re-start" BFS on undiscovered node.
  - Must pass *statuses* between calls.

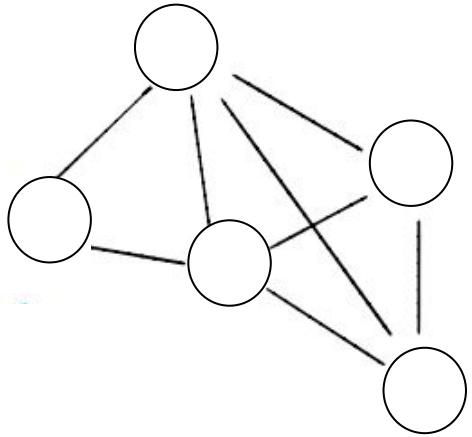
Making Full BFS: Modify bfs to accept statuses

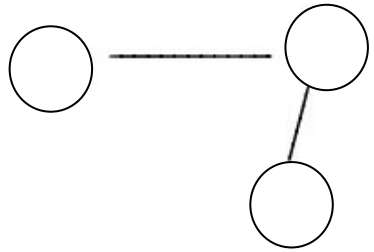
```
def bfs(graph, source, status=None):
"""Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}
        # ...
```

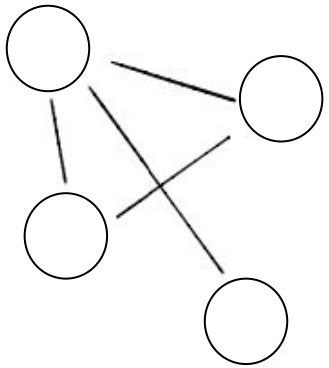
Making Full BFS: Call bfs multiple times

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

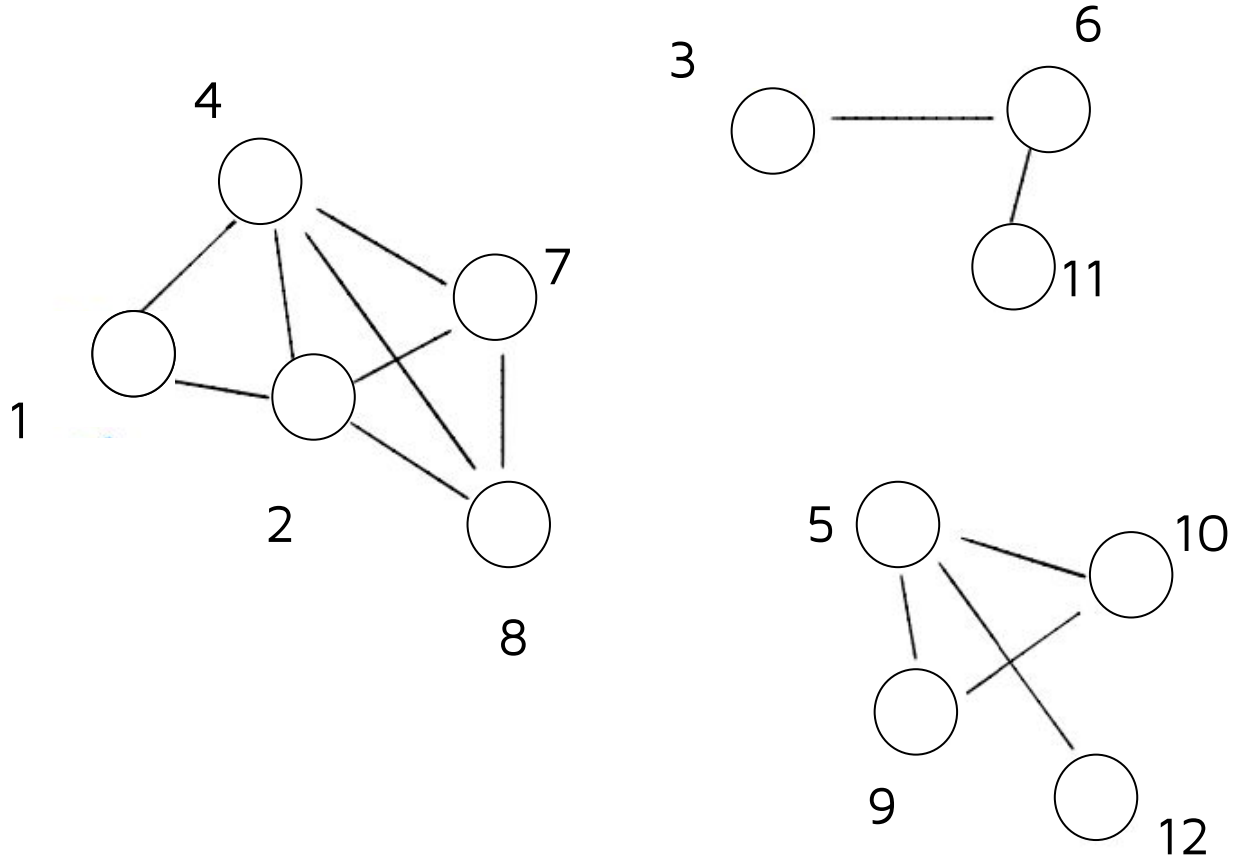
Example



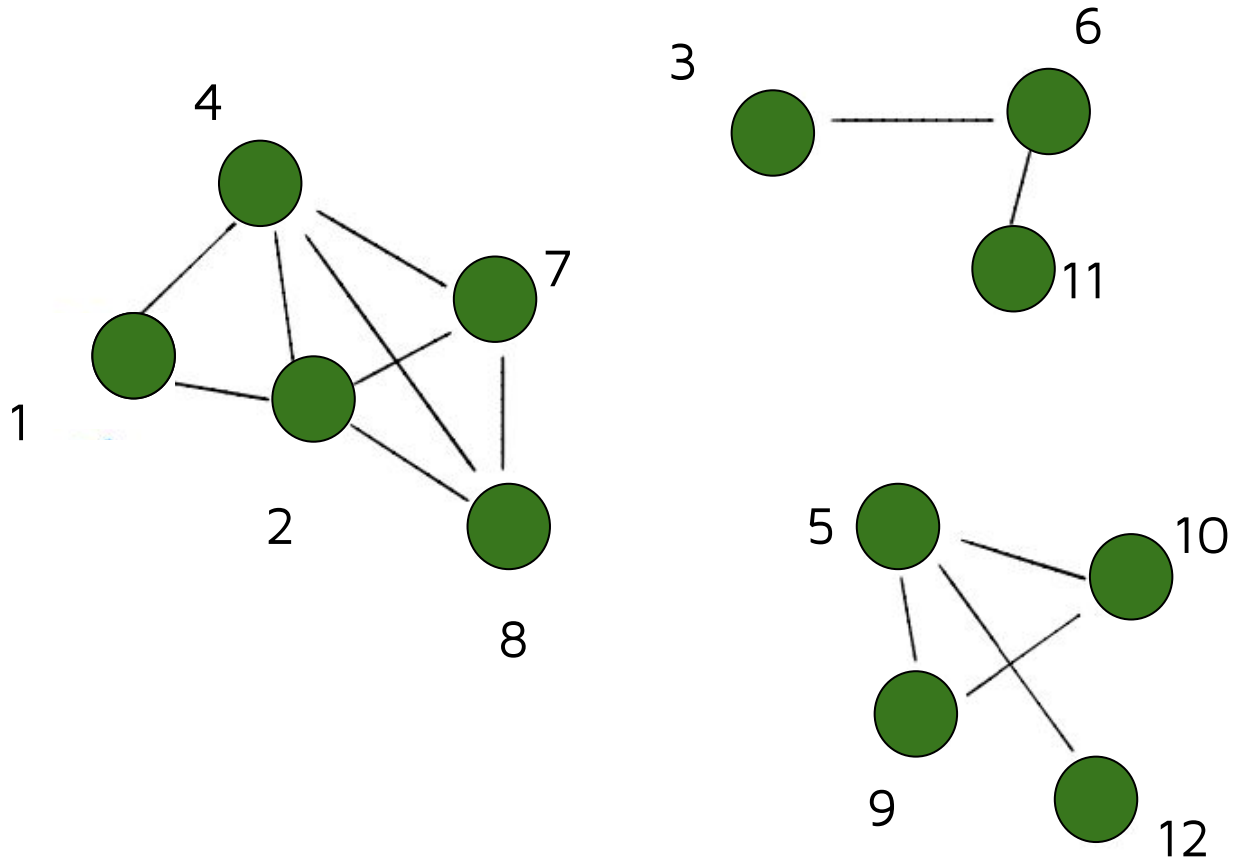




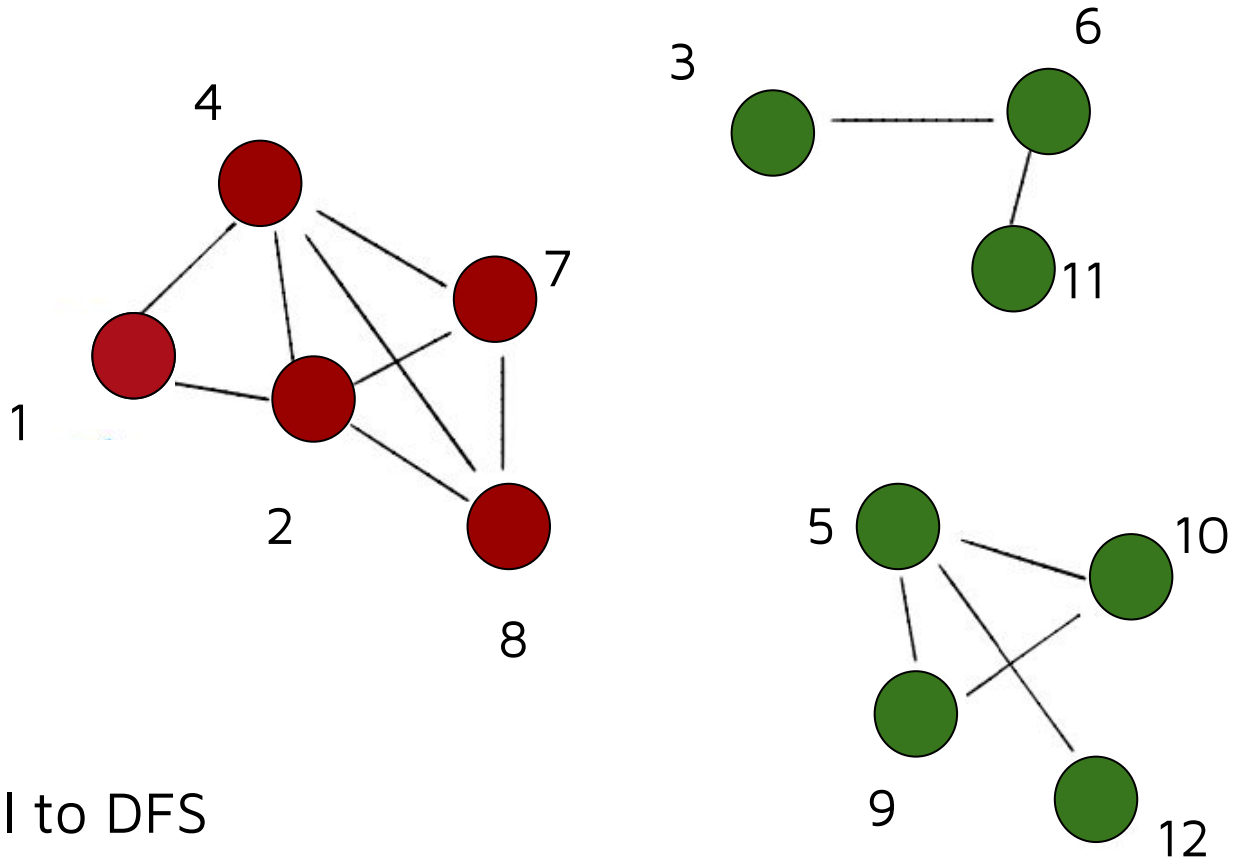
Example



Example

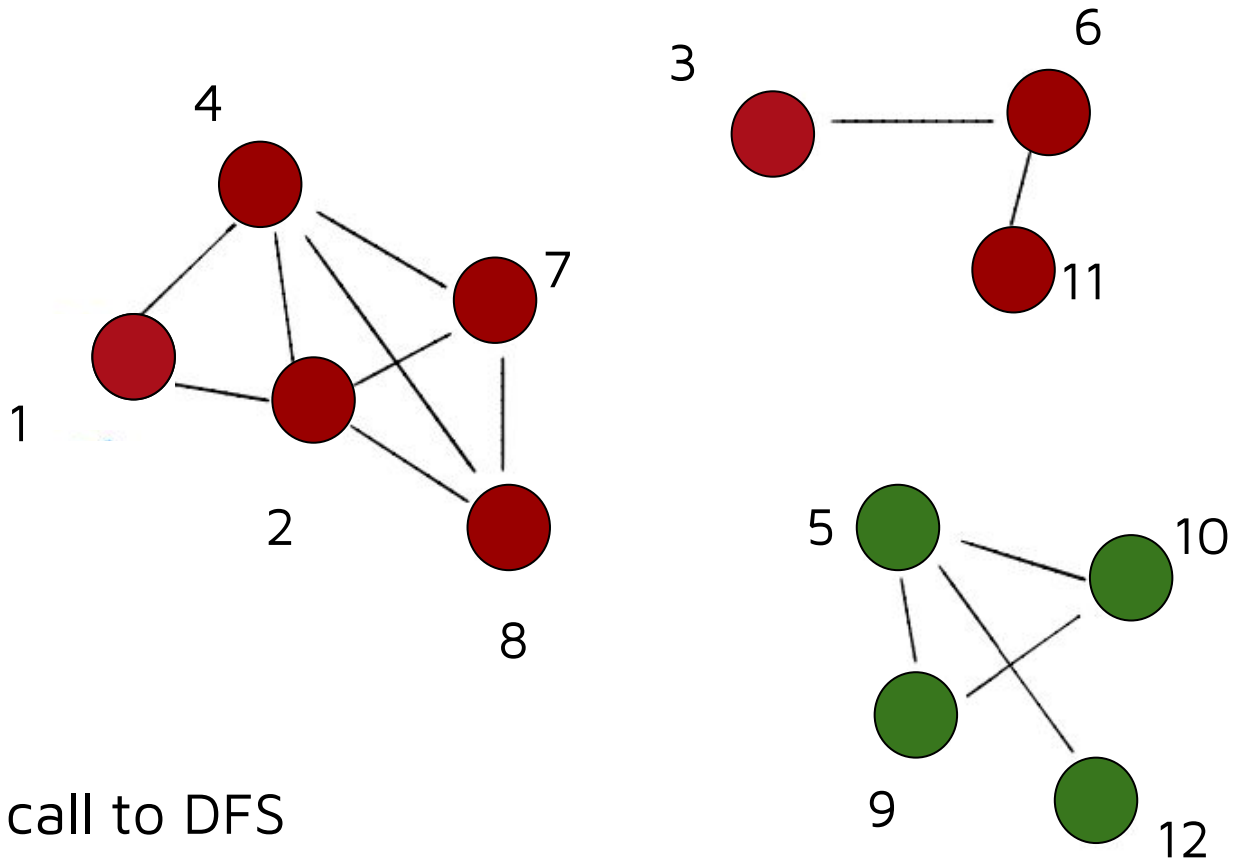


```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
```



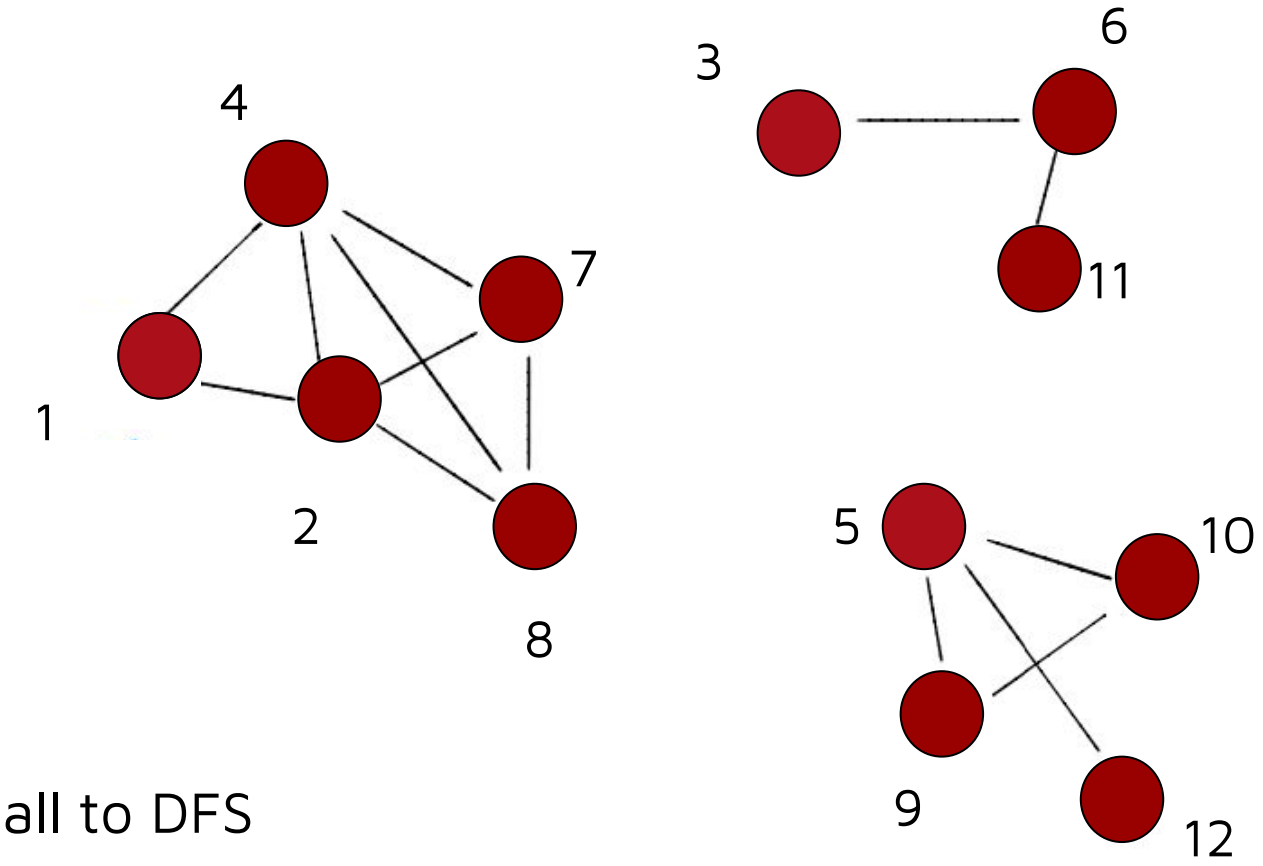
First call to DFS

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



Second call to DFS

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



Third call to DFS

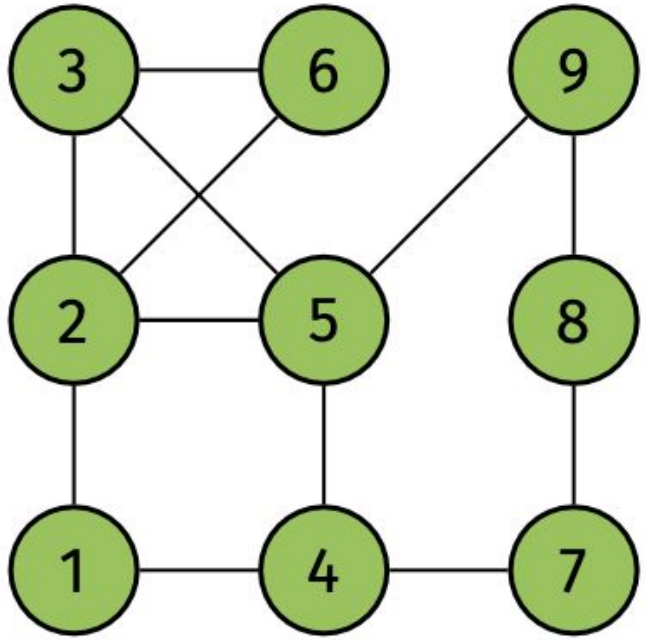
Observation

If there are k connected components, bfs in line 5 is called exactly k times

```
1   def full_bfs(graph):
2       status = {node: 'undiscovered' for node in graph.nodes}
3       for node in graph.nodes:
4           if status[node] == 'undiscovered':
5               bfs(graph, node, status)
```

Exercise

How many times is each node added to the queue in a BFS of the graph below?



A: Once

B: Twice

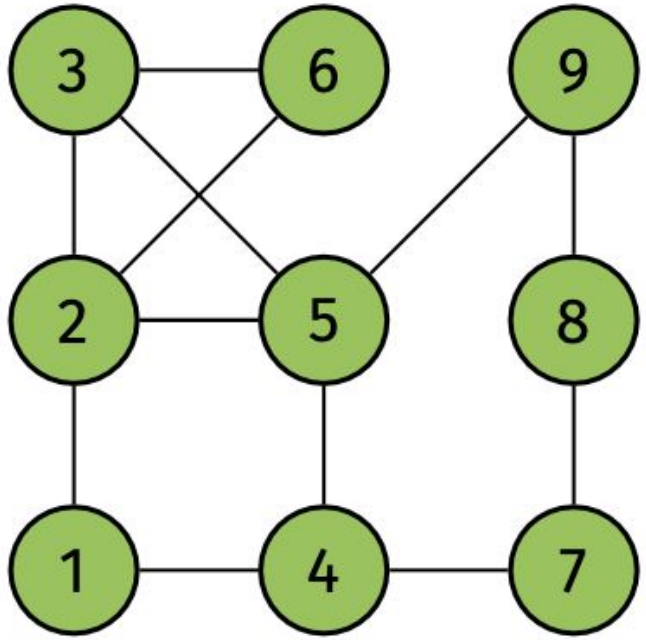
C: E times

D: V times

E: V + E times

Exercise

How many times is each node added to the queue in a BFS of the graph below?



A: Once

B: Twice

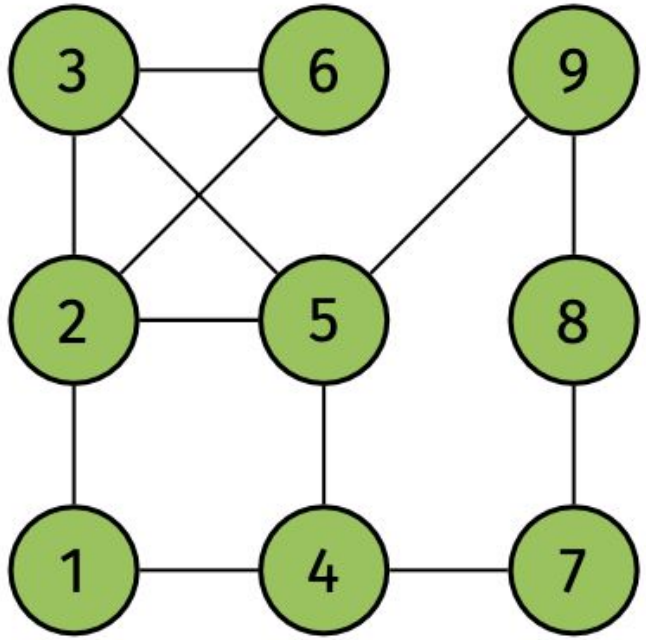
C: E times

D: V times

E: V + E times

Exercise

How many times is each edge "explored" in a BFS of the graph below?



A: Once

B: Twice

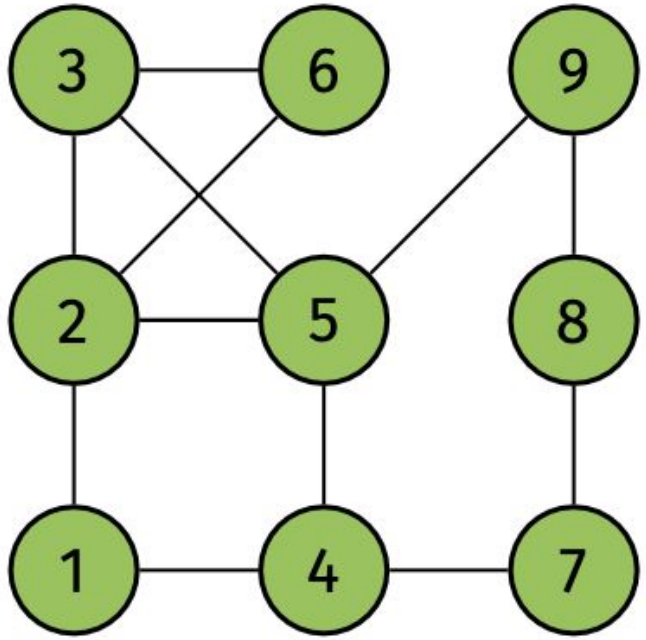
C: E times

D: V times

E: V + E times

Exercise

How many times is each edge "explored" in a BFS of the graph below?



A: Once

B: Twice

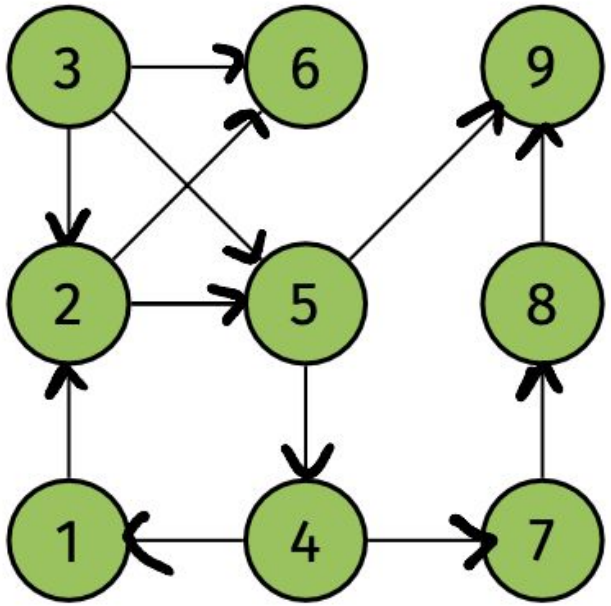
C: E times

D: V times

E: V + E times

Exercise

How many times is each edge "explored" in a BFS of the **directed** graph below?



A: Once

B: Twice

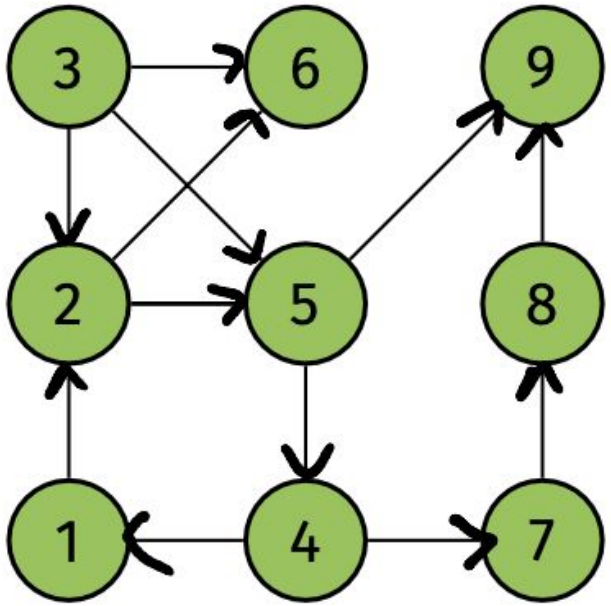
C: E times

D: V times

E: V + E times

Exercise

How many times is each edge "explored" in a BFS of the **directed** graph below?



A: Once

B: Twice

C: E times

D: V times

E: V + E times

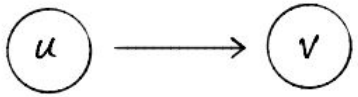
Key Properties of full_bfs

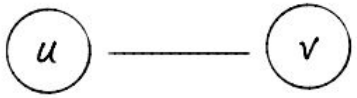
- Each node added to queue **exactly once.**
- Each edge is *explored* **exactly**:
  - **once** if graph is **directed**.
  - **twice** if graph is **undirected**.

Time Complexity of full_bfs

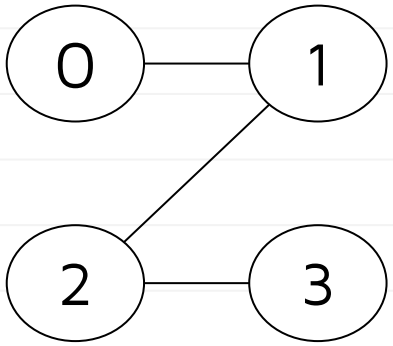
- Analyzing `full_bfs` is easier than analyzing `bfs`.
  - `full_bfs` visits all nodes, no matter the graph.
- Result will be **upper bound** on time complexity of bfs.
- We'll use an **aggregate analysis**.

```python
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```





Exercise 4: Complexity




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

What is the time complexity in terms of $|V|$ and $|E|$?

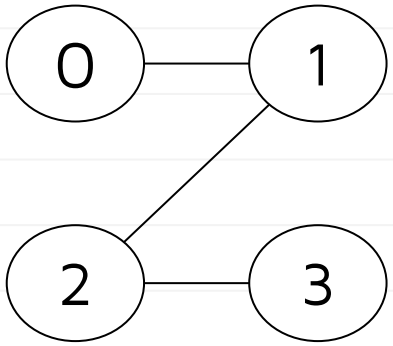
A: **Θ (|E| + |V|)**

C: Θ (|E|)

B: Θ (|E| * |V|)

D: Θ (|V|)

Exercise 4: Complexity



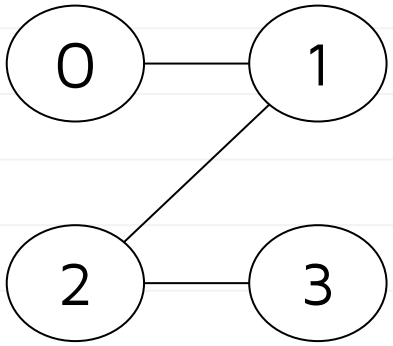

```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

u is 0

v=1

Exercise 4: Complexity



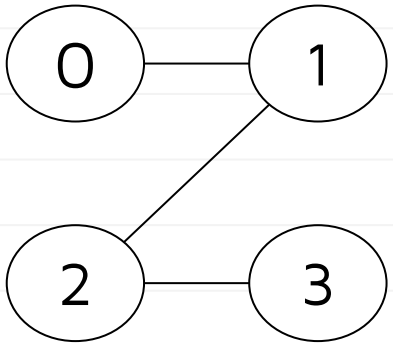

```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

u is 0 u is 1

v=1 v=0, 2

Exercise 4: Complexity

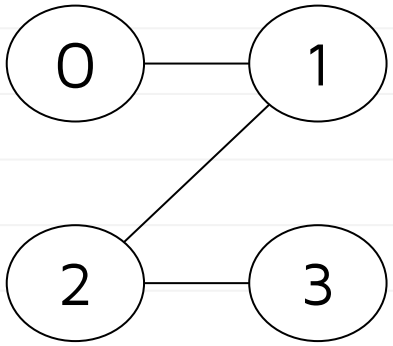



```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

u is 0	u is 1	u is 2
v=1	v=0, 2	v=1, 3

Exercise 4: Complexity

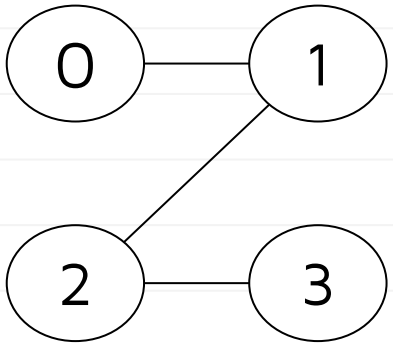



```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

u is 0	u is 1	u is 2	u is 3
v=1	v=0, 2	v=1, 3	v = 2

Exercise 4: Complexity

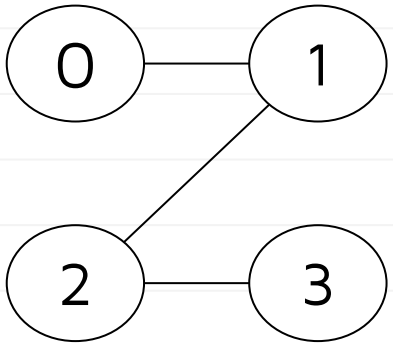



```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

u is 0	u is 1	u is 2	u is 3	
v=1	v=0, 2	v=1, 3	v = 2	Total of 6 iterations

Exercise 4: Complexity




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

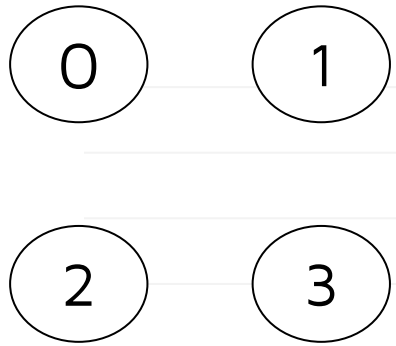
u is 0 u is 1 u is 2 u is 3

v=1 v=0, 2 v=1, 3 v = 2 Total of 6 iterations (E)

Exercise 5:

```
adj = [
    [],
    [],
    [],
    []
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```



u is 0 u is 1 u is 2 u is 3

\- \- \- \- Total of 4 iterations (V)

Thank you!

Do you have any questions?

CampusWire!