PSO 8

Graph

I can’t wait for spring break

Any fun plans

Unfortunately busy week so no slides today :(

Unfortunately busy week so no slides today :(

JUST KIDDING

Question 1

(Adjacency-list Representation)

1. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of a vertex? How long does it take to compute the in-degree of a vertex?

2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

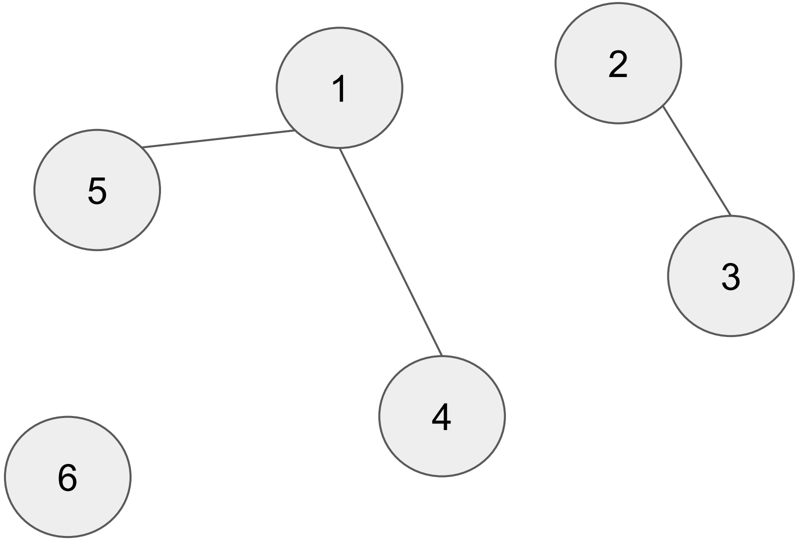
3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

What is an adjacency list?

Adjacency list

A linked list per vertex

E.g. if undirected..

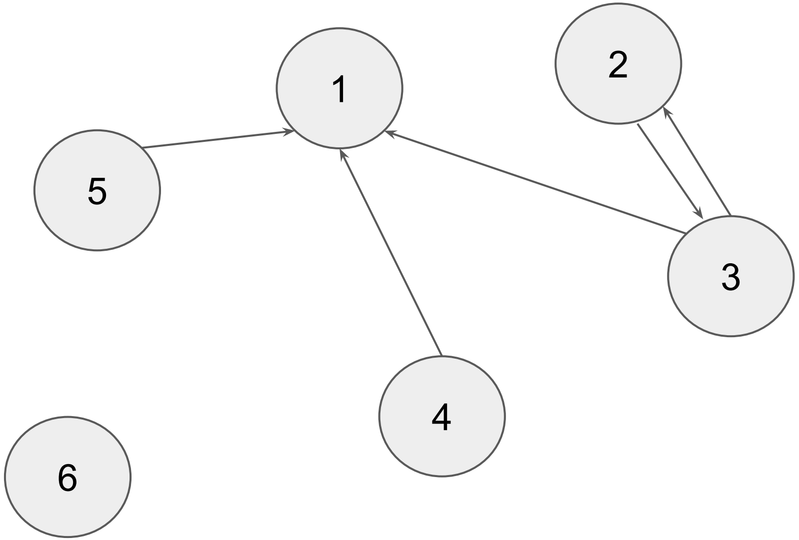


Vertex	Adjacency
1	
2	
3	
4	
5	
6	

Adjacency list

A linked list per vertex

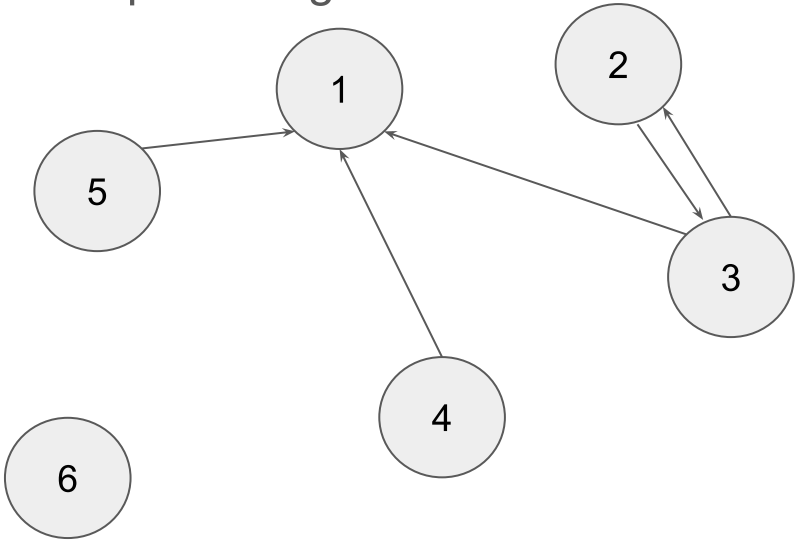
E.g. if **directed**..



Vertex	Adjacency (*points to*)
1	
2	
3	
4	
5	
6	

1. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of a vertex? How long does it take to compute the in-degree of a vertex?

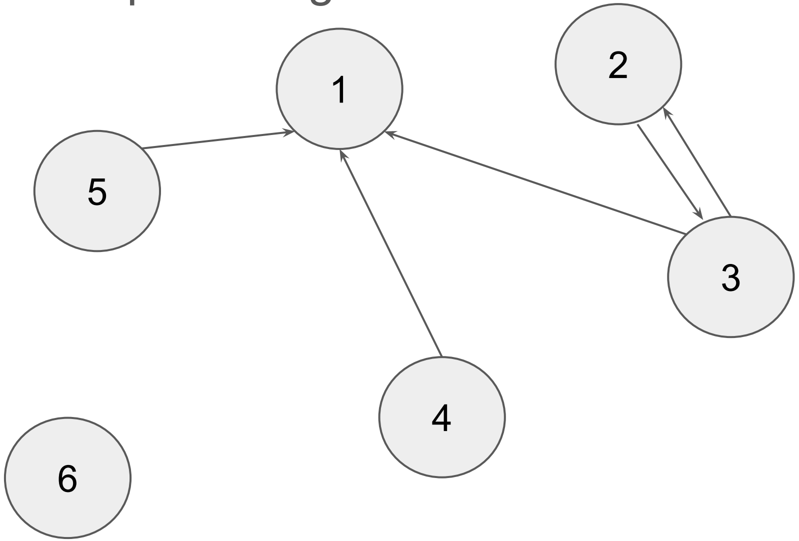
Example: indeg. of 1?



Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

1. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of a vertex? How long does it take to compute the in-degree of a vertex?

Example: indeg. of 1?

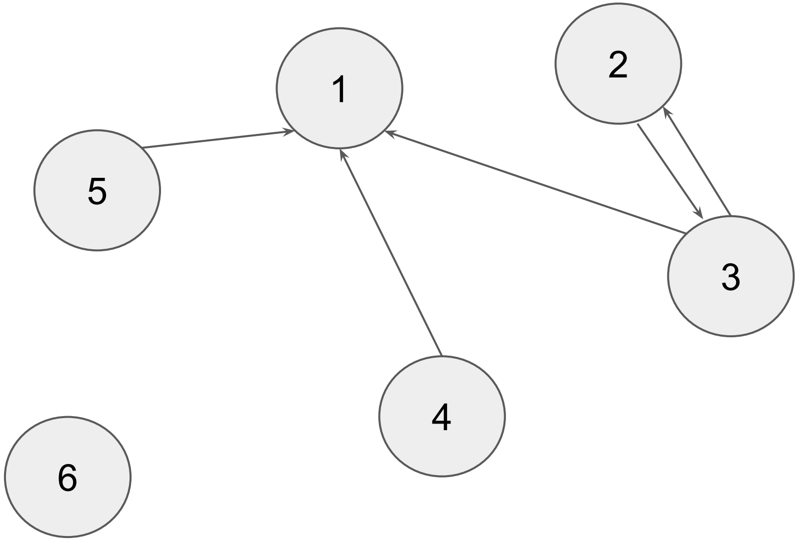


Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

O(|Adjaceny list|)

1. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of a vertex? How long does it take to compute the in-degree of a vertex?

Try counting the indegree for v = 1



Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

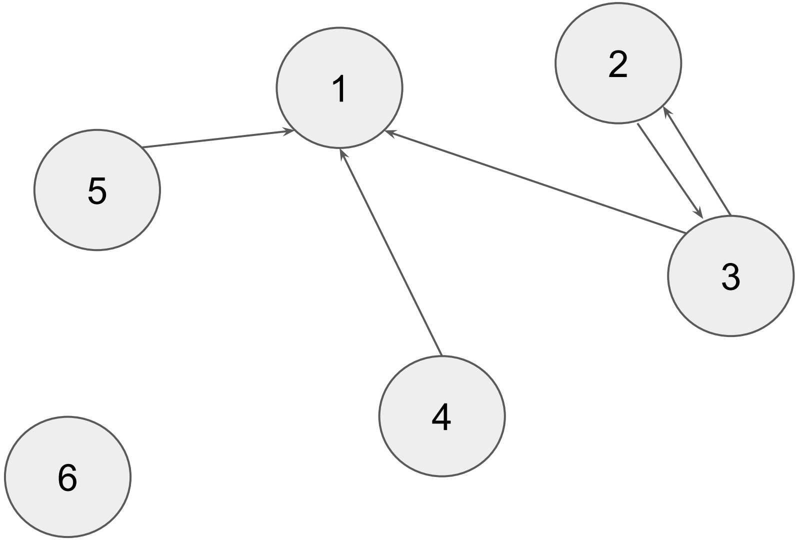
1. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of a vertex? How long does it take to compute the in-degree of a vertex?

For indeg. of vertex i:

Iterate over each vertex list other than i,

Count for every instant of i you see

O(|E|) time



Vertex	Adjacency (*points to*)
1	
2	3
3	2, 1
4	1
5	1
6	

Question 1

(Adjacency-list Representation)

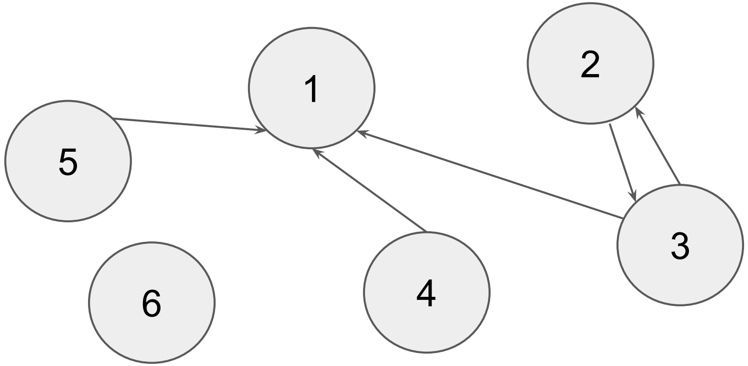
1. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of a vertex? How long does it take to compute the in-degree of a vertex?

2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

Let’s see how this looks..

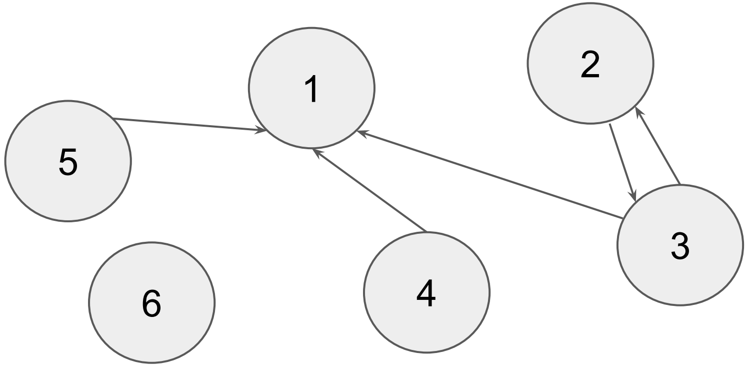
2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

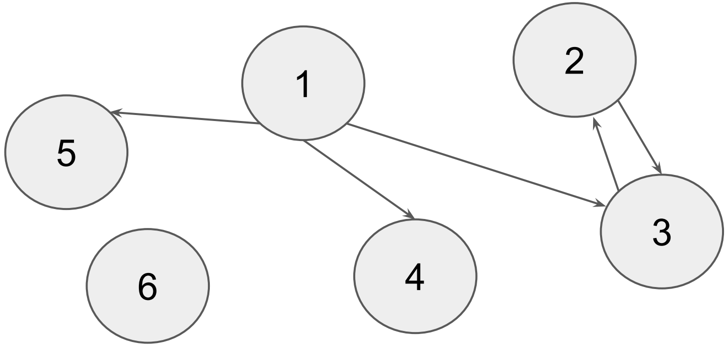
We want to go from this

2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



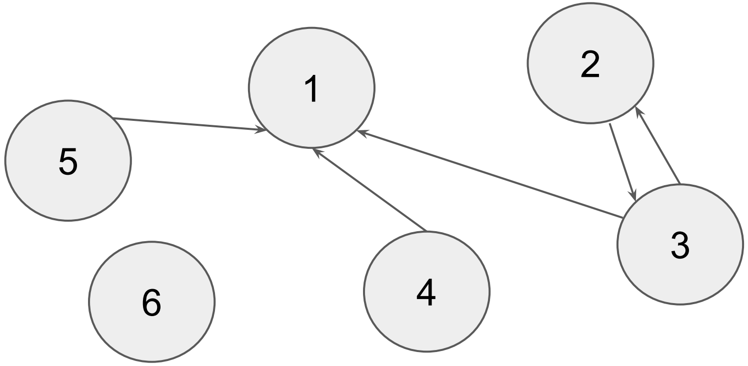
Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

We want to go from this to this



Vertex	Adjacency (*points to*)
1	3,4
2	3
3	2
4	
5	1
6	

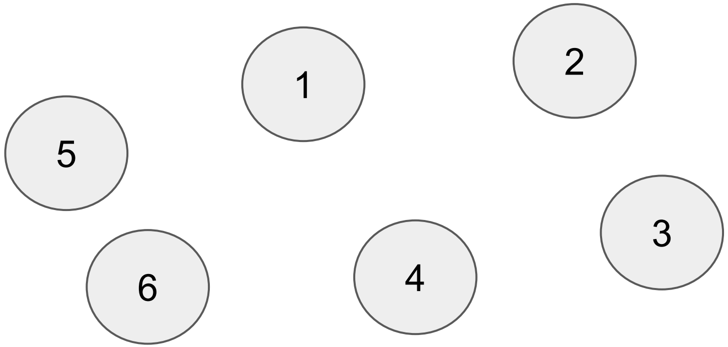
2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

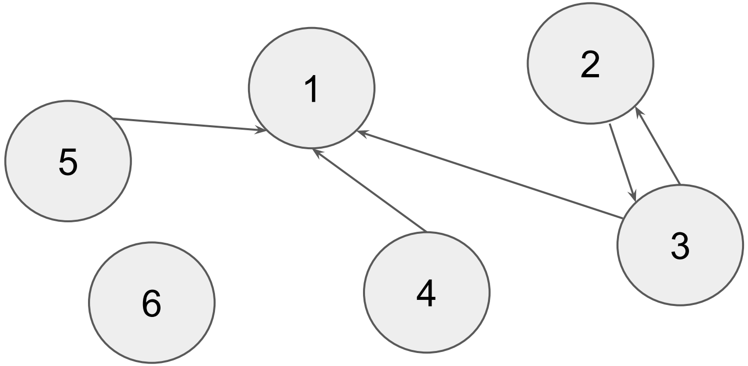
Easiest algorithm

1. Iterate through each vertex list i
2. Add the “reverse” to the new adjacency list



Vertex	Adjacency (*points to*)
1	3,4
2	3
3	2
4	
5	1
6	

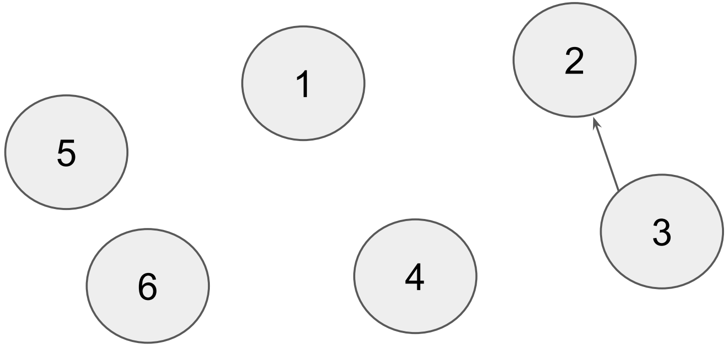
2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

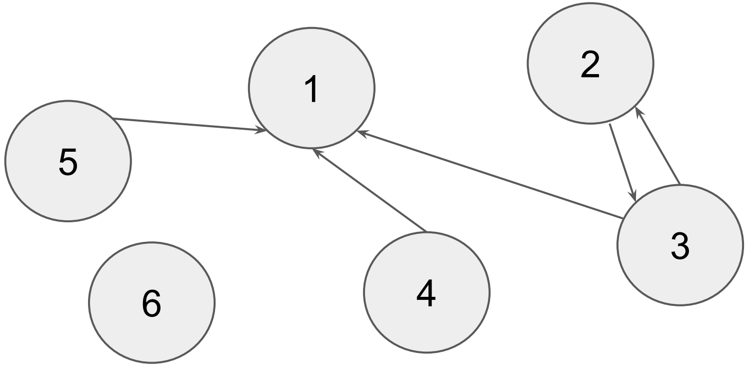
Easiest algorithm

1. Iterate through each vertex list i
2. Add the “reverse” to the new adjacency list



Vertex	Adjacency (*points to*)
1	3,4
2	3
3	2
4	
5	1
6	

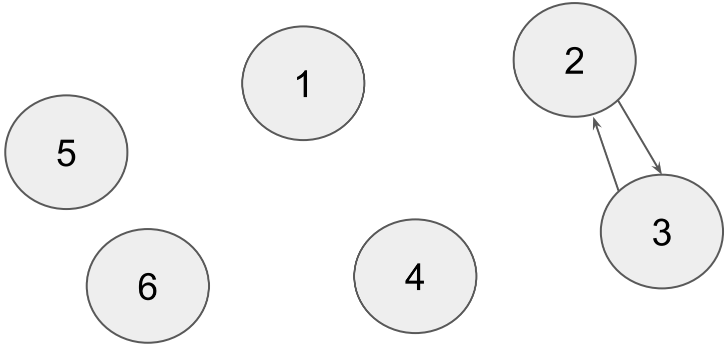
2. The transpose of a directed graph $G = (V, E)$ is the graph $G^{\top} = (V, E^{\top})$, where $E^{\top} := \{(v, u) : (u, v) \in E\}$. In other words, $G^{\top}$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^{\top}$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

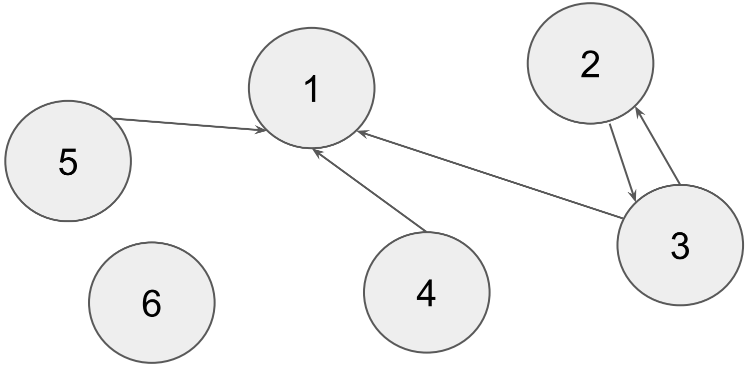
Easiest algorithm

1. Iterate through each vertex list i
2. Add the "reverse" to the new adjacency list



Vertex	Adjacency (*points to*)
1	3,4
2	3
3	2
4	
5	1
6	

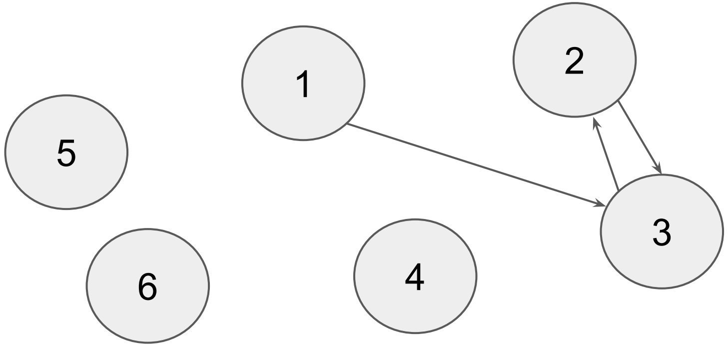
2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

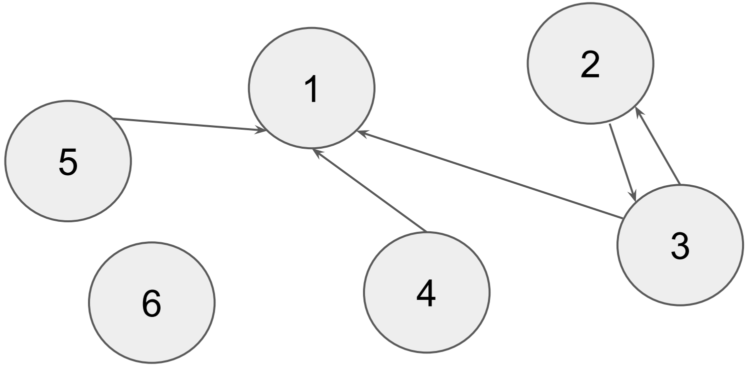
Easiest algorithm

1. Iterate through each vertex list i
2. Add the "reverse" to the new adjacency list



Vertex	Adjacency (*points to*)
1	3,4
2	3
3	2
4	
5	1
6	

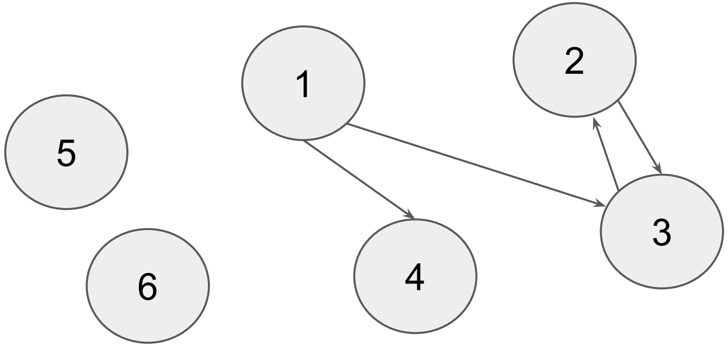
2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

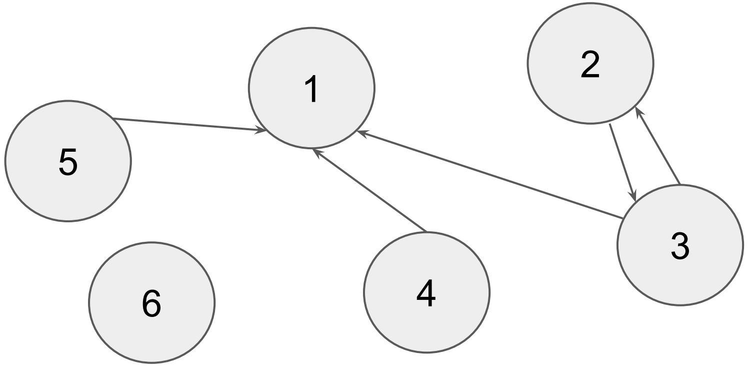
Easiest algorithm

1. Iterate through each vertex list i
2. Add the "reverse" to the new adjacency list



Vertex	Adjacency (*points to*)
1	3,4
2	3
3	2
4	
5	1
6	

2. The transpose of a directed graph $G = (V, E)$ is the graph $G^{\top} = (V, E^{\top})$, where $E^{\top} := \{(v, u) : (u, v) \in E\}$. In other words, $G^{\top}$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^{\top}$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

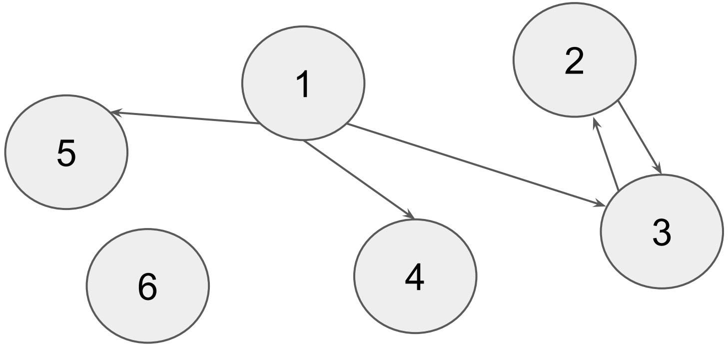


Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

Easiest algorithm

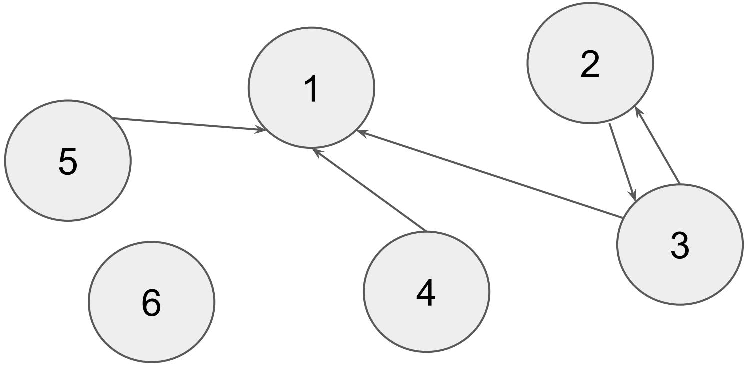
1. Iterate through each vertex list i
2. Add the "reverse" to the new adjacency list

Runtime?



Vertex	Adjacency (*points to*)
1	3,4
2	3
3	
4	
5	1
6	

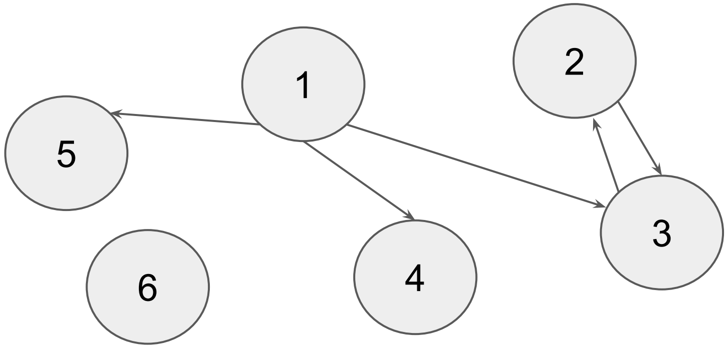
2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



Vertex	Adjacency (*points to*)
1	
2	3
3	2,1
4	1
5	1
6	

Easiest algorithm
1. Iterate through each vertex list i
2. Add the "reverse" to the new adjacency list

Runtime? O(|V| + |E|)



Vertex	Adjacency (*points to*)
1	3,4
2	3
3	
4	
5	1
6	

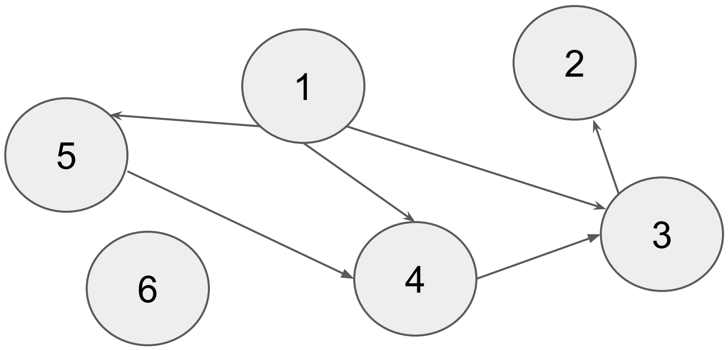
Question 1

(Adjacency-list Representation)

1. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of a vertex? How long does it take to compute the in-degree of a vertex?

2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



Square of this graph?

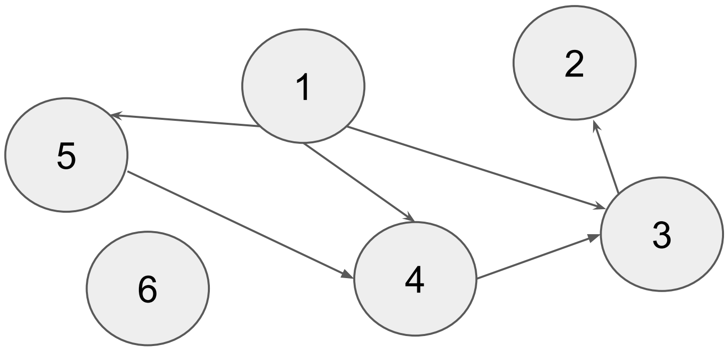
Question 1

(Adjacency-list Representation)

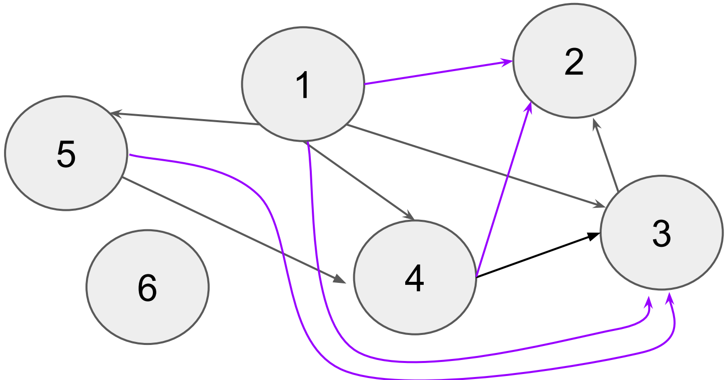
1. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of a vertex? How long does it take to compute the in-degree of a vertex?

2. The transpose of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top := \{(v, u) : (u, v) \in E\}$. In other words, $G^\top$ is G with all its edges reversed. Describe an efficient algorithm for computing $G^\top$ from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

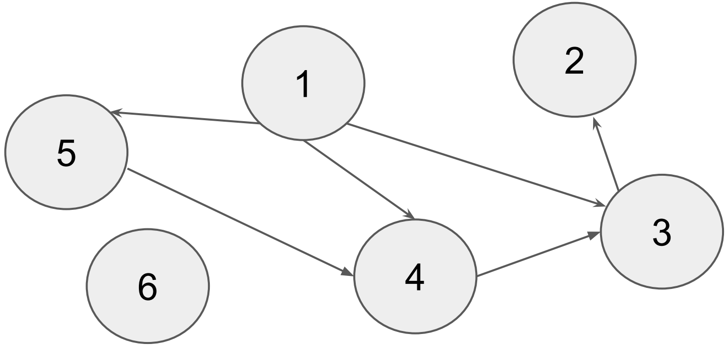


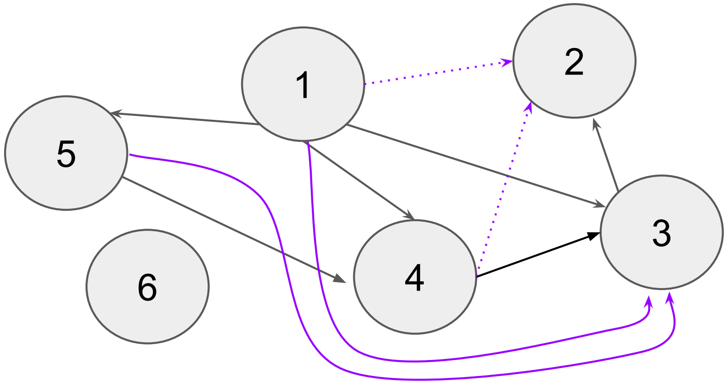
Square of this graph?



3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

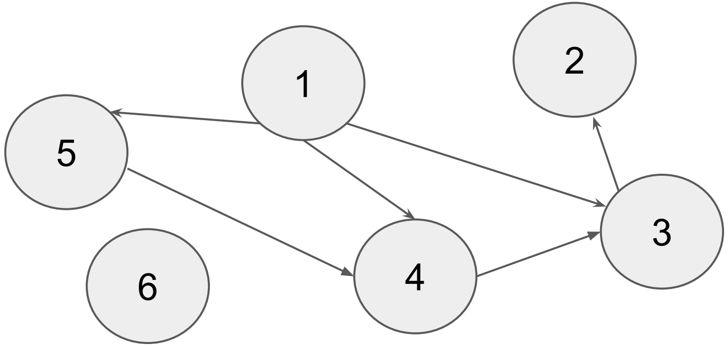
How do we get 3's edges (ignore every else for now)

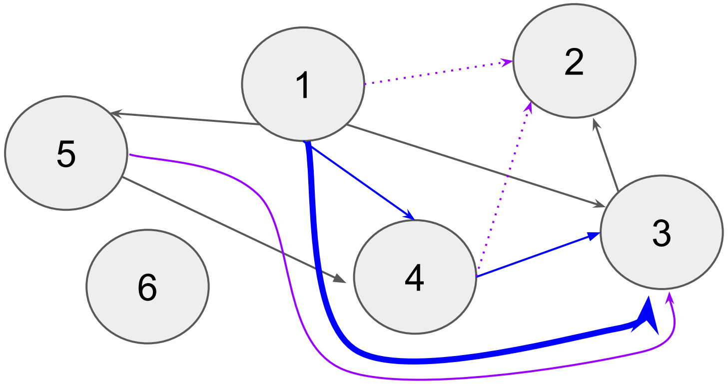




3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

How do we get 3's edges (ignore every else for now)

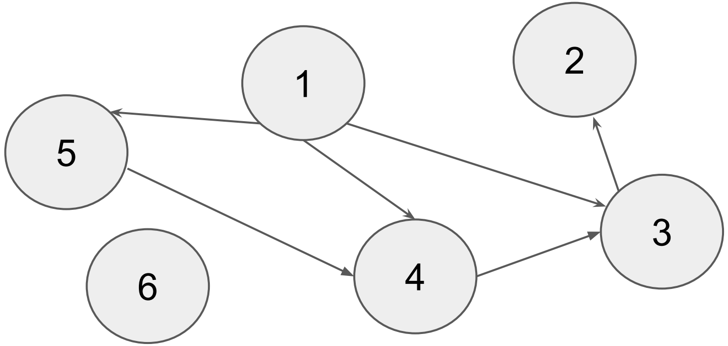


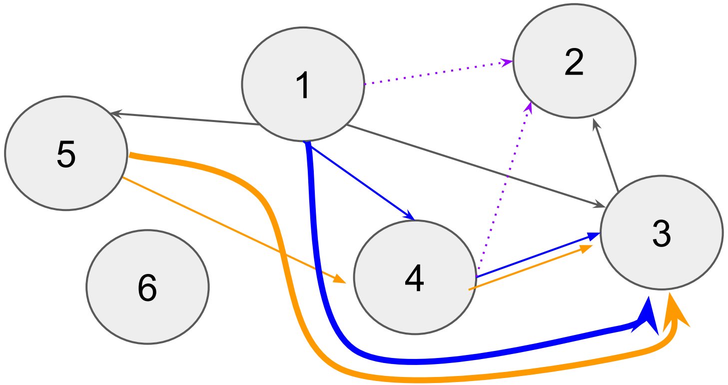


edge comes from ___

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

How do we get 3's edges (ignore every else for now)





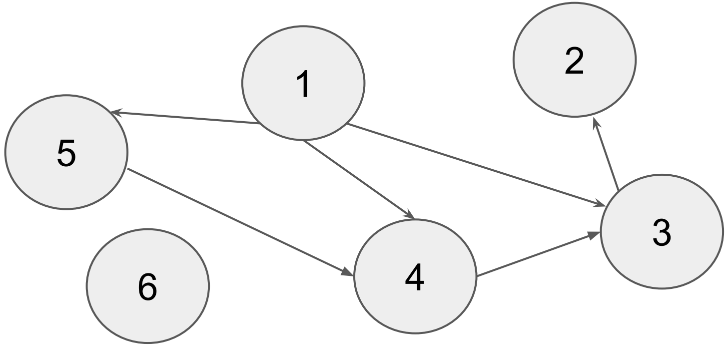
edge comes from _

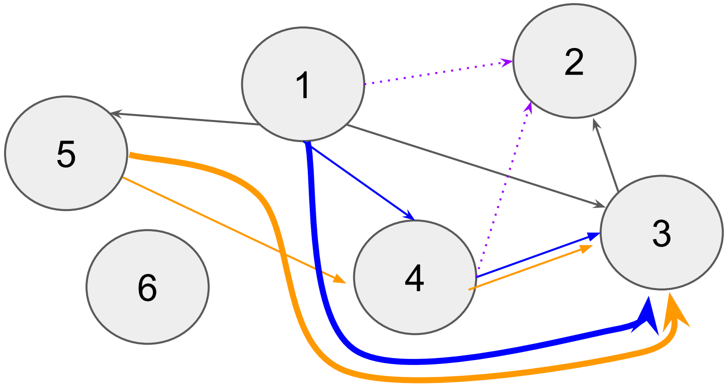
edge comes from _

They both share a (3,4) edge

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

How do we get 3's edges (ignore every else for now)



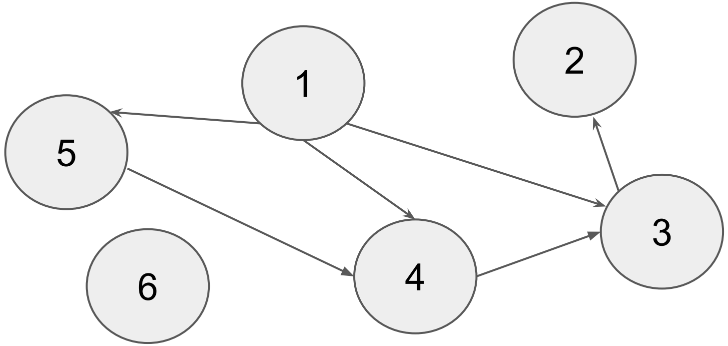


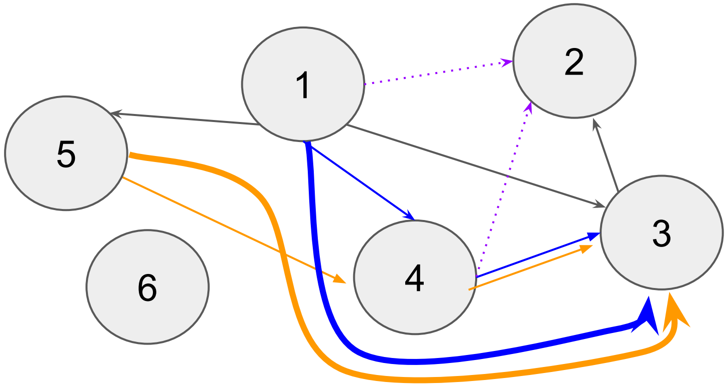
They both share a (3,4) edge

- **Idea**: when adding edge (3,4),
  Add all edges pointing to 3

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

How do we get 3's edges (ignore every else for now)





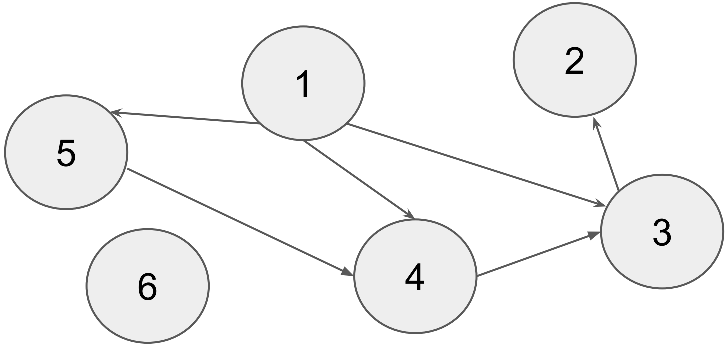
They both share a (3,4) edge

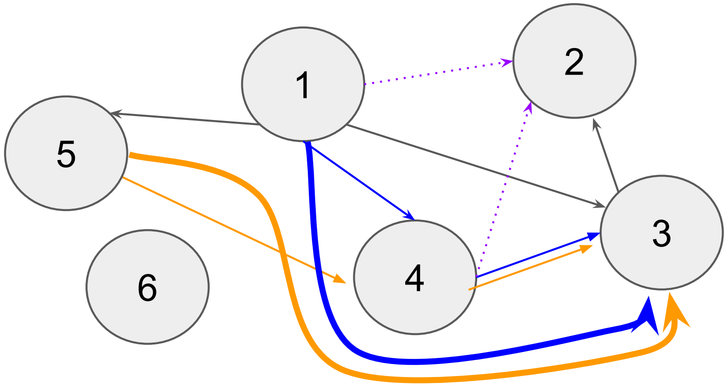
- **Idea**: when adding edge (i,j),

  Add all edges pointing to i (how do we get this?)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.

How do we get 3's edges (ignore every else for now)





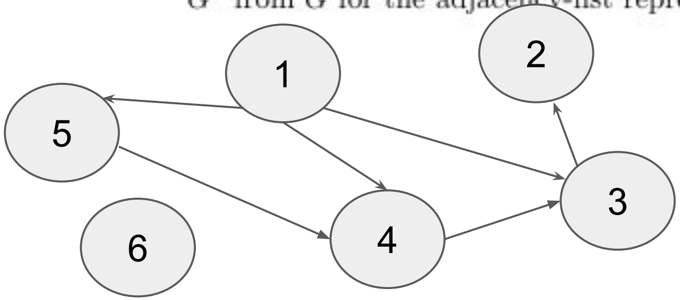
They both share a (3,4) edge

- **Idea**: when adding edge (i,j),

  Add all edges pointing to i to j (how do we get this?)

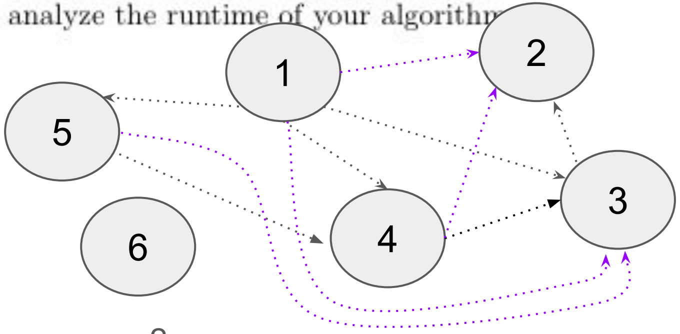
  G^T.adjList(i) is exactly this!

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	**3**,4,5
2	
3	2
4	3
5	4
6	

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

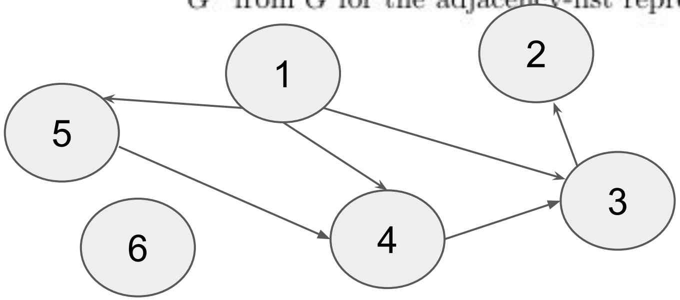
Vertex	Adj
1	
2	
3	
4	
5	
6	

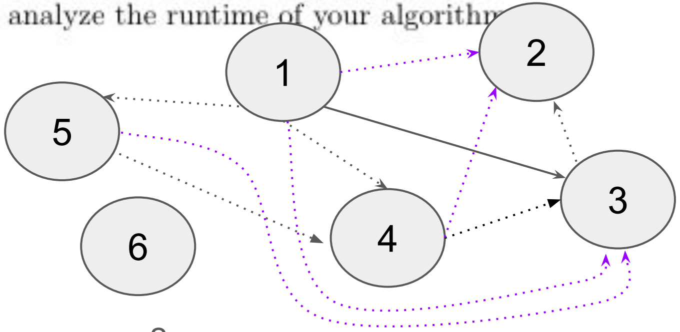
Idea: when adding edge (i,j), Add all edges pointing to i to j

For i in |V|:

For **j** in G.adjList(i):

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G $\quad$ G^T $\quad$ G^2

Vertex	Adj
1	**3**,4,5
2	
3	2
4	3
5	4
6	

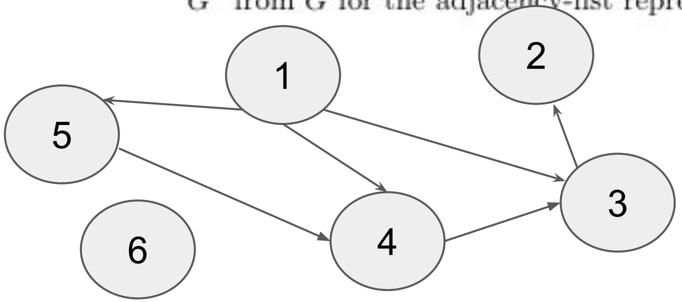
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

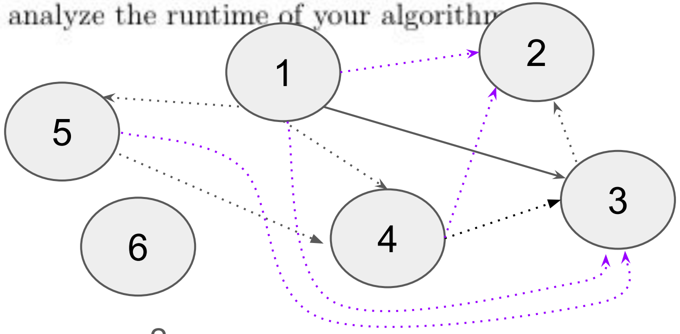
Vertex	Adj
1	3
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), Add all edges pointing to i to j

```
For i in |V|:
        For j in G.adjList(i):
                add j to G^2.adjList(i)
```

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G

G^T

G^2

Vertex	Adj
1	**3**,4,5
2	
3	2
4	3
5	4
6	

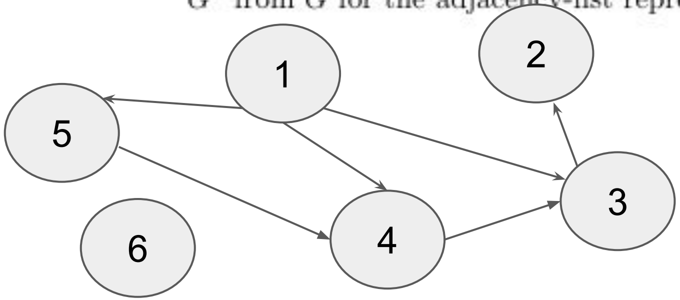
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

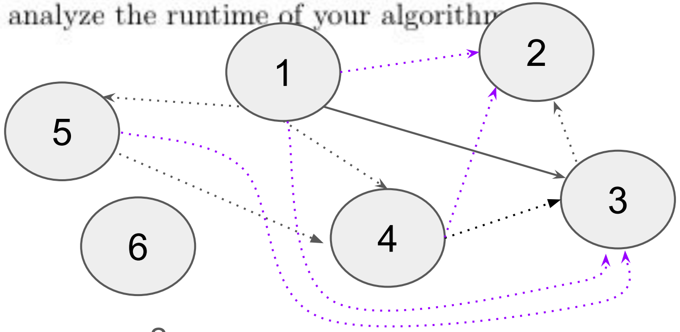
Vertex	Adj
1	3
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

```
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
```

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G

G^T

G^2

Vertex	Adj
1	**3**,4,5
2	
3	2
4	3
5	4
6	

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

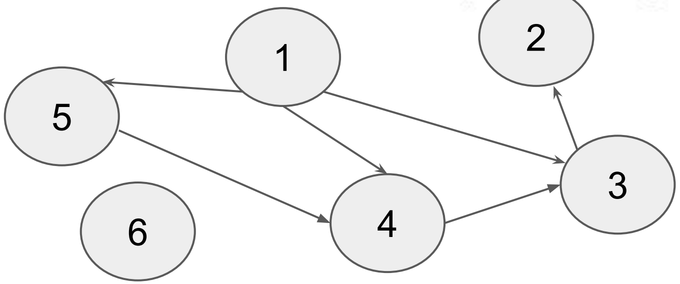
Vertex	Adj
1	3
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

For i in |V|:
    For **j** in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
              G^2.adjList(k)

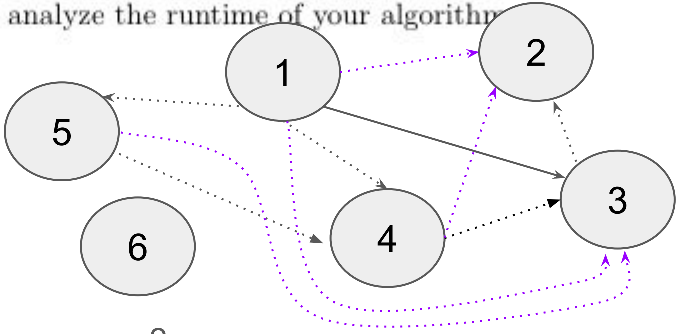
Bad example lol
Lets continue

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,**4**,5
2	
3	2
4	3
5	4
6	

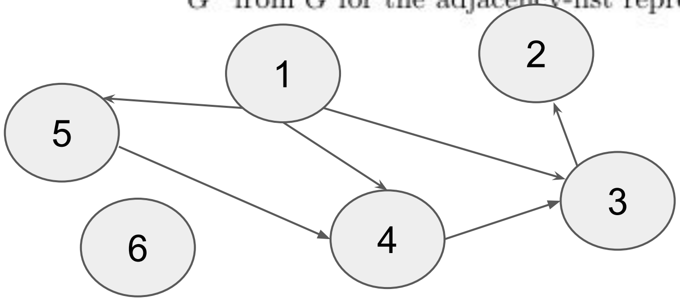
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

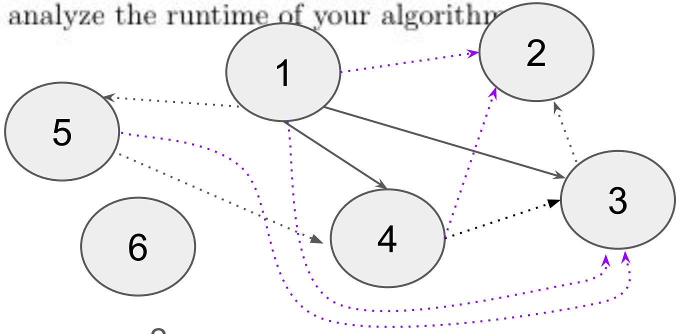
Vertex	Adj
1	3
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to G^2.adjList(k)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G $\qquad$ G^T $\qquad$ G^2

Vertex	Adj
1	3,**4**,5
2	
3	2
4	3
5	4
6	

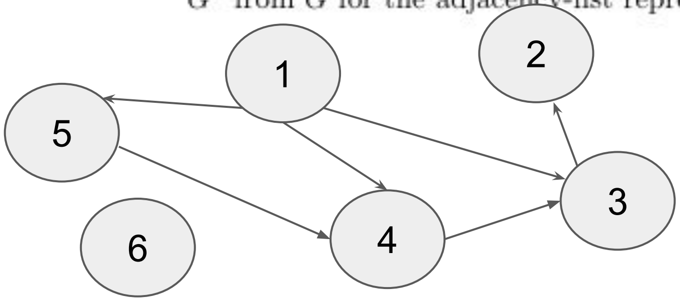
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

Vertex	Adj
1	3,4
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

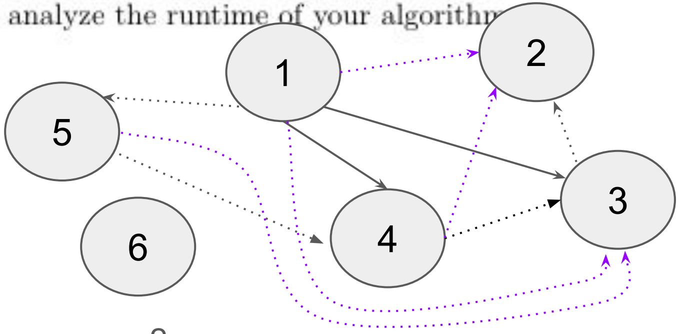
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to G^2.adjList(k)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,**4**,5
2	
3	2
4	3
5	4
6	

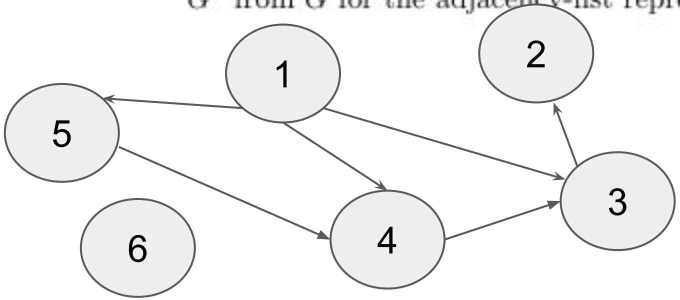
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

Vertex	Adj
1	3,4
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

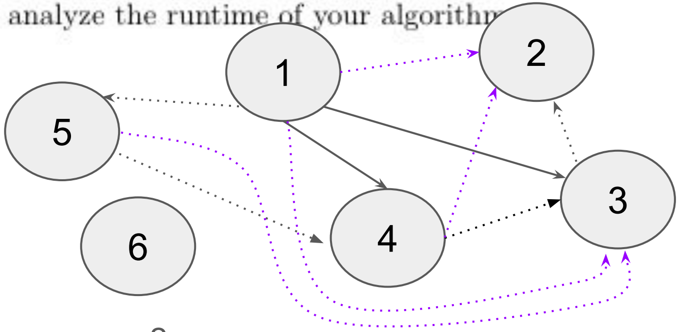
```
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
             G^2.adjList(k)
```

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,4,**5**
2	
3	2
4	3
5	4
6	

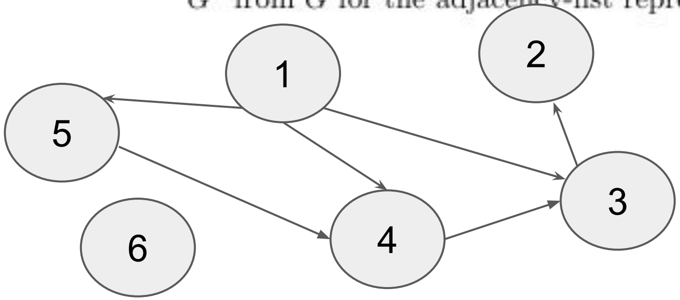
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

Vertex	Adj
1	3,4
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

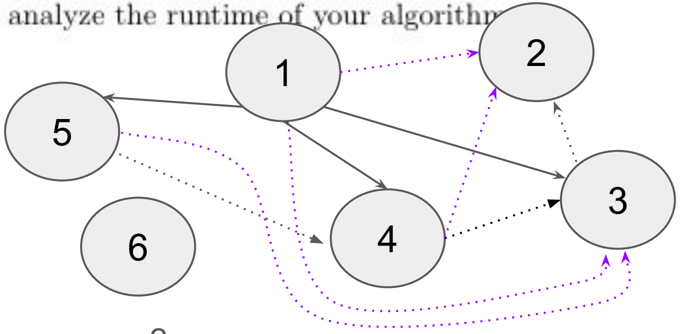
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
                G^2.adjList(k)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,4,**5**
2	
3	2
4	3
5	4
6	

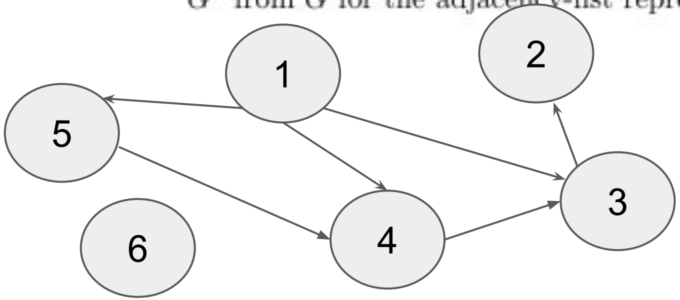
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

Vertex	Adj
1	3,4,5
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

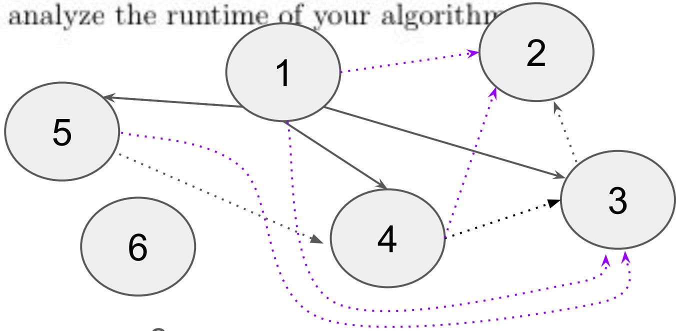
```
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
              G^2.adjList(k)
```

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,4,**5**
2	
3	2
4	3
5	4
6	

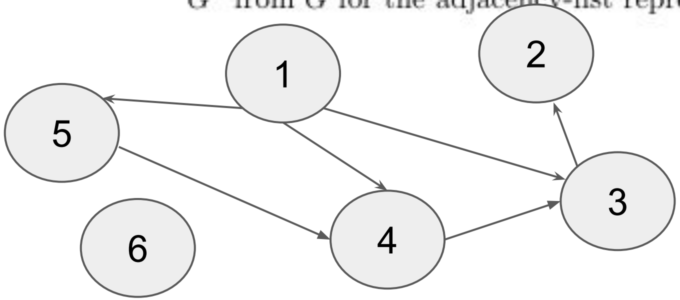
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

Vertex	Adj
1	3,4,5
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

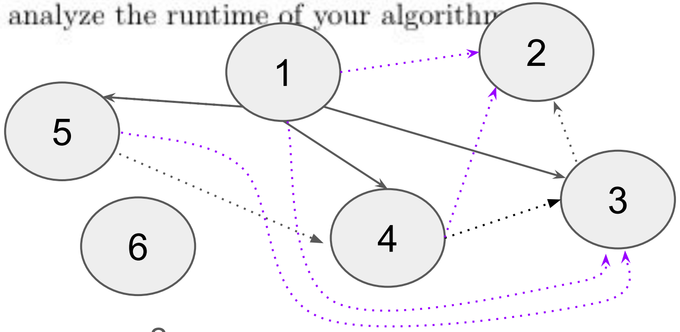
```
For i in |V|:
    For j in G.adjList(i):
        add j to G².adjList(i)
        for k in Gᵀ.adjList(i):
            add j to
             G².adjList(k)
```

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,4,5
2	
3	**2**
4	3
5	4
6	

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

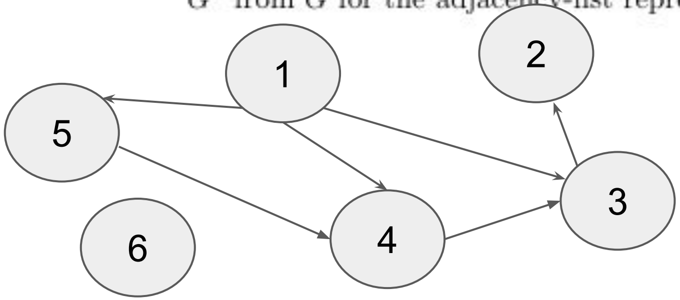
Vertex	Adj
1	3,4,5
2	
3	
4	
5	
6	

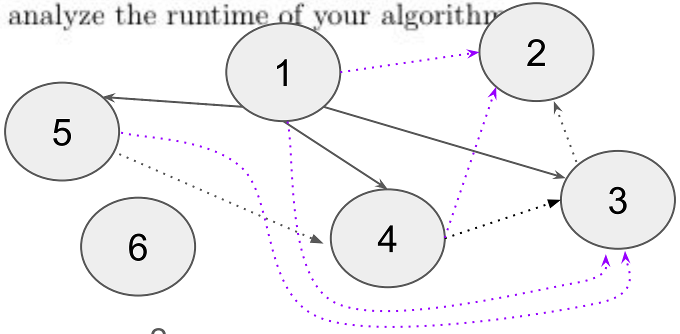
Idea: when adding edge (i,j), **Add all edges pointing to i to j**

For i in |V|:

```
For j in G.adjList(i):
    add j to G^2.adjList(i)
    for k in G^T.adjList(i):
        add j to
          G^2.adjList(k)
```

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G

Vertex	Adj
1	3,4,5
2	
3	**2**
4	3
5	4
6	

G^T

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

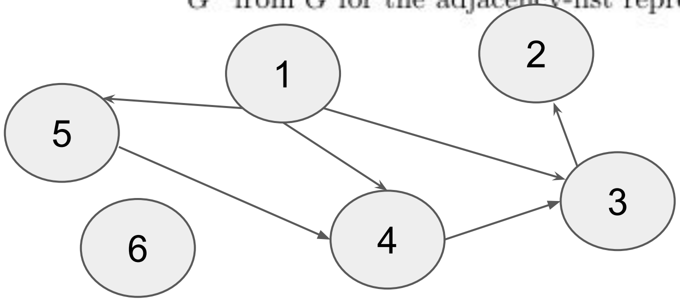
G^2

Vertex	Adj
1	3,4,5
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

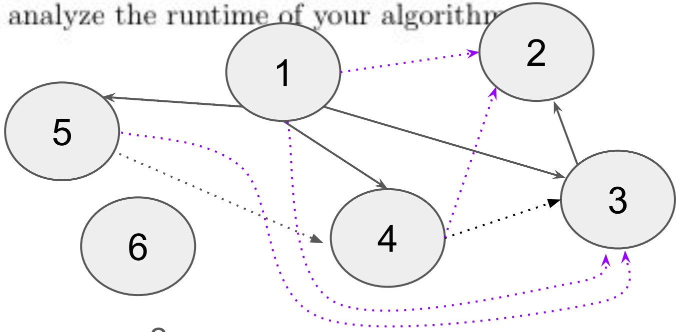
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
              G^2.adjList(k)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,4,5
2	
3	**2**
4	3
5	4
6	

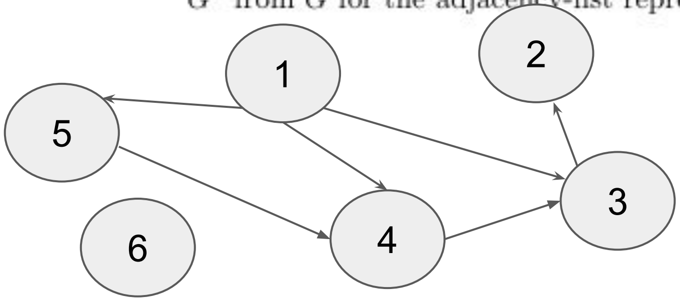
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

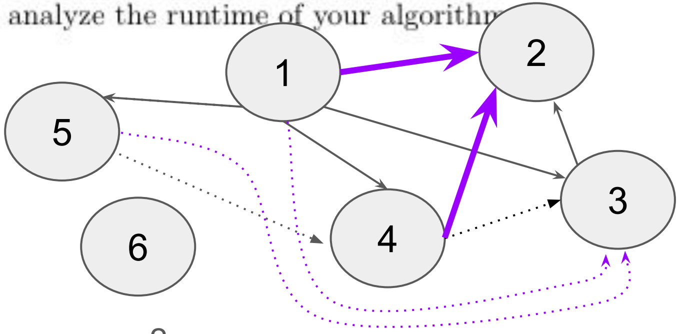
Vertex	Adj
1	3,4,5
2	
3	
4	
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to G^2.adjList(k)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G $\qquad$ G^T $\qquad$ G^2

G

Vertex	Adj
1	3,4,5
2	
3	**2**
4	3
5	4
6	

G^T

Vertex	Adj
1	
2	3
3	**1 4**
4	1 5
5	1
6	

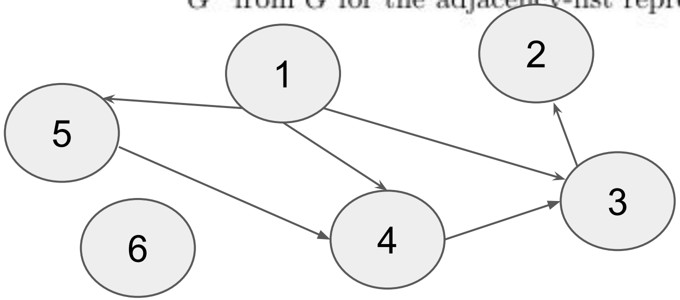
G^2

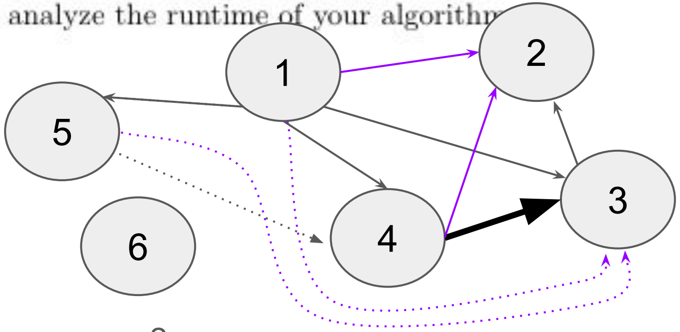
Vertex	Adj
1	3,4,5,**2**
2	
3	
4	**2**
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

```
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
              G^2.adjList(k)
```

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G

Vertex	Adj
1	3,4,5
2	
3	2
4	**3**
5	4
6	

G^T

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

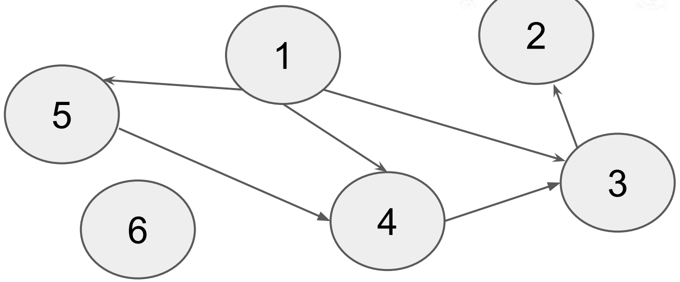
G^2

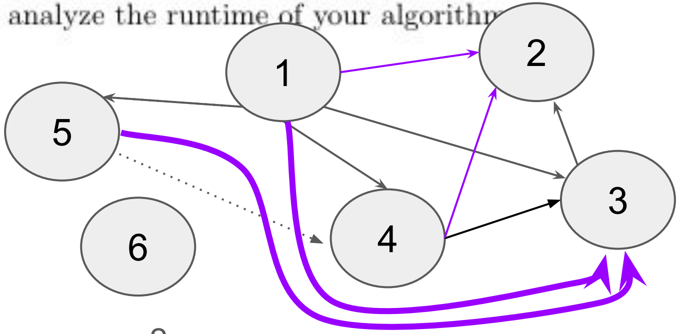
Vertex	Adj
1	3,4,5,2
2	
3	
4	2,3
5	
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to G^2.adjList(k)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G G^T G^2

G

Vertex	Adj
1	3,4,5
2	
3	2
4	**3**
5	4
6	

G^T

Vertex	Adj
1	
2	3
3	1 4
4	**1 5**
5	1
6	

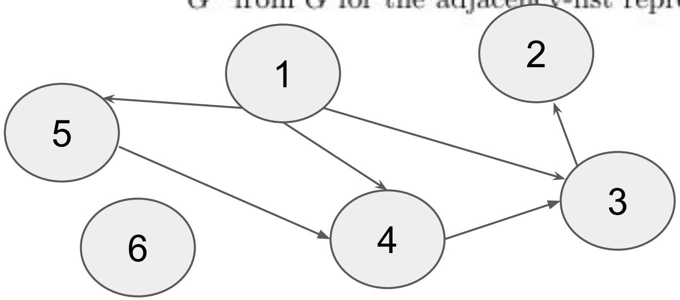
G^2

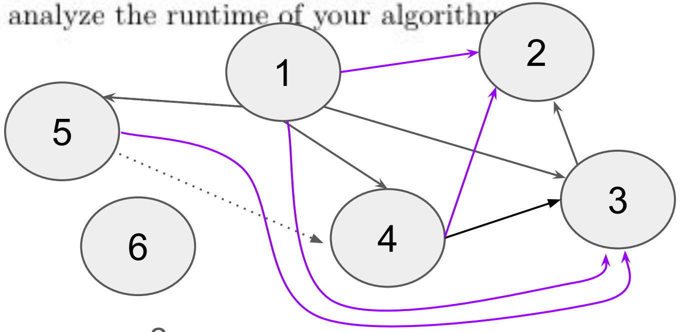
Vertex	Adj
1	3,4,5,2,**3**
2	
3	
4	2,3
5	**3**
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

```
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
              G^2.adjList(k)
```

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm





G

G^T

G^2

Vertex	Adj
1	3,4,5
2	
3	2
4	3
5	**4**
6	

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

Vertex	Adj
1	3,4,5,2,3
2	
3	
4	2,3
5	3
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

For i in |V|:

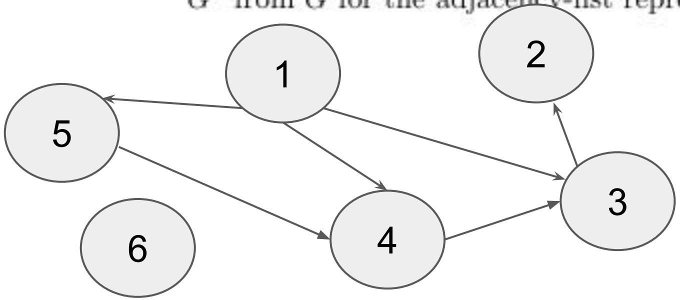
For j in G.adjList(i):

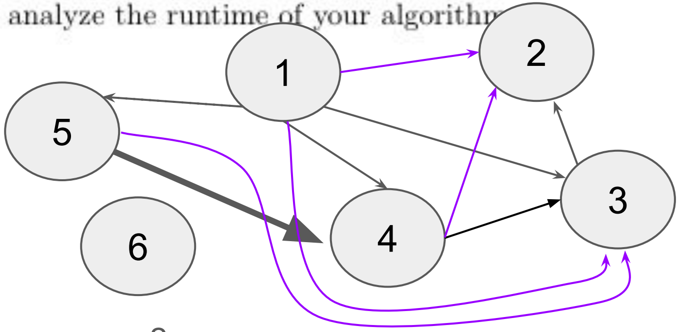
add j to G^2.adjList(i)

for k in G^T.adjList(i):

add j to G^2.adjList(k)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G

Vertex	Adj
1	3,4,5
2	
3	2
4	3
5	**4**
6	

G^T

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

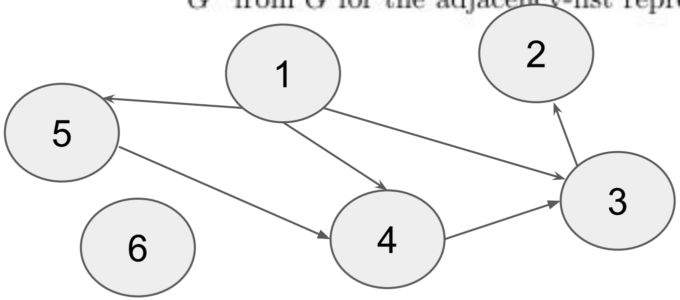
G^2

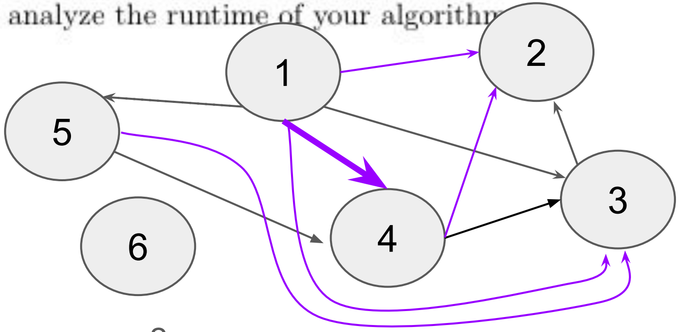
Vertex	Adj
1	3,4,5,2,3
2	
3	
4	2,3
5	3,4
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

For i in |V|:
For j in G.adjList(i):
add j to G^2.adjList(i)
for k in G^T.adjList(i):
add j to G^2.adjList(k)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G

Vertex	Adj
1	3,4,5
2	
3	2
4	3
5	**4**
6	

G^T

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	**1**
6	

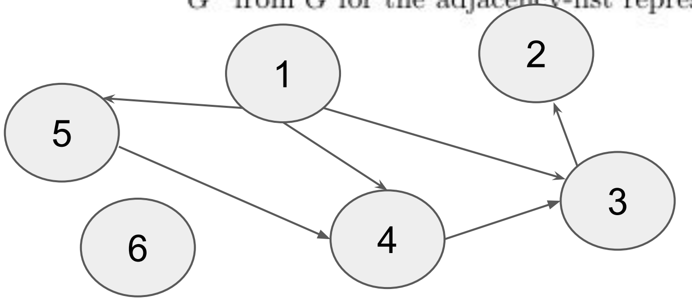
G^2

Vertex	Adj
1	3,4,5,2,3, 4
2	
3	
4	2,3
5	3,4
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

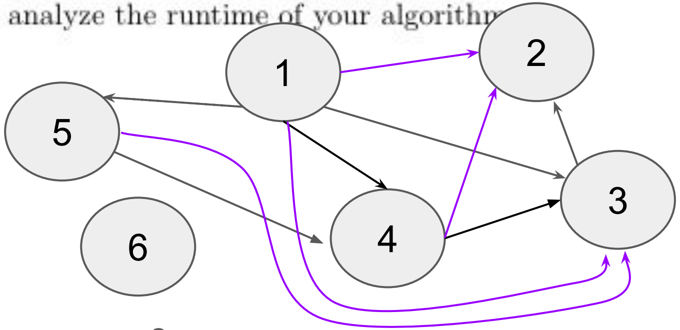
```
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
              G^2.adjList(k)
```

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,4,5
2	
3	2
4	3
5	4
6	

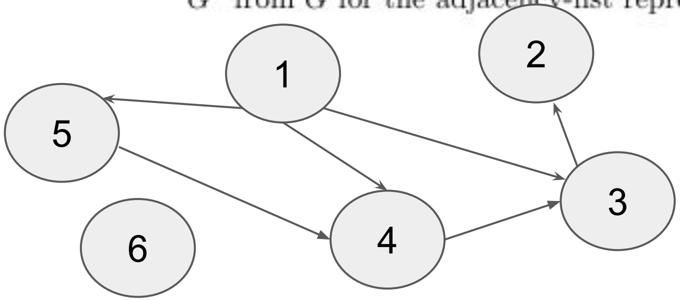
Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

Vertex	Adj
1	3,4,5,2,3, 4
2	
3	
4	2,3
5	3,4
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

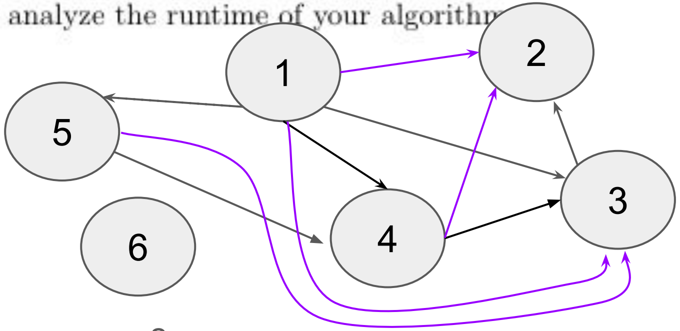
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to G^2.adjList(k)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,4,5
2	
3	2
4	3
5	4
6	

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

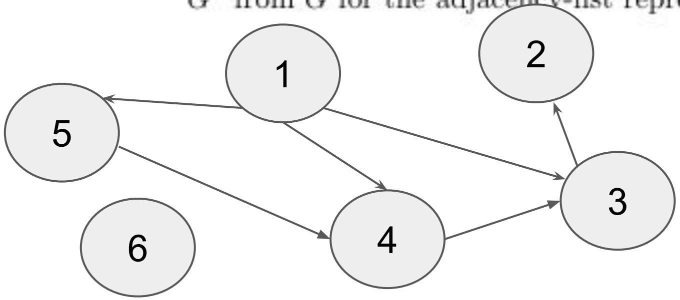
Vertex	Adj
1	3,4,5,2,3, 4
2	
3	
4	2,3
5	3,4
6	

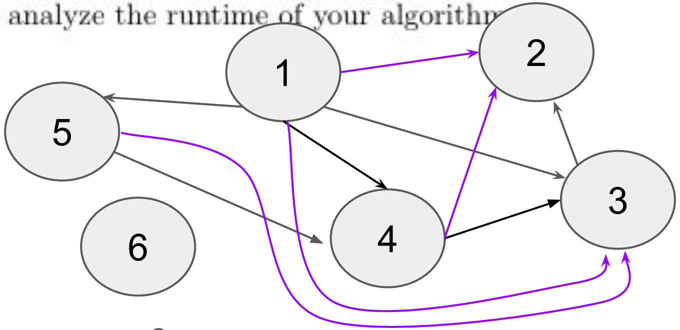
Idea: when adding edge (i,j), **Add all edges pointing to i to j**

For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to G^2.adjList(k)

Time complexity?

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G

Vertex	Adj
1	3,4,5
2	
3	2
4	3
5	4
6	

G^T

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

G^2

Vertex	Adj
1	3,4,5,2,3,4
2	
3	
4	2,3
5	3,4
6	

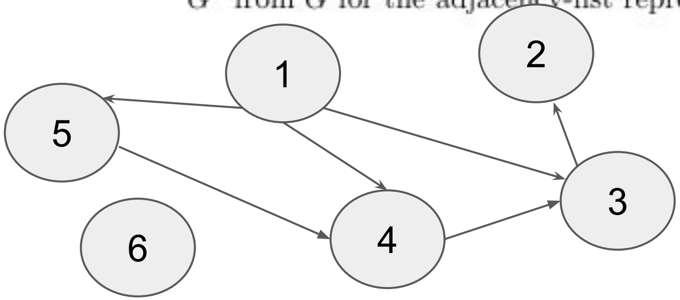
Idea: when adding edge (i,j), **Add all edges pointing to i to j**

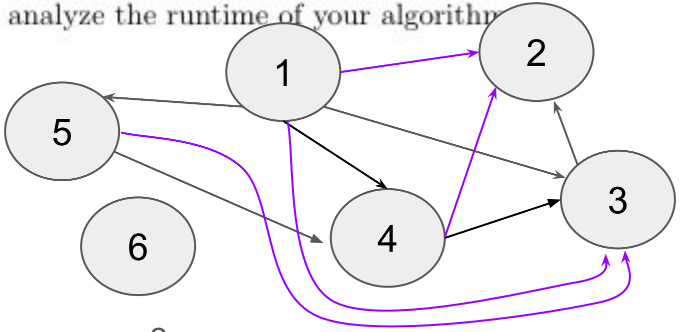
```
For i in |V|:
        For j in G.adjList(i):
                add j to G^2.adjList(i)
                for k in G^T.adjList(i):
                        add j to
                          G^2.adjList(k)
```

Time complexity?

Time to process each edge in G = Look through an adj list <= |V|

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.





G

Vertex	Adj
1	3,4,5
2	
3	2
4	3
5	4
6	

G^T

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

G^2

Vertex	Adj
1	3,4,5,2,3, 4
2	
3	
4	2,3
5	3,4
6	

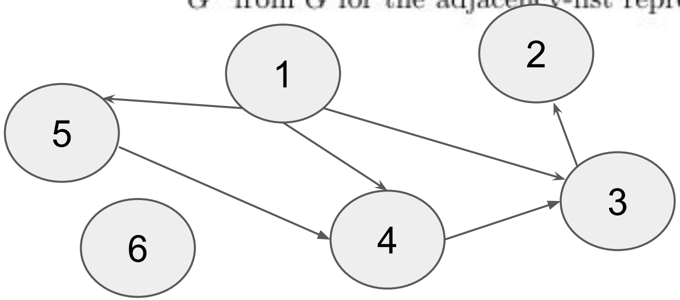
Idea: when adding edge (i,j), **Add all edges pointing to i to j**

```
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
              G^2.adjList(k)
```

Time complexity?

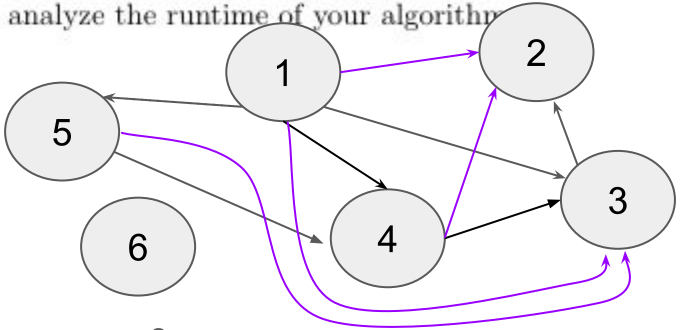
Time to process each edge in G = O(|V|)

3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representations of G and analyze the runtime of your algorithm.



G

G^T



G^2

Vertex	Adj
1	3,4,5
2	
3	2
4	3
5	4
6	

Vertex	Adj
1	
2	3
3	1 4
4	1 5
5	1
6	

Vertex	Adj
1	3,4,5,2,3, 4
2	
3	
4	2,3
5	3,4
6	

Idea: when adding edge (i,j), **Add all edges pointing to i to j**

```
For i in |V|:
    For j in G.adjList(i):
        add j to G^2.adjList(i)
        for k in G^T.adjList(i):
            add j to
             G^2.adjList(k)
```

Time complexity?

O(|E||V|)

Question 2

(Adjacency-matrix Representation)

1. Give an adjacency-matrix representation for a complete binary search tree on 7 vertices numbered from 1 to 7.

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.

What in the world in an adjacency matrix?

Question 2

(Adjacency-matrix Representation)

1. Give an adjacency-matrix representation for a complete binary search tree on 7 vertices numbered from 1 to 7.

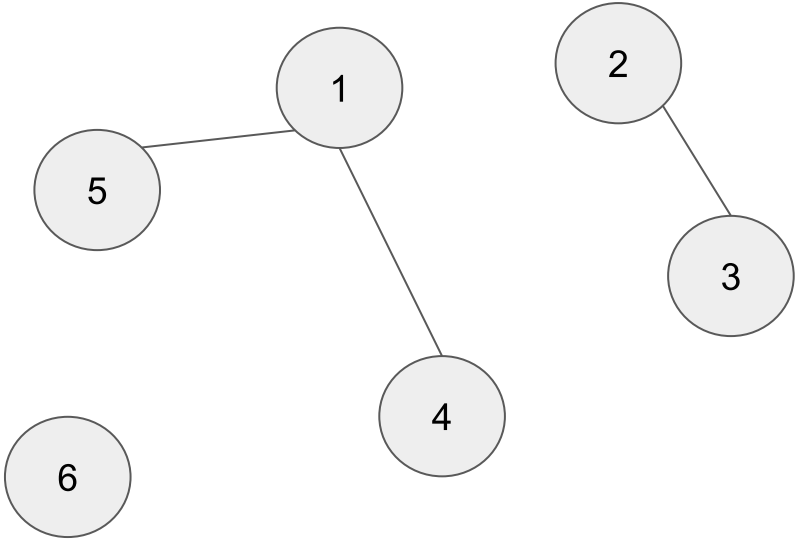
2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.

What in the world in an adjacency matrix?

Adjacency Matrix

Edges represented in a |V| x |V| matrix

E.g. if undirected..



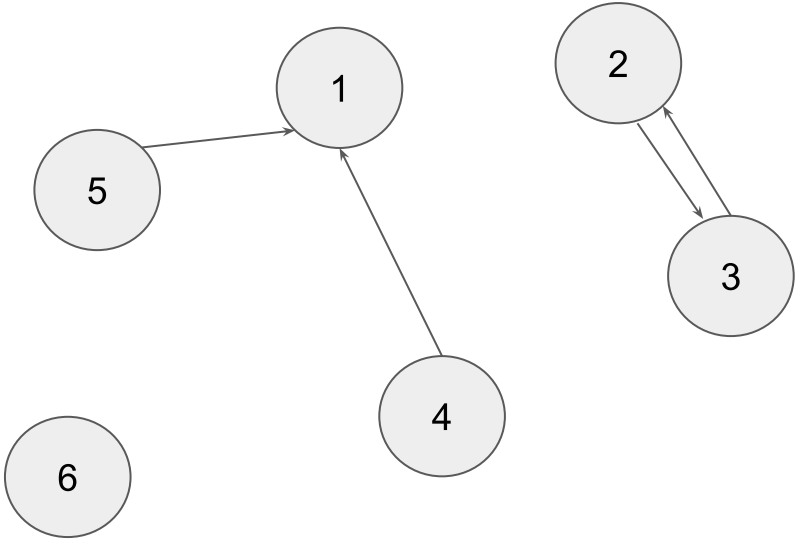
$$
\begin{array}{cc}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{pmatrix} \; & \; & \; & \; & \; & \; \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix}
\end{array}
$$

Adjacency Matrix

"Row goes to column"

Edges represented in a |V| x |V| matrix

E.g. if **directed**..



1 2 3 4 5 6

1
2
3
4
5
6

Question 2

(Adjacency-matrix Representation)

1. Give an adjacency-matrix representation for a complete binary search tree on 7 vertices numbered from 1 to 7.

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.

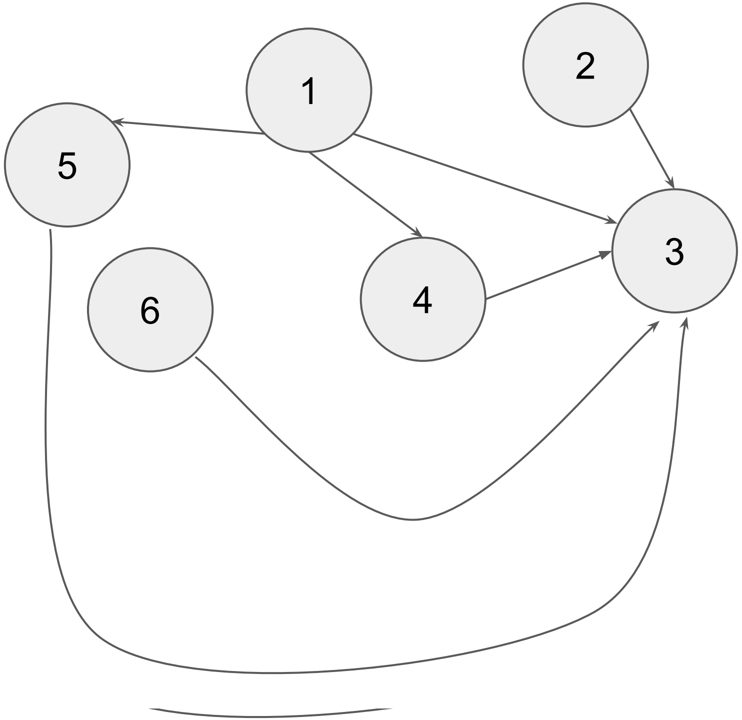
Someone give me a complete binary search tree

Question 2

(Adjacency-matrix Representation)

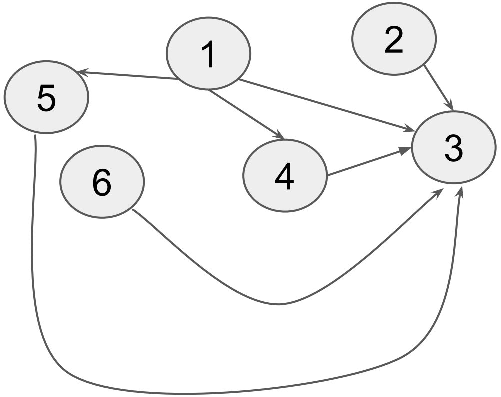
1. Give an adjacency-matrix representation for a complete binary search tree on 7 vertices numbered from 1 to 7.

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.



$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 0 & 1 & 1 & 1 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 0 & 0 \\
5 & 0 & 0 & 1 & 0 & 0 & 0 \\
6 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
$$

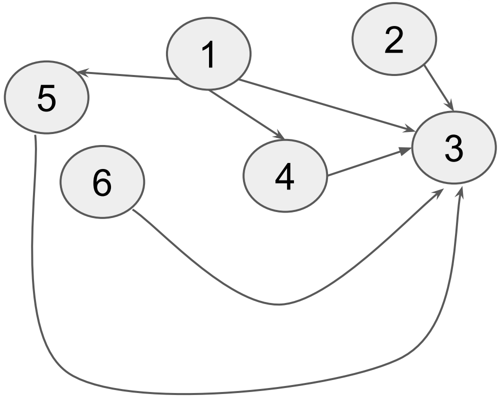
2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.



What do I notice about the adjacency matrix (specifically column 3)

	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

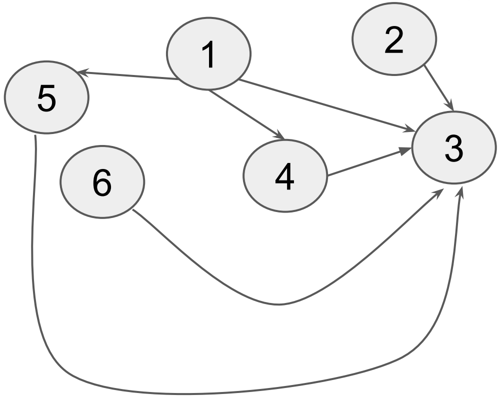
2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V|-1$ and out-degree 0, given an adjacency-matrix for G.



Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V|-1$ and out-degree 0, given an adjacency-matrix for G.

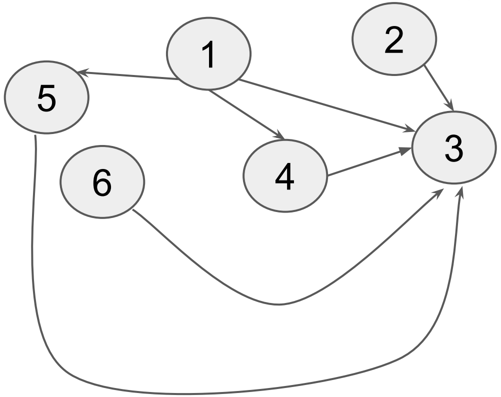


Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

Obs 2: row 3 is all 0s

$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 0 & 1 & 1 & 1 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 0 & 0 \\
5 & 0 & 0 & 1 & 0 & 0 & 0 \\
6 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
$$

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

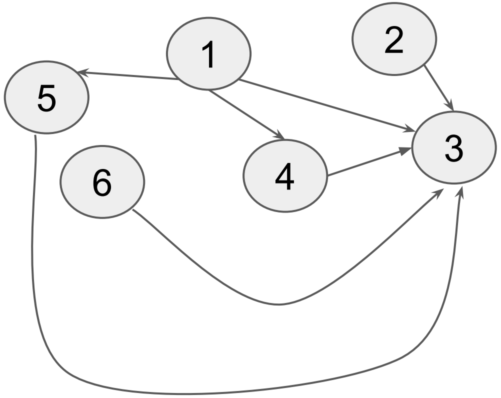
Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

Obs 2: row 3 is all 0s

Algorithm:

1. Start at (i,j) = (1,1) entry

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

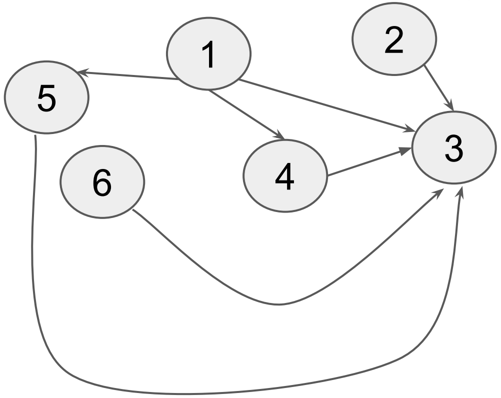
Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

Obs 2: row 3 is all 0s

Algorithm:

1. Start at (i,j) = (1,1) entry
2. while i < |V| :

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V|-1$ and out-degree 0, given an adjacency-matrix for G.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

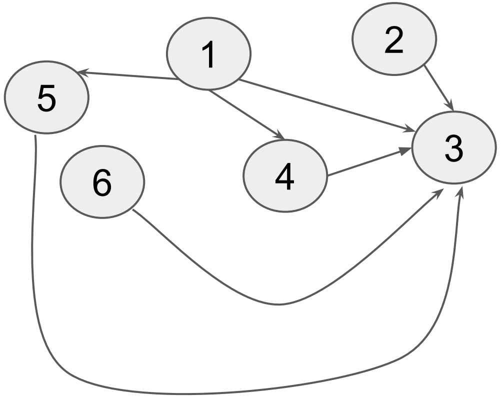
Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

Obs 2: row 3 is all 0s

Algorithm:

1. Start at (i,j) = (1,1) entry
2. while i < |V| :
    a. If entry(i,j) = 0: i += 1 \\go right

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

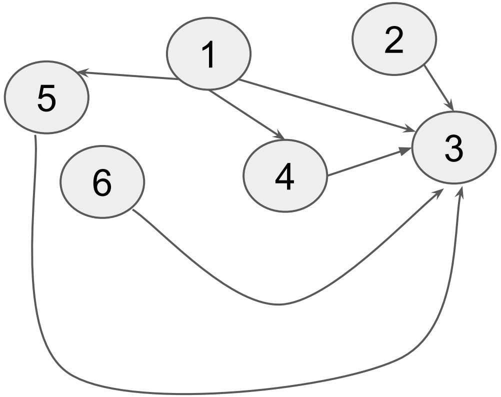
Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

Obs 2: row 3 is all 0s

Algorithm:

1. Start at (i,j) = (1,1) entry
2. while i < |V| :
    a. If entry(i,j) = 0: i += 1 \\go right
    b. else: j += 1 \\go down

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V|-1$ and out-degree 0, given an adjacency-matrix for G.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

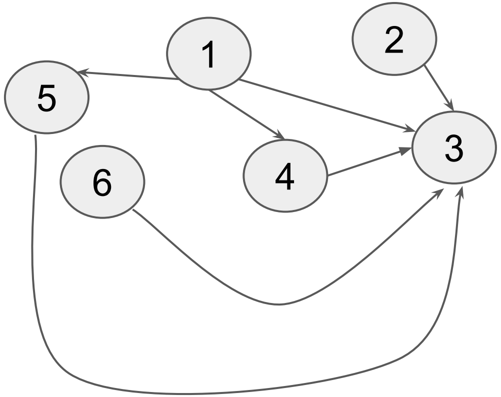
Obs 2: row 3 is all 0s

Algorithm:

1. Start at (i,j) = (1,1) entry
2. while i < |V| :
   a. If entry(i,j) = 0: i += 1 \\go right
   b. else: j += 1 \\go down

We only go down when entry(i,j) = 1

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

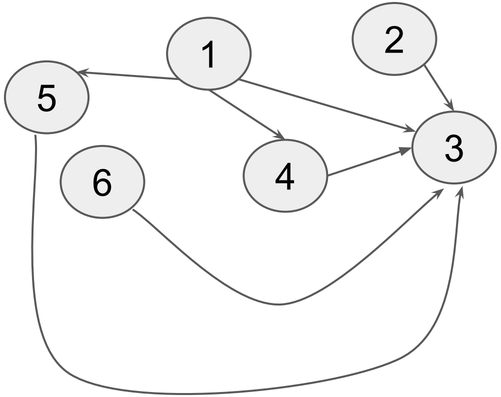
Obs 2: row 3 is all 0s

Algorithm:

1. Start at (i,j) = (1,1) entry
2. while i < |V| :
   a. If entry(i,j) = 0: i += 1 \\go right
   b. else: j += 1 \\go down

We only go down when entry(i,j) = 1
By obs 1, we will go down 2 times at most

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

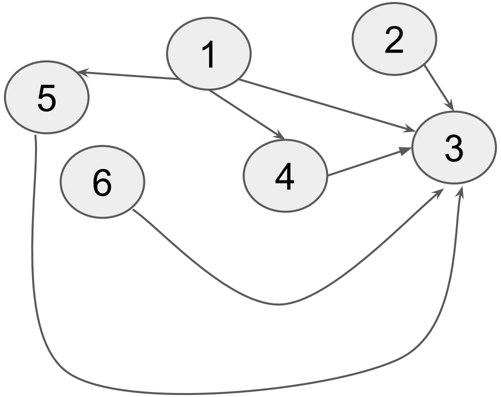
Obs 2: row 3 is all 0s

Algorithm:

1. Start at (i,j) = (1,1) entry
2. while i < |V| :
   a. If entry(i,j) = 0: i += 1 \\go right
   b. else: j += 1 \\go down

We only go down when entry(i,j) = 1
By obs 1, we will go down 2 times at most
We go right when entry(i,j) = 0

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

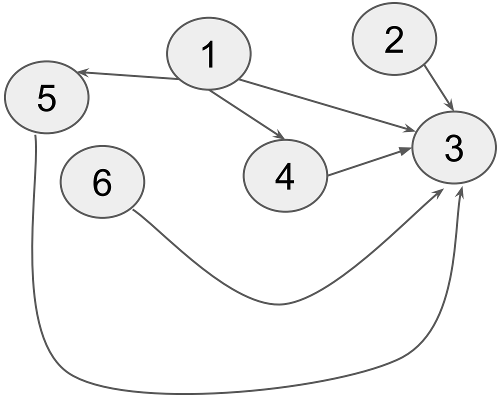
Obs 2: row 3 is all 0s

Algorithm:

1. Start at (i,j) = (1,1) entry
2. while i < |V| :
   a. If entry(i,j) = 0: i += 1 \\go right
   b. else: j += 1 \\go down

We only go down when entry(i,j) = 1
By obs 1, we will go down 2 times at most
We go right when entry(i,j) = 0

2. Show how to determine in $O(|V|)$ time, whether a directed graph G contains a **universal-sink**, i.e. a vertex with in-degree $|V| - 1$ and out-degree 0, given an adjacency-matrix for G.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

Obs 1: If universal sink, col 3 has 1 in every entry except (3,3)

Obs 2: row 3 is all 0s

Algorithm:

1. Start at (i,j) = (1,1) entry
2. while i < |V| :
   a. If entry(i,j) = 0: i += 1 \\go right
   b. else: j += 1 \\go down

We only go down when entry(i,j) = 1
By obs 1, we will go down 2 times at most
We go right when entry(i,j) = 0 By obs 2, we go right |V| times

Question 3

(Graphs)

1. True or False:

(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

(2) There exists a simple, undirected graph G with n vertices, whose vertex degrees are $0, 1, 2, \ldots, n-1$. (assume $n > 3$)

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

We can try..

1. True or False:

(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

Hmm if I can't come up with an example, prob false

The answer key says..

False. For an undirected graph, the total degree should be a even number. But $5 * 3 = 15$, which is odd.

But I don't know what this means lol

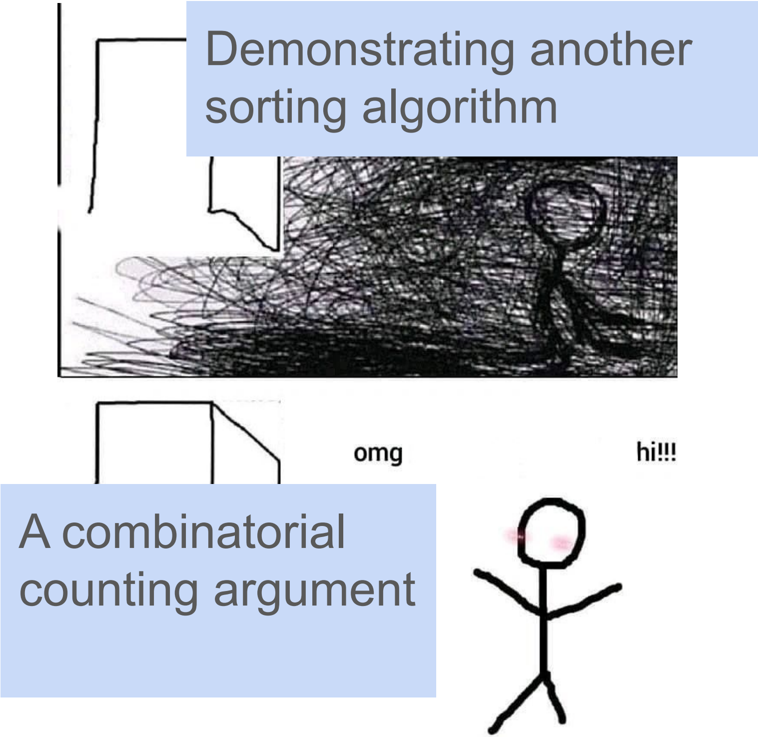
Why should the degree be an even number?

1. True or False:

(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

Hmm if I can’t come up with an example, prob false

We can prove it by counting



1. True or False:

(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

Hmm if I can’t come up with an example, prob false

We can prove it by counting

Assume for the sake of contradiction, this is possible

A combinatorial counting argument

1. True or False:

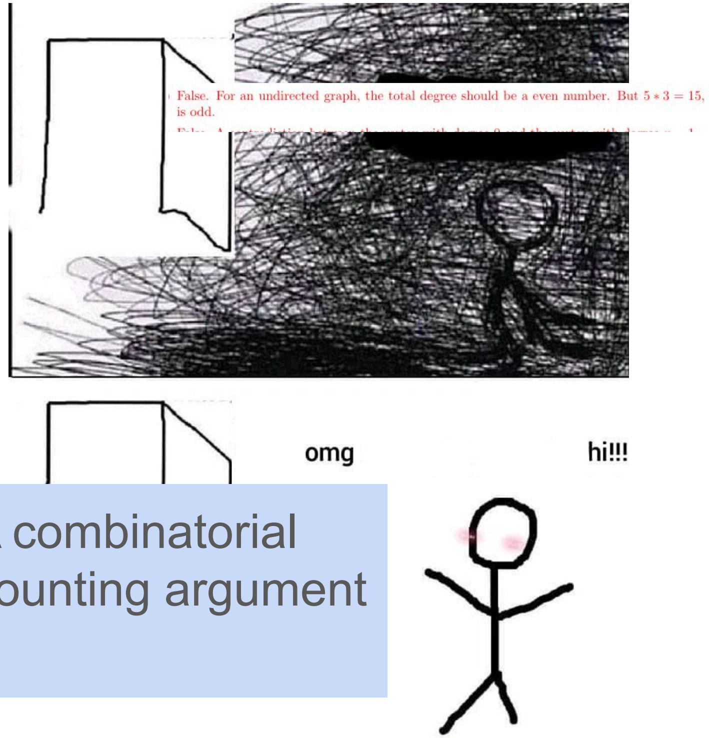
(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

Hmm if I can’t come up with an example, prob false

We can prove it by counting

Assume for the sake of contradiction, this is possible

Then there must be at least 15 edges (true)



A combinatorial counting argument

1. True or False:

(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

Hmm if I can’t come up with an example, prob false

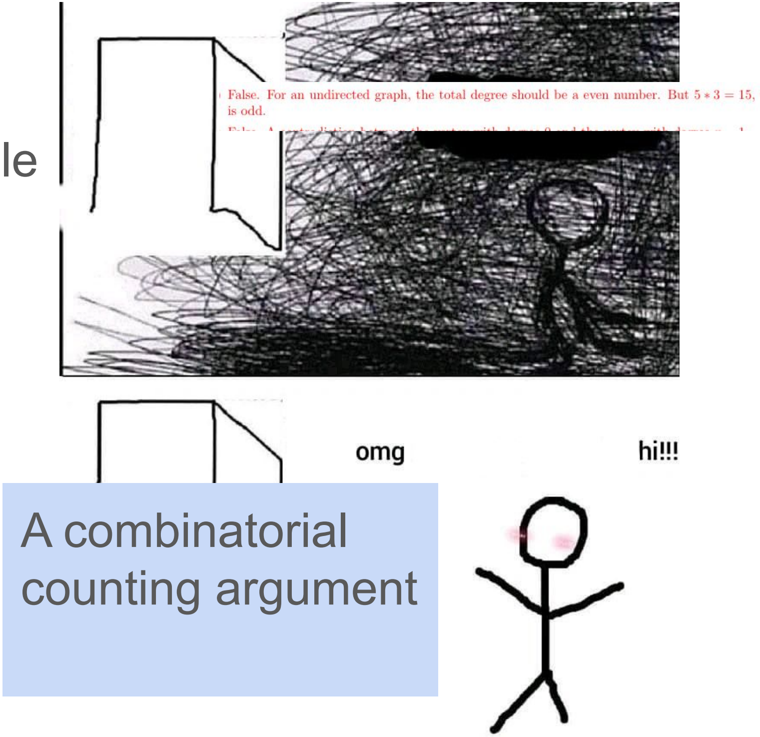
We can prove it by counting

Assume for the sake of contradiction, this is possible

Then there must be at least 15 edges (true)

But the maximum number of edges in a

graph of 5 nodes is..



1. True or False:

(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

Hmm if I can’t come up with an example, prob false

We can prove it by counting

Assume for the sake of contradiction, this is possible

Then there must be at least 15 edges (true)

But the maximum number of edges in a

graph of 5 nodes is.. (5 choose 2) = 10

This is less than 15, contradiction!

A combinatorial counting argument

Question 3

(Graphs)

1. True or False:

(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

(2) There exists a simple, undirected graph G with n vertices, whose vertex degrees are $0, 1, 2, \ldots, n-1$. (assume $n > 3$)

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

We can try.. (again)

(2) There exists a simple, undirected graph G with n vertices, whose vertex degrees are $0, 1, 2, \ldots, n-1$. (assume $n > 3$)

We can't do this because a vertex with degree n - 1 connects to all other vertices

There cannot be a vertex with 0 degree

Easy peasy

Question 3

(Graphs)

1. True or False:

(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

(2) There exists a simple, undirected graph G with n vertices, whose vertex degrees are $0, 1, 2, \ldots, n-1$. (assume $n > 3$)

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

Side note: I would say this is a theorem

Lemmas: intermediate results used to prove theorems
Corollaries: easy follow-ups to to theorems

IMO this stands on its own

Question 3

(Graphs)

1. True or False:

(1) There exists a simple, undirected graph with 5 nodes, each of degree 3.

(2) There exists a simple, undirected graph G with n vertices, whose vertex degrees are $0, 1, 2, \ldots, n-1$. (assume $n > 3$)

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

Side note: I would say this is a theorem

Lemmas: intermediate results used to prove theorems

Corollaries: easy follow-ups to to theorems

IMO this stands on its own

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

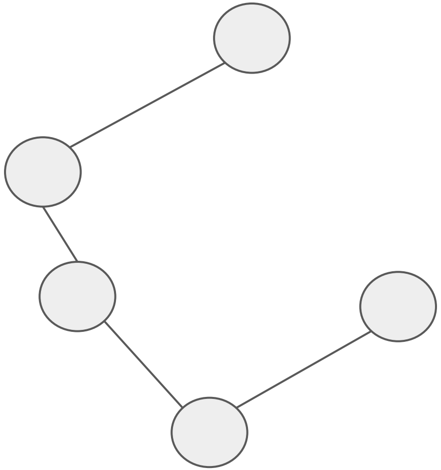
(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 2 -> 3

Suppose G is connected and acyclic



$|V| = 5, |E| = 4$

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

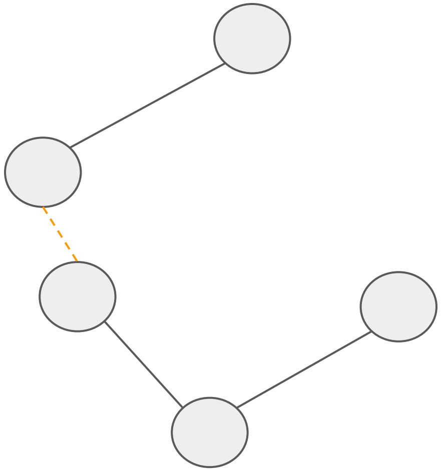
(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

Can |E| < |V| - 1?



|V| = 5, |E| = 3

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

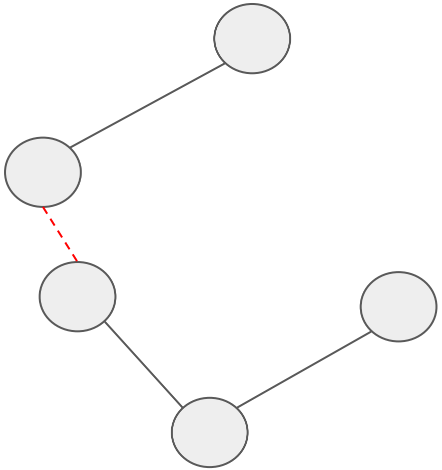
WTS: 1 + 2 -> 3

Suppose G is connected and acyclic

Can |E| < |V| - 1?

Intuition: lose connectivity

Lets prove this



|V| = 5, |E| = 3

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

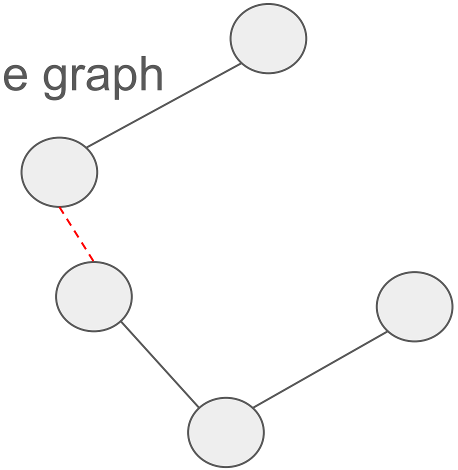
(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

Connectivity implies I can take a walk to every vertex on the graph



|V| = 5, |E| = 3

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

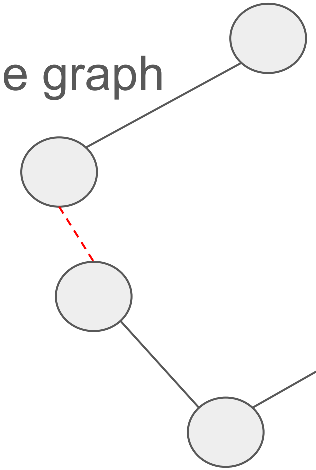
<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

Connectivity implies I can take a walk to every vertex on the graph

Say I talk this walk starting on some vertex,

and mark every edge I step on



|V| = 5, |E| = 3

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

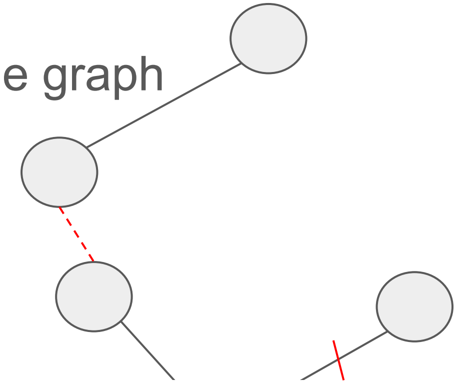
<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

Connectivity implies I can take a walk to every vertex on the graph

Say I talk this walk starting on some vertex,

and mark every edge I step on



|V| = 5, |E| = 3

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

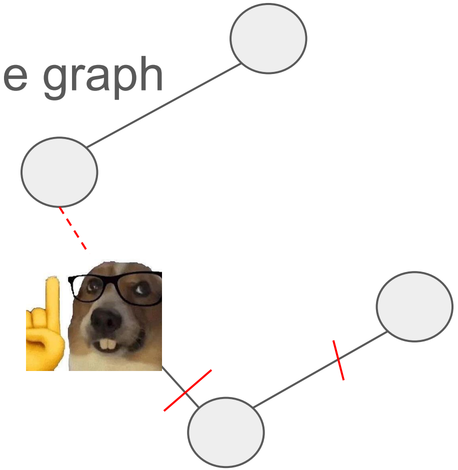
<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

Connectivity implies I can take a walk to every vertex on the graph

Say I talk this walk starting on some vertex,

and mark every edge I step on



|V| = 5, |E| = 3

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

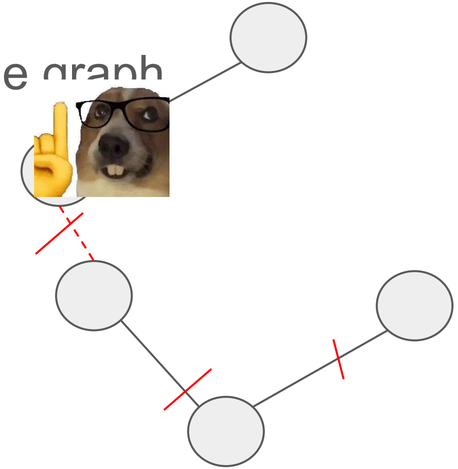
<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

Connectivity implies I can take a walk to every vertex on the graph

Say I talk this walk starting on some vertex,

and mark every edge I step on



|V| = 5, |E| = 3

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

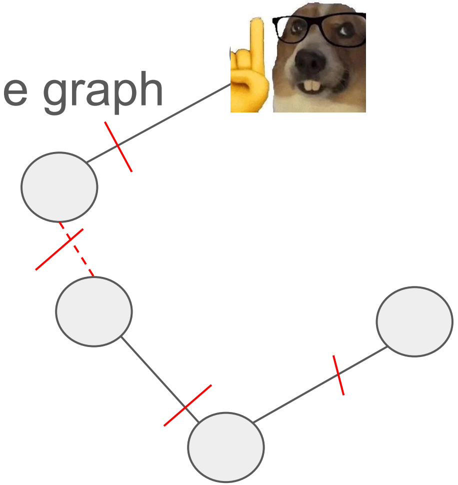
<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

Connectivity implies I can take a walk to every vertex on the graph

Say I talk this walk starting on some vertex,

and mark every edge I step on



$|V| = 5$, $|E| = 3$

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 2 -> 3

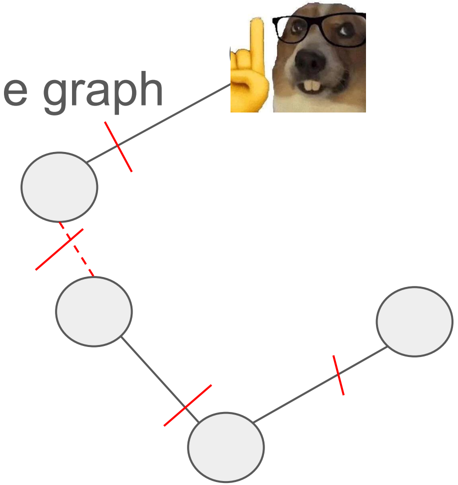
Suppose G is connected and acyclic

Connectivity implies I can take a walk to every vertex on the graph

Say I talk this walk starting on some vertex,

and mark every edge I step on

For each unique vertex I visit, I had to take an edge there



|V| = 5, |E| = 3

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

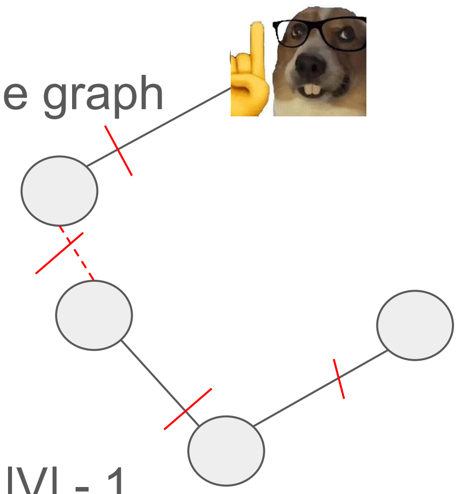
Connectivity implies I can take a walk to every vertex on the graph

Say I talk this walk starting on some vertex,

and mark every edge I step on

For each unique vertex I visit, I had to take an edge there

Since I visit |V| - 1 unique vertices (minus the start), |E| >= |V| - 1



|V| = 5, |E| = 3

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

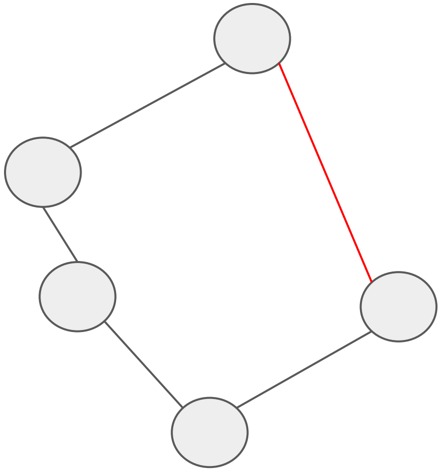
(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 2 -> 3

Suppose G is connected and acyclic

We showed |E| >= |V| - 1

Can |E| > |V| - 1?



|V| = 5, |E| = 5

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

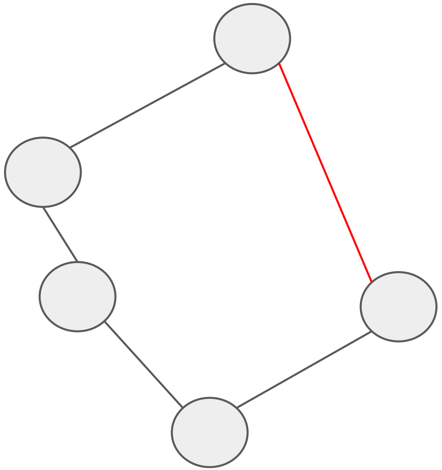
(3) G satisfies $|E| = |V| - 1$.

<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

Can $|E| > |V| - 1$?

(Anywhere I add the edge will create a cycle)



$|V| = 5$, $|E| = 5$

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

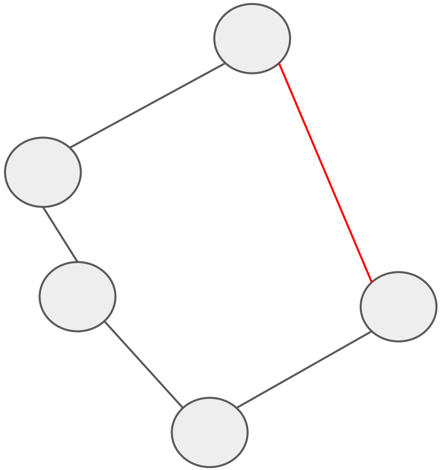
(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 2 -> 3

Suppose G is connected and acyclic

Assume for the sake of contradiction |E| > |V| - 1

(Anywhere I add the edge will create a cycle)



|V| = 5, |E| = 5

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

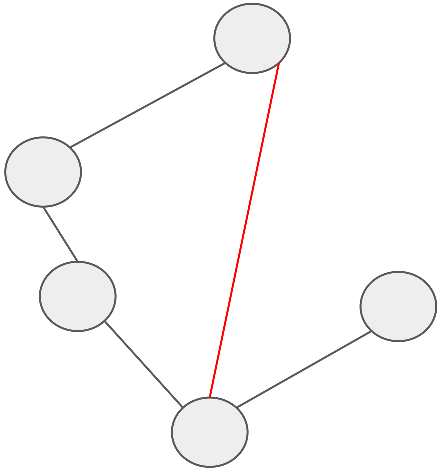
(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 2 -> 3

Suppose G is connected and acyclic

Assume for the sake of contradiction $|E| > |V| - 1$

(Anywhere I add the edge will create a cycle)



$|V| = 5$, $|E| = 5$

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

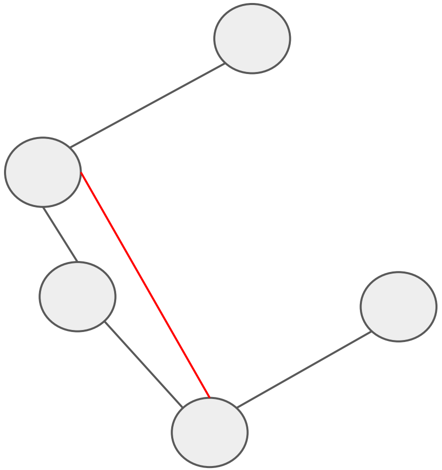
(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 2 -> 3

Suppose G is connected and acyclic

Assume for the sake of contradiction |E| > |V| - 1

(Anywhere I add the edge will create a cycle)



|V| = 5, |E| = 5

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

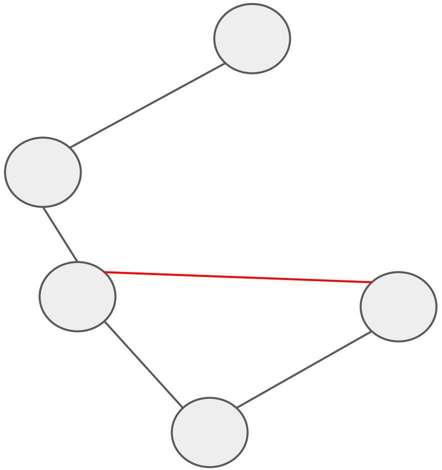
WTS: 1 + 2 -> 3

Suppose G is connected and acyclic

Assume for the sake of contradiction $|E| > |V| - 1$

(Anywhere I add the edge will create a cycle)

We argue this formally



$|V| = 5$, $|E| = 5$

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

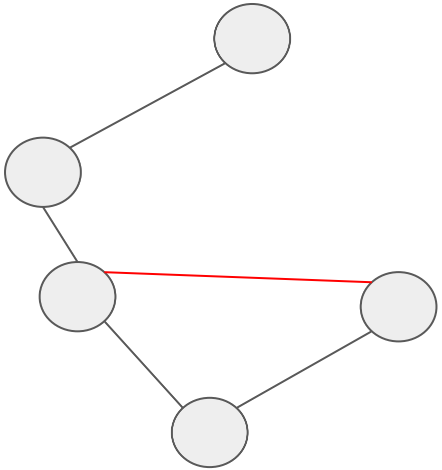
(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 2 -> 3

Suppose G is connected and acyclic

Assume for the sake of contradiction |E| > |V| - 1

Connected implies longest path in the graph is through all |V| nodes.



|V| = 5, |E| = 5

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

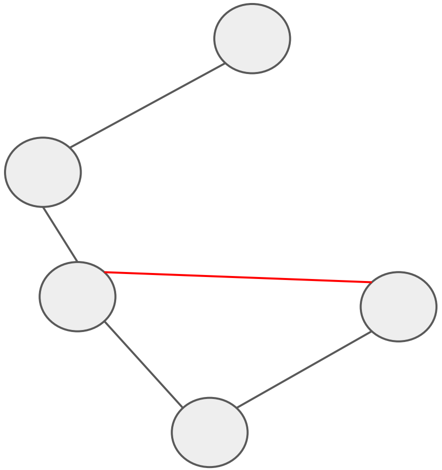
<u>WTS</u>: 1 + 2 -> 3

Suppose G is connected and acyclic

Assume for the sake of contradiction |E| > |V| - 1

Connected implies longest path in the graph is through all |V| nodes.

But a path of |V| nodes only has |V| - 1 edges



|V| = 5, |E| = 5

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 2 -> 3

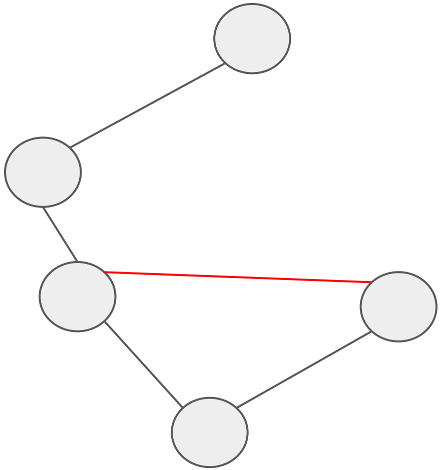
Suppose G is connected and acyclic

Assume for the sake of contradiction |E| > |V| - 1

Connected implies longest path in the graph is through all |V| nodes.

But a path of |V| nodes only has |V| - 1 edges

|V| nodes with |V| edges forms a cycle, contradiction!



|V| = 5, |E| = 5

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

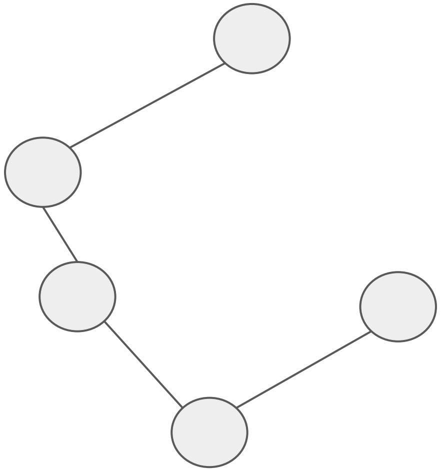
(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 3 -> 2

Suppose connected and |E| = |V| - 1

Can there be a cycle?



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

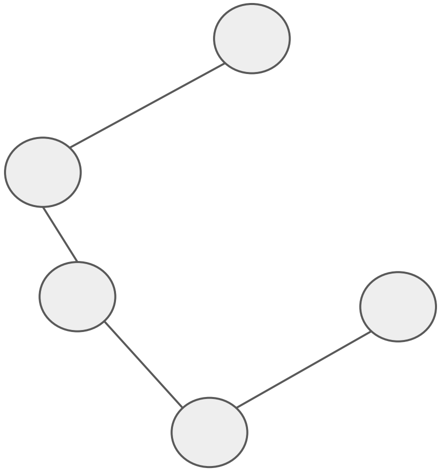
<u>WTS</u>: 1 + 3 -> 2

Suppose connected and |E| = |V| - 1

Can there be a cycle?

No, we showed to be connected, we need at least |V| - 1

edges.



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 3 -> 2

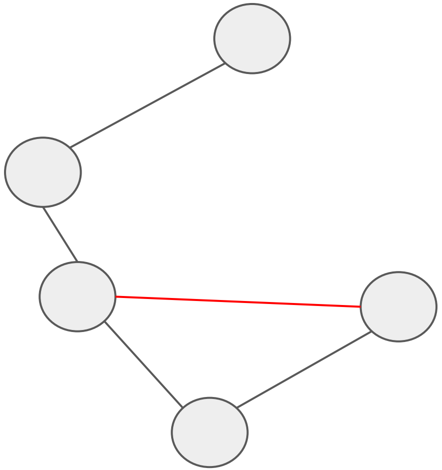
Suppose connected and |E| = |V| - 1

Can there be a cycle?

No, we showed to be connected, we need at least |V| - 1

edges.

Suppose there is a cycle.



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 1 + 3 -> 2

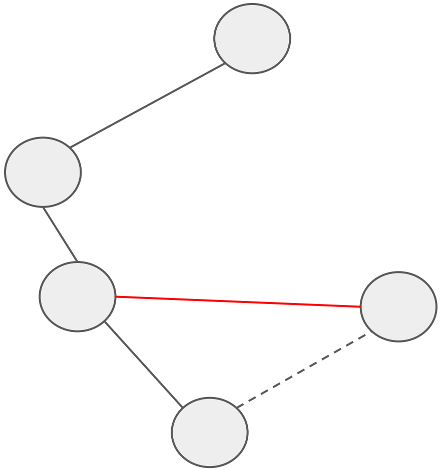
Suppose connected and |E| = |V| - 1

Can there be a cycle?

No, we showed to be connected, we need at least |V| - 1 edges.

Suppose there is a cycle.

There is an edge you can delete to get rid of the cycle



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

<u>WTS</u>: 1 + 3 -> 2

Suppose connected and |E| = |V| - 1

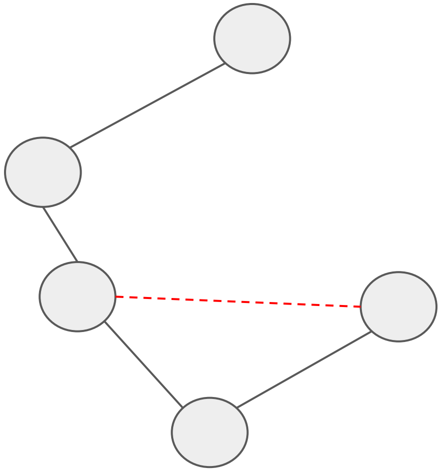
Can there be a cycle?

No, we showed to be connected, we need at least |V| - 1 edges.

Suppose there is a cycle.

There is an edge you can delete to get rid of the cycle

We still have connectivity with |V| -2 edges, contradiction



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

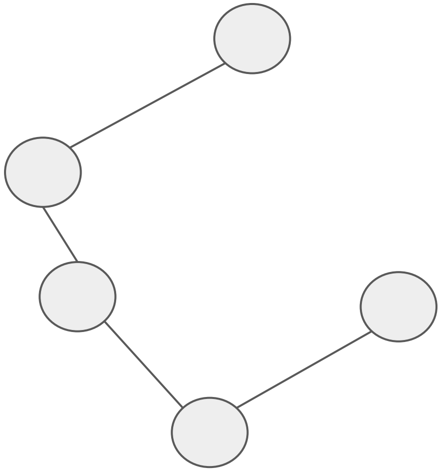
(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 2 + 3 -> 1

Suppose acyclic and |E| = |V| - 1.



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

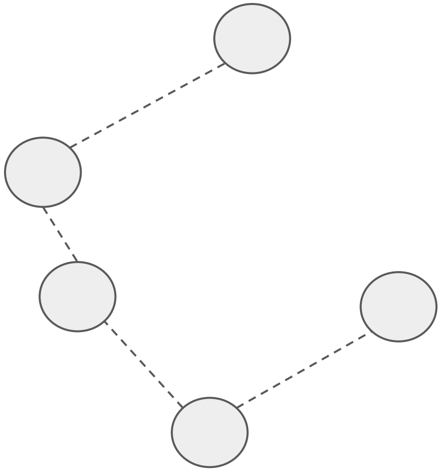
(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 2 + 3 -> 1

Suppose acyclic and |E| = |V| - 1.

Suppose I remove all |E| edges from the graph



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

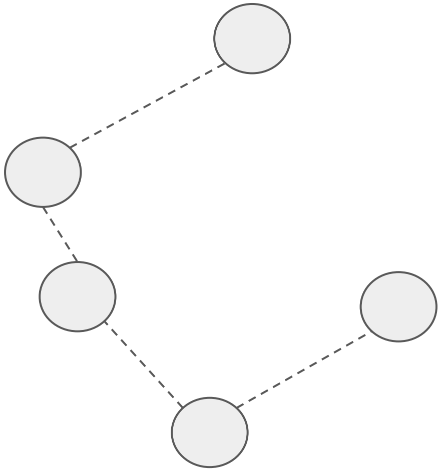
(3) G satisfies $|E| = |V| - 1$.

<u>WTS</u>: 2 + 3 -> 1

Suppose acyclic and |E| = |V| - 1.

Suppose I remove all |E| edges from the graph

I place them back one by one.



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

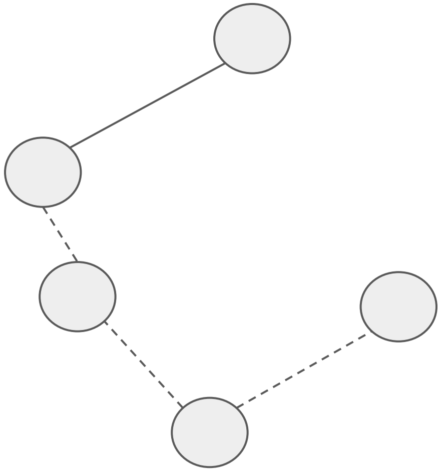
(3) G satisfies $|E| = |V| - 1$.

<u>WTS</u>: 2 + 3 -> 1

Suppose acyclic and |E| = |V| - 1.

Suppose I remove all |E| edges from the graph

I place them back one by one.



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

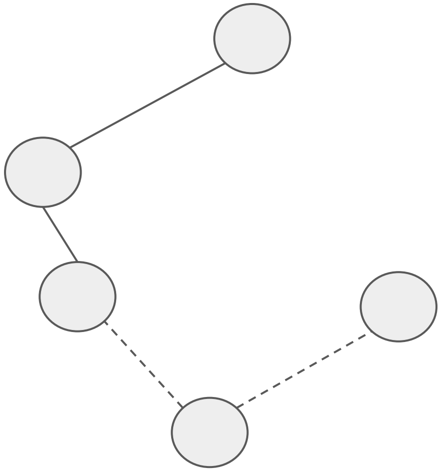
WTS: 2 + 3 -> 1

Suppose acyclic and |E| = |V| - 1.

Suppose I remove all |E| edges from the graph

I place them back one by one.

What do I notice?



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 2 + 3 -> 1

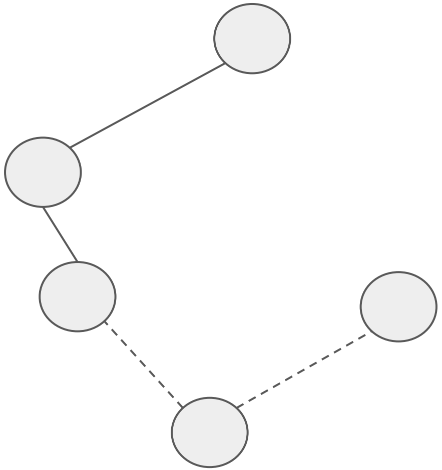
Suppose acyclic and |E| = |V| - 1.

Suppose I remove all |E| edges from the graph

I place them back one by one.

By acyclic property, any edge I add back has to contain a

unique vertex + a seen vertex (except the starting edge)



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 2 + 3 -> 1

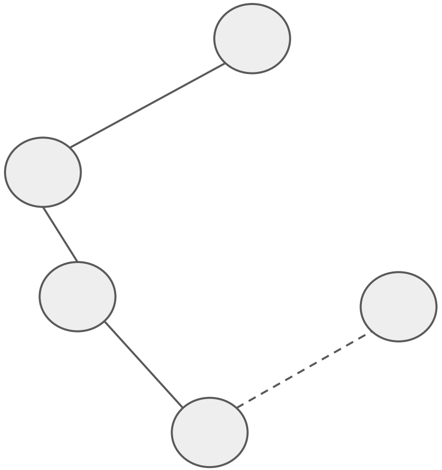
Suppose acyclic and |E| = |V| - 1.

Suppose I remove all |E| edges from the graph

I place them back one by one.

By acyclic property, any edge I add back has to contain a

unique vertex + a seen vertex (except the starting edge)



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

WTS: 2 + 3 -> 1

Suppose acyclic and |E| = |V| - 1.

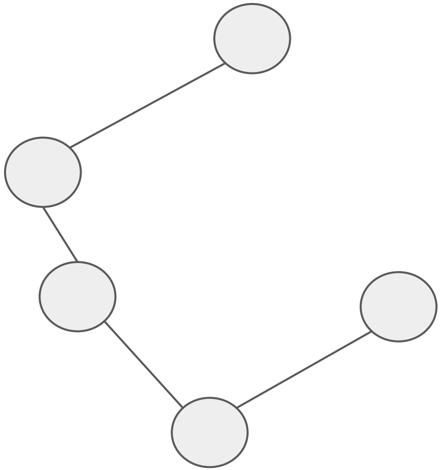
Suppose I remove all |E| edges from the graph

I place them back one by one.

By acyclic property, any edge I add back has to contain a

unique vertex + a seen vertex (except the starting edge)

The edge I start with has 2 unique vertices



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

<u>WTS</u>: 2 + 3 -> 1

Suppose acyclic and |E| = |V| - 1.

Suppose I remove all |E| edges from the graph

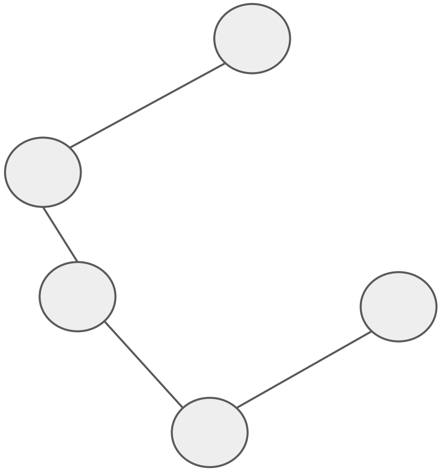
I place them back one by one.

By acyclic property, any edge I add back has to contain a

unique vertex + a seen vertex (except the starting edge)

The edge I start with has 2 unique vertices

vertices seen = (2) + (|E| - 1) (1) = 2 + |V| - 2 = |V|



|V| = 5, |E| = 4

2. A tree is the most widely used special type of graph, in a sense that it is the minimal connected graph. Prove the following important lemma:

Let G be an undirected graph, any two of the following properties imply the third property, and that G is a tree.

(1) G is connected;

(2) G is acyclic;

(3) G satisfies $|E| = |V| - 1$.

<u>WTS</u>: 2 + 3 -> 1

Suppose acyclic and |E| = |V| - 1.

Suppose I remove all |E| edges from the graph

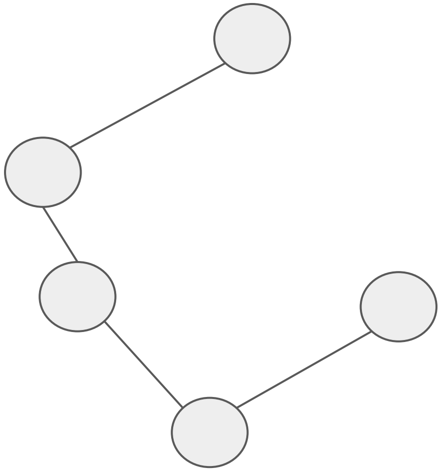
I place them back one by one.

By acyclic property, any edge I add back has to contain a

unique vertex + a seen vertex (except the starting edge)

The edge I start with has 2 unique vertices

vertices seen = (2) + (|E| - 1) (1) = 2 + |V| - 2 = |V|, hence connected



|V| = 5, |E| = 4

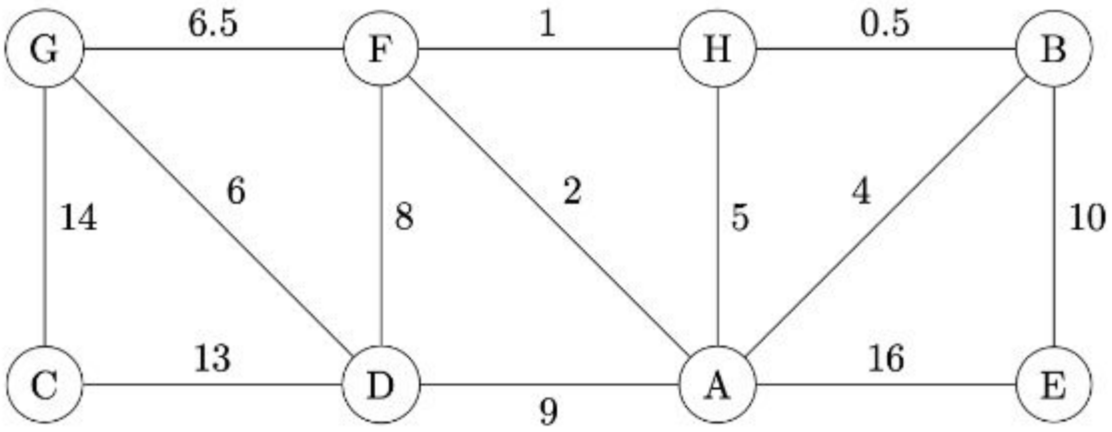
Question 4

(Review)

Consider the following undirected graph drawn below. For each part below we are only asking for the order in which edges are added. Assume that the graph is represented in adjacency-list form and that each adjacency-list is given in lexicographic order.

- List the order that edges are added to the BFS tree if we run BFS starting at node A.
- List the order that edges are added to the DFS tree if we run DFS starting at node A.

BFS



DFS

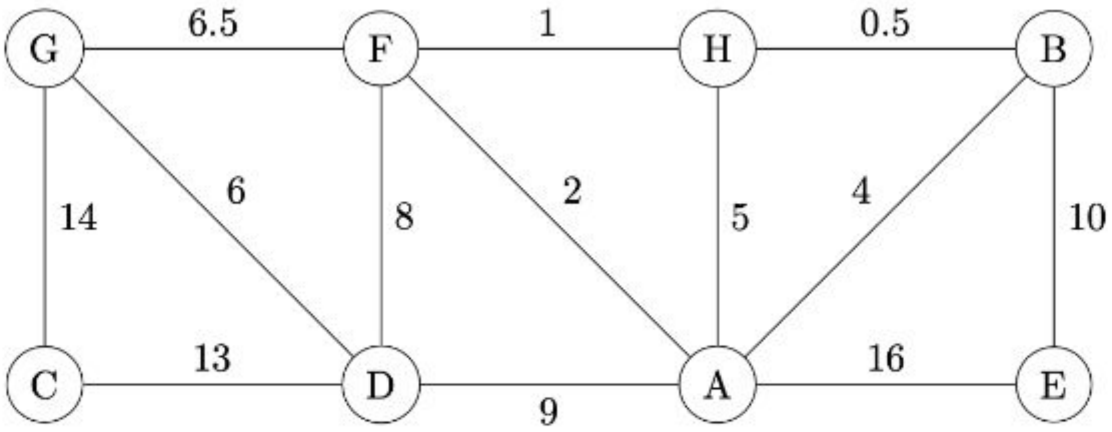
Question 4

(Review)

Consider the following undirected graph drawn below. For each part below we are only asking for the order in which edges are added. Assume that the graph is represented in adjacency-list form and that each adjacency-list is given in lexicographic order.

- List the order that edges are added to the BFS tree if we run BFS starting at node A.
- List the order that edges are added to the DFS tree if we run DFS starting at node A.

BFS



DFS

Question 5

(Breadth-first search)

1. What is the running time of BFS if we represent its input graph by an adjacency-matrix instead of the adjacency-list representation?

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\text{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

BFS(s):

stack/queue(?) visit;

Add s to visit

while visit nonempty:

v = visit.pop()

If v not seen yet, add its neighbors to visit

Lets analyze cost

Question 5

(Breadth-first search)

1. What is the running time of BFS if we represent its input graph by an adjacency-matrix instead of the adjacency-list representation?
2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\text{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

BFS(s):

stack/queue(?) visit;

Add s to visit

while visit nonempty:

v = visit.pop()

If v not seen yet, add its neighbors to visit

What's the cost of this when seen is a hashmap?

Question 5

(Breadth-first search)

1. What is the running time of BFS if we represent its input graph by an adjacency-matrix instead of the adjacency-list representation?

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\text{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

```
BFS(s):
    stack/queue(?) visit;
    Add s to visit
    while visit nonempty:
        v = visit.pop()
        If v not seen yet, add its neighbors to visit
```

What's the cost of this when seen is a hashmap? O(1)

What's the cost of this?

Question 5

(Breadth-first search)

1. What is the running time of BFS if we represent its input graph by an adjacency-matrix instead of the adjacency-list representation?
2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\text{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

BFS(s):

stack/queue(?) visit;

Add s to visit

while visit nonempty:

v = visit.pop()

If v not seen yet, add its neighbors to visit

What's the cost of this when seen is a hashmap? O(1)

What's the cost of this? O(# v's neighbors) <= O(|V|)

Question 5

(Breadth-first search)

1. What is the running time of BFS if we represent its input graph by an adjacency-matrix instead of the adjacency-list representation?

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\mathrm{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

BFS(s):

stack/queue(?) visit;

Add s to visit

while visit nonempty:

v = visit.pop()

If v not seen yet, <u>add its neighbors to visit</u>

What's the cost of this when seen is a hashmap? O(1)

What's the cost of this? O(# v's neighbors) <= O(|V|)

How many iterations of the while loop?

Question 5

(Breadth-first search)

1. What is the running time of BFS if we represent its input graph by an adjacency-matrix instead of the adjacency-list representation?

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\text{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

BFS(s):

stack/queue(?) visit;

Add s to visit

while visit nonempty:

v = visit.pop()

If v not seen yet, add its neighbors to visit

What's the cost of this when seen is a hashmap? O(1)

What's the cost of this? O(# v's neighbors) <= O(|V|)

How many iterations of the while loop? O(|V|)

Total cost: $O(|V|^2)$

Question 5

(Breadth-first search)

1. What is the running time of BFS if we represent its input graph by an adjacency-matrix instead of the adjacency-list representation?

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\text{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

```
BFS(s):
    stack/queue(?) visit;
    Add s to visit
    while visit nonempty:
        v = visit.pop()
        If v not seen yet, add its neighbors to visit
```

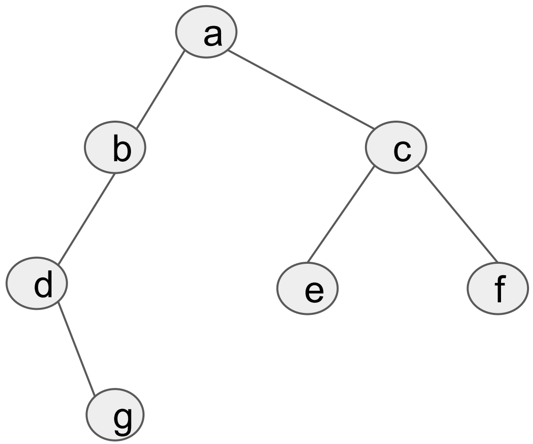
Diameter = length of longest path in the tree

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\text{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

Diameter = length of longest path in the tree

Suppose this is my tree

From visual inspection, clear that diameter is g to f



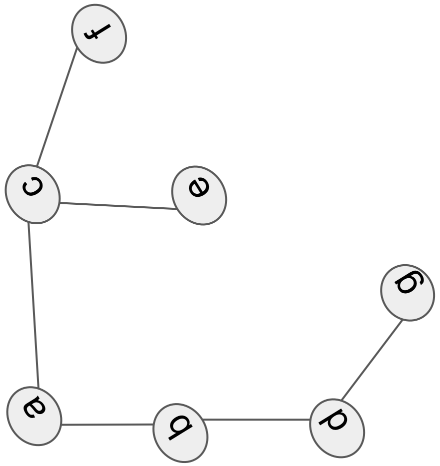
2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\mathrm{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

Diameter = length of longest path in the tree

Suppose this is my tree

From visual inspection, clear that diameter is g to f

But who says we know where the “root” is?



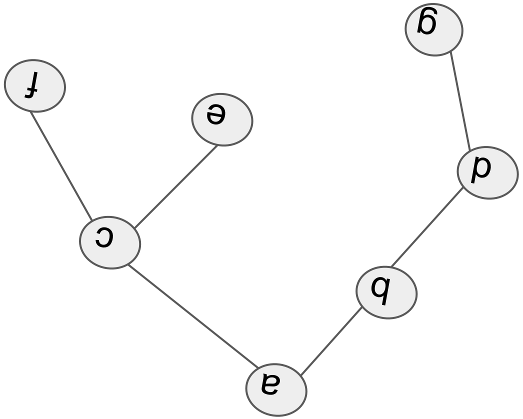
2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree dia(T) be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

Diameter = length of longest path in the tree

Suppose this is my tree

From visual inspection, clear that diameter is g to f

But who says we know where the “root” is?



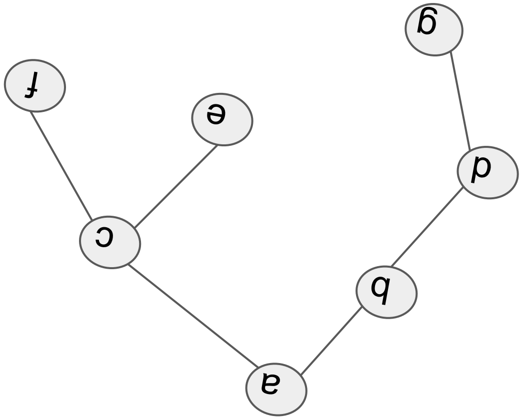
Fun fact: this is how trees look irl
(I touched grass 7 years ago)

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree dia(T) be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

Diameter = length of longest path in the tree

Suppose this is my tree

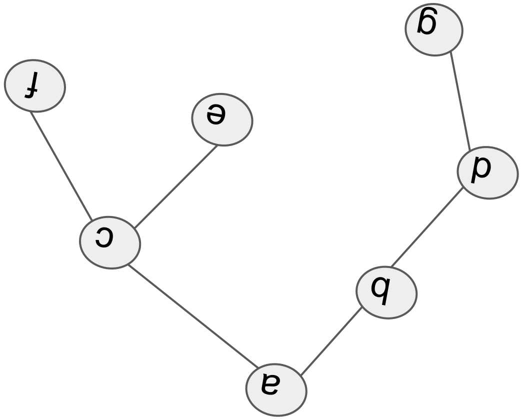
So the best we can do is just to run BFS from some starting node.



2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\text{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

Diameter = length of longest path in the tree

Suppose this is my tree



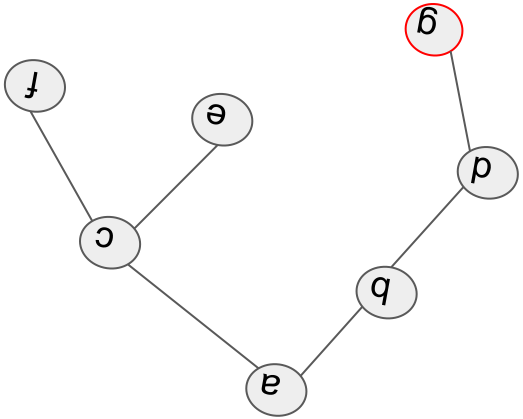
So the best we can do is just to run BFS from some starting node.

Say we run BFS(c) and return last vertex seen

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\text{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

Diameter = length of longest path in the tree

Suppose this is my tree



So the best we can do is just to run BFS from some starting node.

Say we run BFS(c) and return last vertex seen

We get g (this was one end point) How do we get the other? (vertex f)

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree dia(T) be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

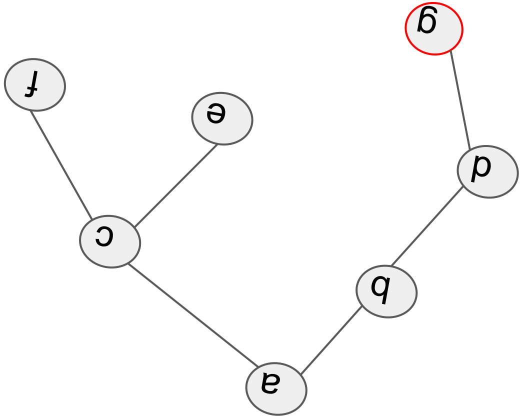
Diameter = length of longest path in the tree

Suppose this is my tree

So the best we can do is just to run BFS from some starting node.

Say we run BFS(c) and return last vertex seen

We get g (this was one end point) How do we get the other? (vertex f)

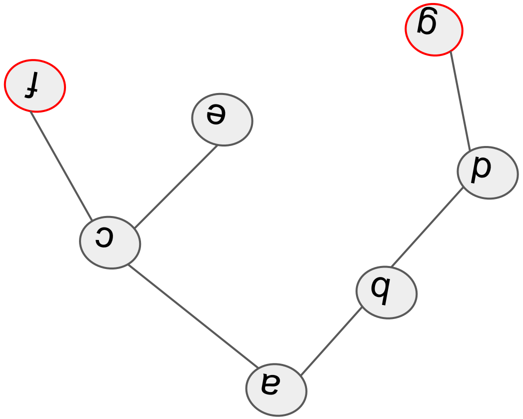


Run BFS(g)

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree dia(T) be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

Diameter = length of longest path in the tree

Suppose this is my tree



So the best we can do is just to run BFS from some starting node.

Say we run BFS(c) and return last vertex seen

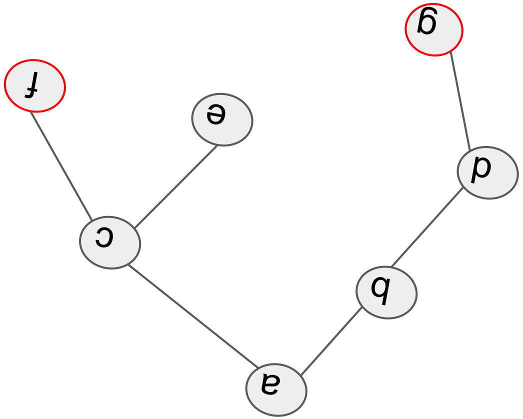
We get g (this was one end point) How do we get the other? (vertex f)

Run BFS(g)

2. **(Diameter of a tree)** We know that the BFS finds the shortest path from the source s to each reachable vertex. Now let $T = (V, E)$ be a tree and define The *diameter* of a tree $\mathrm{dia}(T)$ be the largest of all shortest-path distances in the tree. Think about how to use BFS to compute the diameter of a tree.

Diameter = length of longest path in the tree

Suppose this is my tree



So the best we can do is just to run BFS from some starting node.

Say we run BFS(c) and return last vertex seen

We get g (this was one end point) How do we get the other? (vertex f)

Run BFS(g)

Proof in the solution.pdf : kinda tedious but intuitively works

Question 6

(Depth-first search)

1. Is is possible that a vertex u of a directed graph G can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G?
2. **TRUE or False**: A directed graph G contains a path from u to v, and if u is visited before v in a DFS of G, then v must be a descendant of u in the corresponding DFS tree.

When in doubt, draw it out

Question 6

(Depth-first search)

1. Is is possible that a vertex u of a directed graph G can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G?

2. **TRUE or False**: A directed graph G contains a path from u to v, and if u is visited before v in a DFS of G, then v must be a descendant of u in the corresponding DFS tree.

When in doubt, draw it out

HAVE A GREAT SPRING BREAK!