

# طراحی الگوریتم ها (CE221)

## جلسه دوازدهم: گراف بدون وزن، BFS، و DFS

**سجاد شیرعلی شمرضا**

**بهار 1401**

**دوشنبه، 15 فروردین 1401**

# اطلاع رسانی

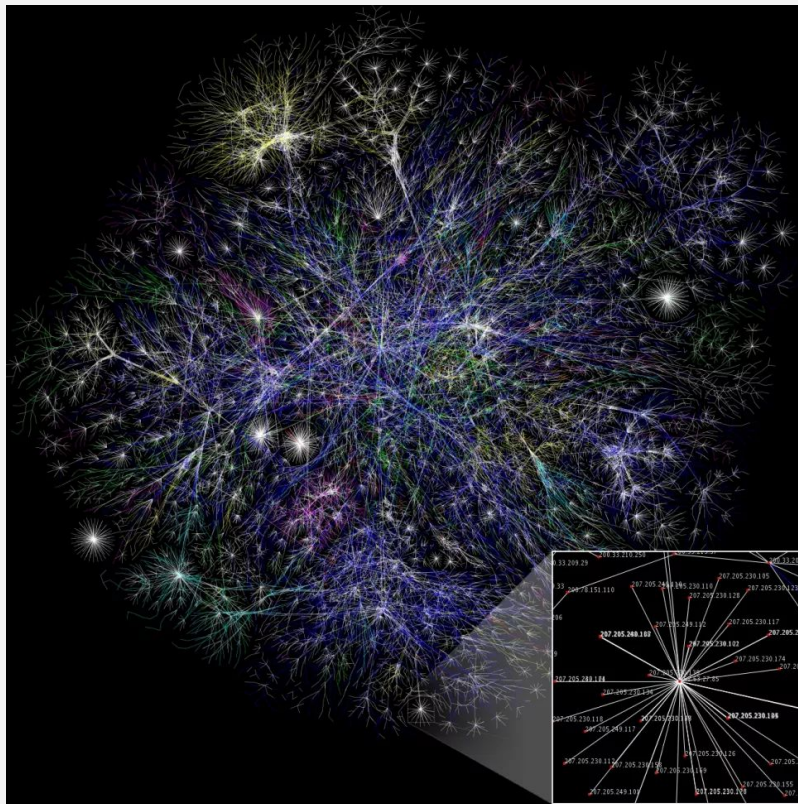
- بخش مرتبط کتاب برای این جلسه: 22

# گراف

**تعریف و نمونه**

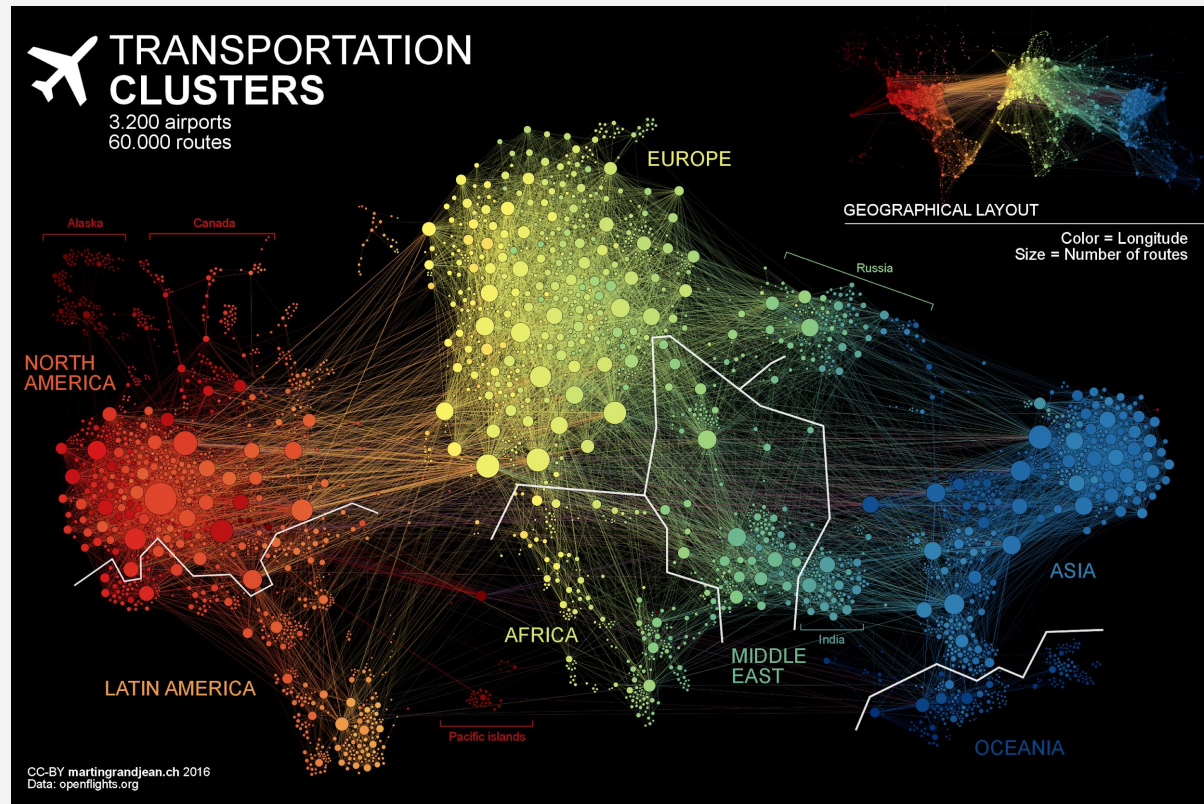
# GRAPH EXAMPLES

Partial graph of the Internet (in 2005), where each “node” is an IP address, and the “edges” between them reveal connectivity delays (shorter lines = closer IP addresses)



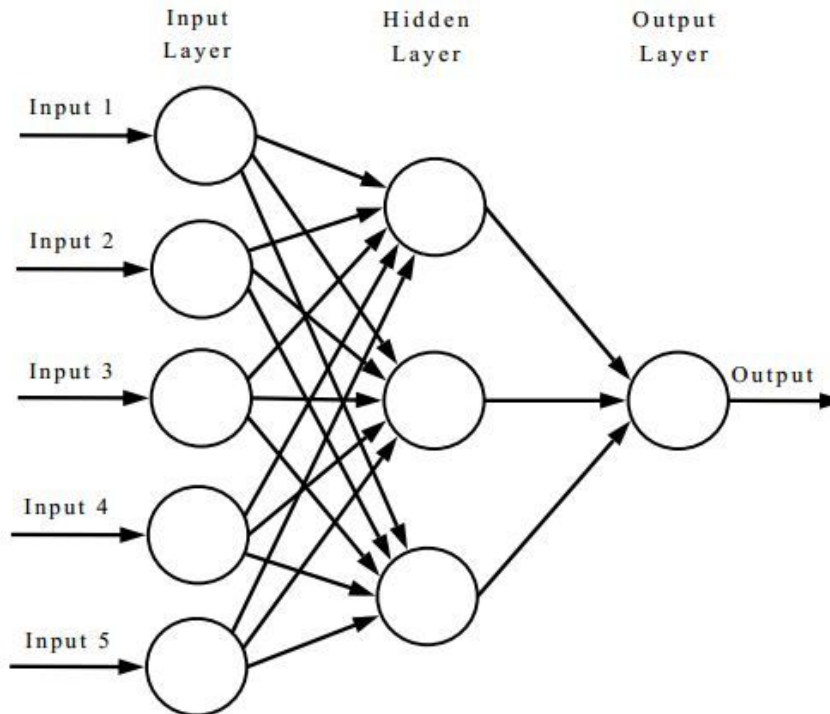
# GRAPH EXAMPLES

Each “node” is an airport,  
and flight routes are  
represented by the “edge”  
in between them



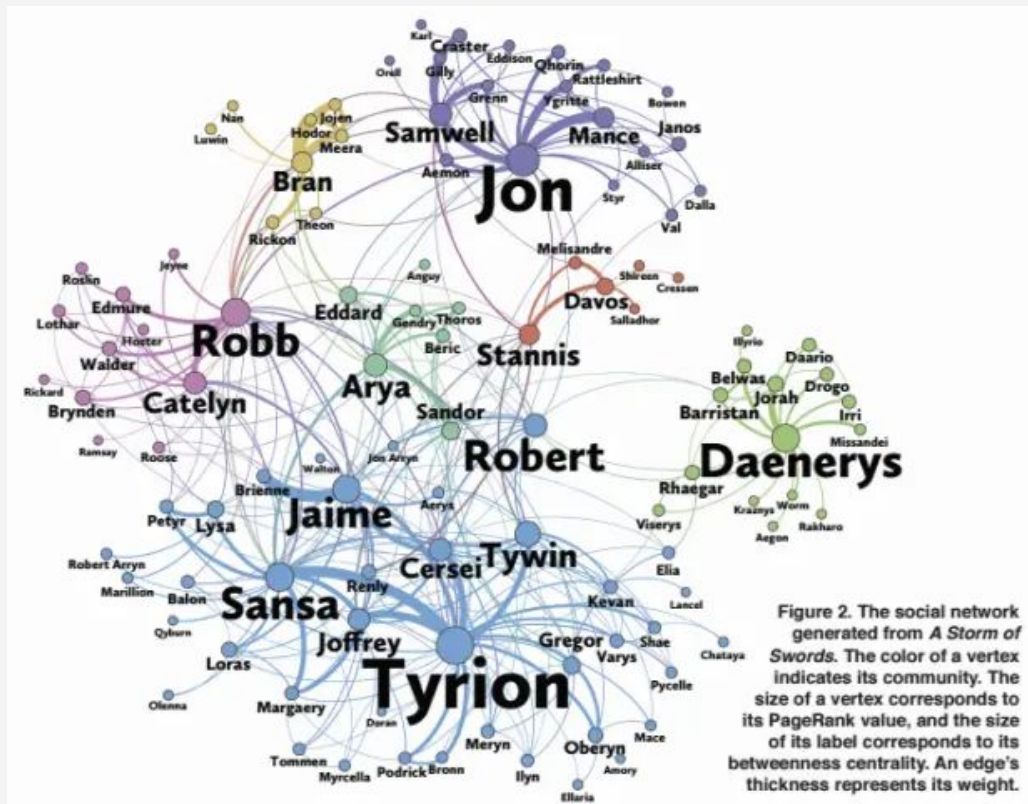
# GRAPH EXAMPLES

Neural networks! Each “node” represents a module of the neural network, and “edge” represent output/input relationships



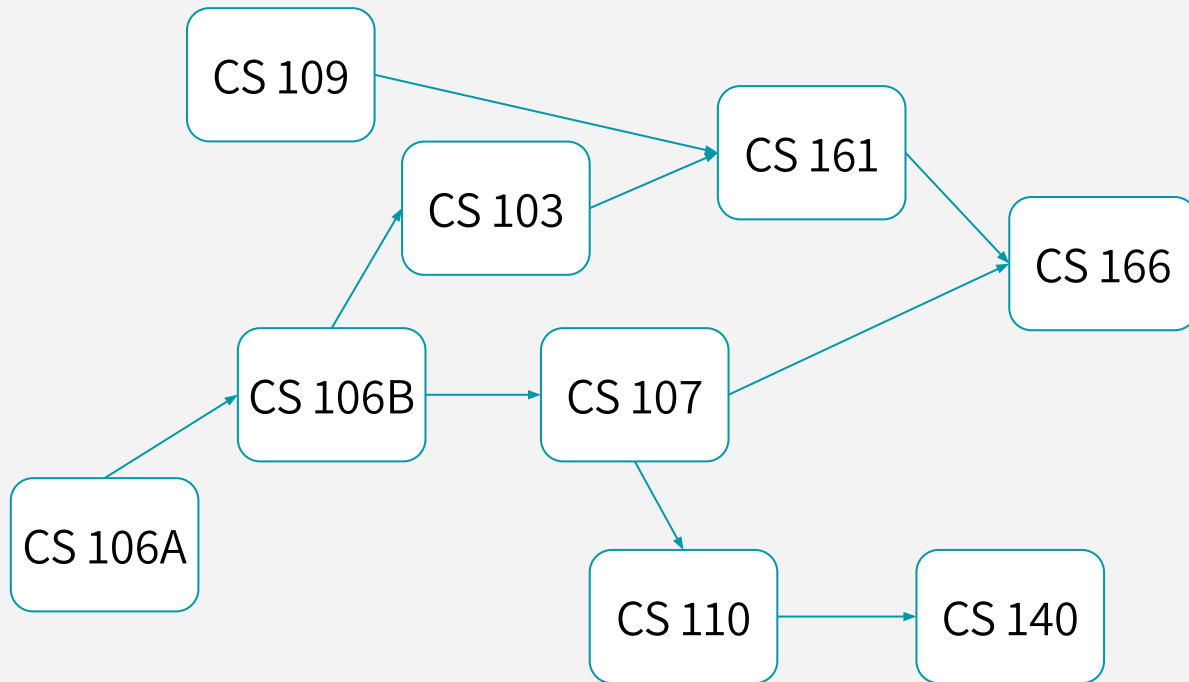
# GRAPH EXAMPLES

Graph of characters in the third book of Game of Thrones, where each “node” is a character, and “edge” reveal frequency of interaction (i.e. 2 names appearing within 15 words of one another).



# GRAPH EXAMPLES

CS prerequisites!  
“nodes” are classes  
and an “edge” from  
class A to class B  
means “class B  
depends on class A”





# WHAT ARE GRAPHS USED FOR?

- There are a lot of diverse problems that can be represented as graphs, and we want to answer questions about them
- For example:
  - How do we most efficiently route packets across the internet?
  - Are there natural “clusters” or “communities” in a graph?
  - Which character(s) are least related with \_\_\_\_\_?
  - How should I sign up for classes without violating pre-req constraints?

But first off, some terminology!

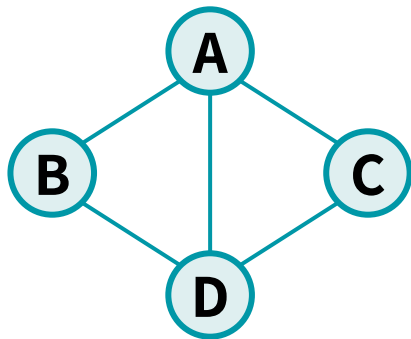
# 2 KINDS OF GRAPHS

We'll deal with both kinds of graphs in this class.

## UNDIRECTED GRAPHS

An undirected graph has  
a set of vertices ( $V$ ) & a set of edges ( $E$ )

Formally,  
 $G = (V, E)$



$V = \{A, B, C, D\}$

$E = \{ \{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{C, D\} \}$

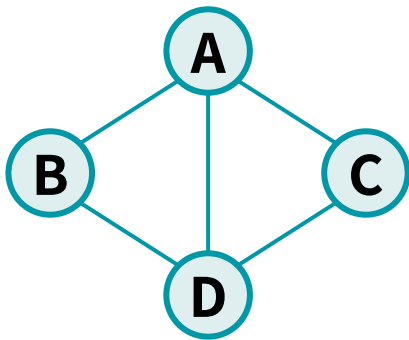
# 2 KINDS OF GRAPHS

We'll deal with both kinds of graphs in this class.

## UNDIRECTED GRAPHS

An undirected graph has  
a set of vertices ( $V$ ) & a set of edges ( $E$ )

Formally,  
 $G = (V, E)$

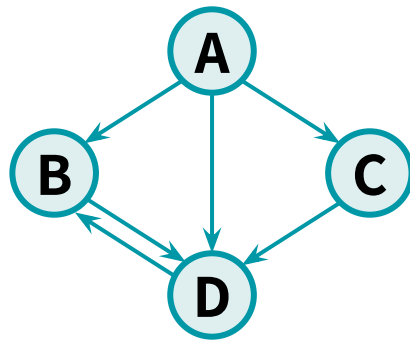


$V = \{A, B, C, D\}$

$E = \{ \{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{C, D\} \}$

## DIRECTED GRAPHS

A directed graph has  
a set of vertices ( $V$ ) & a set of **DIRECTED** edges ( $E$ )



Formally,  
 $G = (V, E)$

$V = \{A, B, C, D\}$

$E = \{ [A, B], [A, C], [A, D], [B, D], [C, D], [D, B] \}$

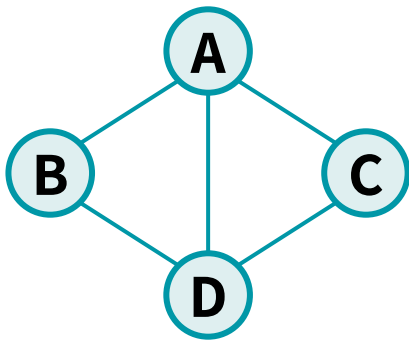
# 2 KINDS OF GRAPHS

We'll deal with both kinds of graphs in this class.

## UNDIRECTED GRAPHS

An undirected graph has  
a set of vertices ( $V$ ) & a set of edges ( $E$ )

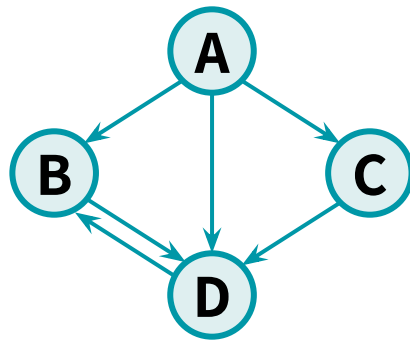
Formally,  
 $G = (V, E)$



The **degree** of vertex D is 3  
Vertex D's **neighbors** are A, B, and C

## DIRECTED GRAPHS

A directed graph has  
a set of vertices ( $V$ ) & a set of **DIRECTED** edges ( $E$ )



Formally,  
 $G = (V, E)$

The **in-degree** of vertex D is 3. The **out-degree** of vertex D is 1.  
Vertex D's **incoming neighbors** are A, B, & C  
Vertex D's **outgoing neighbor** is B

# 2 KINDS OF GRAPHS

We'll deal with both kinds of graphs in this class.

## UNDIRECTED GRAPHS

a set

Formally,  
 $G = (V, E)$

Today, we're only working with **unweighted** graphs. These are graphs where edges aren't assigned weights, or all edges are assumed to have the same weight.



The **degree** of vertex D is 3  
Vertex D's **neighbors** are A, B, and C

## DIRECTED GRAPHS

edges (E)

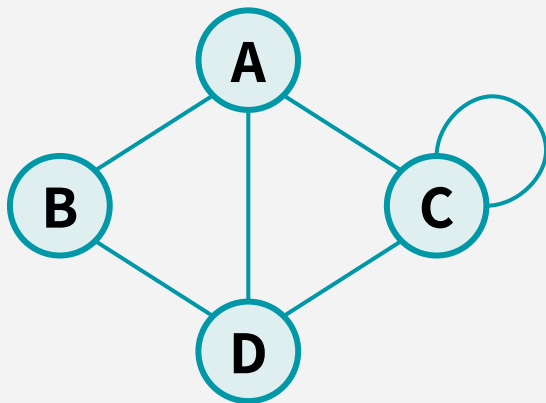
Formally,  
 $G = (V, E)$



The **in-degree** of vertex D is 3. The **out-degree** of vertex D is 1.  
Vertex D's **incoming neighbors** are A, B, & C  
Vertex D's **outgoing neighbor** is B

# GRAPH REPRESENTATIONS

## OPTION 1: **ADJACENCY MATRIX**

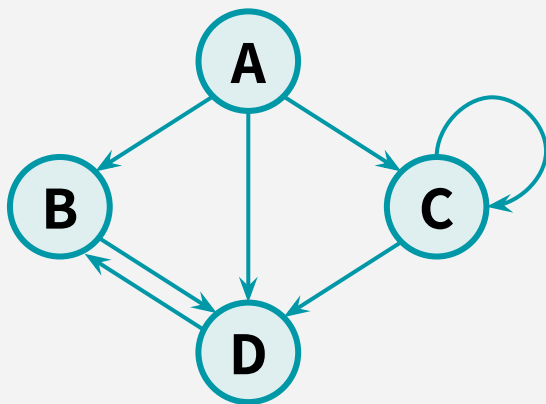


(An undirected graph)

		(destination)			
		A	B	C	D
(source)	A	0	1	1	1
	B	1	0	0	1
	C	1	0	1	1
	D	1	1	1	0

# GRAPH REPRESENTATIONS

## OPTION 1: **ADJACENCY MATRIX**

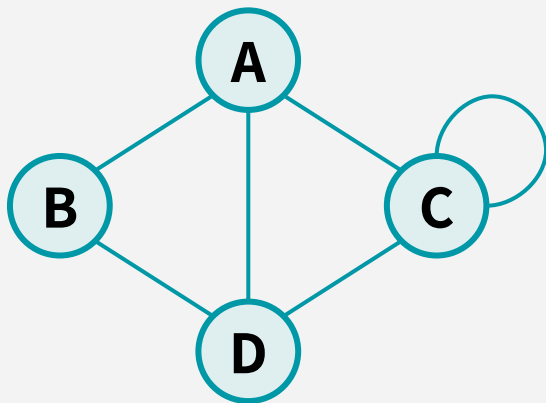


(A directed graph)

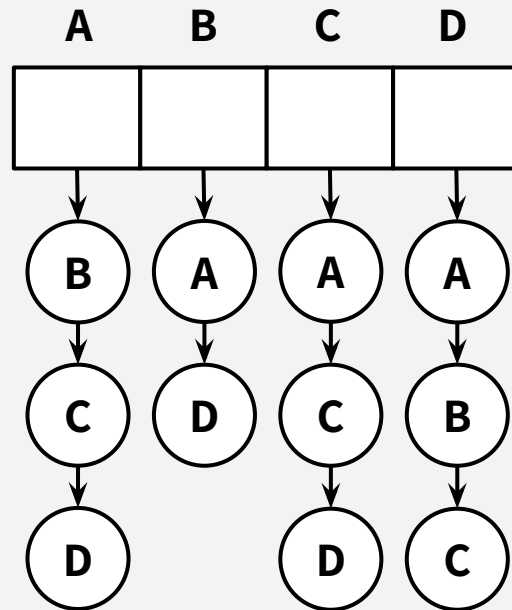
		(destination)			
		A	B	C	D
(source)	A	0	1	1	1
	B	0	0	0	1
	C	0	0	1	1
	D	0	1	0	0

# GRAPH REPRESENTATIONS

## OPTION 2: **ADJACENCY LISTS**



(An undirected graph)

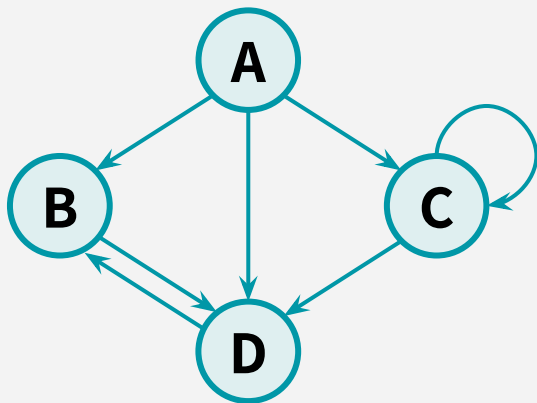


Each list stores a  
node's neighbors

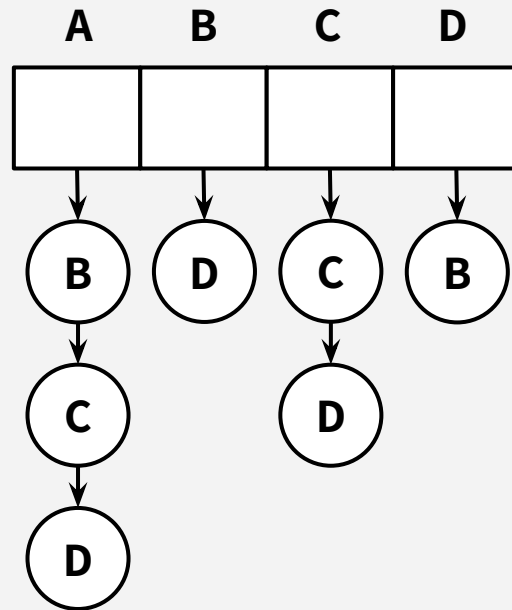


# GRAPH REPRESENTATIONS

## OPTION 2: **ADJACENCY LISTS**



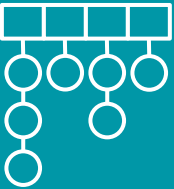
(A directed graph)



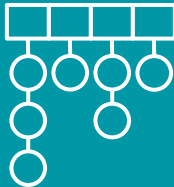
*Tracks outgoing neighbors.*

*(You could also do the same for incoming neighbors as well)*

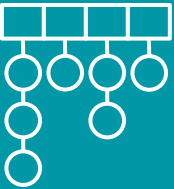
# GRAPH REPRESENTATIONS

For a graph $G = (V, E)$ where $ V  = \mathbf{n}$ , and $ E  = \mathbf{m}$	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
<b>EDGE MEMBERSHIP</b> Is $e = \{v, w\}$ in $E$ ?		
<b>NEIGHBOR QUERY</b> Give me $v$ 's neighbors		
<b>SPACE REQUIREMENTS</b>		

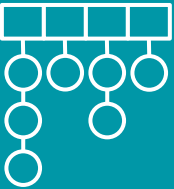
# GRAPH REPRESENTATIONS

<p>For a graph <math>G = (V, E)</math>  where <math> V  = \mathbf{n}</math>, and <math> E  = \mathbf{m}</math></p>	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
<p><b>EDGE MEMBERSHIP</b>  Is <math>e = \{v, w\}</math> in <math>E</math>?</p>	<p><b><math>O(1)</math></b></p>	
<p><b>NEIGHBOR QUERY</b>  Give me <math>v</math>'s neighbors</p>		
<p><b>SPACE REQUIREMENTS</b></p>		

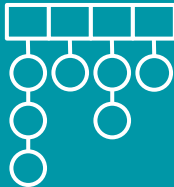
# GRAPH REPRESENTATIONS

For a graph $G = (V, E)$ where $ V  = \mathbf{n}$ , and $ E  = \mathbf{m}$	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
<b>EDGE MEMBERSHIP</b> Is $e = \{v, w\}$ in $E$ ?	<b><math>O(1)</math></b>	
<b>NEIGHBOR QUERY</b> Give me $v$ 's neighbors	<b><math>O(n)</math></b>	
<b>SPACE REQUIREMENTS</b>		

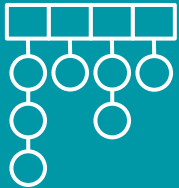
# GRAPH REPRESENTATIONS

For a graph $G = (V, E)$ where $ V  = \mathbf{n}$ , and $ E  = \mathbf{m}$	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
<b>EDGE MEMBERSHIP</b> Is $e = \{v, w\}$ in $E$ ?	<b><math>O(1)</math></b>	
<b>NEIGHBOR QUERY</b> Give me $v$ 's neighbors	<b><math>O(n)</math></b>	
<b>SPACE REQUIREMENTS</b>	<b><math>O(n^2)</math></b>	

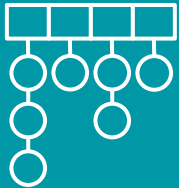
# GRAPH REPRESENTATIONS

<p>For a graph <math>G = (V, E)</math>          where <math> V  = \mathbf{n}</math>, and <math> E  = \mathbf{m}</math></p>	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
<p><b>EDGE MEMBERSHIP</b>          Is <math>e = \{v, w\}</math> in <math>E</math>?</p>	<p><b><math>O(1)</math></b></p>	<p><b><math>O(\deg(v))</math> or  <math>O(\deg(w))</math></b></p>
<p><b>NEIGHBOR QUERY</b>          Give me <math>v</math>'s neighbors</p>	<p><b><math>O(n)</math></b></p>	
<p><b>SPACE REQUIREMENTS</b></p>	<p><b><math>O(n^2)</math></b></p>	

# GRAPH REPRESENTATIONS

<p>For a graph <math>G = (V, E)</math>  where <math> V  = \mathbf{n}</math>, and <math> E  = \mathbf{m}</math></p>	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
<p><b>EDGE MEMBERSHIP</b>  Is <math>e = \{v, w\}</math> in <math>E</math>?</p>	<p><b><math>O(1)</math></b></p>	<p><b><math>O(\deg(v))</math> or  <math>O(\deg(w))</math></b></p>
<p><b>NEIGHBOR QUERY</b>  Give me <math>v</math>'s neighbors</p>	<p><b><math>O(n)</math></b></p>	<p><b><math>O(\deg(v))</math></b></p>
<p><b>SPACE REQUIREMENTS</b></p>	<p><b><math>O(n^2)</math></b></p>	

# GRAPH REPRESENTATIONS

<p>For a graph <math>G = (V, E)</math>  where <math> V  = \mathbf{n}</math>, and <math> E  = \mathbf{m}</math></p>	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
<p><b>EDGE MEMBERSHIP</b>  Is <math>e = \{v, w\}</math> in <math>E</math>?</p>	<p><b><math>O(1)</math></b></p>	<p><b><math>O(\deg(v))</math> or  <math>O(\deg(w))</math></b></p>
<p><b>NEIGHBOR QUERY</b>  Give me <math>v</math>'s neighbors</p>	<p><b><math>O(n)</math></b></p>	<p><b><math>O(\deg(v))</math></b></p>
<p><b>SPACE REQUIREMENTS</b></p>	<p><b><math>O(n^2)</math></b></p>	<p><b><math>O(n + m)</math></b></p>



# GRAPH REPRESENTATIONS

For a graph $G = (V, E)$ where $ V  = \mathbf{n}$ , and $ E  = \mathbf{m}$	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
<b>EDGE MEMBERSHIP</b> Is $e = \{v, w\}$ in $E$ ?	<b><math>O(1)</math></b>	<b><math>O(\deg(v))</math> or <math>O(\deg(w))</math></b>
<b>NEIGHBOR QUERY</b> Give me $v$ 's neighbors	<b><math>O(n)</math></b>	<b><math>O(\deg(v))</math></b>
<b>SPACE REQUIREMENTS</b>	<b><math>O(n^2)</math></b>	<b><math>O(n + m)</math></b>

Generally, better for  
sparse graphs  
(where  $m \ll n^2$ ).

**We'll assume this  
representation,  
unless otherwise  
stated.**



سوال؟

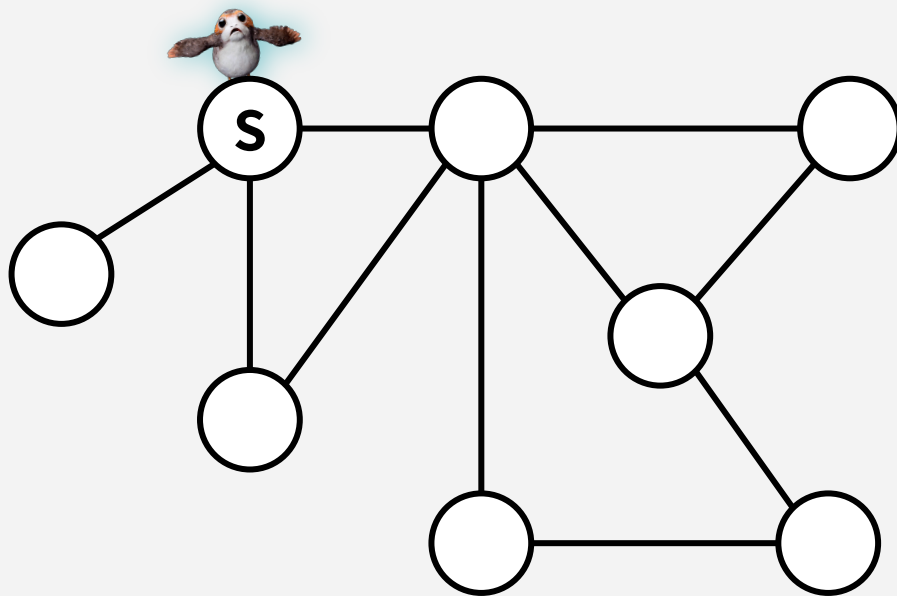
# جستجوی عمق اول (BFS)

**یک روش پیمایش گراف**

# BREADTH-FIRST SEARCH

## **An analogy:**

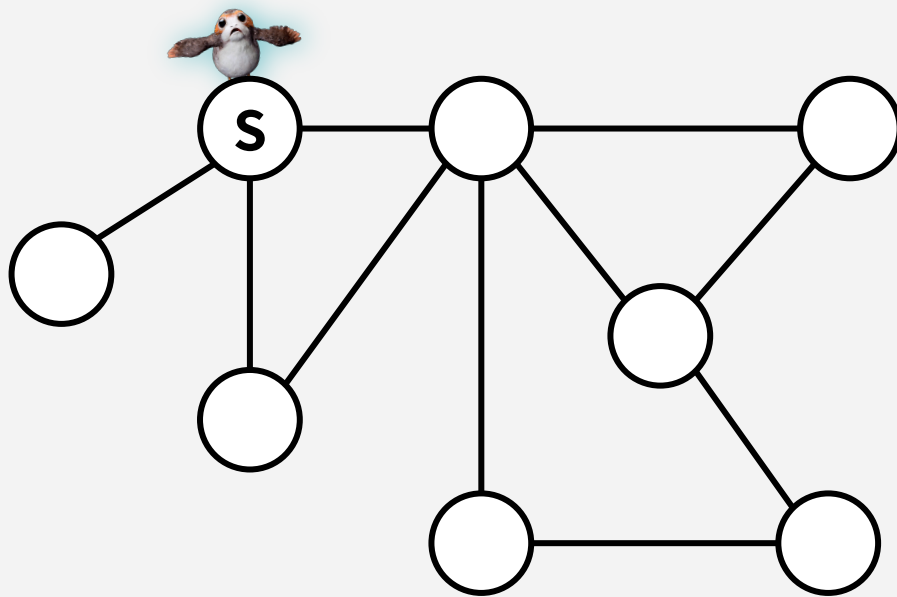
A bird is exploring a labyrinth from above (with a bird's eye view)



# BREADTH-FIRST SEARCH

## An analogy:

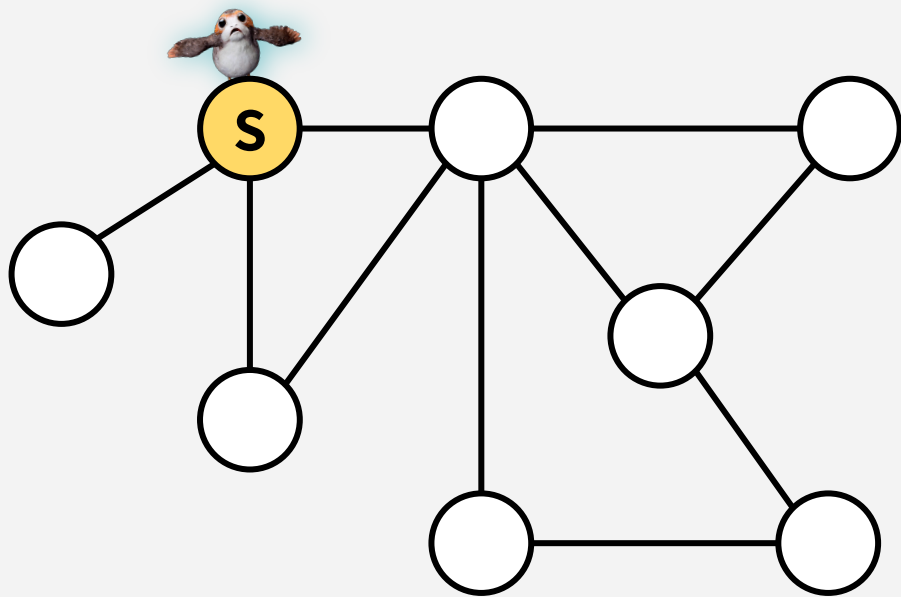
A bird is exploring a labyrinth from above (with a bird's eye view)



# BREADTH-FIRST SEARCH

## An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)

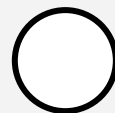
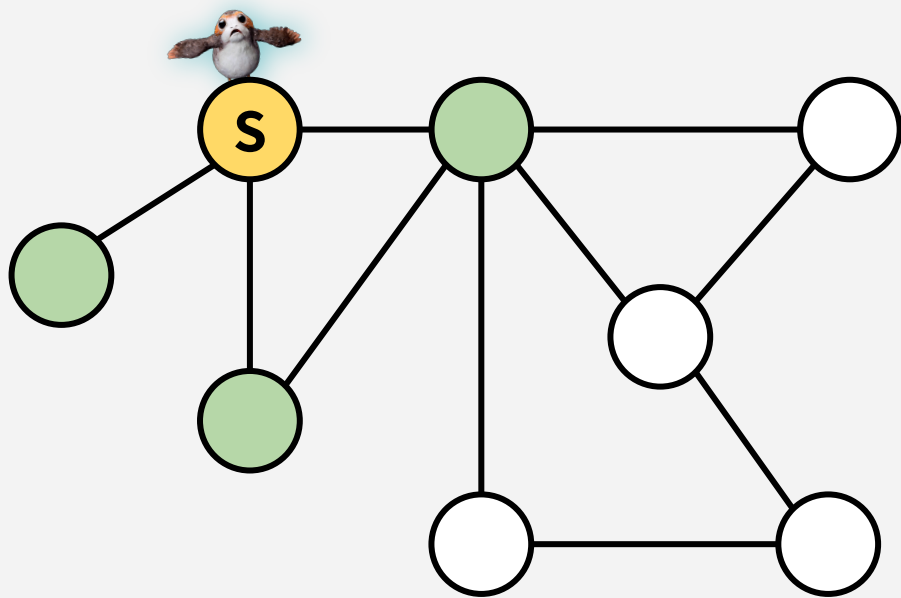


-  unvisited
-  **Layer 0:** reachable in 0 steps
-  **Layer 1:** reachable in 1 step
-  **Layer 2:** reachable in 2 steps
-  **Layer 3:** reachable in 3 steps

# BREADTH-FIRST SEARCH

## An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)



unvisited



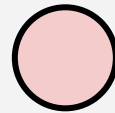
**Layer 0:** reachable  
in 0 steps



**Layer 1:** reachable  
in 1 step



**Layer 2:** reachable  
in 2 steps

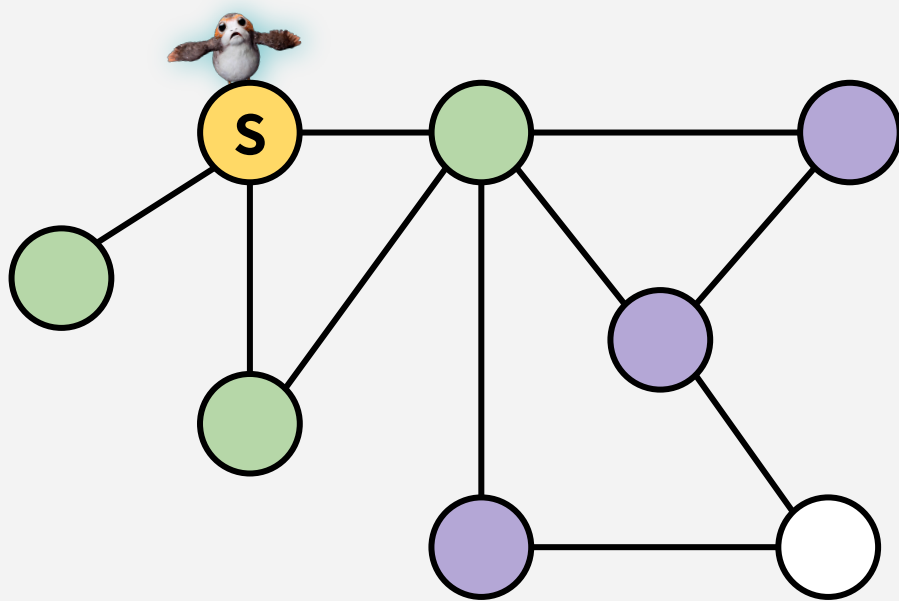


**Layer 3:** reachable  
in 3 steps

# BREADTH-FIRST SEARCH

## An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)



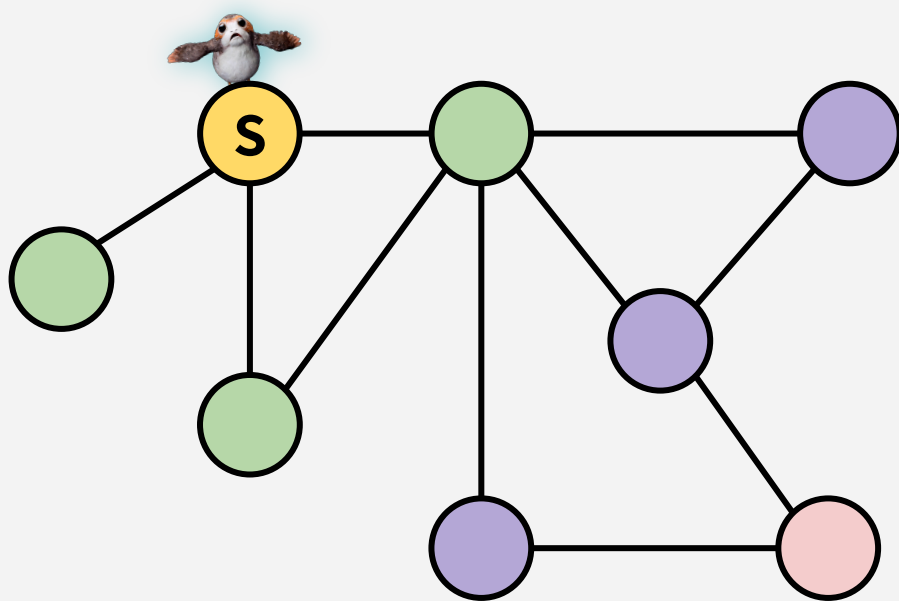
-  unvisited
-  **Layer 0:** reachable in 0 steps
-  **Layer 1:** reachable in 1 step
-  **Layer 2:** reachable in 2 steps
-  **Layer 3:** reachable in 3 steps



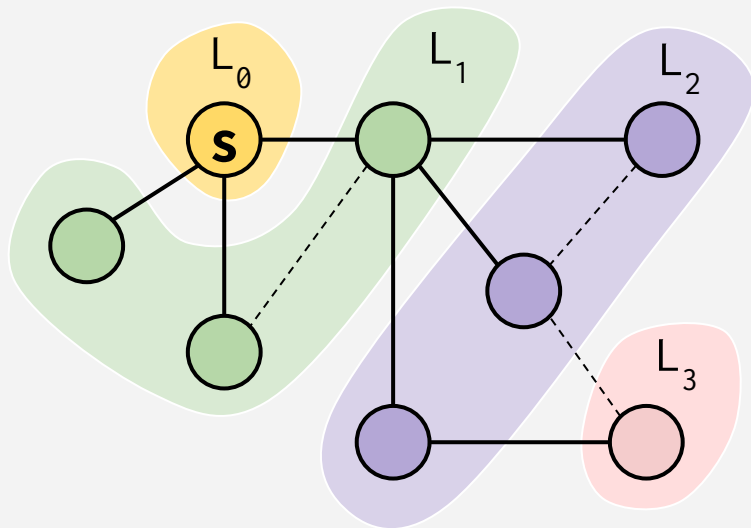
# BREADTH-FIRST SEARCH

## An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)



# BREADTH-FIRST SEARCH



$L_i$  = The set of nodes we can reach in  $i$  steps from  $s$

**BFS(s):**

Set  $L_i = []$  for  $i = 0, \dots, n-1$

$L_0 = s$

for  $i = 0, \dots, n-1$ :

for  $u$  in  $L_i$ :

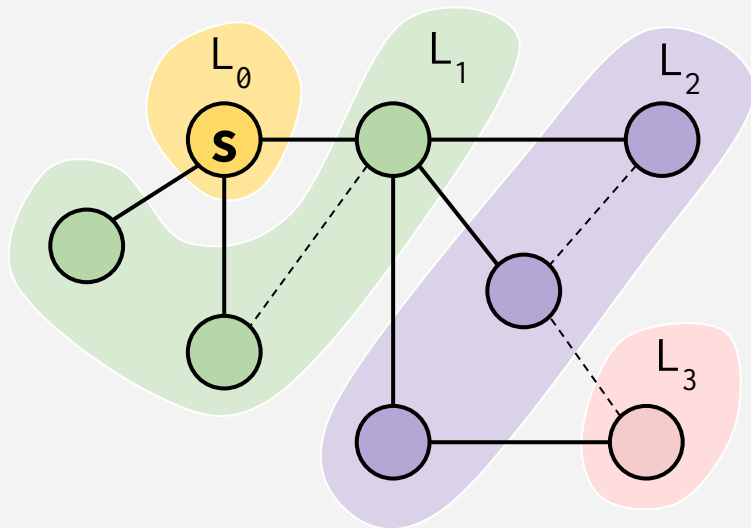
for  $v$  in  $u$ .neighbors:

if  $v$  not yet visited:

mark  $v$  as visited

add  $v$  to  $L_{i+1}$

# BREADTH-FIRST SEARCH



$L_i$  = The set of nodes we can reach in  $i$  steps from  $s$

**BFS(s):**

Set  $L_i = []$  for  $i = 0, \dots, n-1$

$L_0 = s$

for  $i = 0, \dots, n-1$ :

for  $u$  in  $L_i$ :

for  $v$  in  $u$ .neighbors:

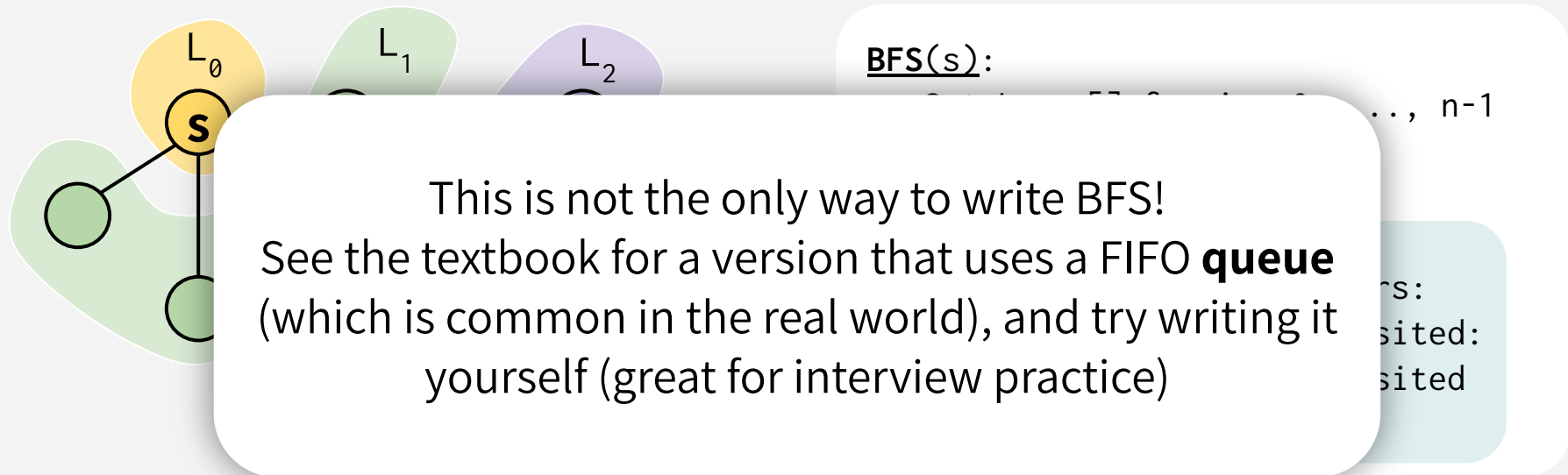
if  $v$  not yet visited:

mark  $v$  as visited

add  $v$  to  $L_{i+1}$

Go through all nodes in  $L_i$  and add their  
unvisited neighbors to  $L_{i+1}$

# BREADTH-FIRST SEARCH



$L_i$  = The set of nodes we can reach in  $i$  steps from  $s$

Go through all nodes in  $L_i$  and add their  
unvisited neighbors to  $L_{i+1}$

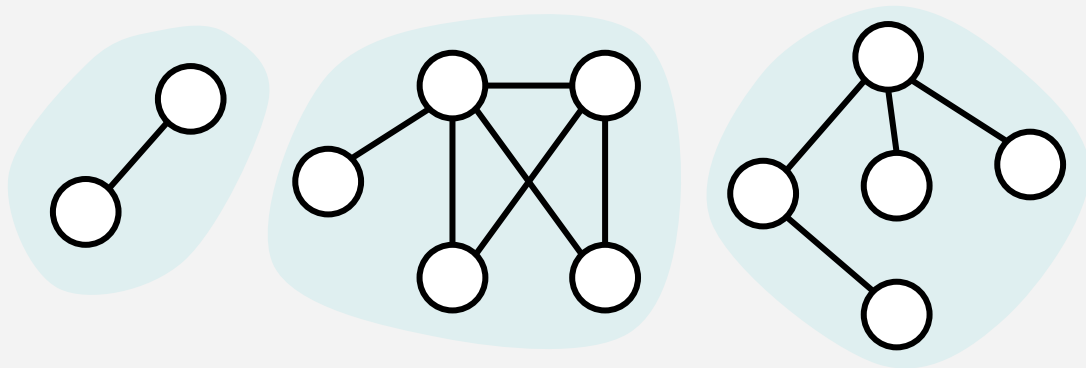
# BREADTH-FIRST SEARCH

**BFS finds all the nodes reachable from the starting point!**

# BREADTH-FIRST SEARCH

**BFS finds all the nodes reachable from the starting point!**

In undirected graphs, this is equivalent to finding the node's **connected component**.



# BREADTH-FIRST SEARCH: RUNTIME

To explore a graph's  **$i^{\text{th}}$  connected component** ( $n_i$  nodes,  $m_i$  edges):

We visit each vertex in the CC exactly once (“visit” = grab from its  $L_i$ ).

# BREADTH-FIRST SEARCH: RUNTIME

To explore a graph's  **$i^{\text{th}}$  connected component** ( $n_i$  nodes,  $m_i$  edges):

We visit each vertex in the CC exactly once (“visit” = grab from its  $L_i$ ).

At each vertex  $v$ , we:

- Do some bookkeeping:  **$O(1)$**
- Loop over  $v$ 's neighbors & check if they are visited (& then potentially mark the neighbor & place in  $L_{i+1}$ ):  $O(1)$  per neighbor  $\rightarrow$   **$O(\deg(v))$**  total.



# BREADTH-FIRST SEARCH: RUNTIME

To explore a graph's  **$i^{\text{th}}$  connected component** ( $n_i$  nodes,  $m_i$  edges):

We visit each vertex in the CC exactly once (“visit” = grab from its  $L_i$ ).

At each vertex  $v$ , we:

- Do some bookkeeping:  **$O(1)$**
- Loop over  $v$ 's neighbors & check if they are visited (& then potentially mark the neighbor & place in  $L_{i+1}$ ):  $O(1)$  per neighbor  $\rightarrow$   **$O(\deg(v))$**  total.

$$\textbf{Total: } \sum_v O(\deg(v)) + \sum_v O(1) = \textbf{O}(m_i + n_i)$$

# BREADTH-FIRST SEARCH: RUNTIME

To explore **the entire graph** (n nodes, m edges):

A graph might have multiple connected components! To **explore the whole graph**, we would call our BFS routine once for each connected component (note that each vertex and each edge participates in exactly one connected component). The combined running time would be:

$$O(\sum_i m_i + \sum_i n_i) = \mathbf{O(m + n)}$$

# BREADTH-FIRST SEARCH

## Why is it called breadth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

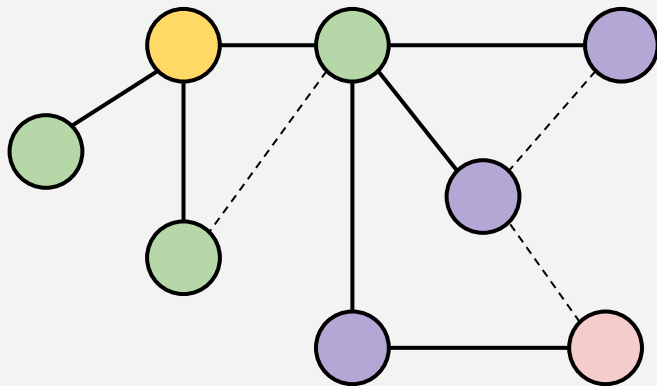
We go as “broadly” as we can when building each layer of the tree

# BREADTH-FIRST SEARCH

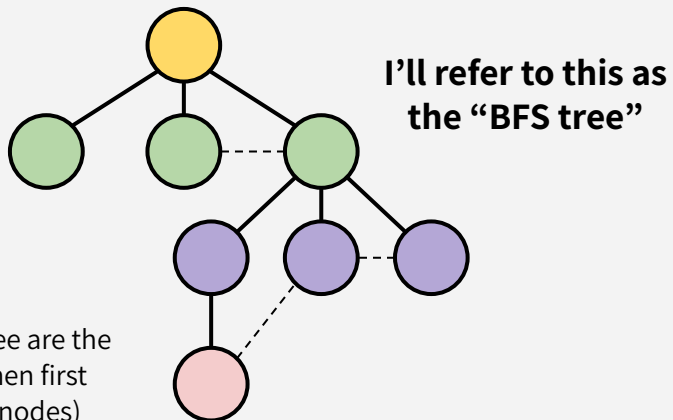
## Why is it called breadth-first?

We are implicitly building a **tree**!  
(It's a tree because we never revisit a node)

We go as “broadly” as we can when building each layer of the tree



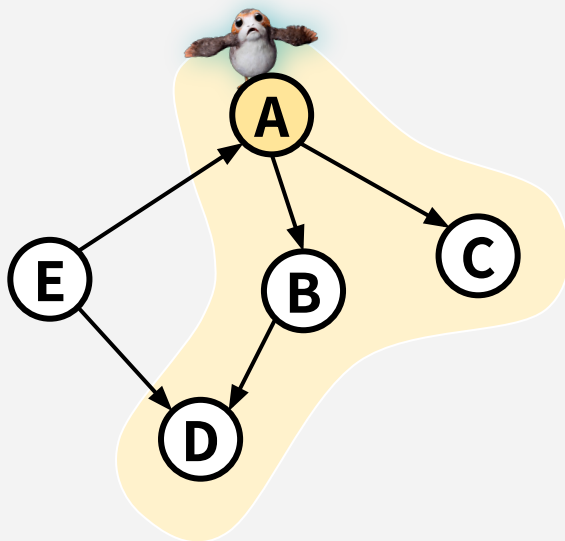
(Edges in the BFS tree are the ones traversed when first finding unvisited nodes)



# BREADTH-FIRST SEARCH

## BFS works fine on directed graphs too!

From a start node  $x$ , BFS would find all nodes **reachable** from  $x$ .  
(In directed graphs, “connected component” isn’t as well defined... more on that later!)



### Verify this on your own:

running BFS from A  
would still find all nodes  
reachable from A (E isn't  
reachable from A in this  
directed graph).

# BREADTH-FIRST SEARCH

## **What are some applications of BFS?**

Finding a node's connected component (just run BFS)!  
*(or in directed graphs, finding reachable nodes from a starting node)*

Single-source shortest paths

Testing bipartiteness

And more...



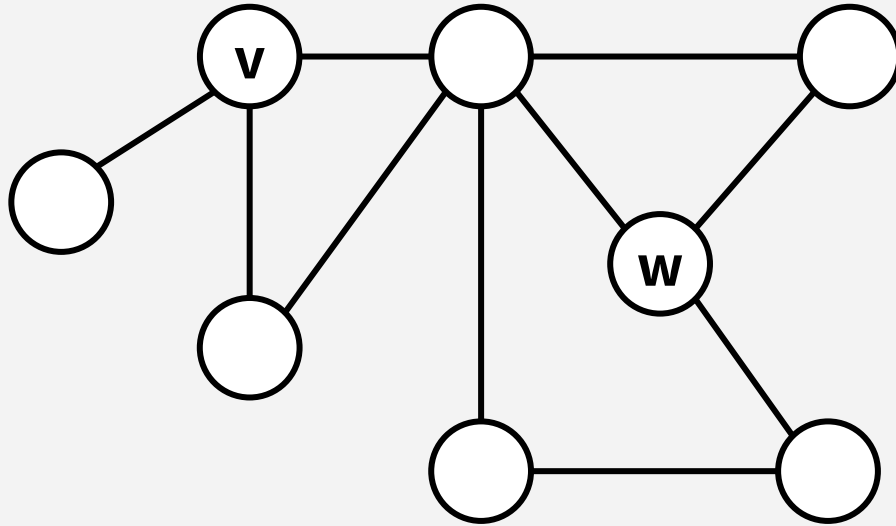
سوال؟

پیدا کردن کوتاه ترین مسیر با  
جستجوی عمق اول



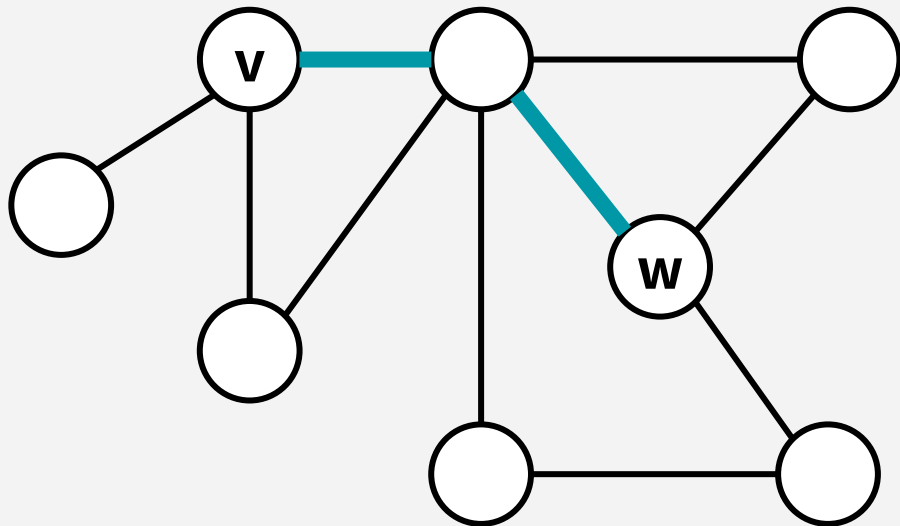
# SHORTEST PATH: THE TASK

**How long is the shortest path between vertices  $v$  and  $w$ ?**



# SHORTEST PATH: THE TASK

**How long is the shortest path between vertices  $v$  and  $w$ ?**

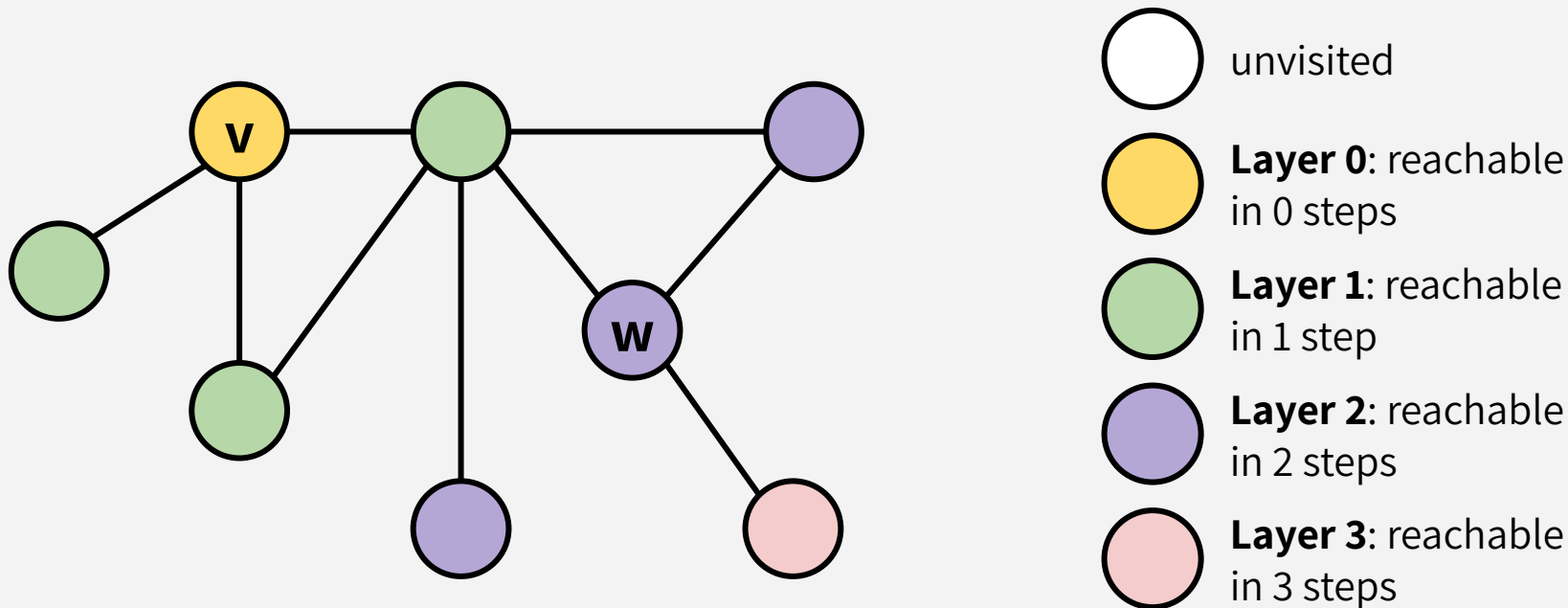


From visually inspecting the graph, we can see that the shortest path from  $v$  to  $w$  is 2 (there are 2 edges on that path)!

There are paths of length 3, 4, or 5 as well, but we can't do any better than 2.

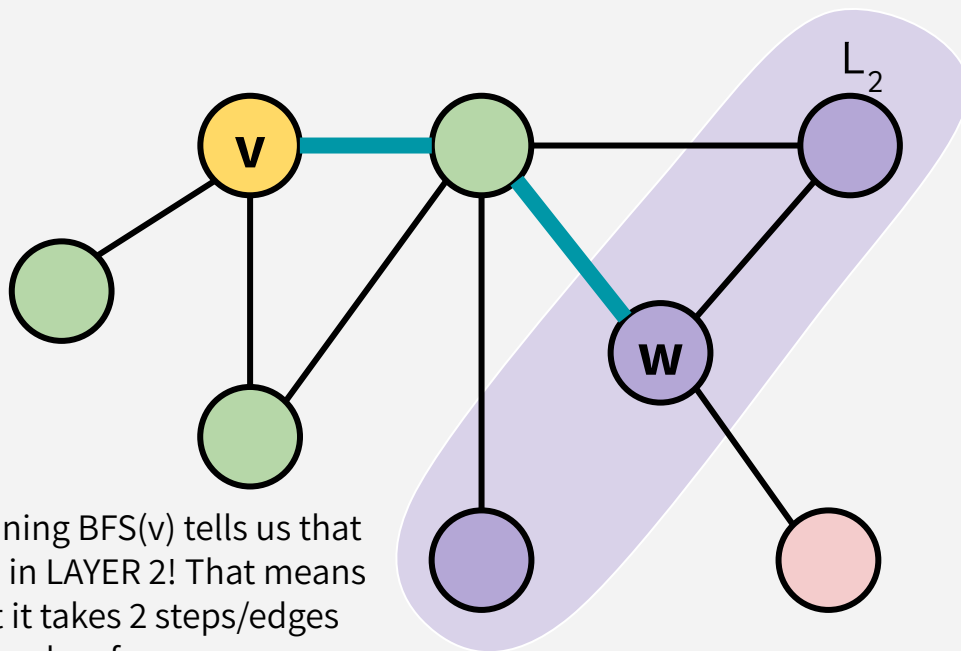
# SHORTEST PATH: THE TASK

How long is the shortest path between vertices  $v$  and  $w$ ?



# SHORTEST PATH: THE TASK

How long is the shortest path between vertices **v** and **w**?

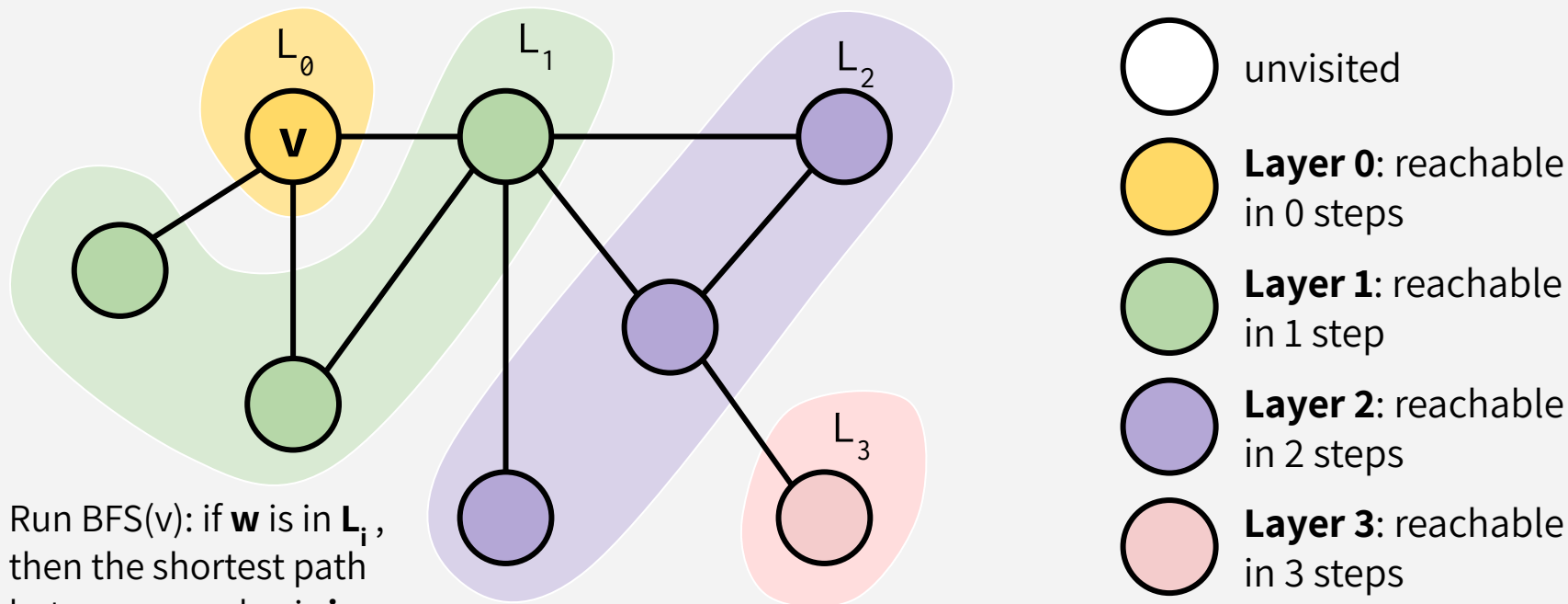


Running BFS(**v**) tells us that **w** is in LAYER 2! That means that it takes 2 steps/edges to reach **w** from **v**.

-  unvisited
-  **Layer 0:** reachable in 0 steps
-  **Layer 1:** reachable in 1 step
-  **Layer 2:** reachable in 2 steps
-  **Layer 3:** reachable in 3 steps

# SINGLE-SOURCE SHORTEST PATH

How long is the shortest path between vertices  $v$  & *all other vertices*  $w$ ?



Run  $\text{BFS}(v)$ : if  $w$  is in  $L_i$ ,  
then the shortest path  
between  $v$  and  $w$  is  $i$

# SINGLE-SOURCE SHORTEST PATH

**How long is the shortest path between vertices  $v$  & *all other vertices*  $w$ ?**

**findAllDistances( $v$ ):**

perform BFS( $v$ )  $\rightarrow$  gives us all  $L_i$

for all  $w$  in  $V$ :

$d[w] = \infty$

for each  $L_i$ :

for all  $w$  in  $L_i$ :

$d[w] = i$

# SINGLE-SOURCE SHORTEST PATH

**How long is the shortest path between vertices  $v$  & *all other vertices*  $w$ ?**

**findAllDistances**( $v$ ):

perform BFS( $v$ )  $\rightarrow$  gives us all  $L_i$

for all  $w$  in  $V$ :

$d[w] = \infty$

for each  $L_i$ :

for all  $w$  in  $L_i$ :

$d[w] = i$

This is something  
you can repurpose  
on your problem set  
or quiz questions:  
**you may assume  
that running BFS( $v$ )  
also provides you  
all the levels  $L_i$**

**Runtime:  $O(m+n)$**



سوال؟

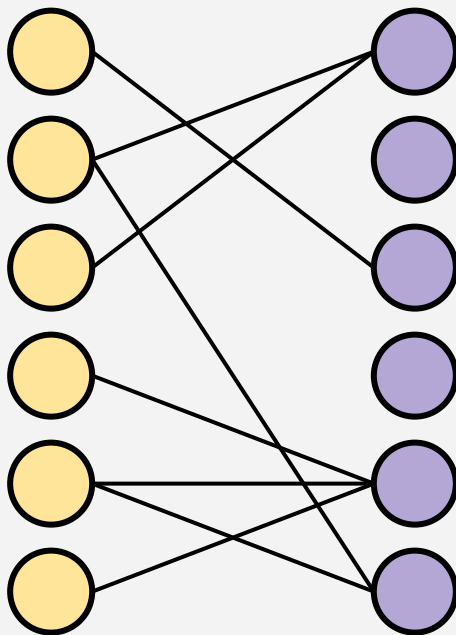


# آزمایش دوبخشی بودن گراف

**استفاده از جستجوی عمق اول برای آزمایش دوبخشی بودن گراف**

# BIPARTITE GRAPHS

A graph is **bipartite** iff there exists a 2-coloring such that there are no edges between same-colored vertices

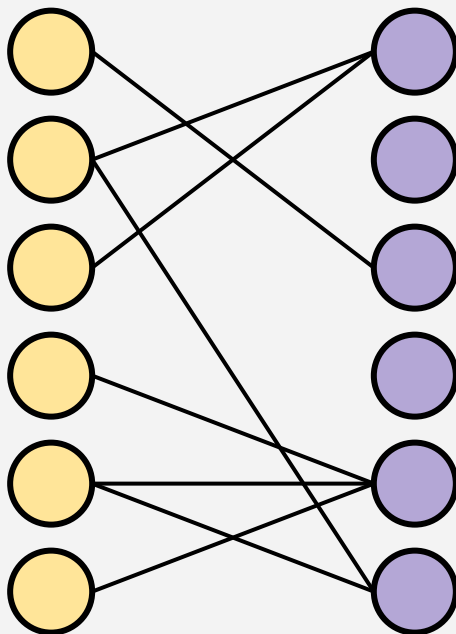


# BIPARTITE GRAPHS

A graph is **bipartite** iff there exists a 2-coloring such that there are no edges between same-colored vertices

Example 1:

You're planning a cross-grade buddy system for 3rd and 4th graders, and you polled everyone's preferences for buddies. Can you verify that no students were listing someone from their same grade as one of their top choices?

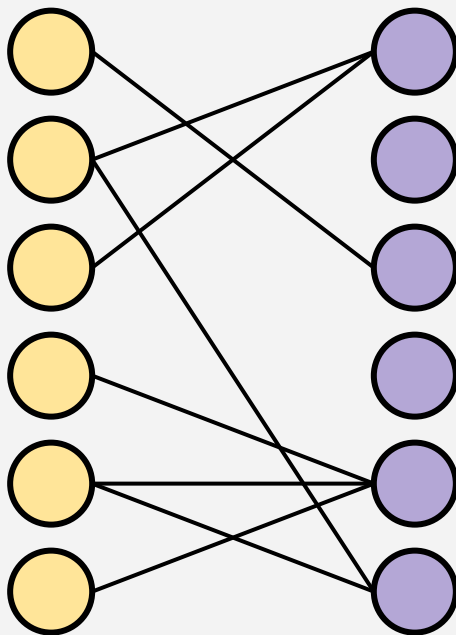


# BIPARTITE GRAPHS

A graph is **bipartite** iff there exists a 2-coloring such that there are no edges between same-colored vertices

Example 1:

You're planning a cross-grade buddy system for 3rd and 4th graders, and you polled everyone's preferences for buddies. Can you verify that no students were listing someone from their same grade as one of their top choices?

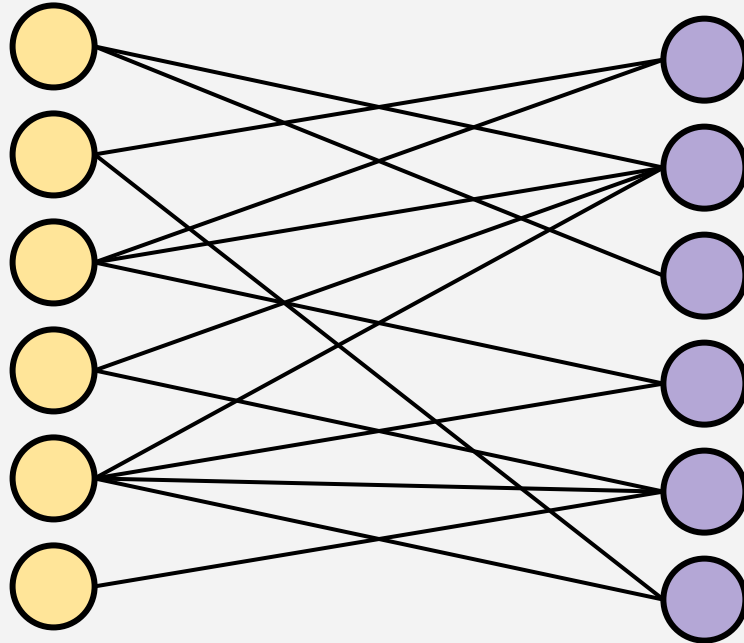


Example 2:

You have a bunch of fish and two fish tanks; some pairs of fish will fight if they're in the same tank. Can you separate the fish so that there's no fighting?

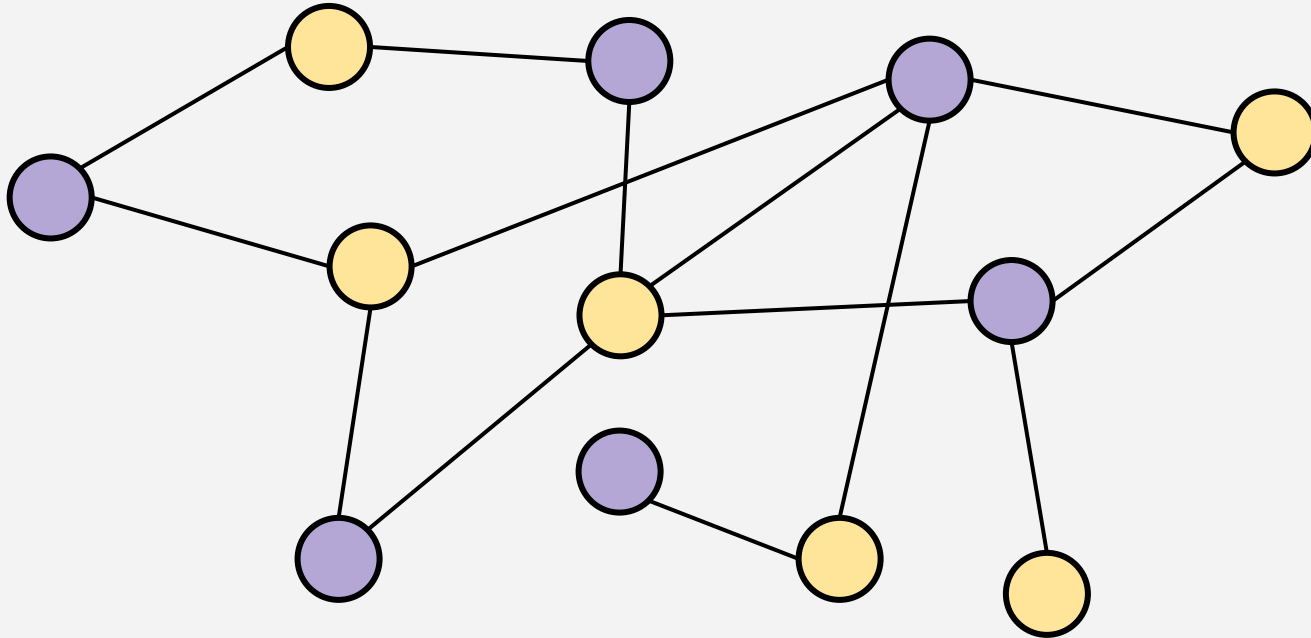
# BIPARTITE GRAPHS

Is this graph bipartite?



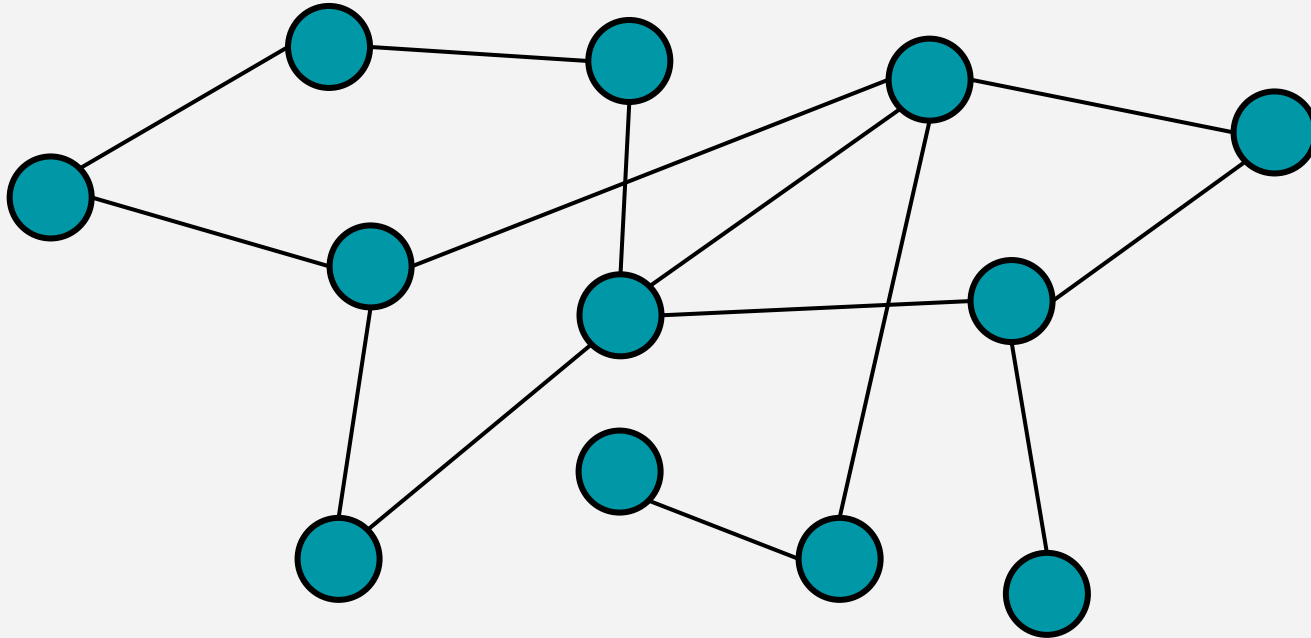
# BIPARTITE GRAPHS

How about this one?



# BIPARTITE GRAPHS

How about this one?



# BIPARTITE GRAPHS

## **Application of BFS:**

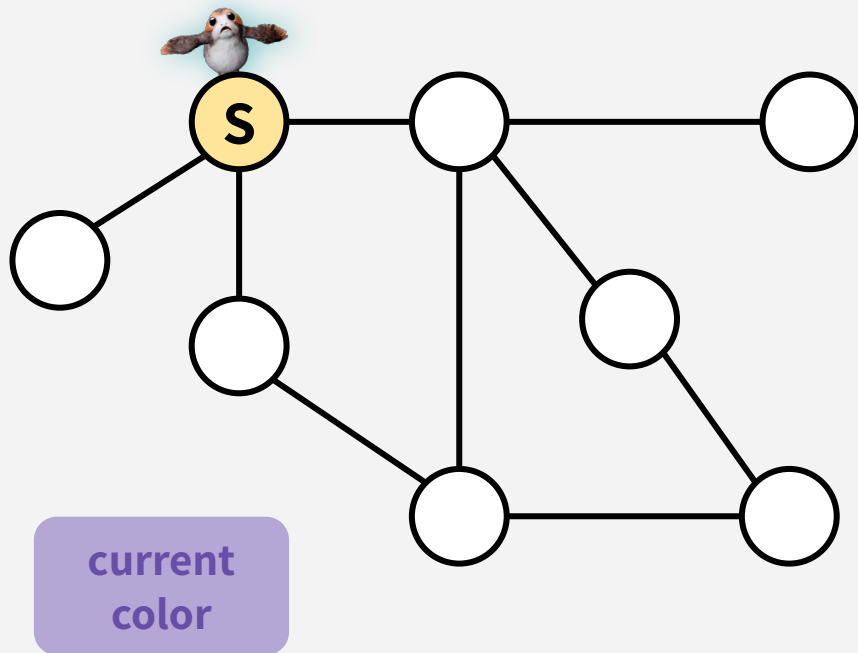
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

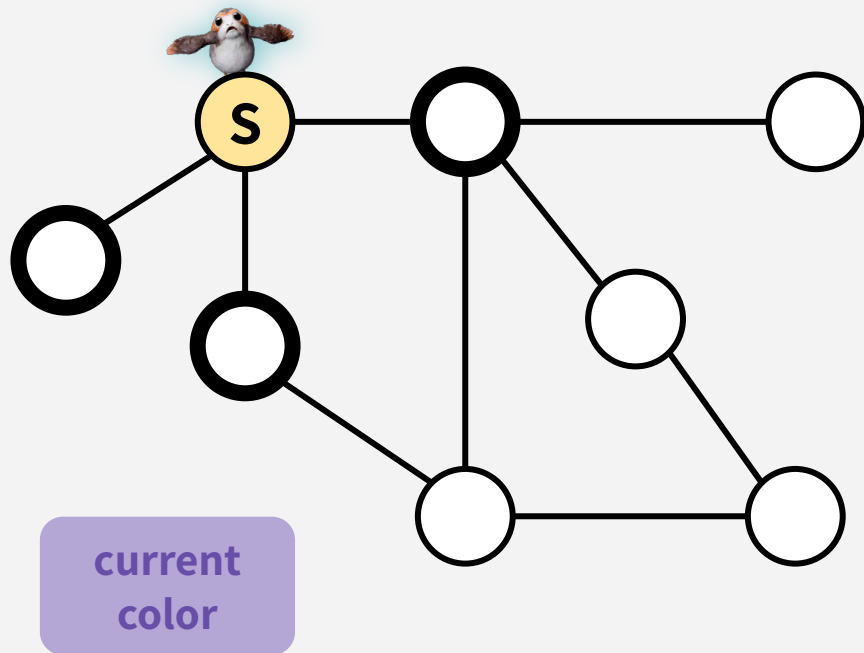
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

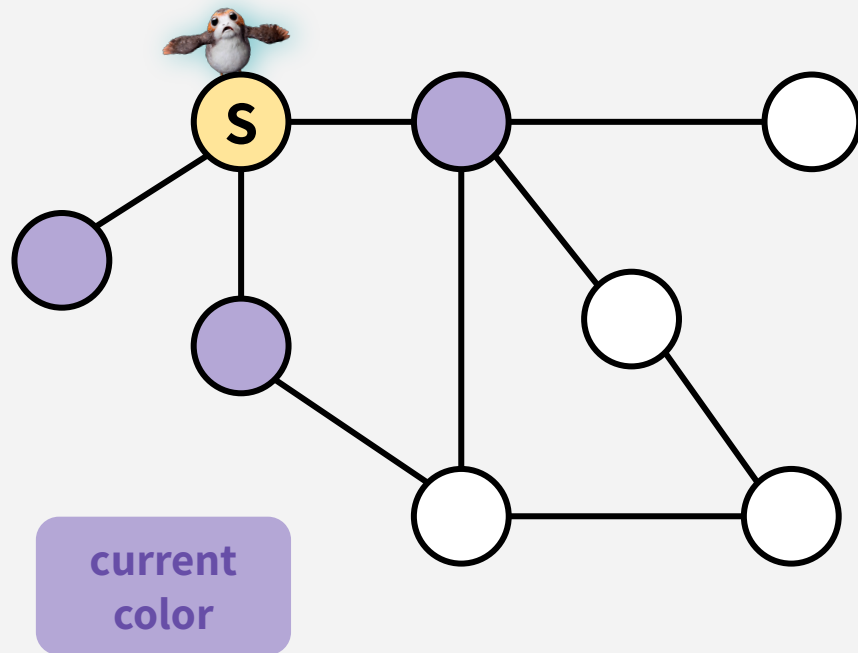
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

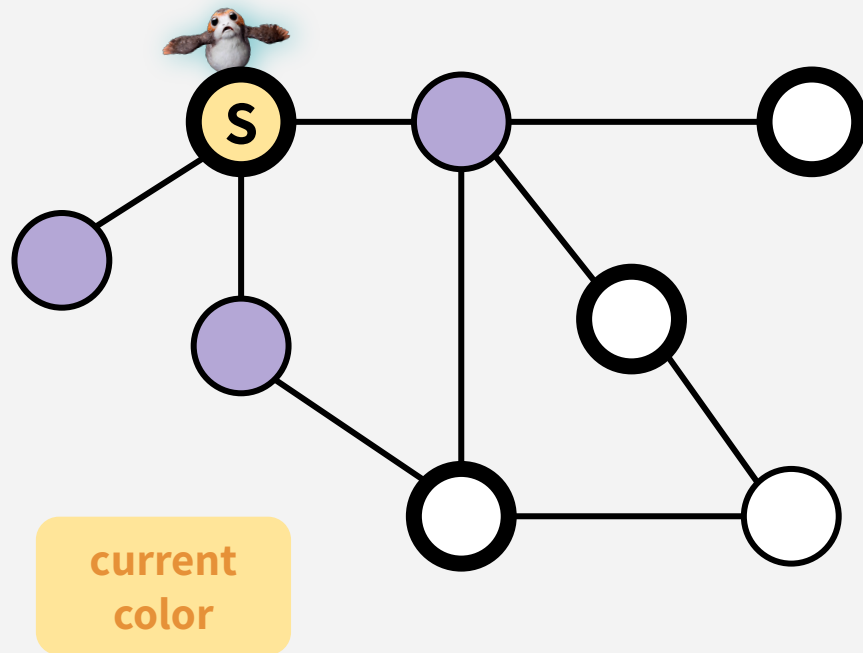
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

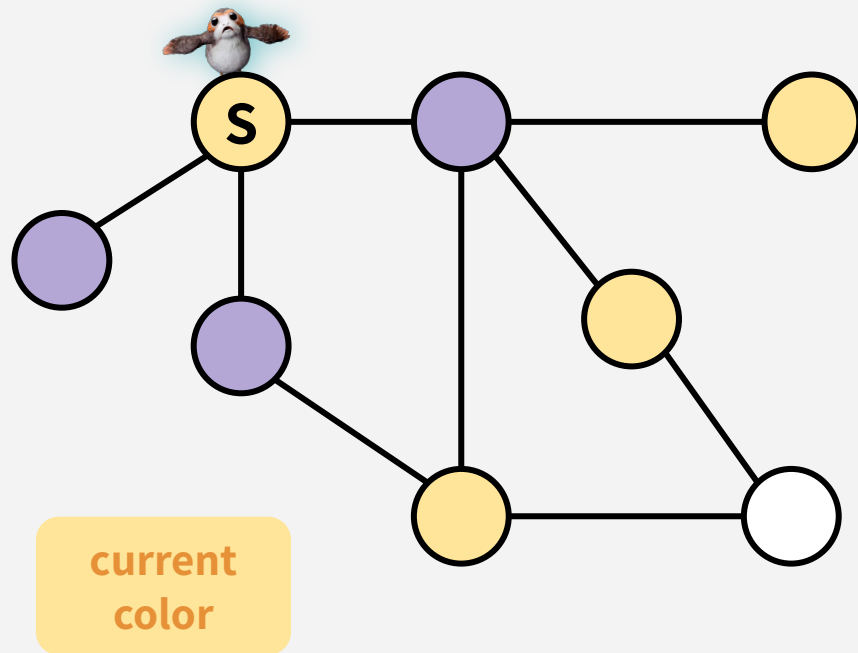
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

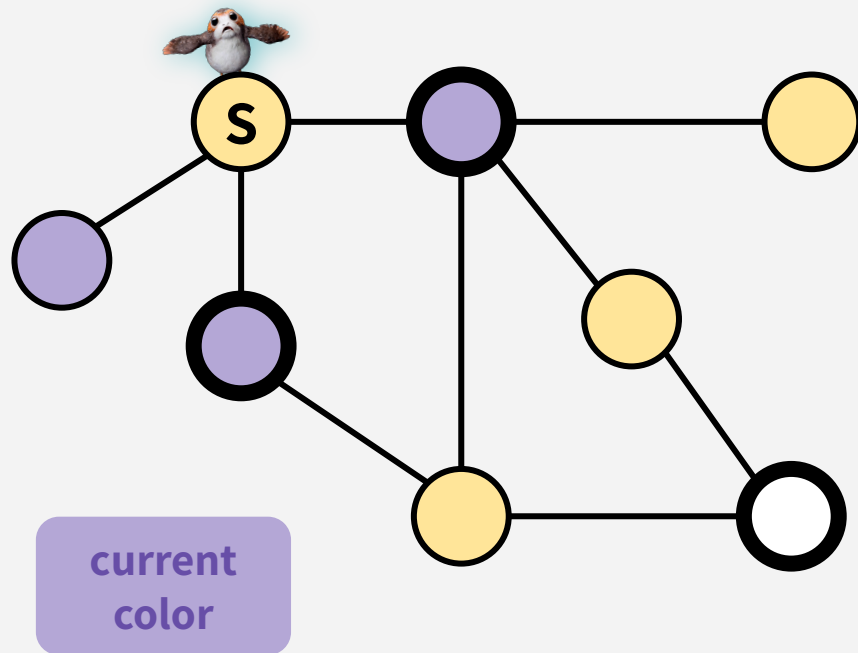
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

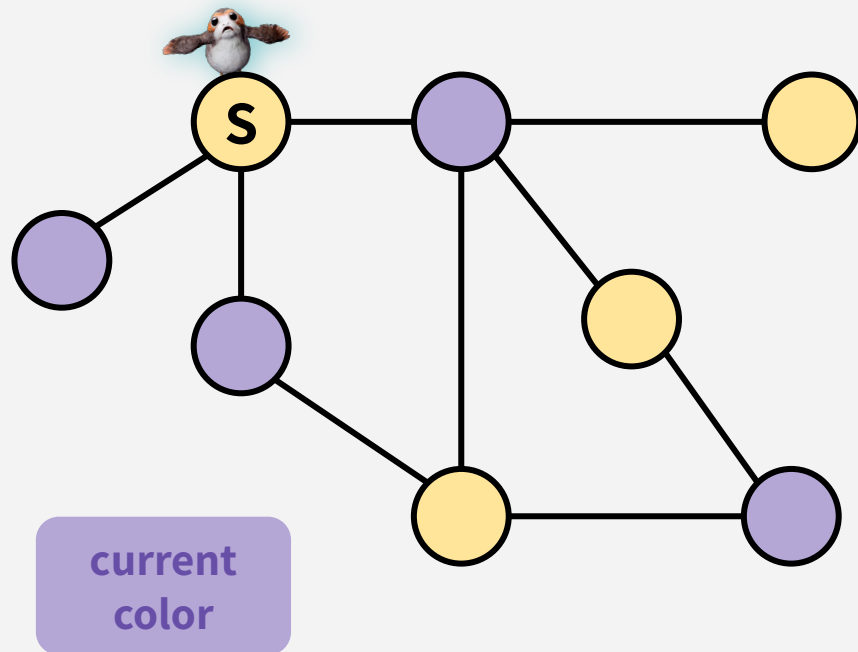
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

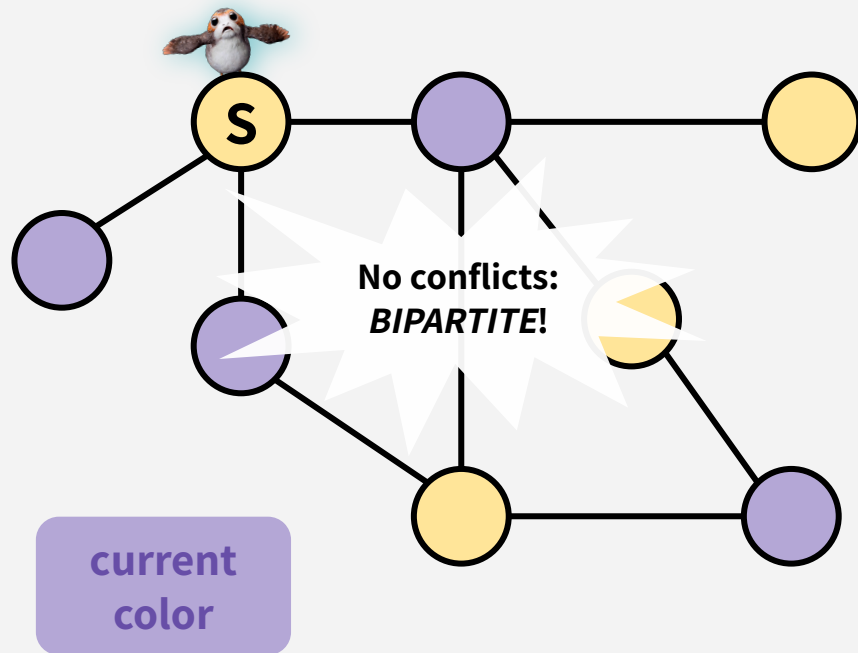
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!

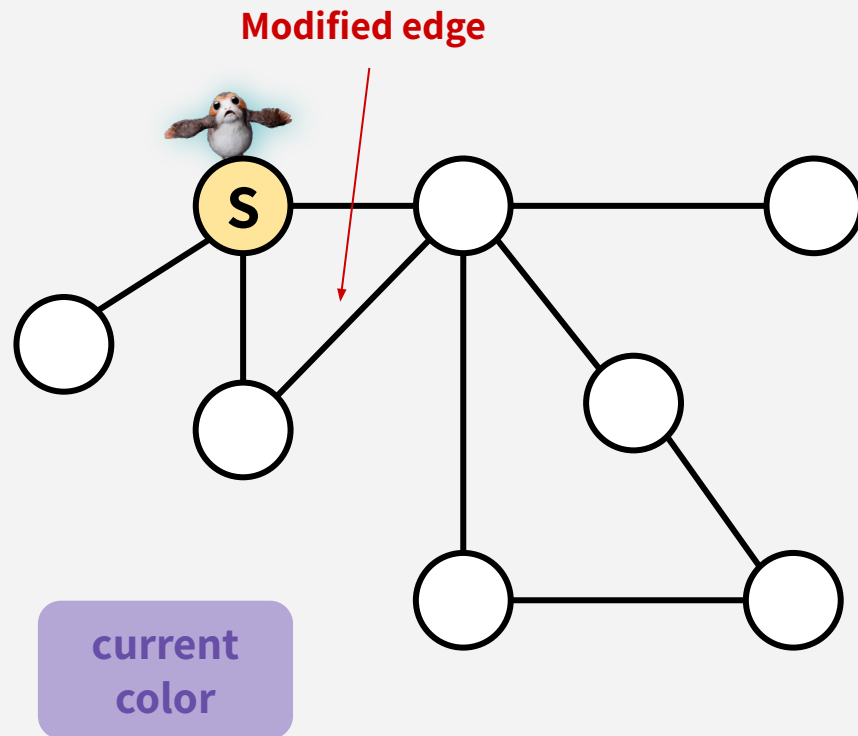




# BIPARTITE GRAPHS

## Application of BFS:

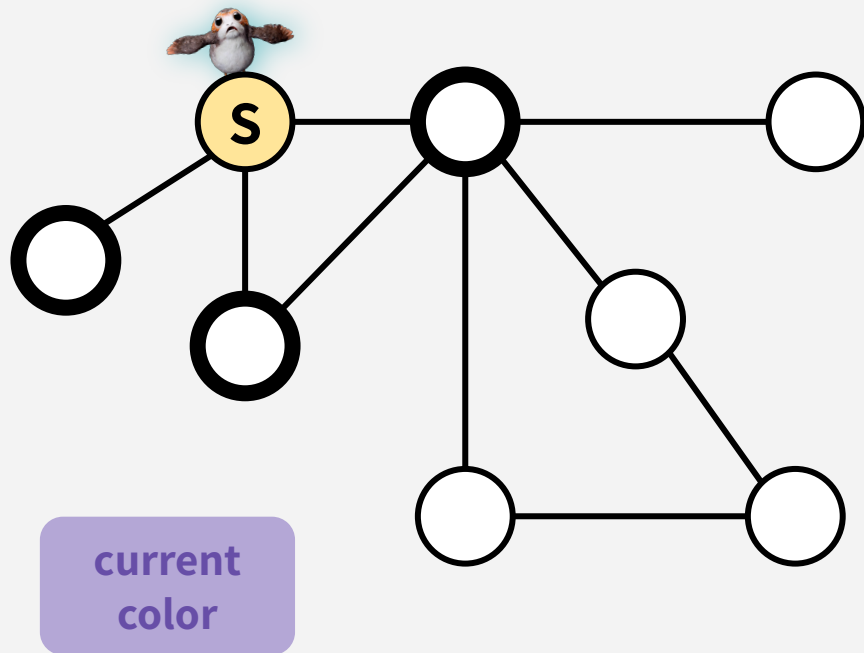
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

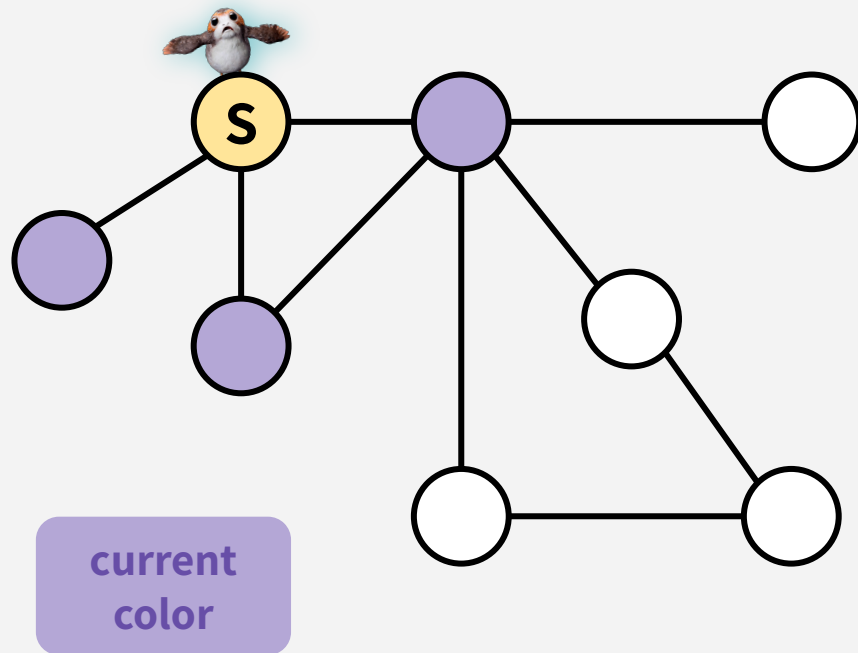
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

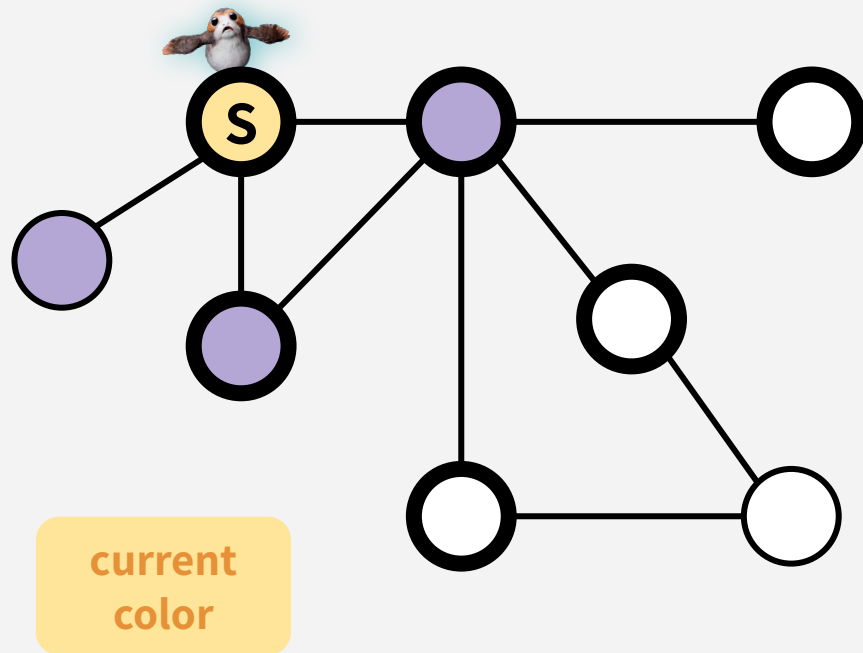
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

## Application of BFS:

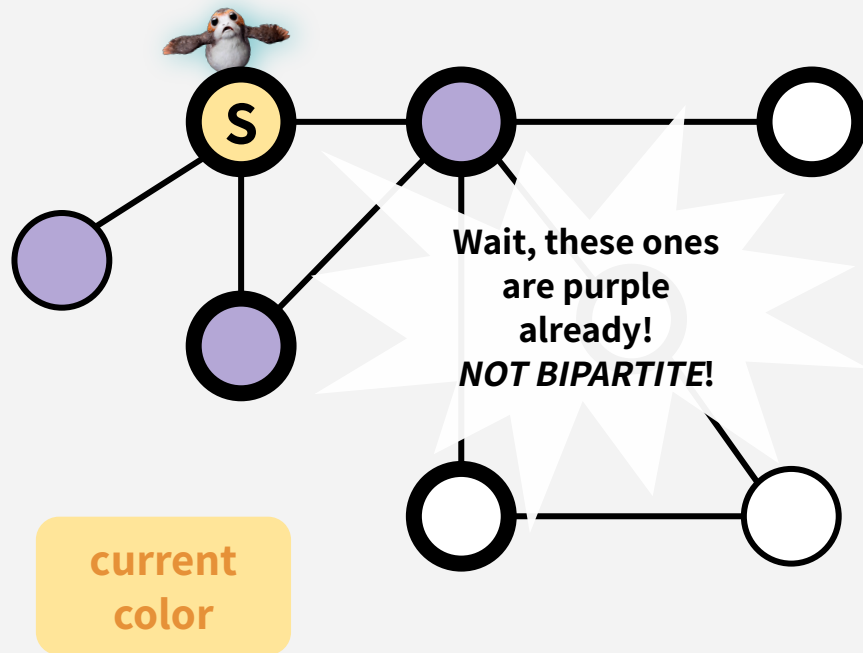
- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

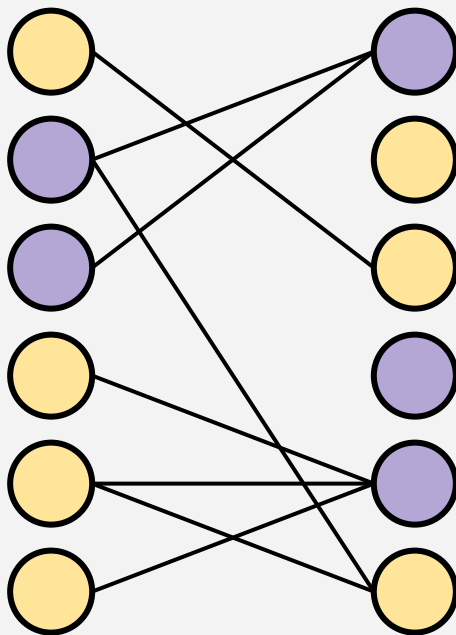
## Application of BFS:

- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!



# BIPARTITE GRAPHS

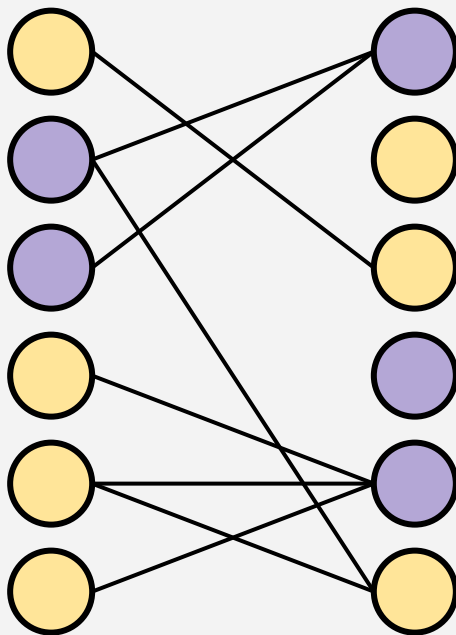
But wait... there exists many poor colorings on legitimate bipartite graphs.  
Just because **the BFS coloring technique** doesn't work, why do we just throw up our hands and say **no** coloring works?



# BIPARTITE GRAPHS

But wait... there exists many poor colorings on legitimate bipartite graphs.  
Just because **the BFS coloring technique** doesn't work, why do we just throw up our hands and say **no** coloring works?

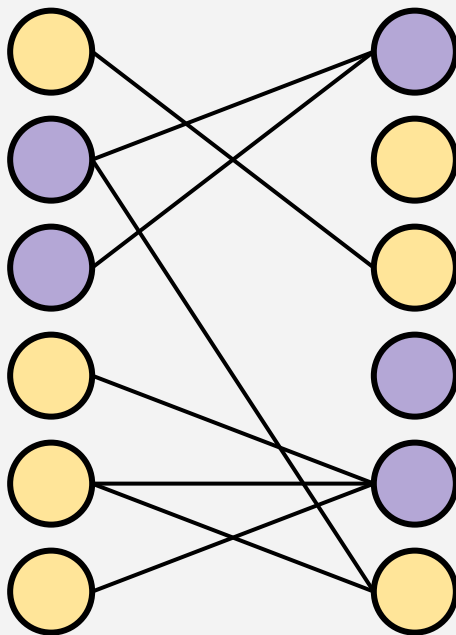
We can come up with plenty of  
bad coloring on this legitimate  
bipartite graph...



# BIPARTITE GRAPHS

But wait... there exists many poor colorings on legitimate bipartite graphs.  
Just because **the BFS coloring technique** doesn't work, why do we just throw up our hands and say **no** coloring works?

We can come up with plenty of  
bad coloring on this legitimate  
bipartite graph...



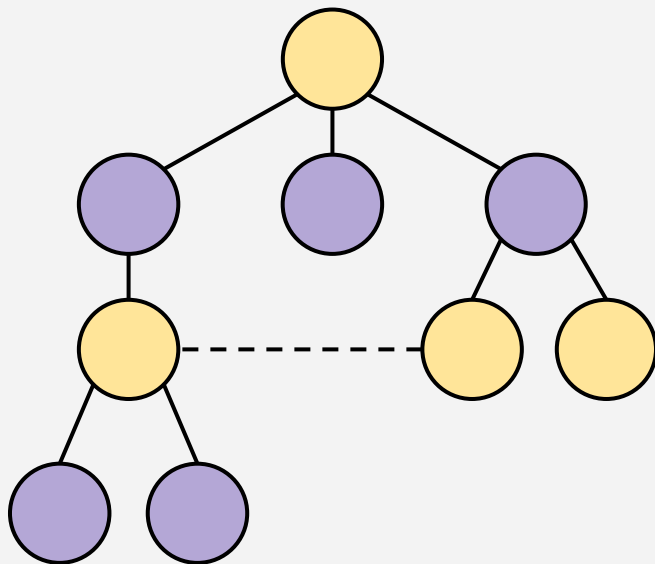
We need to prove that if BFS  
encounters a conflict (tries to  
color two neighbors the same  
color!), then there's no way the  
graph could be bipartite.



# BIPARTITE GRAPHS

If BFS tries to color two neighbors the same color, then it's found a **cycle of odd length** in the graph

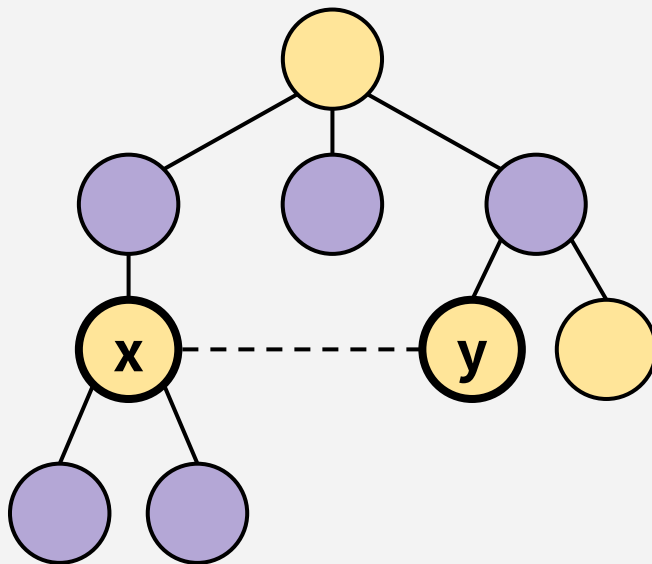
This is the BFS tree. Each level in this tree corresponds to each “BFS level”. Our BFS coloring technique basically tries to alternate colors across levels.



# BIPARTITE GRAPHS

If BFS tries to color two neighbors the same color, then it's found a **cycle of odd length** in the graph

This is the BFS tree. Each level in this tree corresponds to each "BFS level". Our BFS coloring technique basically tries to alternate colors across levels.

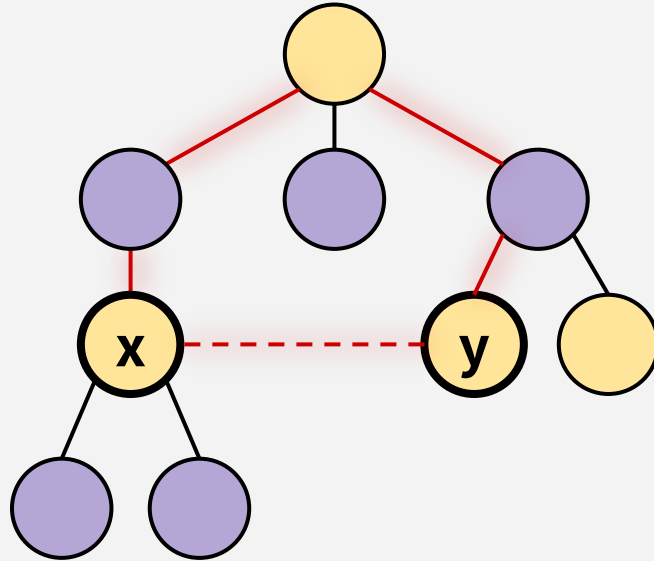


These neighbors are the conflict! BFS will try to color one of **x** or **y** purple, but it's already been colored yellow.

# BIPARTITE GRAPHS

If BFS tries to color two neighbors the same color, then it's found a **cycle of odd length** in the graph

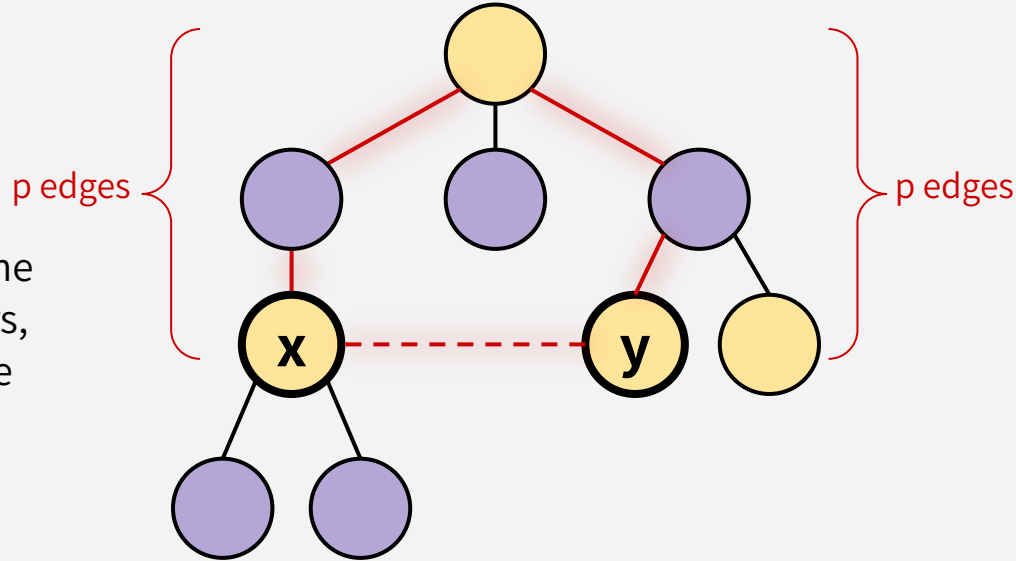
If **x** and **y** are the same color & are neighbors, then they are on the same level.



# BIPARTITE GRAPHS

If BFS tries to color two neighbors the same color, then it's found a **cycle of odd length** in the graph

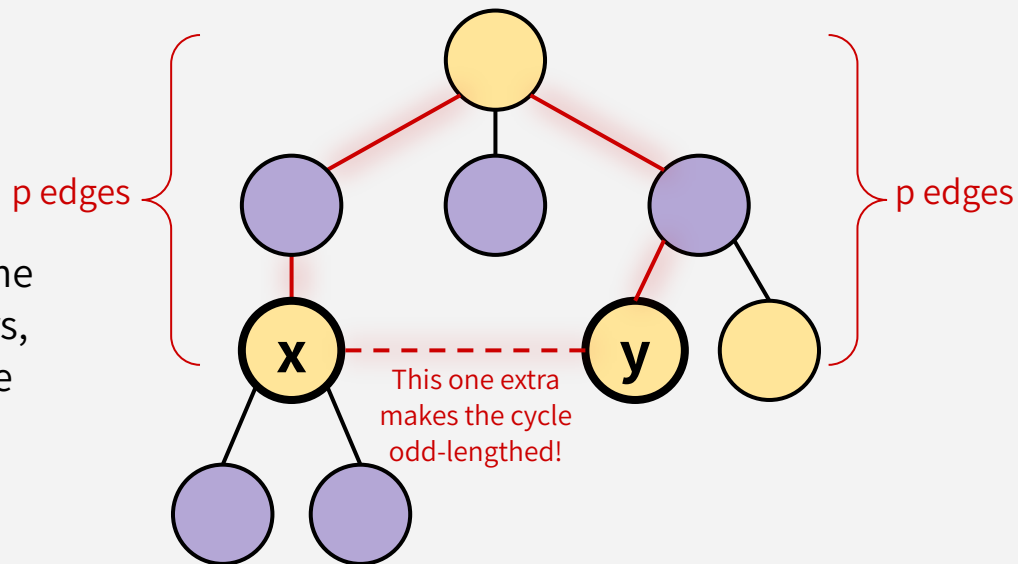
If **x** and **y** are the same color & are neighbors, then they are on the same level.



# BIPARTITE GRAPHS

If BFS tries to color two neighbors the same color, then it's found a **cycle of odd length** in the graph

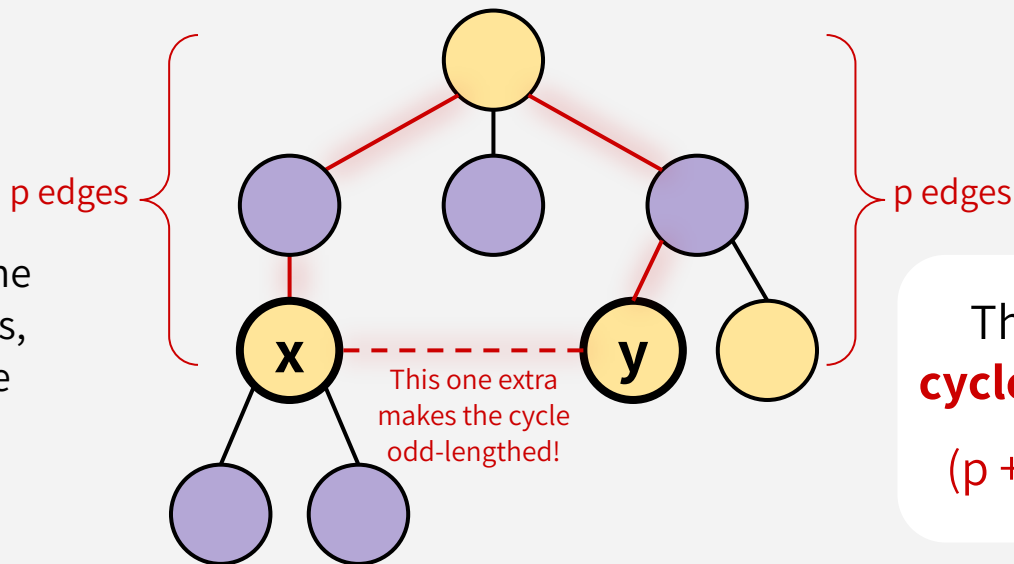
If **x** and **y** are the same color & are neighbors, then they are on the same level.



# BIPARTITE GRAPHS

If BFS tries to color two neighbors the same color, then it's found a **cycle of odd length** in the graph

If **x** and **y** are the same color & are neighbors, then they are on the same level.



Thus, there is a **cycle of odd length**

$$(p + p + 1) = \text{odd \#}$$

# BIPARTITE GRAPHS

If BFS tries to color two neighbors the same color,

It's impossible to color a cycle of odd length with two colors such that no two neighbors have the same color. Therefore, it's impossible to two-color the graph such that no adjacent vertices are colored the same.

**So, BFS colors two neighbors the same color iff the graph is not bipartite.**



$$(p + p + 1) = \text{odd \#}$$

# BFS & BIPARTITE GRAPHS RECAP

**BFS can be used to detect bipartite-ness of a graph in time  $O(n + m)$ ,** since all that coloring business is just  $O(1)$  extra work per node or edge.

This is one example of how you can take advantage of the “layers” that BFS constructs to reason about how to accomplish a task that might not seem like a “classic” BFS-shortest-path task (which you might be more familiar with).





سوال؟

# جستجوی عمق اول (DFS)

**یک روش دیگر برای پیمایش گراف**

# BFS vs. DFS

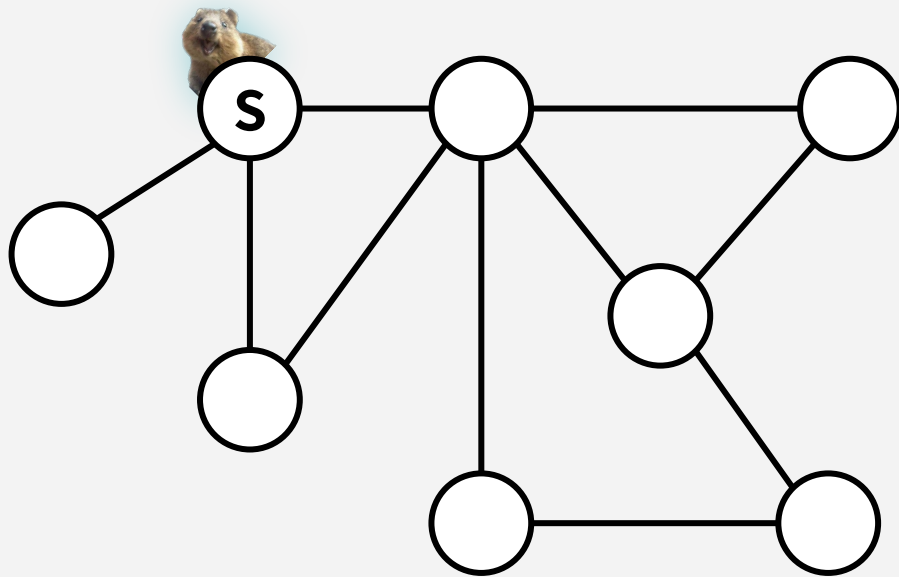
## **Literally just BREADTH vs DEPTH:**

While BFS first explores the nodes closest to the “source” and then moves outwards in layers, DFS goes as far down a path as it can before it comes back to explore other options.

# DEPTH-FIRST SEARCH

## **An analogy:**

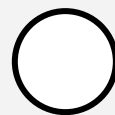
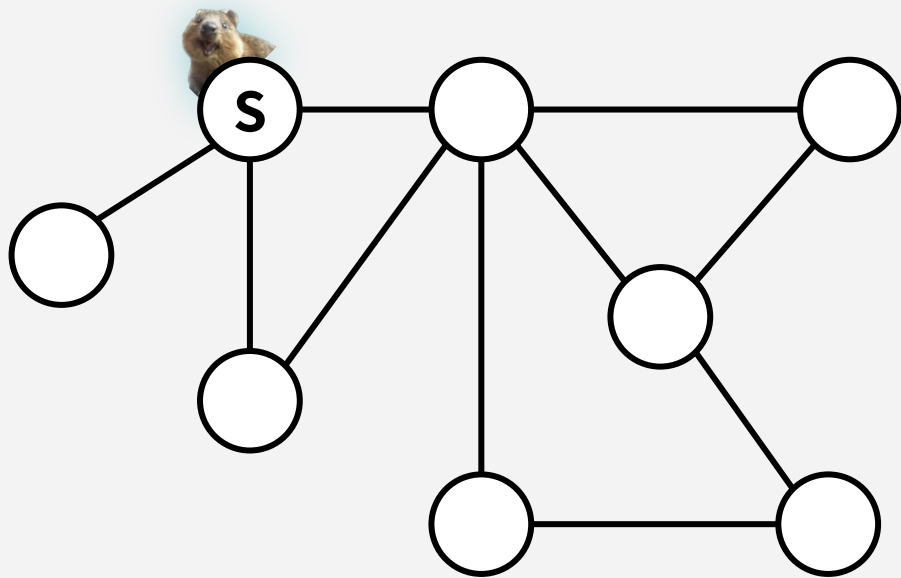
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out

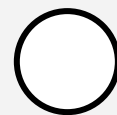
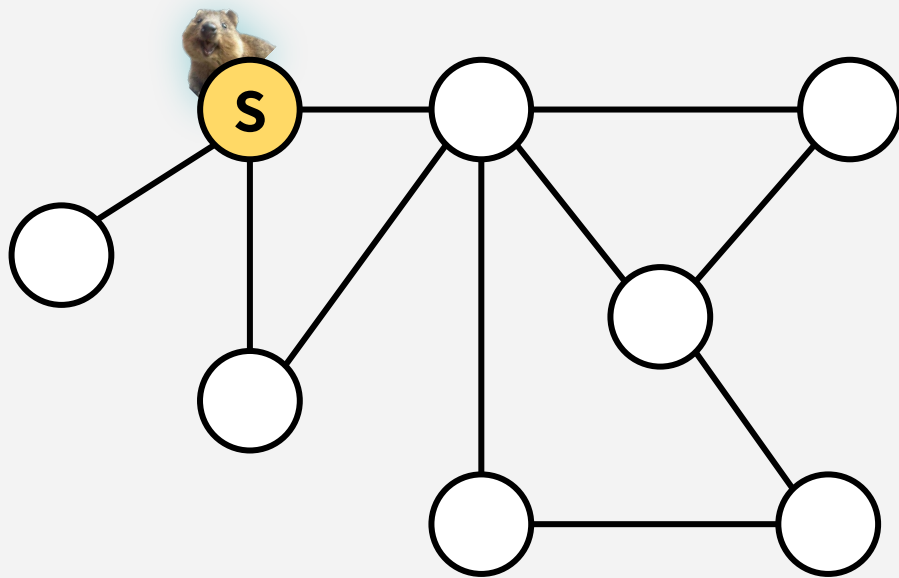


visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out

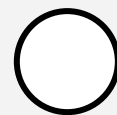
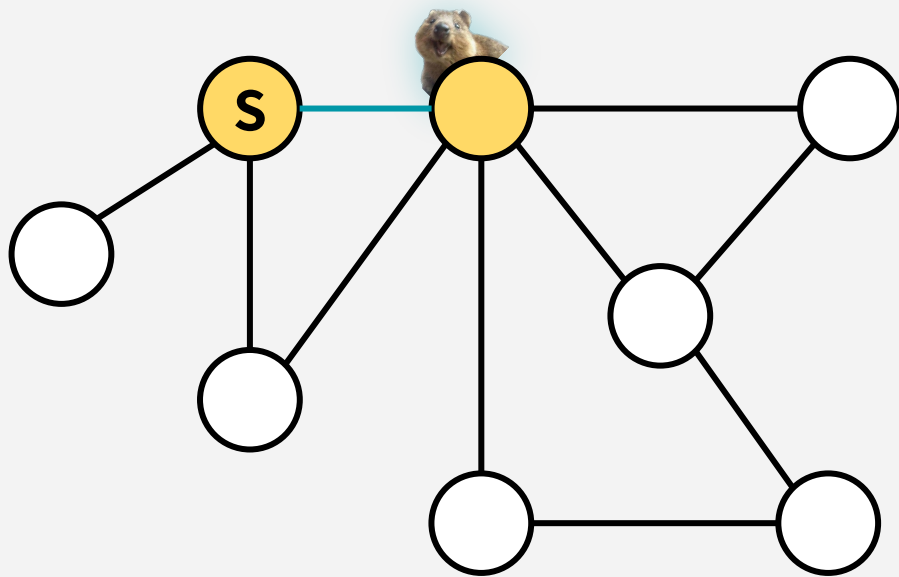


visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out

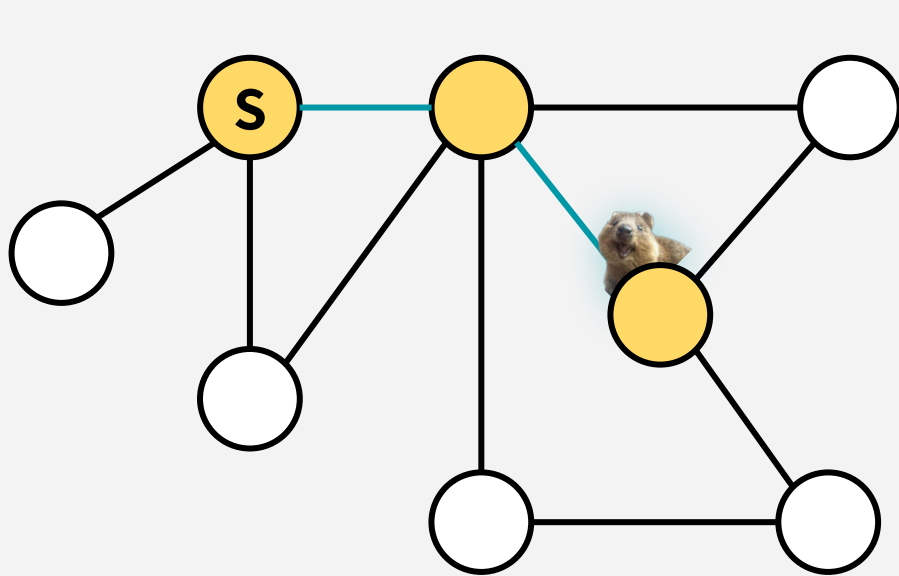


visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

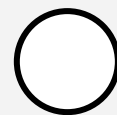
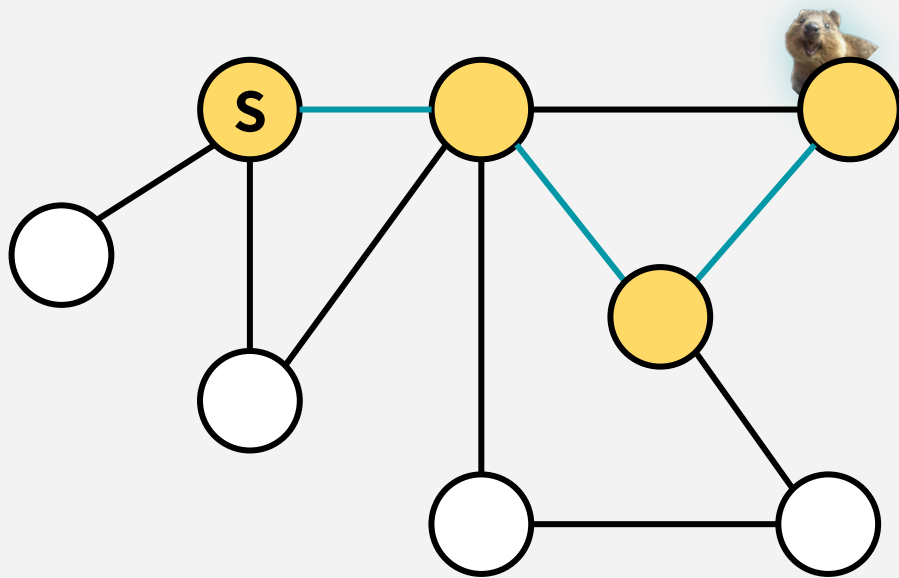




# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out

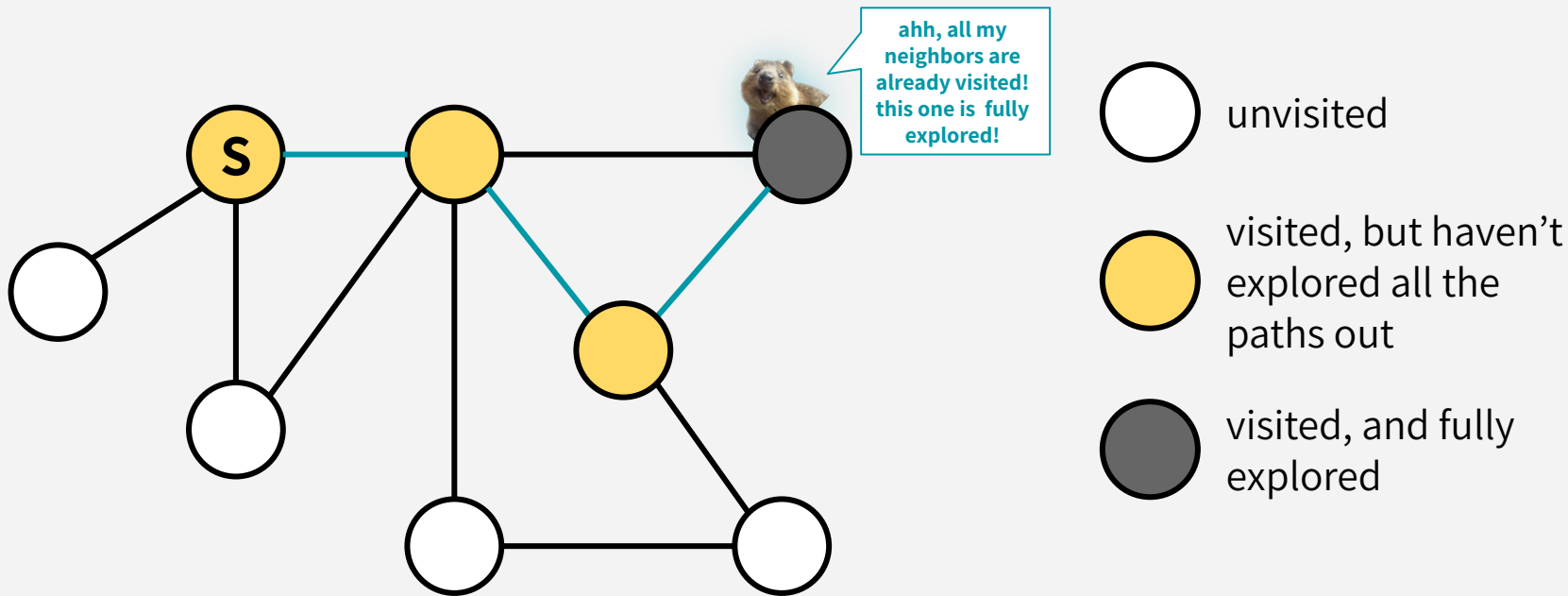


visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

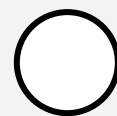
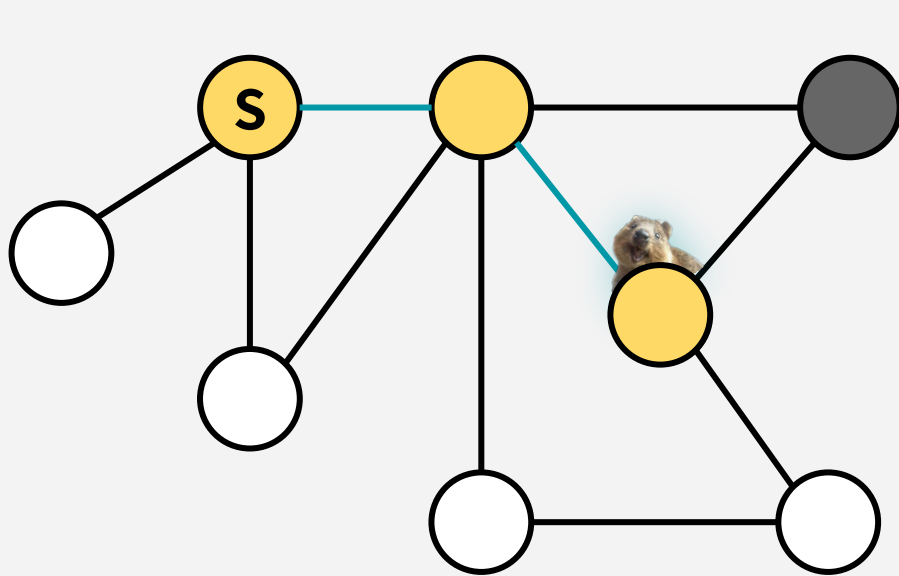
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't explored all the paths out

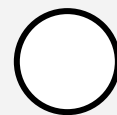
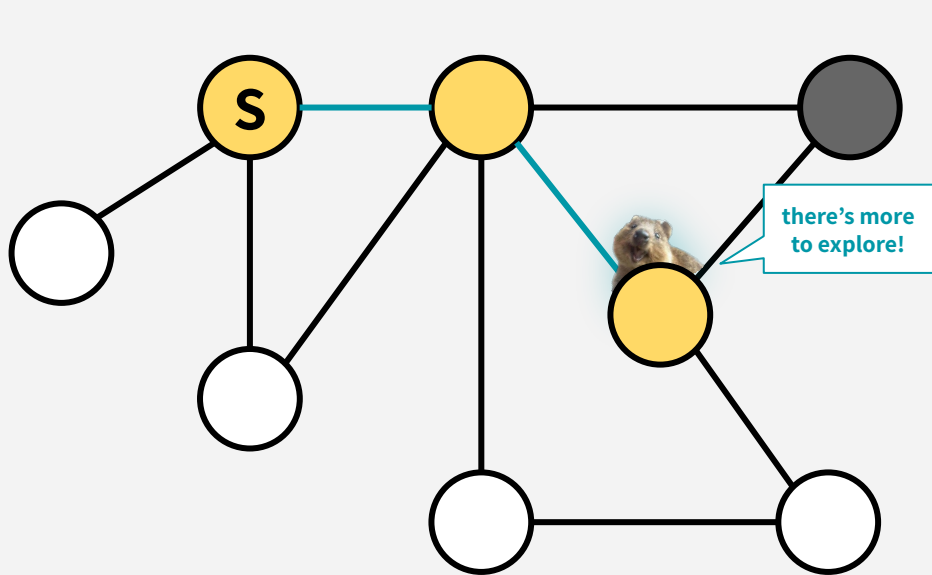


visited, and fully explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out

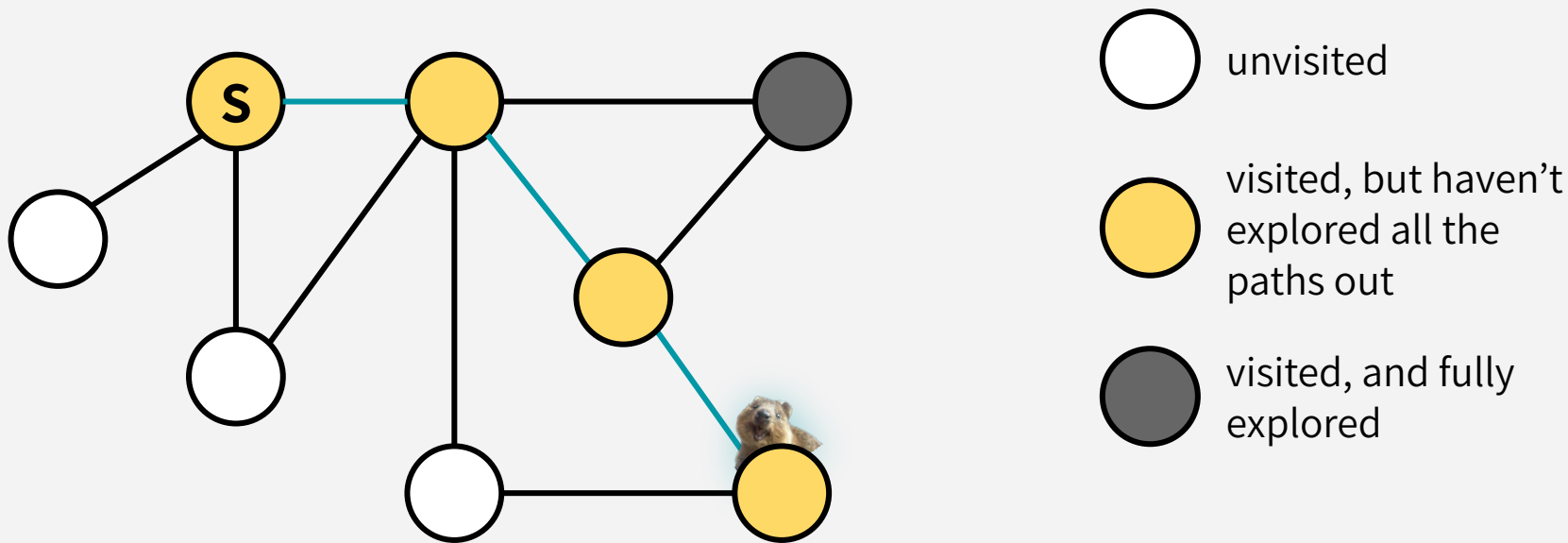


visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

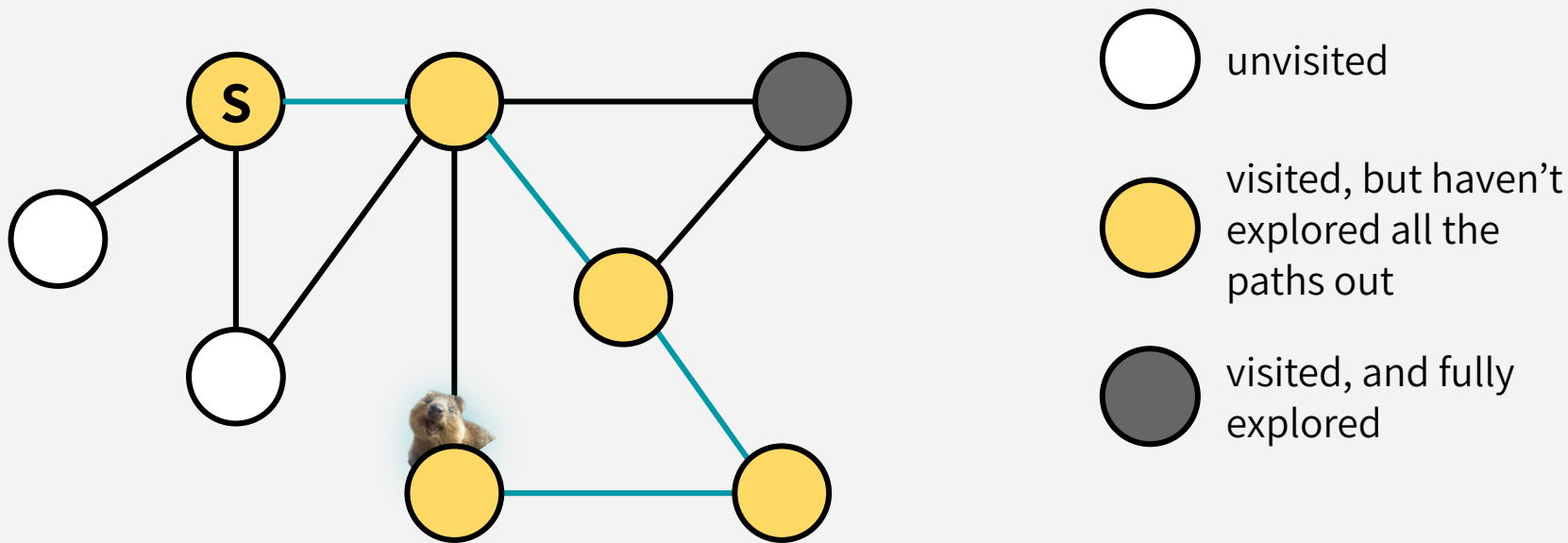
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



# DEPTH-FIRST SEARCH

## An analogy:

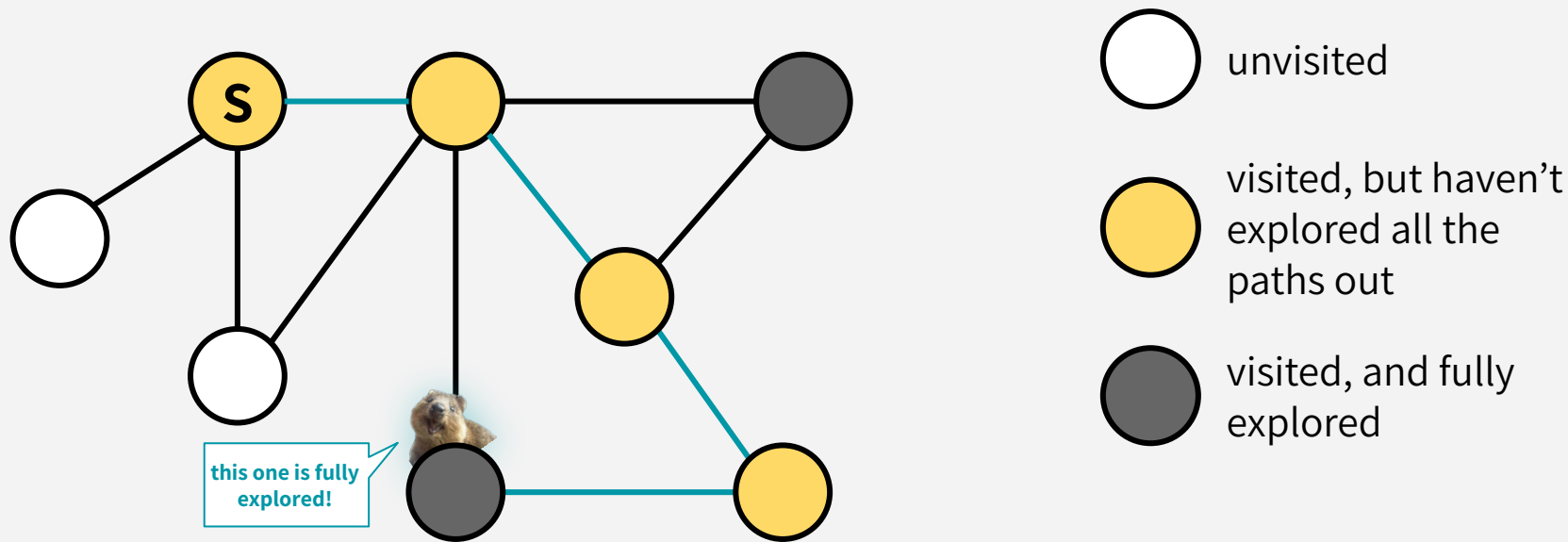
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



# DEPTH-FIRST SEARCH

## An analogy:

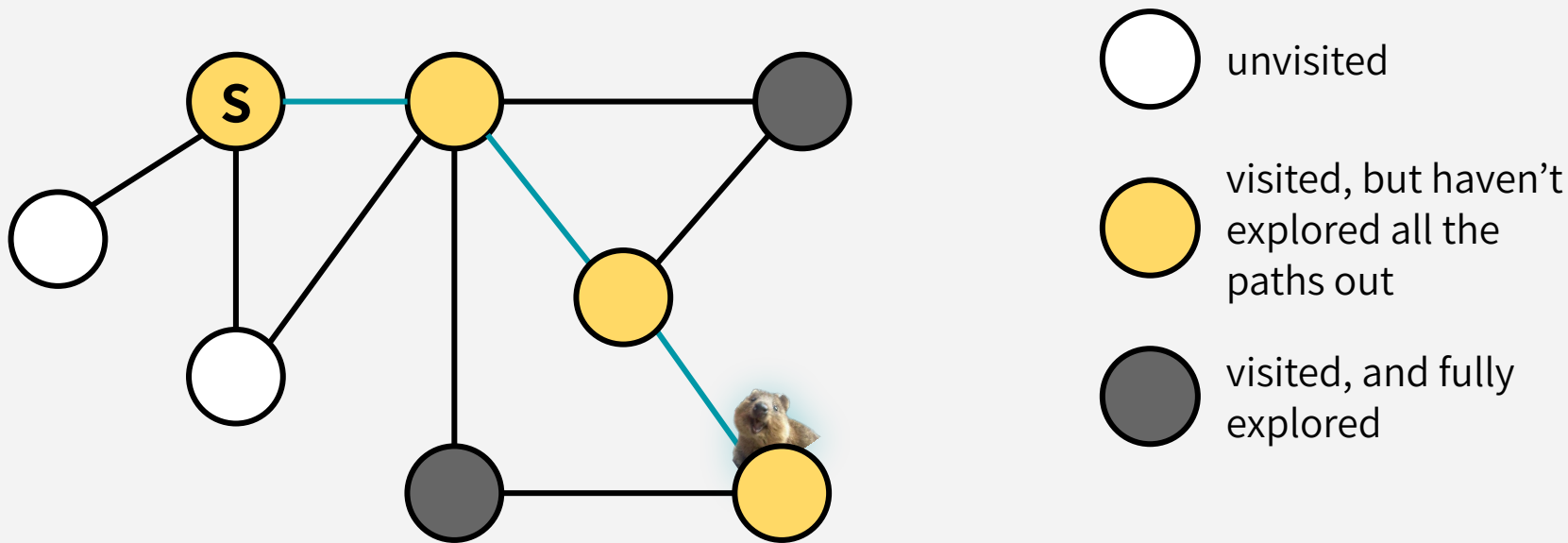
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

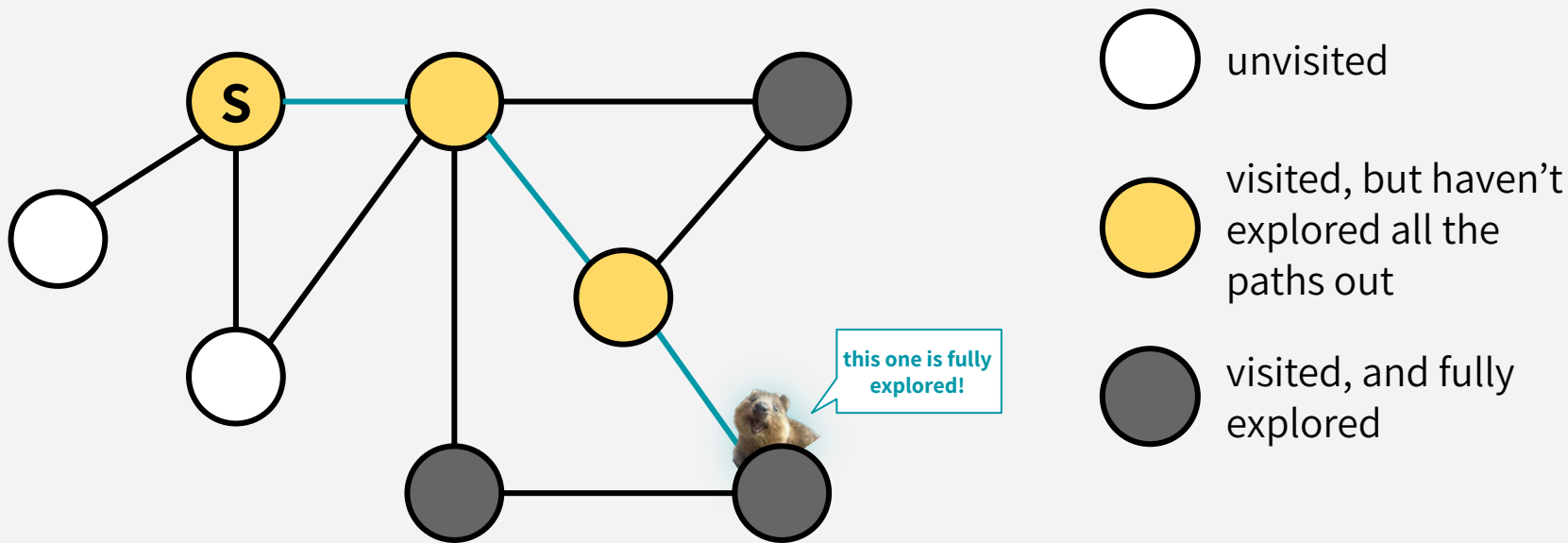




# DEPTH-FIRST SEARCH

## An analogy:

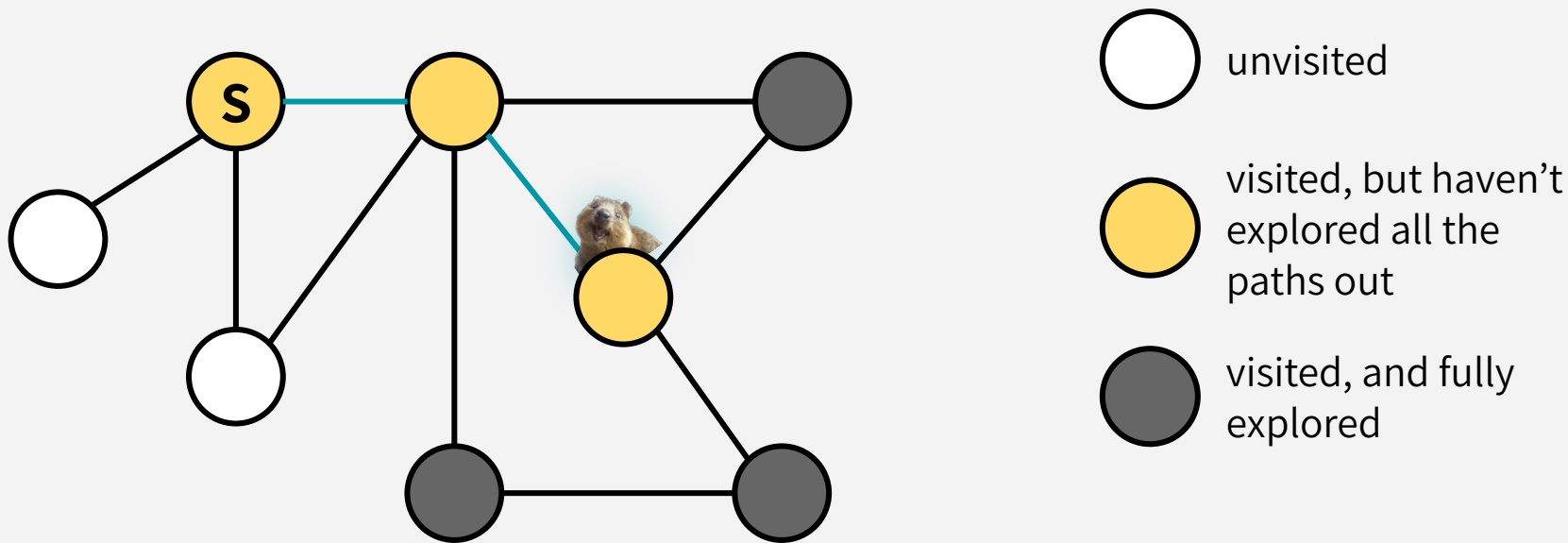
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



# DEPTH-FIRST SEARCH

## An analogy:

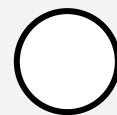
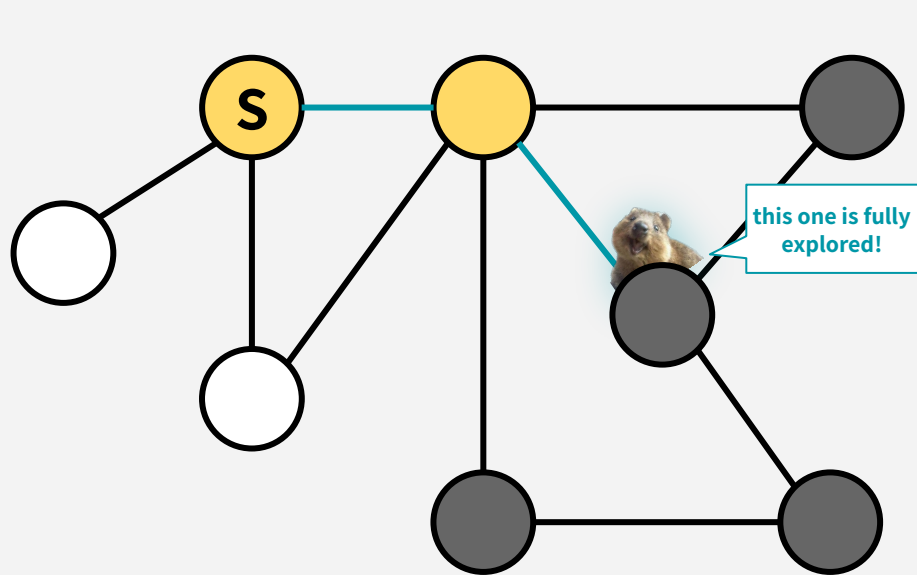
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out

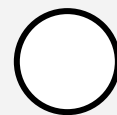
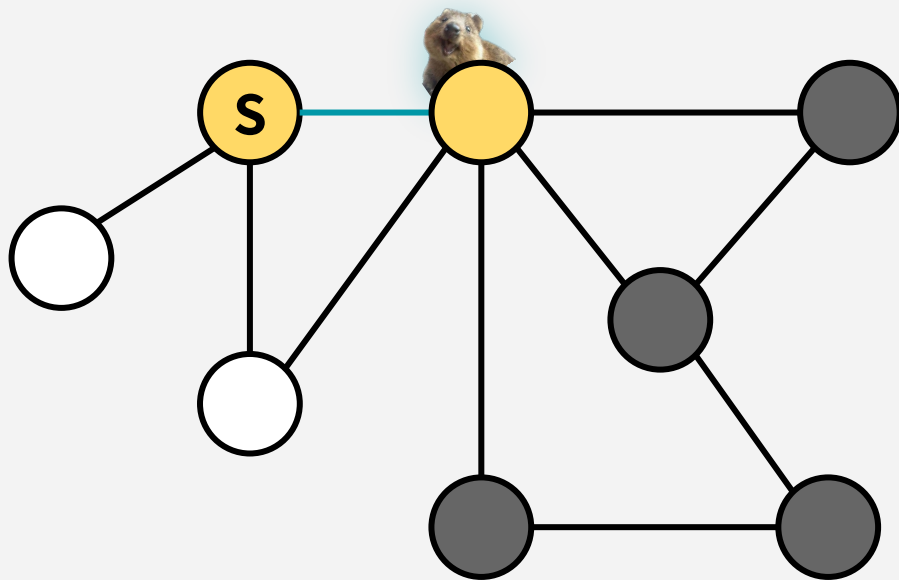


visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out

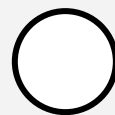
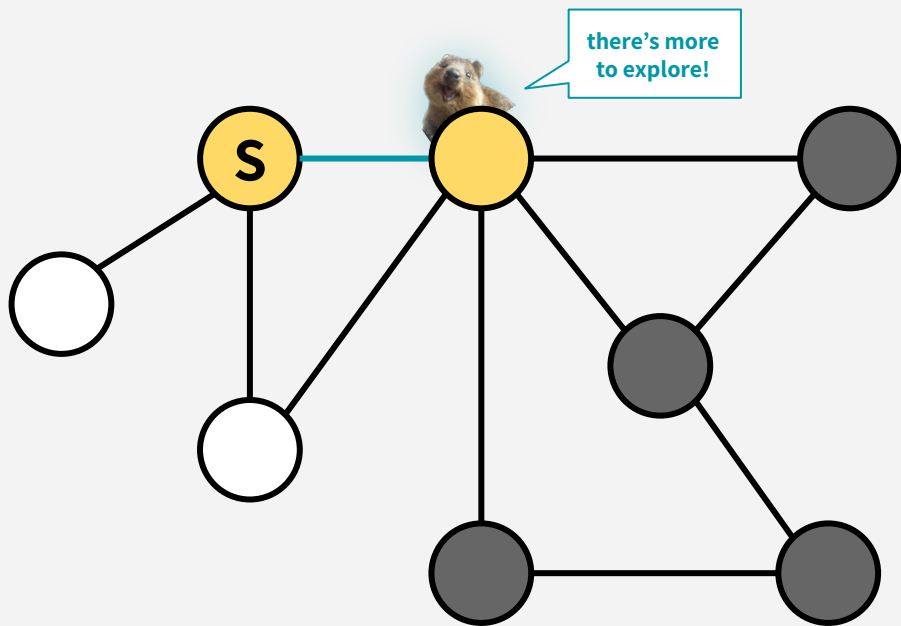


visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't explored all the paths out

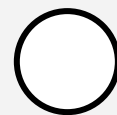
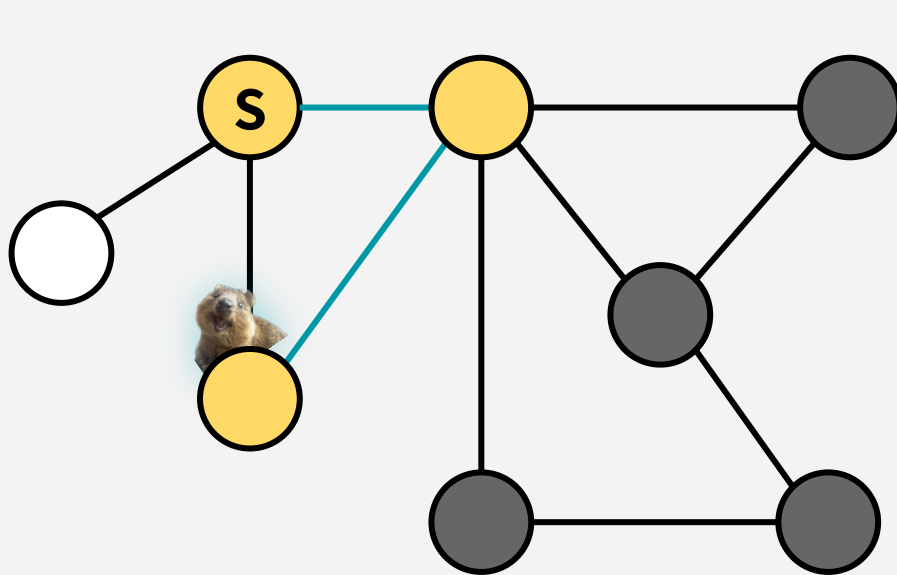


visited, and fully explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out

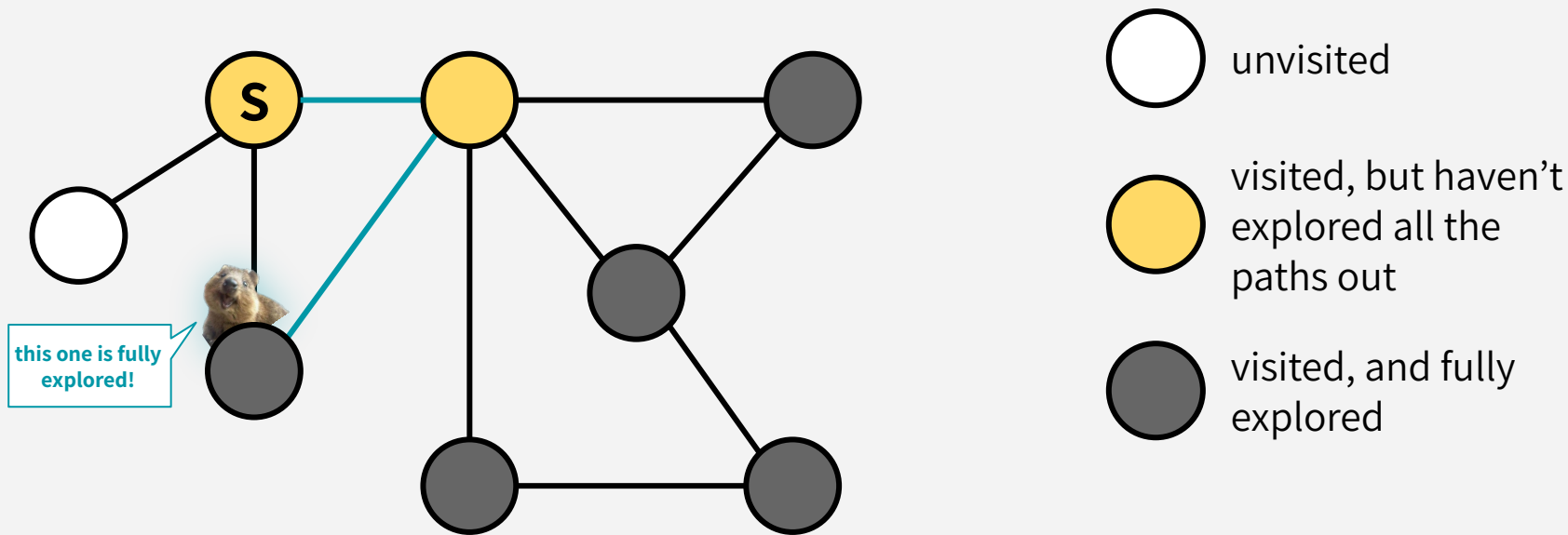


visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

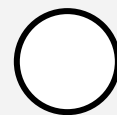
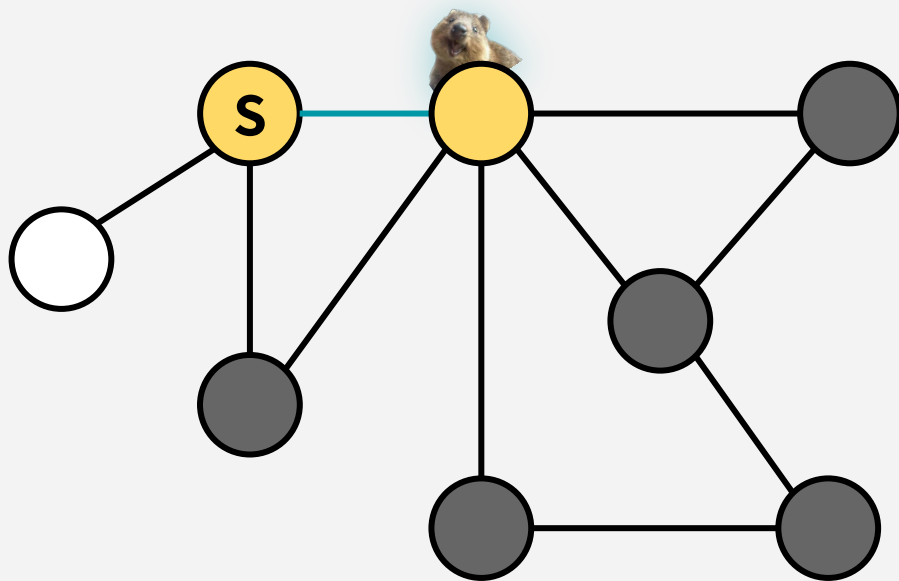
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out



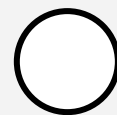
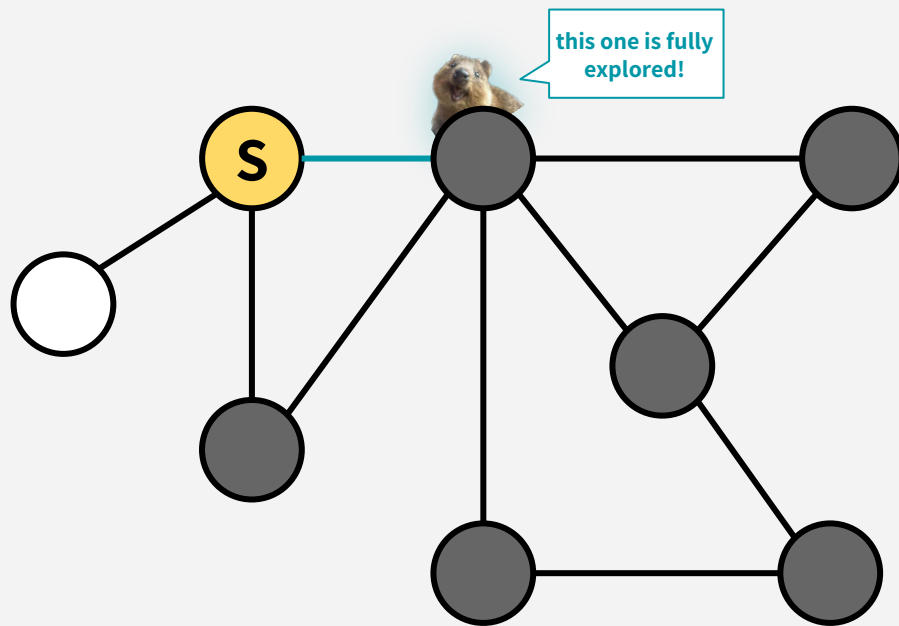
visited, and fully  
explored



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't explored all the paths out

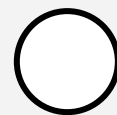
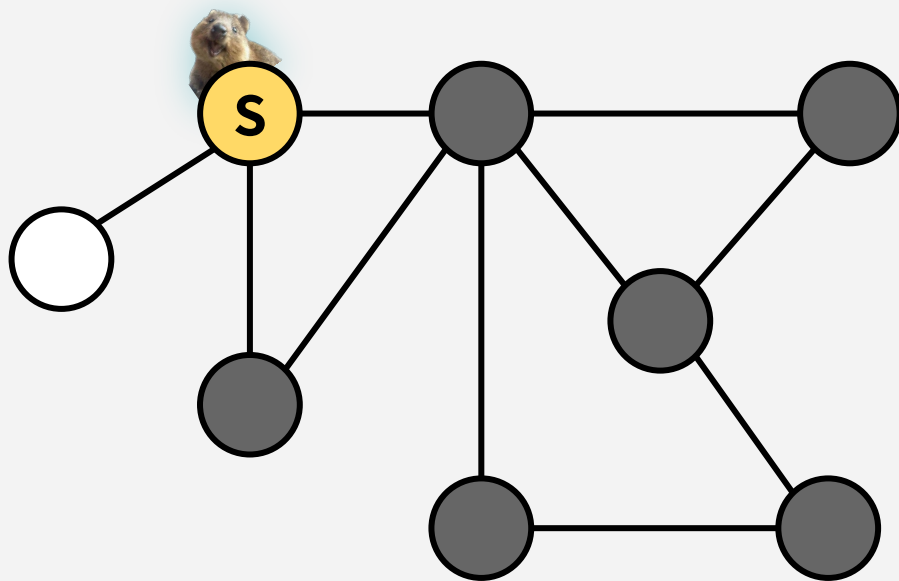


visited, and fully explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't  
explored all the  
paths out



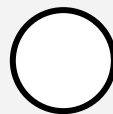
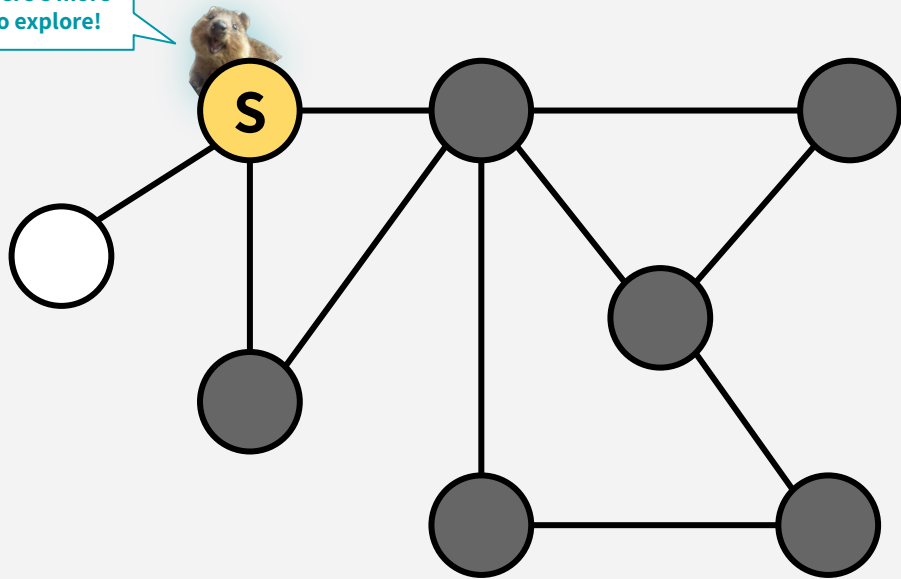
visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

there's more  
to explore!



unvisited



visited, but haven't  
explored all the  
paths out

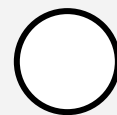
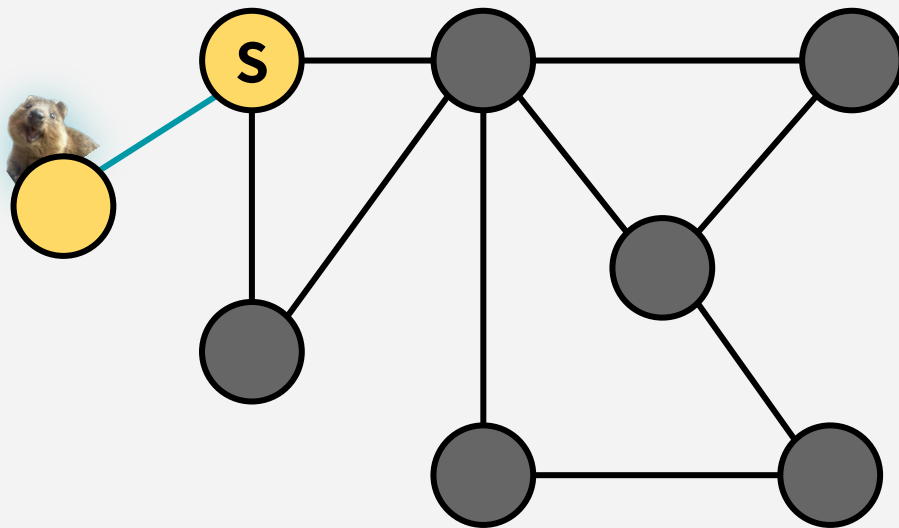


visited, and fully  
explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited



visited, but haven't explored all the paths out



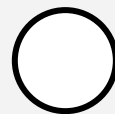
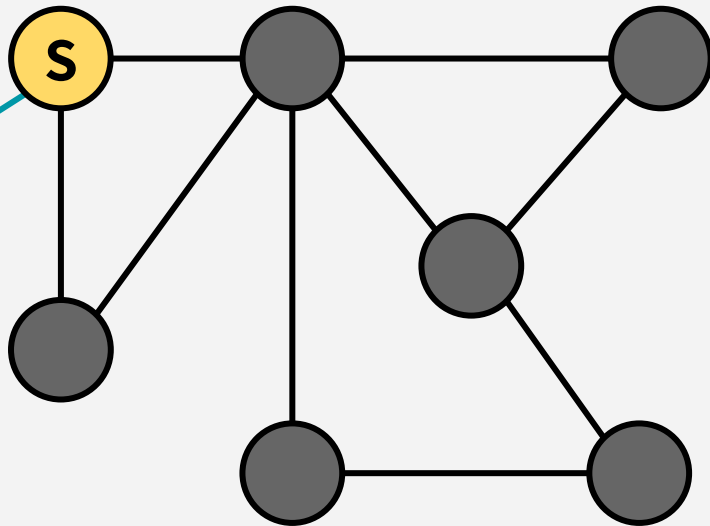
visited, and fully explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

this one is fully explored!



unvisited



visited, but haven't explored all the paths out

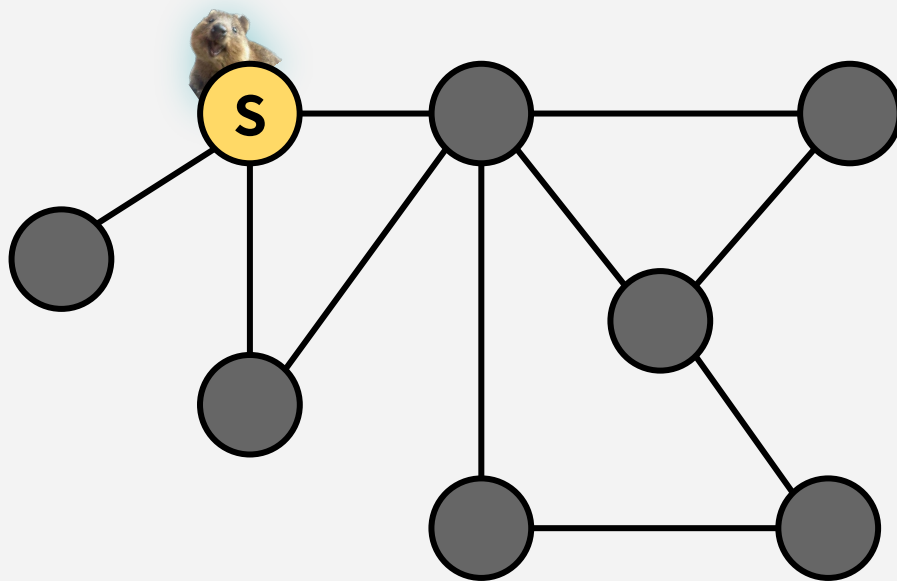


visited, and fully explored

# DEPTH-FIRST SEARCH

## An analogy:

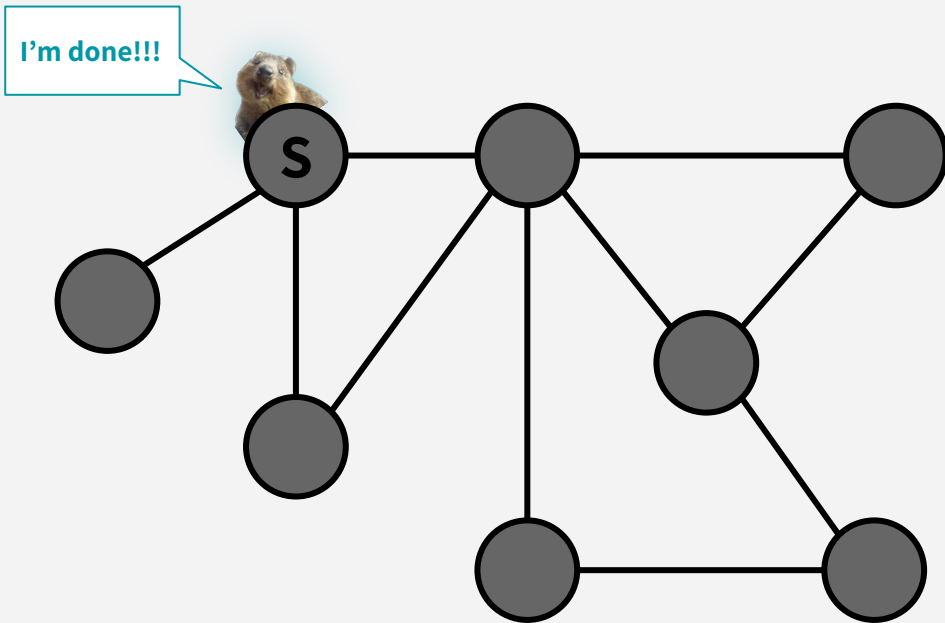
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



-  unvisited
-  visited, but haven't explored all the paths out
-  visited, and fully explored

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

I'm done!!!

In addition to keeping track of the visited status of nodes, we're going to keep track of:

**The time we first enter it**, i.e. mark it as .

**The time we finish it**, i.e. mark it as .

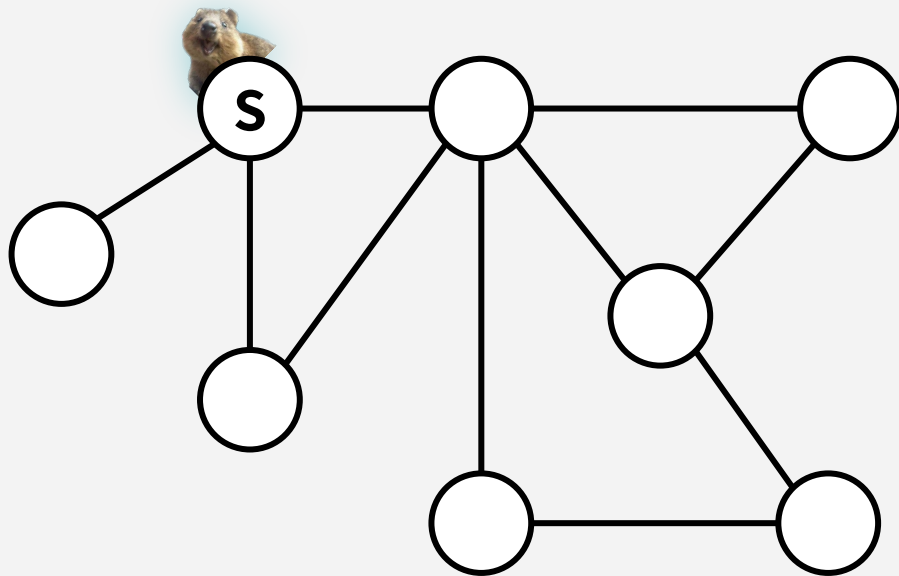
You've probably seen other ways to implement DFS, all this extra bookkeeping will be useful for us later!



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

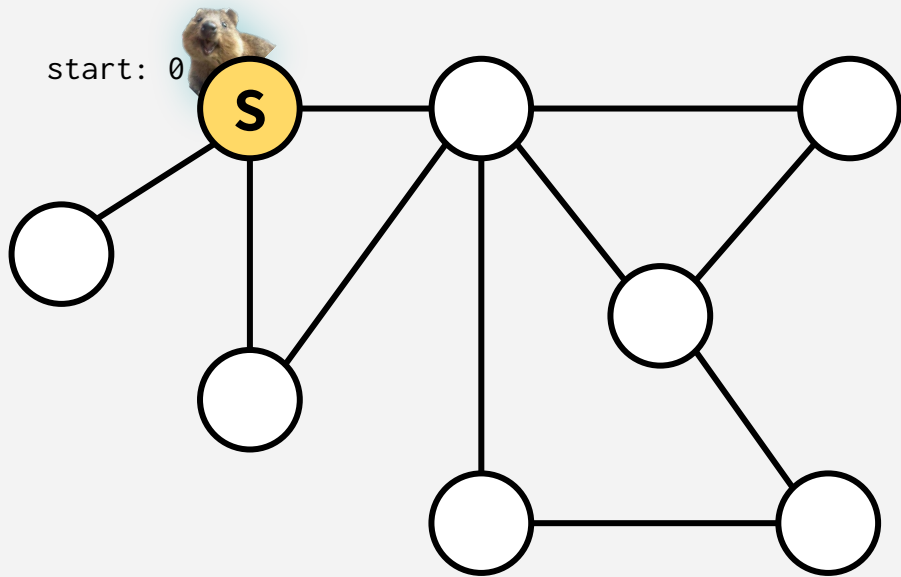


```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

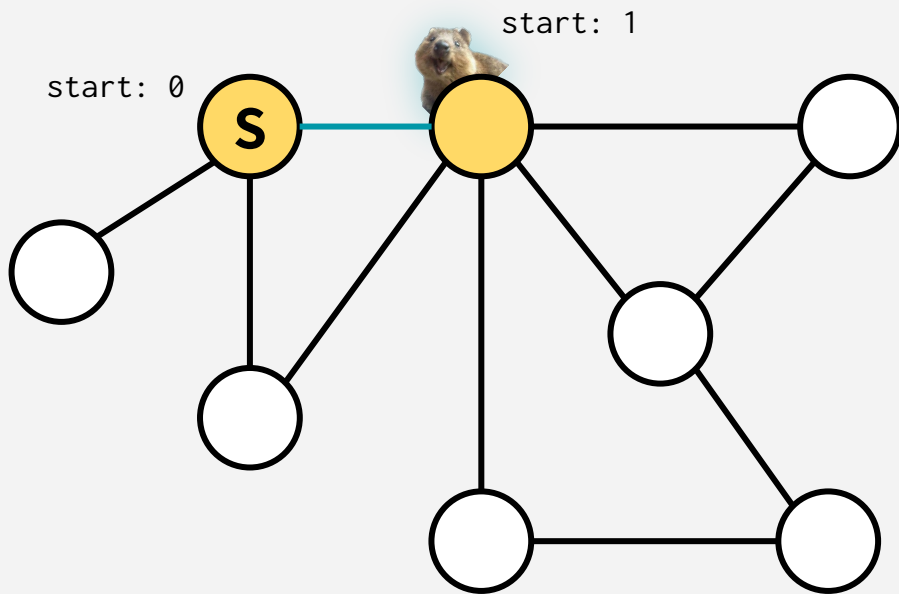


```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

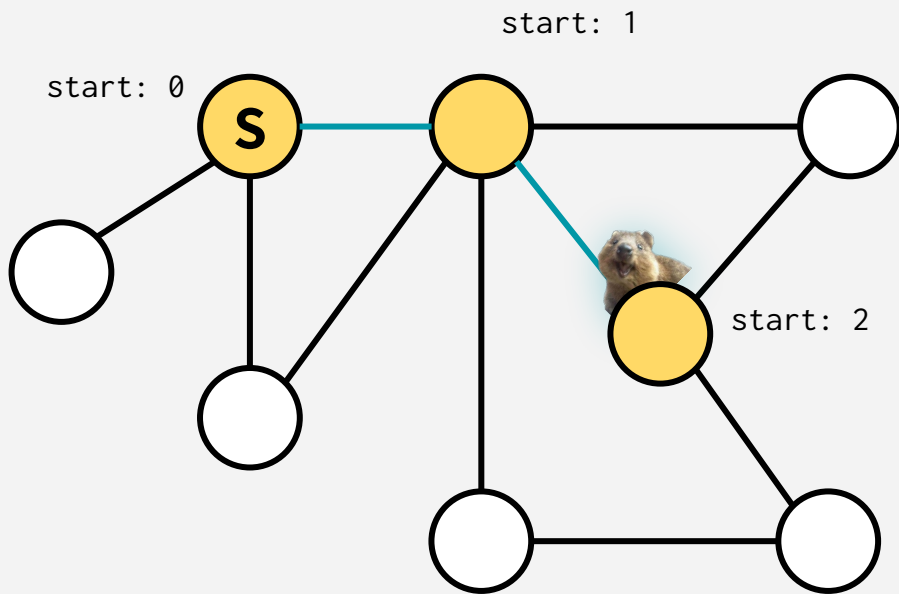


```
DFS(w, currTime):  
    w.start = currTime  
    currTime++  
    mark w as visited  
    for v in w.neighbors:  
        if v is unvisited:  
            currTime =  
                DFS(v, currTime)  
            currTime++  
    w.finish = currTime  
    mark w as finished  
    return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

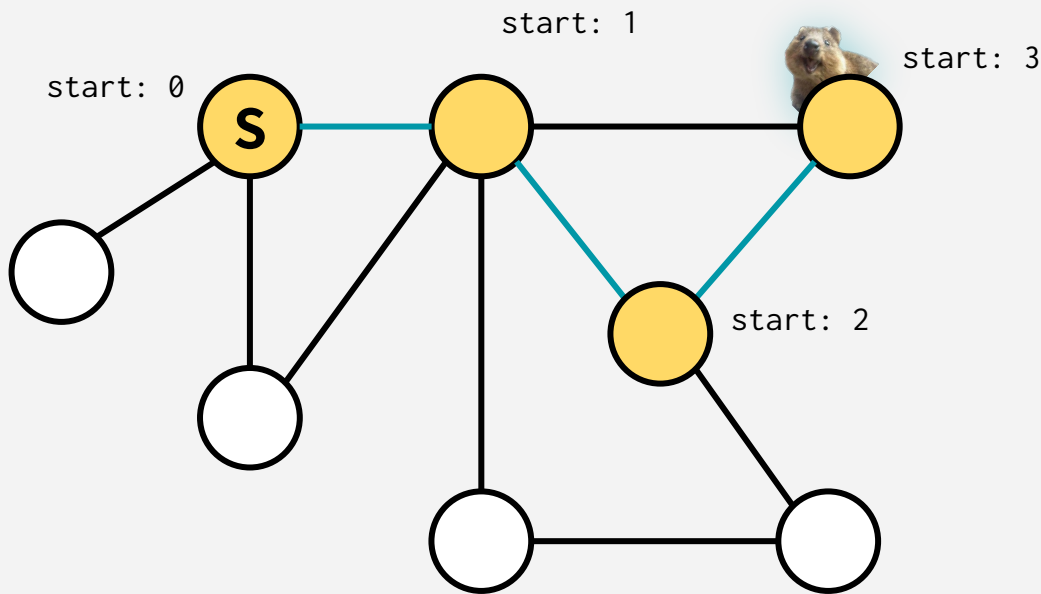


```
DFS(w, currTime):  
    w.start = currTime  
    currTime++  
    mark w as visited  
    for v in w.neighbors:  
        if v is unvisited:  
            currTime =  
                DFS(v, currTime)  
            currTime++  
    w.finish = currTime  
    mark w as finished  
    return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

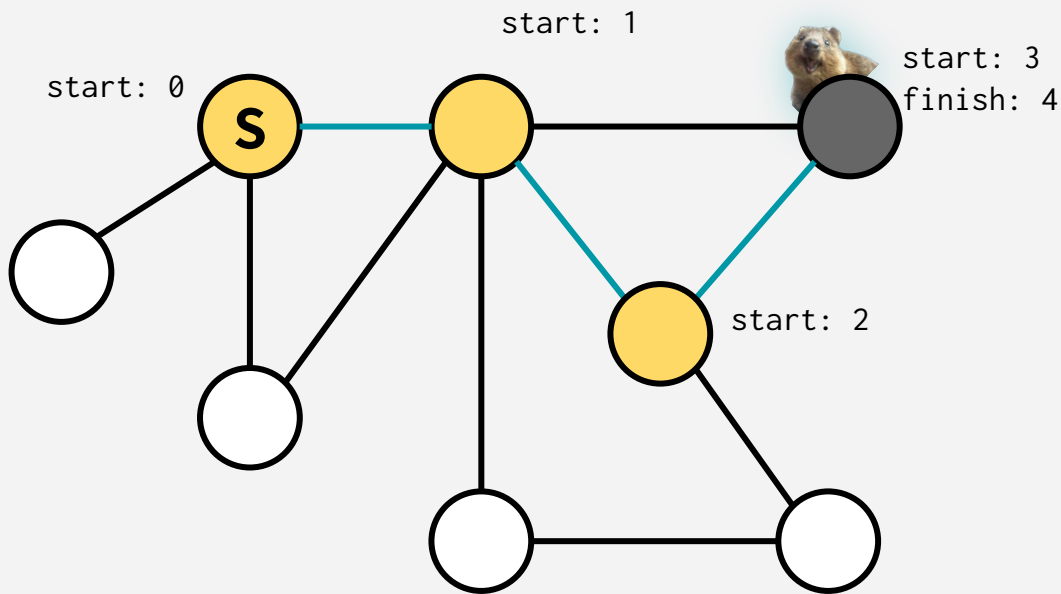


```
DFS(w, currTime):  
    w.start = currTime  
    currTime++  
    mark w as visited  
    for v in w.neighbors:  
        if v is unvisited:  
            currTime =  
                DFS(v, currTime)  
            currTime++  
    w.finish = currTime  
    mark w as finished  
    return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



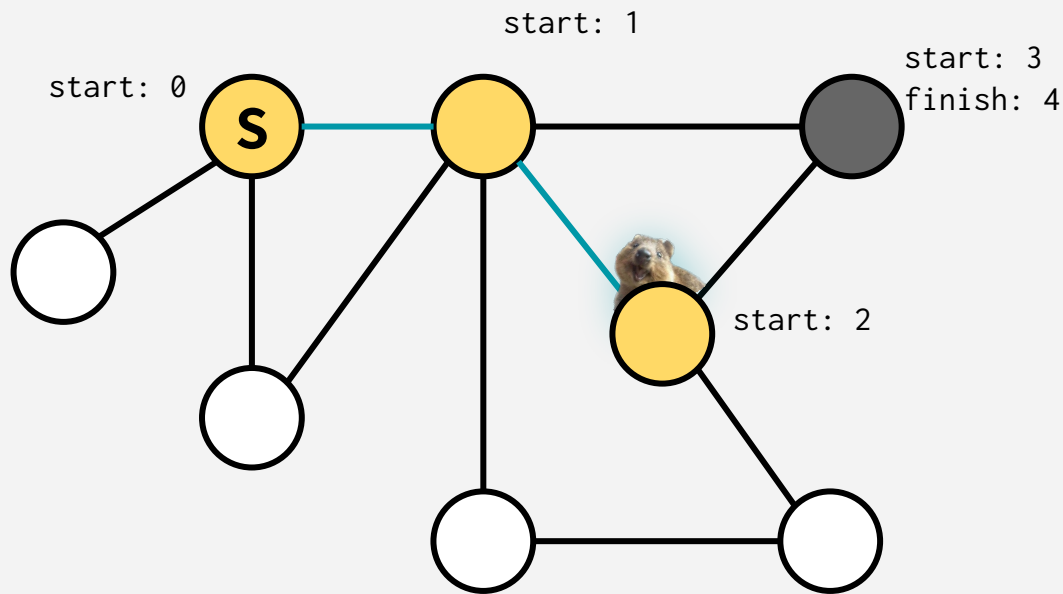
DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

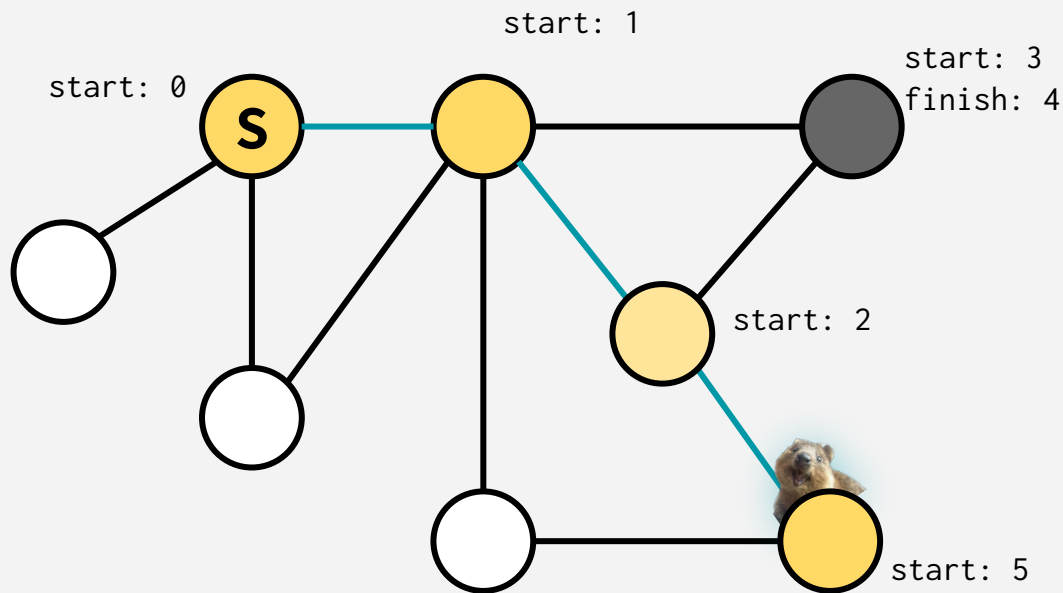


```
DFS(w, currTime):  
    w.start = currTime  
    currTime++  
    mark w as visited  
    for v in w.neighbors:  
        if v is unvisited:  
            currTime =  
                DFS(v, currTime)  
            currTime++  
    w.finish = currTime  
    mark w as finished  
    return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



## DFS(w, currTime):

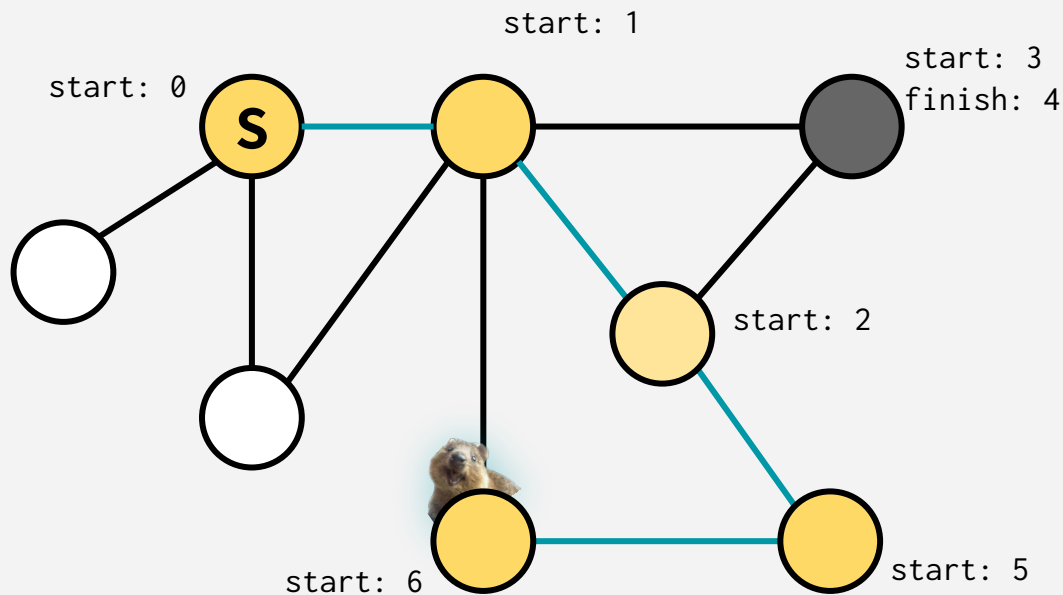
```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



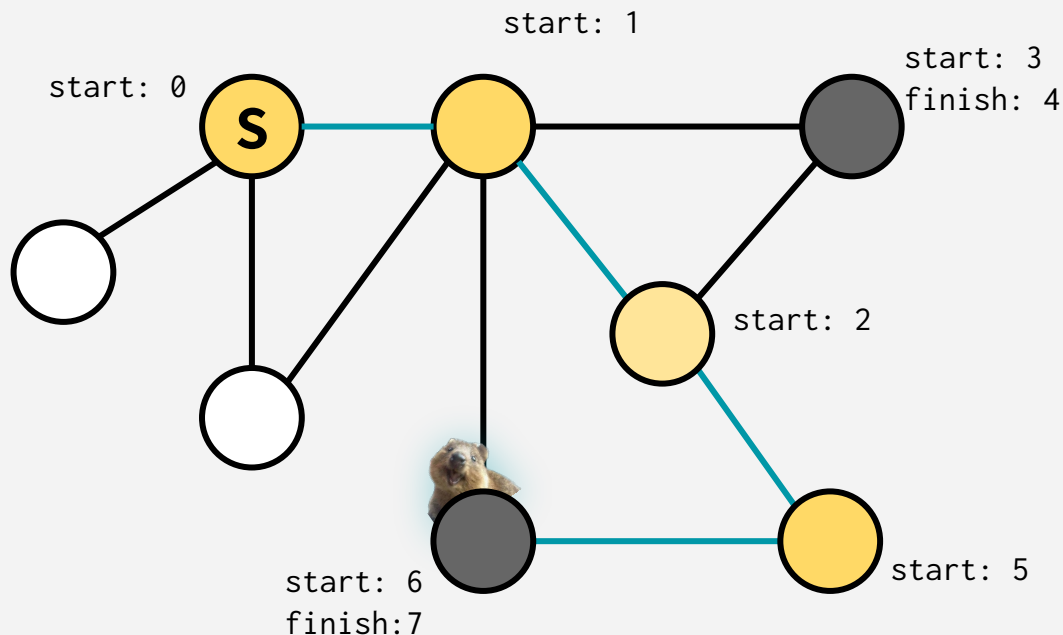
### DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



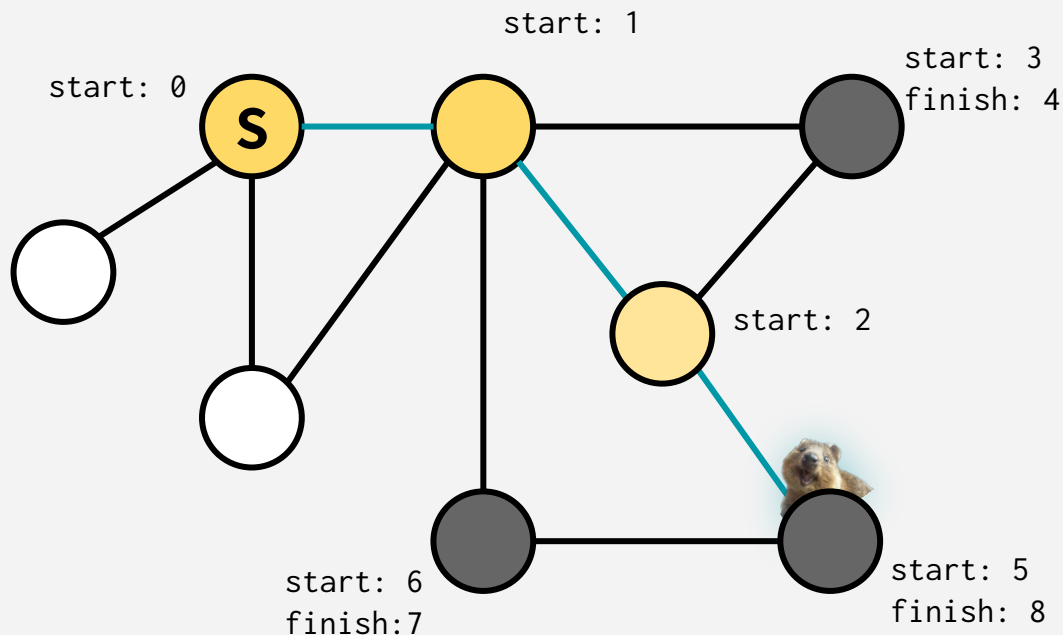
## DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



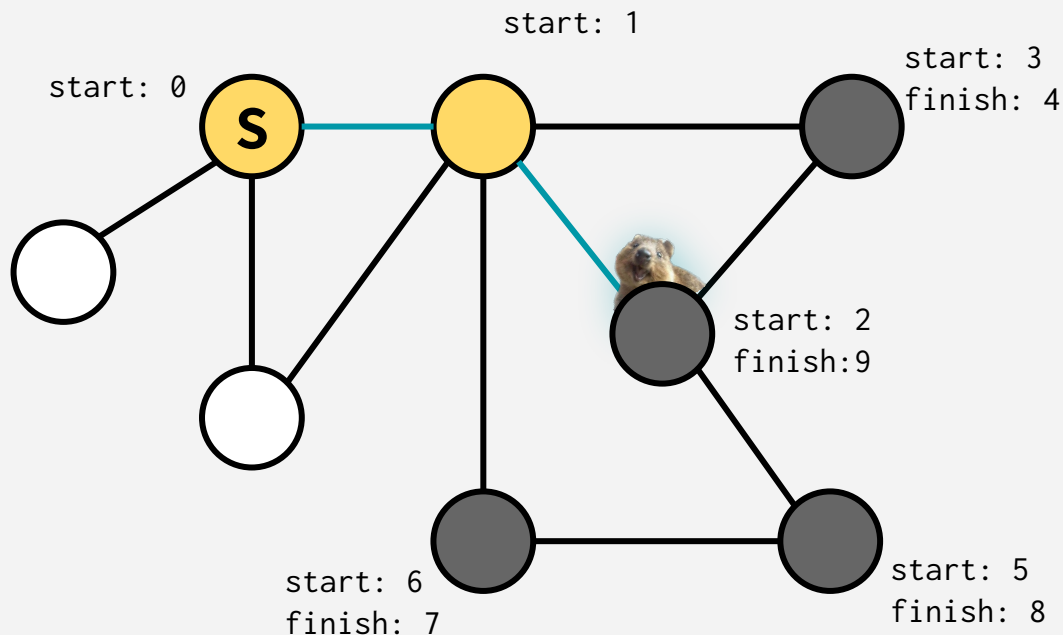
## DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



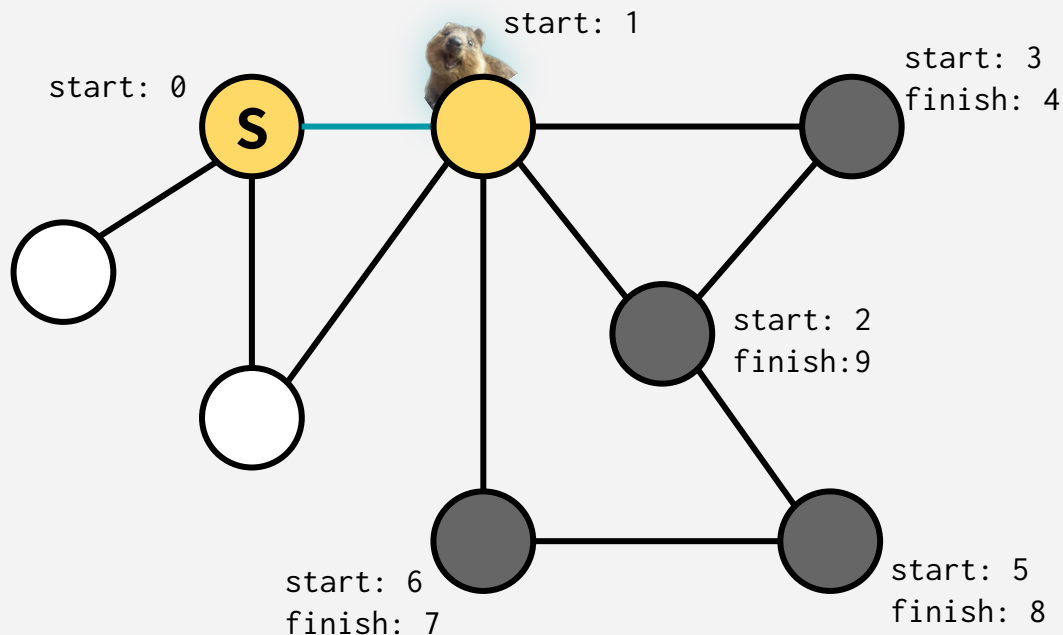
### DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



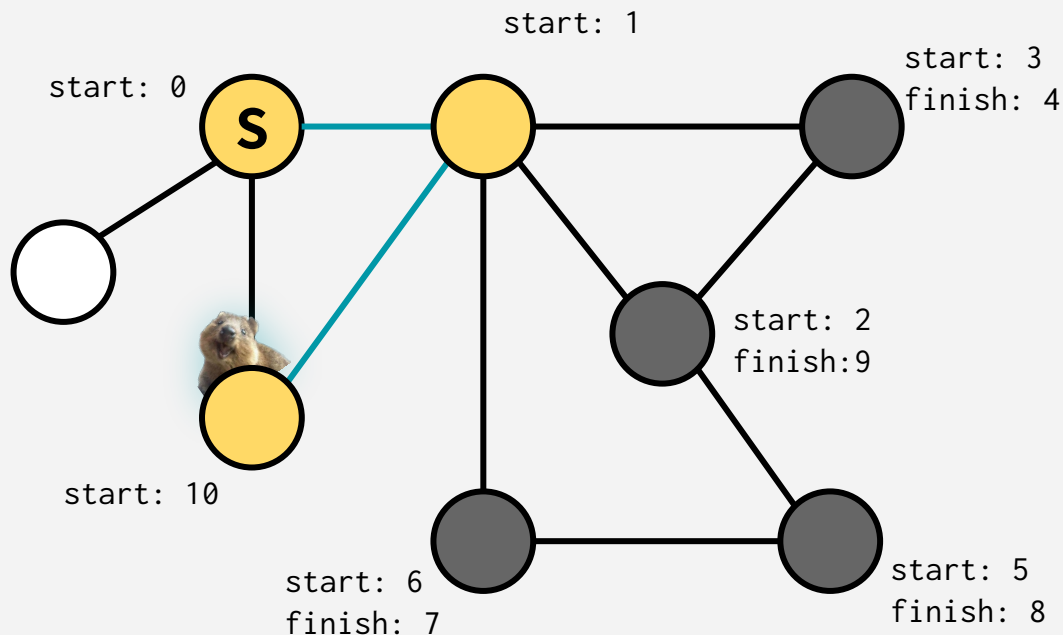
### DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



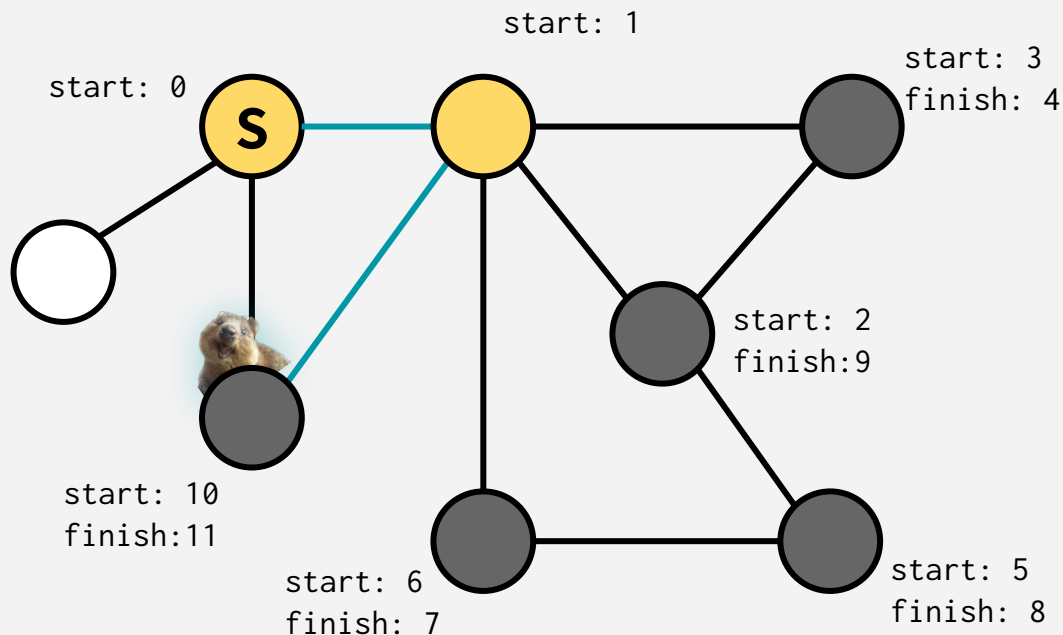
### DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



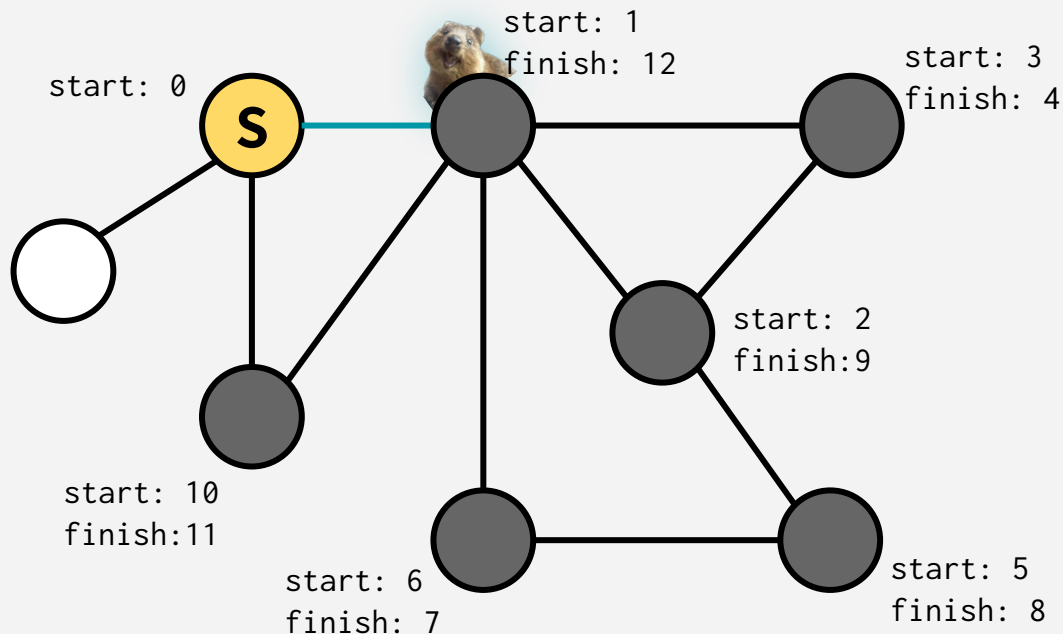
## DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DFS(w, currTime):

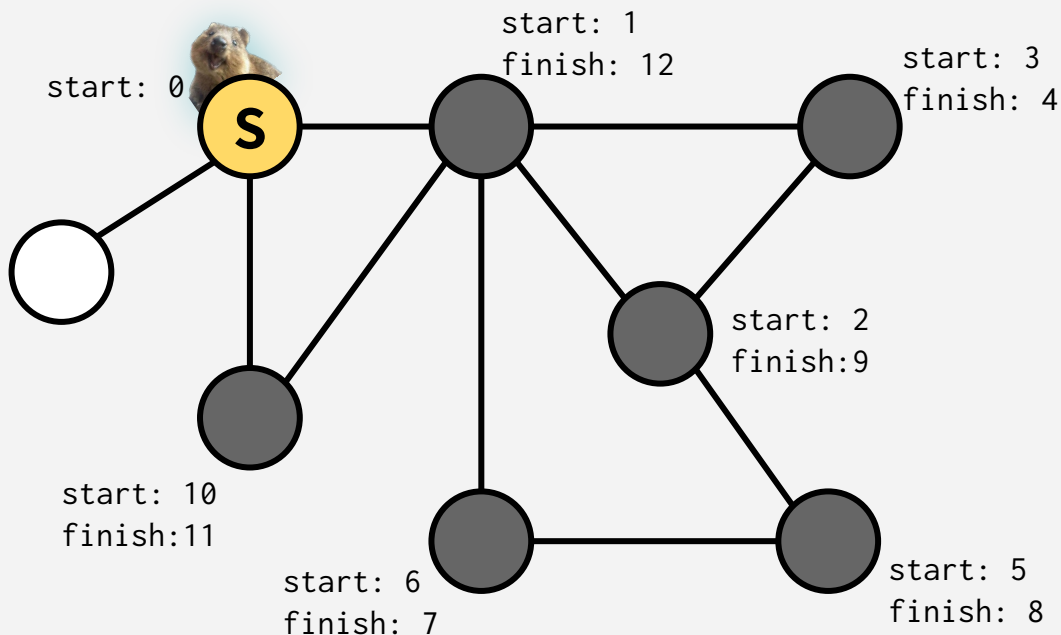
```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```



# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



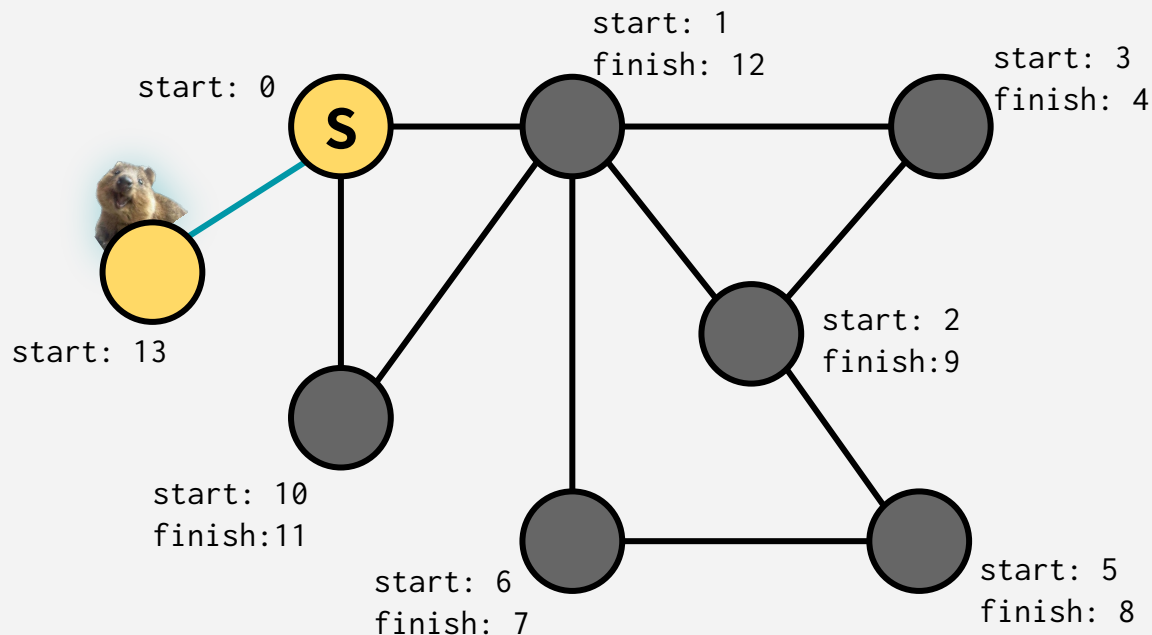
### DFS(w, currTime):

```
w.start = currTime  
currTime++  
mark w as visited  
for v in w.neighbors:  
    if v is unvisited:  
        currTime =  
            DFS(v, currTime)  
        currTime++  
w.finish = currTime  
mark w as finished  
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



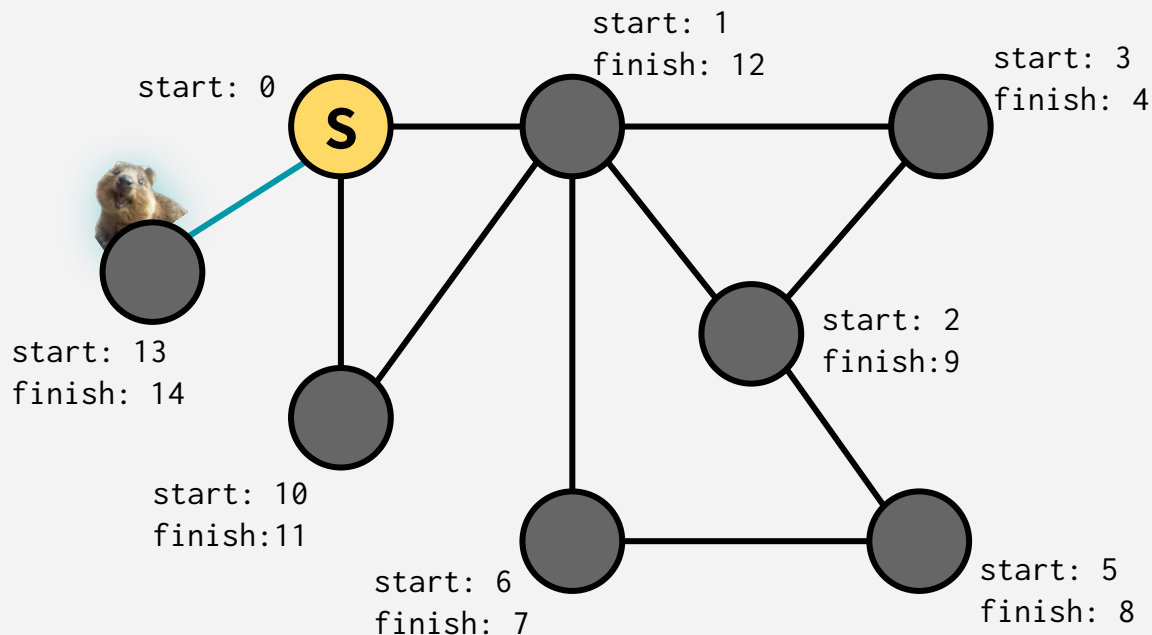
## DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



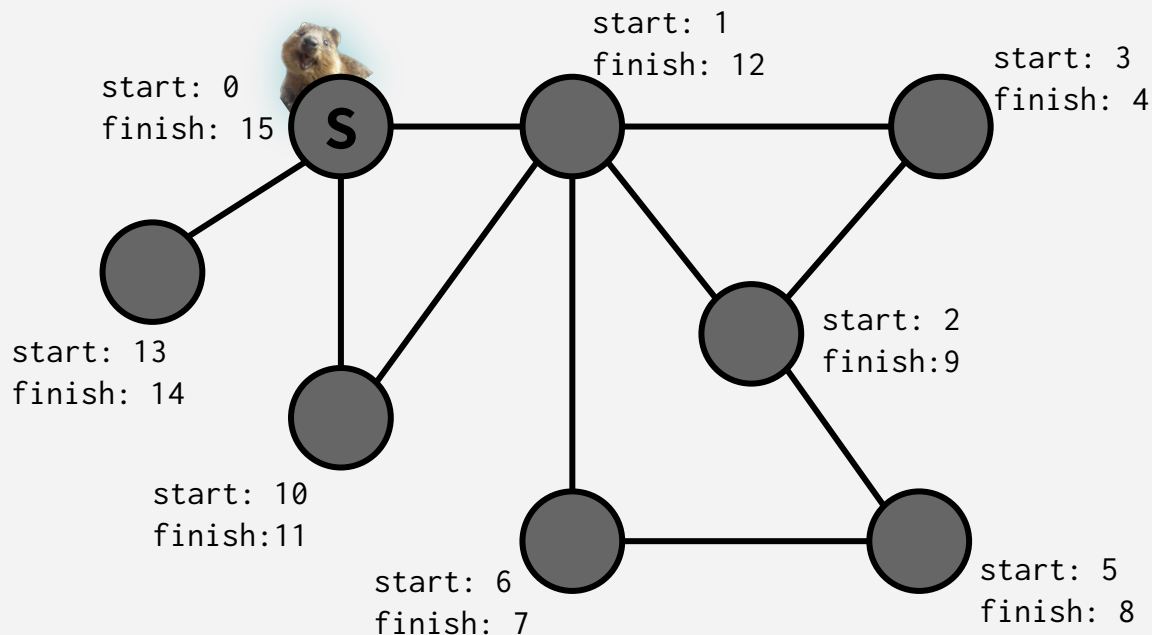
## DFS(w, currTime):

```
w.start = currTime
currTime++
mark w as visited
for v in w.neighbors:
    if v is unvisited:
        currTime =
            DFS(v, currTime)
        currTime++
w.finish = currTime
mark w as finished
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



### DFS(w, currTime):

```
w.start = currTime  
currTime++  
mark w as visited  
for v in w.neighbors:  
    if v is unvisited:  
        currTime =  
            DFS(v, currTime)  
        currTime++  
w.finish = currTime  
mark w as finished  
return currTime
```

# DEPTH-FIRST SEARCH

## An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

This is not the only way to write DFS!  
See the textbook for an iterative version, and try  
writing it yourself (great for interview practice)

start: 13  
finish: 14

start:  
finish:

start: 6  
finish: 7

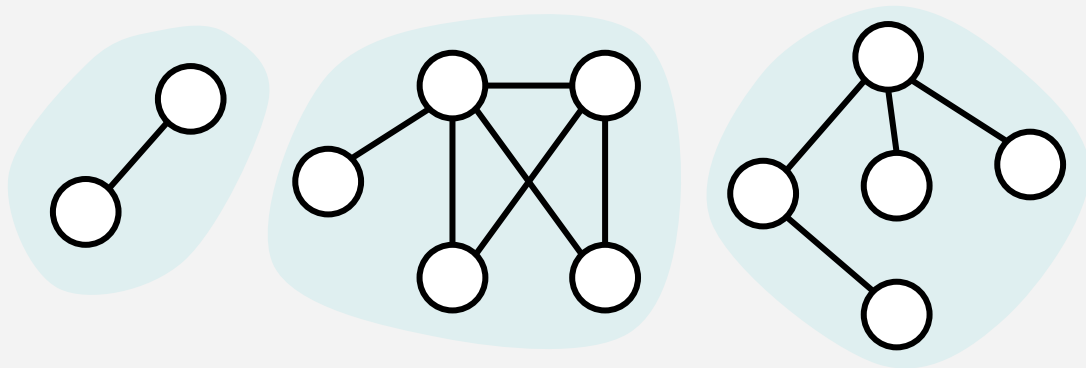
start: 5  
finish: 8

return currTime

# DEPTH-FIRST SEARCH

**Like BFS, DFS finds all the nodes reachable from the starting point!**

In undirected graphs, this is equivalent to finding a **connected component**.



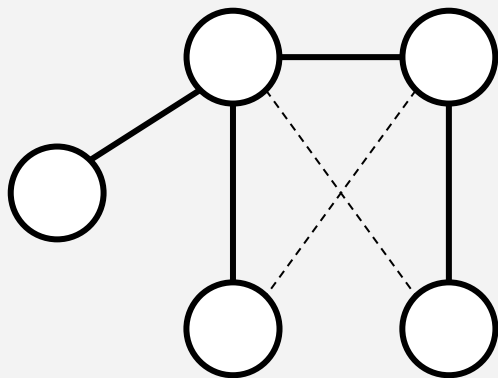
# DEPTH-FIRST SEARCH

## Why is it called depth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We're going as "deep" as we can before "bubbling" back up.



(Edges in the DFS tree are the ones traversed when first finding unvisited nodes)

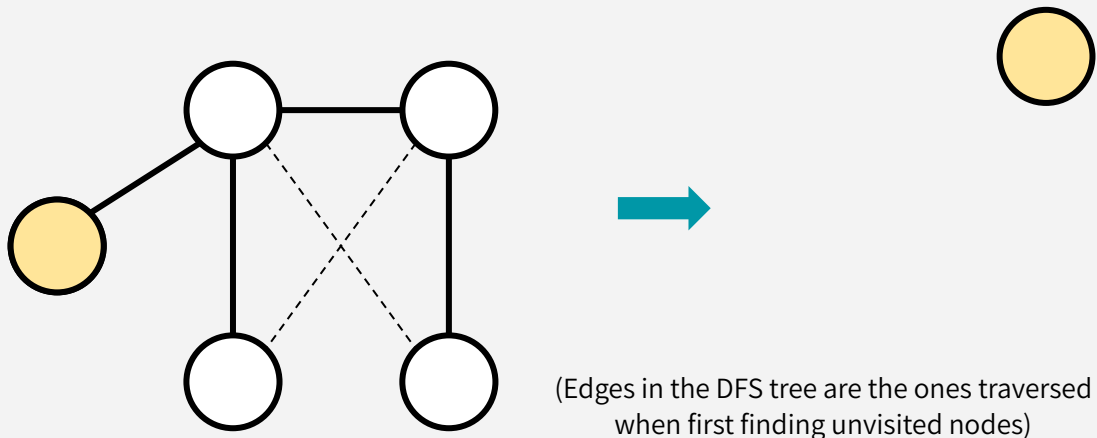
# DEPTH-FIRST SEARCH

## Why is it called depth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We're going as "deep" as we can before "bubbling" back up.





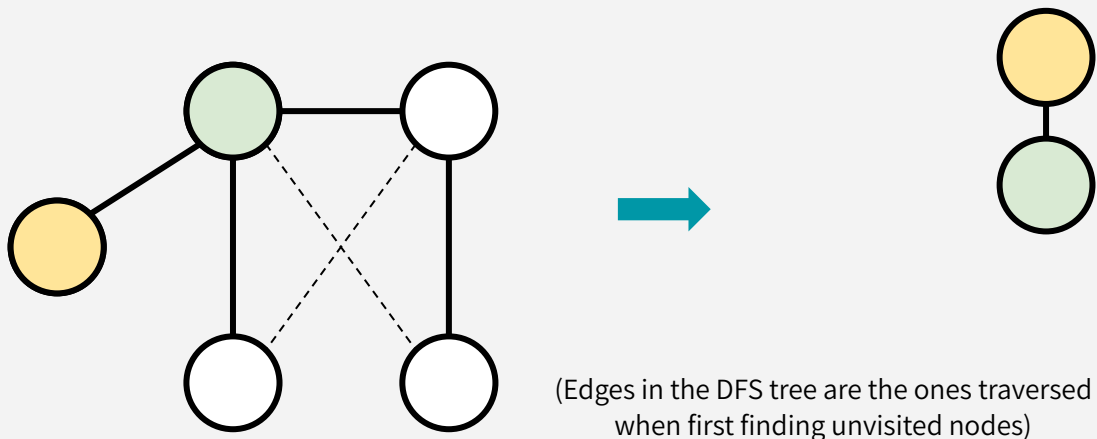
# DEPTH-FIRST SEARCH

## Why is it called depth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We're going as “deep” as we can before “bubbling” back up.



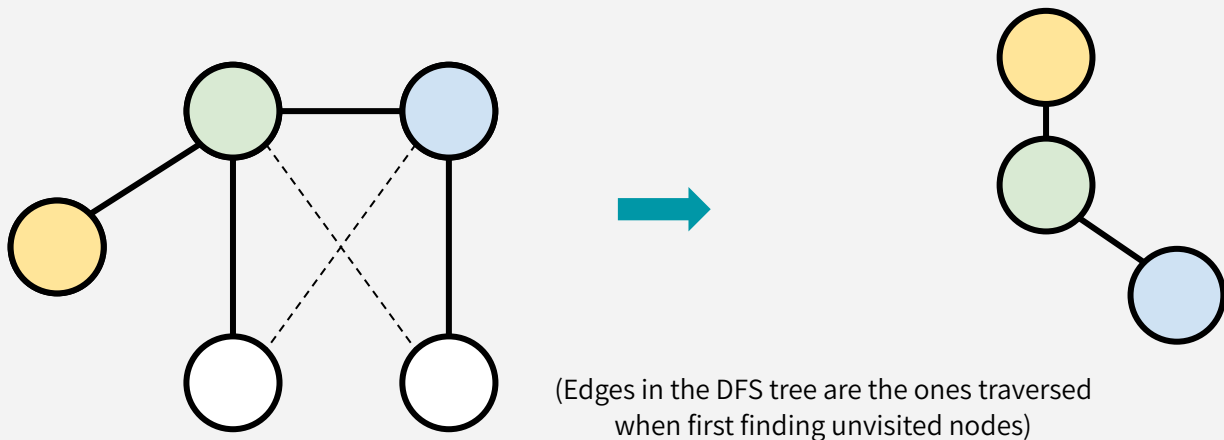
# DEPTH-FIRST SEARCH

## Why is it called depth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We're going as "deep" as we can before "bubbling" back up.



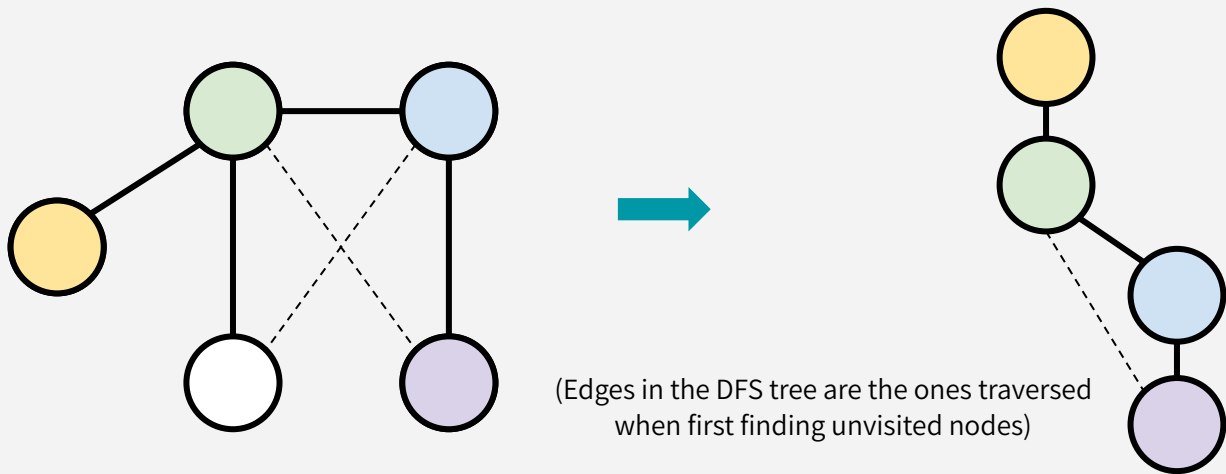
# DEPTH-FIRST SEARCH

## Why is it called depth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We're going as "deep" as we can before "bubbling" back up.



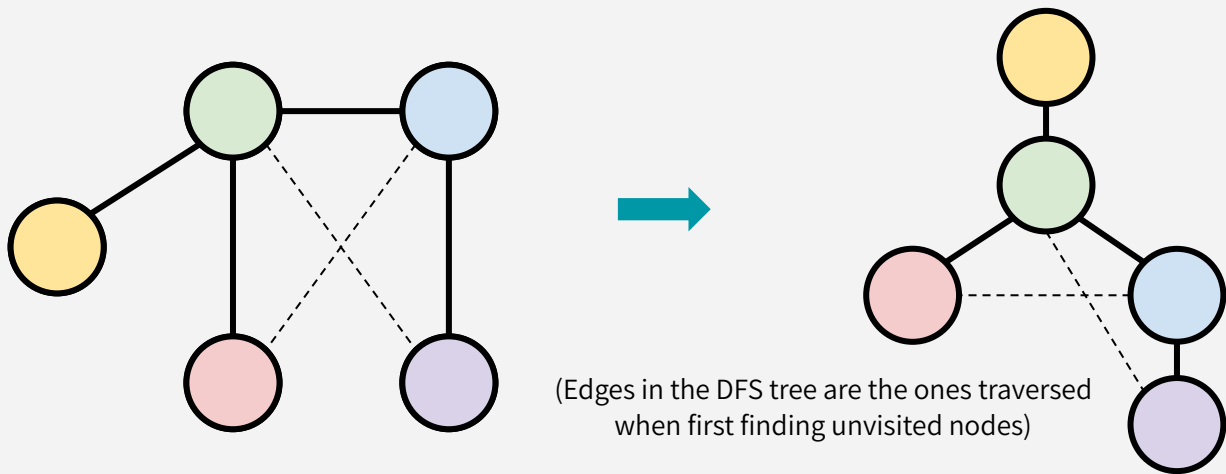
# DEPTH-FIRST SEARCH

## Why is it called depth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We're going as "deep" as we can before "bubbling" back up.



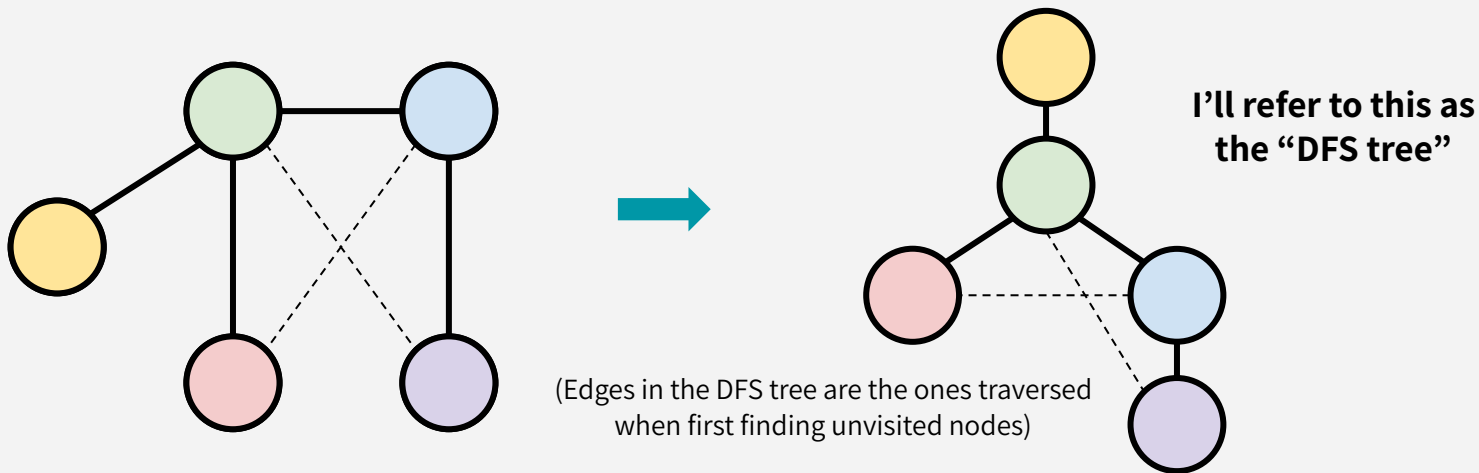
# DEPTH-FIRST SEARCH

## Why is it called depth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We're going as "deep" as we can before "bubbling" back up.



# DEPTH-FIRST SEARCH: RUNTIME

To explore a graph's  **$i^{\text{th}}$  connected component** ( $n_i$  nodes,  $m_i$  edges):

We visit each vertex in the CC exactly once (“visit” = “call DFS on”).

At each vertex  $v$ , we:

- Do some bookkeeping:  **$O(1)$**
- Loop over  $v$ 's neighbors & check if they are visited (& then potentially make a recursive call):  $O(1)$  per neighbor  $\rightarrow$   **$O(\deg(v))$**  total.

$$\textbf{Total: } \sum_v O(\deg(v)) + \sum_v O(1) = \textbf{O}(m_i + n_i)$$

# DEPTH-FIRST SEARCH: RUNTIME

To explore **the entire graph** (n nodes, m edges):

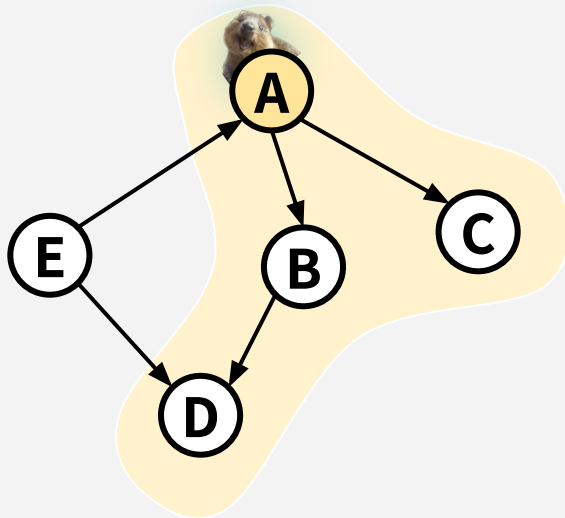
A graph might have multiple connected components! To **explore the whole graph**, we would call our DFS routine once for each connected component (note that each vertex and each edge participates in exactly one connected component). The combined running time would be:

$$O(\sum_i m_i + \sum_i n_i) = \mathbf{O(m + n)}$$

# DEPTH-FIRST SEARCH

## DFS works fine on directed graphs too!

From a start node  $x$ , DFS would find all nodes **reachable** from  $x$ .  
(In directed graphs, “connected component” isn’t as well defined... more on that later!)



### Verify this on your own:

running DFS from A  
would still find all nodes  
reachable from A (E isn't  
reachable from A in this  
directed graph).





سوال؟

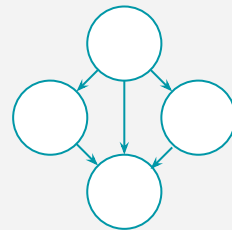
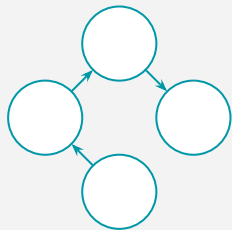
# مرتب سازی توپولوژیکی

**یک کاربرد از جستجوی عمق اول برای مسائل دارای پیش نیاز**

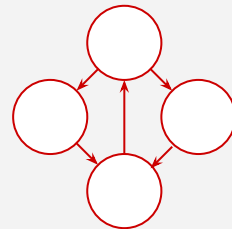
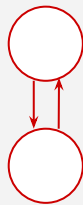
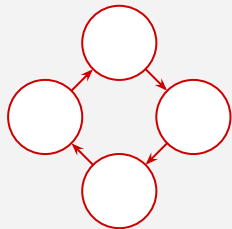
# ASIDE: DIRECTED ACYCLIC GRAPHS

A **Directed Acyclic Graph (DAG)** is a directed graph with *no directed cycles*.

These are DAGs:



These are not DAGs:



# TOPOLOGICAL SORTING: THE TASK

**Given a DAG, find an ordering of vertices so that all of the dependency requirements are met**

Example applications:

Given a package dependency graph, in what order should packages be installed?

Given a course prerequisites graph, in what order should we take classes?

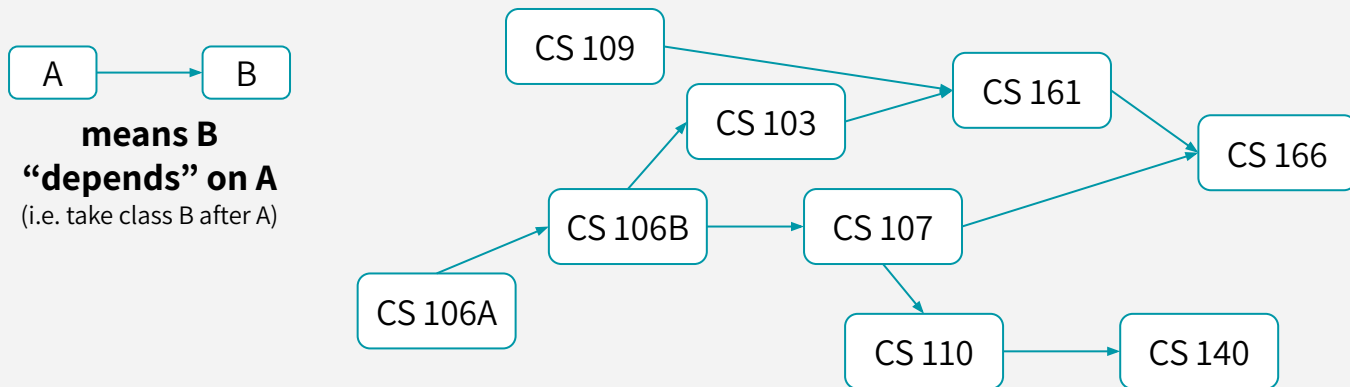
# TOPOLOGICAL SORTING: THE TASK

**Given a DAG, find an ordering of vertices so that all of the dependency requirements are met**

## Example applications:

Given a package dependency graph, in what order should packages be installed?

Given a course prerequisites graph, in what order should we take classes?



# TOPOLOGICAL SORTING: THE TASK

**Given a DAG, find an ordering of vertices so that all of the dependency requirements are met**

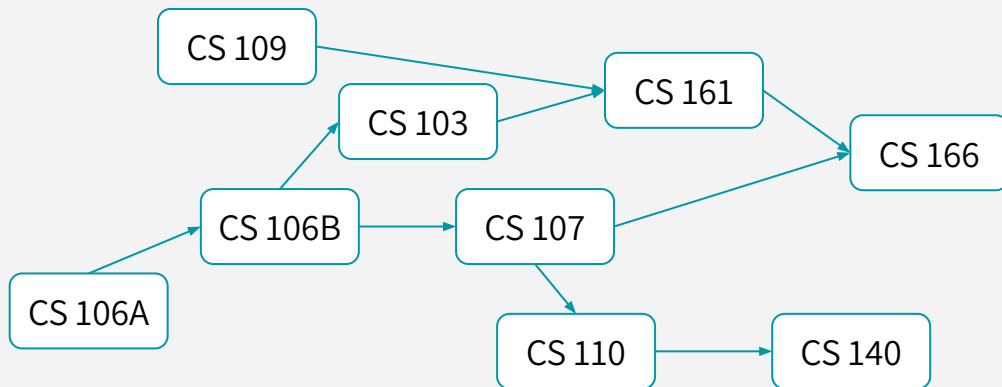
## Example applications:

Given a package dependency graph, in what order should packages be installed?

Given a course prerequisites graph, in what order should we take classes?



**means B  
“depends” on A**  
(i.e. take class B after A)



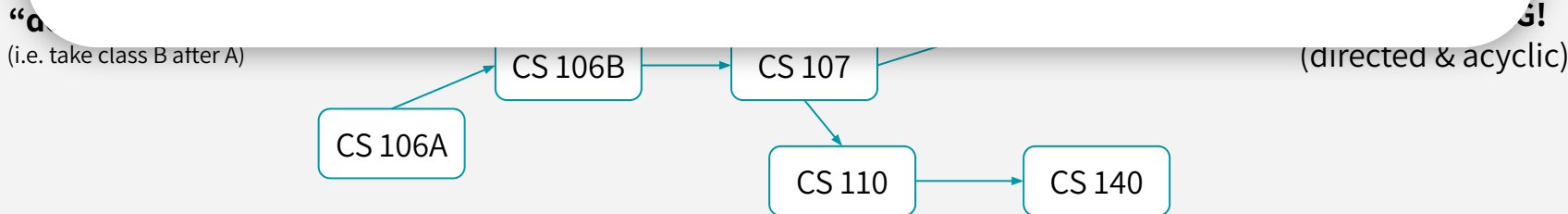
**This prerequisite  
graph is a DAG!**  
(directed & acyclic)

# TOPOLOGICAL SORTING: THE TASK

**Given a DAG, find an ordering of vertices so that all of the dependency requirements are met**

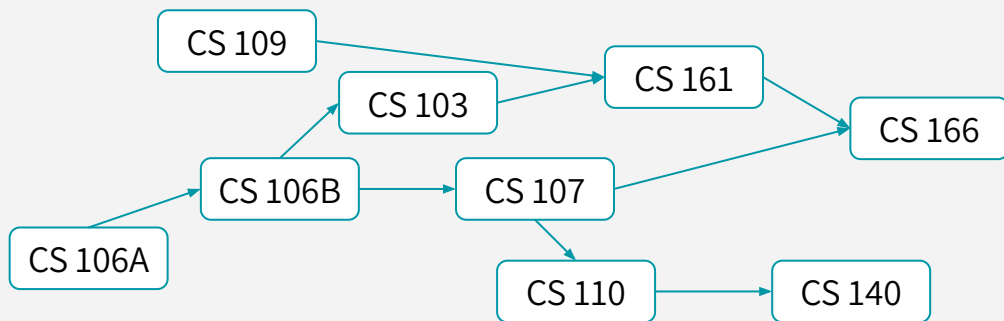
What does “meeting the dependency requirements” mean?

**We want to produce an ordering such that:**  
for every edge  $(v, w)$  in  $E$ ,  $v$  must appear before  $w$  in the ordering  
(e.g. CS103 must come before CS161)



# TOPOLOGICAL SORTING: THE TASK

It's helpful to think of this as “**linearizing**” the graph, where all edges point to the **right**

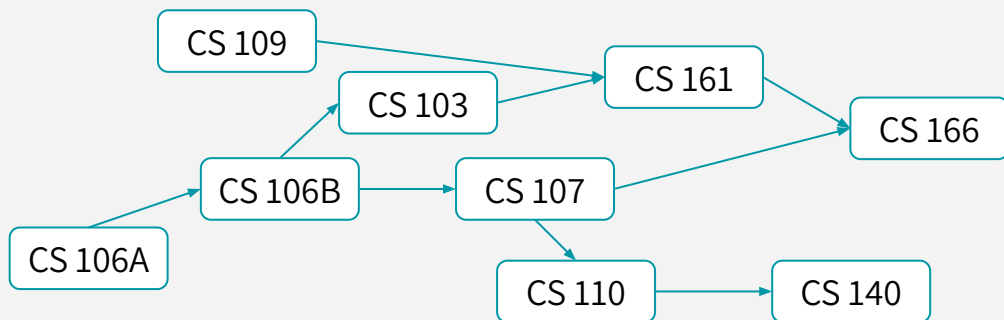


**A correct “toposort” of this DAG:**

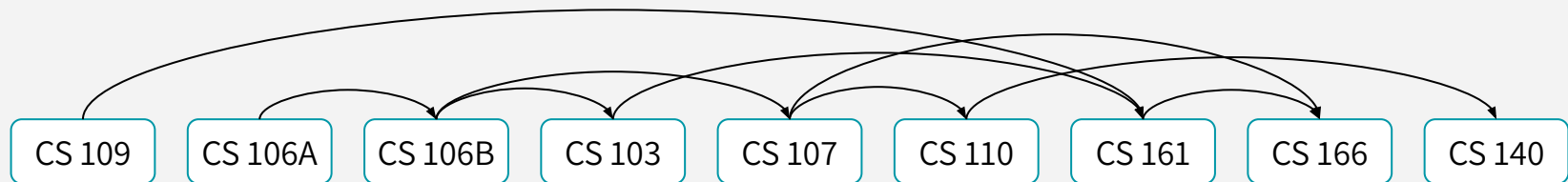


# TOPOLOGICAL SORTING: THE TASK

It's helpful to think of this as “**linearizing**” the graph, where all edges point to the **right**

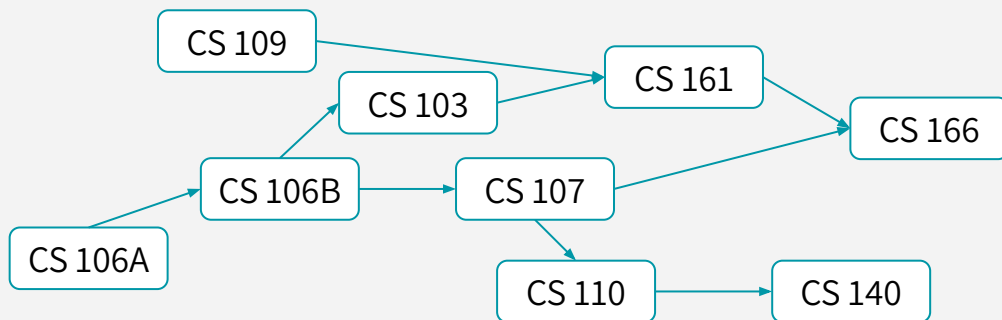


**A correct “toposort” of this DAG:**

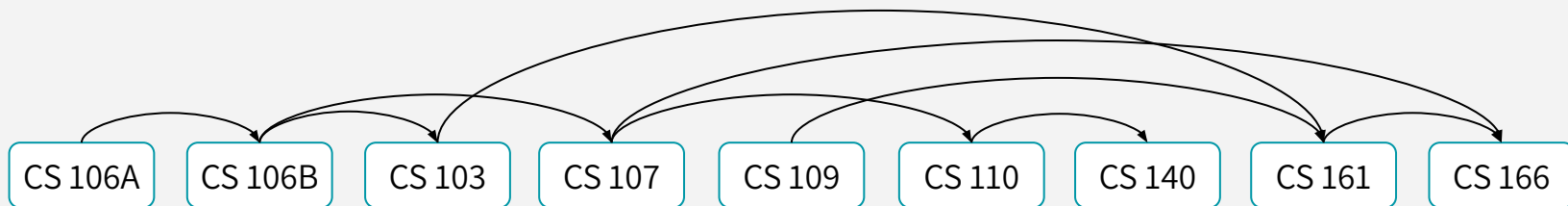


# TOPOLOGICAL SORTING: THE TASK

It's helpful to think of this as “**linearizing**” the graph, where all edges point to the **right**

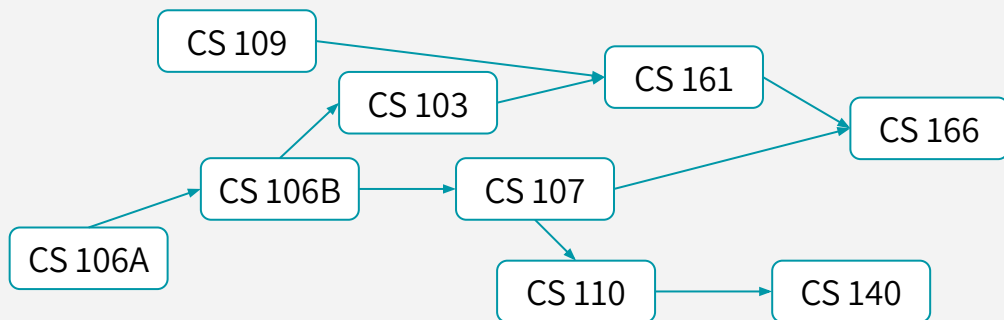


**Also a correct toposort of this DAG:**

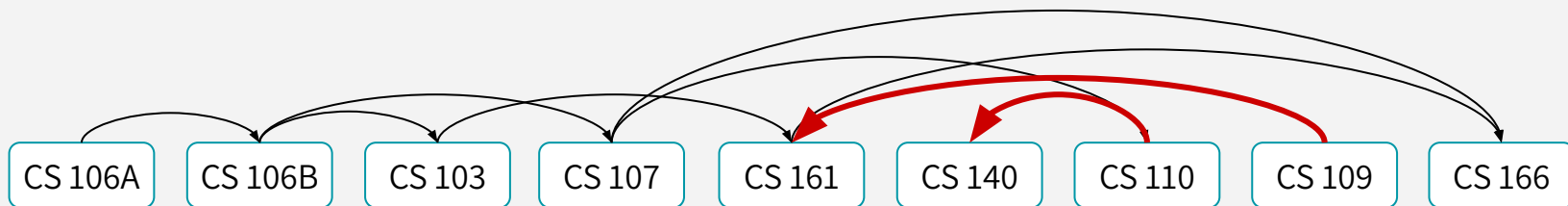


# TOPOLOGICAL SORTING: THE TASK

It's helpful to think of this as “**linearizing**” the graph, where all edges point to the **right**



**Not a correct toposort of this DAG:**



# TOPOSORT ON NON-DAGS?

**We assume these “dependency” graphs are all DAGs!**

What about other graphs? Undirected graphs? Directed graphs with cycles?

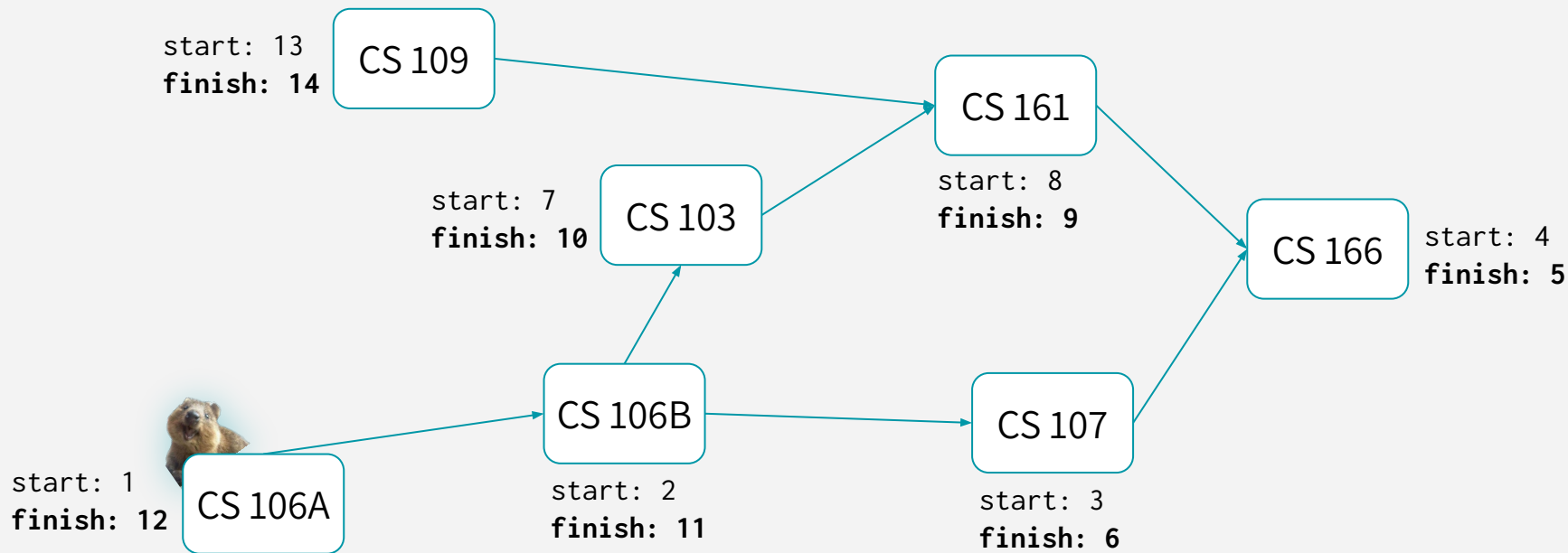
Toposort gives us a priority ordering of nodes (e.g. more intro classes are “higher priority” than more advanced classes). Edges in DAGs clearly illustrate priority: edge from **x** to **y** means **x** has priority over **y**.

In an undirected graph, if there’s an **x-y** edge, which node has “priority”?

In a graph with cycles, if **x** and **y** are part of a cycle, then **x** can reach **y** and **y** can also reach **x**... so which node has “priority”?

# DFS WILL GET US A TOPOSORT

Let's run DFS. What do you notice about the finish times? What does it have to do with toposort?



# DFS WILL GET US A TOPOSORT

Let's run DFS. What do you notice about the finish times? What does it have to do with toposort?

**CLAIM:** In general, if there's an edge from  $\mathbf{v} \rightarrow \mathbf{w}$ ,  $\mathbf{v}$ 's finish time will be *larger* than  $\mathbf{w}$ 's finish time

Let's consider two cases: (1) DFS visits  $\mathbf{v}$  first, or (2) DFS visits  $\mathbf{w}$  first.

start: 1  
finish: 12

CS 106A

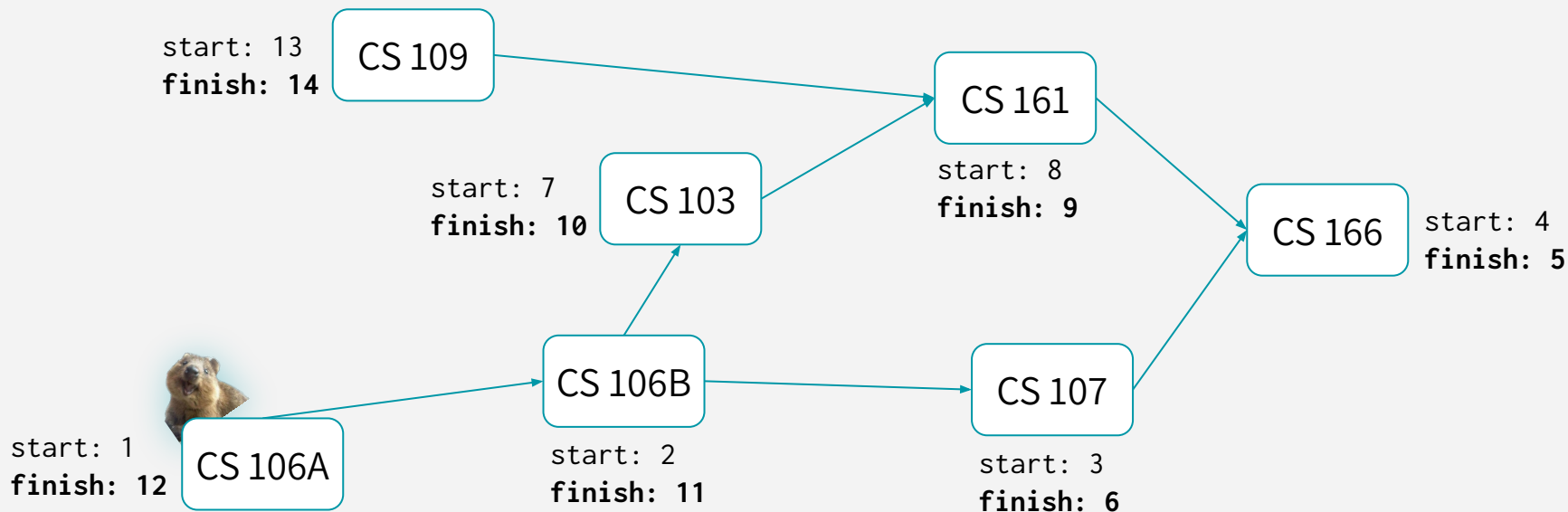
start: 2  
finish: 11

start: 3  
finish: 6

# DFS WILL GET US A TOPOSORT

**CLAIM:** In general, if there's an edge from  $\mathbf{v} \rightarrow \mathbf{w}$ ,  $\mathbf{v}$ 's finish time will be *larger* than  $\mathbf{w}$ 's finish time

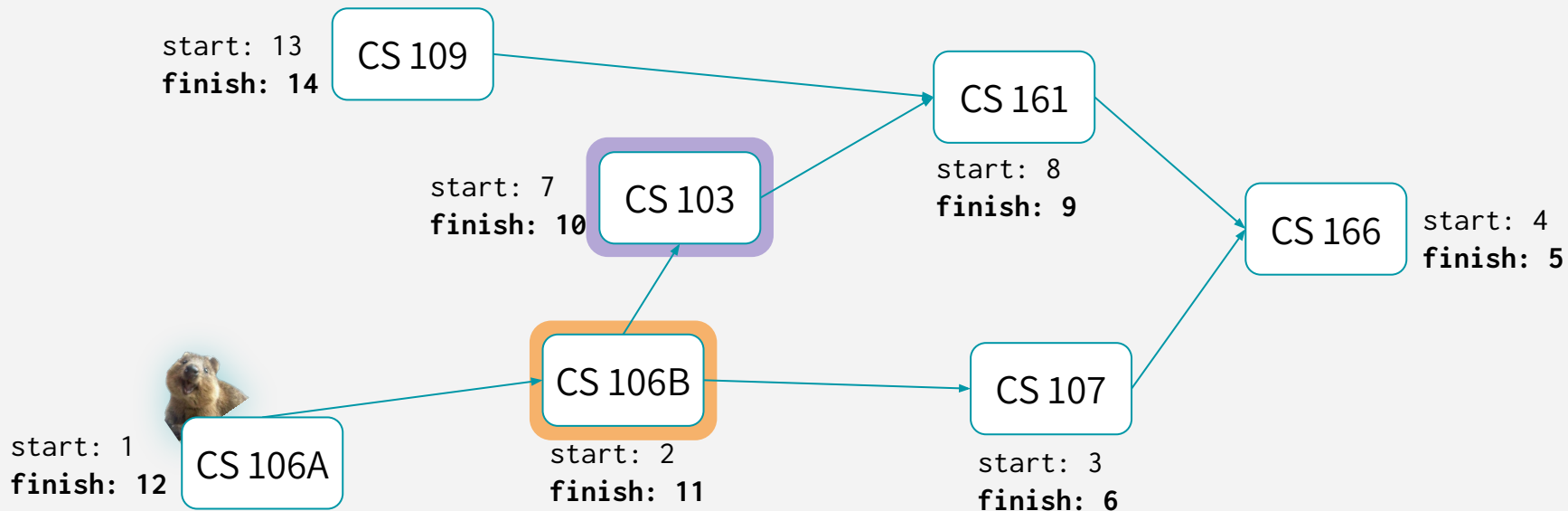
CASE 1:  $\mathbf{v} \rightarrow \mathbf{w}$ , and  $\mathbf{v}$  is **discovered first** by DFS



# DFS WILL GET US A TOPOSORT

**CLAIM:** In general, if there's an edge from  $\mathbf{v} \rightarrow \mathbf{w}$ ,  $\mathbf{v}$ 's finish time will be *larger* than  $\mathbf{w}$ 's finish time

CASE 1:  $\mathbf{v} \rightarrow \mathbf{w}$ , and  $\mathbf{v}$  is **discovered first** by DFS

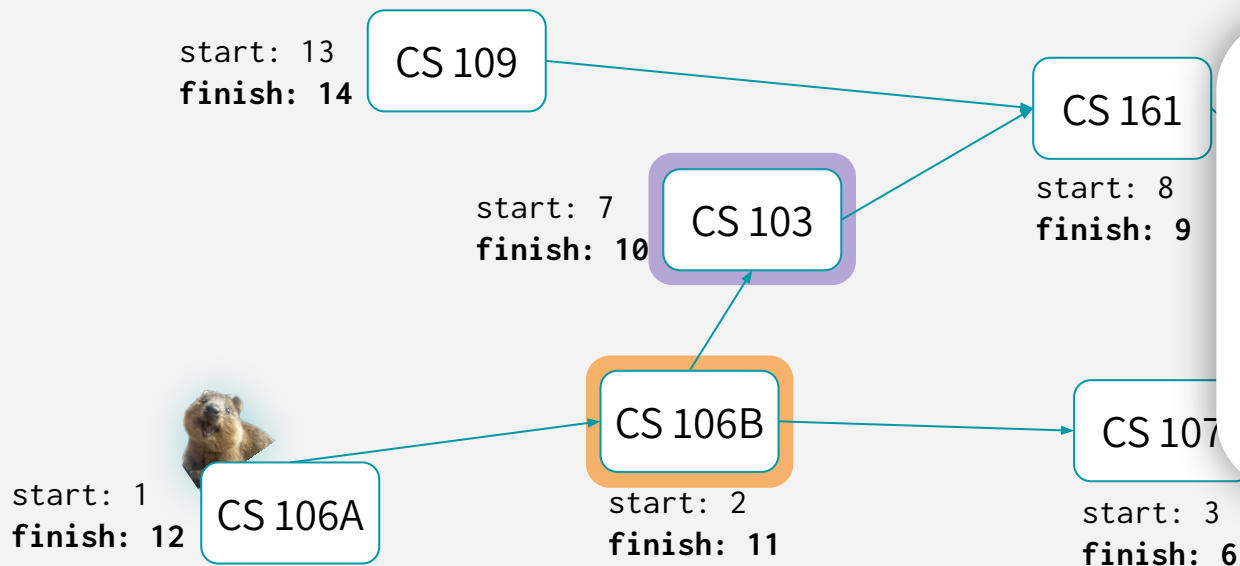




# DFS WILL GET US A TOPOSORT

**CLAIM:** In general, if there's an edge from  $v \rightarrow w$ ,  $v$ 's finish time will be *larger* than  $w$ 's finish time

CASE 1:  $v \rightarrow w$ , and  $v$  is **discovered first** by DFS

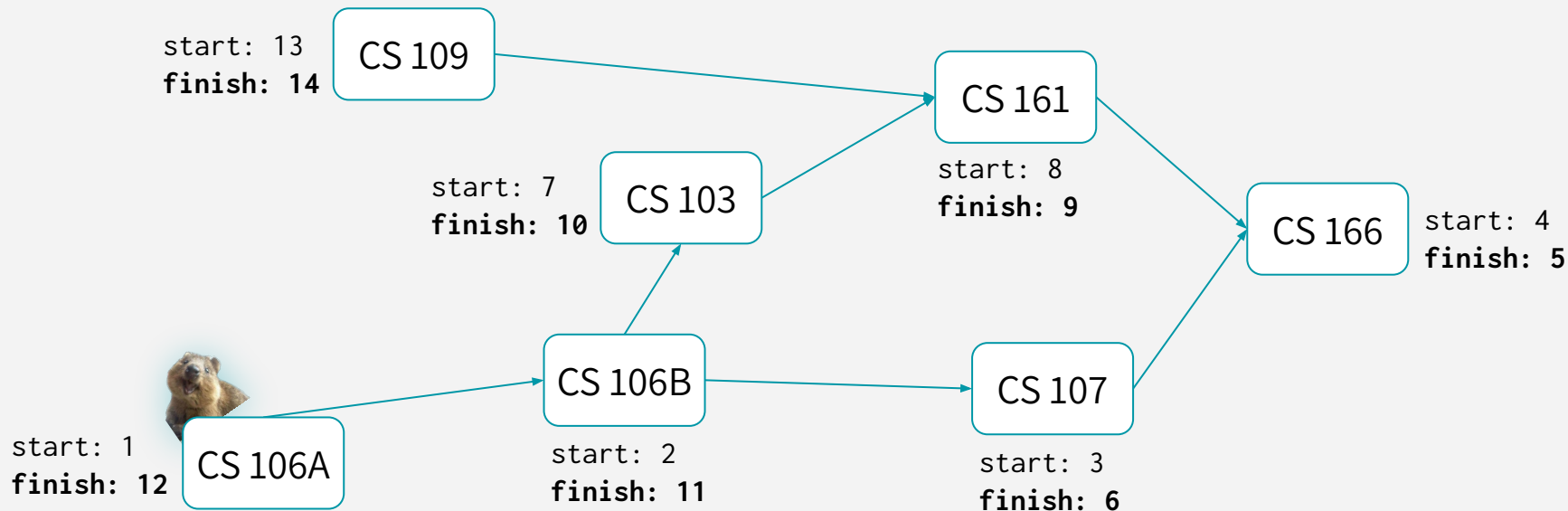


When  $v$  is discovered by DFS, this call will eventually discover  $w$  & recursively call DFS on  $w$ . Then,  $w$  will get its finish time before  $v$  gets its finish time, so  $v.\text{finish} > w.\text{finish}$ !

# DFS WILL GET US A TOPOSORT

**CLAIM:** In general, if there's an edge from  $\mathbf{v} \rightarrow \mathbf{w}$ ,  $\mathbf{v}$ 's finish time will be *larger* than  $\mathbf{w}$ 's finish time

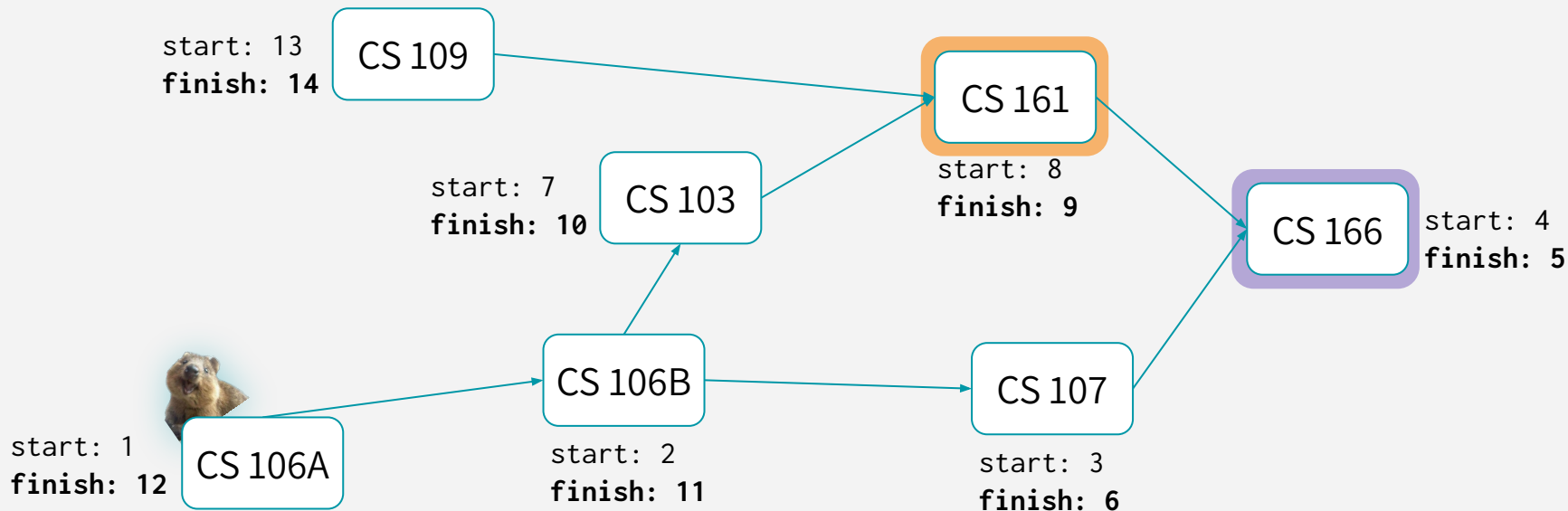
CASE 2:  $\mathbf{v} \rightarrow \mathbf{w}$ , and  $\mathbf{w}$  is **discovered first** by DFS



# DFS WILL GET US A TOPOSORT

**CLAIM:** In general, if there's an edge from  $v \rightarrow w$ ,  $v$ 's finish time will be *larger* than  $w$ 's finish time

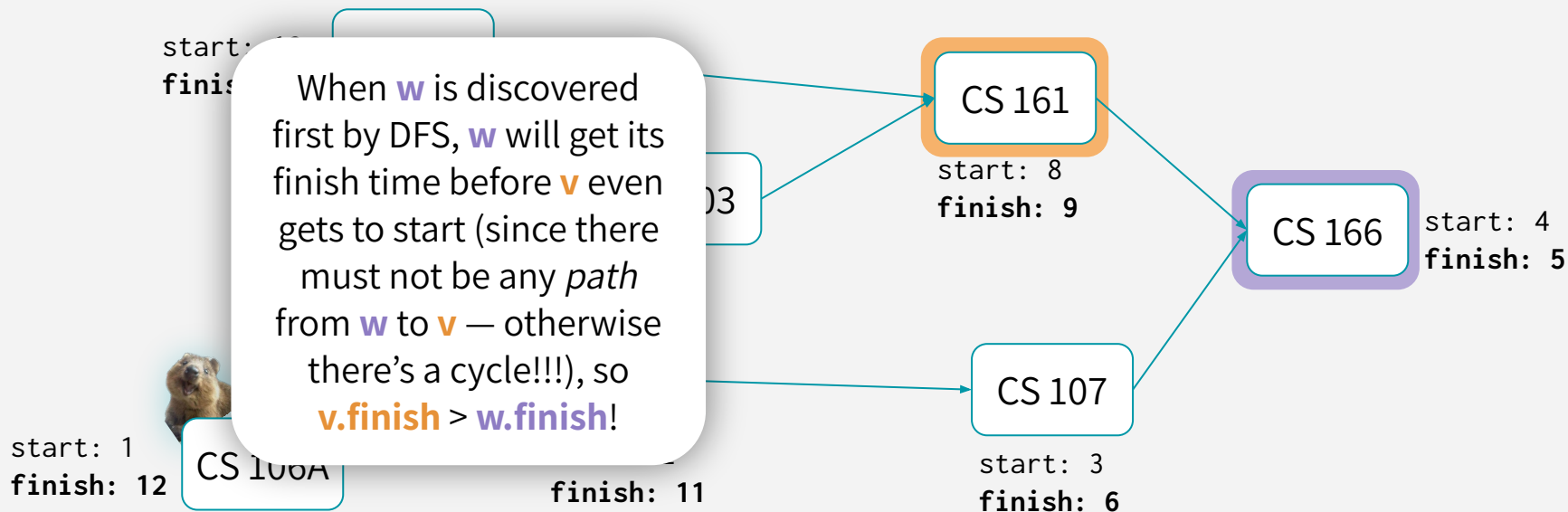
CASE 2:  $v \rightarrow w$ , and  $w$  is **discovered first** by DFS



# DFS WILL GET US A TOPOSORT

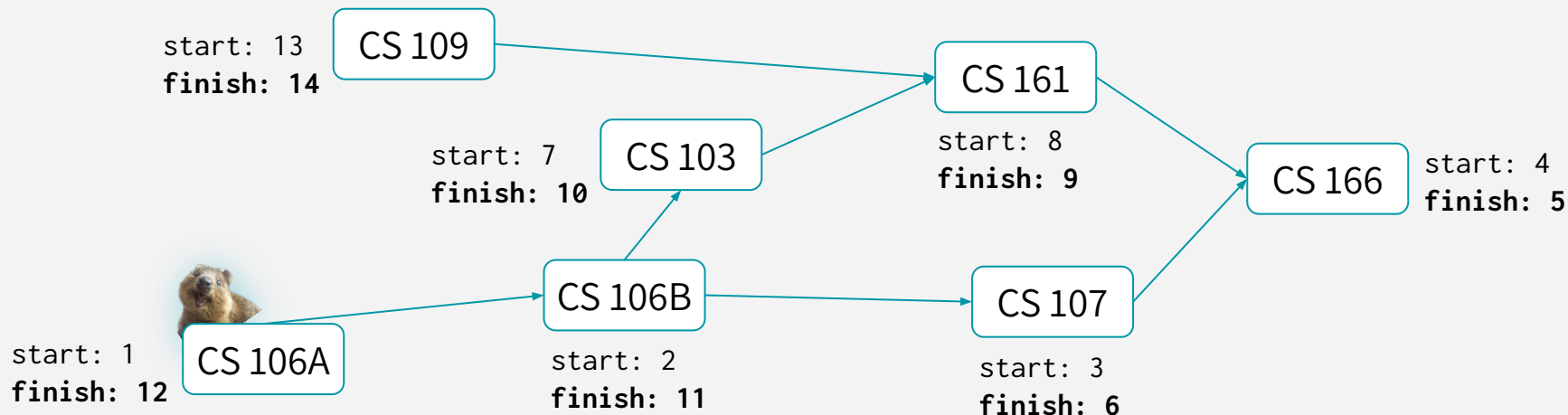
**CLAIM:** In general, if there's an edge from  $\mathbf{v} \rightarrow \mathbf{w}$ ,  $\mathbf{v}$ 's finish time will be *larger* than  $\mathbf{w}$ 's finish time

CASE 2:  $\mathbf{v} \rightarrow \mathbf{w}$ , and  $\mathbf{w}$  is **discovered first** by DFS



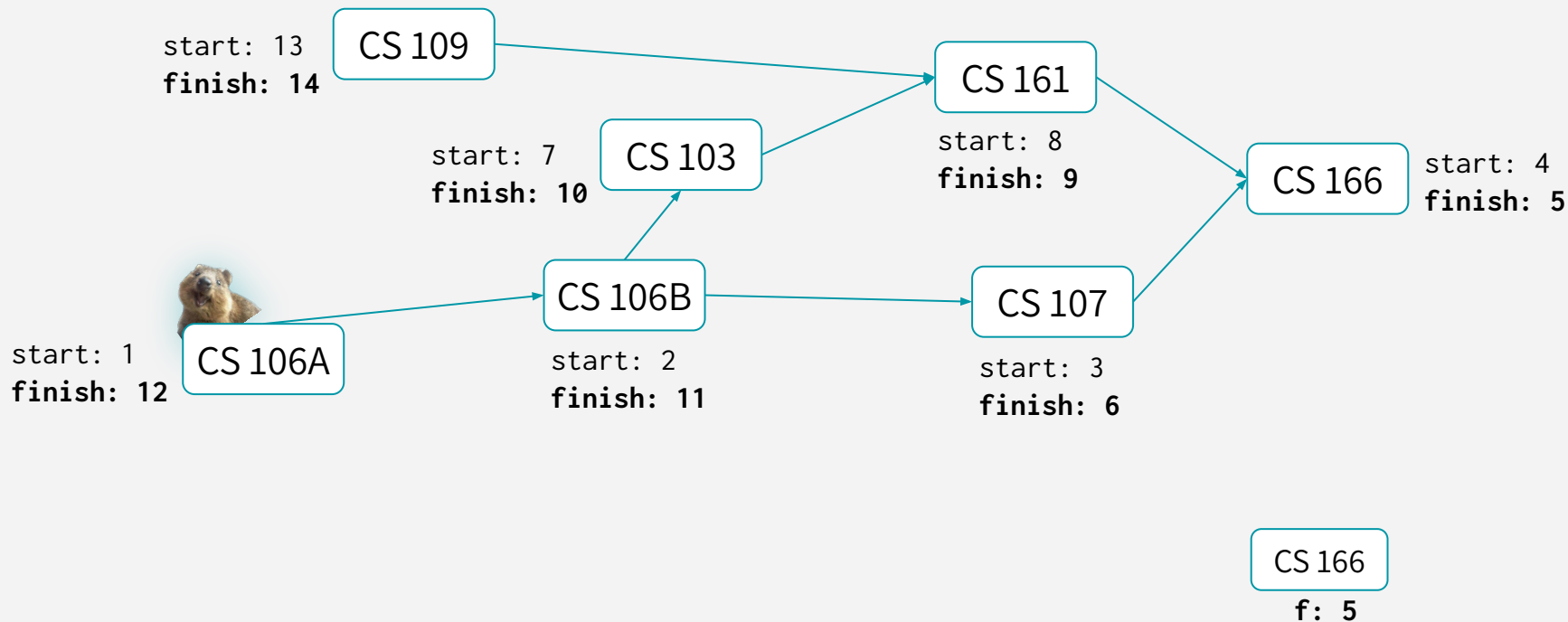
# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



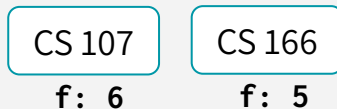
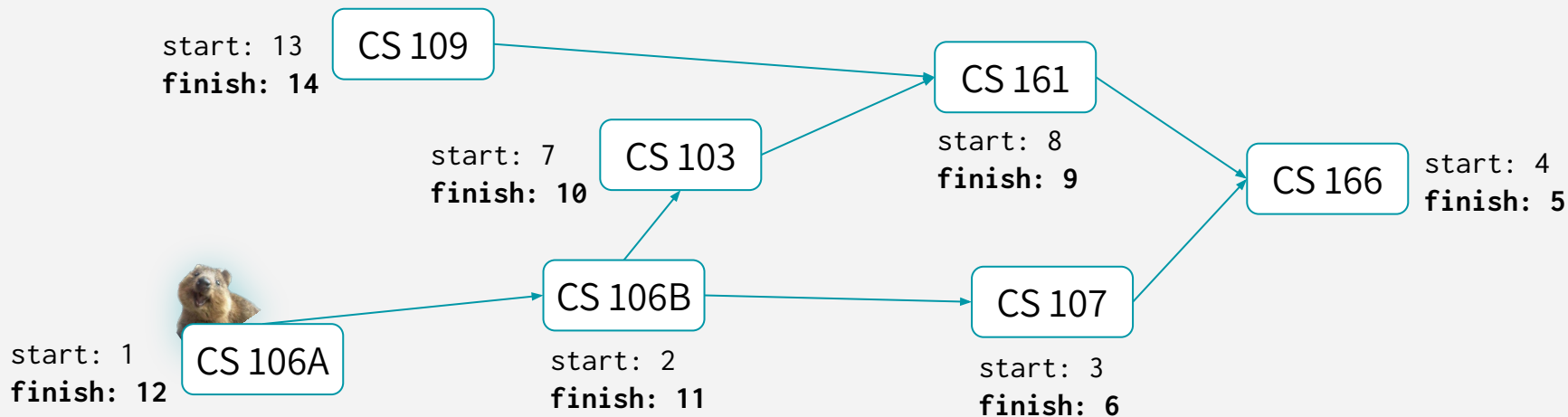
# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



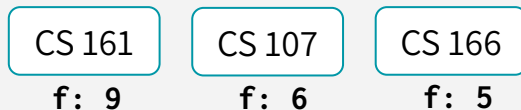
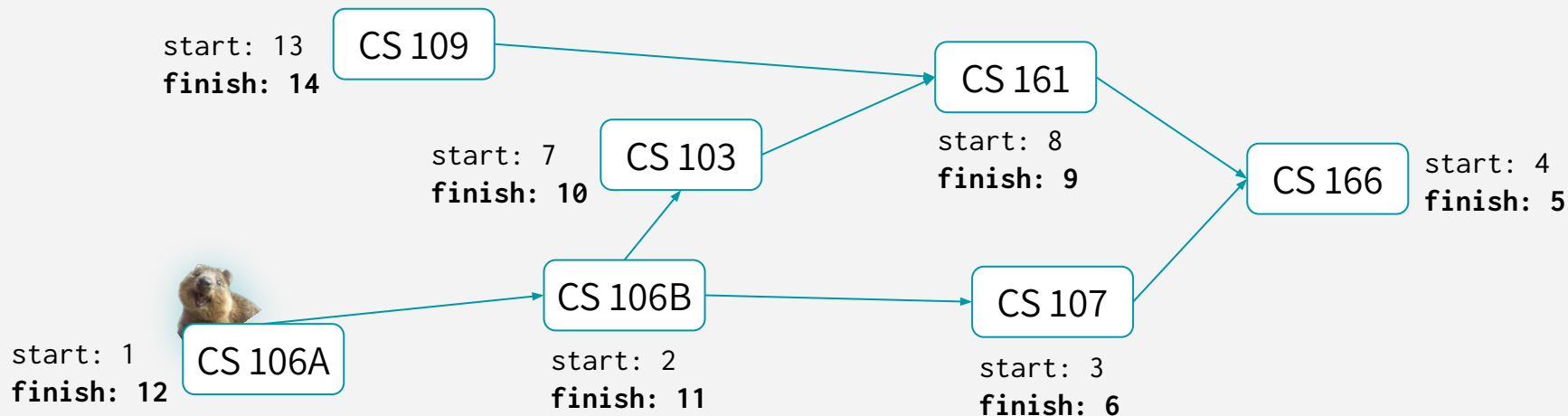
# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



# DFS WILL GET US A TOPOSORT

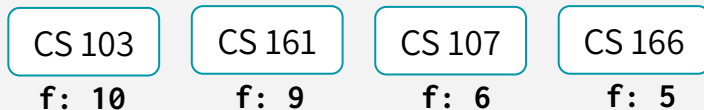
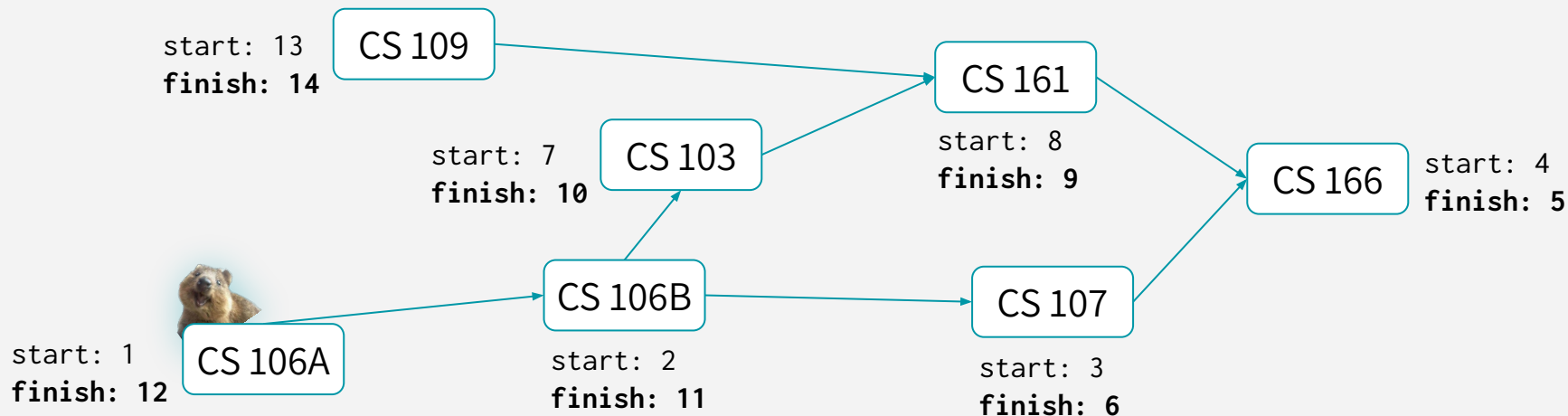
**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.





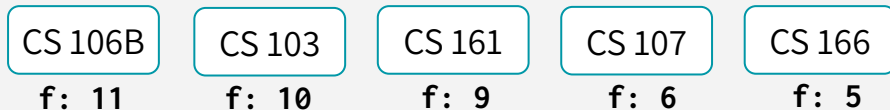
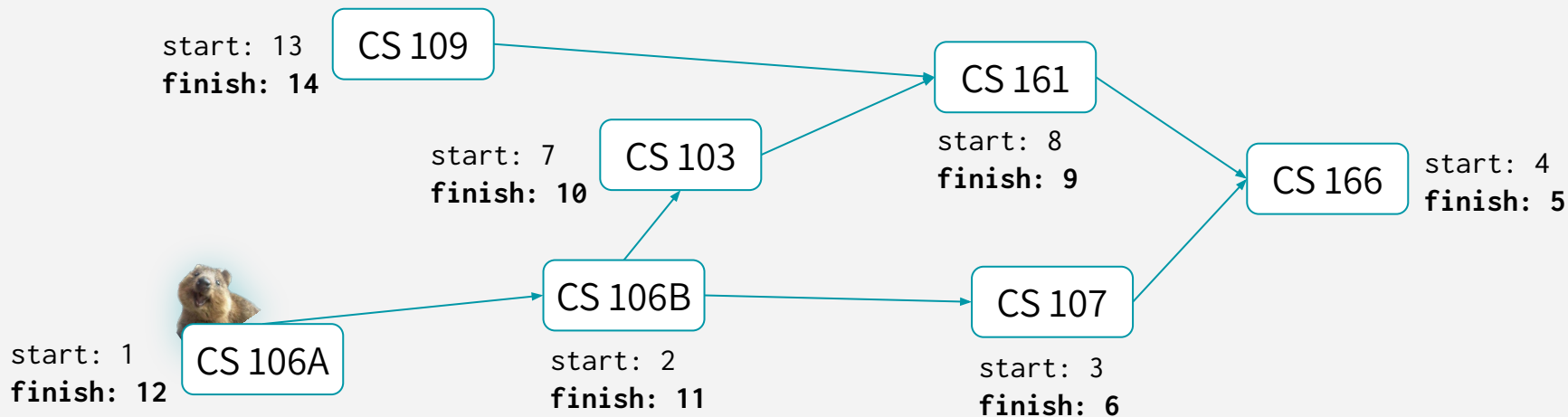
# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



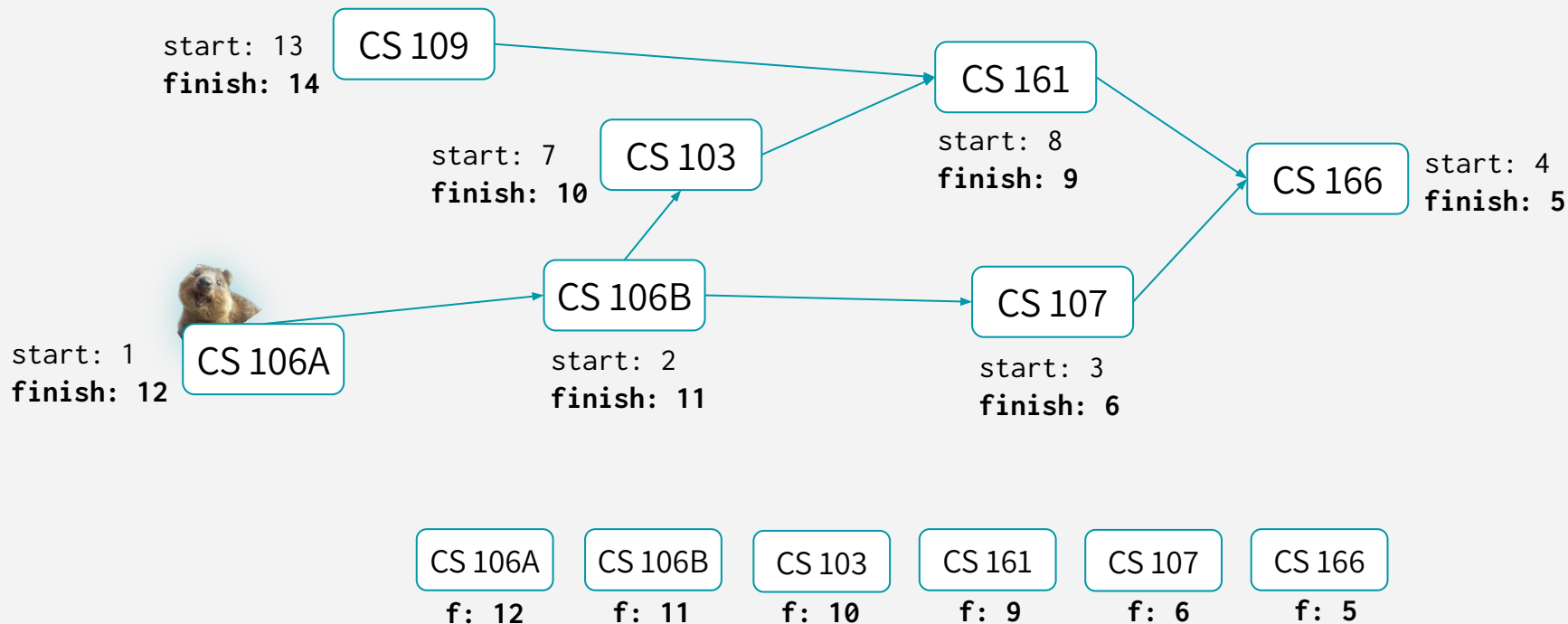
# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



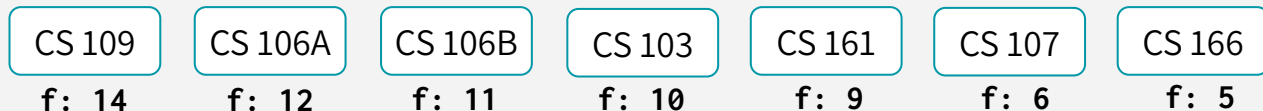
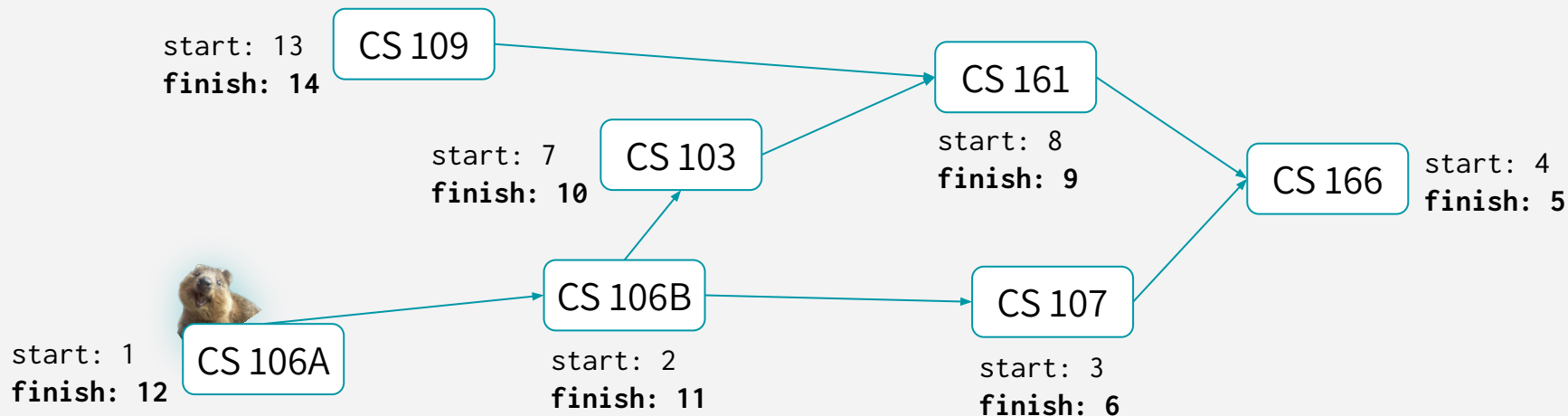
# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



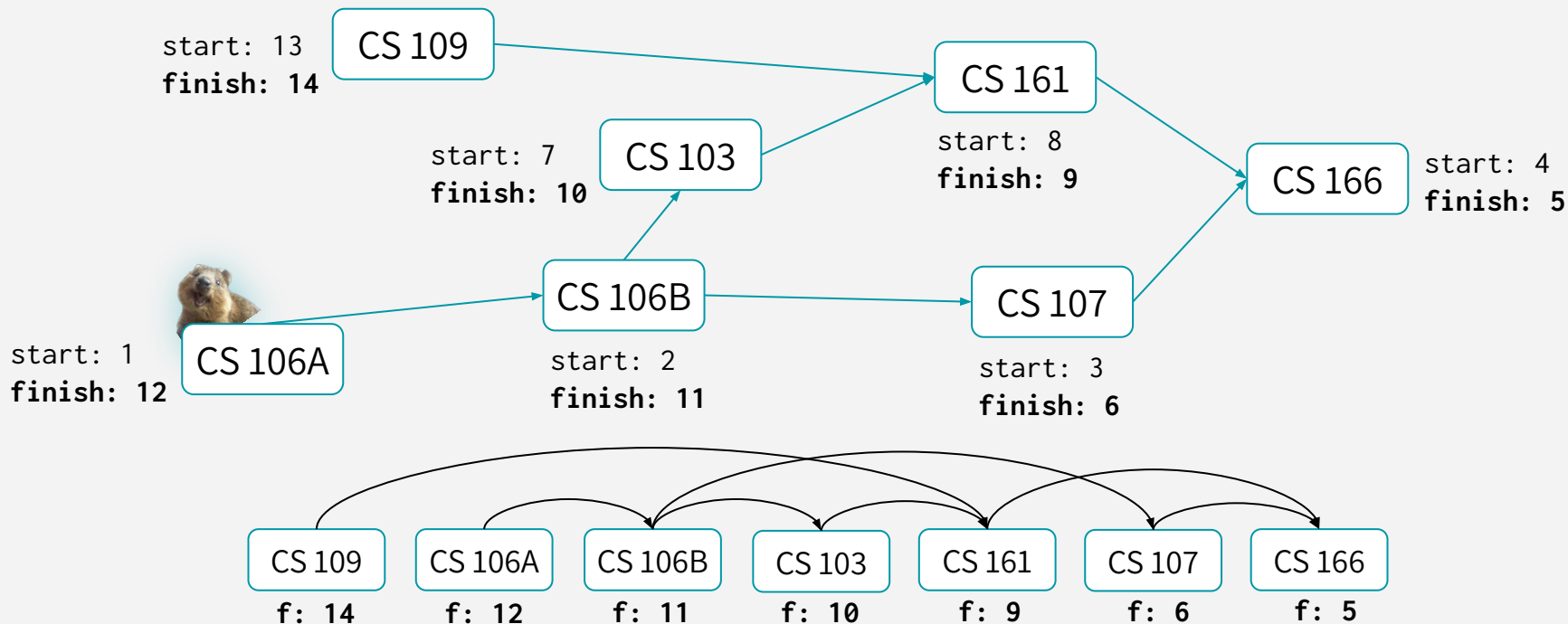
# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



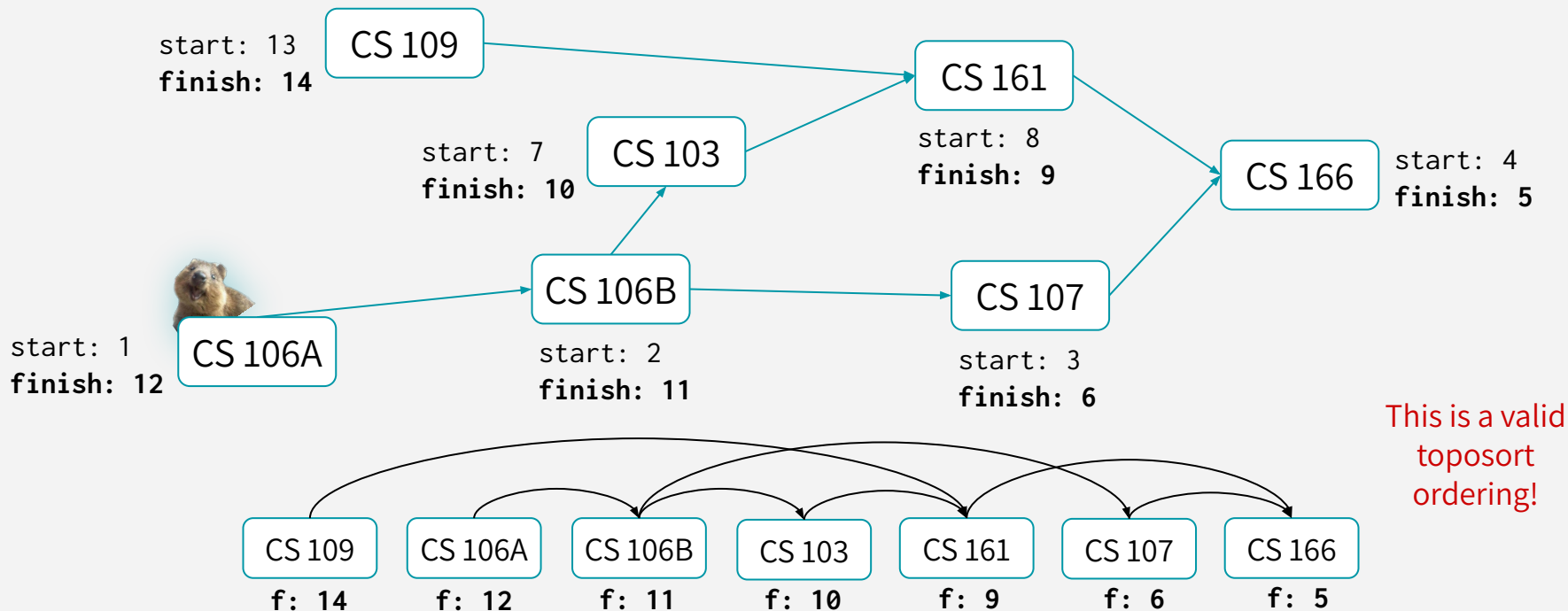
# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



# DFS WILL GET US A TOPOSORT

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.

start: 13  
finish: 14

CS 109

CS 161

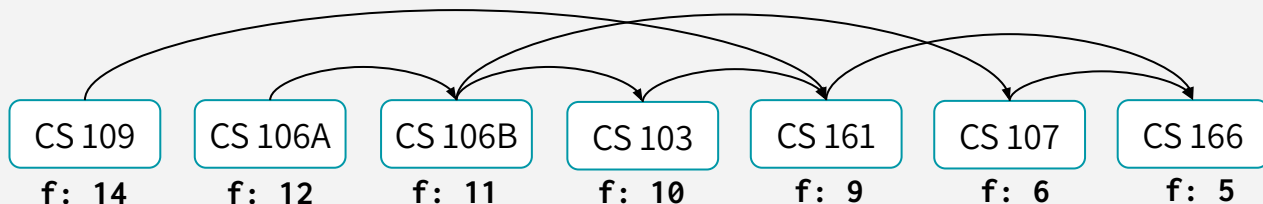
Regardless of which vertex your DFS starts, it'll get you a correct Toposort ordering of your DAG

start: 1  
finish: 12

CS 106A

start: 2  
finish: 11

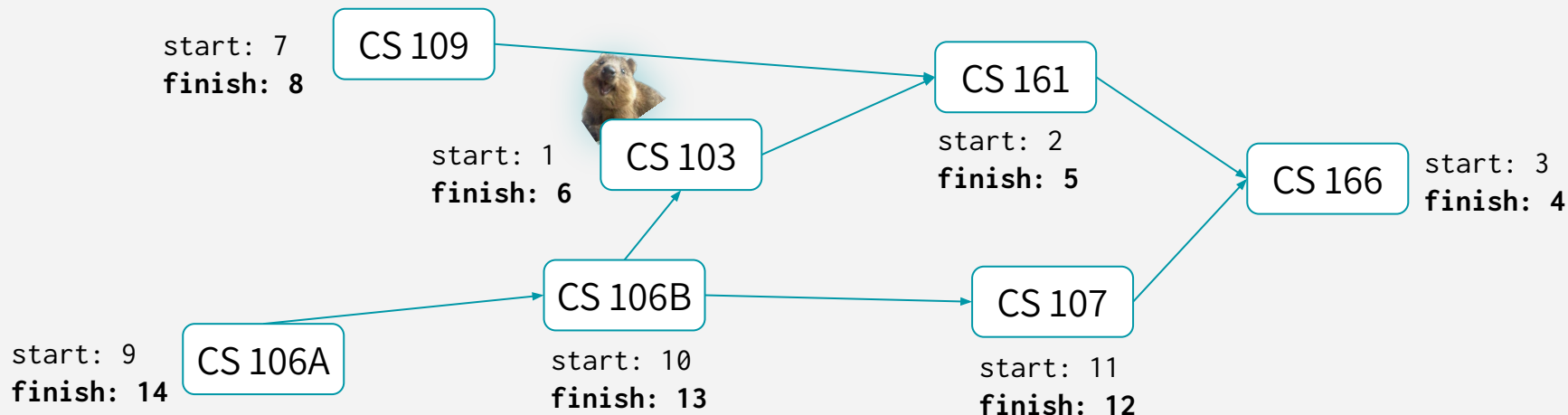
start: 3  
finish: 6



# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.

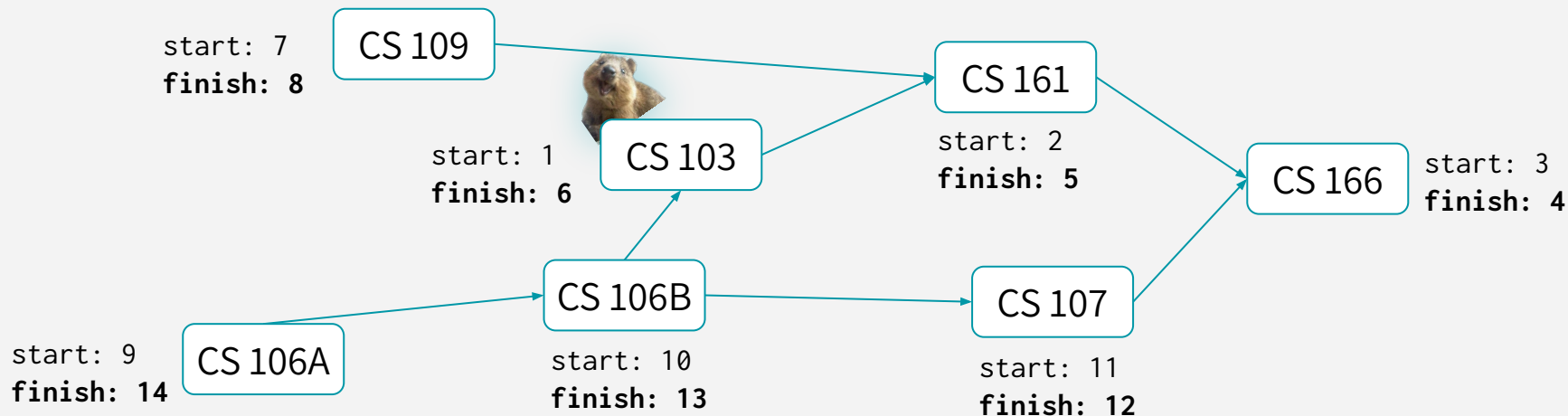




# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



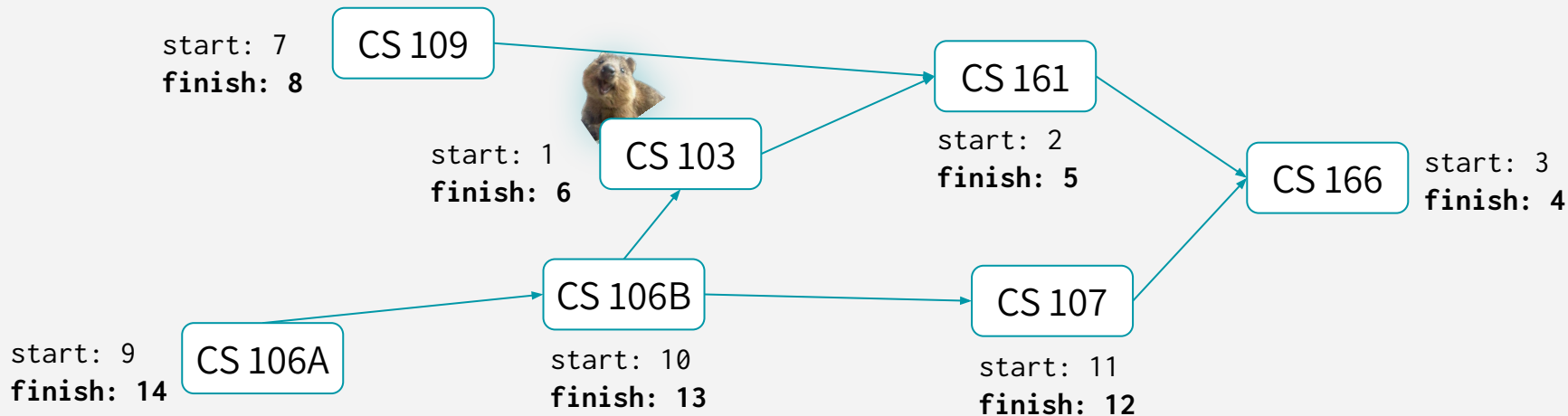
CS 166

f: 4

# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



CS 161

f: 5

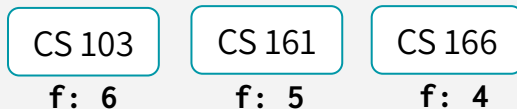
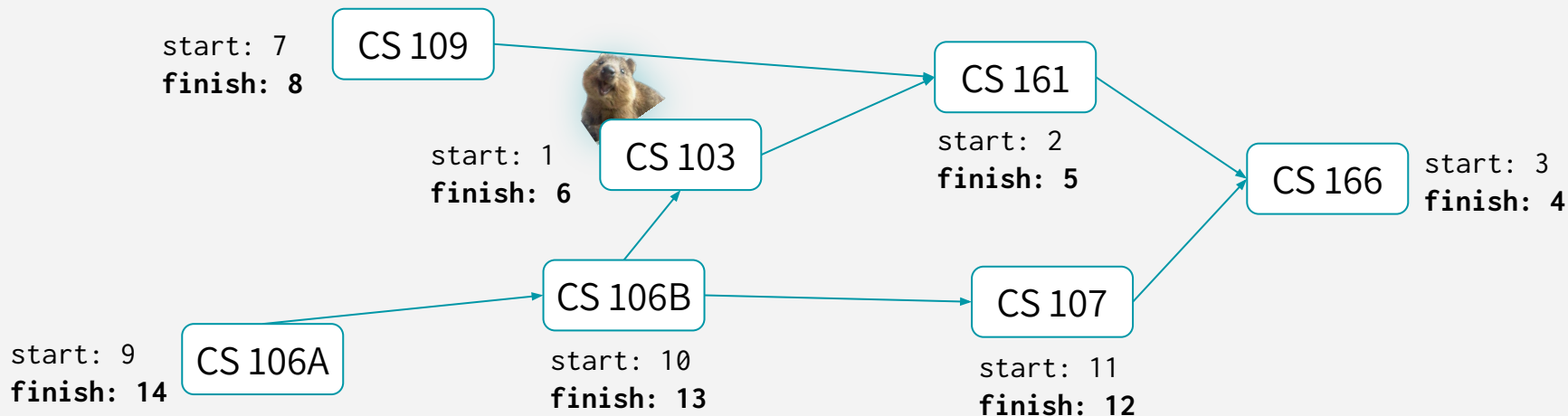
CS 166

f: 4

# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

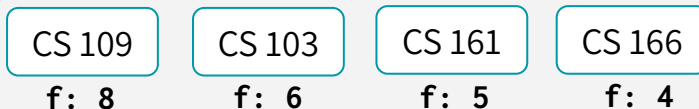
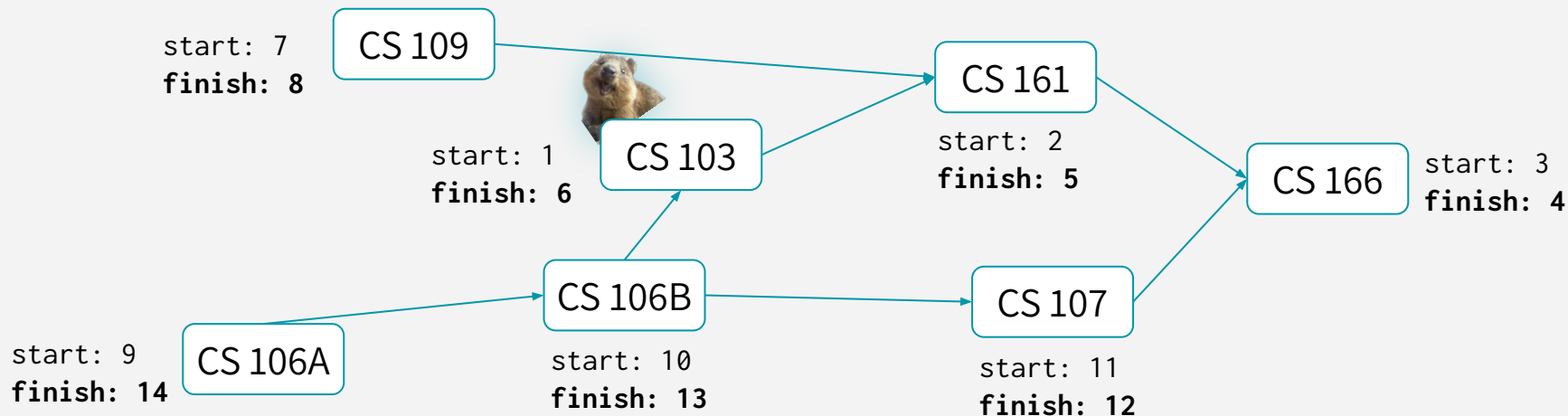
**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

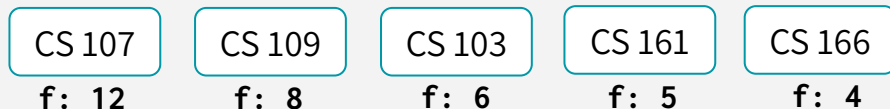
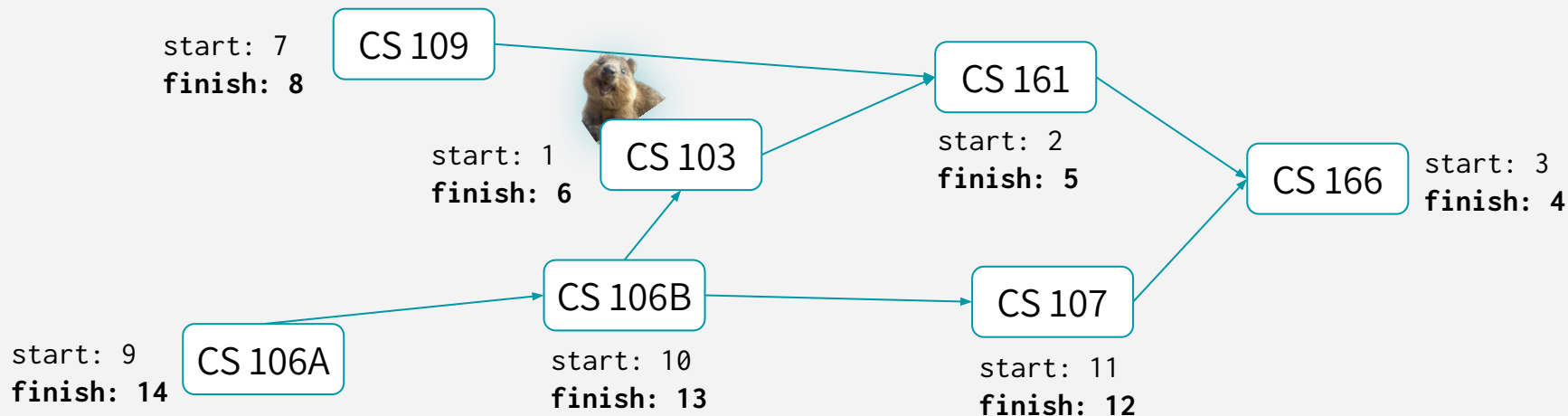
**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

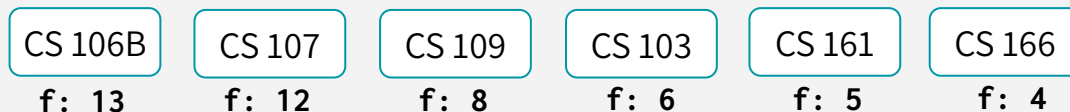
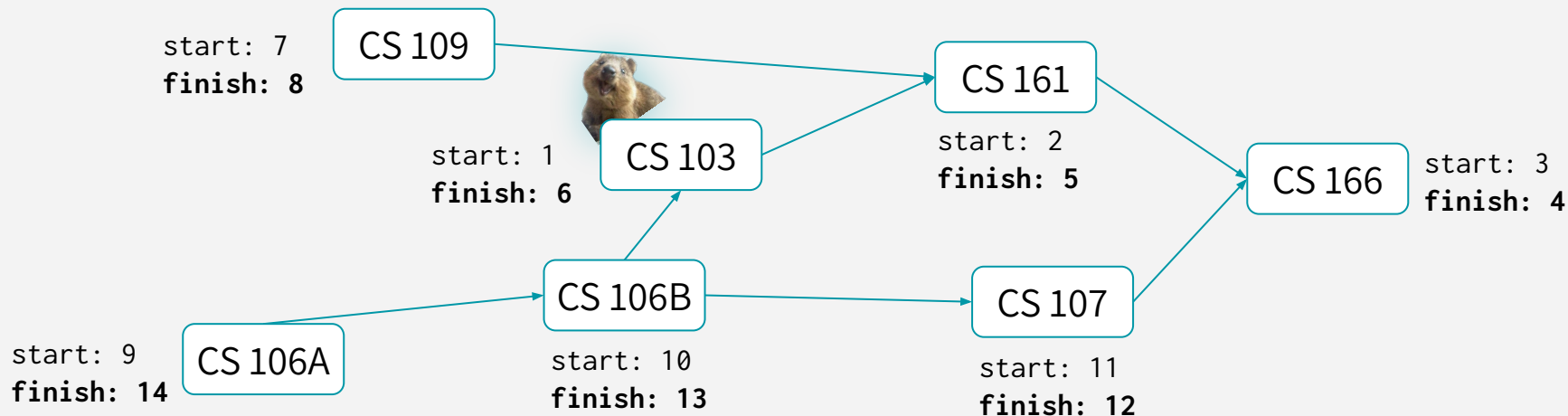
**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

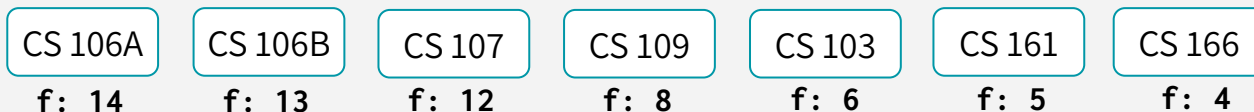
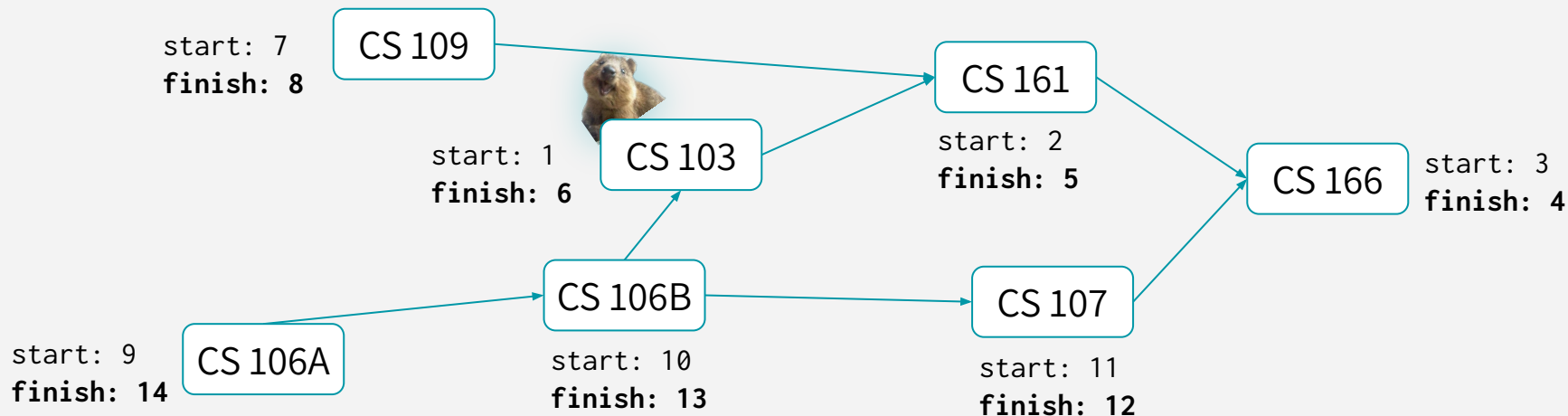
**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

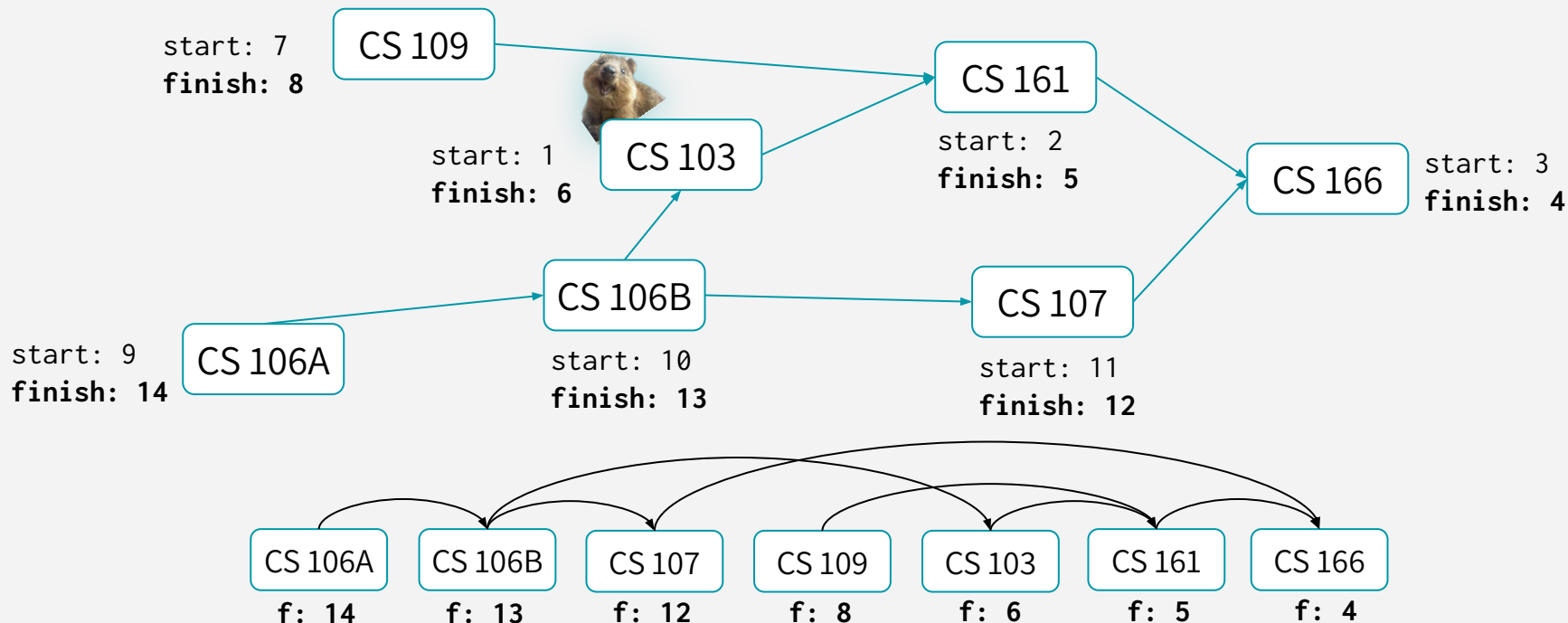
**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.

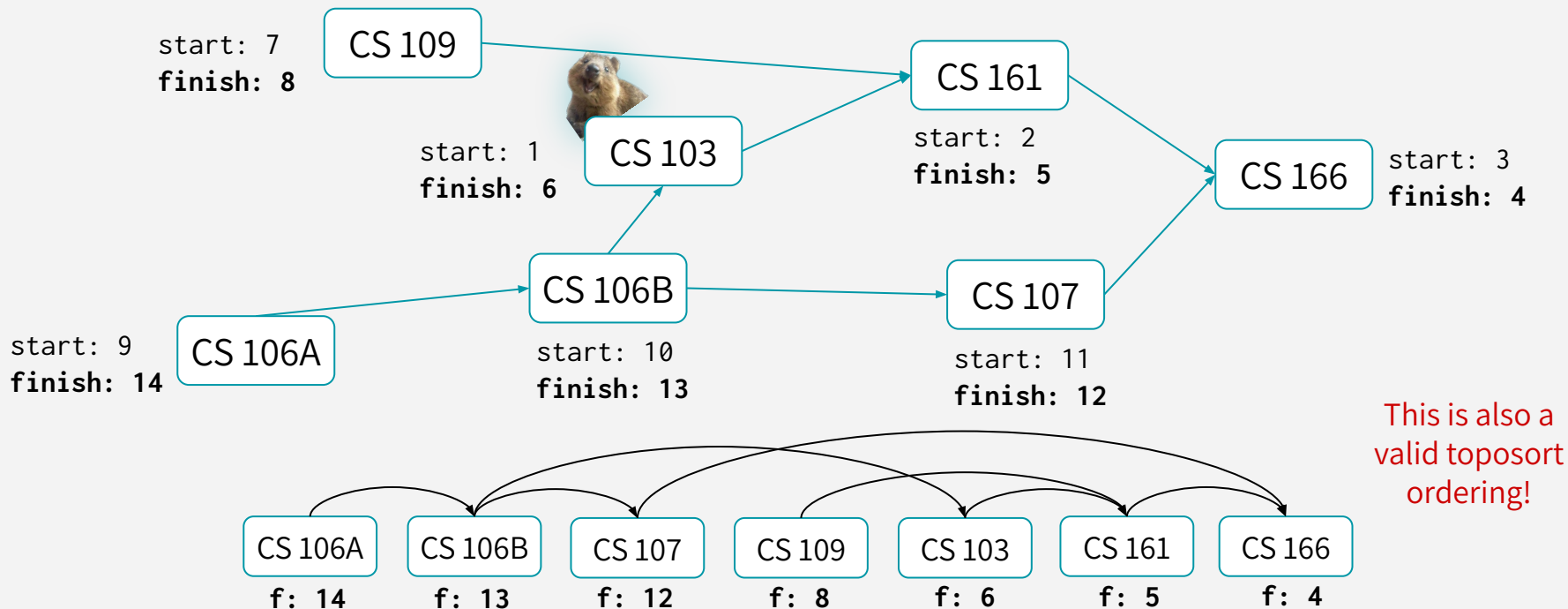




# DFS WILL GET US A TOPOSORT

*Different  
order of  
running*

**TOPOSORT:** Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



# DFS & TOPOSORT RECAP

**DFS can help you solve the Topological Sorting Problem.**

That's just the fancy name for the problem of finding an ordering of the vertices which respect all the dependencies.



سوال؟