ساختمان داده و الگوریتم ها

مبحث شانزدهم:
جستجوی عمق اول (DFS)

سجاد شیرعلی شهرضا

پاییز 1402

شنبه، 18 آذر 1402

Based on Stanford CS161 Slides from Summer 2021 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 22
- تمرین سوم
  - مهلت ارسال: 8 صبح دوشنبه 20 آذر

جستجوی عمق اول (DFS)

یک روش دیگر برای پیمایش گراف

BFS vs. DFS

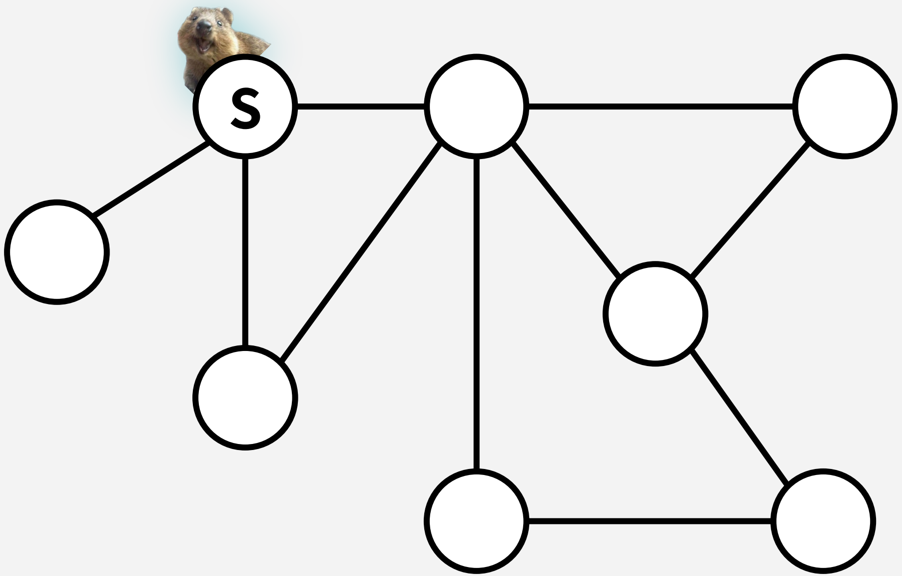
Literally just BREADTH vs DEPTH:

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:

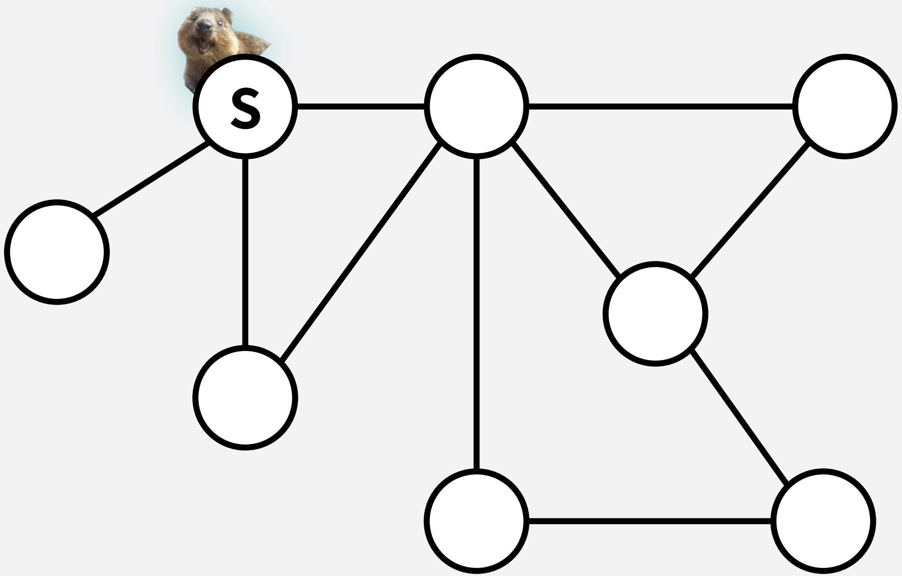
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

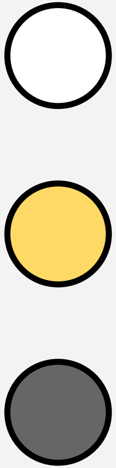


DEPTH-FIRST SEARCH

An analogy:

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





unvisited

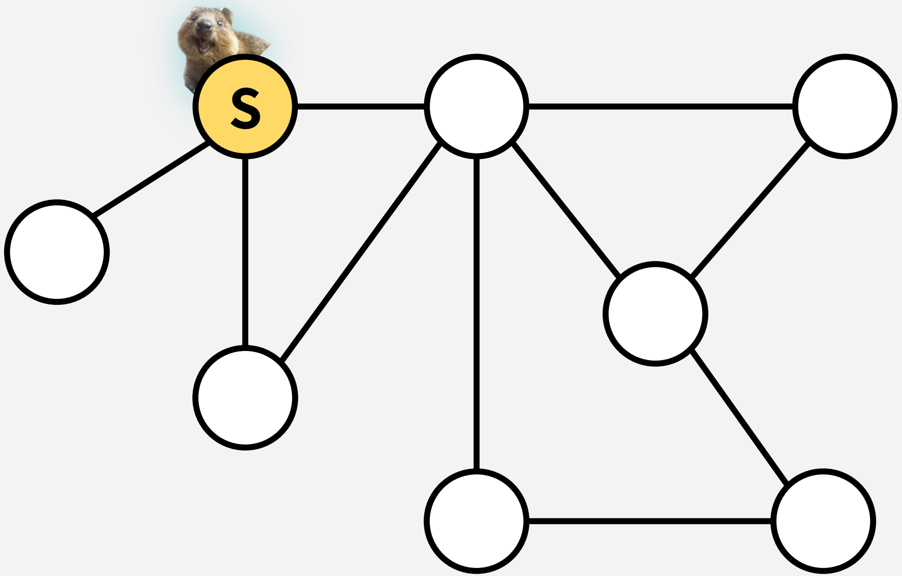
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

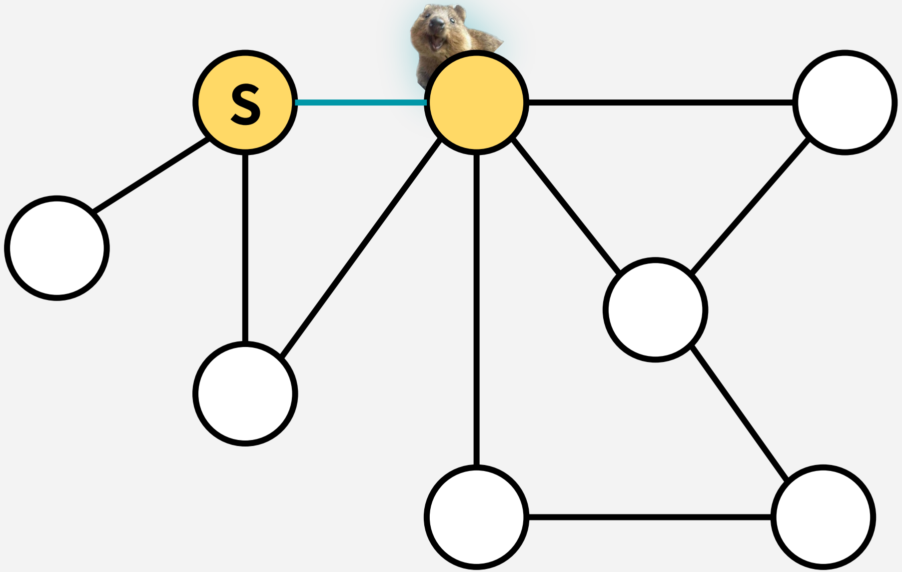
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

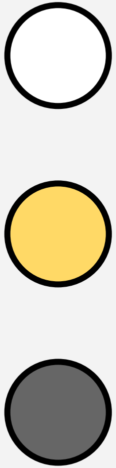


DEPTH-FIRST SEARCH

An analogy:

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





unvisited

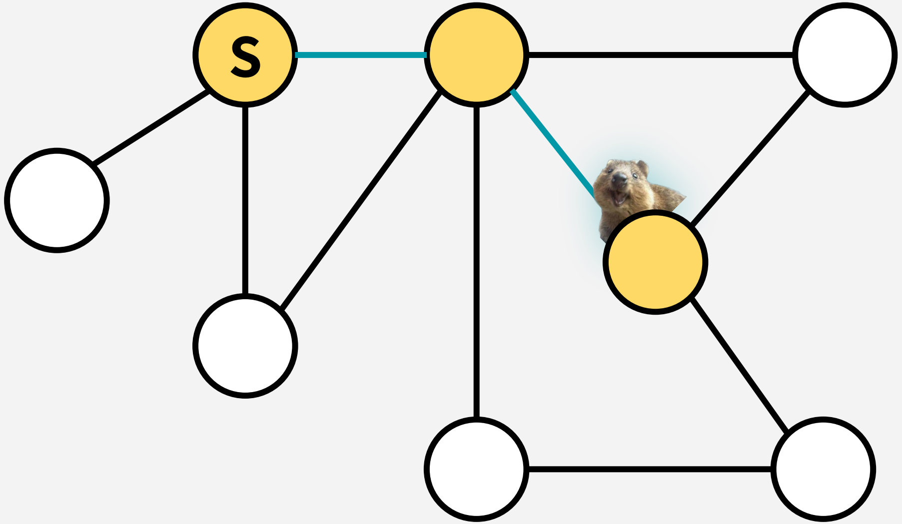
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

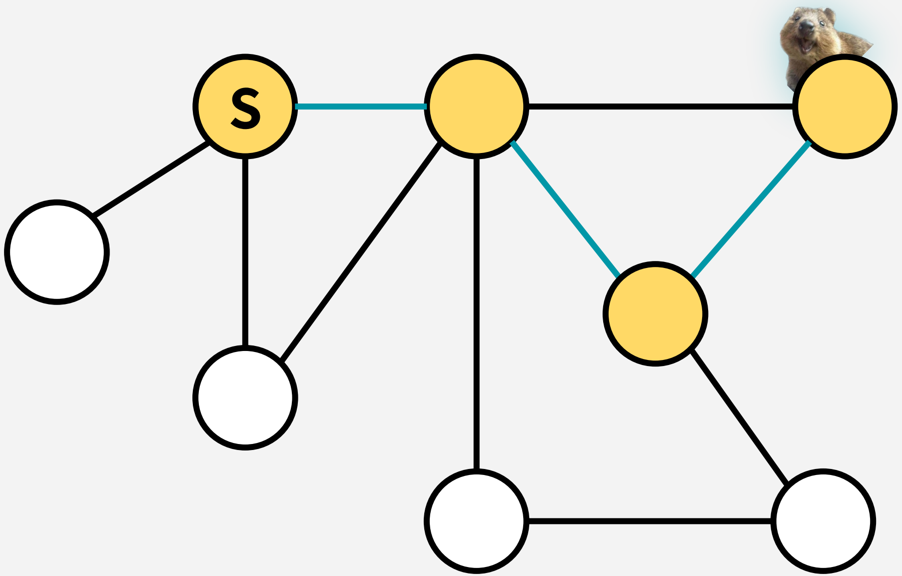
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

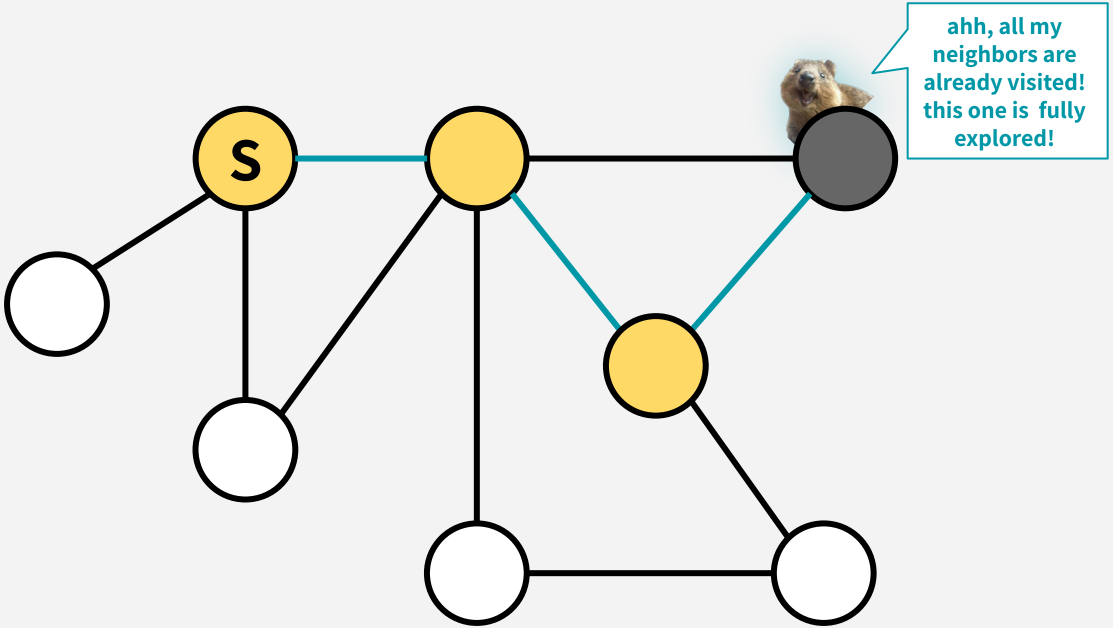
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

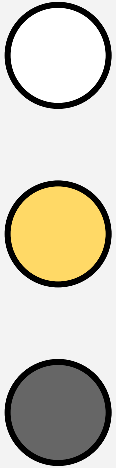


DEPTH-FIRST SEARCH

An analogy:

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





unvisited

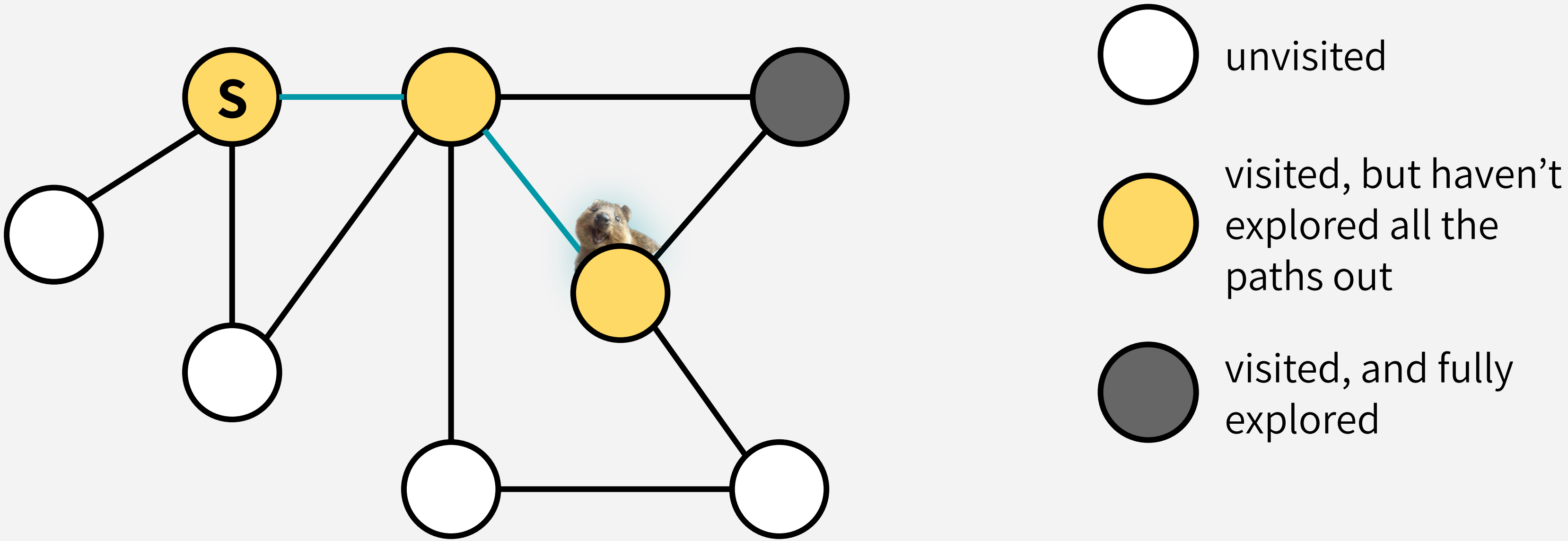
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

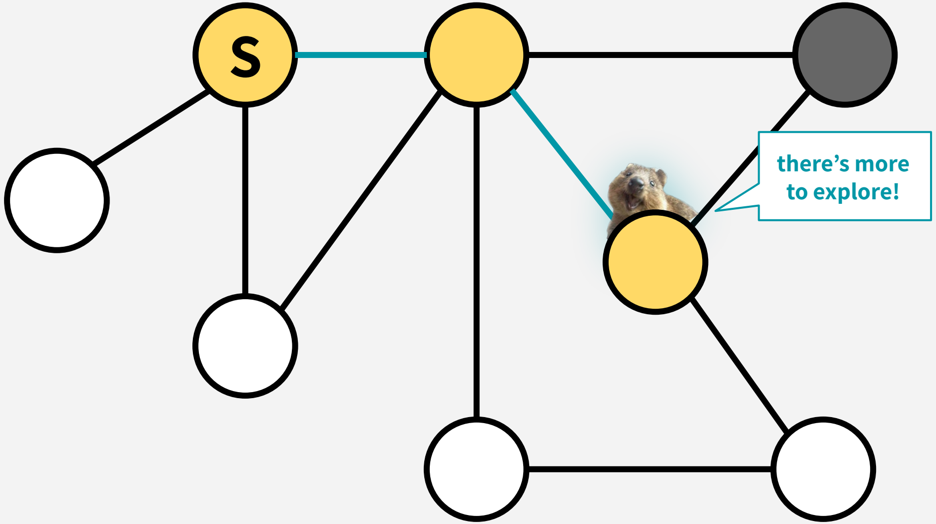
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

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



unvisited

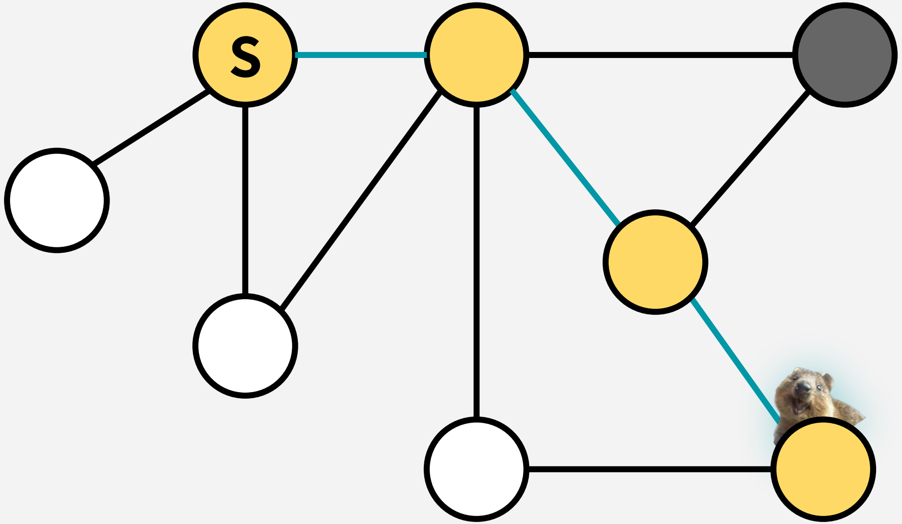
visited, but haven't explored all the paths out

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)

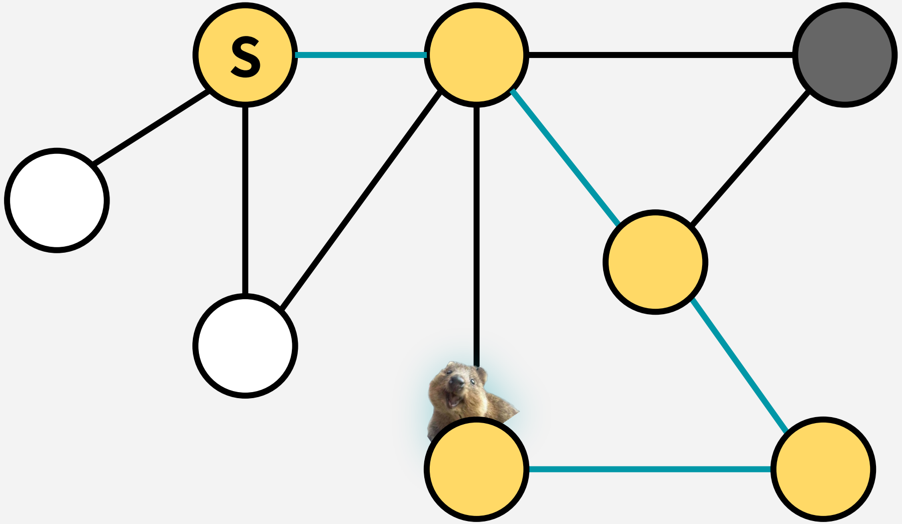


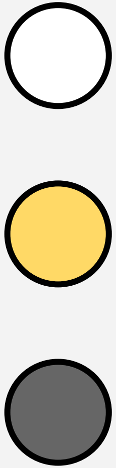
- unvisited
- visited, but haven't explored all the paths out
- visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

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

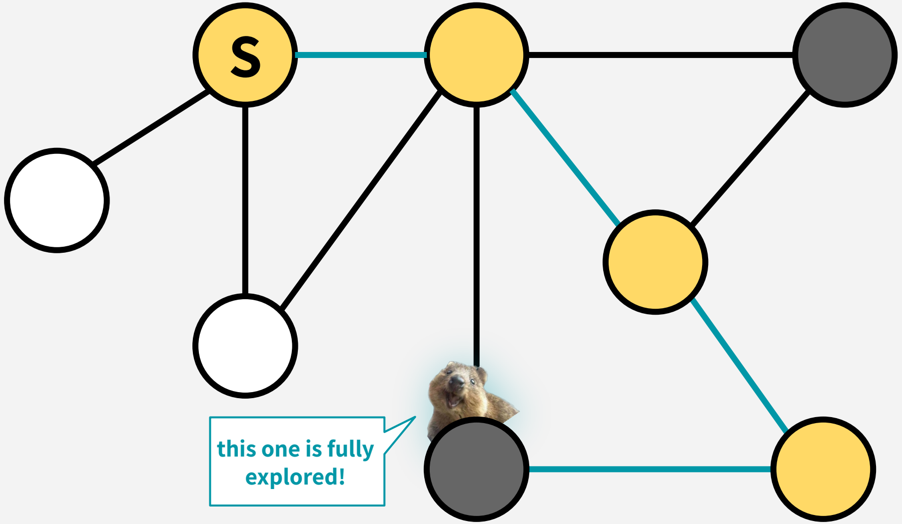




DEPTH-FIRST SEARCH

An analogy:

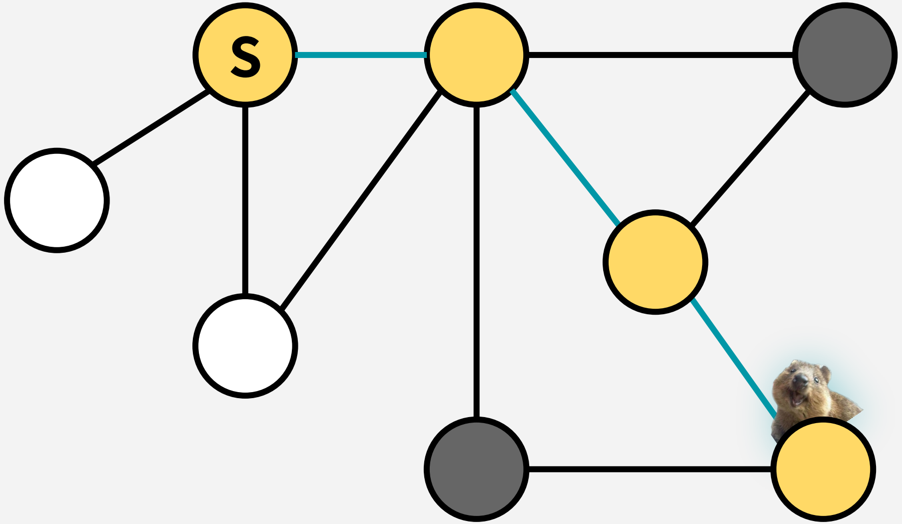
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

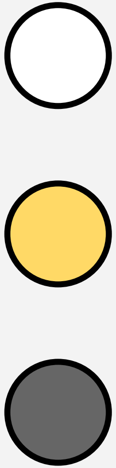


DEPTH-FIRST SEARCH

An analogy:

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





unvisited

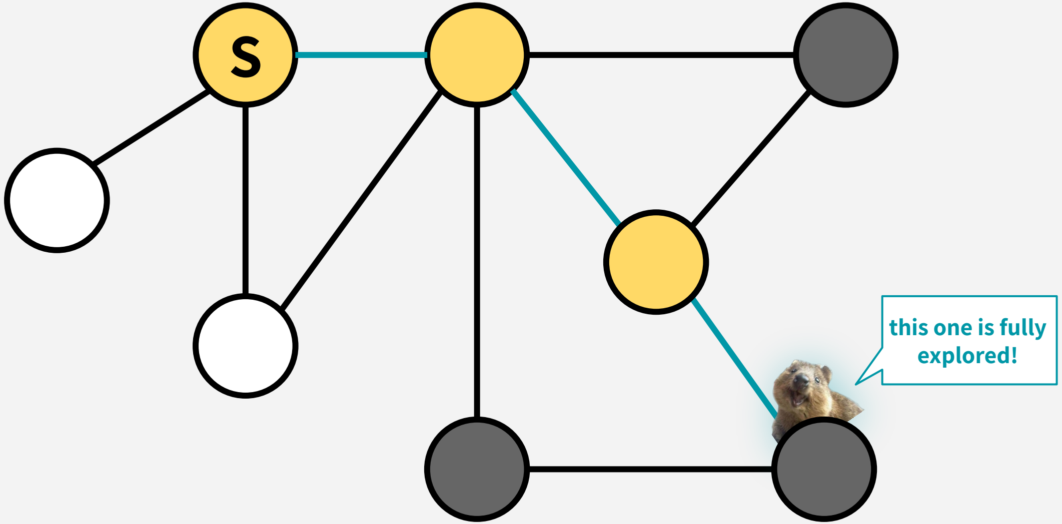
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

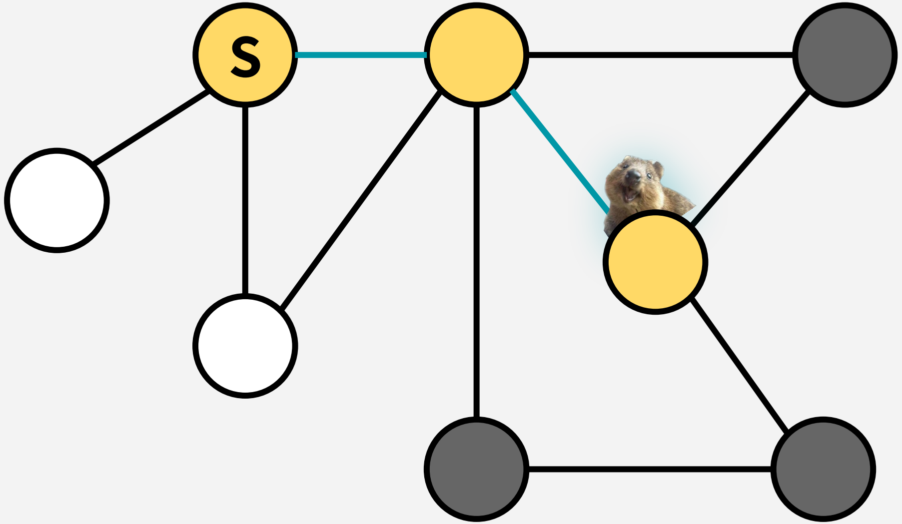
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

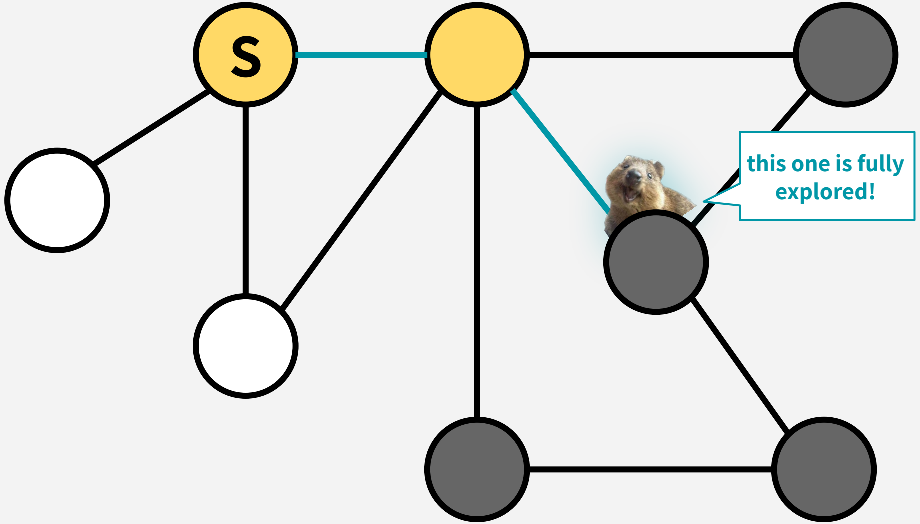
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

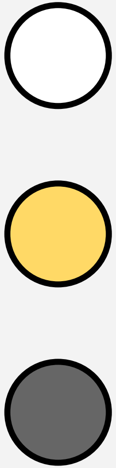


DEPTH-FIRST SEARCH

An analogy:

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





unvisited

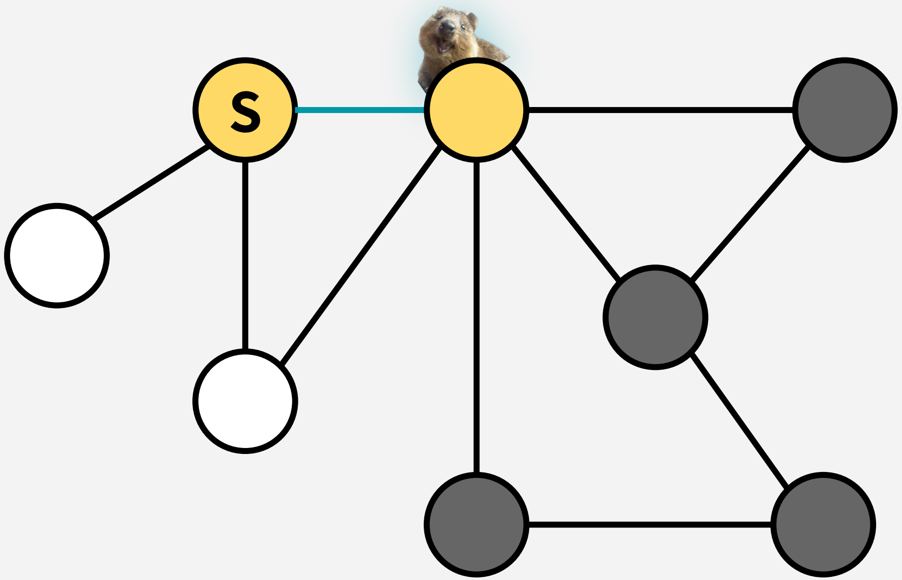
visited, but haven’t explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

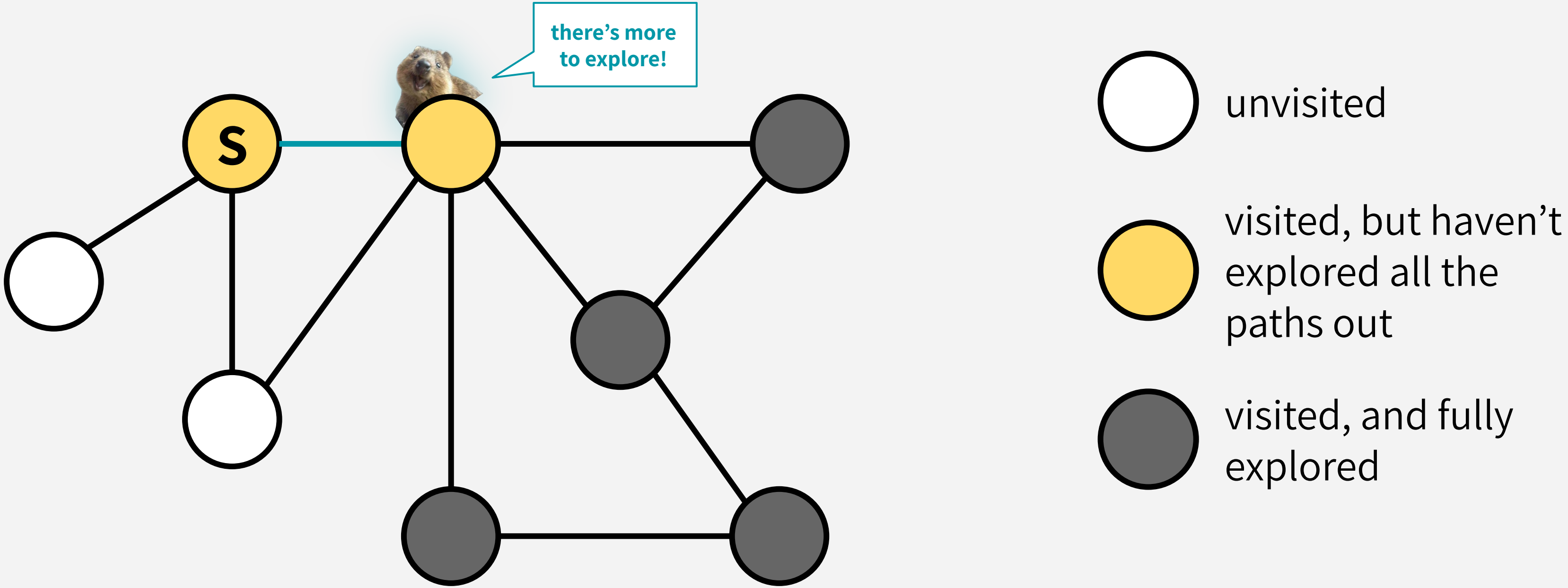
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

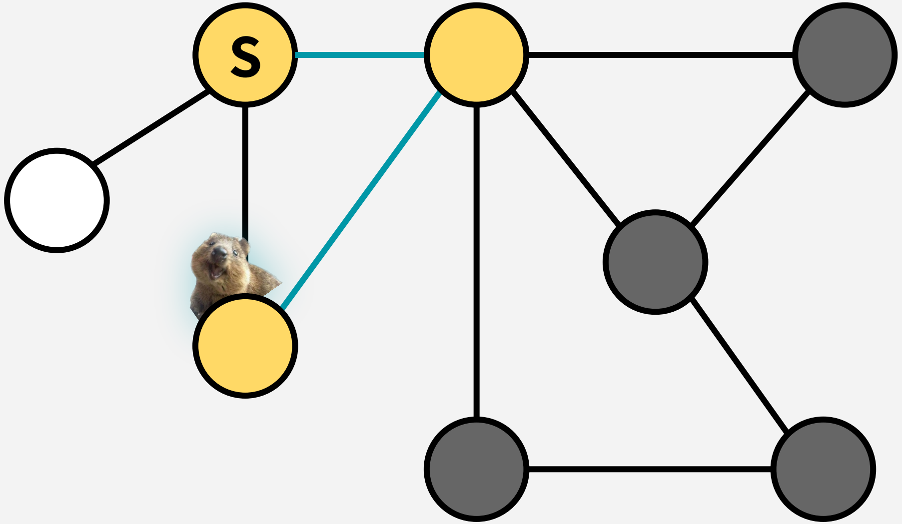
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

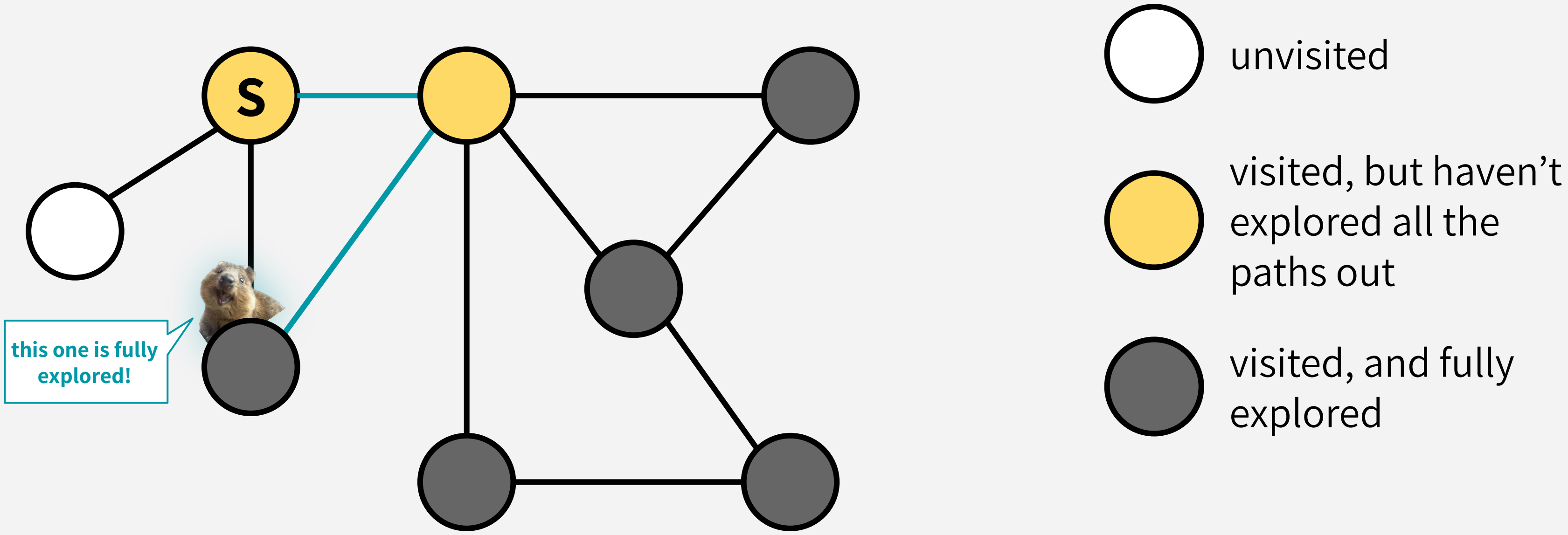
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

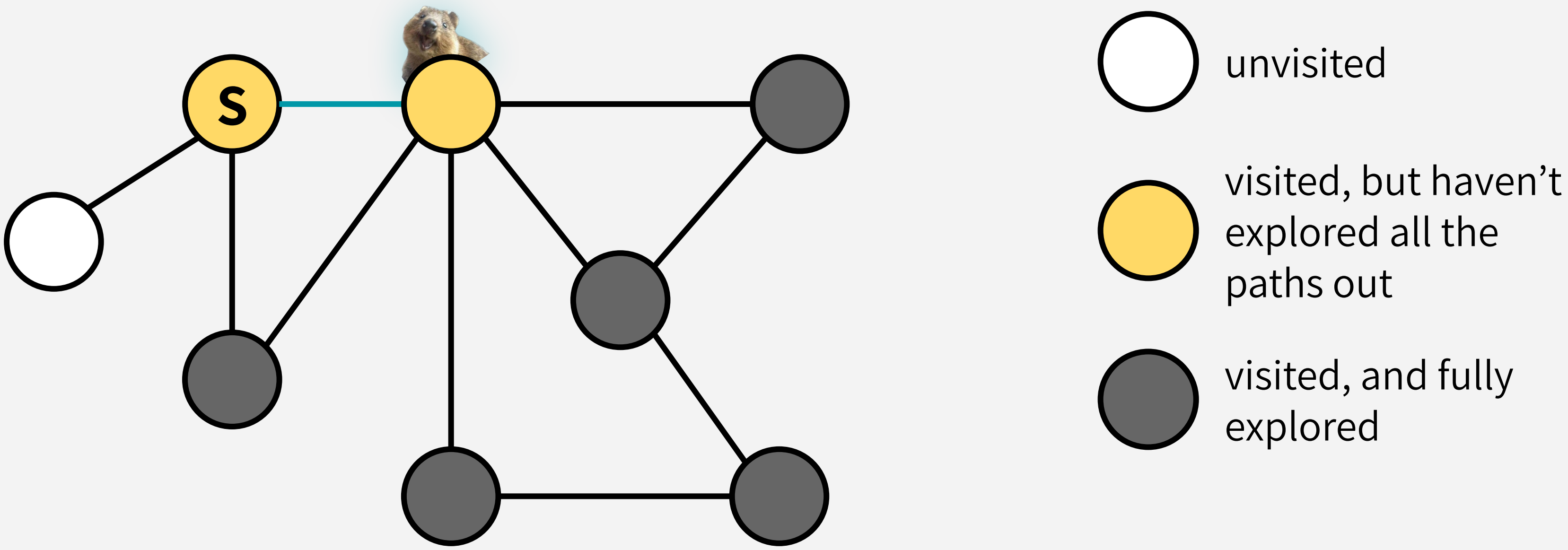
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

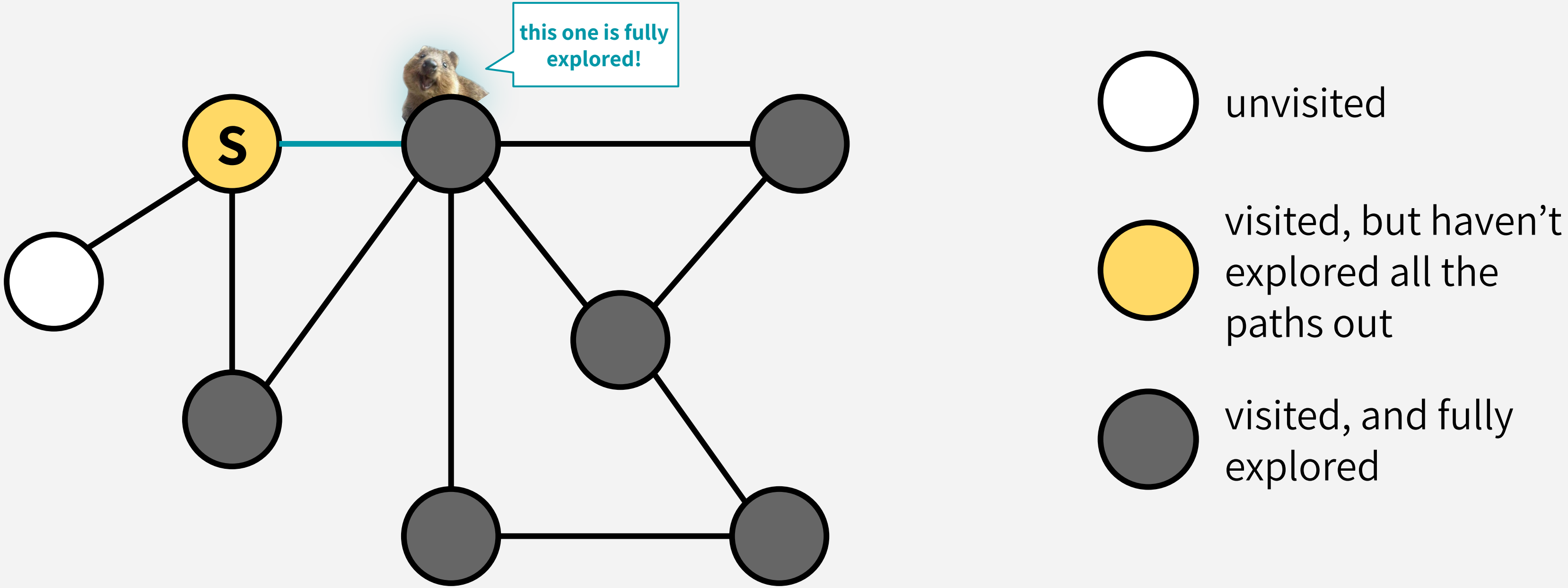
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

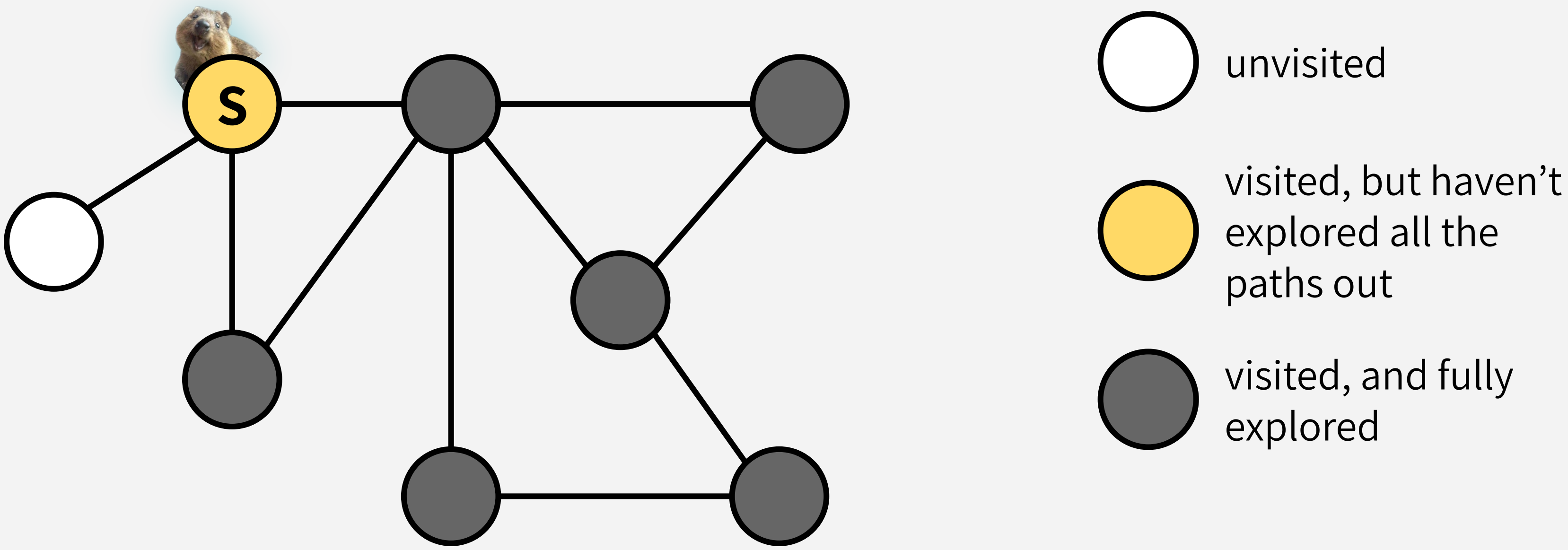
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

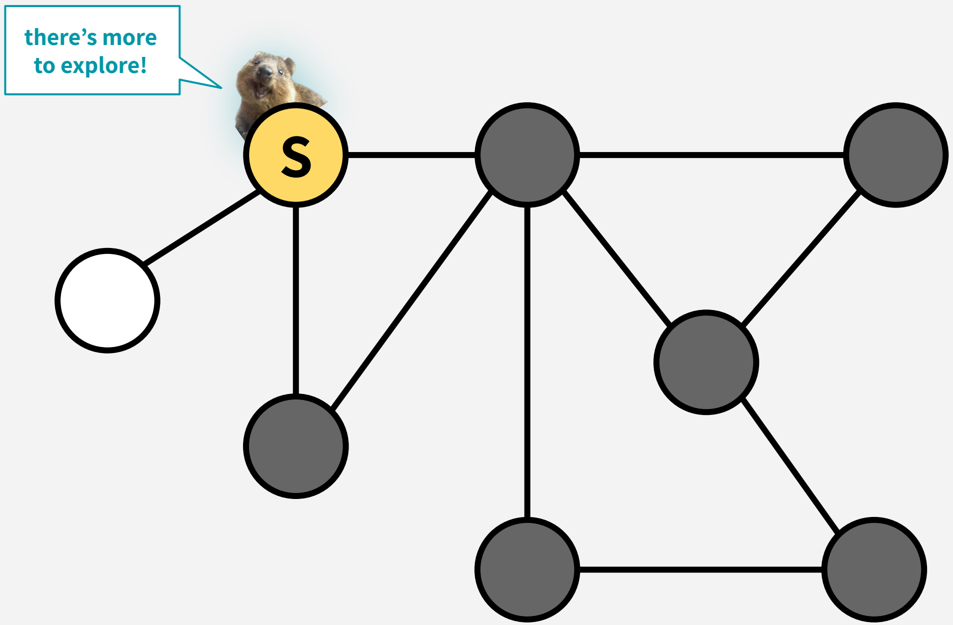
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

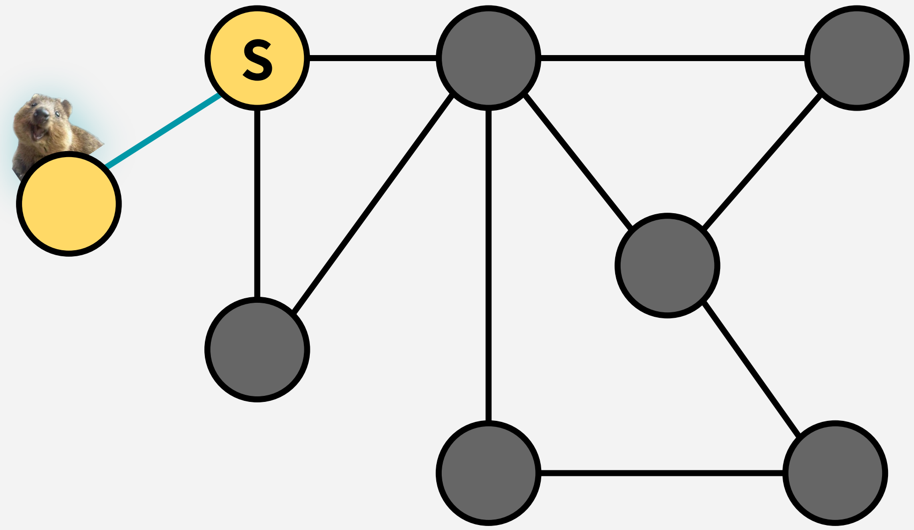
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

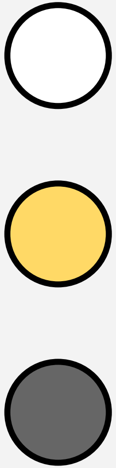


DEPTH-FIRST SEARCH

An analogy:

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

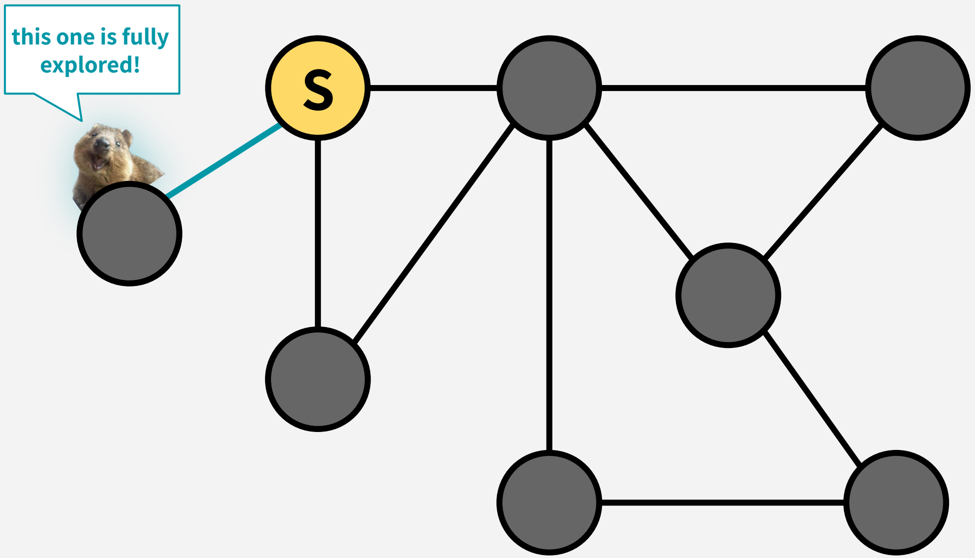




DEPTH-FIRST SEARCH

An analogy:

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



unvisited

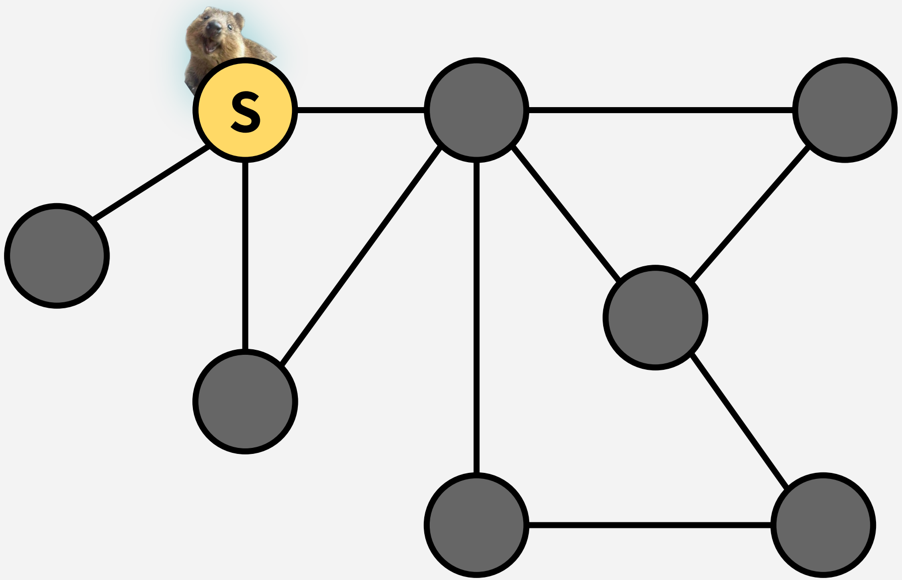
visited, but haven't explored all the paths out

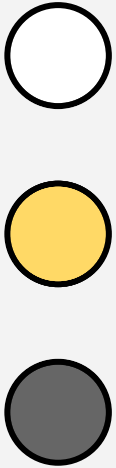
visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

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





unvisited

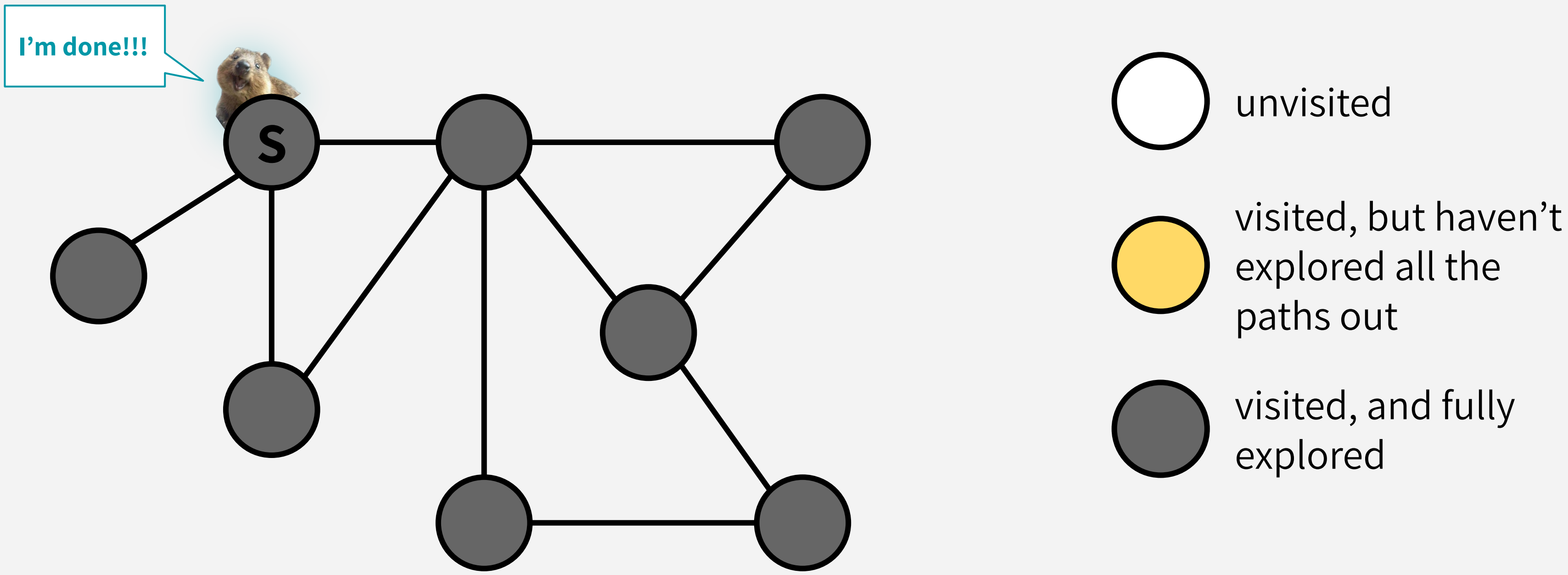
visited, but haven't explored all the paths out

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)



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!

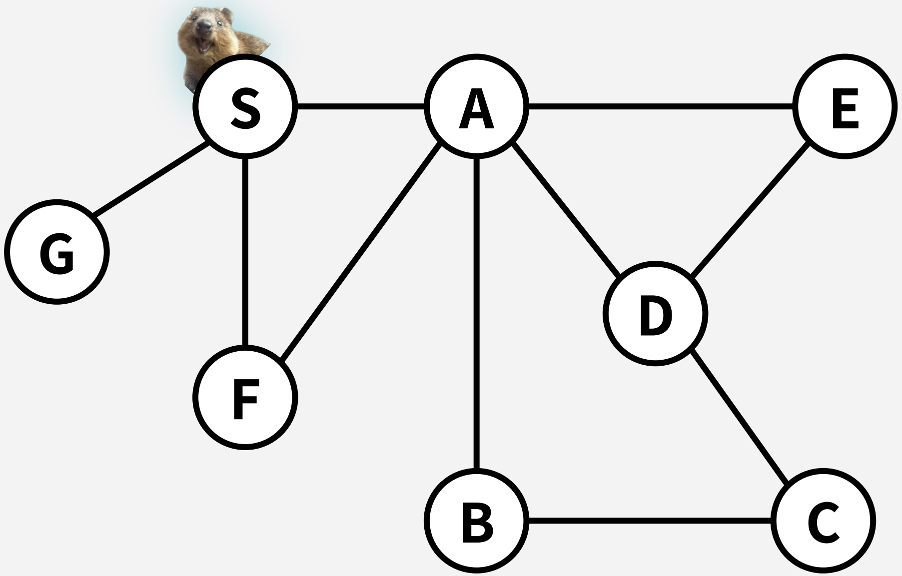
ut haven't
all the

nd fully

DEPTH-FIRST SEARCH

An analogy:

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



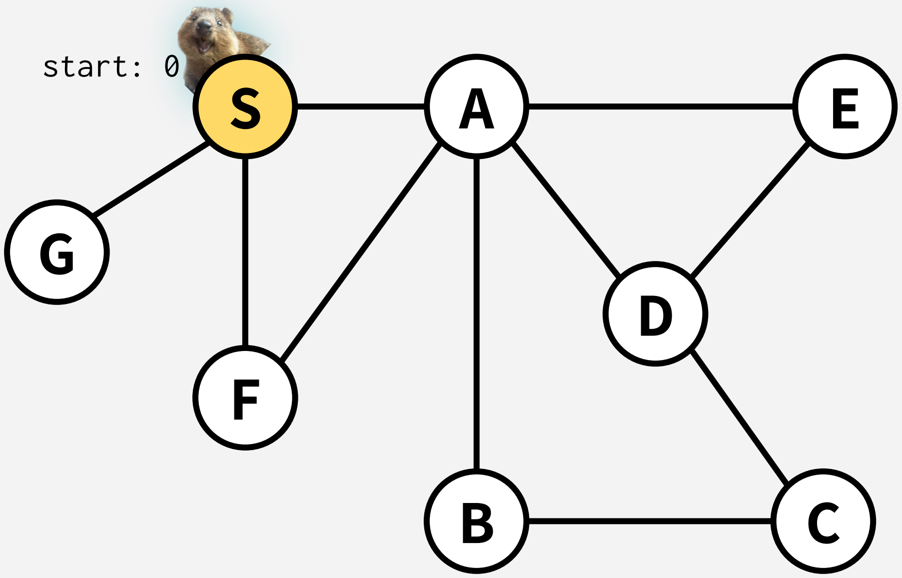

```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DFS (S , 0) **currTime = 0**

DEPTH-FIRST SEARCH

An analogy:

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



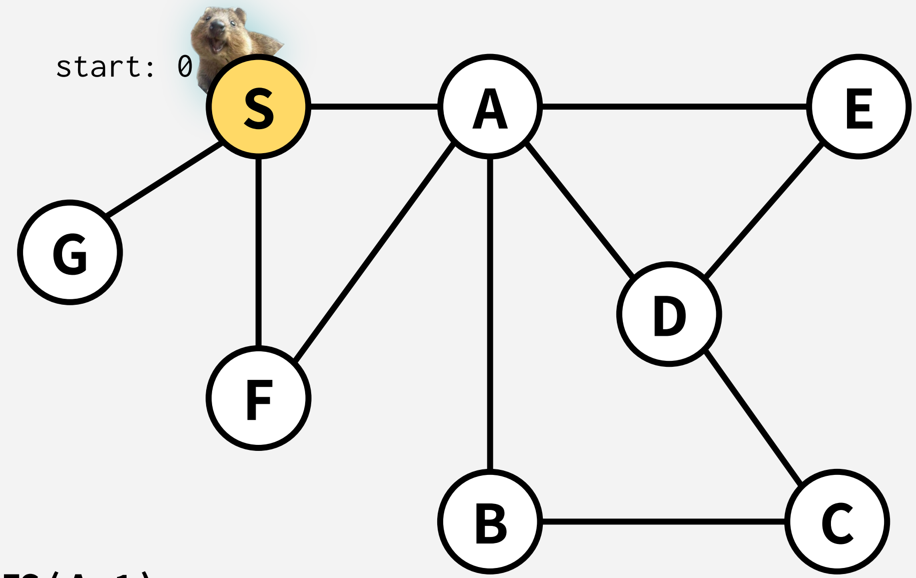

```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DFS(S,0) **currTime = 1**

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
```

DFS(A,1)

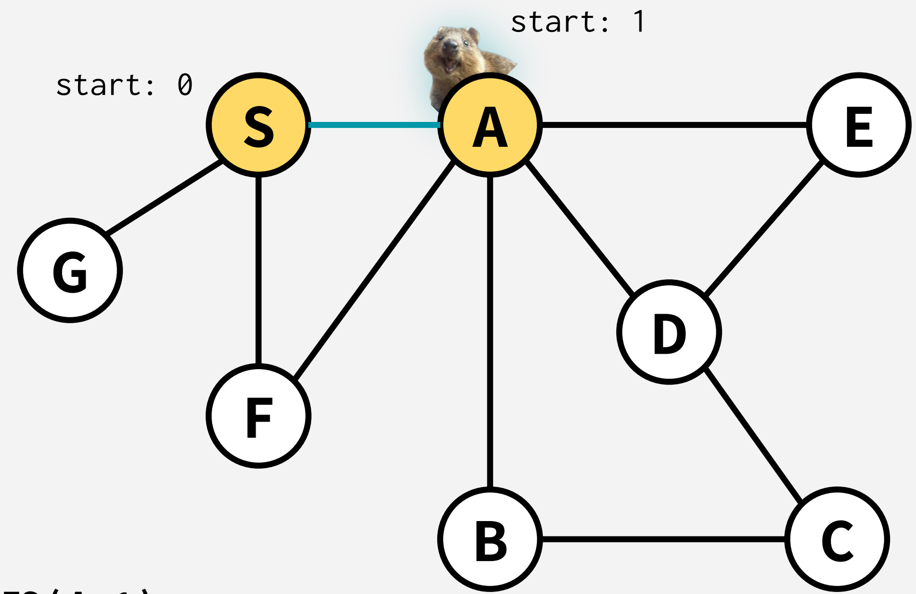
DFS(S,0)

currTime = 1

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
```

DFS(A,1)

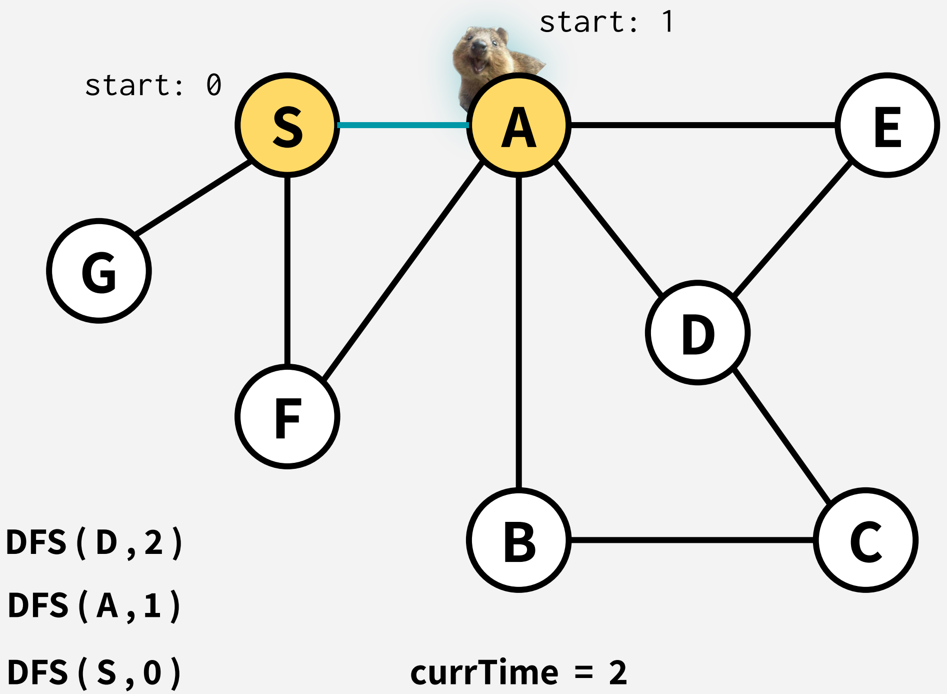
DFS(S,0)

currTime = 2

DEPTH-FIRST SEARCH

An analogy:

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

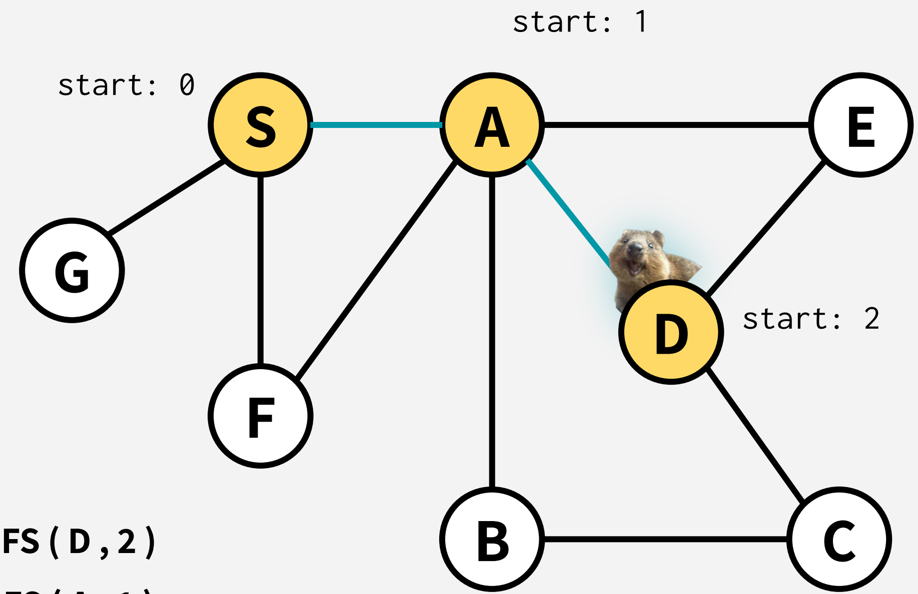



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

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
```

DFS(D,2)

DFS(A,1)

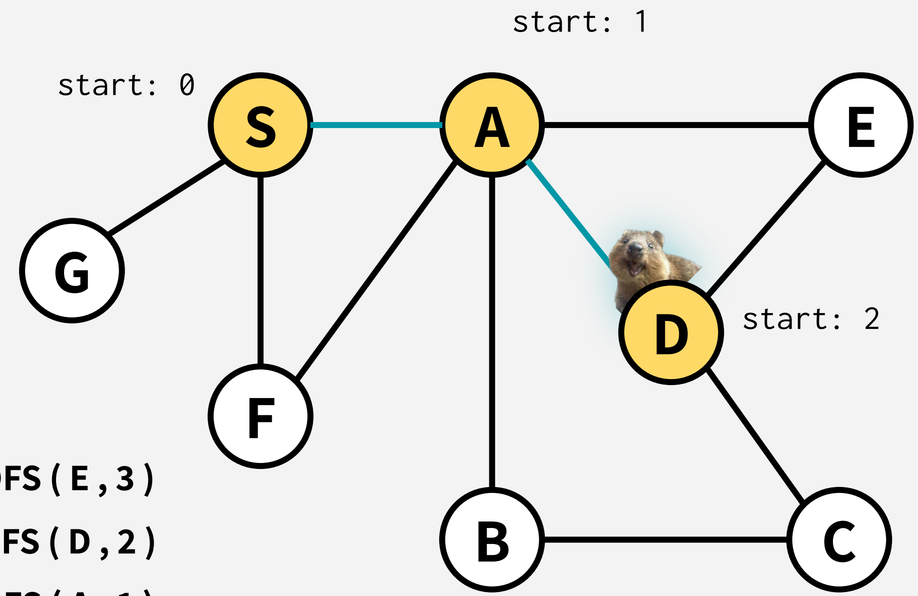
DFS(S,0)

currTime = 3

DEPTH-FIRST SEARCH

An analogy:

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



DFS(E,3)

DFS(D,2)

DFS(A,1)

DFS(S,0)

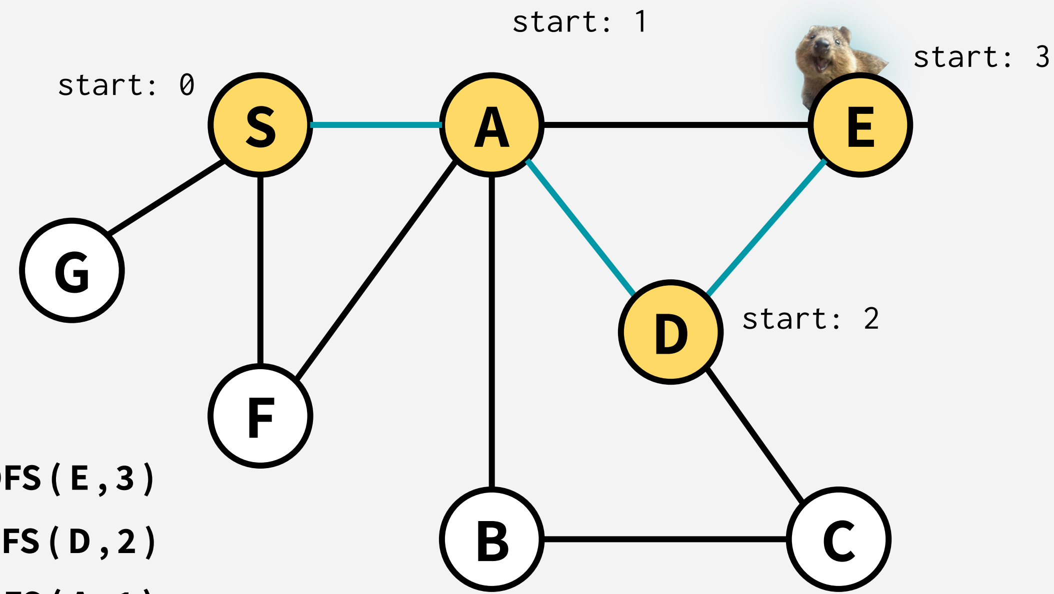
currTime = 3

```
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(E,3)

DFS(D,2)

DFS(A,1)

DFS(S,0)

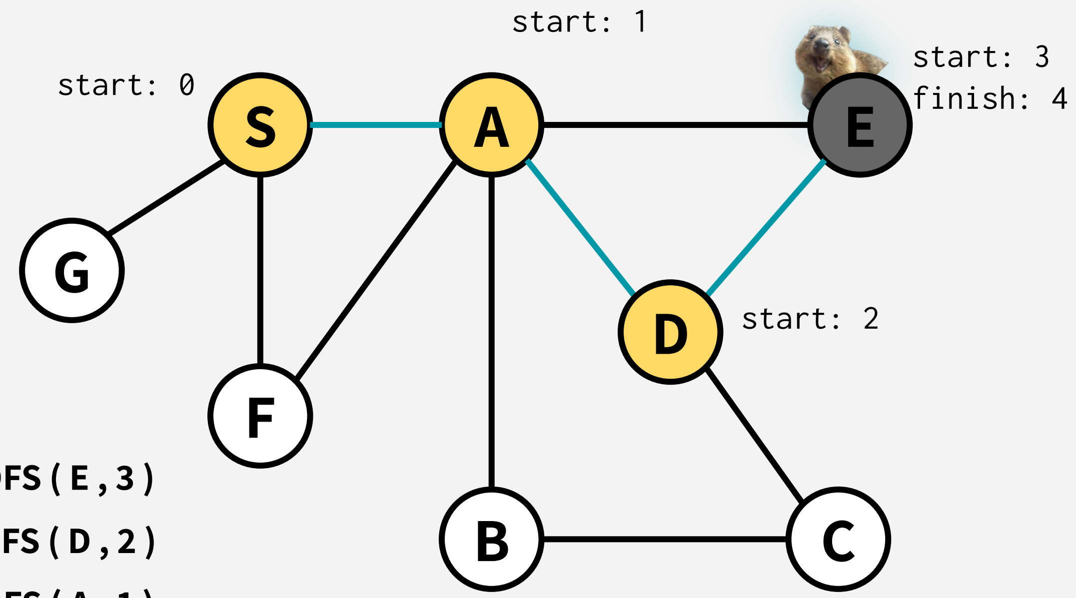
currTime = 4

```
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
```

DFS(E,3)

DFS(D,2)

DFS(A,1)

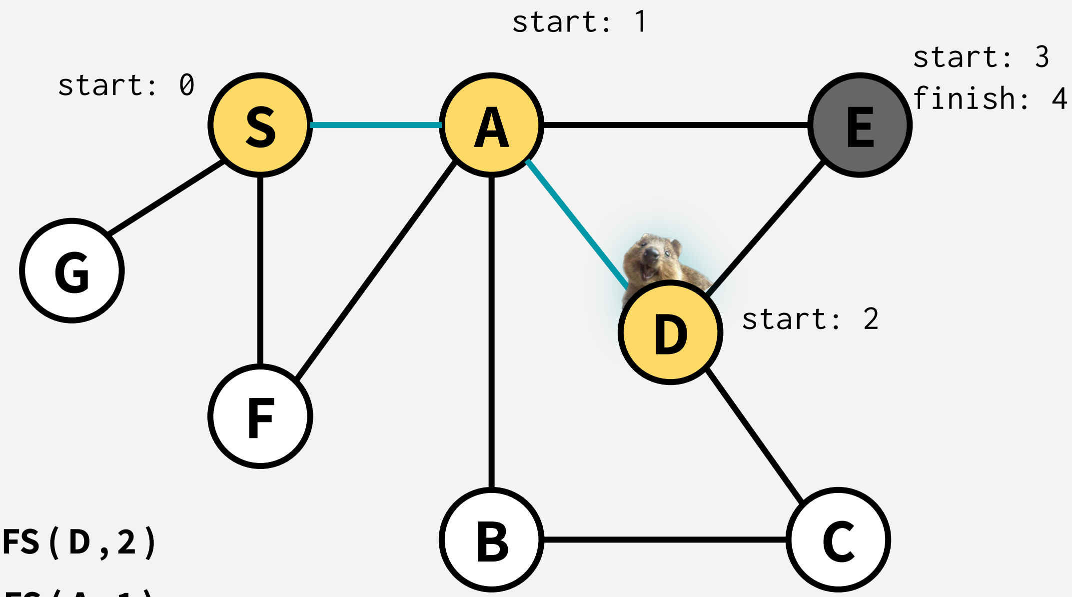
DFS(S,0)

currTime = 4

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
```

DFS(D,2)

DFS(A,1)

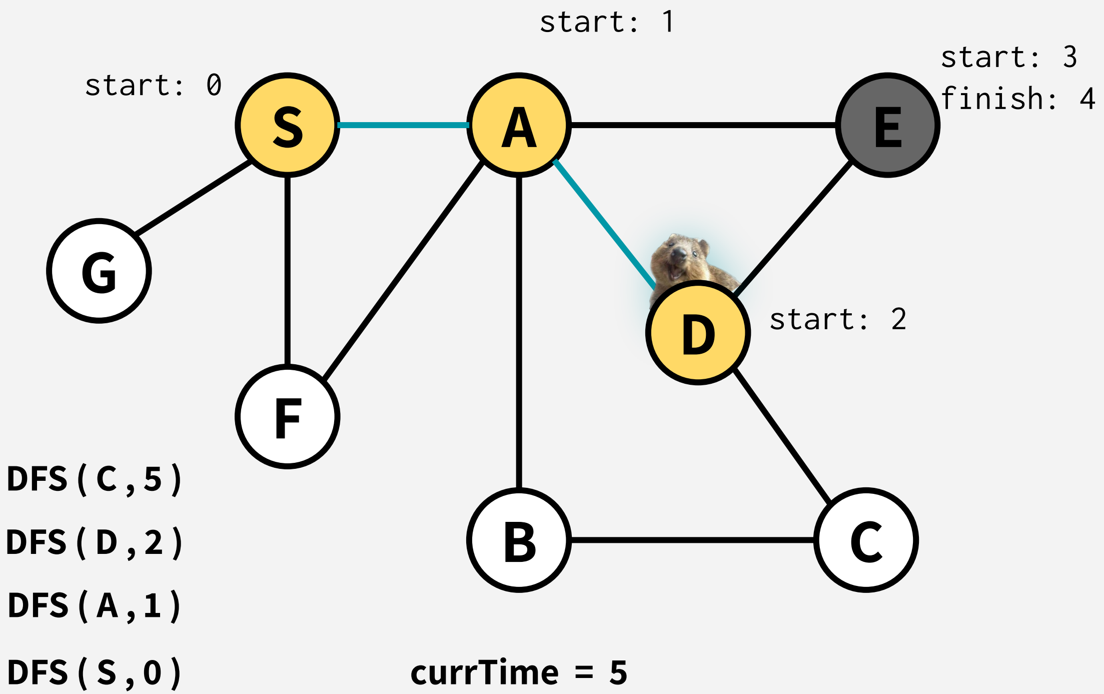
DFS(S,0)

currTime = 5

DEPTH-FIRST SEARCH

An analogy:

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

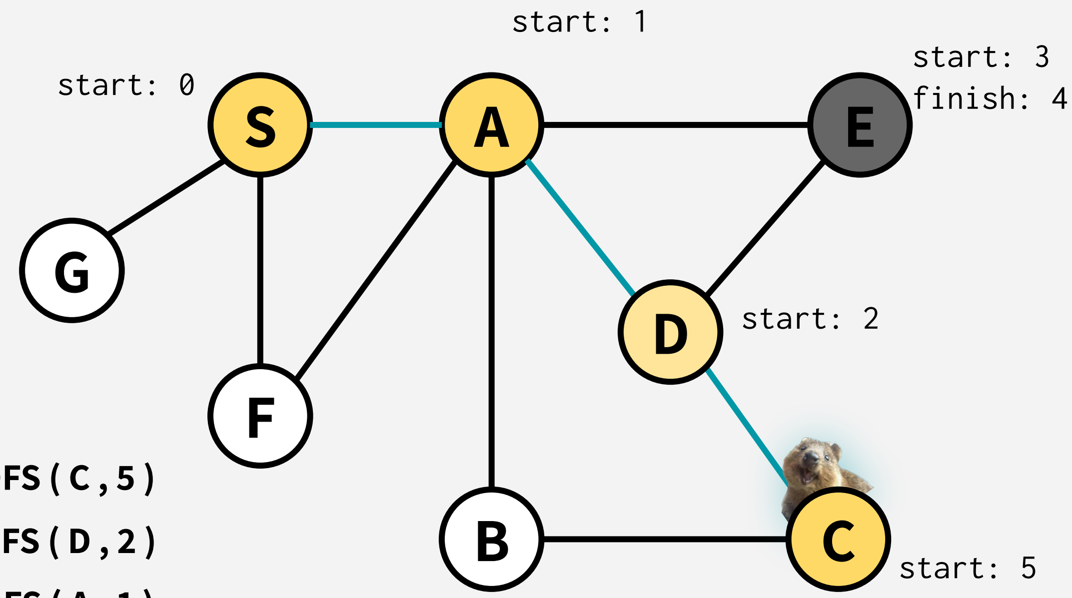



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

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
```

DFS(C,5)

DFS(D,2)

DFS(A,1)

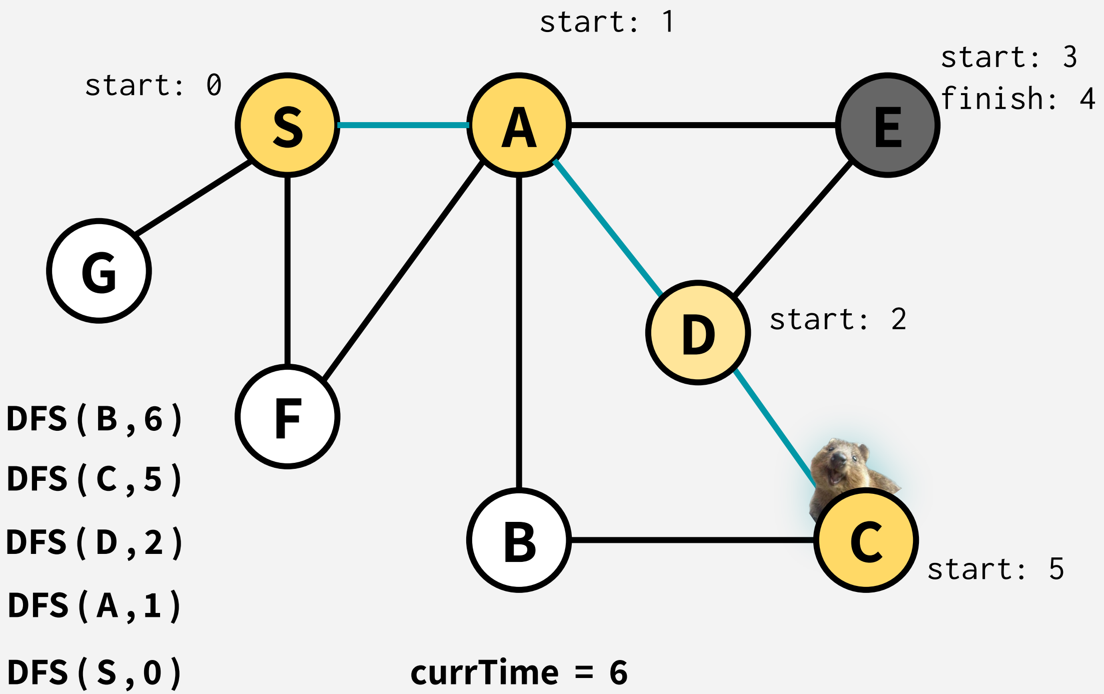
DFS(S,0)

currTime = 6

DEPTH-FIRST SEARCH

An analogy:

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

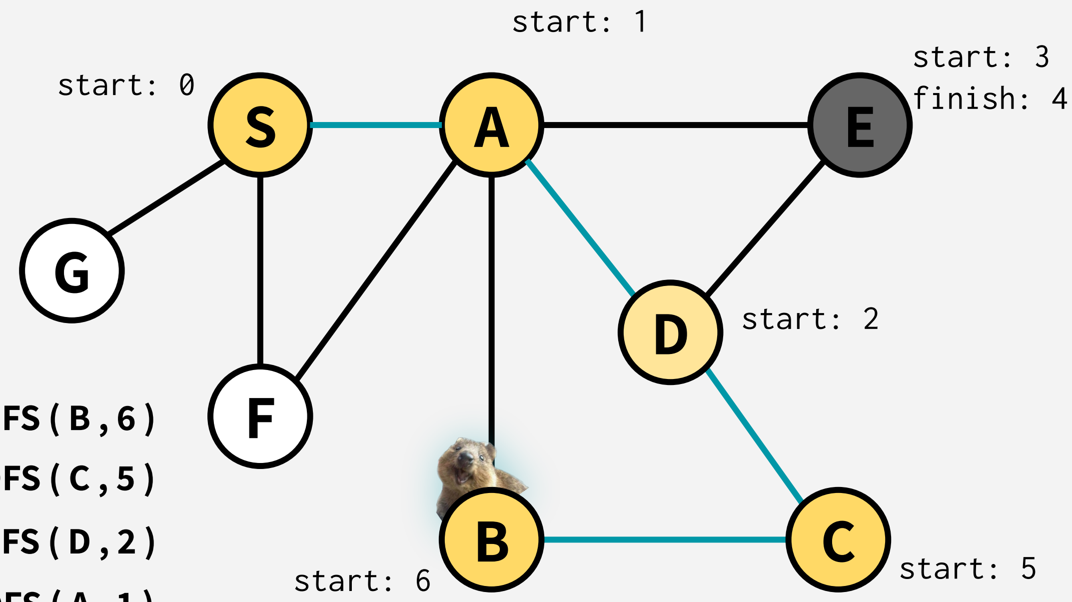



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
               DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

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



DFS(B,6)

DFS(C,5)

DFS(D,2)

DFS(A,1)

DFS(S,0)

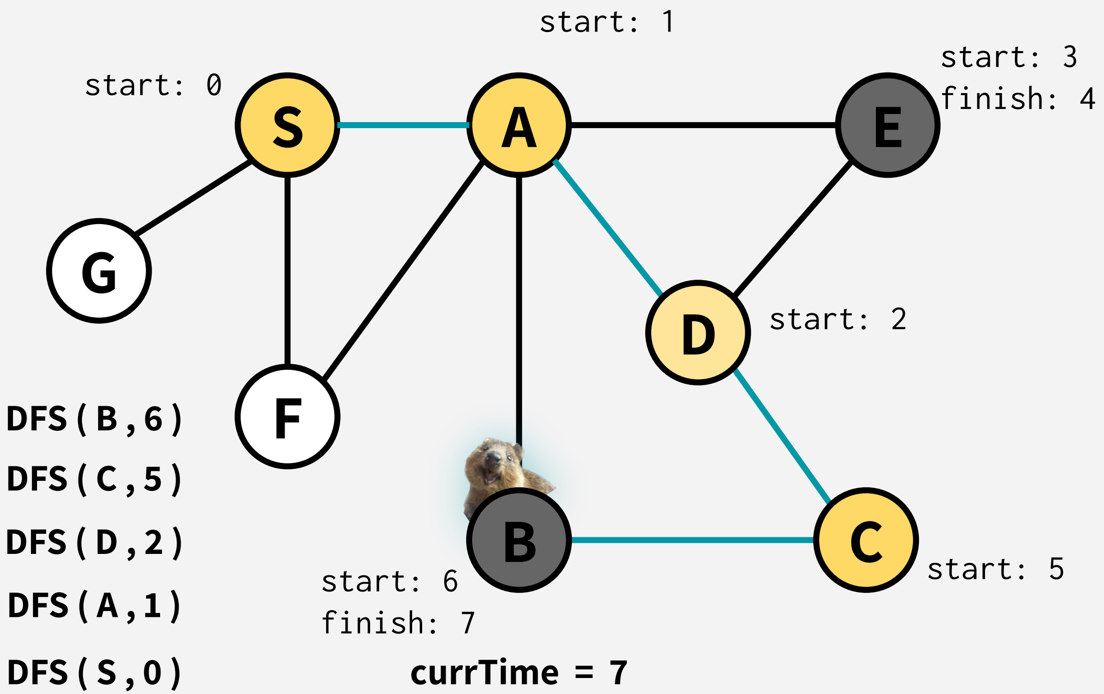
currTime = 7

```
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)

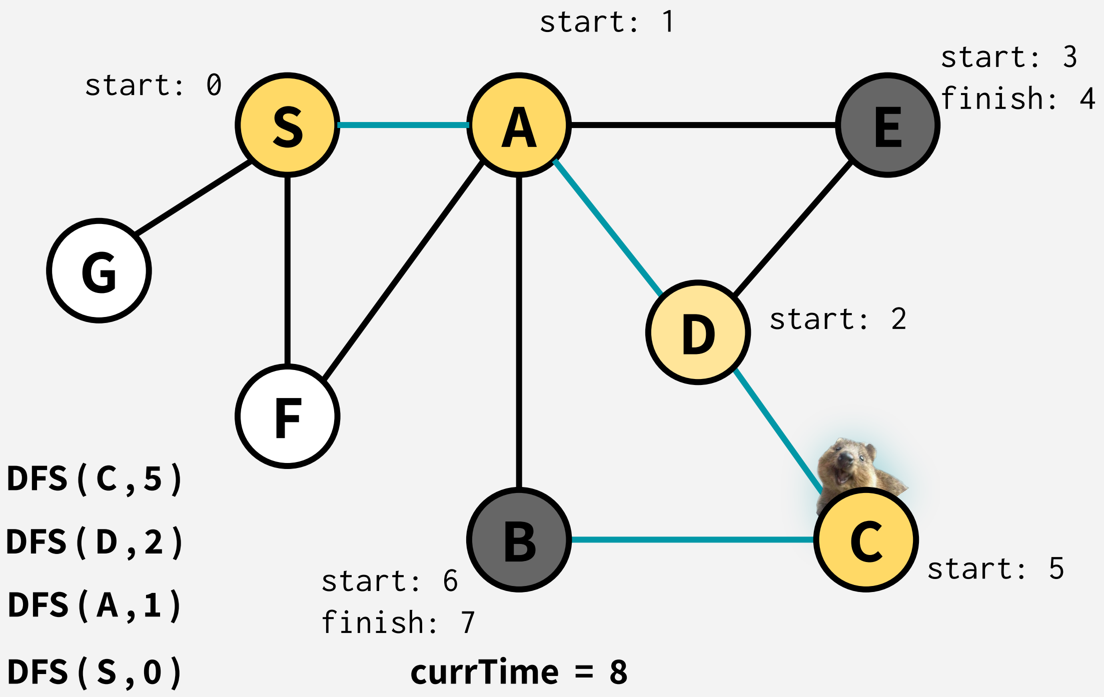



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

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

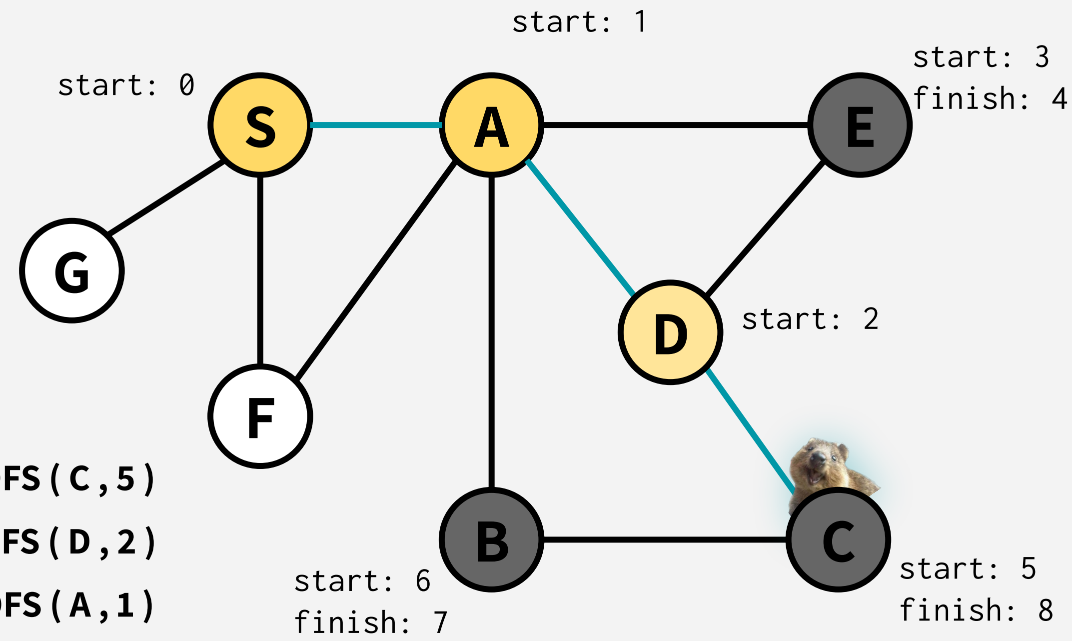



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

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



DFS(C,5)

DFS(D,2)

DFS(A,1)

DFS(S,0)

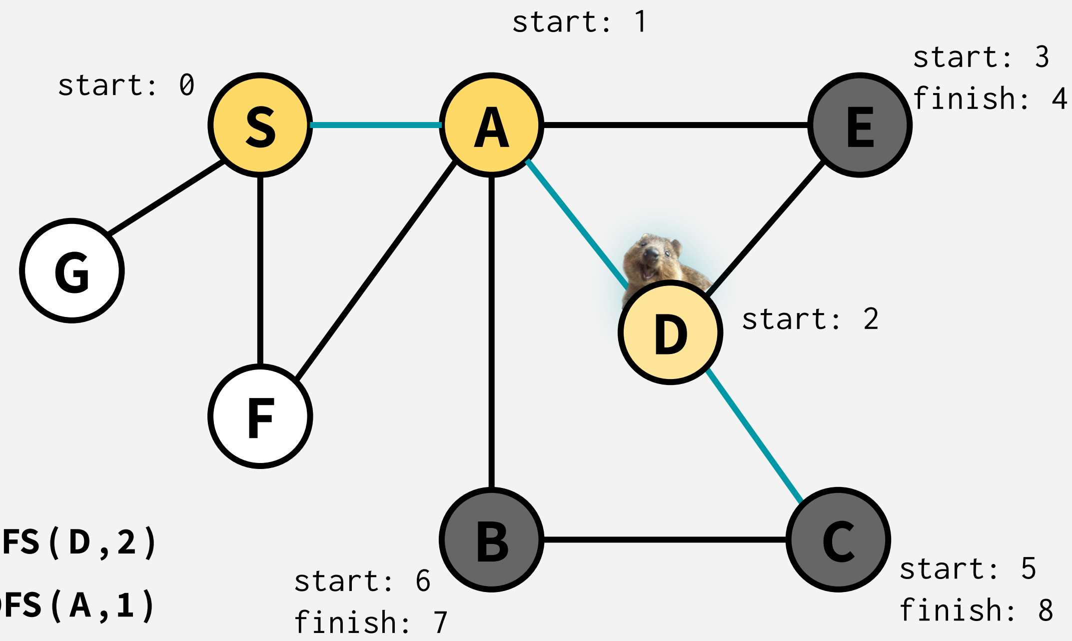
currTime = 8

```
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
```

DFS(D,2)

DFS(A,1)

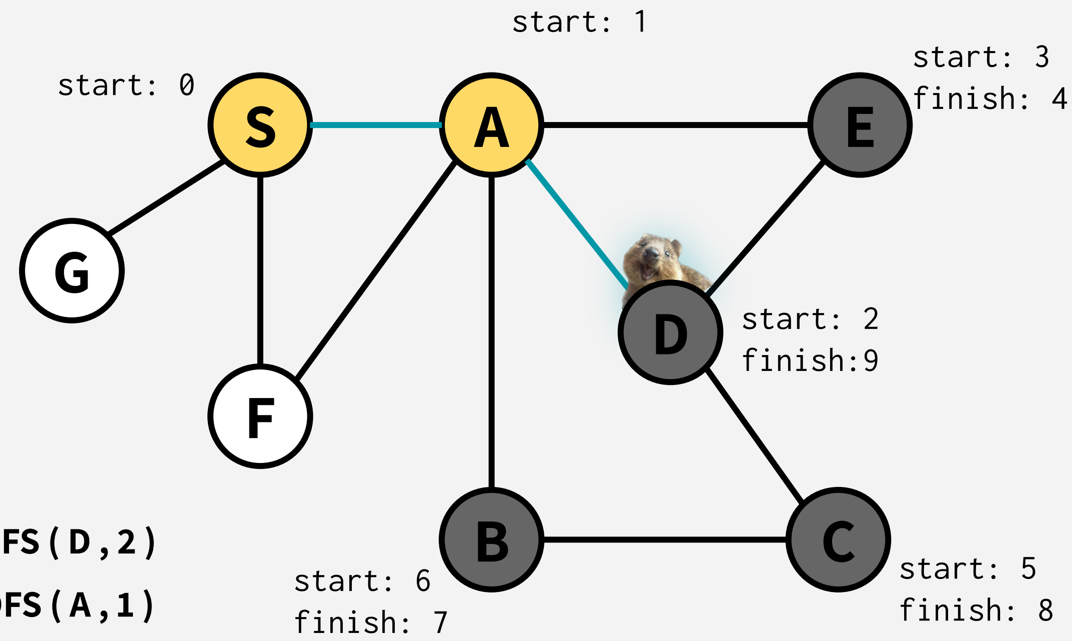
DFS(S,0)

currTime = 9

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
```

DFS(D,2)

DFS(A,1)

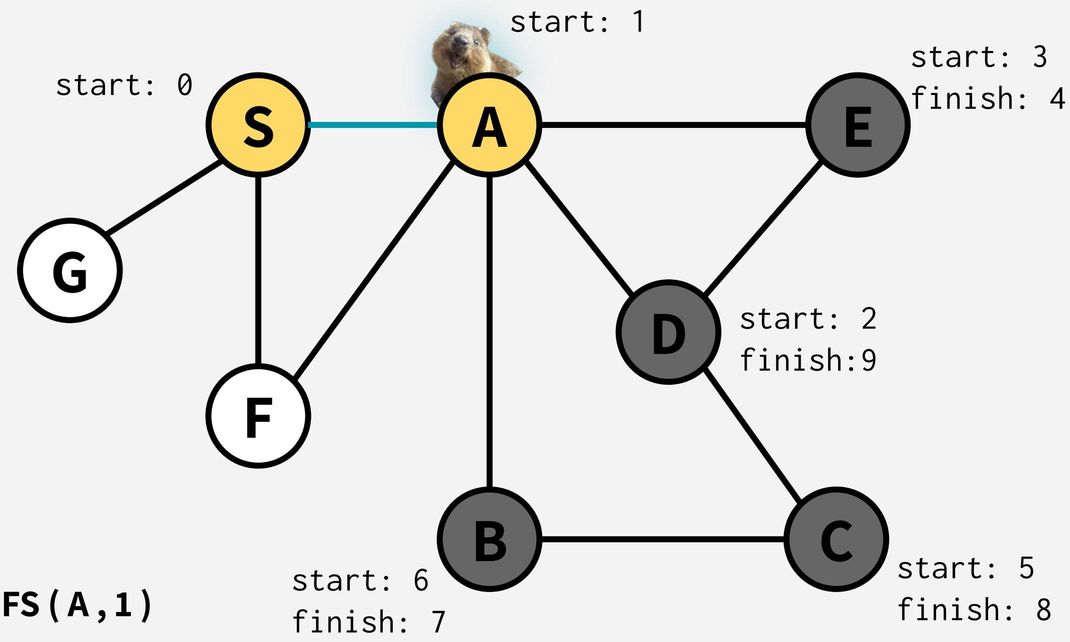
DFS(S,0)

currTime = 9

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
```

DFS(A,1)

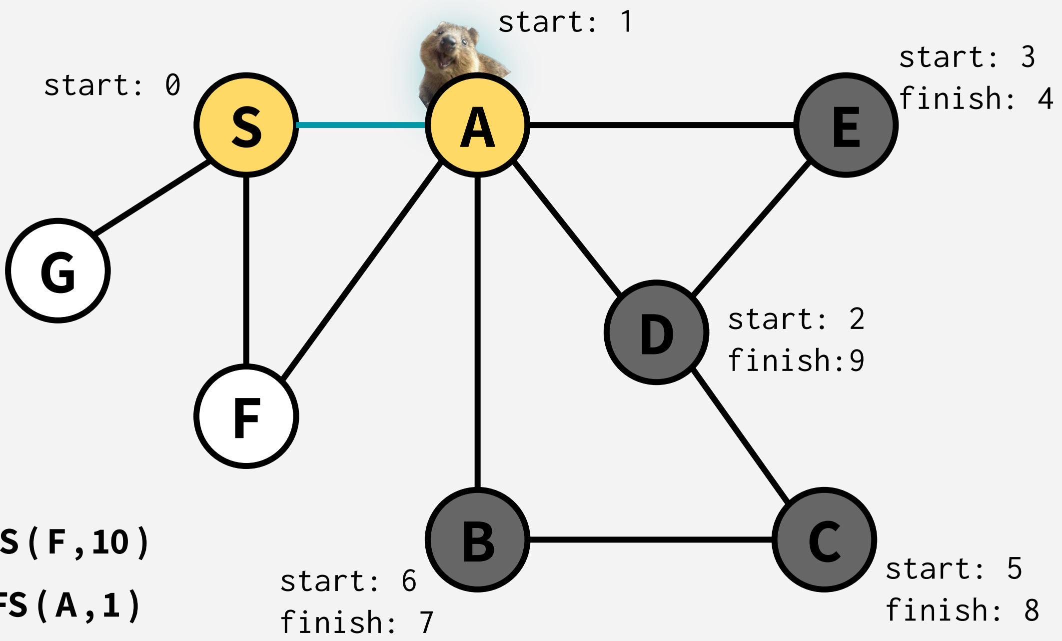
DFS(S,0)

currTime = 10

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
```

DFS(F,10)

DFS(A,1)

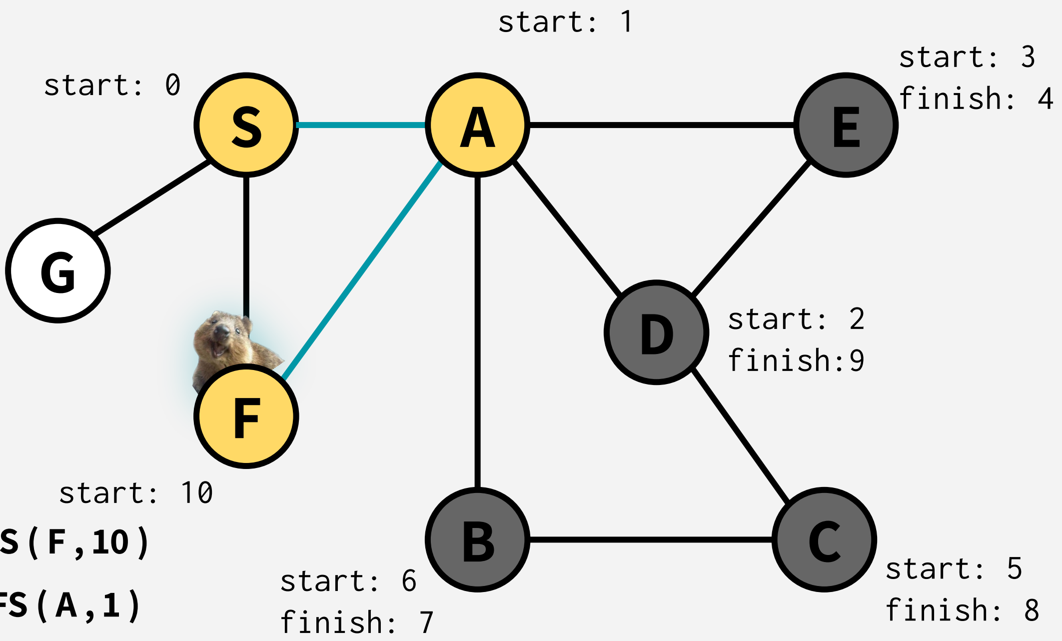
DFS(S,0)

currTime = 10

DEPTH-FIRST SEARCH

An analogy:

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



DFS(F,10)

DFS(A,1)

DFS(S,0)

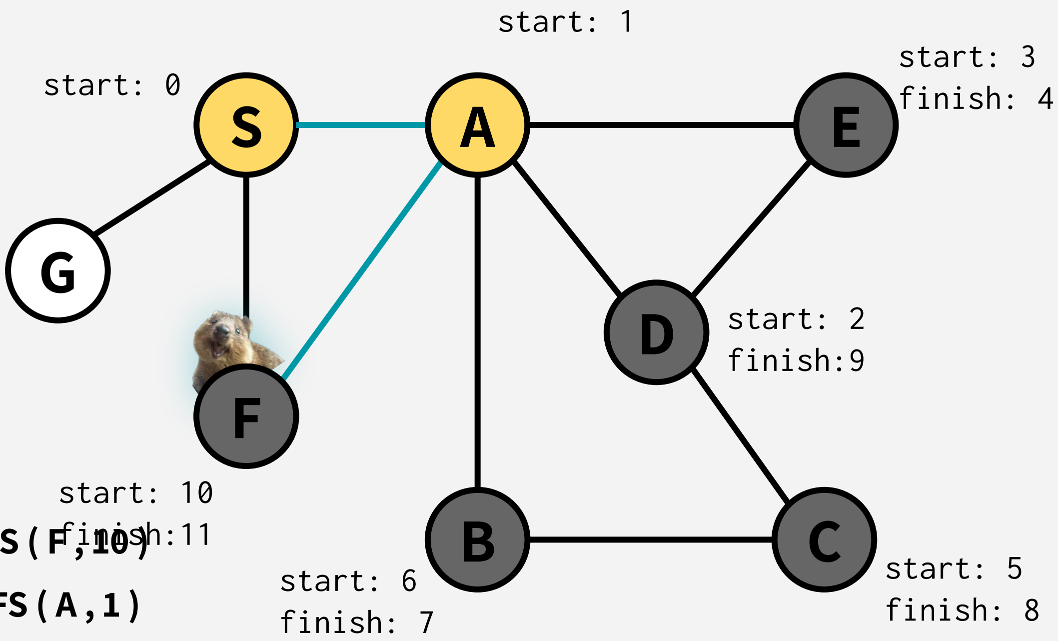
currTime = 11

```
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(F,10)

DFS(A,1)

DFS(S,0)

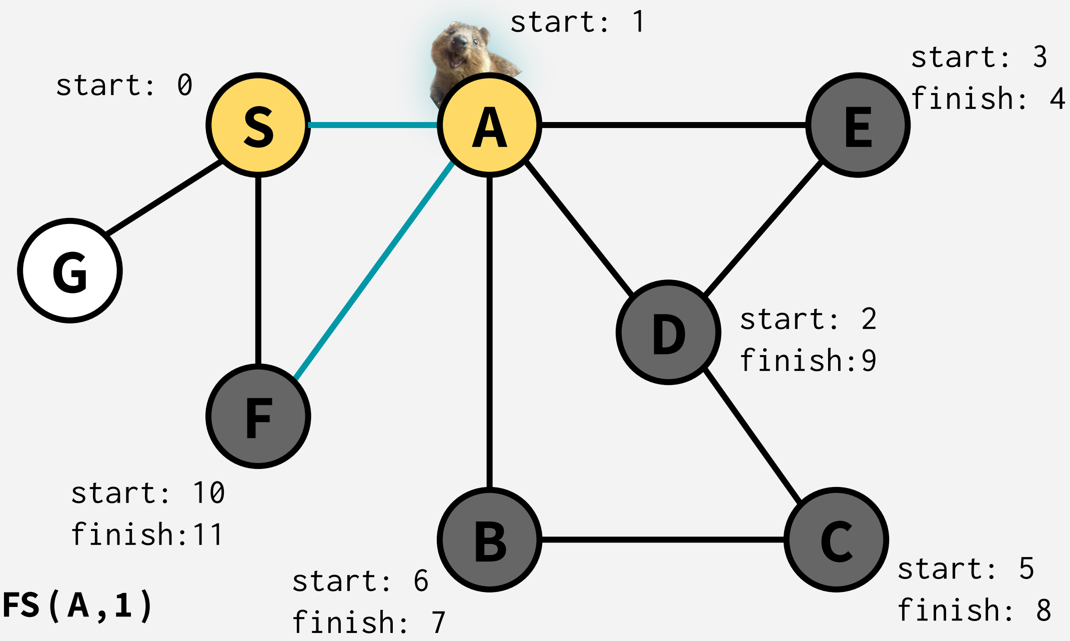
currTime = 11

```
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
```

DFS(A,1)

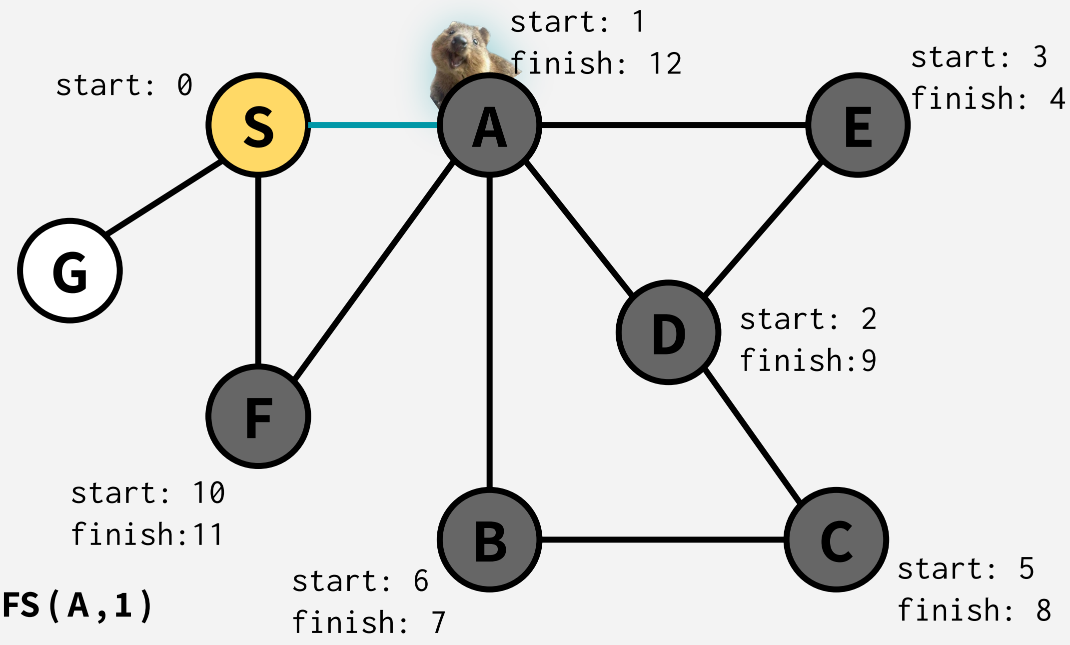
DFS(S,0)

currTime = 12

DEPTH-FIRST SEARCH

An analogy:

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



DFS(A,1)

DFS(S,0)

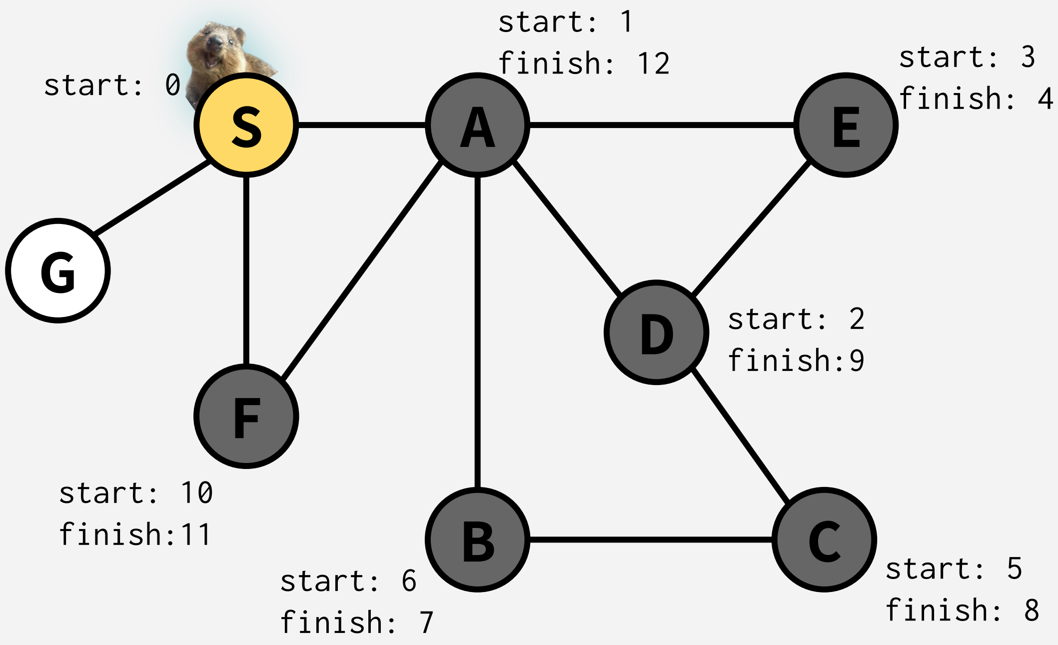
currTime = 12

```
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)



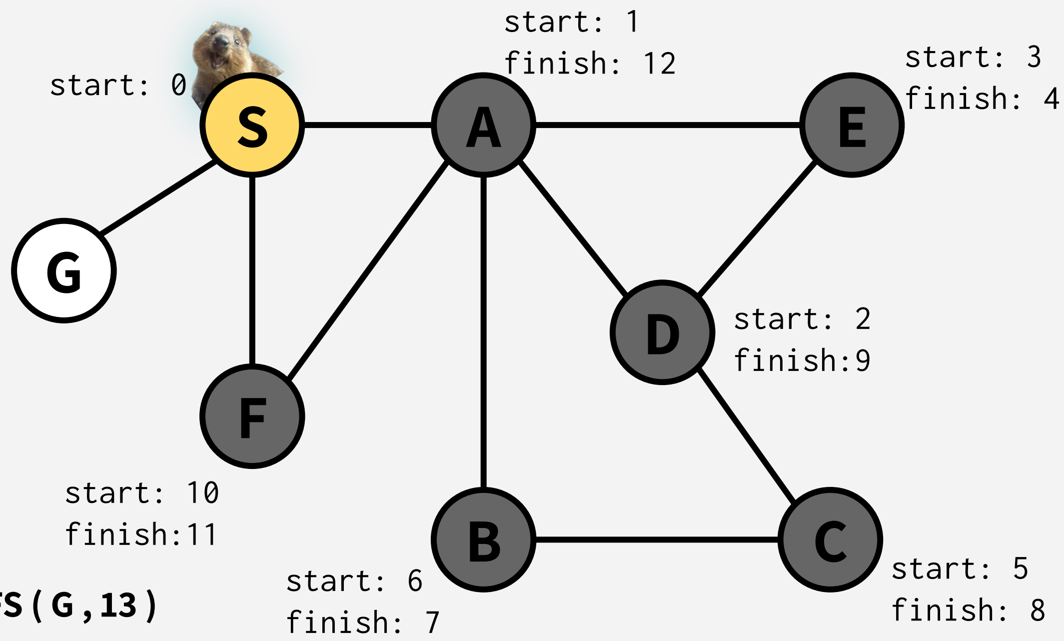

```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DFS(S,0) **currTime = 13**

DEPTH-FIRST SEARCH

An analogy:

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



DFS(G,13)

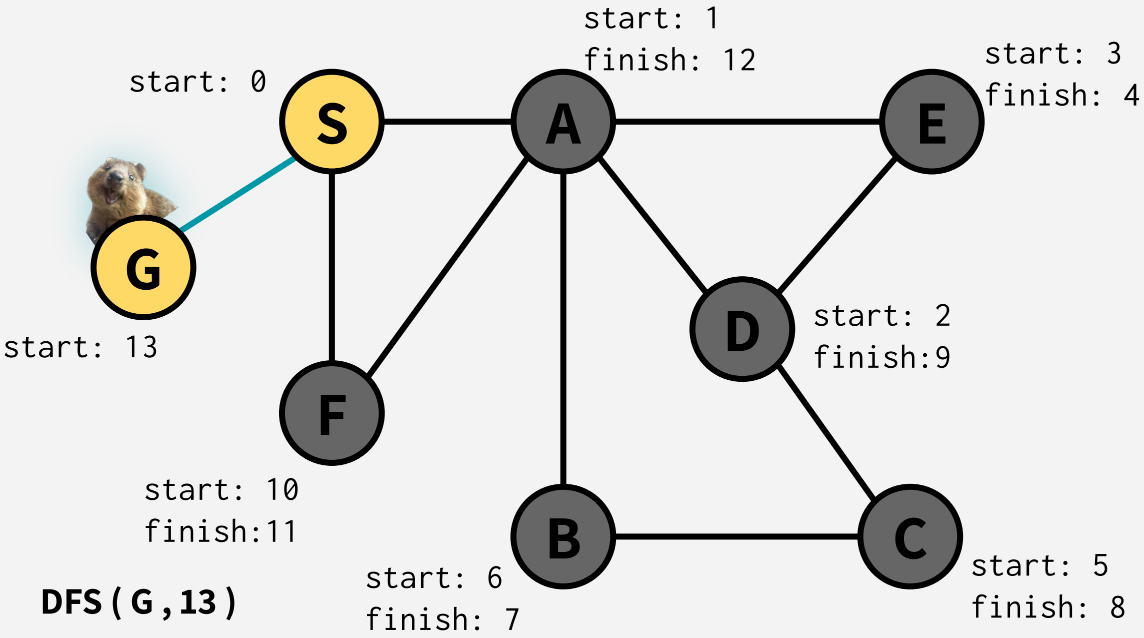
DFS(S,0)

currTime = 13

```
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(G,13)

DFS(S,0)

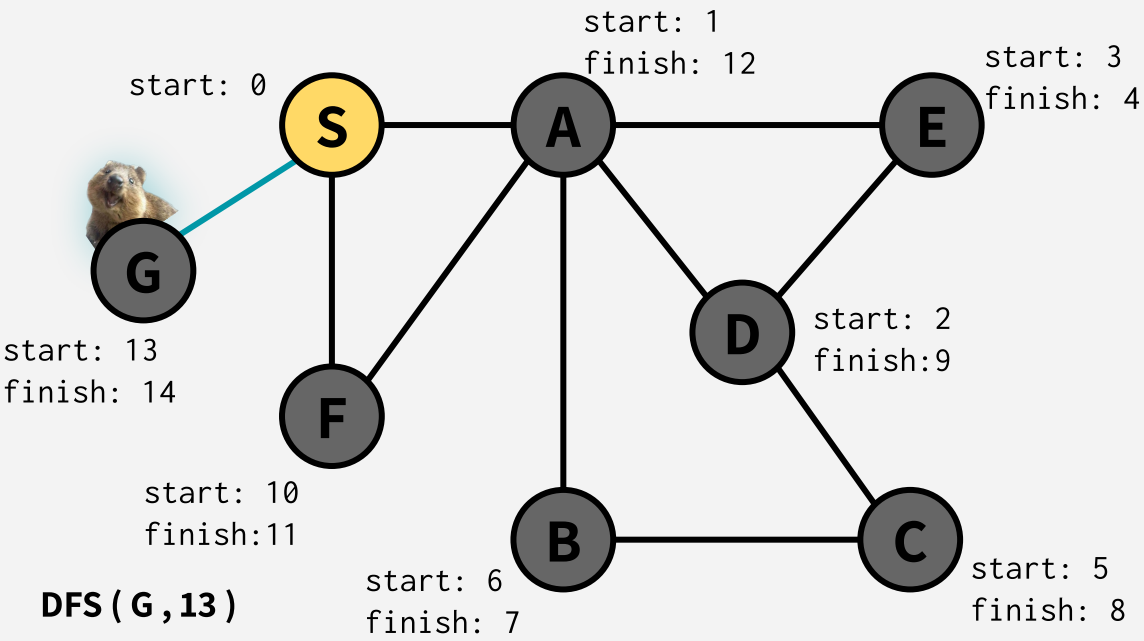
currTime = 14

```
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
```

DFS(G,13)

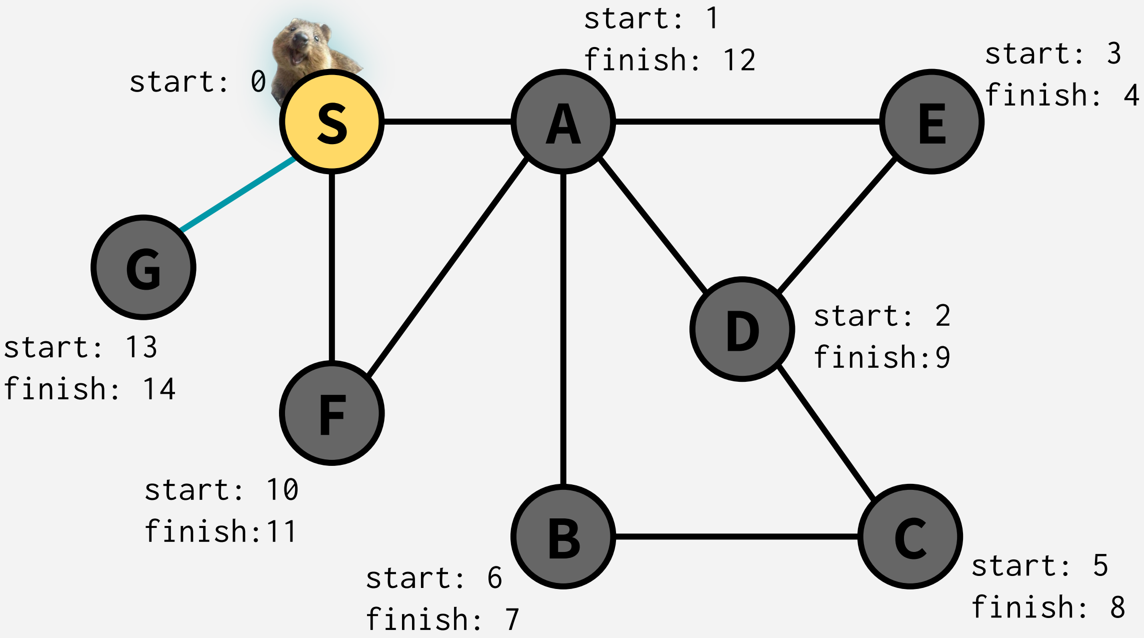
DFS(S,0)

currTime = 14

DEPTH-FIRST SEARCH

An analogy:

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



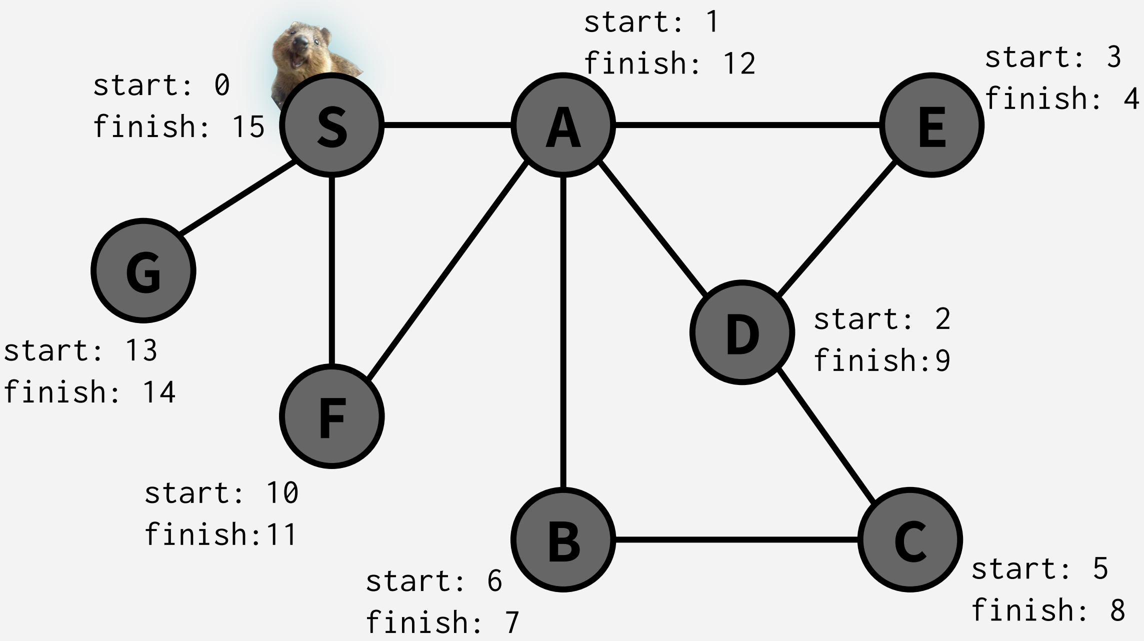

```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DFS(S,0)　　**currTime = 15**

DEPTH-FIRST SEARCH

An analogy:

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



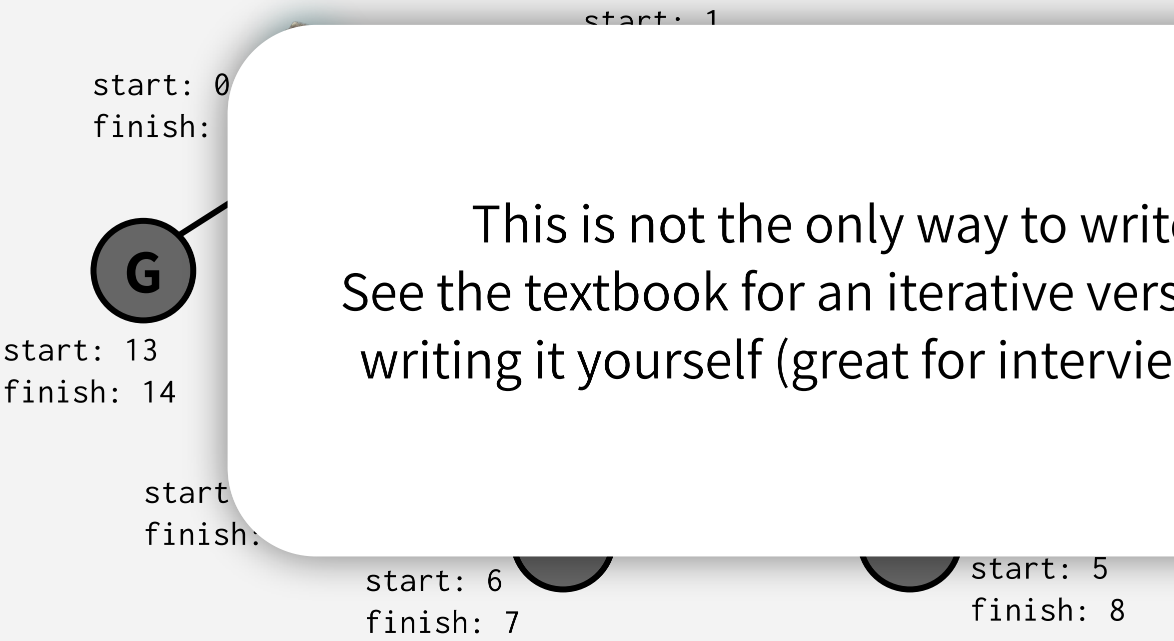

```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DFS(S,0)   **currTime = 15**

DEPTH-FIRST SEARCH

An analogy:

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



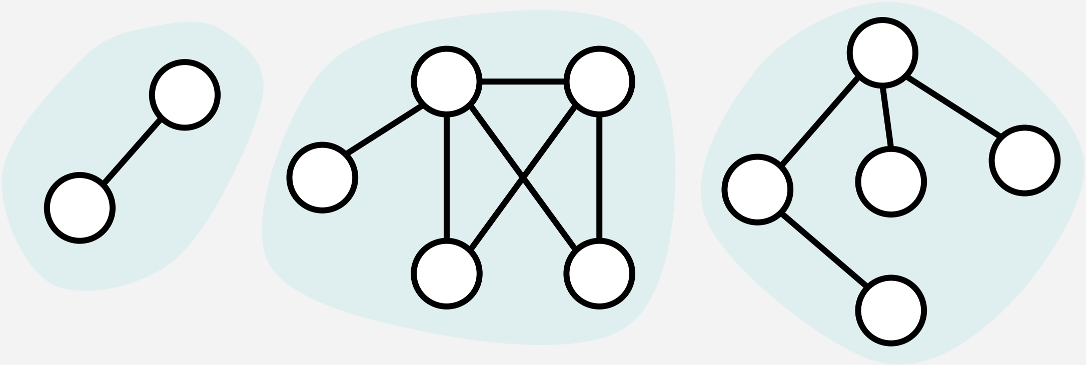
This is not the only way to write DFS!
See the textbook for an iterative version, and try writing it yourself (great for interview practice)

```
ime
ed
bors:
sited:
currTime)
Time
shed
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.**



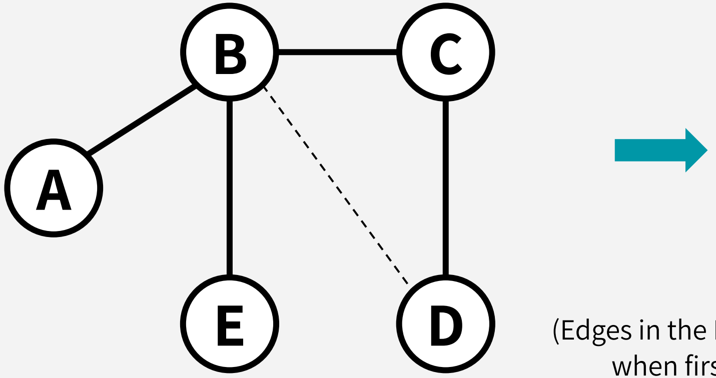
DEPTH-FIRST SEARCH

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)

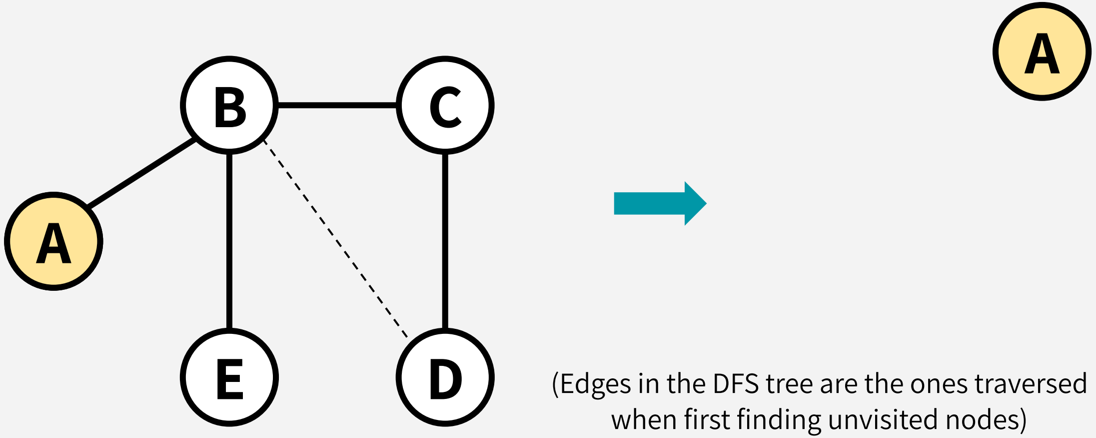
DEPTH-FIRST SEARCH

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.



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

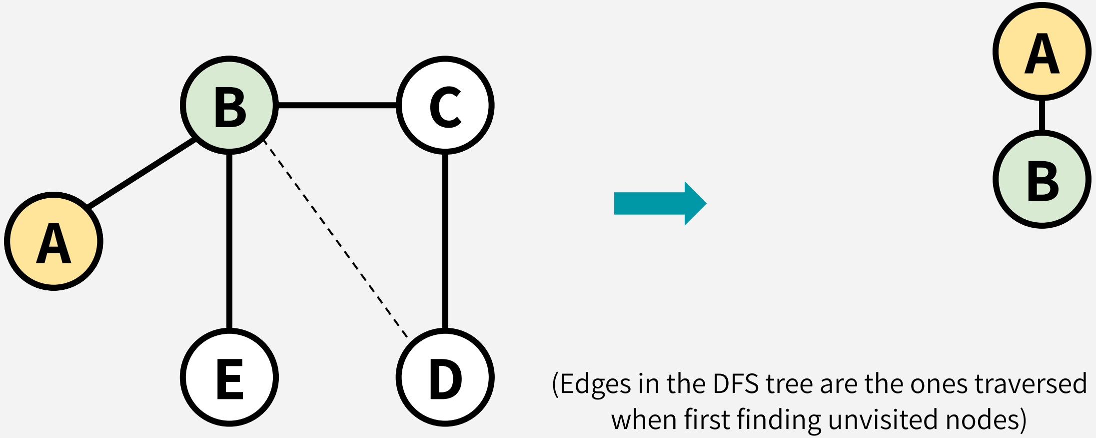
DEPTH-FIRST SEARCH

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.



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

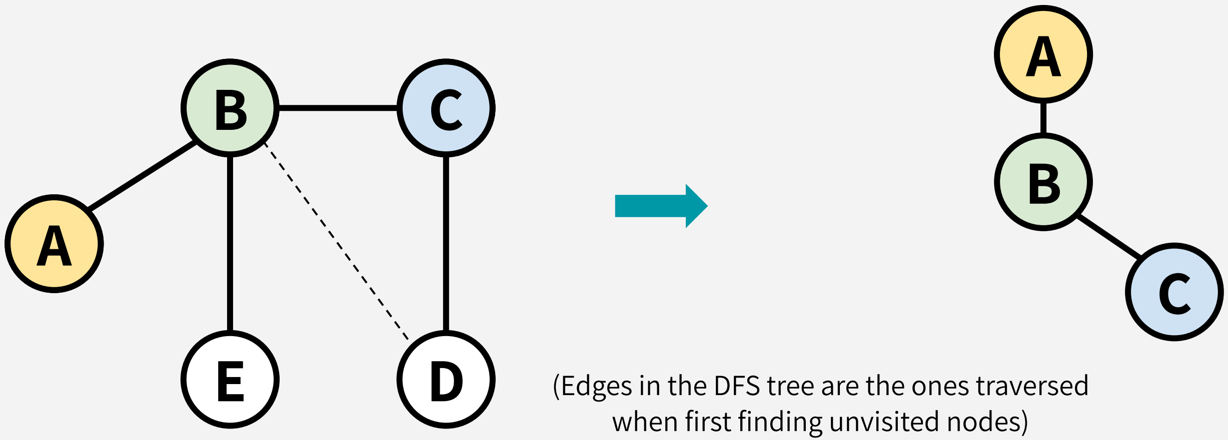
DEPTH-FIRST SEARCH

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.



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

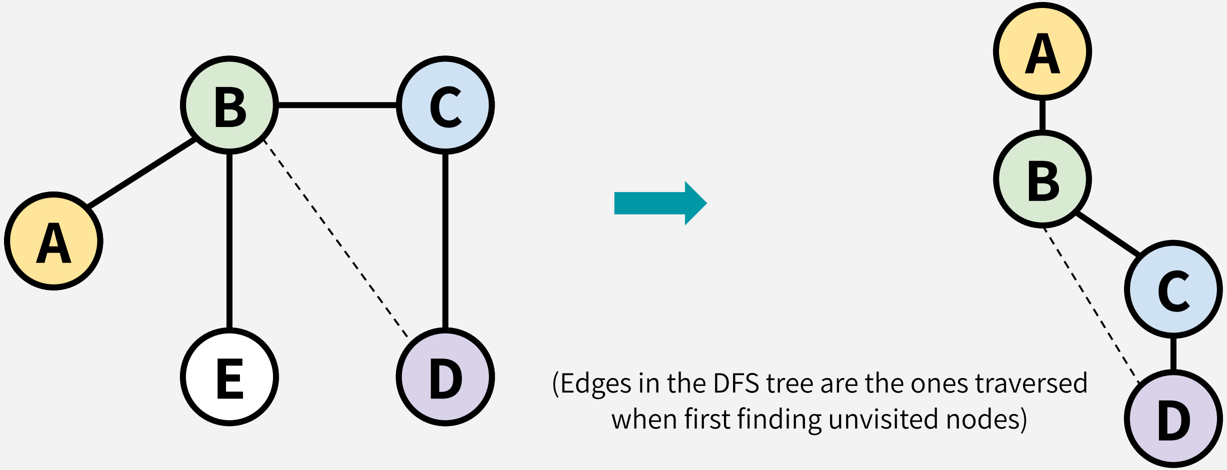
DEPTH-FIRST SEARCH

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.



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

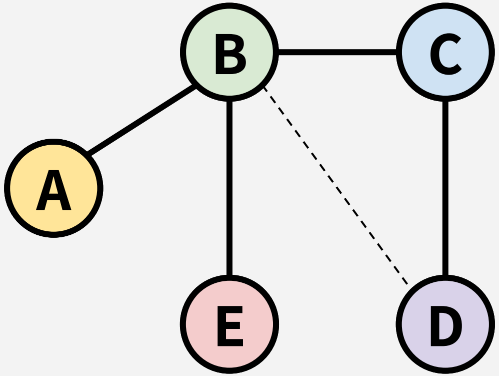
DEPTH-FIRST SEARCH

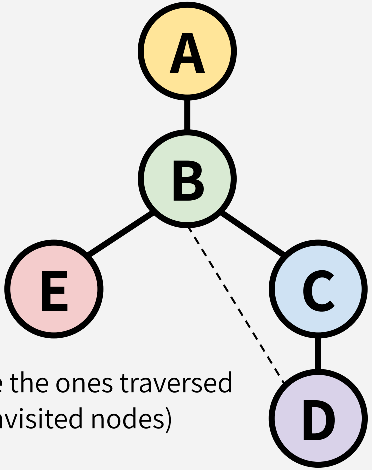
Why is it called depth-first?

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.





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

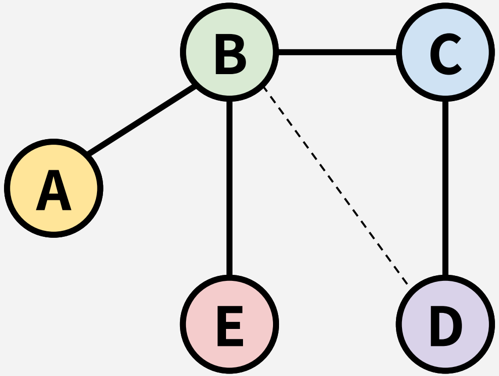
DEPTH-FIRST SEARCH

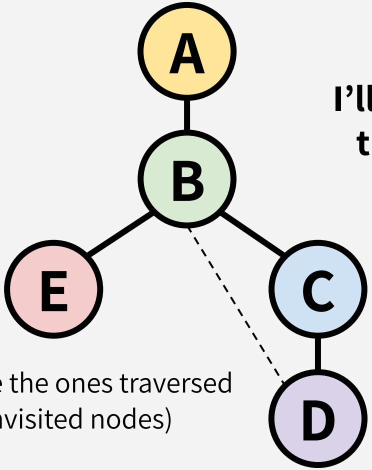
Why is it called depth-first?

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.





I’ll refer to this as the “DFS tree”

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

DEPTH-FIRST SEARCH: RUNTIME

To explore a graph's $\mathbf{i^{th}}$ **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 → **O(deg(v))** total.

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

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:

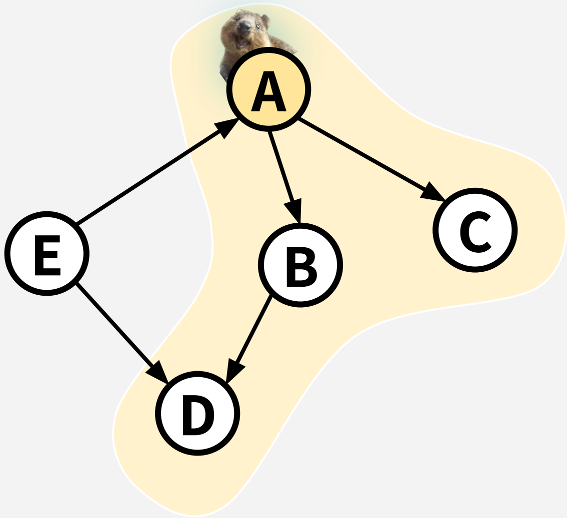
$$O(\sum_i m_i + \sum_i n_i) = \mathbf{O(m + n)}$$

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

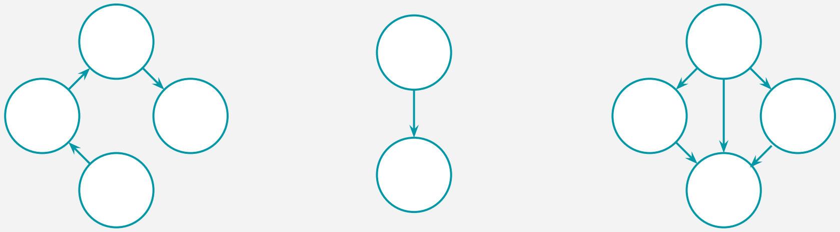
DIRECTED ACYCLIC GRAPH

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

DIRECTED ACYCLIC GRAPH

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

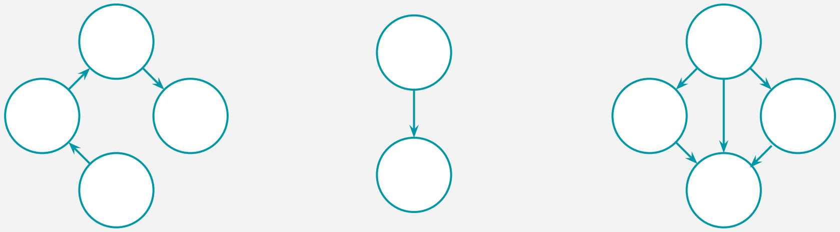
These are DAGs:



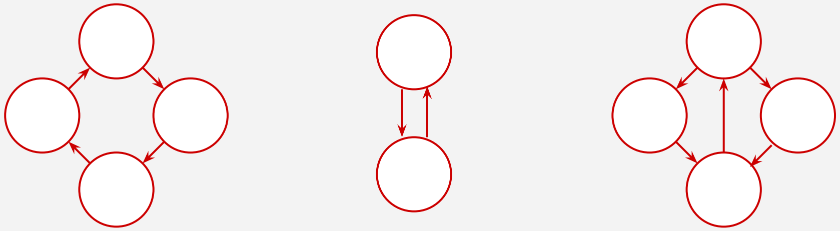
DIRECTED ACYCLIC GRAPH

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?

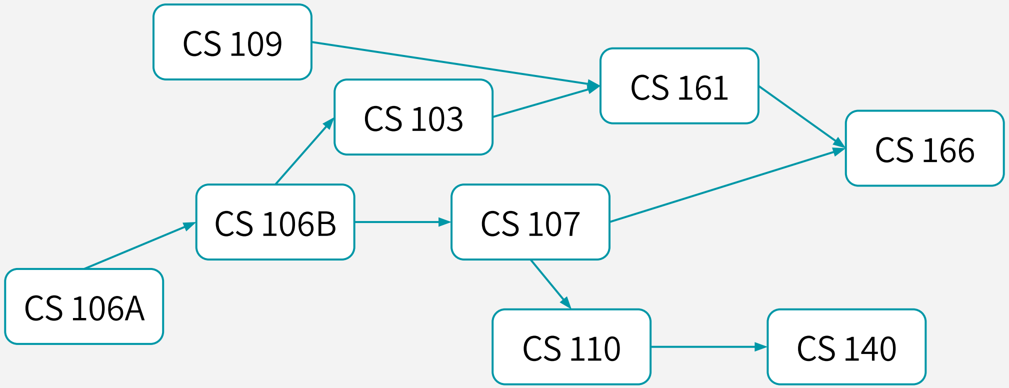
TOPOLOGICAL SORTING: THE TASK

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?

means B "depends" on A
(i.e. take class B after A)



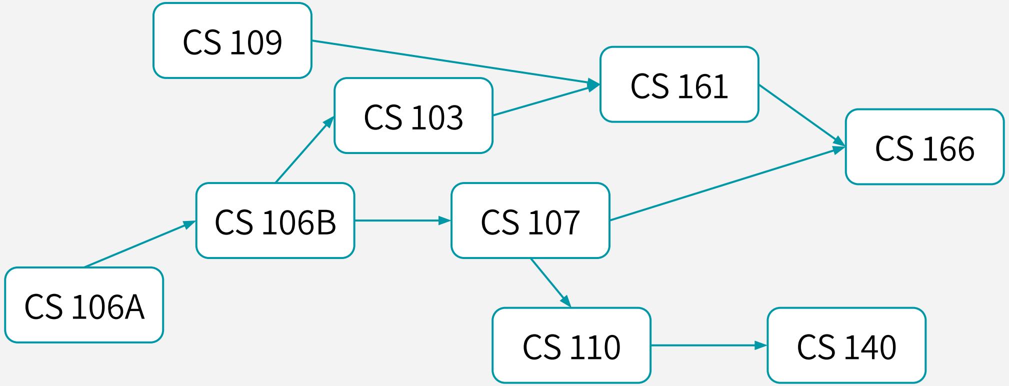
TOPOLOGICAL SORTING: THE TASK

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

<u>Example applications:</u>

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. CS110 must come before CS140)

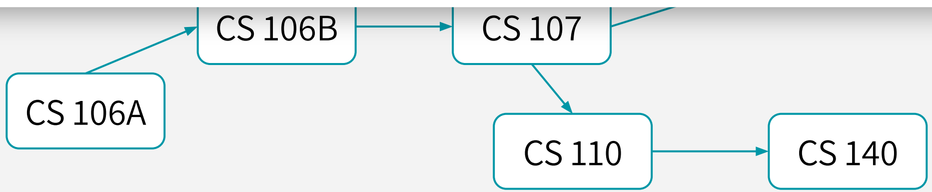
te

“d

G!

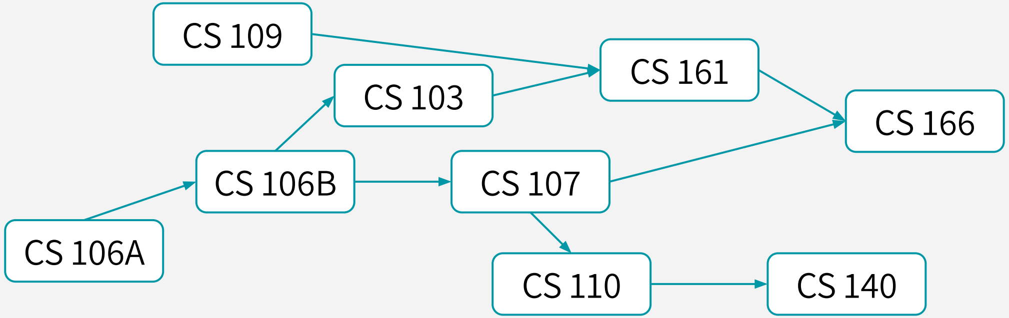
(i.e. take class B after A)

(directed & acyclic)



TOPOLOGICAL SORTING: THE TASK

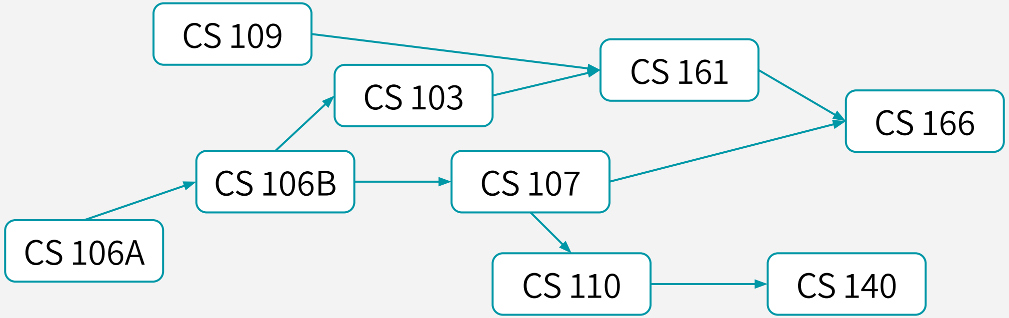
It's helpful to think of this as **"linearizing"** the graph, where all edges point to the **right**



A correct "toposort" of this DAG:

TOPOLOGICAL SORTING: THE TASK

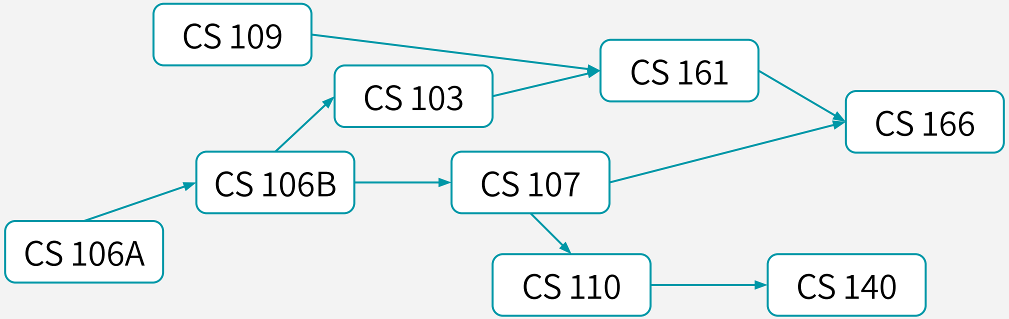
It's helpful to think of this as **"linearizing"** the graph, where all edges point to the **right**



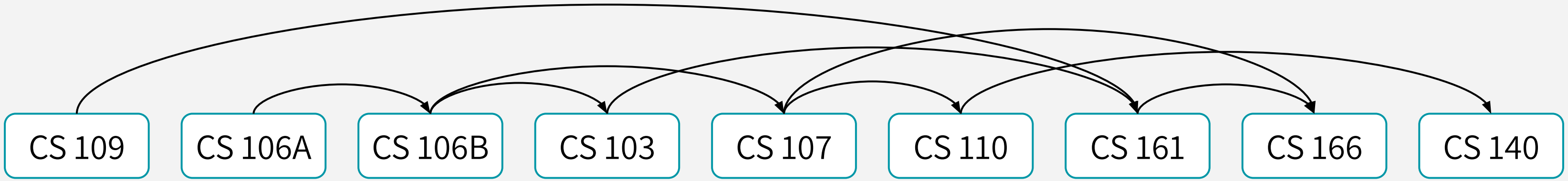
A correct "toposort" of this DAG:

TOPOLOGICAL SORTING: THE TASK

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

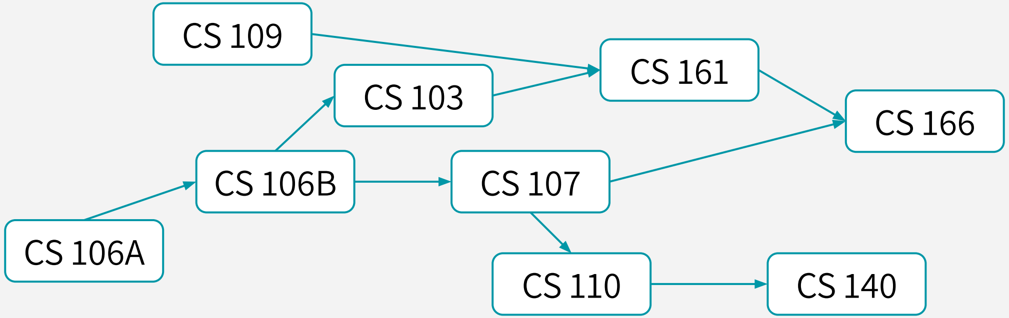


A correct "toposort" of this DAG:



TOPOLOGICAL SORTING: THE TASK

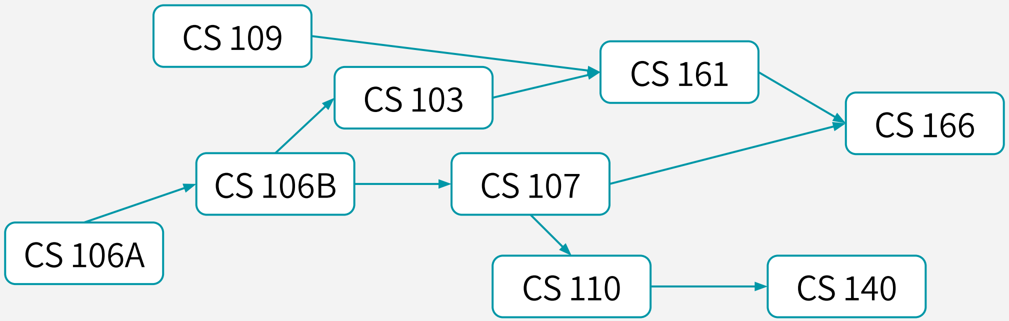
It's helpful to think of this as **"linearizing"** the graph, where all edges point to the **right**



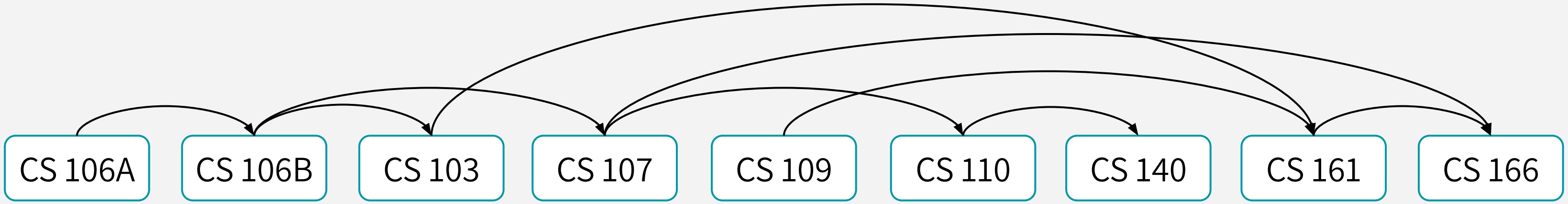
Also a correct toposort of this DAG:

TOPOLOGICAL SORTING: THE TASK

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

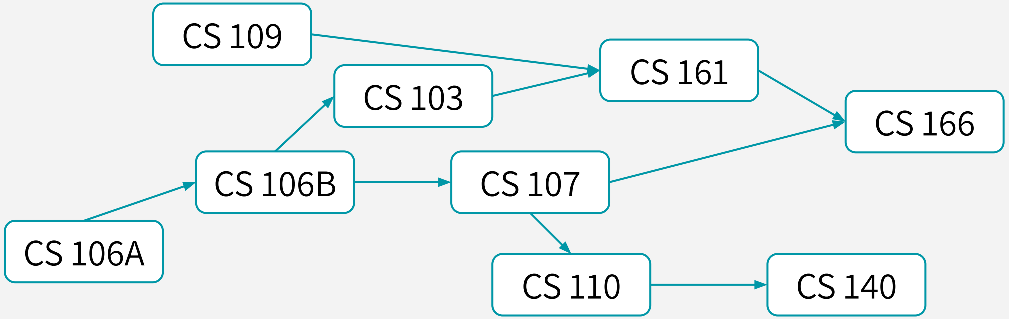


Also a correct toposort of this DAG:



TOPOLOGICAL SORTING: THE TASK

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

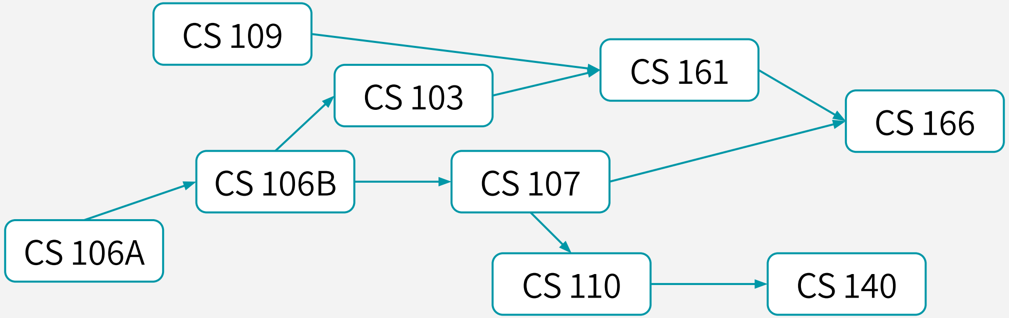


An *INCORRECT* toposort of this DAG:

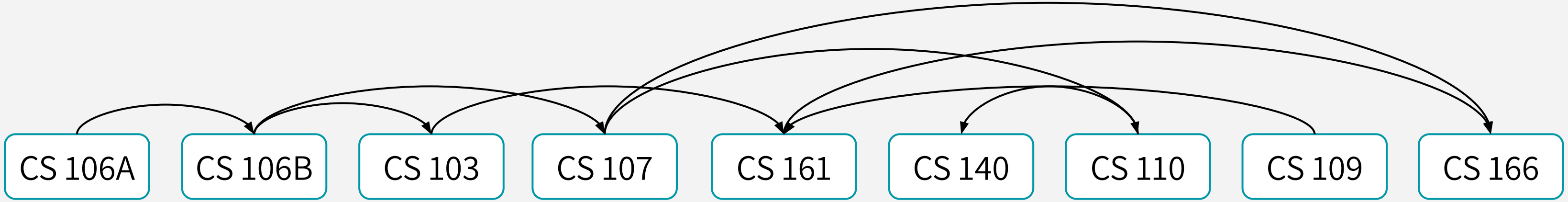
CS 106A | CS 106B | CS 103 | CS 107 | CS 161 | CS 140 | CS 110 | CS 109 | CS 166

TOPOLOGICAL SORTING: THE TASK

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

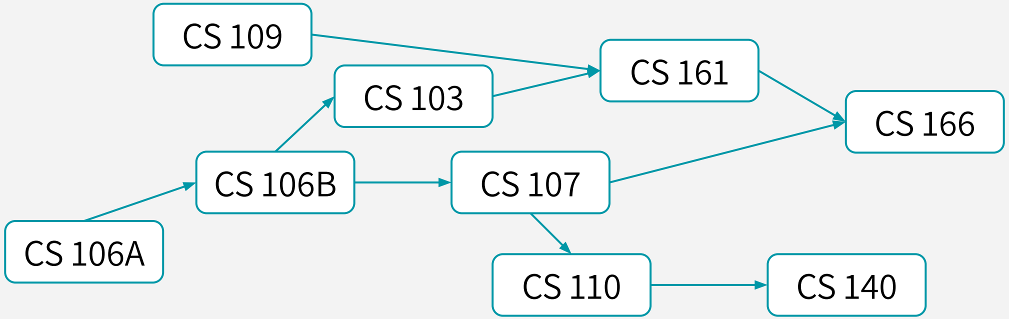


An *INCORRECT* toposort of this DAG:

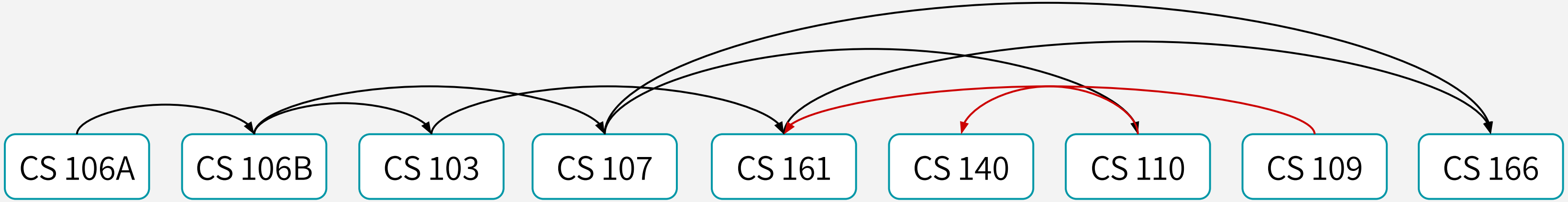


TOPOLOGICAL SORTING: THE TASK

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



An *INCORRECT* 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?

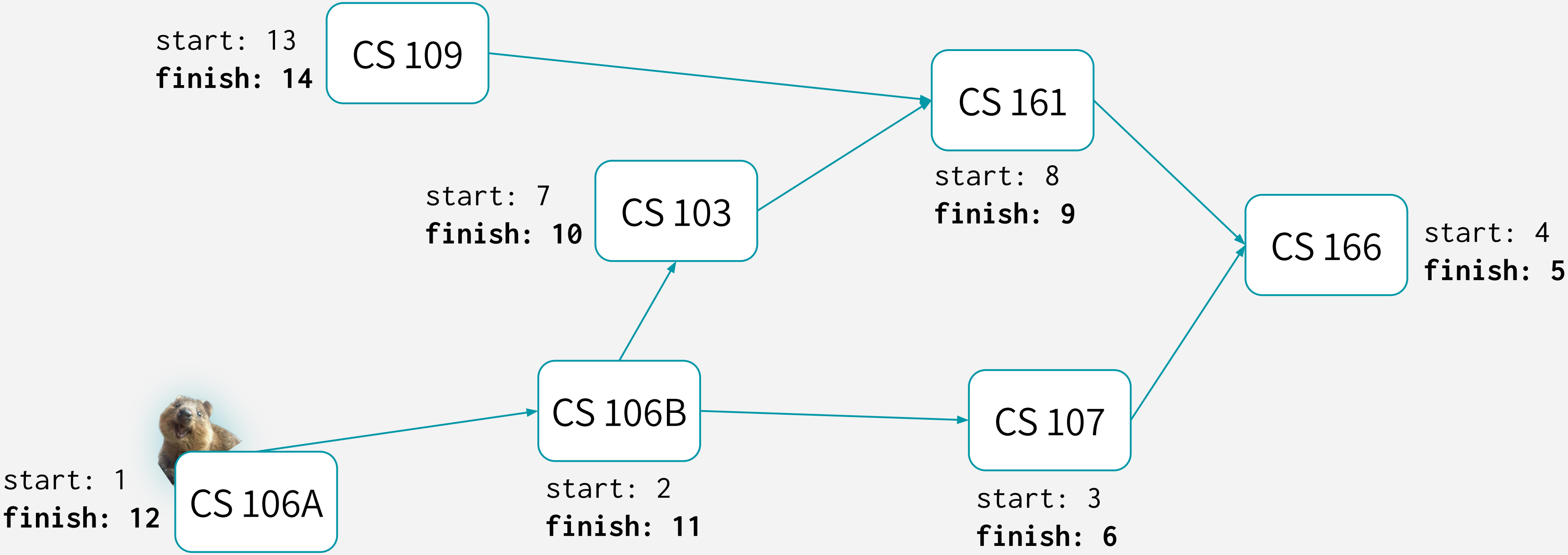
Toposort gives us a priority ordering of nodes (e.g. more intro classes are "higher priority" than more advanced classes). Edges in DAGs clearly illustrate priority: edge from **x** to **y** means **x** has priority over **y**.

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 **v** → **w**, **v**'s finish time will be *larger* than **w**'s finish time

Let's consider two cases: (1) DFS visits **v** first, or (2) DFS visits **w** first.

start: 1
finish: 12

CS 106A

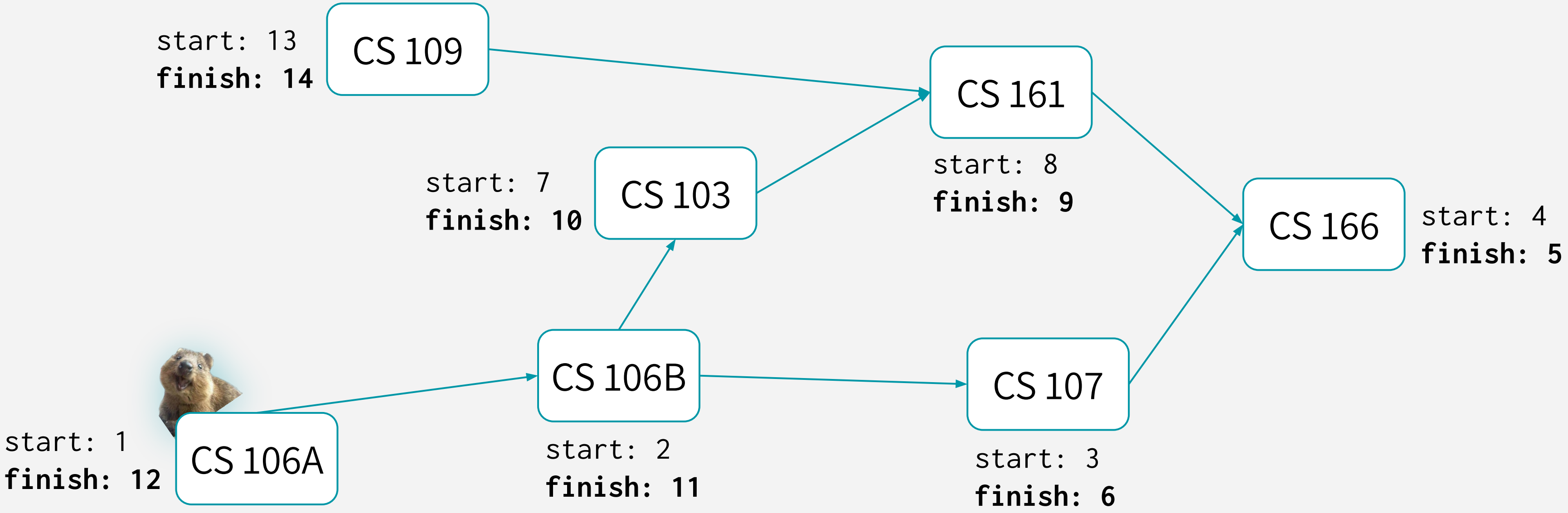
start: 2
finish: 11

start: 3
finish: 6

DFS WILL GET US A TOPOSORT

CLAIM: In general, if there’s an edge from **v** → **w**, **v**’s finish time will be *larger* than **w**’s finish time

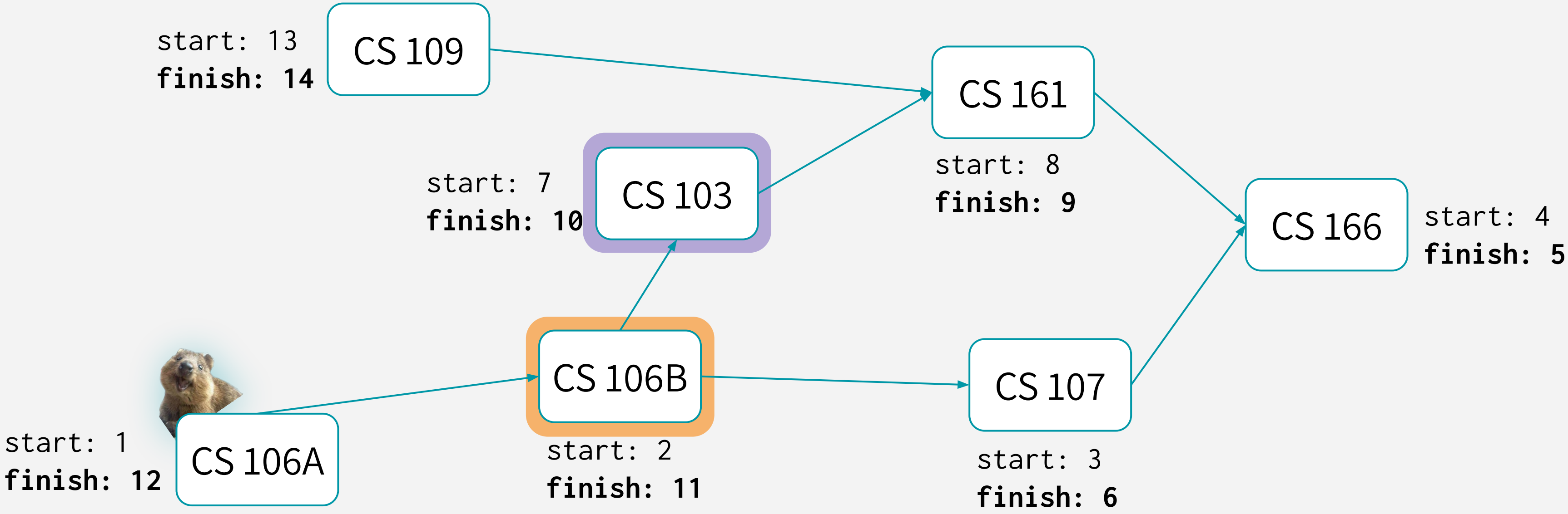
<u>CASE 1</u>: **v** → **w**, and **v is discovered first** by DFS



DFS WILL GET US A TOPOSORT

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

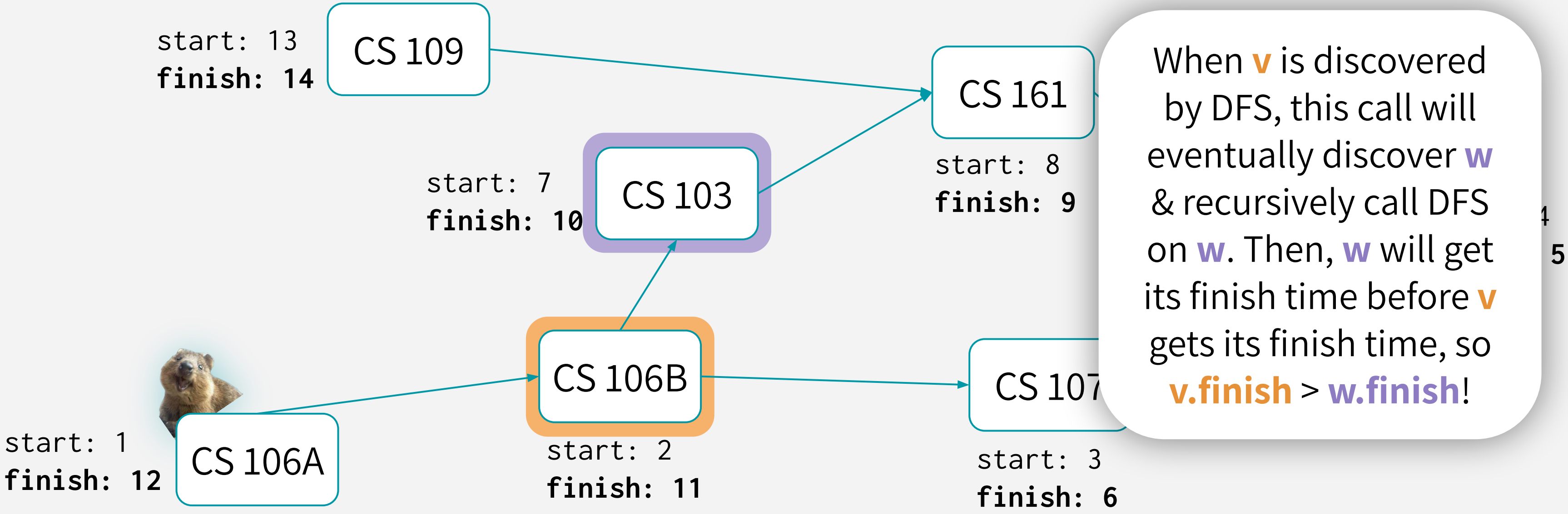
<u>CASE 1</u>: **v** → **w**, and **v** **is discovered first** by DFS



DFS WILL GET US A TOPOSORT

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

CASE 1: **v** → **w**, and **v is discovered first** by DFS

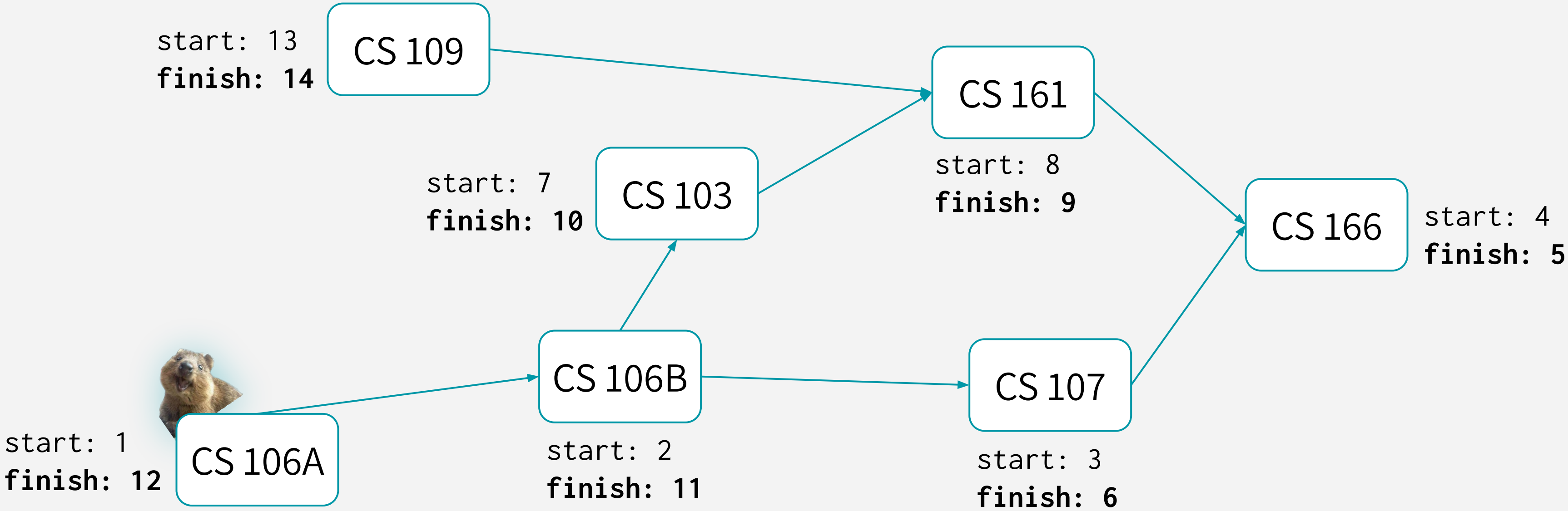


When **v** is discovered by DFS, this call will eventually discover **w** & recursively call DFS on **w**. Then, **w** will get its finish time before **v** gets its finish time, so **v.finish** > **w.finish**!

DFS WILL GET US A TOPOSORT

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

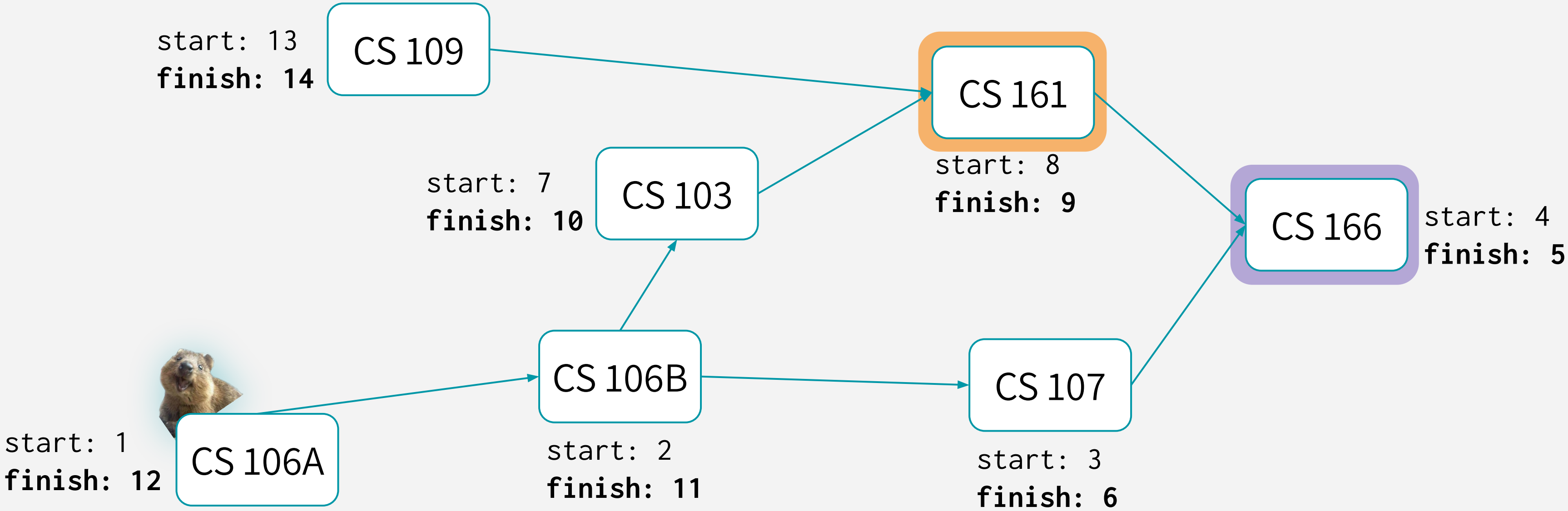
<u>CASE 2</u>: **v** → **w**, and **w is discovered first** by DFS



DFS WILL GET US A TOPOSORT

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

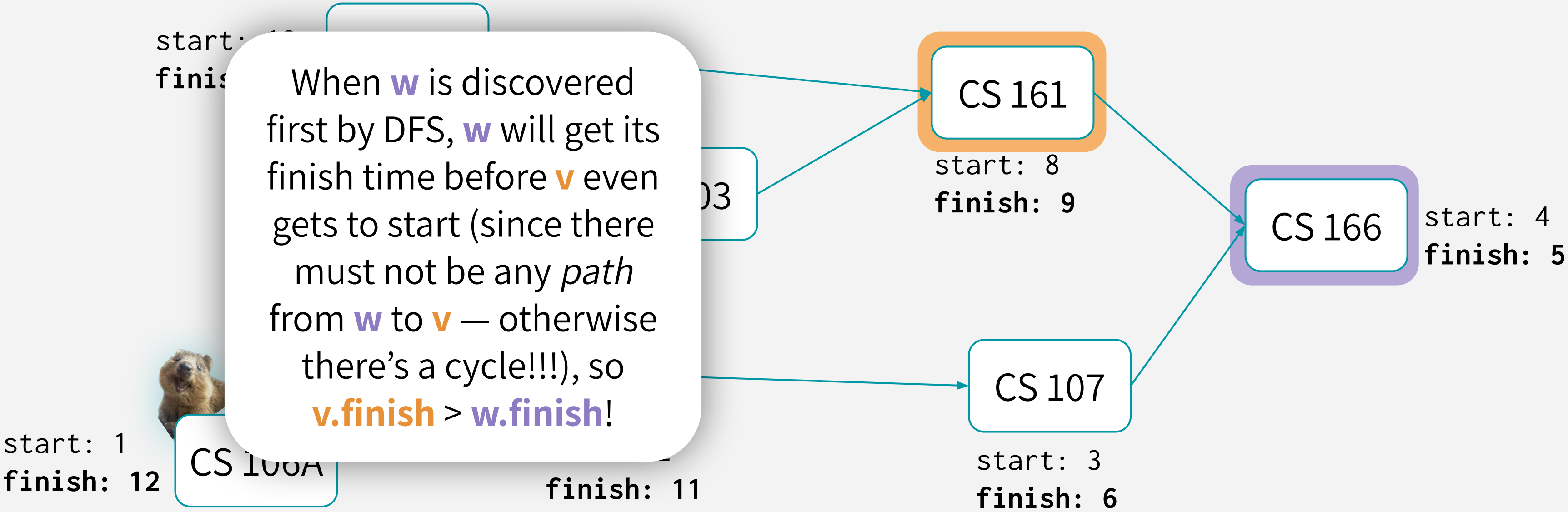
<u>CASE 2</u>: **v** → **w**, and **w** **is discovered first** by DFS



DFS WILL GET US A TOPOSORT

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

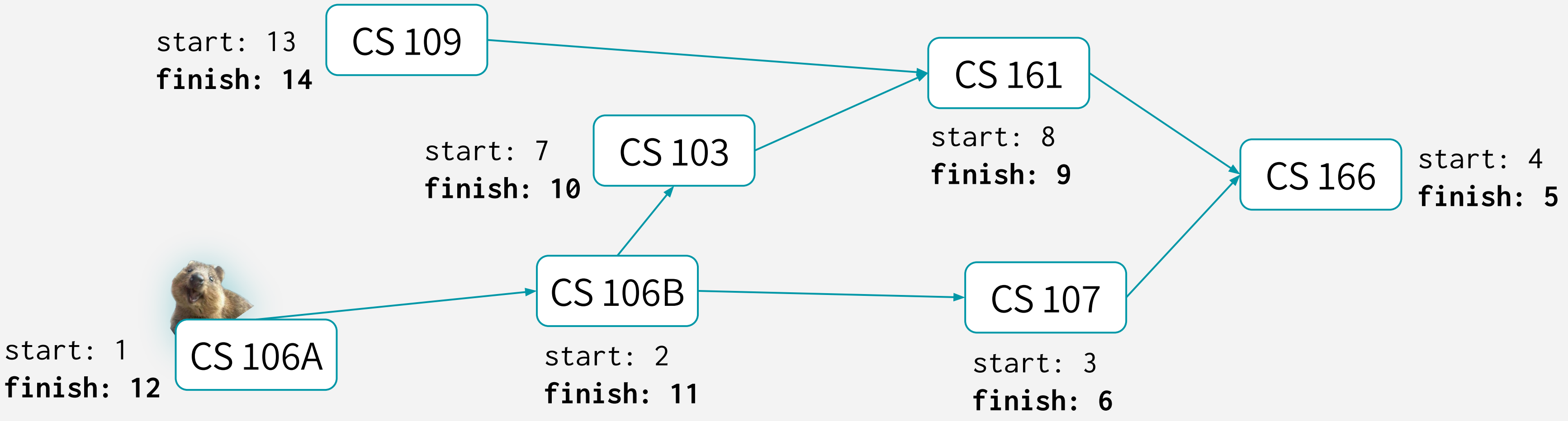
<u>CASE 2</u>: **v** → **w**, and **w is discovered first** by DFS



When **w** is discovered first by DFS, **w** will get its finish time before **v** even gets to start (since there must not be any *path* from **w** to **v** — otherwise there's a cycle!!!), so **v.finish** > **w.finish**!

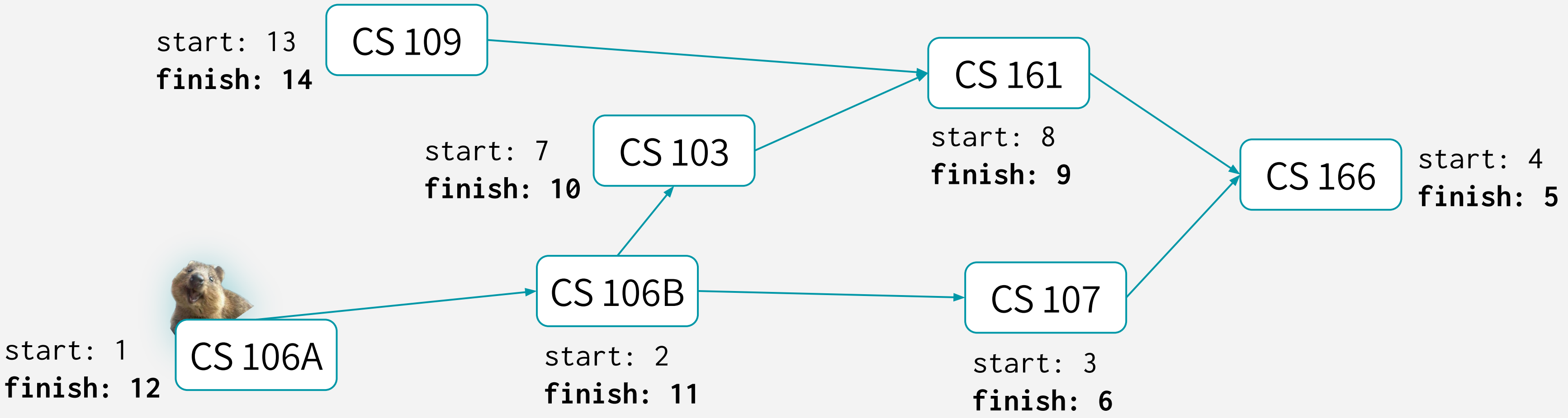
DFS WILL GET US A TOPOSORT

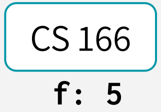
TOPOSORT: Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



DFS WILL GET US A TOPOSORT

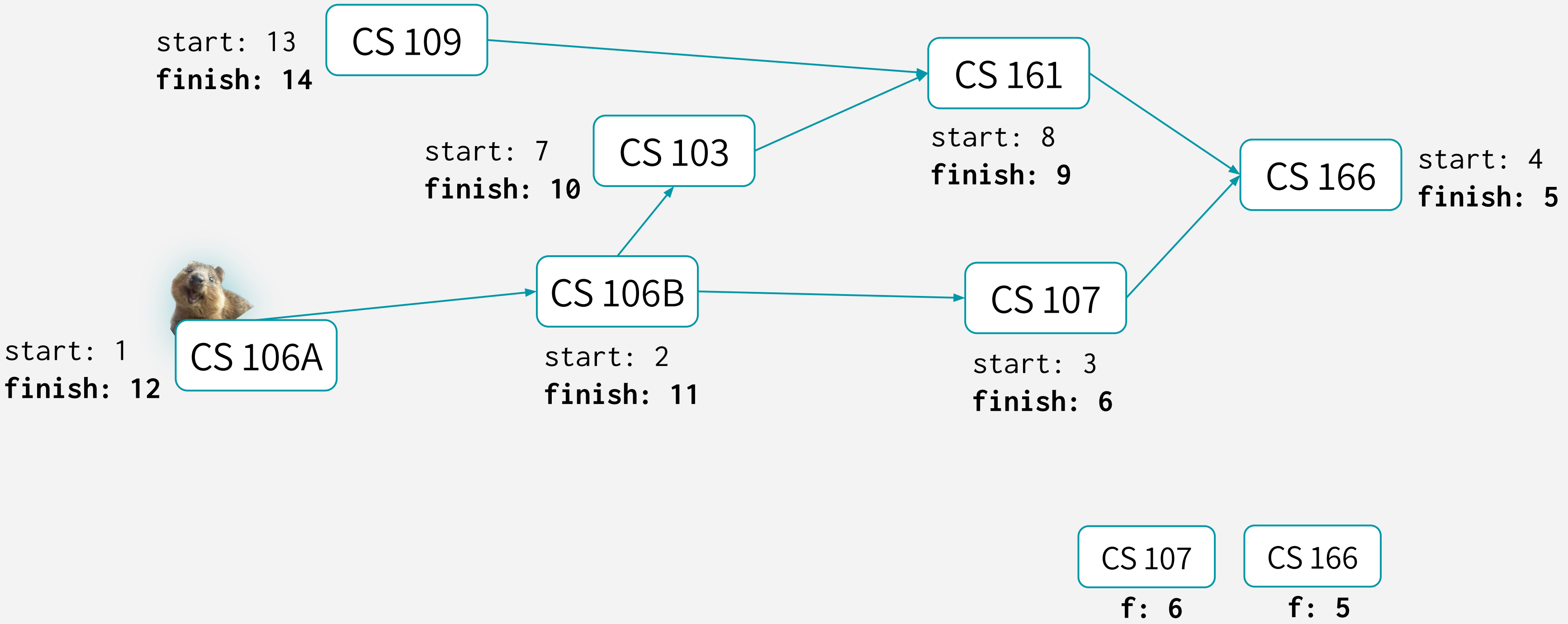
TOPOSORT: Perform DFS. When a vertex gets its finish time, insert it at the start of the list.





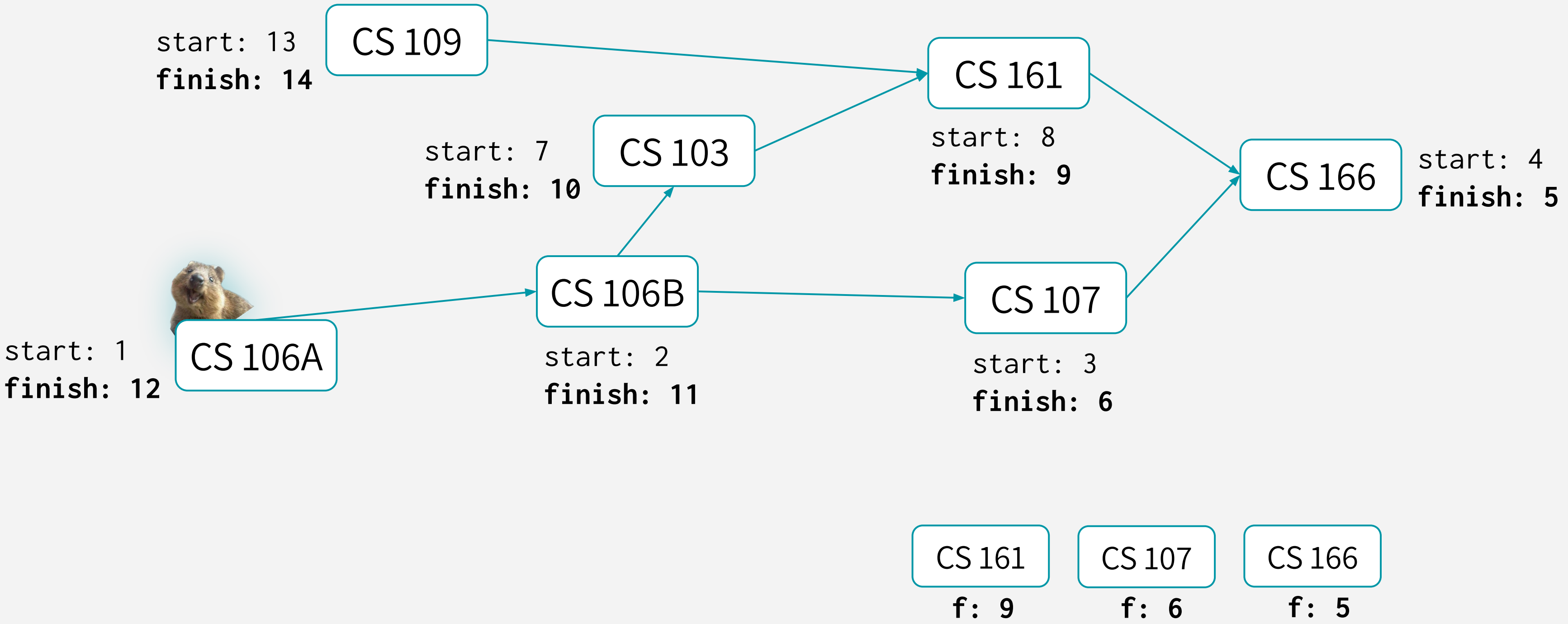
DFS WILL GET US A TOPOSORT

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



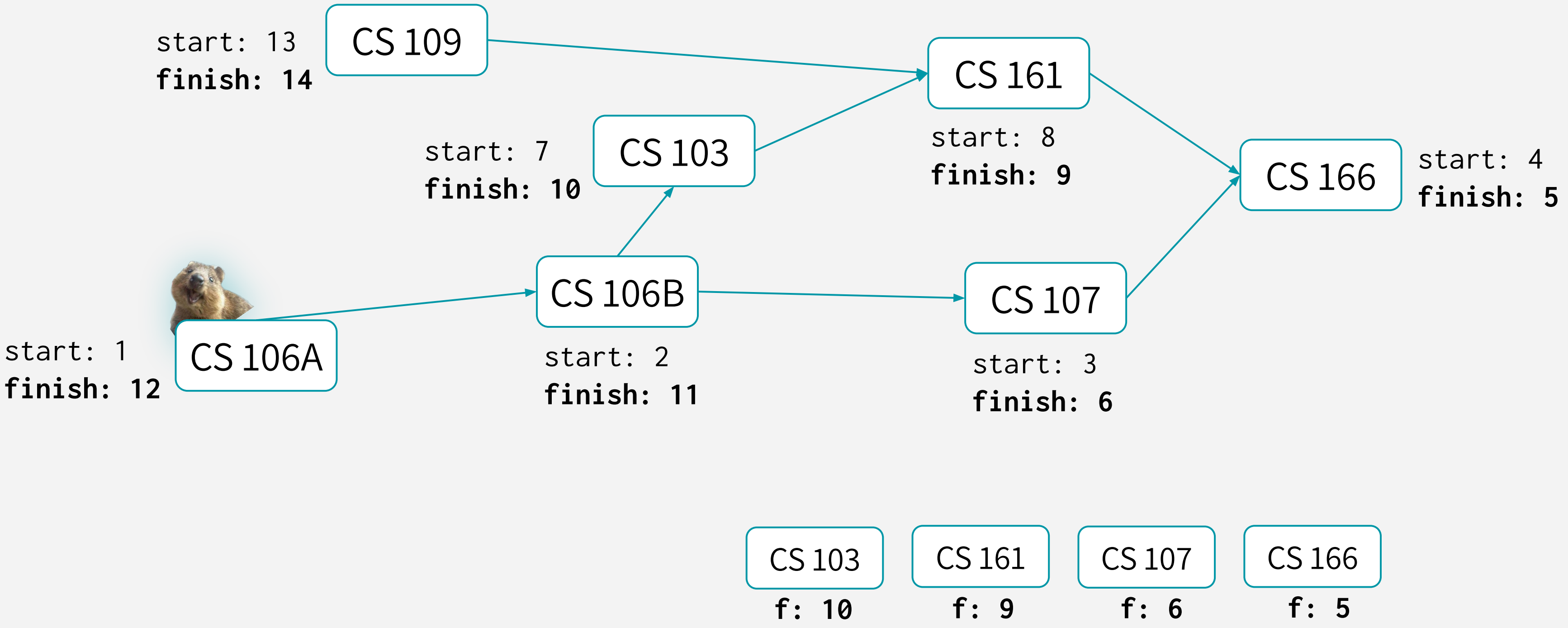
DFS WILL GET US A TOPOSORT

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



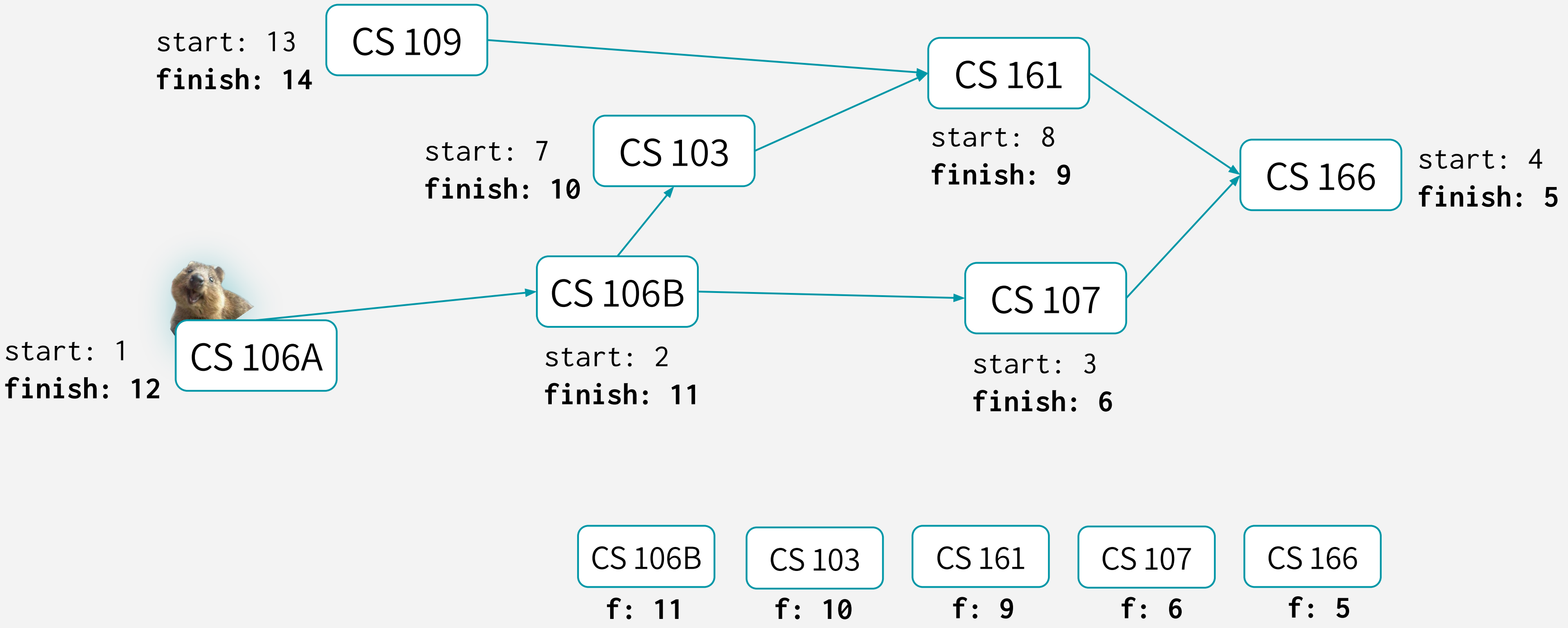
DFS WILL GET US A TOPOSORT

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



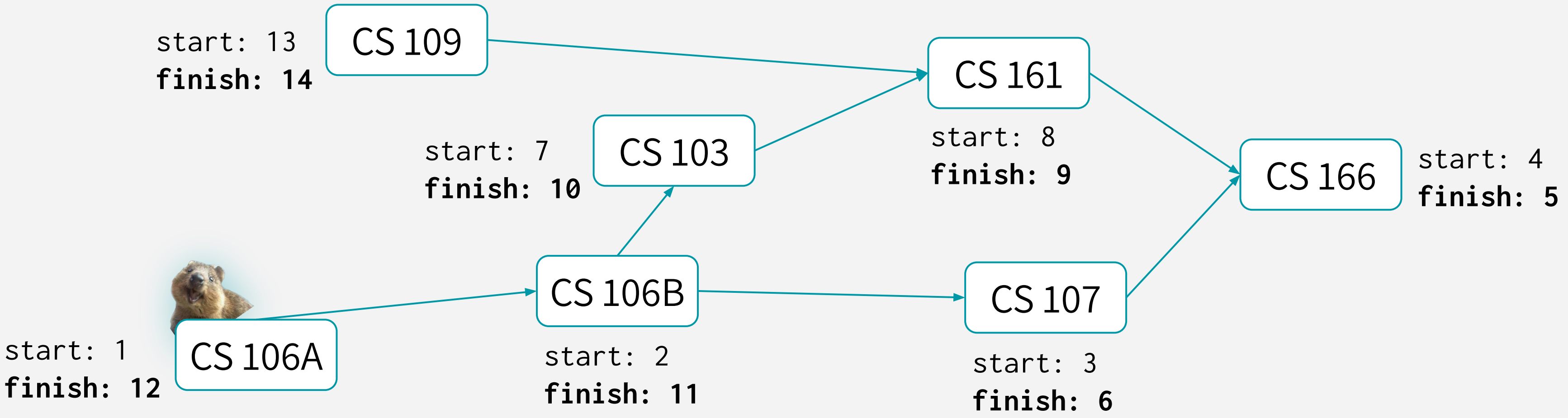
DFS WILL GET US A TOPOSORT

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



DFS WILL GET US A TOPOSORT

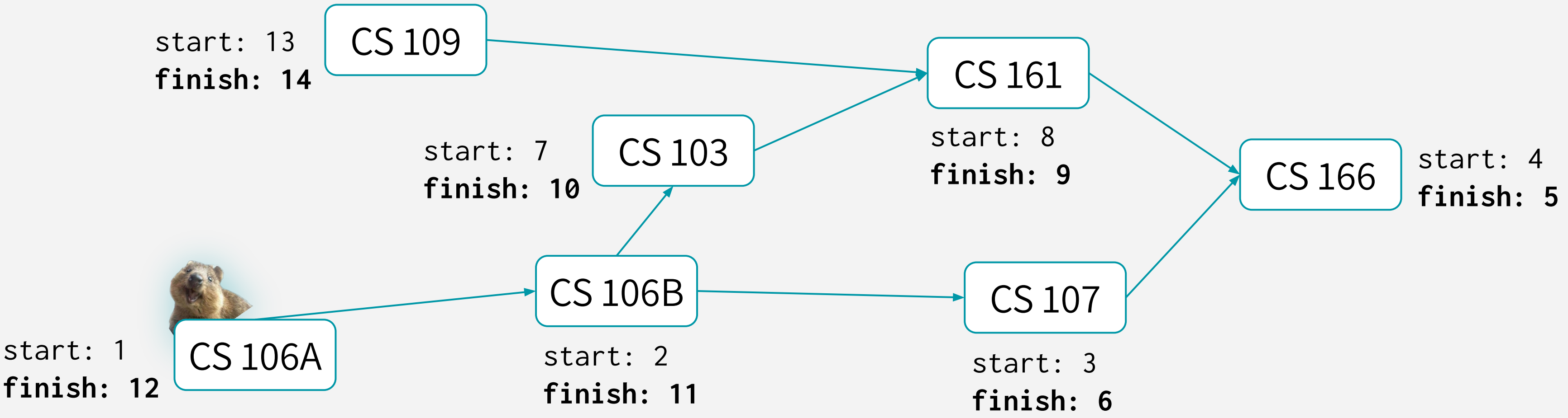
TOPOSORT: Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



CS 106A	CS 106B	CS 103	CS 161	CS 107	CS 166
f: 12	f: 11	f: 10	f: 9	f: 6	f: 5

DFS WILL GET US A TOPOSORT

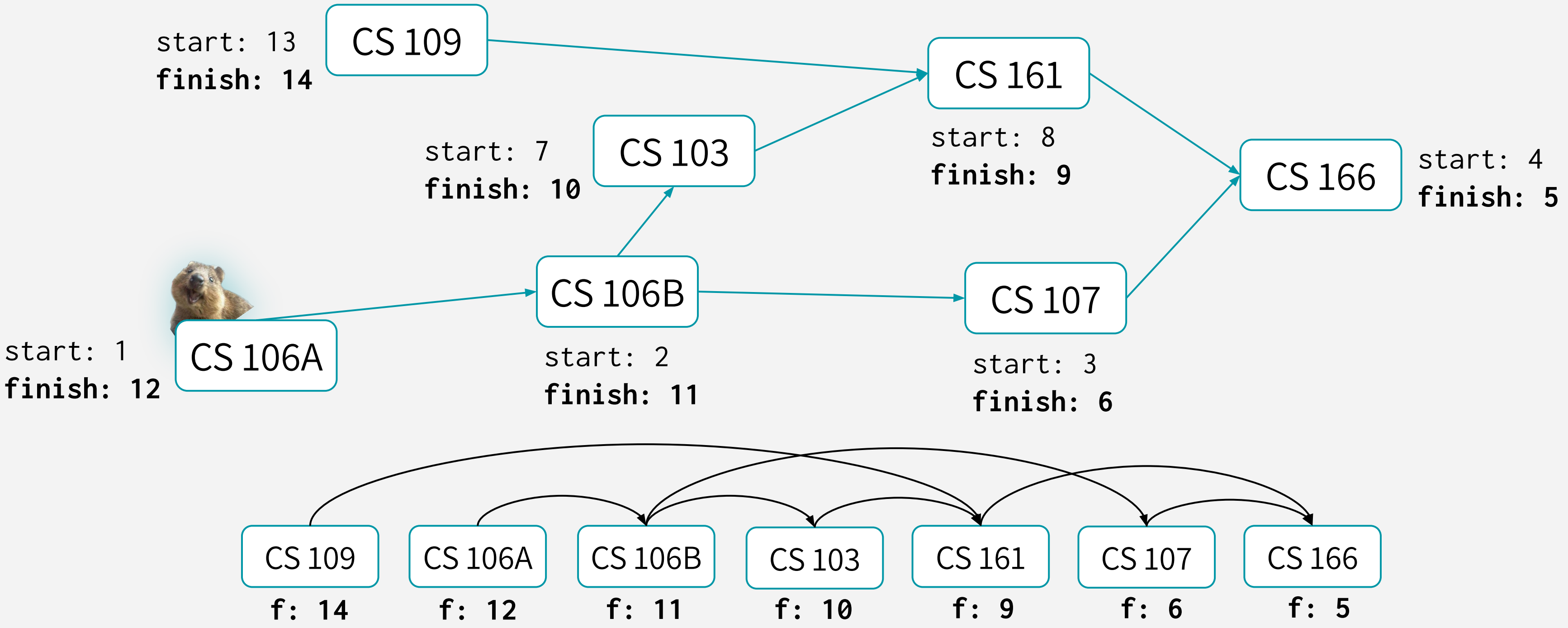
TOPOSORT: Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



CS 109	CS 106A	CS 106B	CS 103	CS 161	CS 107	CS 166
f: 14	f: 12	f: 11	f: 10	f: 9	f: 6	f: 5

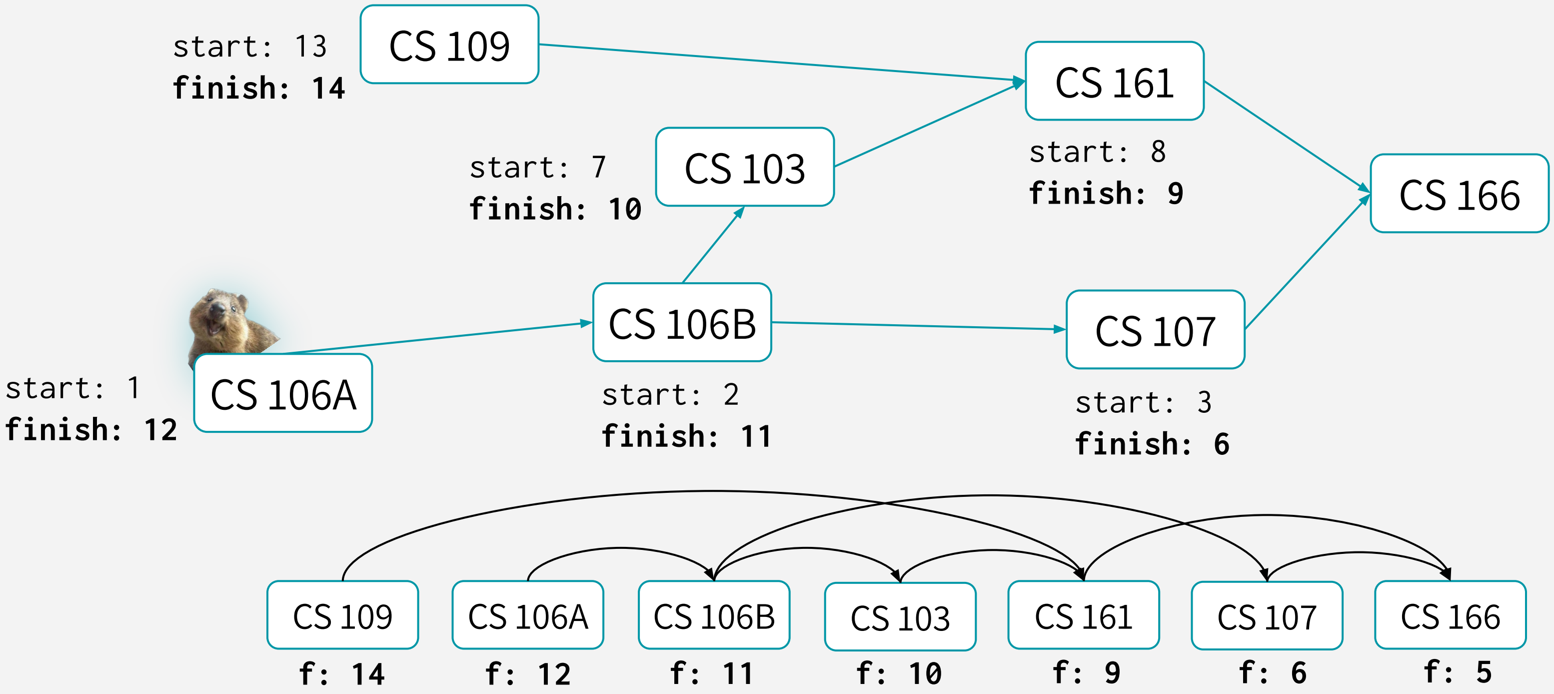
DFS WILL GET US A TOPOSORT

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



DFS WILL GET US A TOPOSORT

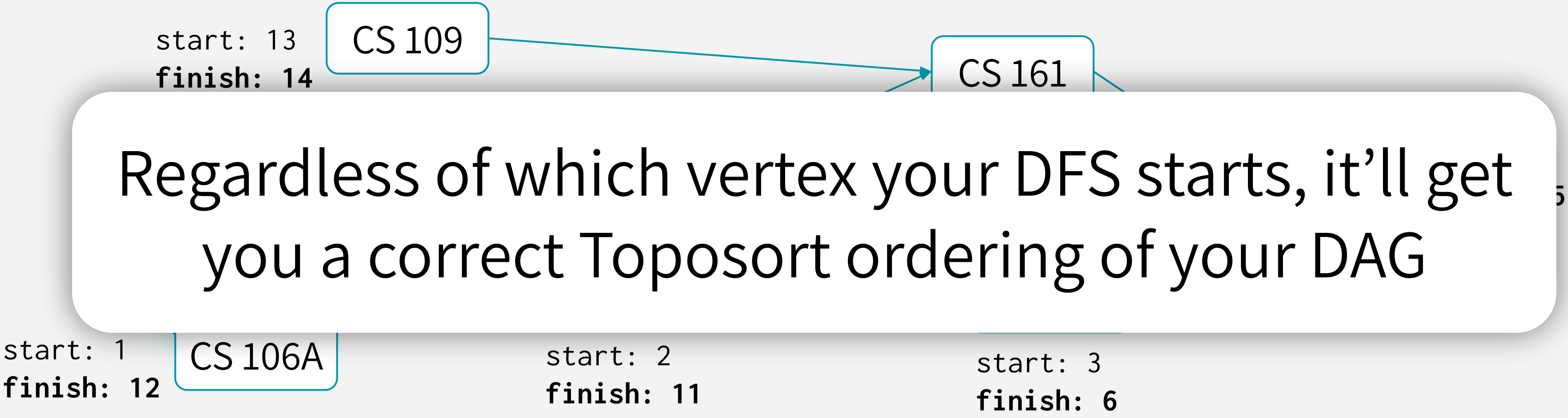
TOPOSORT: Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



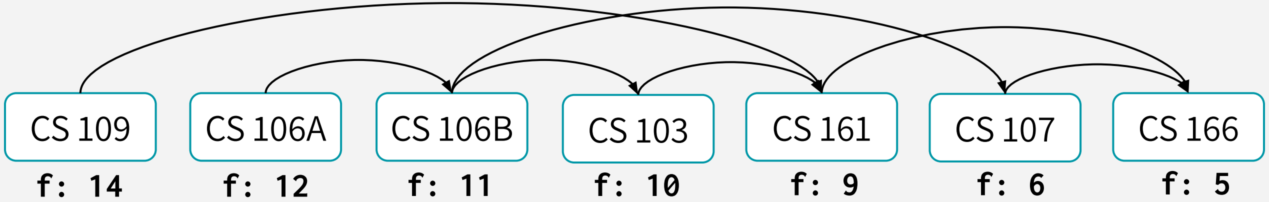
This is a valid toposort ordering!

DFS WILL GET US A TOPOSORT

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



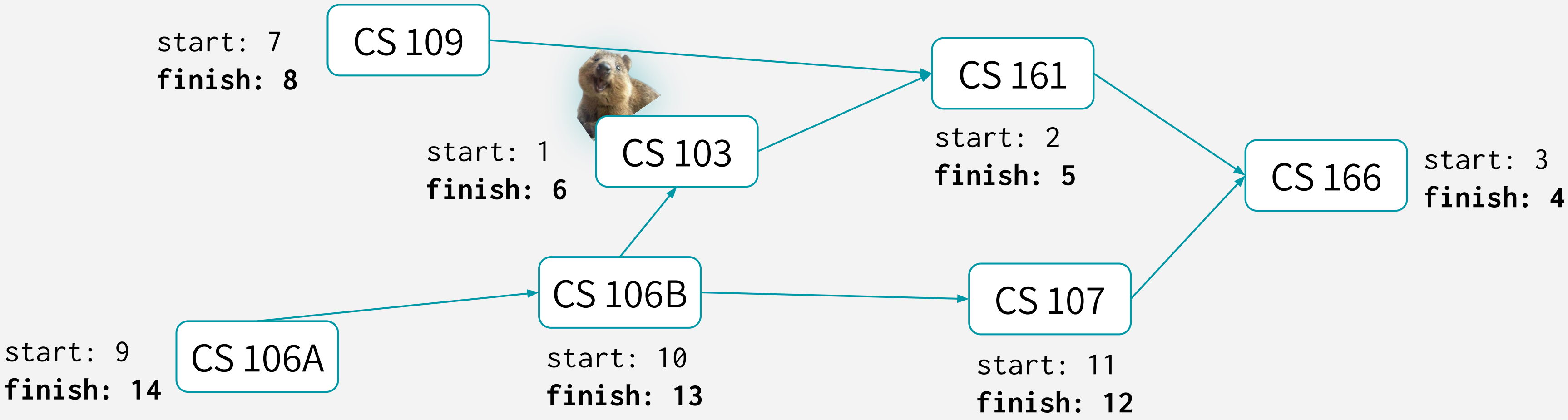
Regardless of which vertex your DFS starts, it'll get you a correct Toposort ordering of your DAG



DFS WILL GET US A TOPOSORT

Different order of running

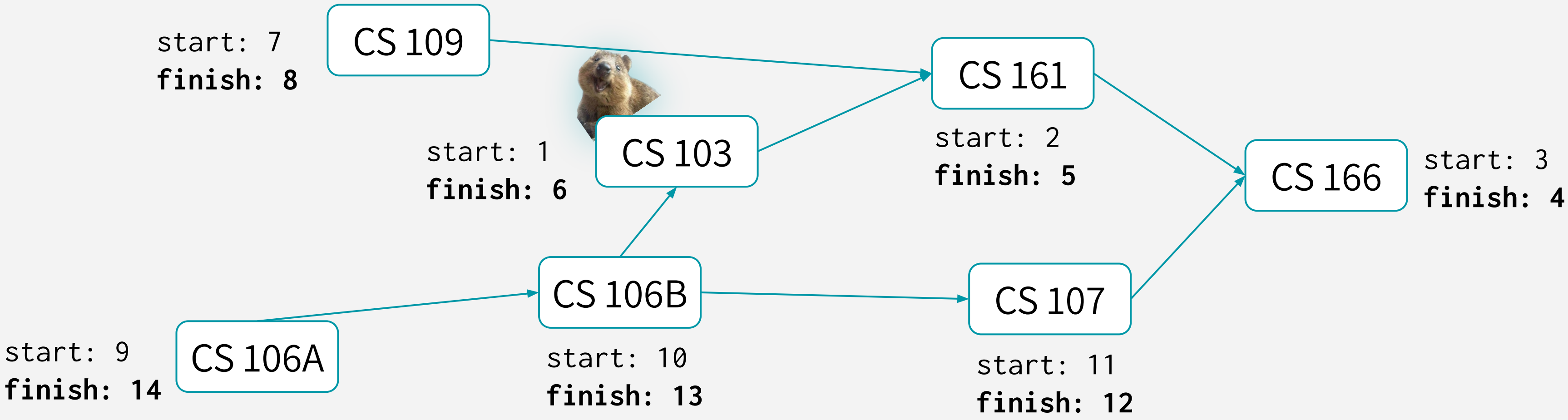
TOPOSORT: Perform DFS. When a vertex gets its finish time, insert it at the start of the list.



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.

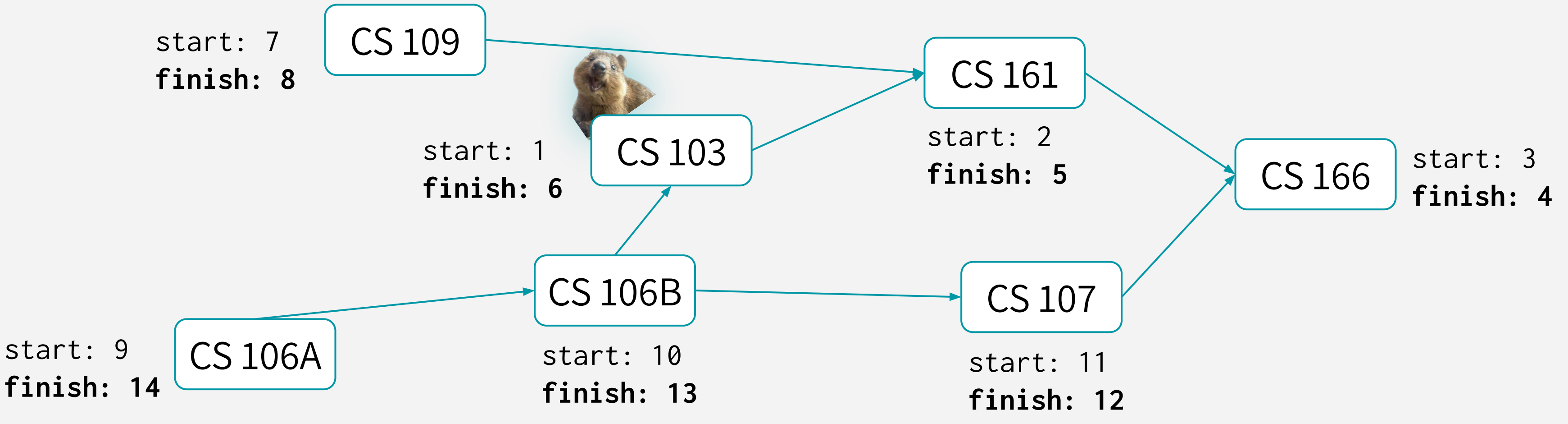


CS 166
f: 4

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.



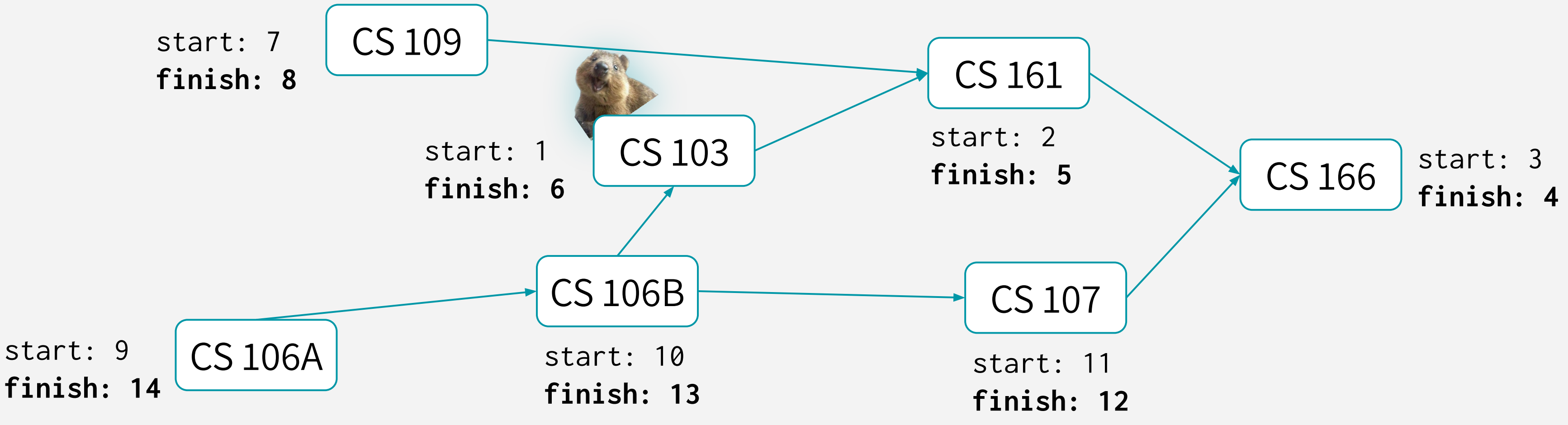
CS 161 f: 5

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.

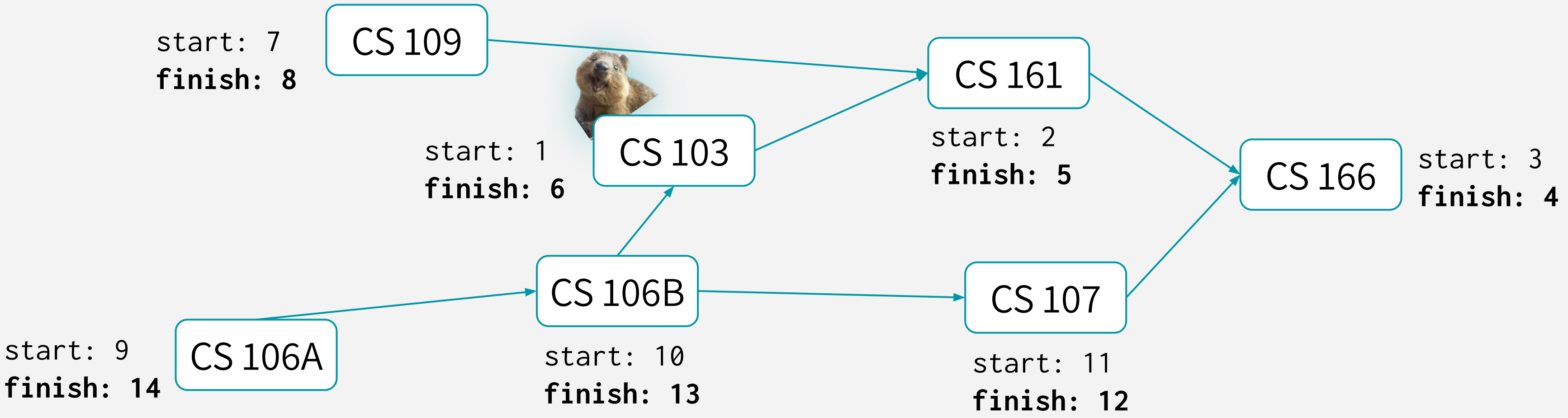


CS 103	CS 161	CS 166
f: 6	f: 5	f: 4

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.

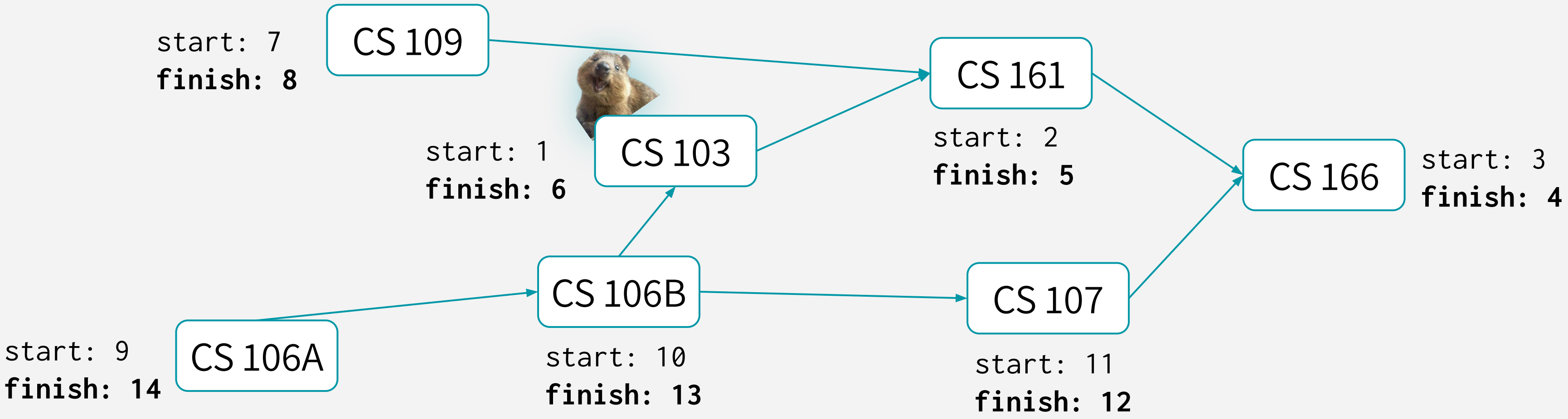


CS 109	CS 103	CS 161	CS 166
f: 8	**f: 6**	**f: 5**	**f: 4**

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.

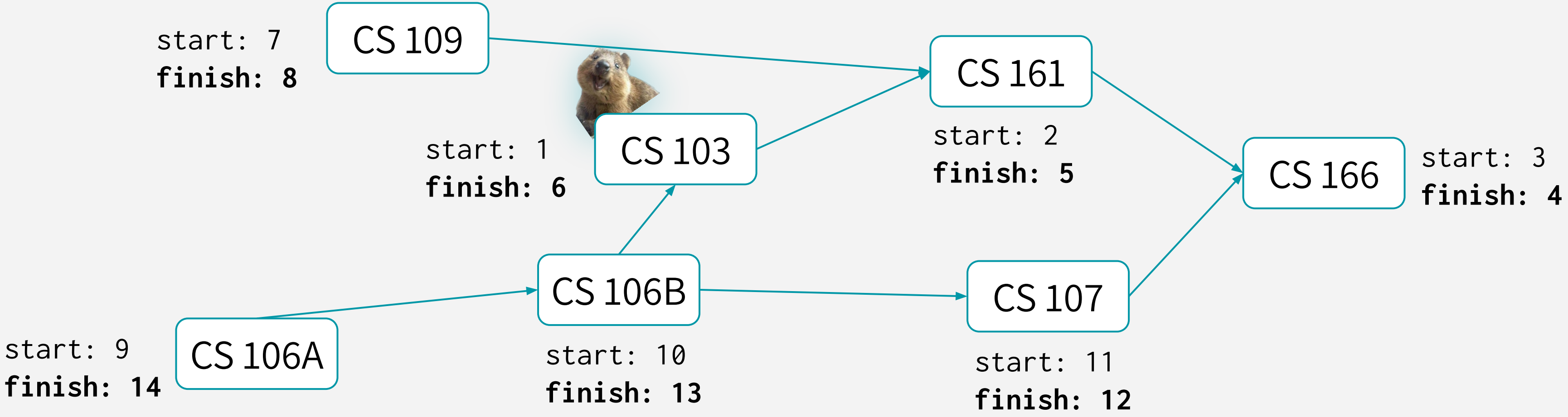


CS 107	CS 109	CS 103	CS 161	CS 166
f: 12	f: 8	f: 6	f: 5	f: 4

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.

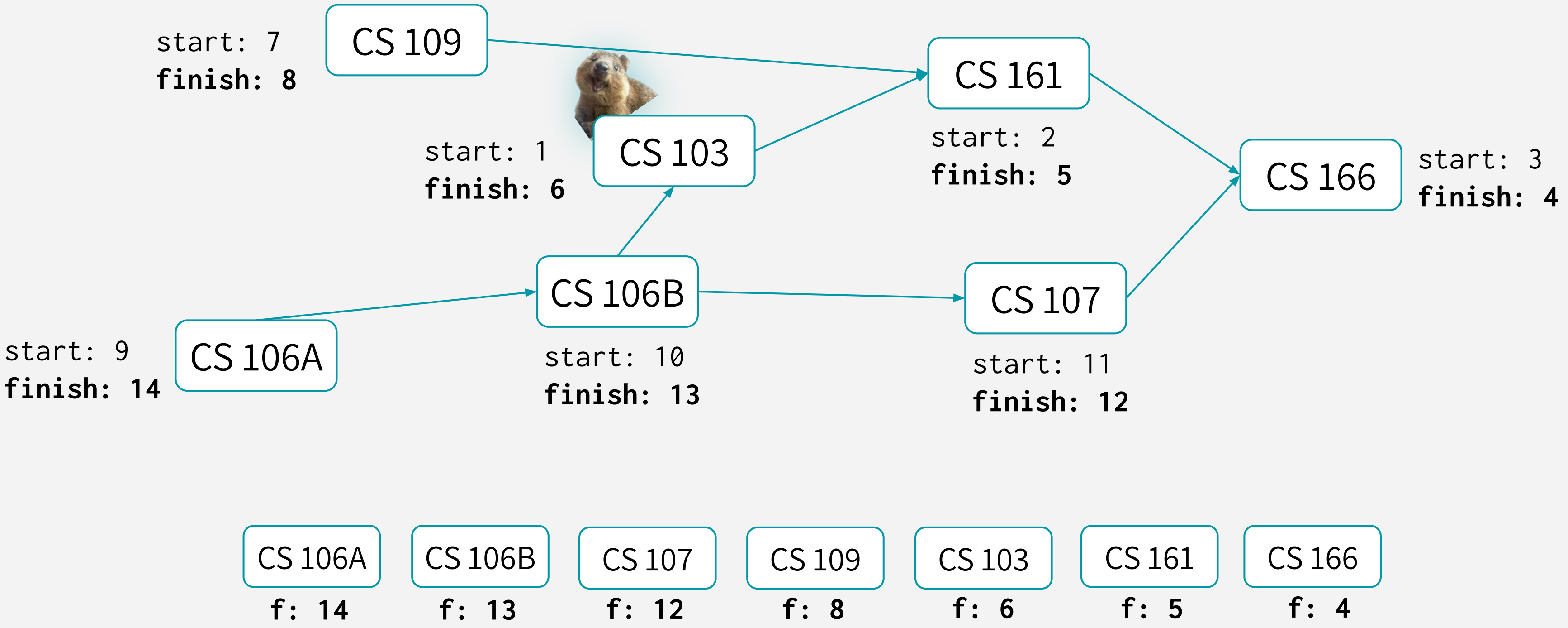


CS 106B	CS 107	CS 109	CS 103	CS 161	CS 166
f: 13	f: 12	f: 8	f: 6	f: 5	f: 4

DFS WILL GET US A TOPOSORT

Different order of running

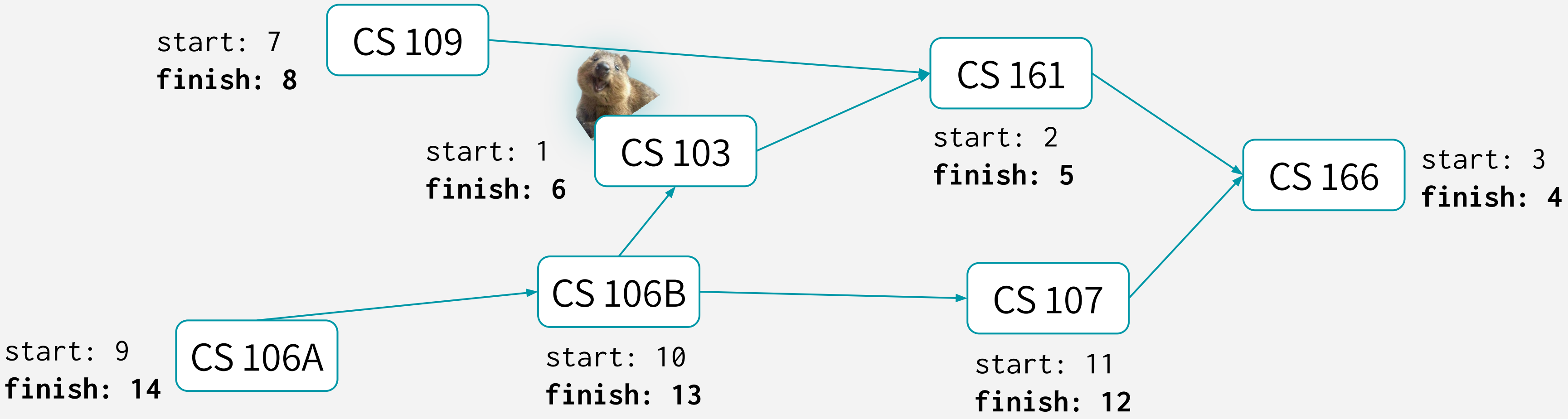
TOPOSORT: Perform DFS. When a vertex gets its finish time, insert it at the start of the list.

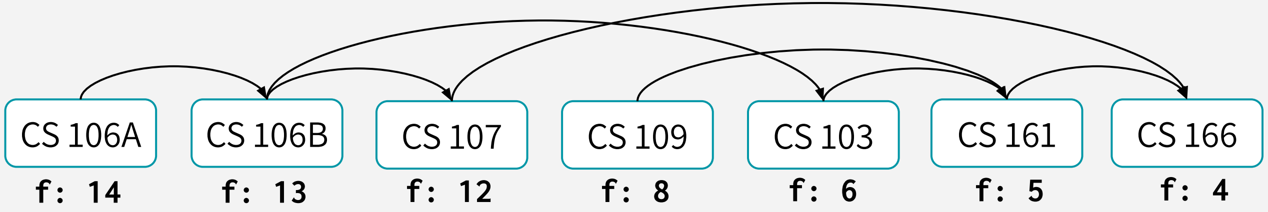


DFS WILL GET US A TOPOSORT

Different order of running

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

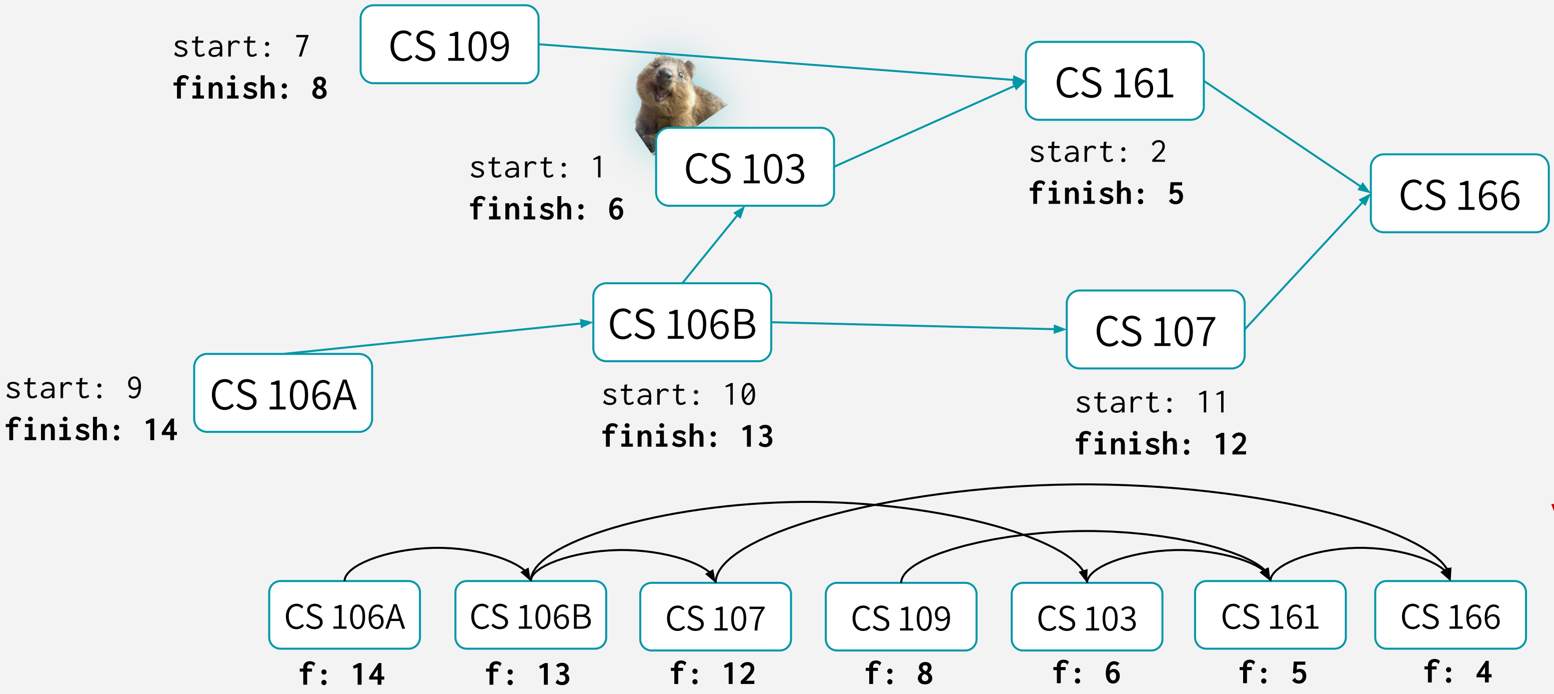




DFS WILL GET US A TOPOSORT

Different order of running

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



This is also a valid toposort ordering!

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؟