Lecture 38: Topological Sort

December 21, 2021

THE GREEDY PARADIGM

Traversing a graph consists of visiting each vertex only one time.

Depth First
Breadth First

DEPTH FIRST SEARCH

- Depth First Search developed by John Hopcraft & Robert Tarjan

Each vertex **V** is visited and then each unvisited vertex adjacent to **V** is visited.

If a vertex **V** has no adjacent vertices or all of its adjacent vertices have been visited, we backtrack to the predecessor of **V**.

The traversal is finished if this visiting and backtracking process leads to the first vertex where the traversal started.

DEPTH FIRST SEARCH

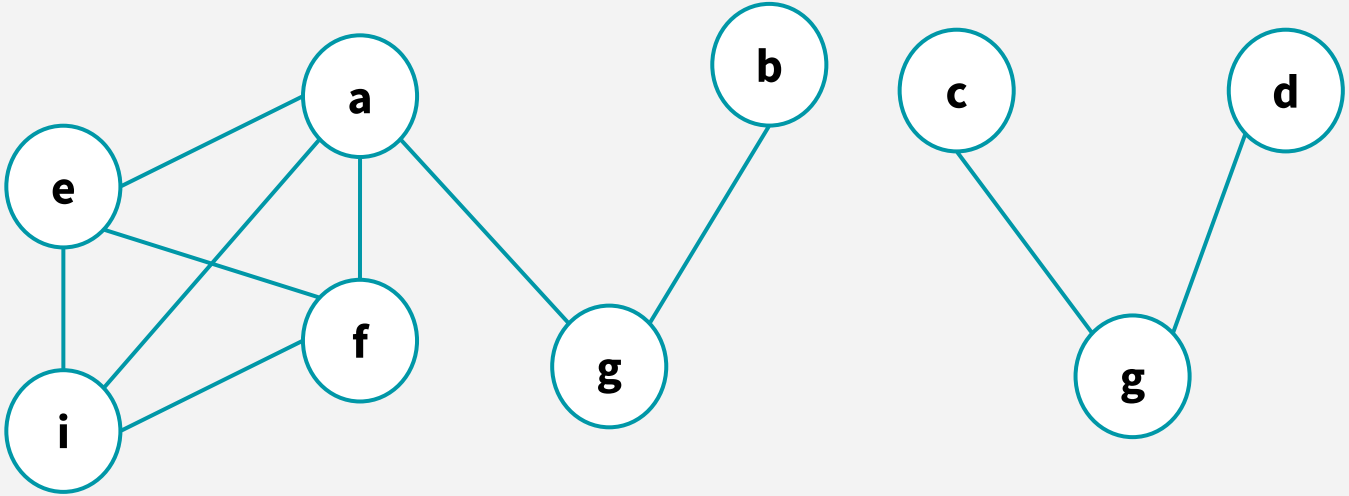
- The traversal is finished if this visiting and backtracking process leads to the first vertex where the traversal started.
- If there are still some unvisited vertices in the graph, the traversal continues restarting for one of the unvisited vertices.
- Although it is not necessary for the proper outcome of this method, the algorithm assigns a unique number to each accessed vertex so that vertices are now re-numbered.

DEPTH FIRST SEARCH

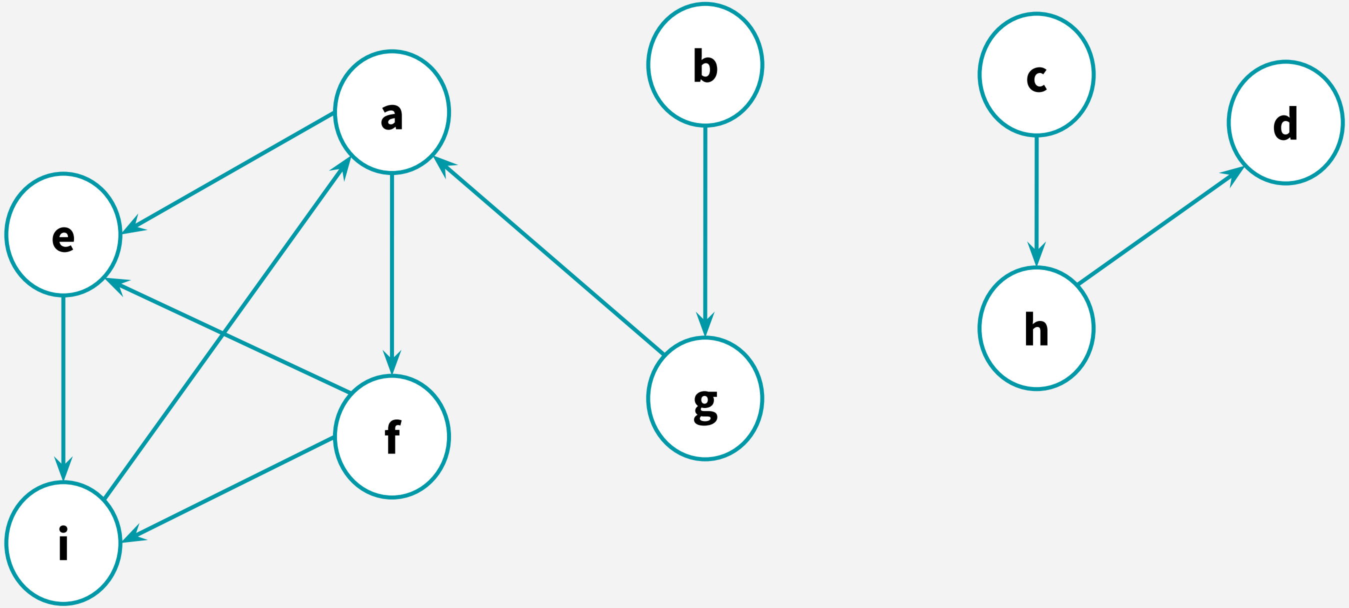
```
depthFirstSearch ( )
        for all vertices v
                num ( v ) = 0;
        edges = null;
        i = 1;
        while there is a vertex v such that
num ( v ) is 0
                DFS ( v )
        output edges;
```

```
DFS( v )
        num ( v ) = i++;
        for all vertices u adjacent to v
                if num ( u ) is 0
                        attach edge ( uv ) to edges;
                        DFS ( u );
```

DFS UNDIRECTED GRAPH EXAMPLE



DFS DIRECTED GRAPH EXAMPLE



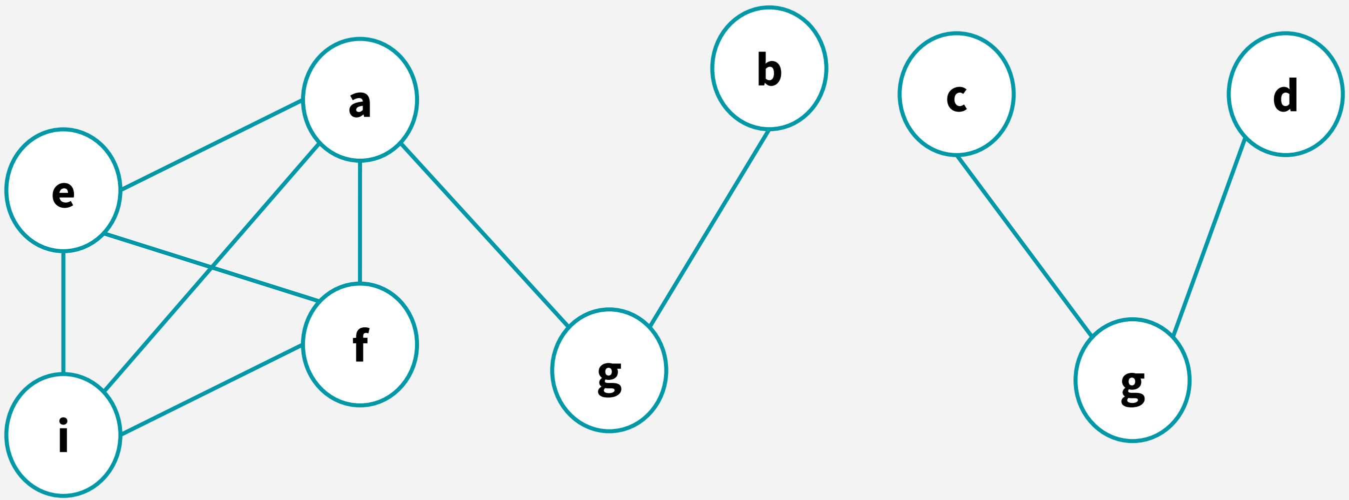
DEPTH FIRST SEARCH

- The complexity of Depth First Search is $O(|V| + |E|)$
  - Initializing num(v) for each vertex v requires $|V|$ steps
  - DFS(v) is called deg(v) times for each v—that is, once for each edge of v, hence, the total number of calls is $2|E|$.
- Worst case reaches $\mathbf{O(|V|^2)}$

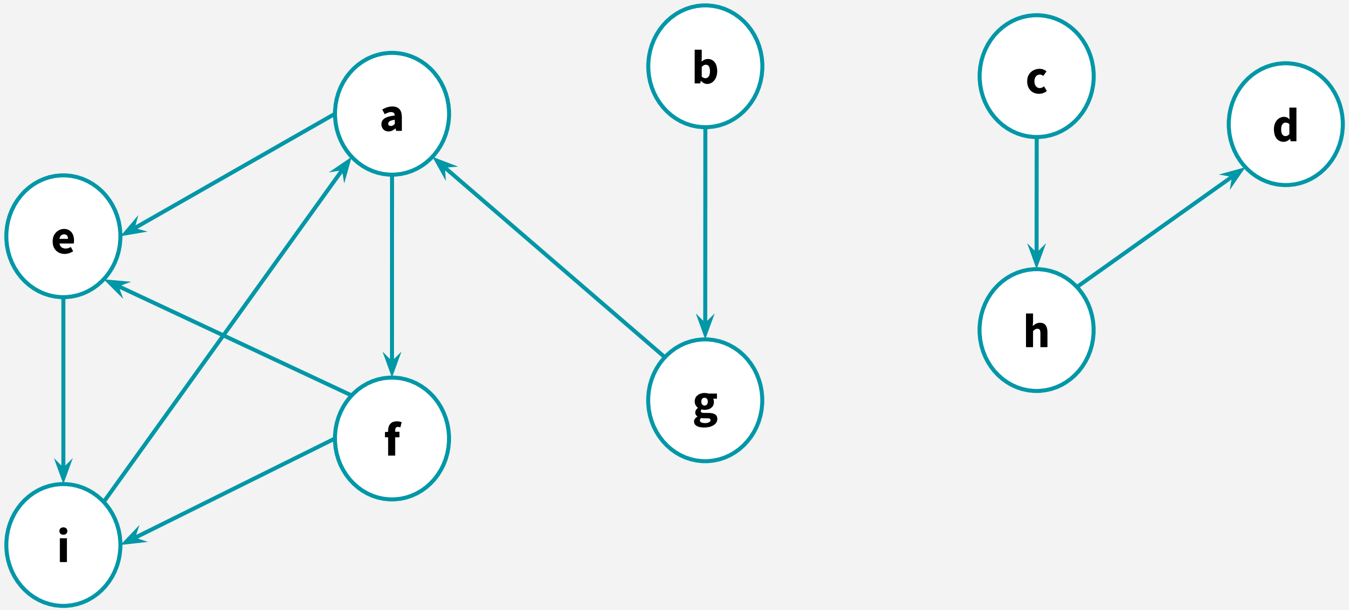
BREADTH FIRST SEARCH

```
breadthFirstSearch ( )
    for all vertices u
        num ( u ) = 0;
    edges = null;
    i = 1;
    while there is a vertex v such that num ( v ) is 0
        num ( v ) = i++;
        enqueue ( v );
        while queue is not empty
            v = dequeue ( );
            for all vertices u adjacent to v
                if num ( u ) is 0
                    num ( u )  = i++;
                    enqueue ( u ) ;
                    attach edge ( vu ) to edges
    output edges;
```

BFS UNDIRECTED GRAPH EXAMPLE



BFS DIRECTED GRAPH EXAMPLE



CYCLE DETECTION UNDIRECTED GRAPH

```
cycleDetectionDFS(v)
    num(v) = i++;
    for all vertices u adjacent to v
        if num(u) is 0
            pred(u) = v;
            cycleDetectionDFS(u);
        else if edge(vu) is not in edges
            pred(u) = v;
            cycle detected;
```

An edge (a back edge) indicates a cycle if it joins two vertices already included in the same spanning subtree.

CYCLE DETECTION DIRECTED GRAPH

```
digraphCycleDetectionDFS(v)
    num(v) = i++;
    for all vertices u adjacent to v
        if num(u) is 0
            pred(u) = v;
            digraphCycleDetectionDFS(u);
        else if num(u) is not ∞
            pred(u) = v;
            cycle detected;
    num(v) = ∞;
```

An edge (a back edge) indicates a cycle if it joins two vertices already included in the same spanning subtree.

TOPOLOGICAL SORT

An application of DFS that deals with “dependency” constraints

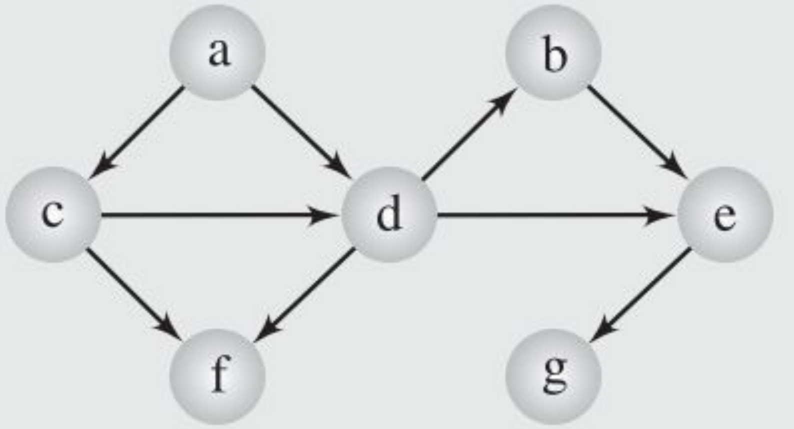
TOPOLOGICAL SORT

```
TS(v)
        num(v) = i++;
        for all vertices u adjacent to v
                if num(u) == 0
                        TS(u);
                else if TSNum(u) == 0
                        error;                  // a cycle detected
        TSNum(v) = j--;                         // after processing all successors of v,
                                                // assign to v a number smaller than
                                                // assigned to any of its successors;
```

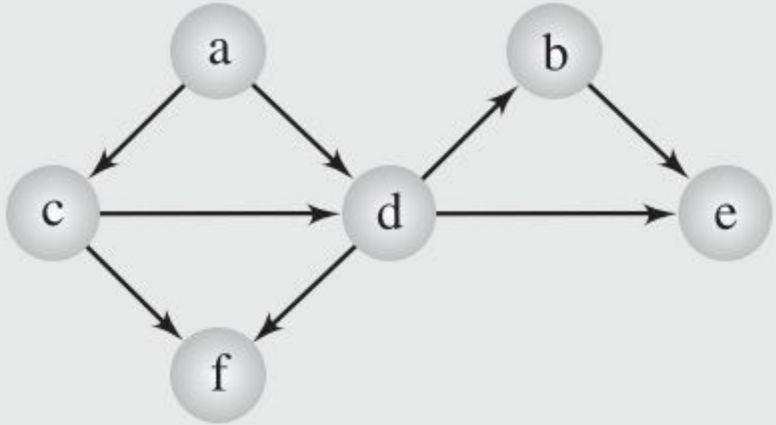
TOPOLOGICAL SORT

```
topologicalSorting(digraph)

    for all vertices v
        num(v) = TSNum(v) = 0;
    i = 1;
    j = |V|;

    while there is a vertex v such that num(v) == 0
        TS(v);

    output vertices according to their TSNum's;
```

TOPOLOGICAL SORT

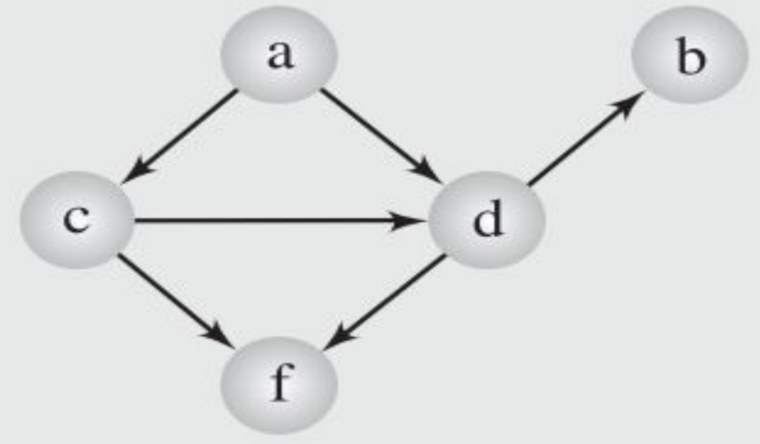


(a)

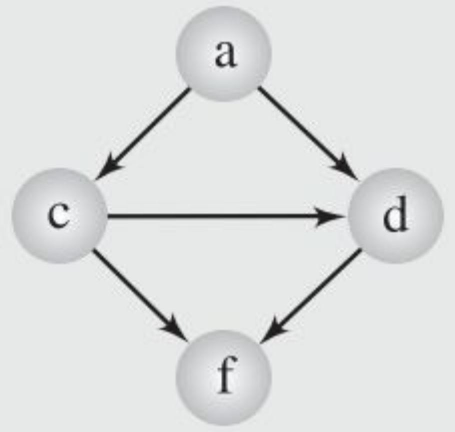


(b)

TOPOLOGICAL SORT

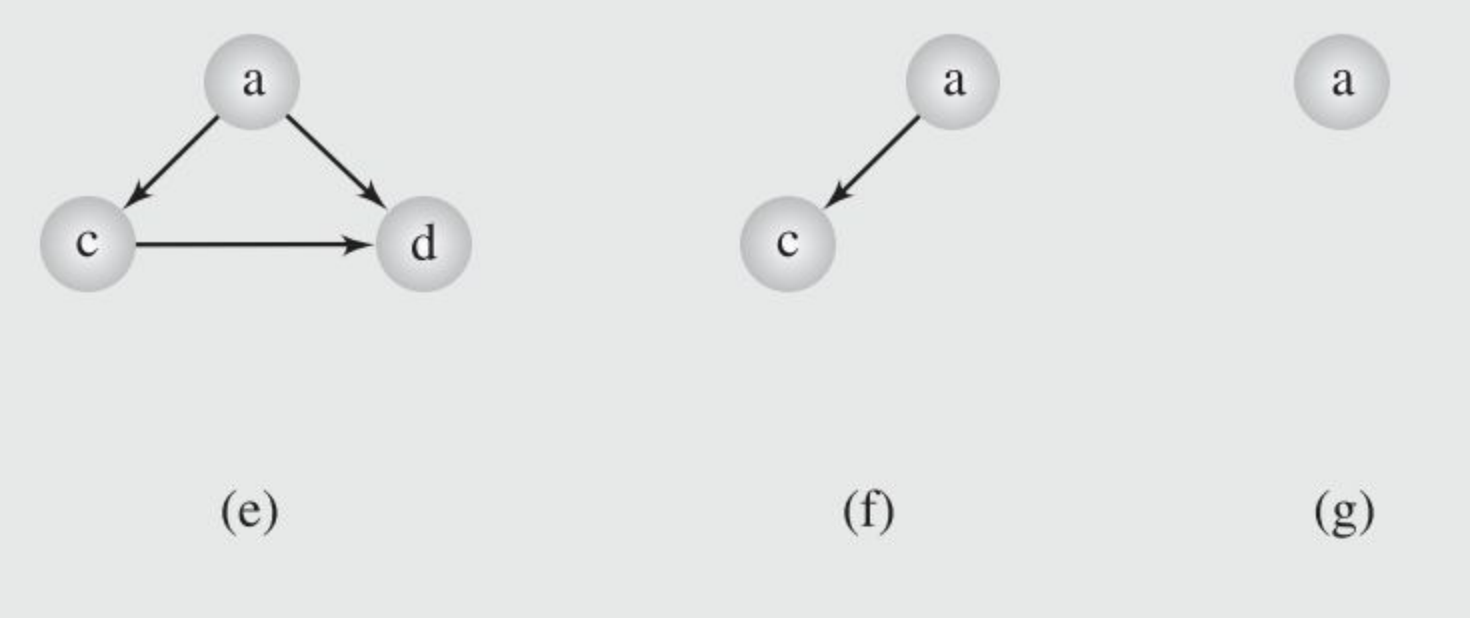


(c)



(d)

TOPOLOGICAL SORT



TOPOLOGICAL SORT

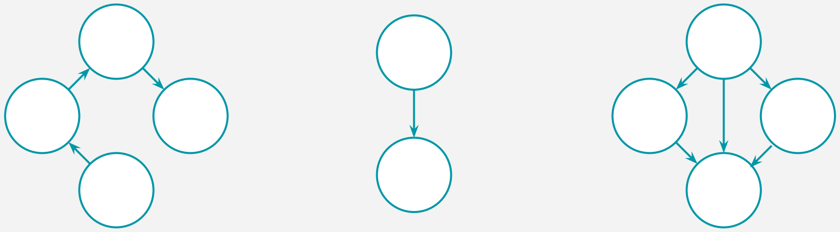
a	1													1
b				4					5					
c		2											2	
d			3									3		
e					5			6						
f										7	4			
g						6	7							

(h)

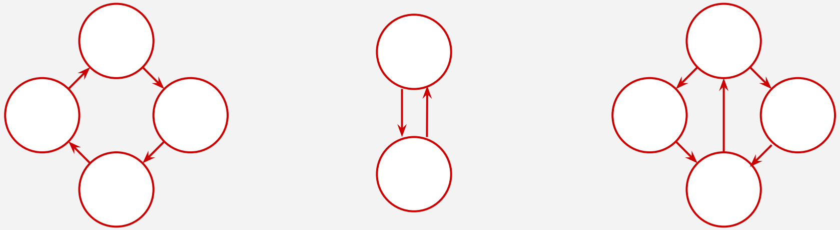
ASIDE: DIRECTED ACYCLIC GRAPHS

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

<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?

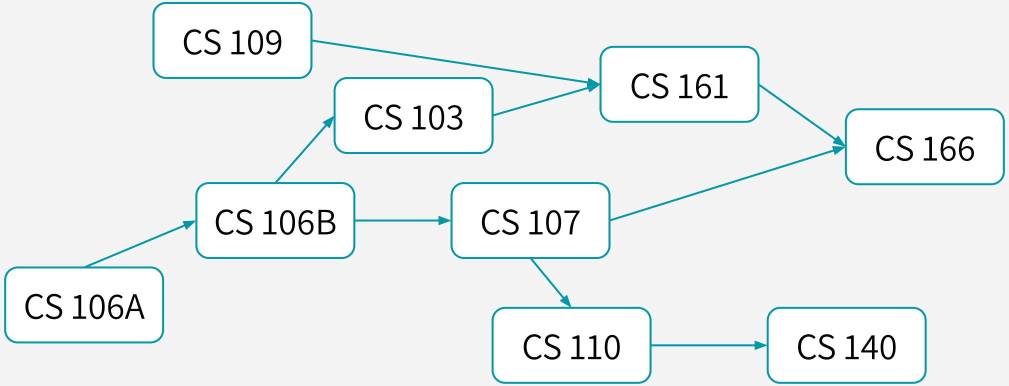
TOPOLOGICAL SORTING: THE TASK

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)



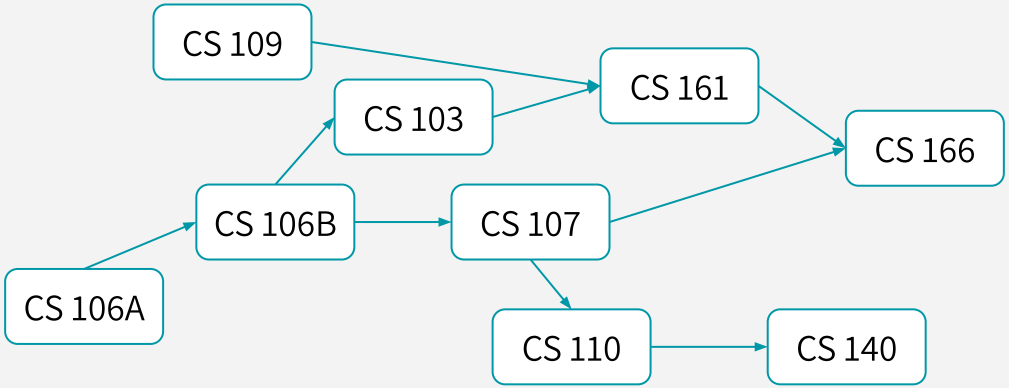
TOPOLOGICAL SORTING: THE TASK

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?

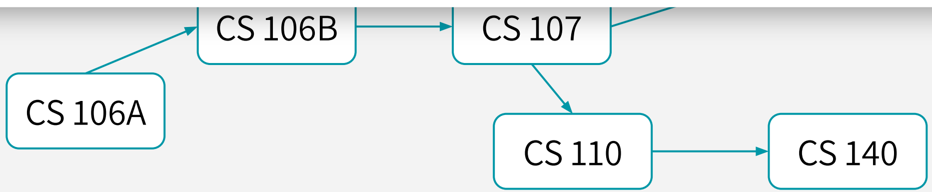
We want to produce an ordering such that:

for every edge **(v, w)** in E, **v** must appear before **w** in the ordering

(e.g. CS103 must come before CS161)

“d

(i.e. take class B after A)



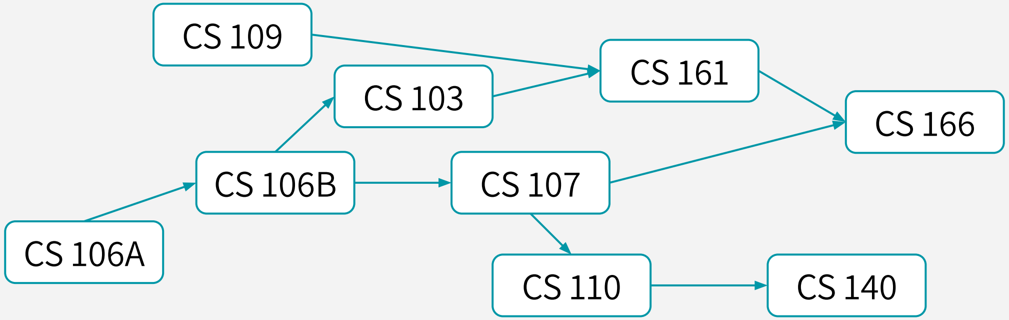
te

G!

(directed & acyclic)

TOPOLOGICAL SORTING: THE TASK

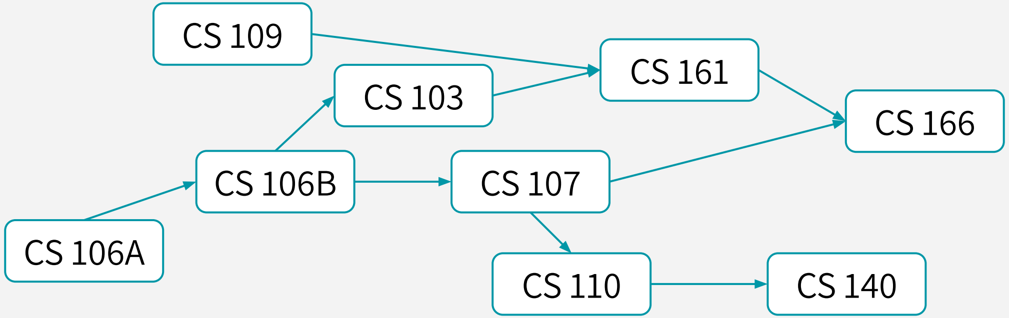
It's helpful to think of this as **"linearizing"** the graph, where all edges point to the **right**



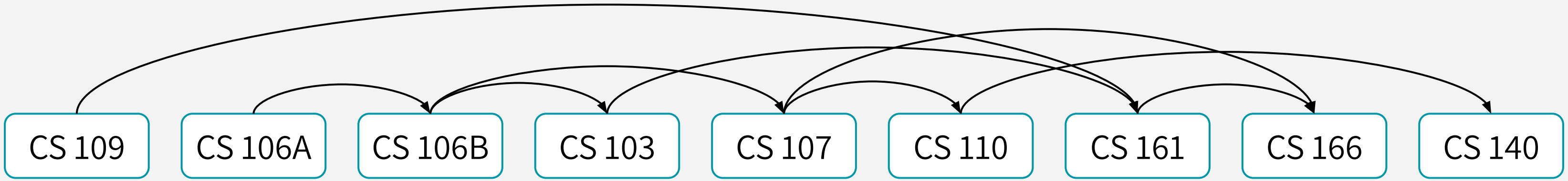
A correct "toposort" of this DAG:

TOPOLOGICAL SORTING: THE TASK

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

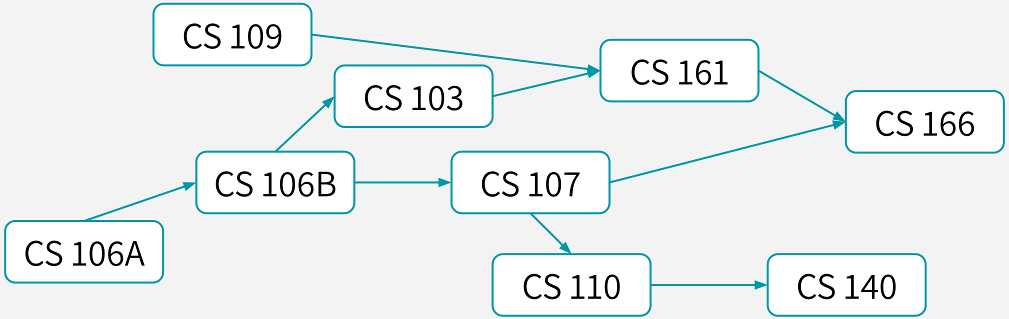


A correct "toposort" of this DAG:

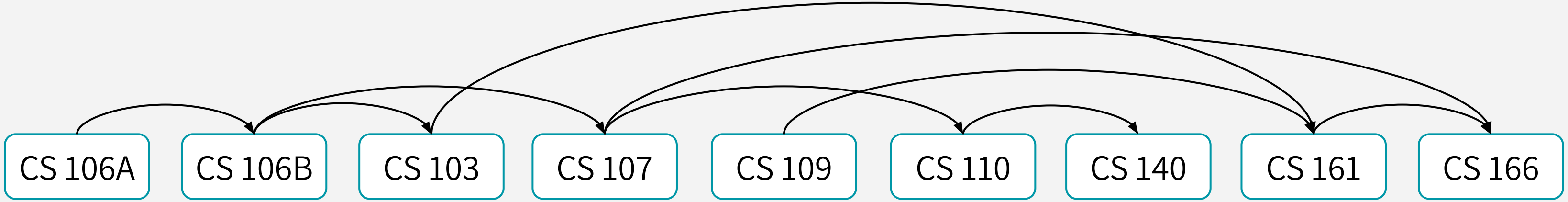


TOPOLOGICAL SORTING: THE TASK

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

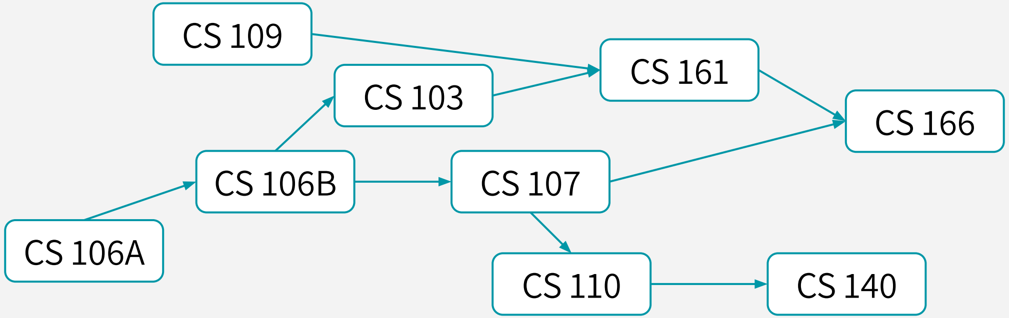


Also a correct toposort of this DAG:

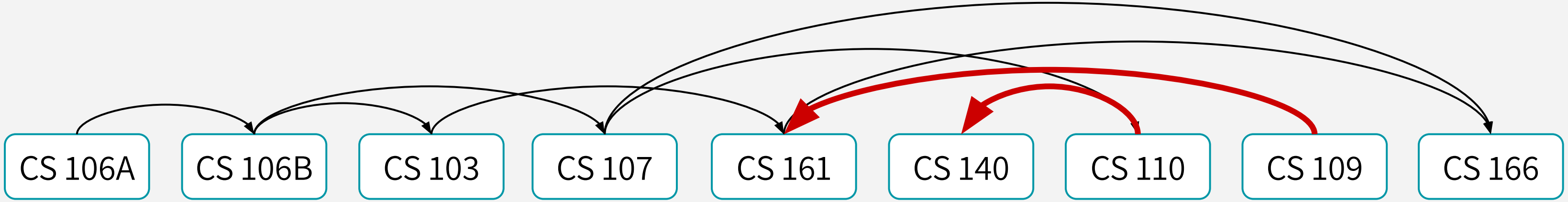


TOPOLOGICAL SORTING: THE TASK

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”?

TOPOSORT ON NON-DAGS?

We assume these “dependency” graphs are all DAGs!

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

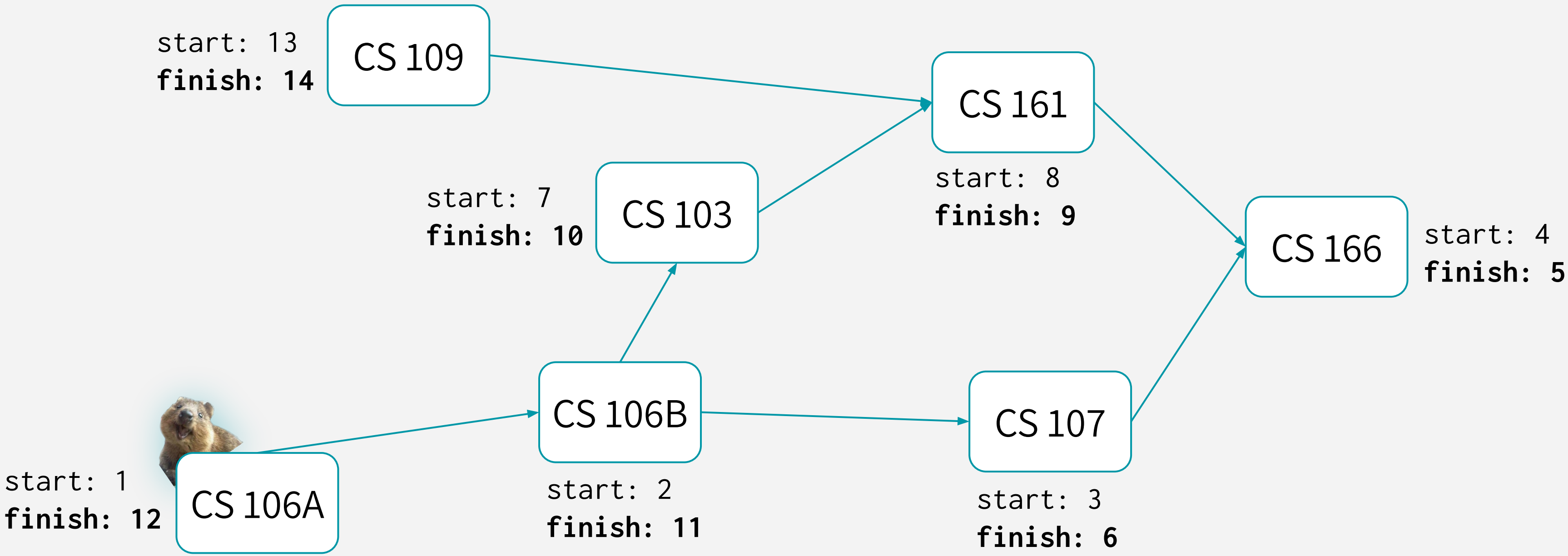
Intuitively, topological sort gives us some kind of priority ordering of the nodes. Performing toposort on a DAG of package installations tells us which packages should be installed first, and toposort on a DAG of course prerequisites could tell us which classes need to be taken first. It's hard to reason about any meaningful conclusions we could derive from a toposort ordering on an undirected or cyclical graph, and that's ultimately why we say toposort should be performed on a directed acyclic graph. Here's some more intuition that might help illuminate why toposort won't give us useful stuff for undirected or cyclic graphs.

In an undirected graph, an undirected edge between two nodes x and y doesn't say anything about which node is in a more privileged/prioritized position — the relationship between x and y is symmetric when an undirected edge connects them. You can't really argue for either node to have explicit priority over the other, so there's no clear reasoning for x to be come before y in a topological ordering, or vice versa.

In a graph with cycles, suppose we have two nodes x and y connected through some cycle. This means that x can reach y, and y can reach x. Some more ways to describe their relationship: "x leads to y" and "y leads to x", "x comes before y" and "y comes before x". Again, there's no clear reasoning for x to come before y in a topological ordering, since we have an equally valid case for y to come before x in the ordering as well.

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

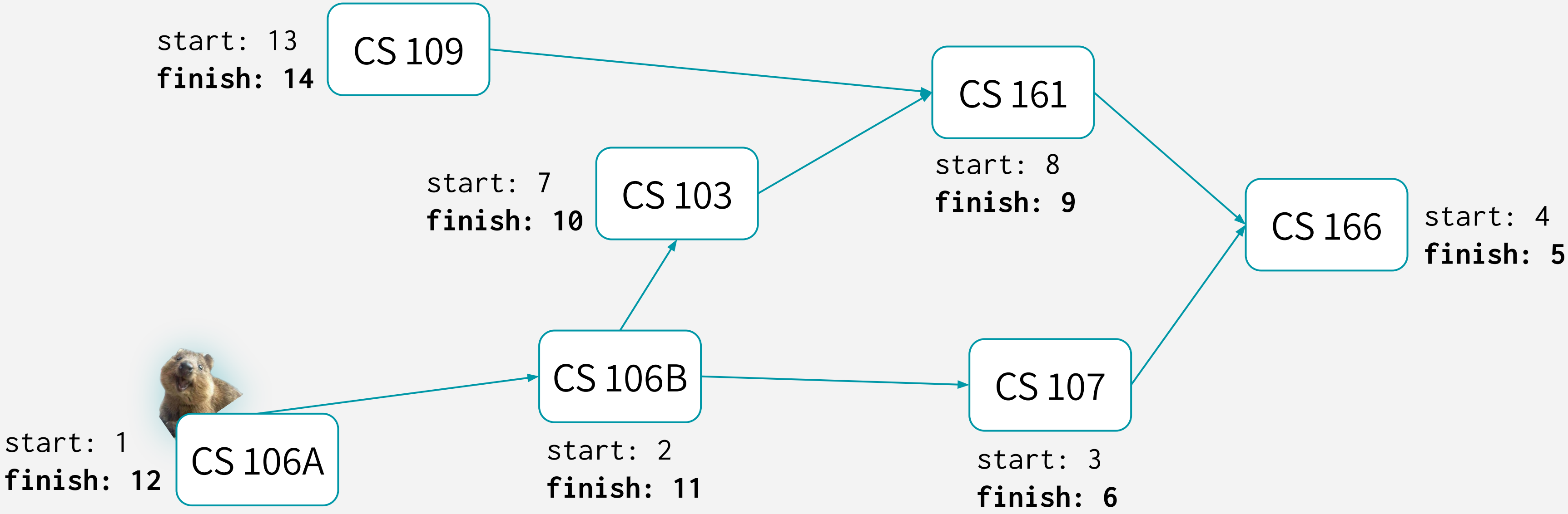
```
start: 2
finish: 11
```

```
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

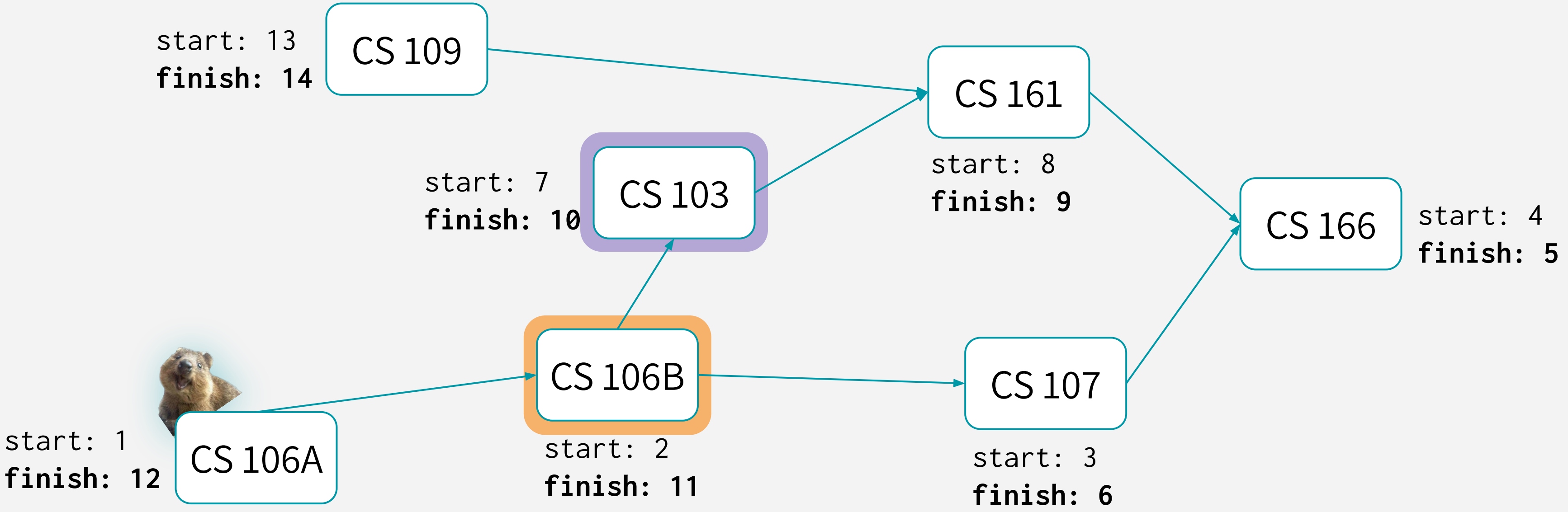
<u>CASE 1</u>: **v** → **w**, and **v is discovered first** by DFS



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

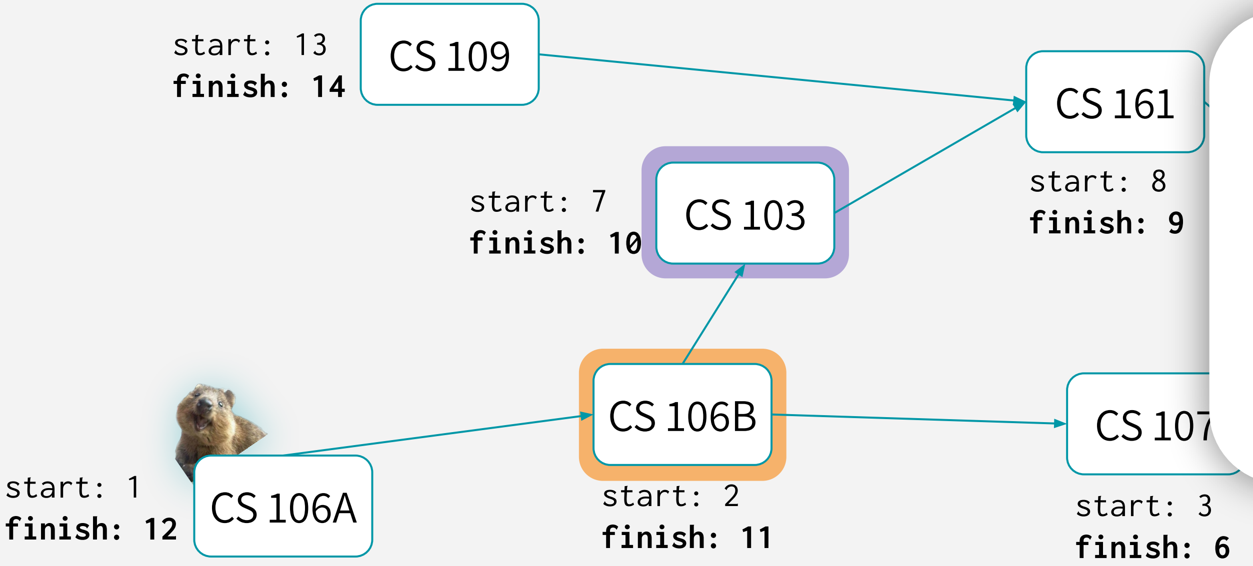
<u>CASE 1</u>: **v** → **w**, and **v is discovered first** by DFS



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

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

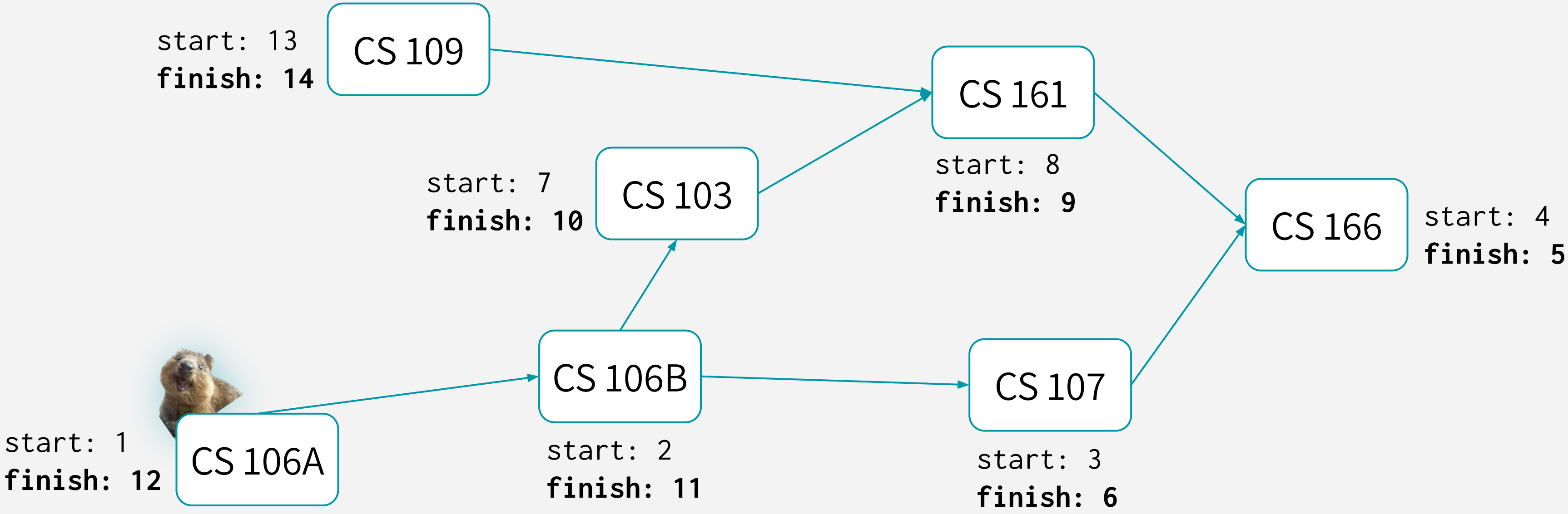


When **v** is discovered by DFS, this call will eventually discover **w** & recursively call DFS on **w**. Then, **w** will get its finish time before **v** gets its finish time, so **v.finish** > **w.finish**!

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

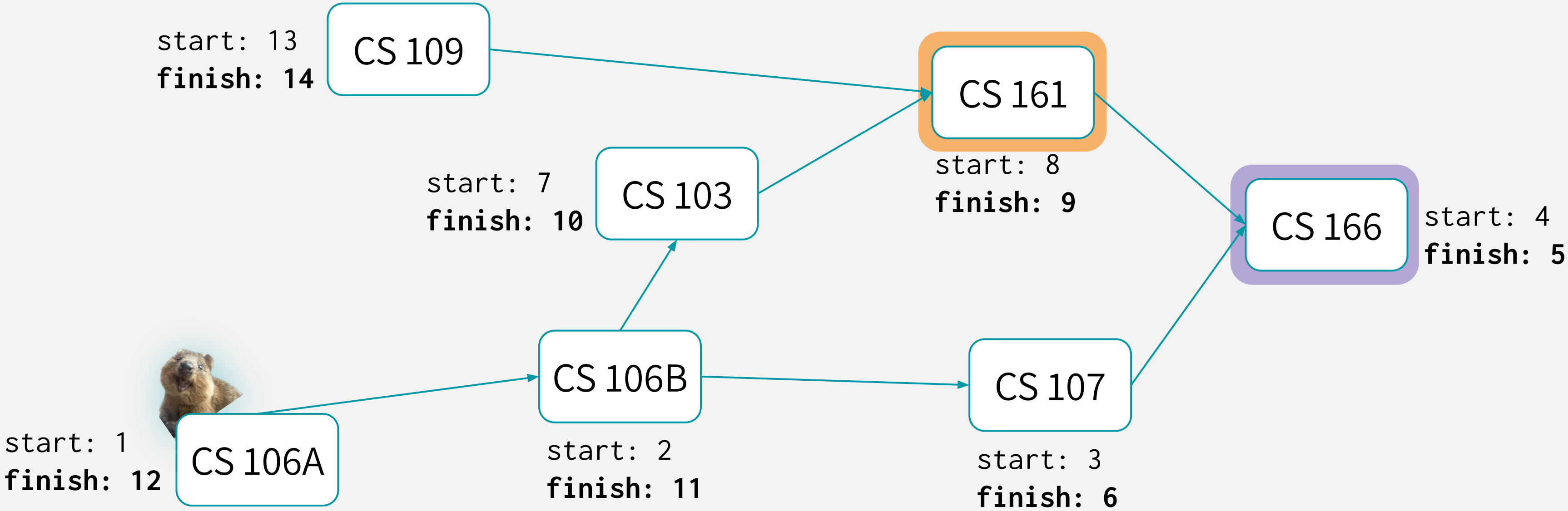
<u>CASE 2</u>: **v** → **w**, and **w** **is discovered first** by DFS



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

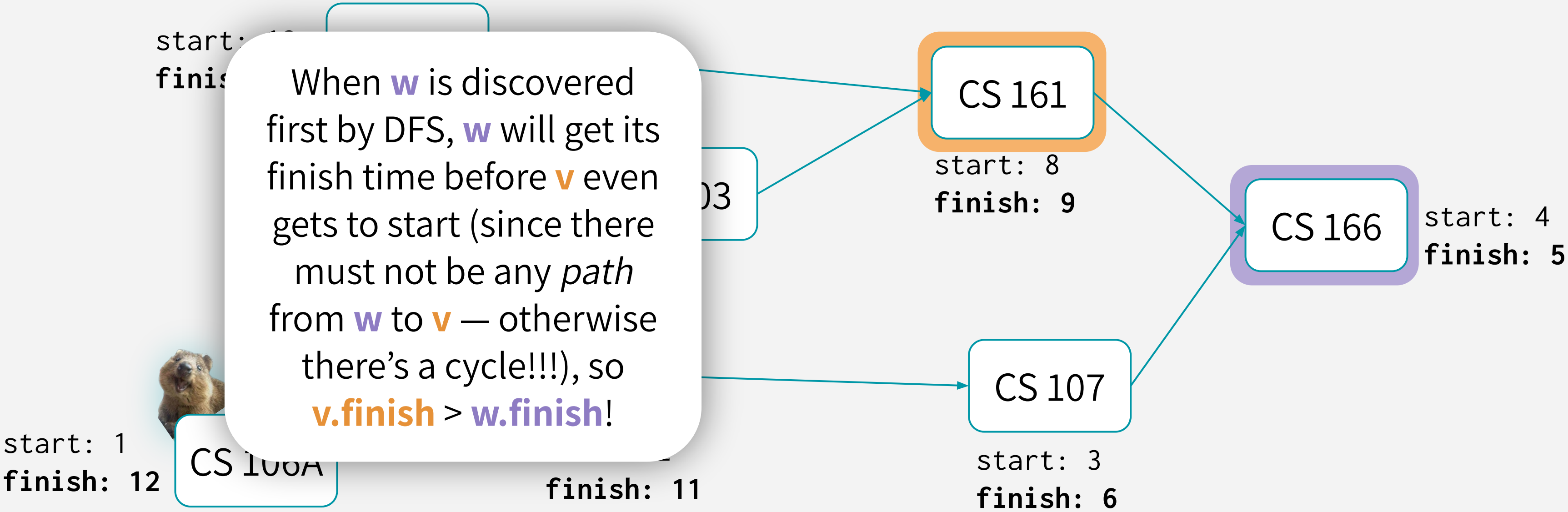
CASE 2: **v** → **w**, and **w is discovered first** by DFS



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

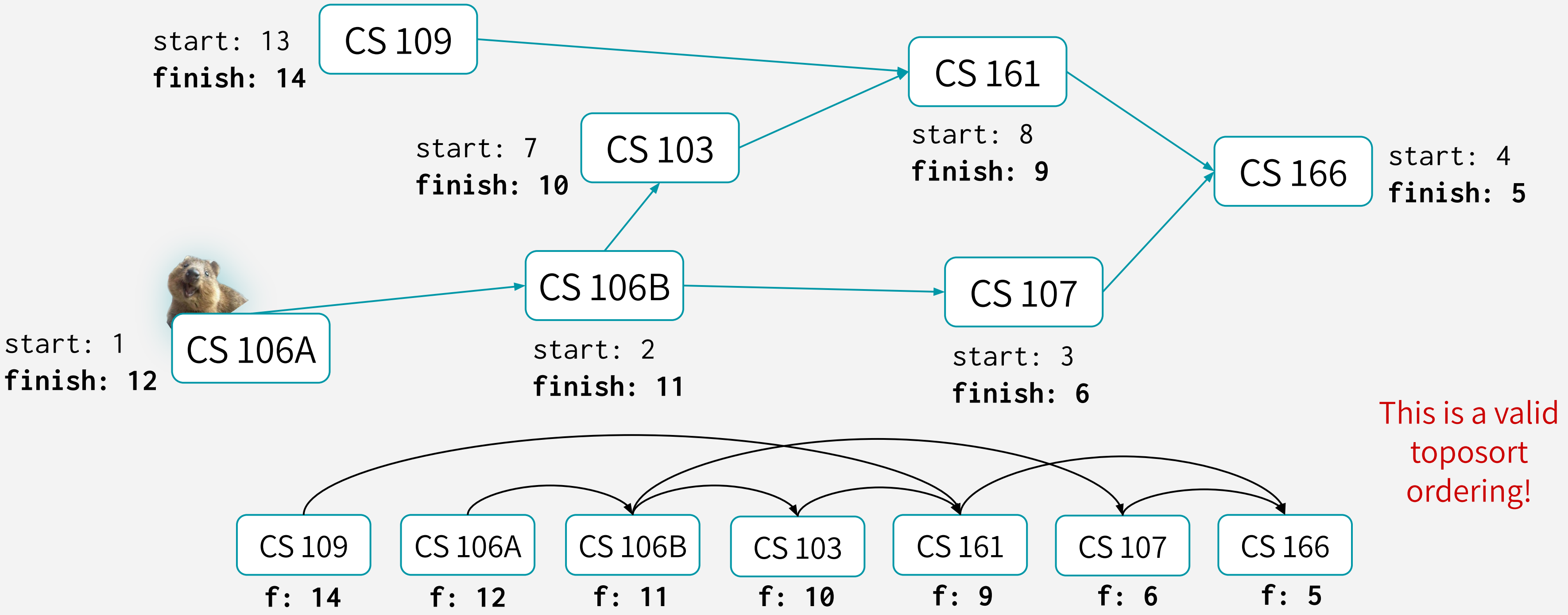
<u>CASE 2</u>: **v** → **w**, and **w is discovered first** by DFS



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**!

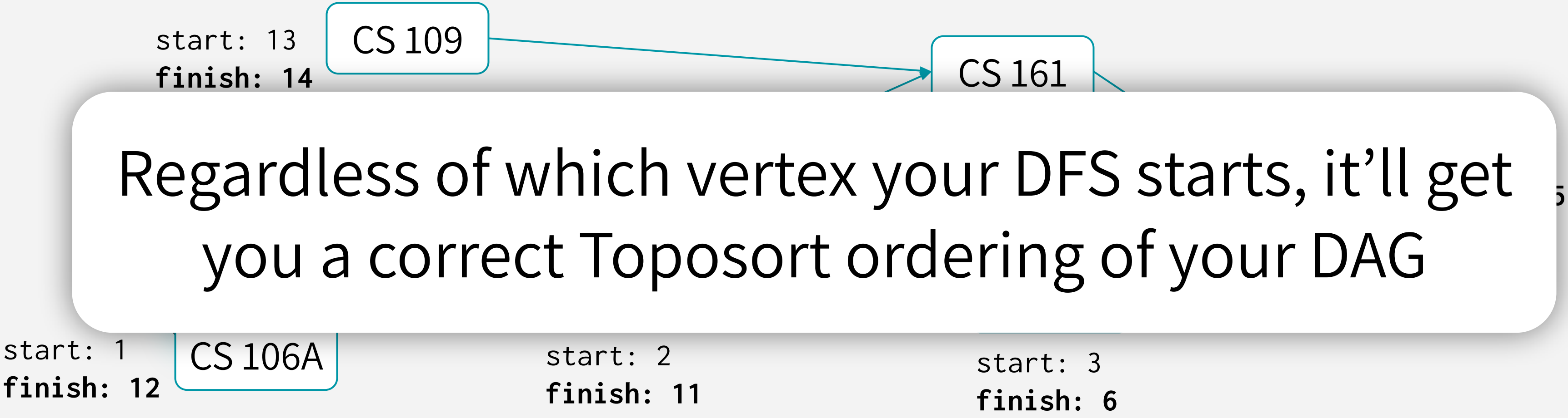
DFS WILL GET US A TOPOSORT

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

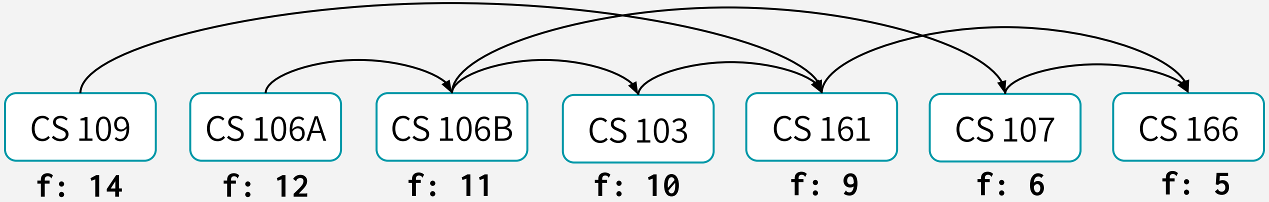


DFS WILL GET US A TOPOSORT

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

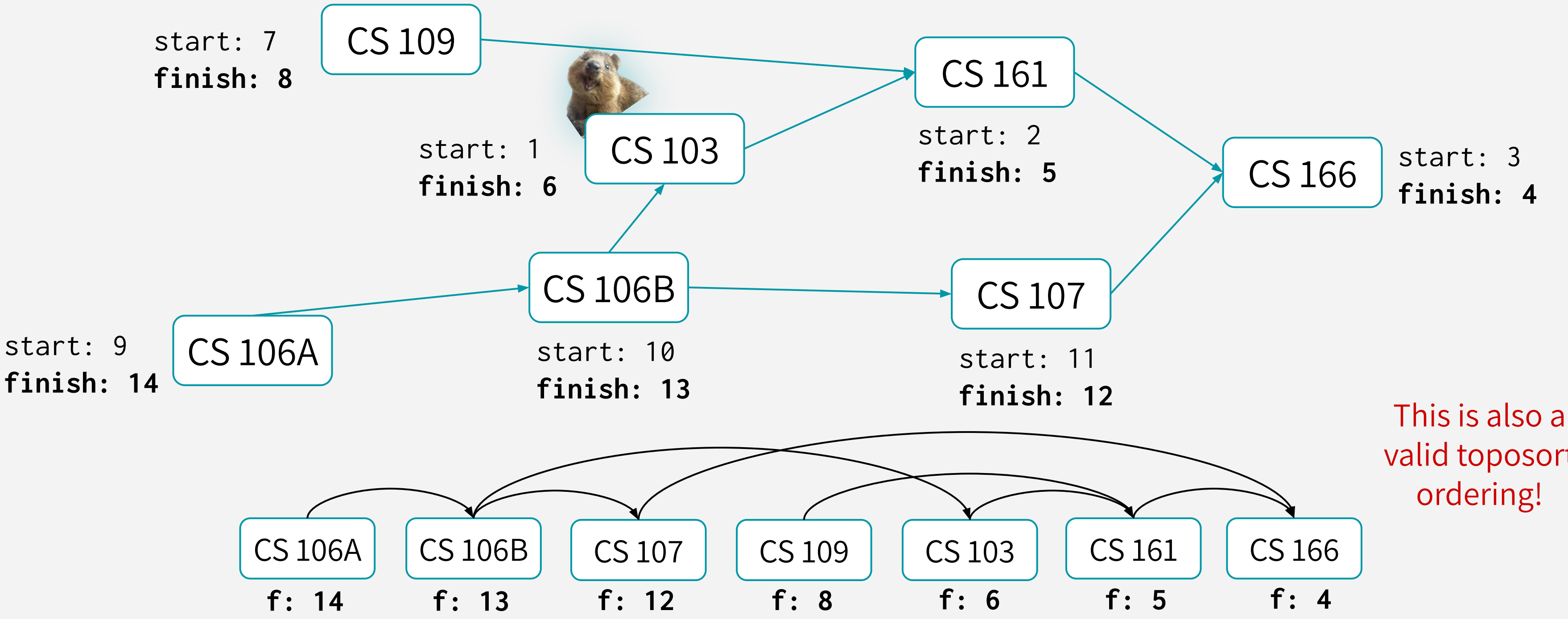


Regardless of which vertex your DFS starts, it'll get you a correct Toposort ordering of your DAG



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 also a valid toposort ordering!