

DFS and Graph Decompositions

CS/MCS 401: Computer Algorithms I

Lecture 10

Professor: Jonathan Liu

Introduce yourself!

Icebreaker: How has this semester compared to previous semesters for you?

Tuesday's Lecture

On Tuesday, we didn't finish developing the algorithm for detecting whether a graph is bipartite.

I've decided to take out one of the planned homework questions and replace it with a question finishing that proof.

Takeaway: With a tiny tweak to BFS, we can create an algorithm that determines whether a graph is bipartite!

Lesson Goals

Today we're going to look at the other major graph traversal algorithm, DFS, and see an application of it.

The Other Traversal Algorithm: DFS



Depth First Search

In short: Keep going “forward” until you run out of vertices to explore, backtrack only when you absolutely have to.

A visualizer:

<https://visualgo.net/en/dfsdfs>

BFS vs DFS

WhateverFirstSearch($G=(V,E)$, starting vertex s):

put s into the bag

while the bag is not empty:

take v from the bag

if v is unvisited:

mark v as visited

for each edge (v, w)

put w into the bag

BFS: The “bag” is a queue. First in, first out.

DFS: The “bag” is a stack. First in, last out.

The bag could be, for example, a priority queue!

Two Versions of DFS

DFS($G=(V,E)$, starting vertex s):

put s into the *stack*

while the *stack* is not empty:

take v from the top of the *stack*

if v is unvisited:

mark v as visited

for each edge (v, w)

put w on top of the *stack*

DFS-Recursive(starting vertex s):

mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

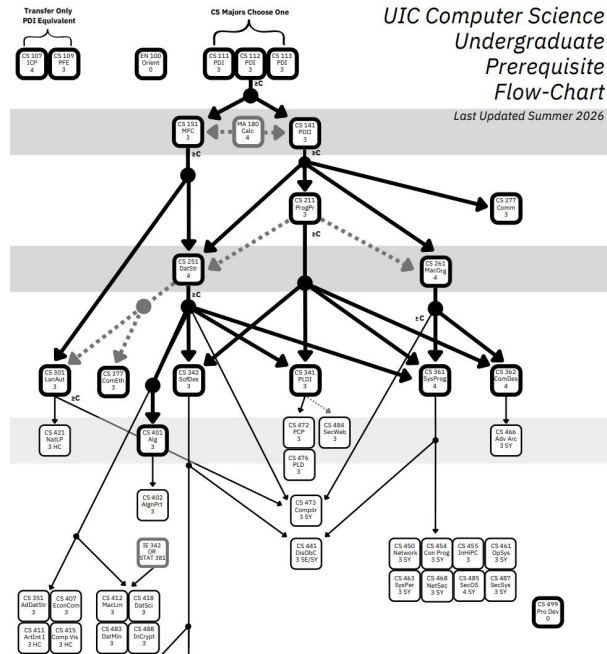
Directed Acyclic Graphs (DAGs)



Directed Acyclic Graphs (DAGs)

A Directed Acyclic Graph is a graph that is both directed and acyclic.

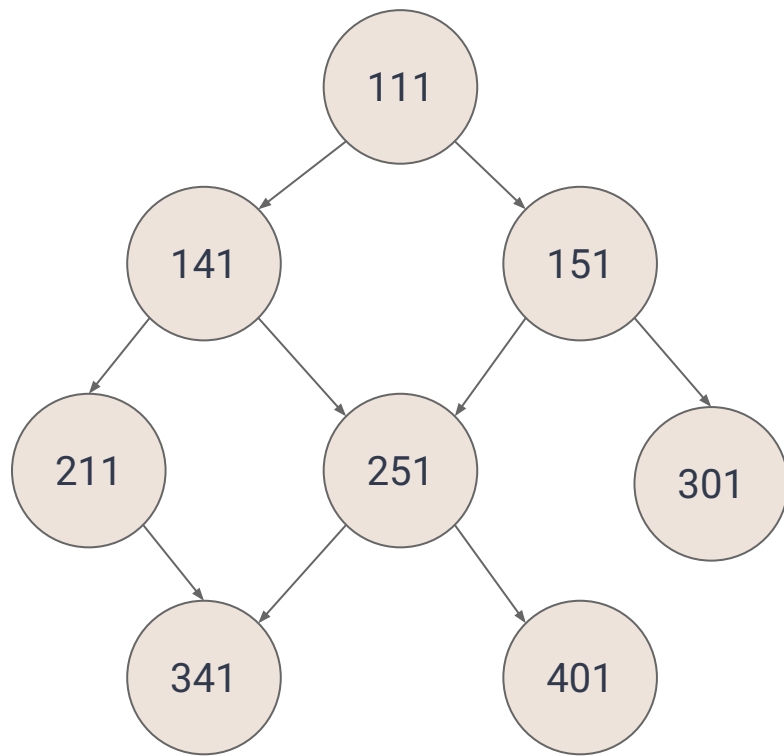
Many real-world graphs are, and must be, DAGs:



Topological Ordering

A **topological order** on a DAG is a list of all of its vertices in an order $v_1, v_2, \dots, v_n$ such that if there is an edge from v_i to v_j then $i < j$ in the list.

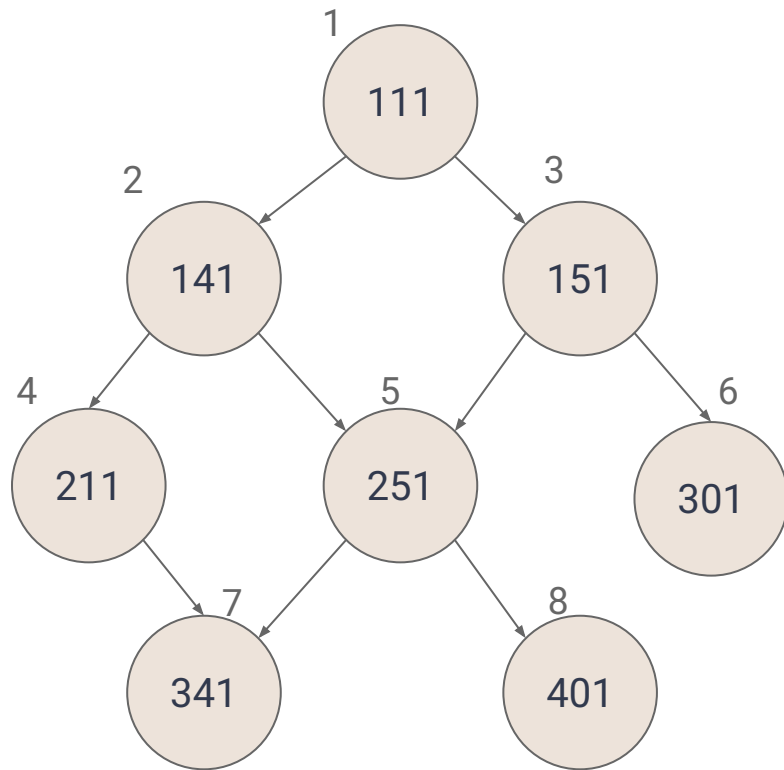
In a list of prerequisites, it's an order in which you can take all the classes such that no prerequisite is unfulfilled.



Topological Ordering

A **topological order** on a DAG is a list of all of its vertices in an order $v_1, v_2, \dots, v_n$ such that if there is an edge from v_i to v_j then $i < j$ in the list.

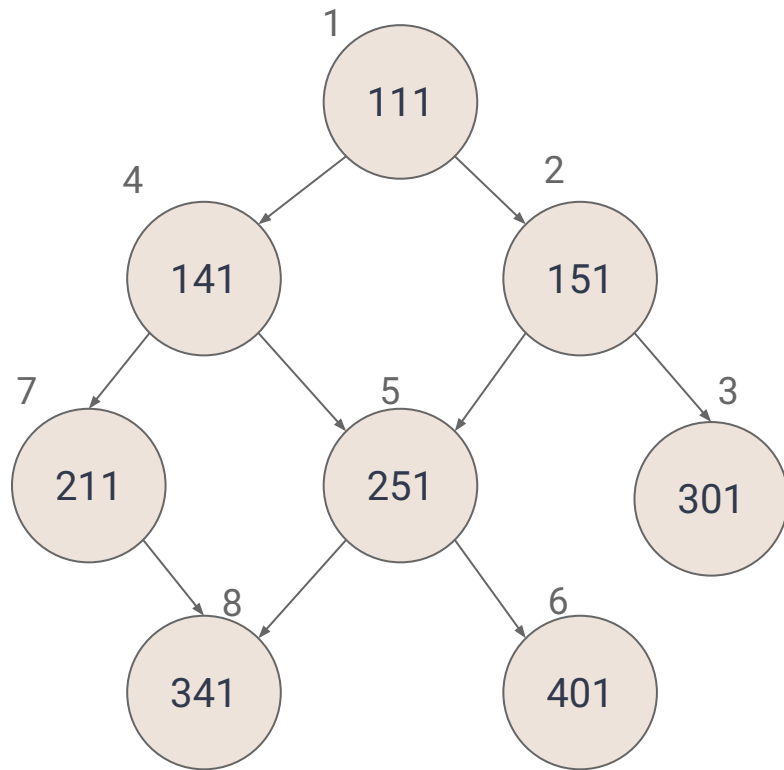
In a list of prerequisites, it's an order in which you can take all the classes such that no prerequisite is unfulfilled.



Topological Ordering

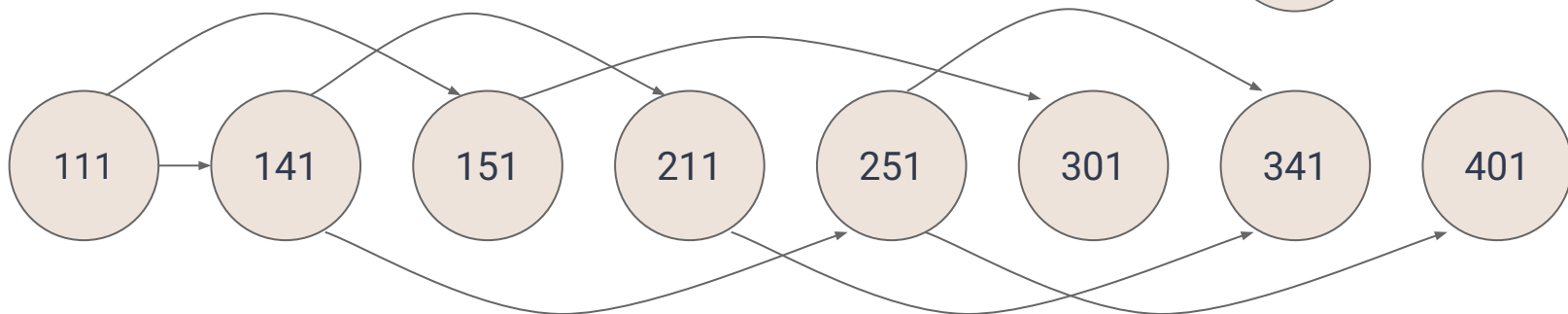
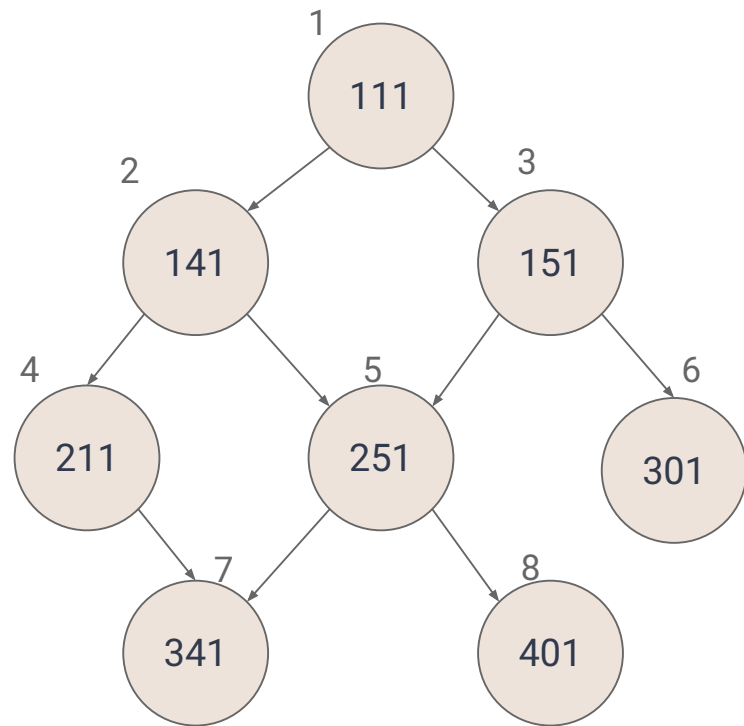
A **topological order** on a DAG is a list of all of its vertices in an order $v_1, v_2, \dots, v_n$ such that if there is an edge from v_i to v_j then $i < j$ in the list.

In a list of prerequisites, it's an order in which you can take all the classes such that no prerequisite is unfulfilled.

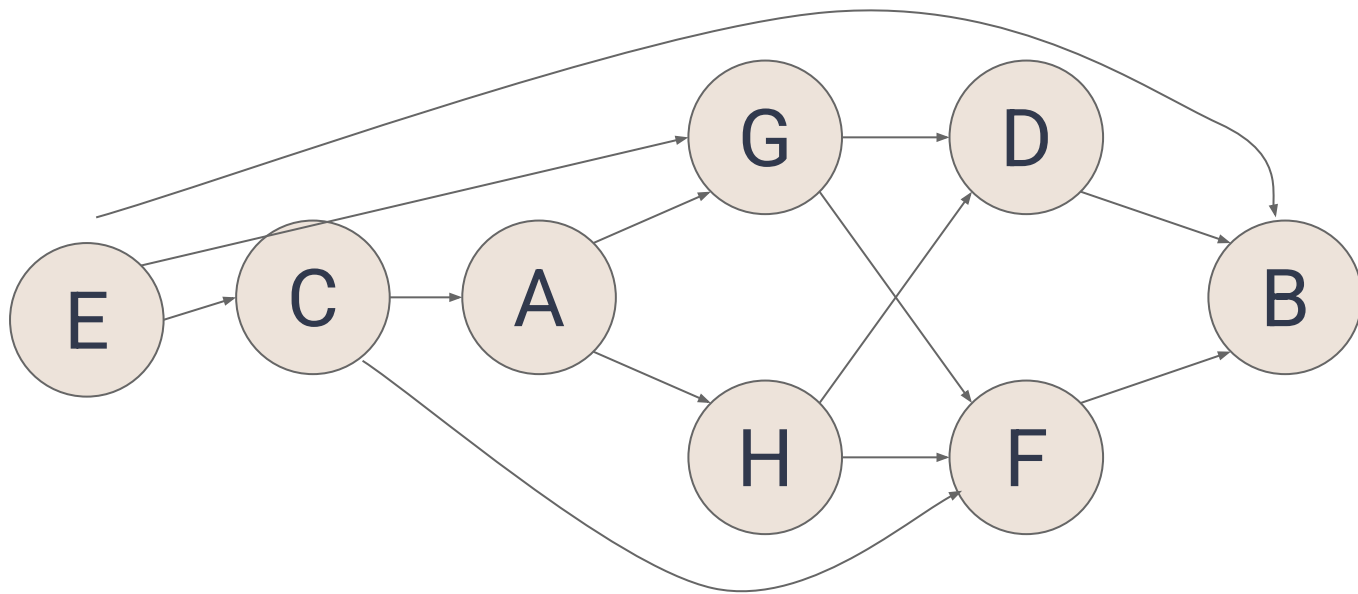


Topological Ordering

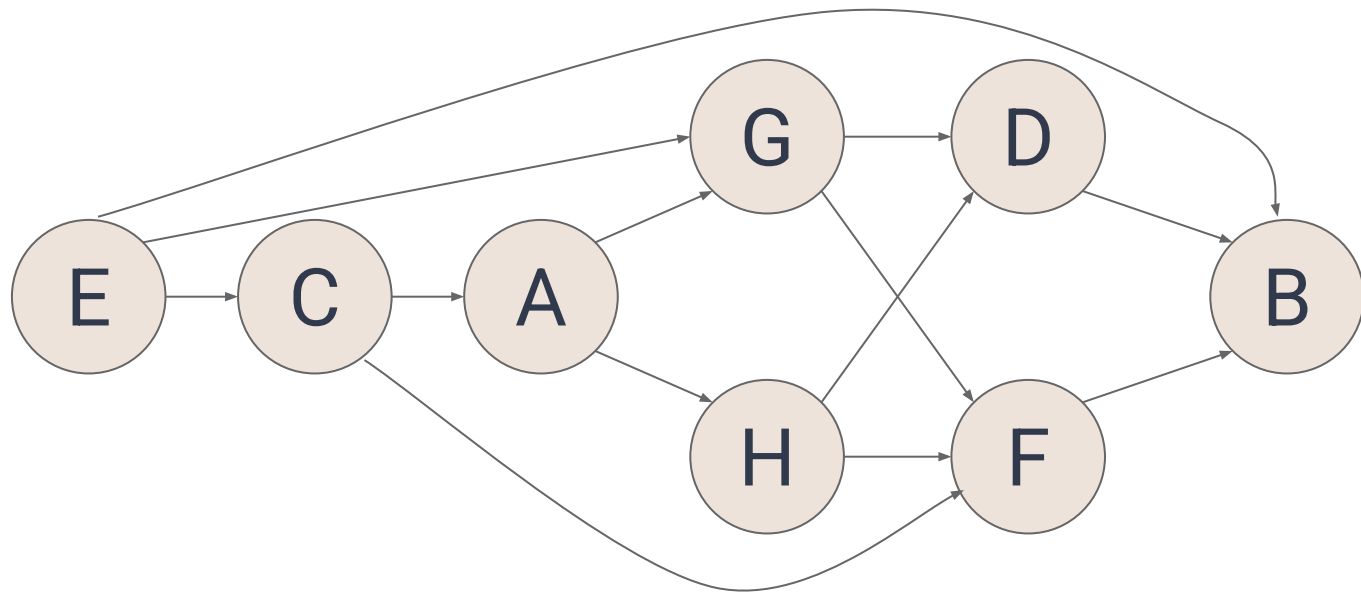
A topological ordering is often called a **linearization**: if you put them in a line in this order, all arrows point to the right.



Topological Ordering (Worksheet)



Topological Ordering (Worksheet)



Valid Orders

ECA GH DF B

ECA GH FD B

ECA HG DF B

ECA HG FD B

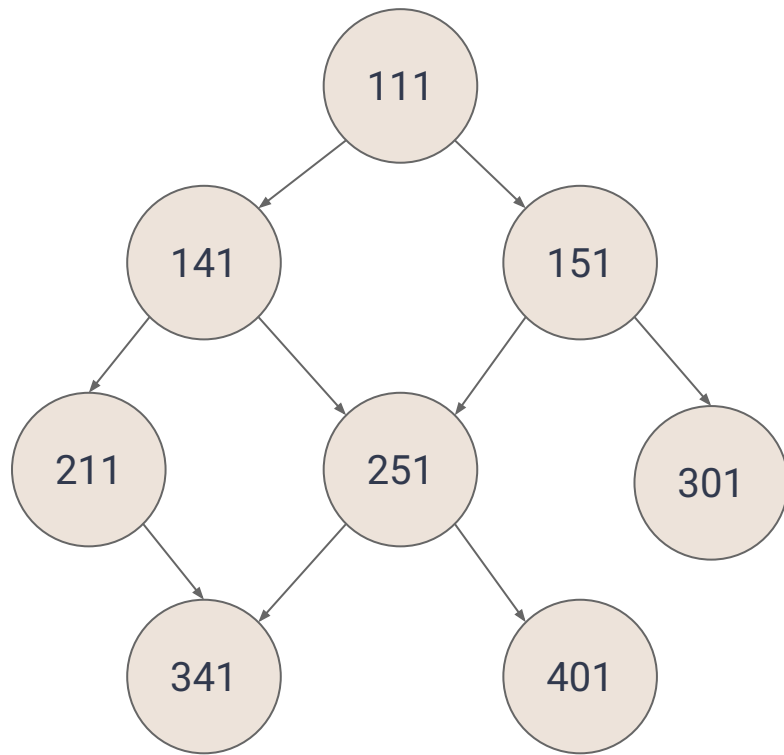
How do we find a topological ordering?

Where should we start?

We should start at a node with no in-edges. This is called a **source**.

A node with no out-edges is called a **sink**.

How do we know there is a **source**?

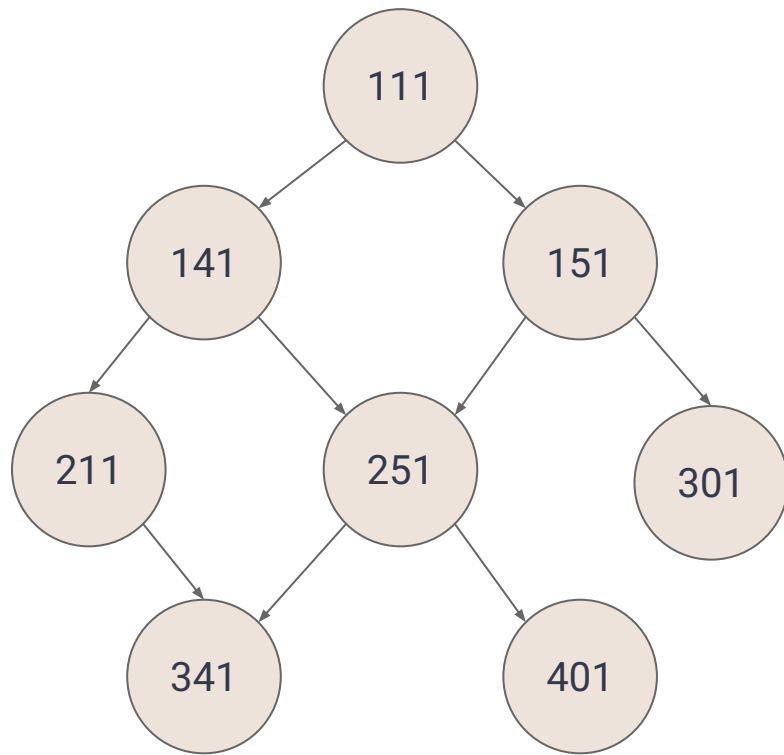


How do we find a topological ordering?

Claim: On a DAG, there must be a vertex that is a **source**.

Assume by contradiction that there is no source. Then every node has arrows pointing into it.

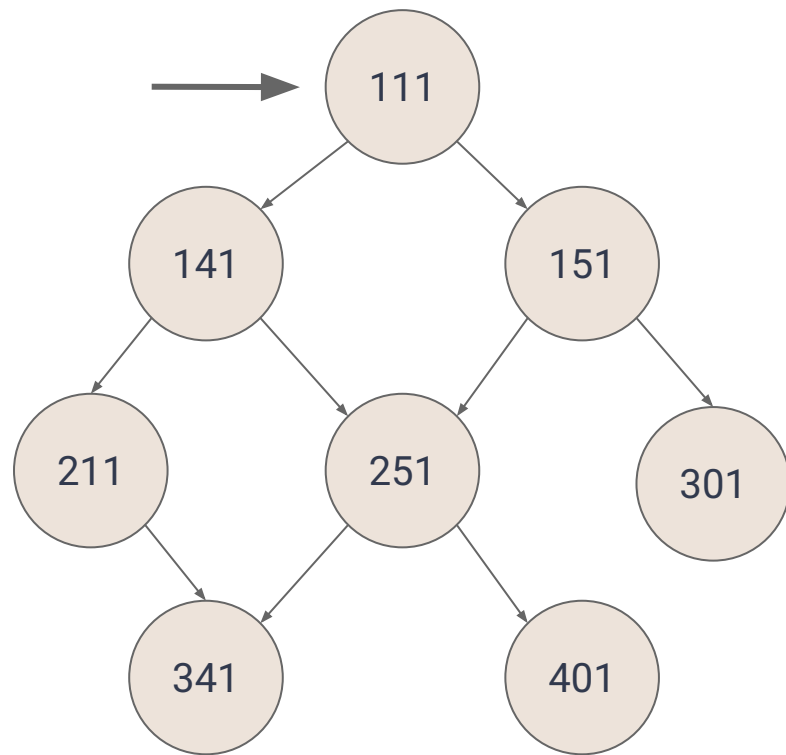
Pick any node. Traverse in reverse. Every node has an arrow pointing into it, so we can follow that arrow backwards. We never have to stop. Because we never have to stop, at some point we have to revisit a node. The path from the first visit to the second makes a cycle.



How do we find a topological ordering?

We could go recursively!

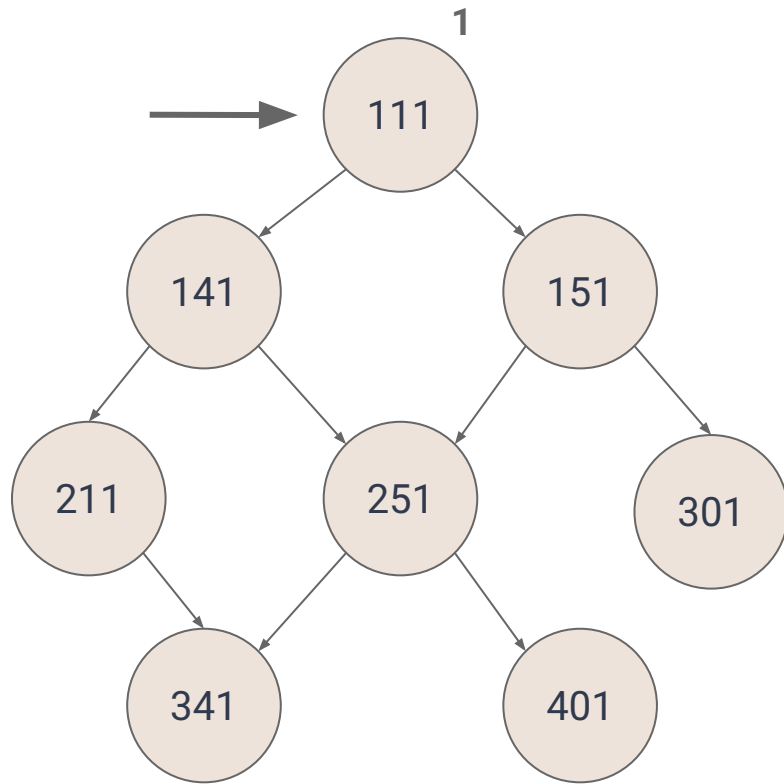
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

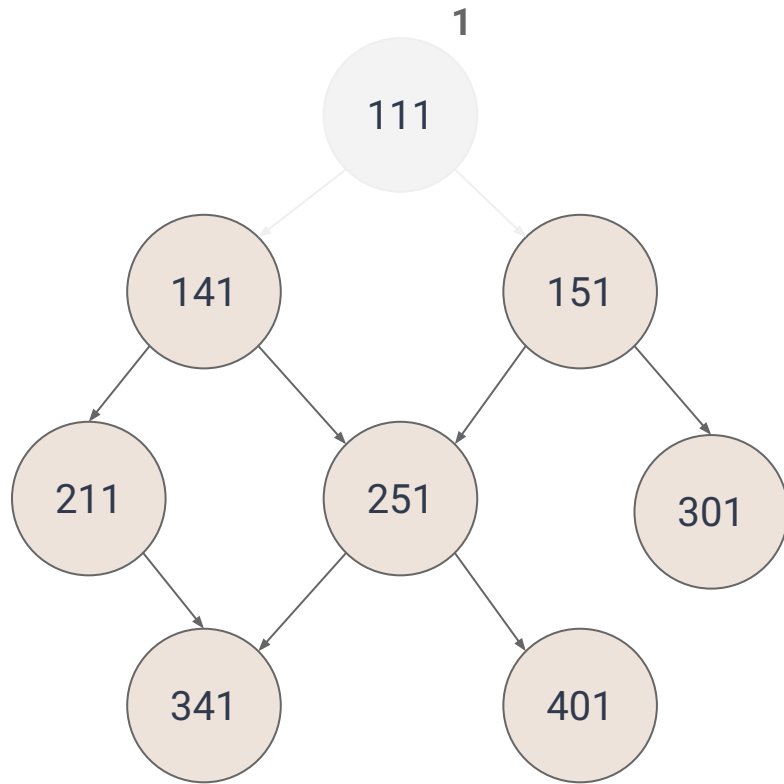
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

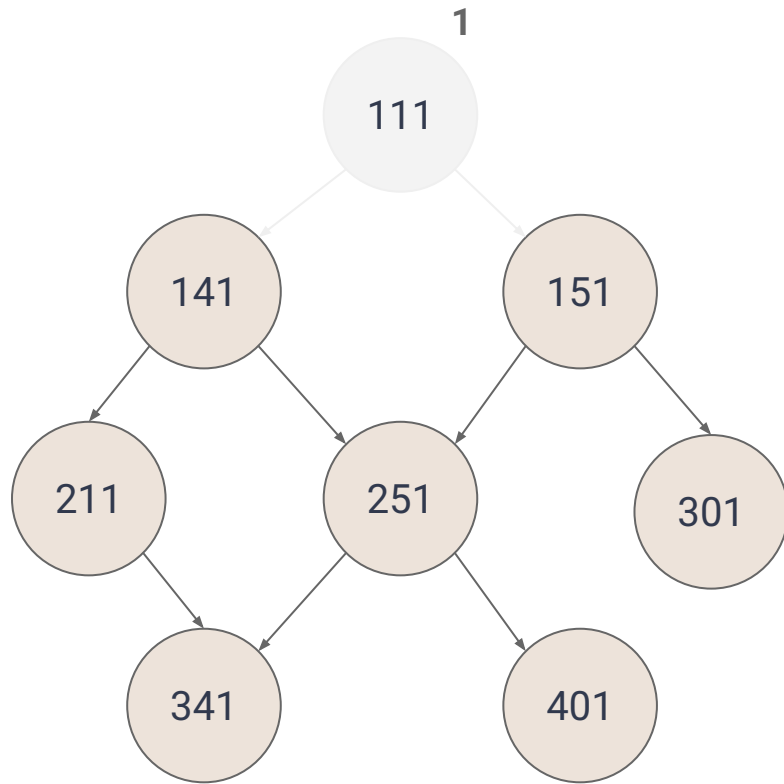
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

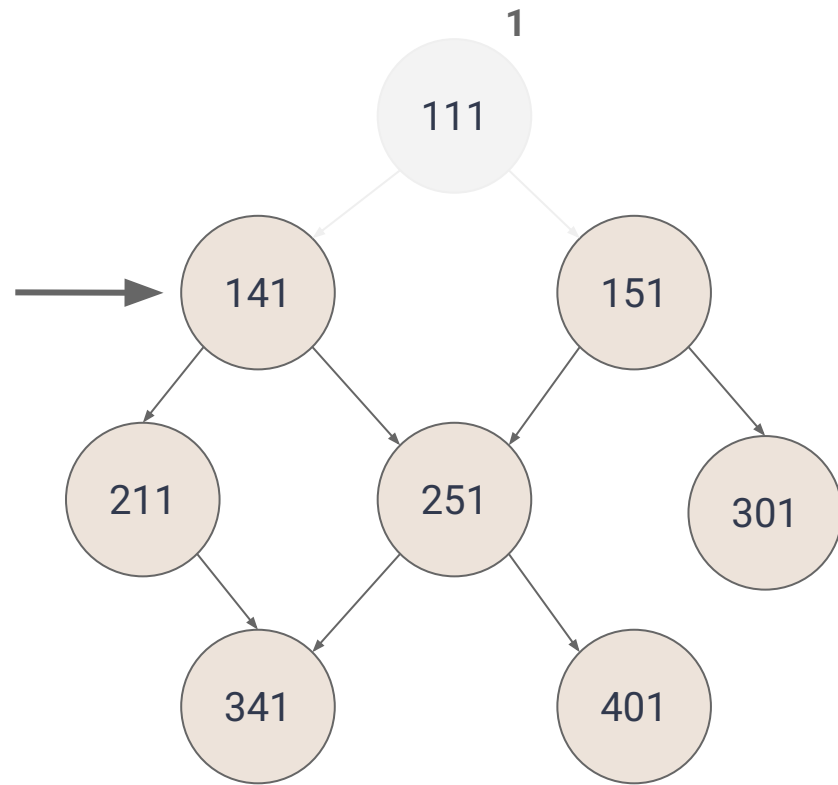
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

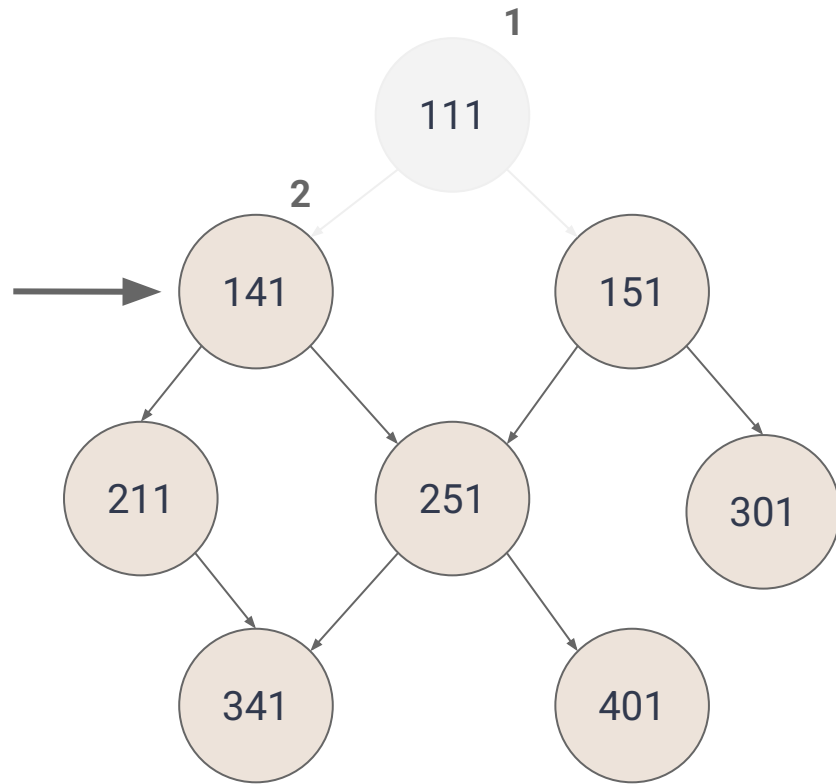
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

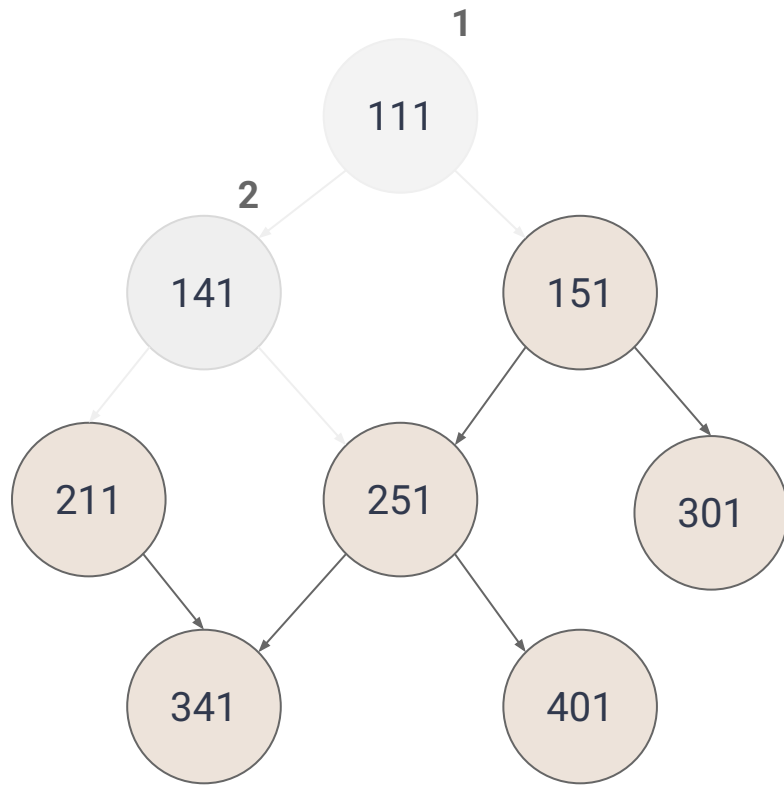
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

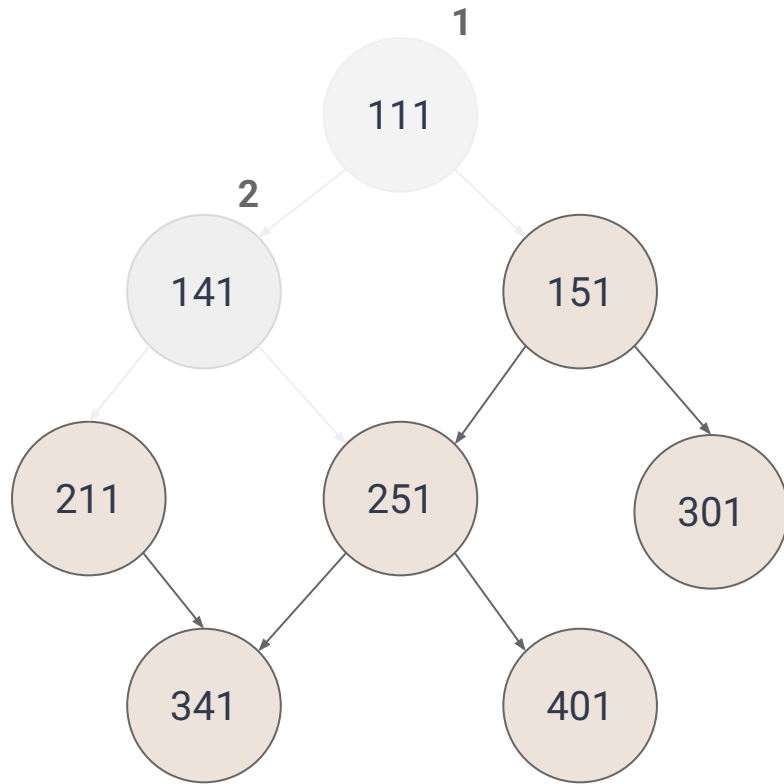
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

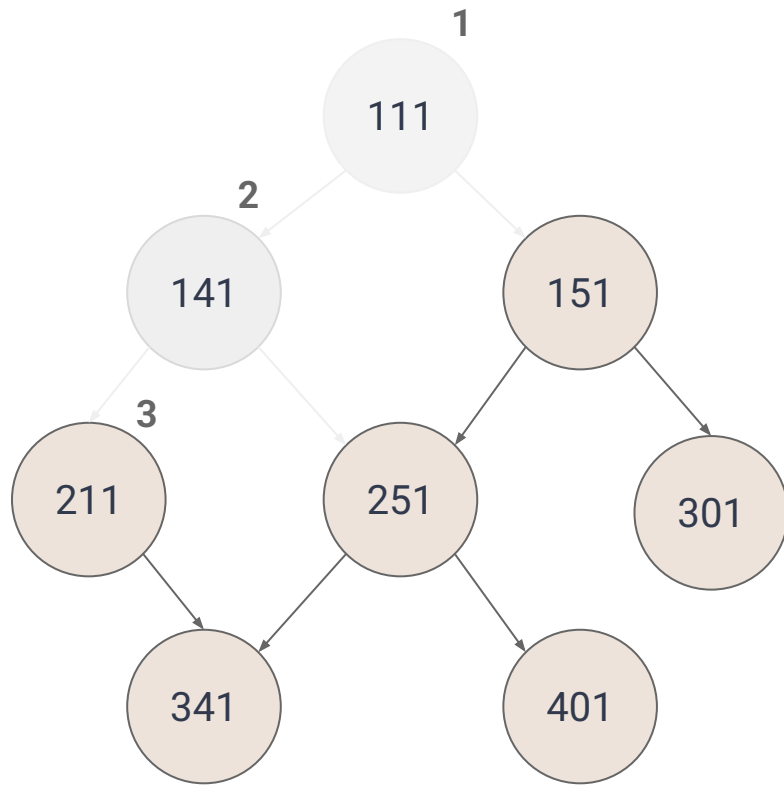
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

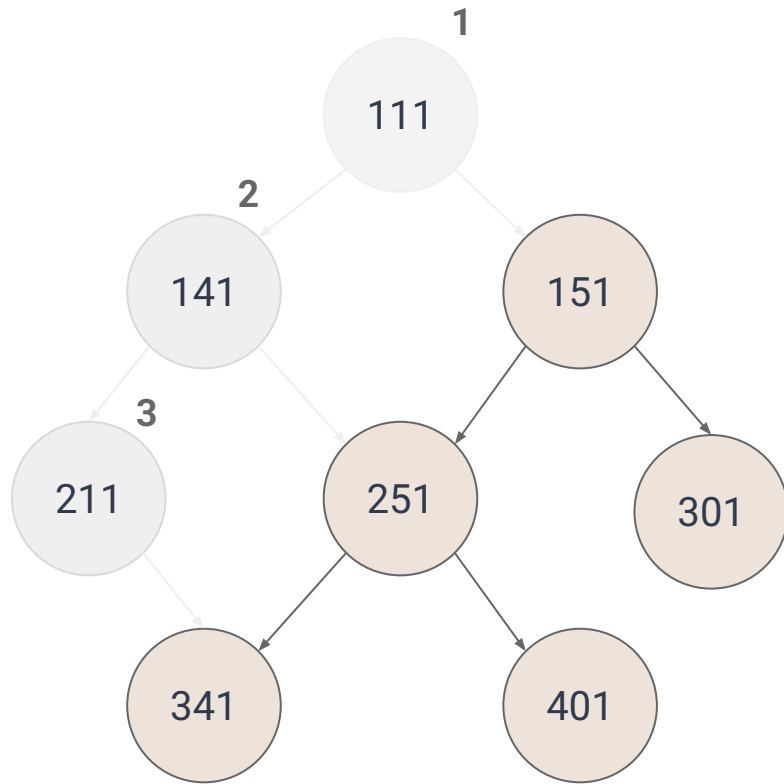
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

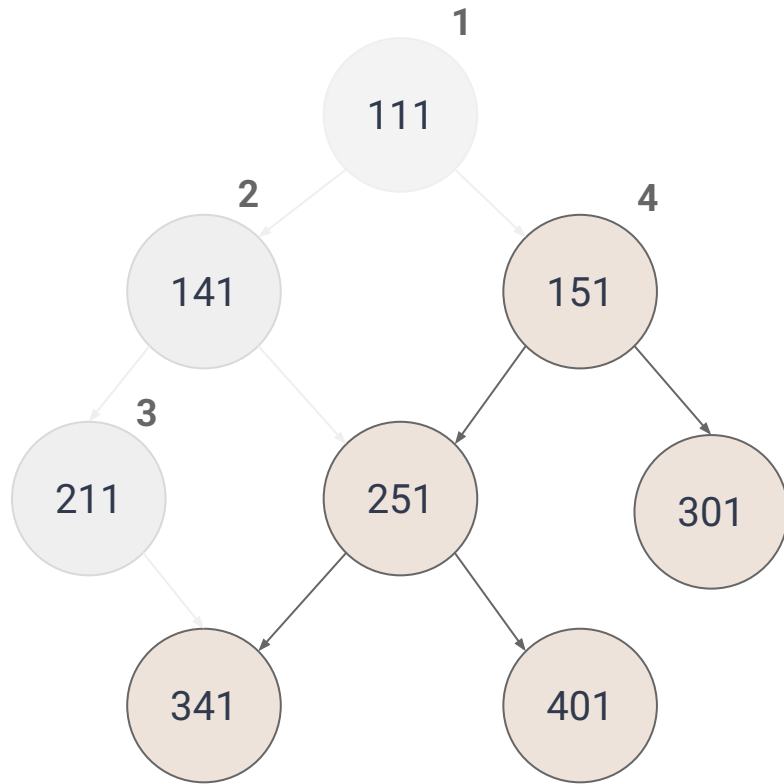
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

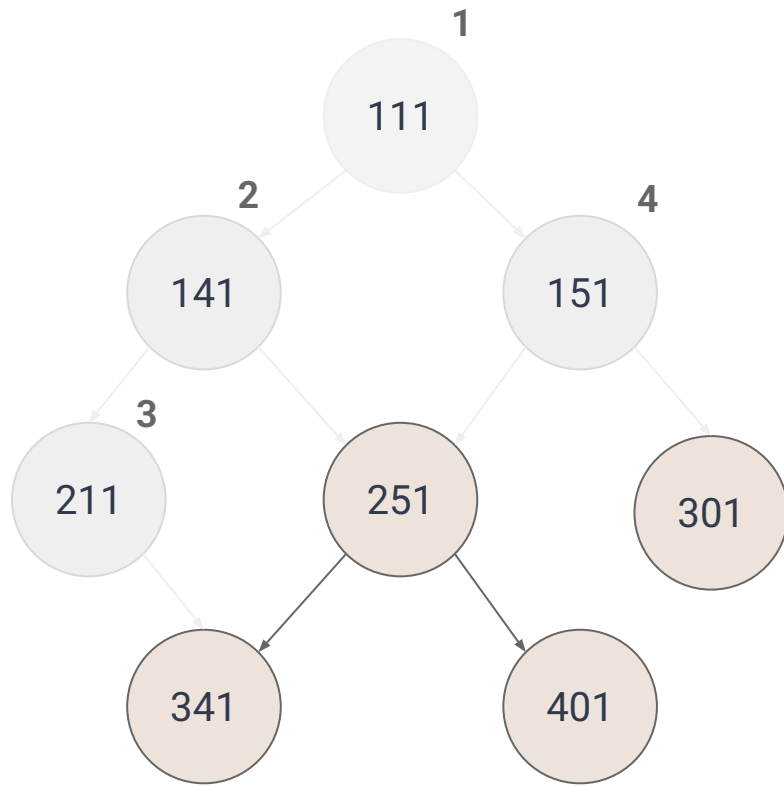
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

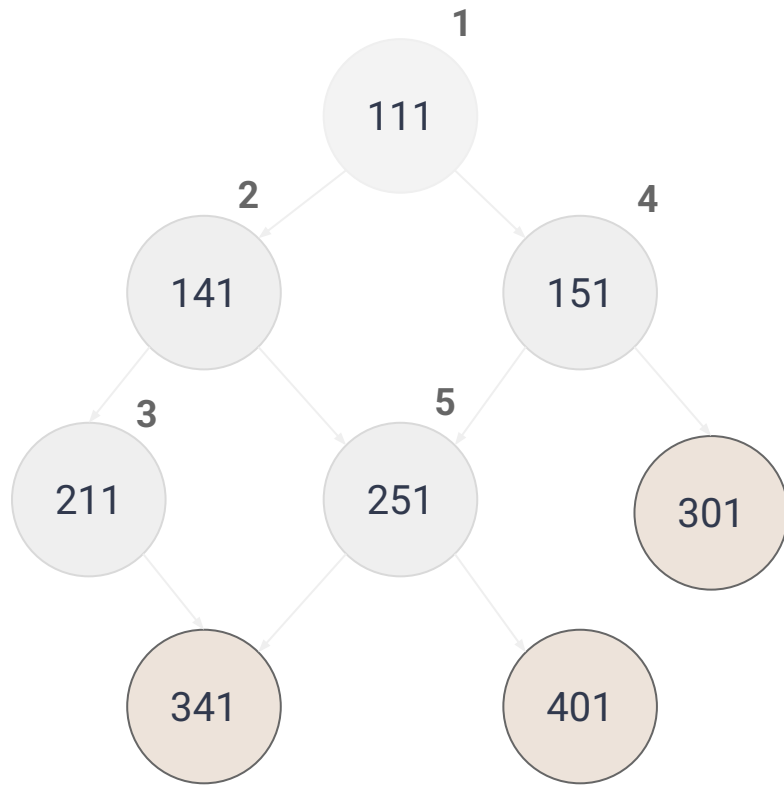
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

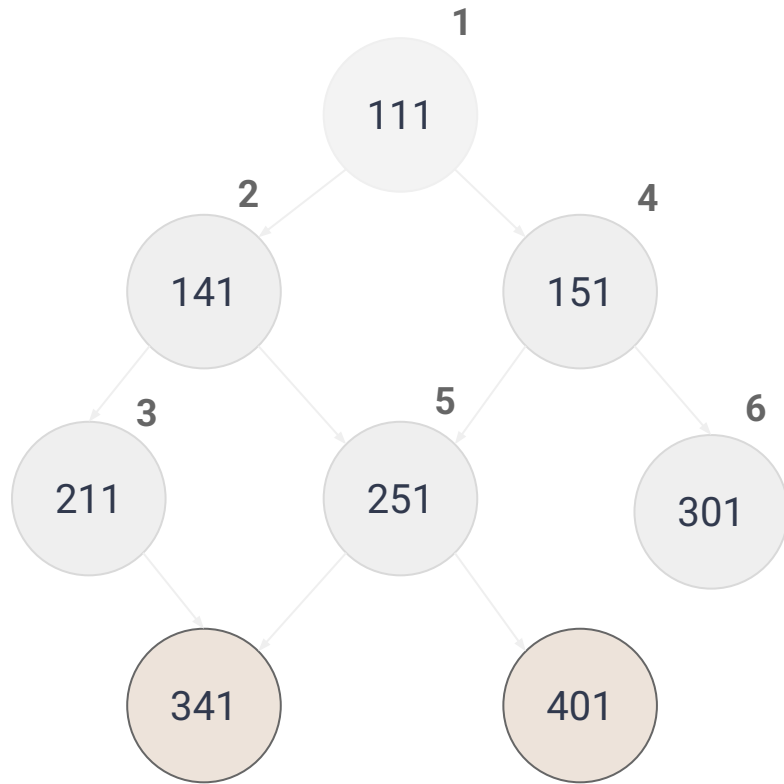
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

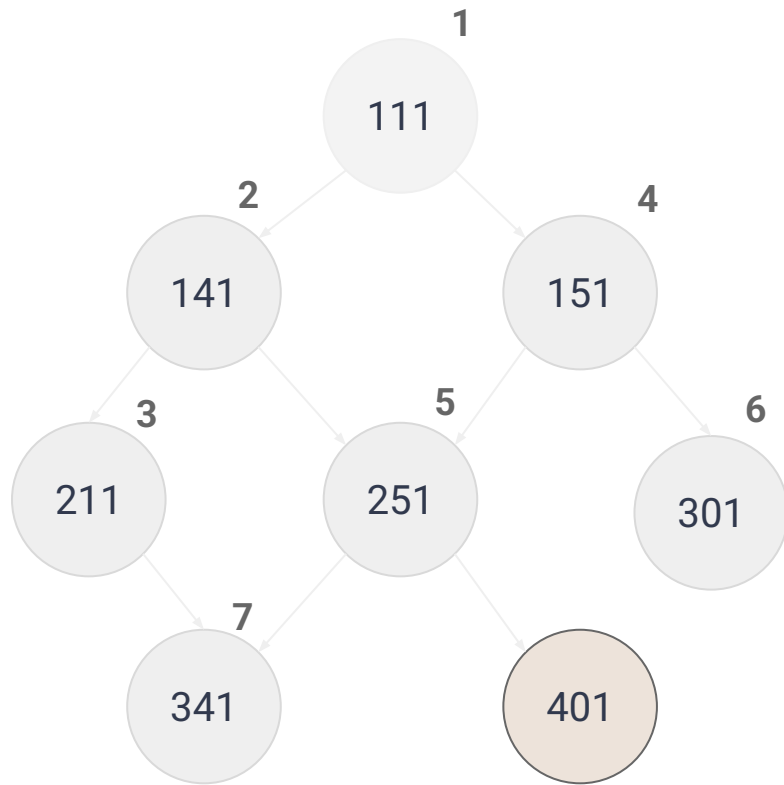
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

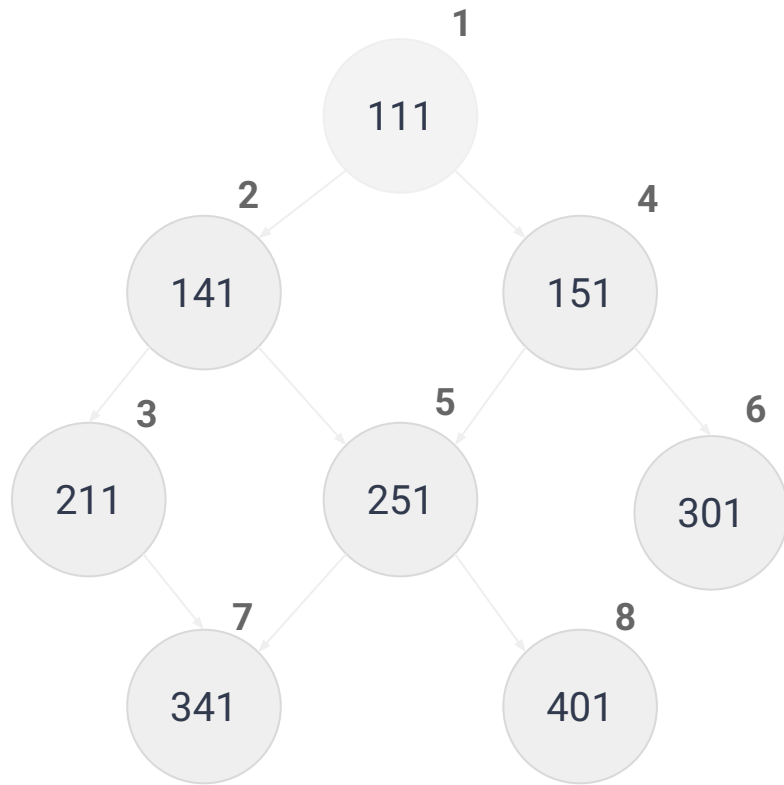
- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.



How do we find a topological ordering?

We could go recursively!

- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.

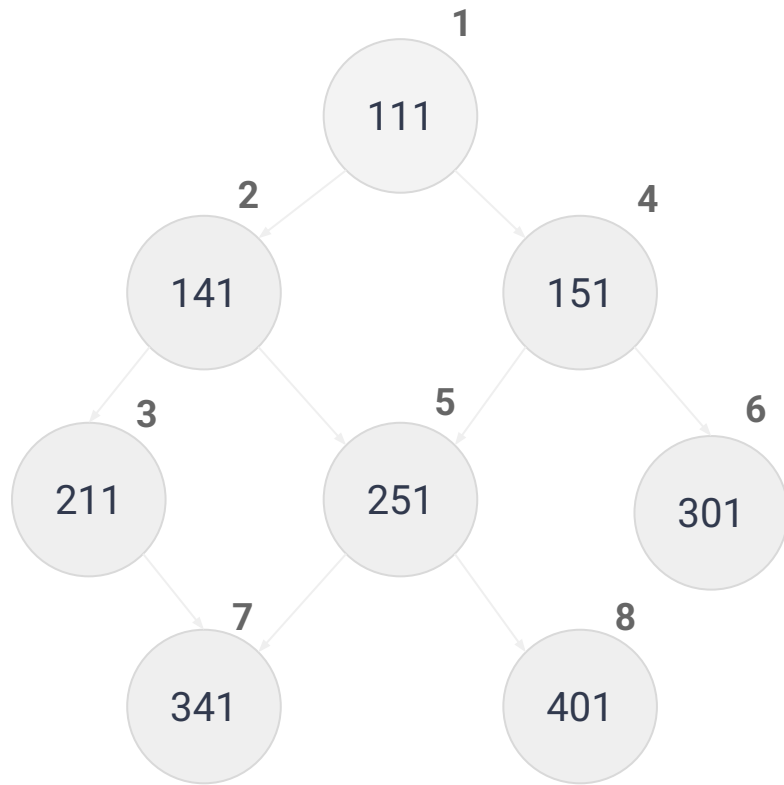


How do we find a topological ordering?

We could go recursively!

- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.

What could be costly about this algorithm?



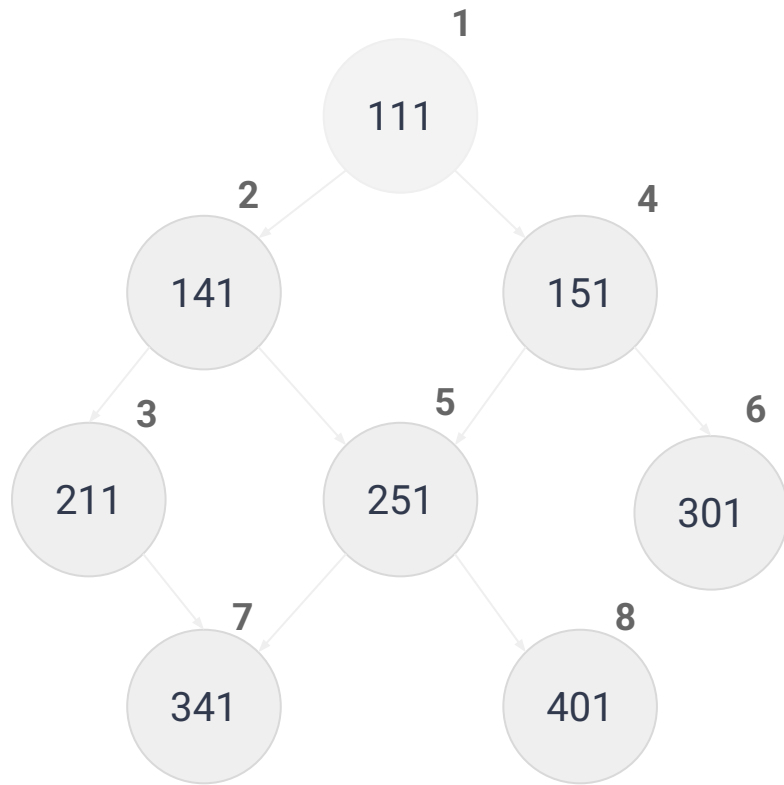
How do we find a topological ordering?

We could go recursively!

- Find a source.
- Say that node is next in the ordering.
- Remove that node.
- The remaining graph must still be a DAG, so we recurse.

What could be costly about this algorithm?

It might be hard to find a source.



Pre-order, Post-order, and Topological Sort



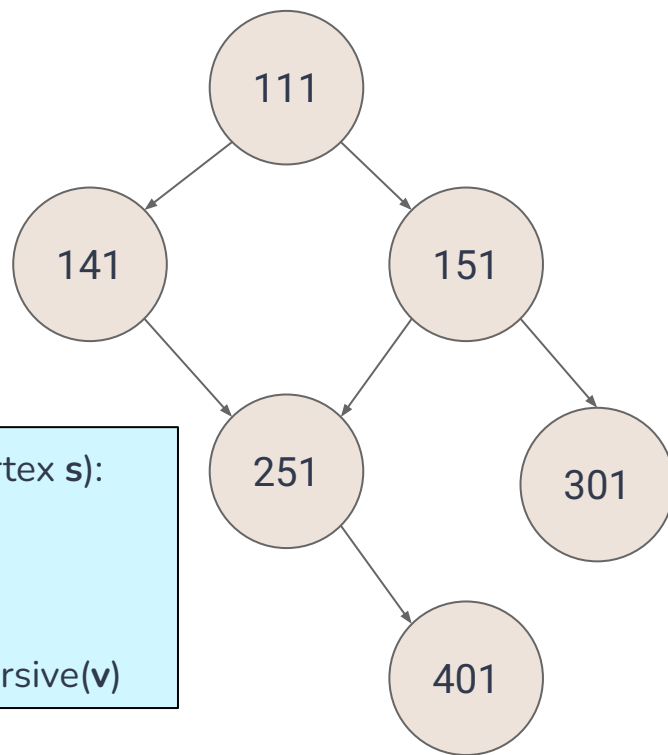
How do we find a topological ordering?

If I run DFS, starting from any vertex, what information might it give me about a topological order?

When **DFS-Recursive(s)** is done, then **s** can be added to the end of the topological ordering.

If we keep track of the order in which we post-visit vertices, that gives us a reverse topological order.

```
DFS-Recursive(starting vertex s):  
  mark s as visited  
  for each edge (s, v):  
    if v is unvisited:  
      DFS-Recursive(v)
```



How do we find a topological ordering?

If I run DFS, starting from any vertex, what information might it give me about a topological order?

When **DFS-Recursive(s)** is done, then **s** can be added to the end of the topological ordering.

If we keep track of the order in which we post-visit vertices, that gives us a reverse topological order.

clock = 1

postvisit(s):

post[s] = clock

clock = clock + 1

DFS-Recursive(starting vertex s):

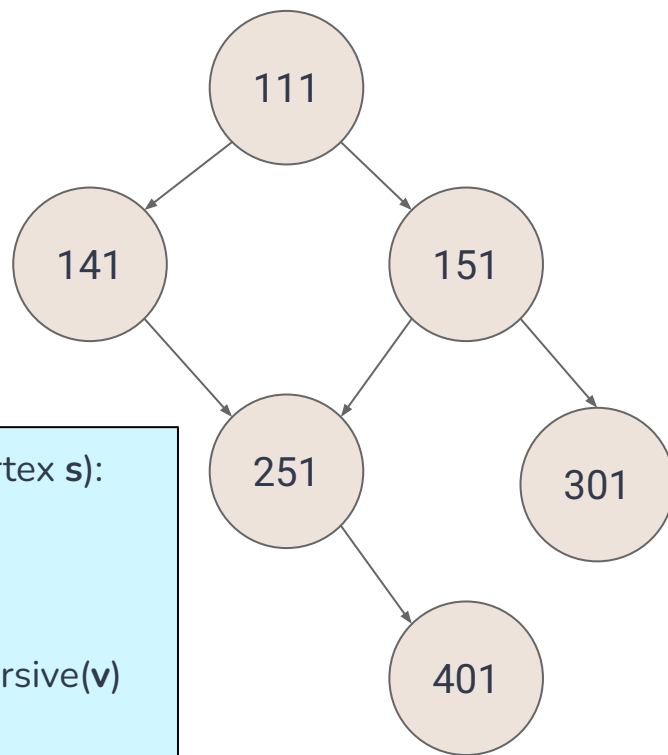
mark **s** as visited

for each edge (**s**, **v**):

if **v** is unvisited:

DFS-Recursive(**v**)

postvisit(s)



Visit Tracking for DFS (Worksheet)

The order in which the vertices are **first** processed.

clock1 = 1
previsit(s):
 pre[s] = clock1
 clock1 = clock1 + 1

DFS-Recursive(starting vertex **s**):
 previsit(s)
 mark **s** as visited
 for each edge (**s**, **v**):
 if **v** is unvisited:
 DFS-Recursive(**v**)
 postvisit(s)

Reminder: Post-visit order is a reverse topological sort!

The order in which the vertices are **done** being visited.

clock2 = 1
postvisit(s):
 post[s] = clock2
 clock2 = clock2 + 1

Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex **s**):

previsit(s)

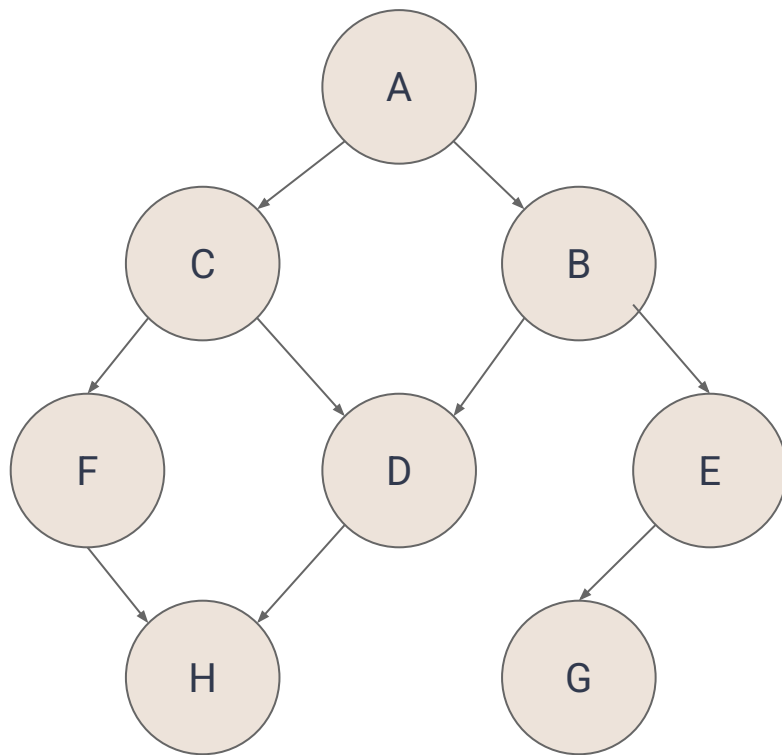
mark **s** as visited

for each edge (**s**, **v**):

if **v** is unvisited:

DFS-Recursive(**v**)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

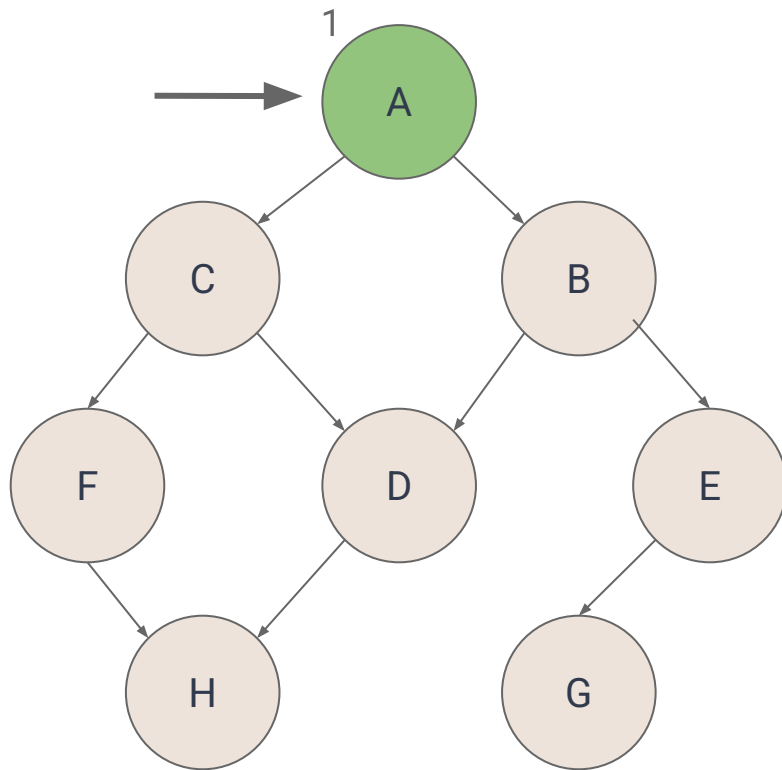
 mark s as visited

 for each edge (s, v) :

 if v is unvisited:

 DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

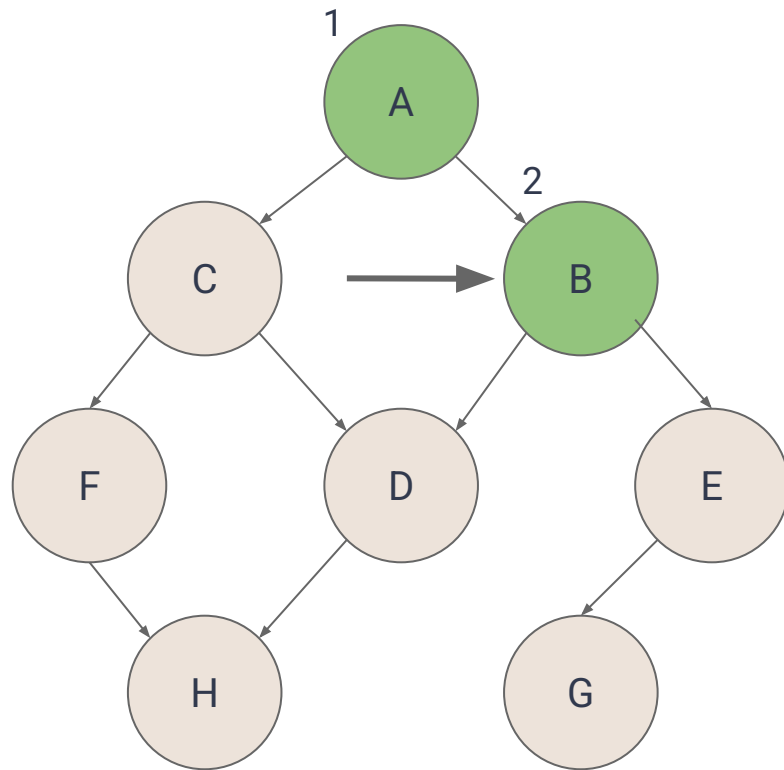
 mark s as visited

 for each edge (s, v) :

 if v is unvisited:

 DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

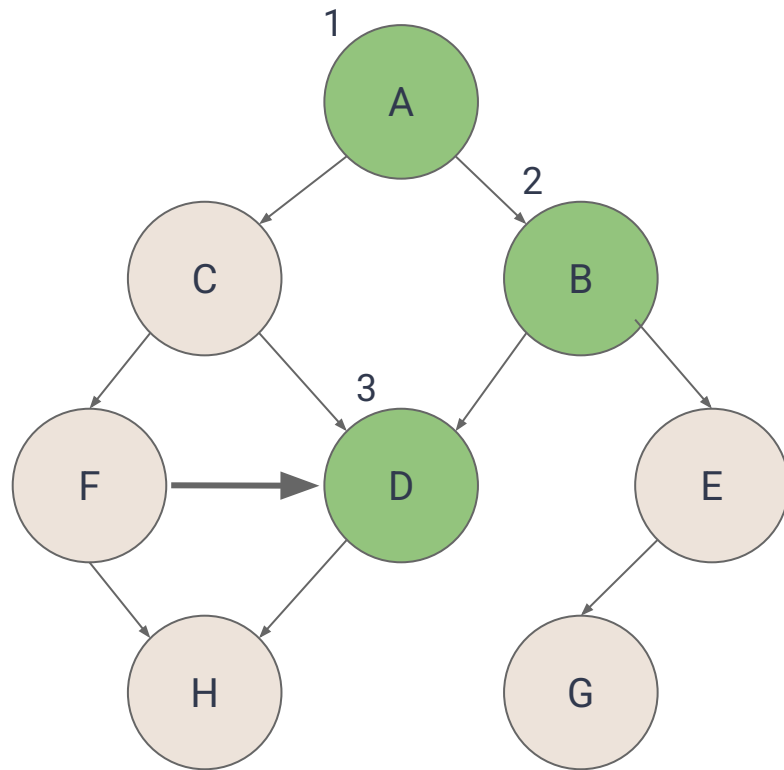
 mark s as visited

 for each edge (s, v) :

 if v is unvisited:

 DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

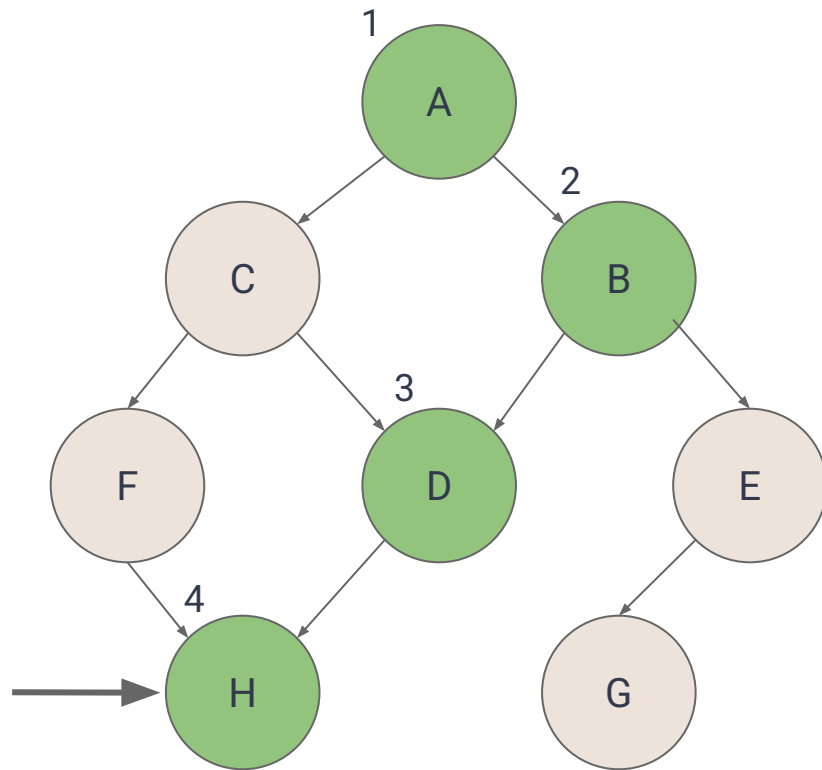
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

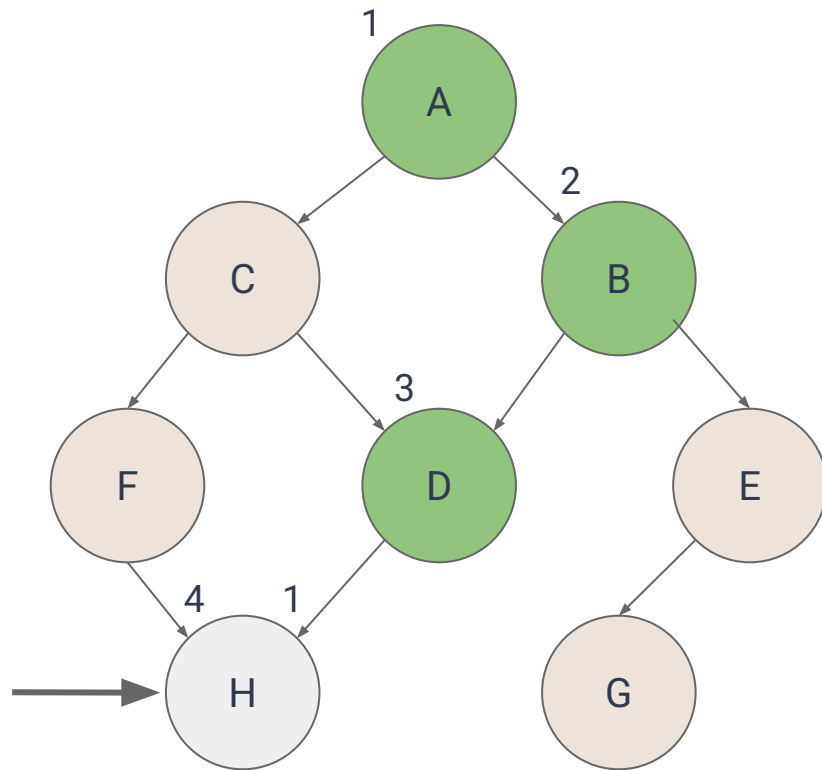
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

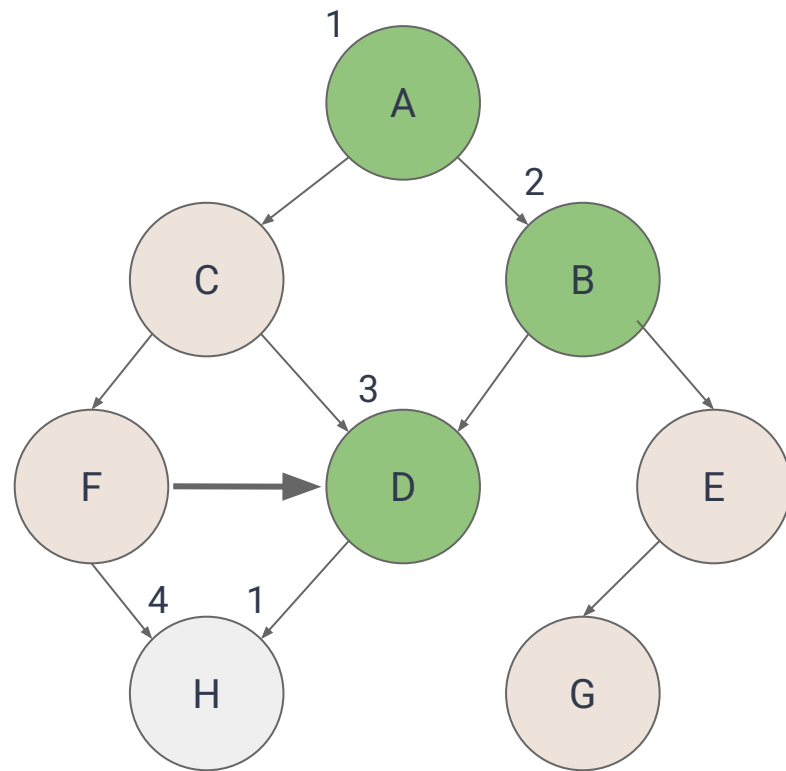
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

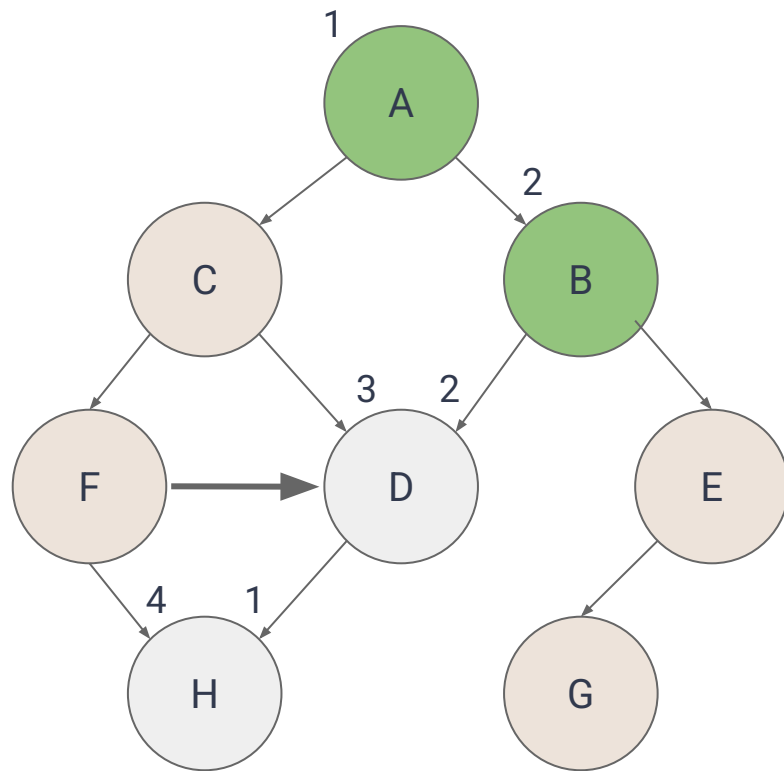
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

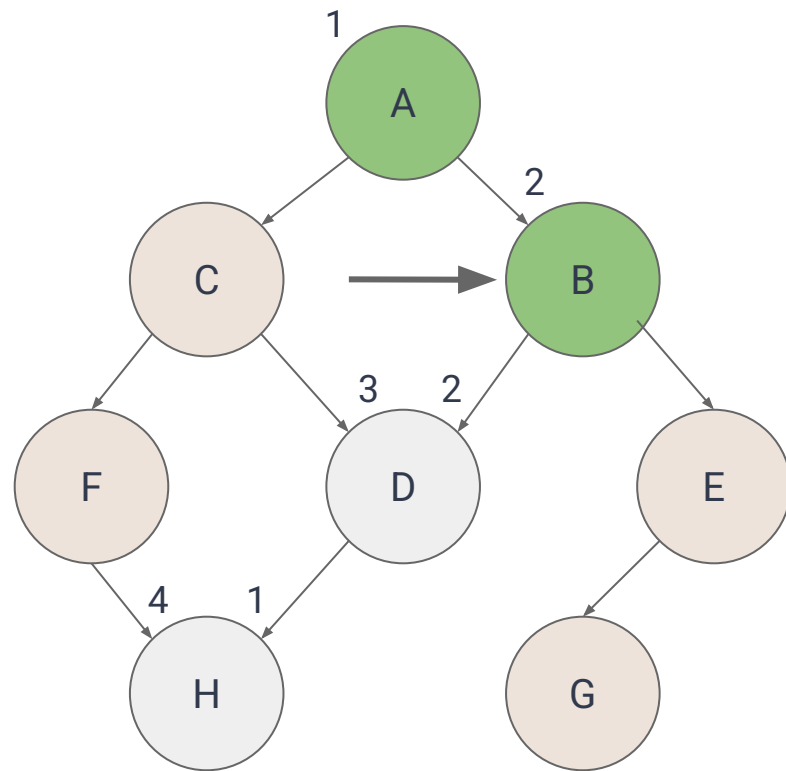
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

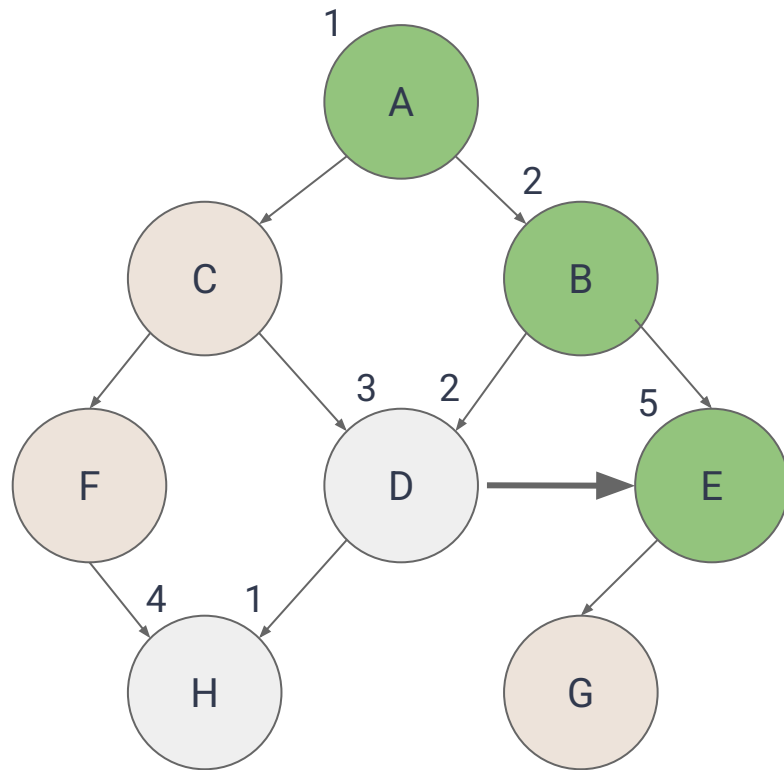
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

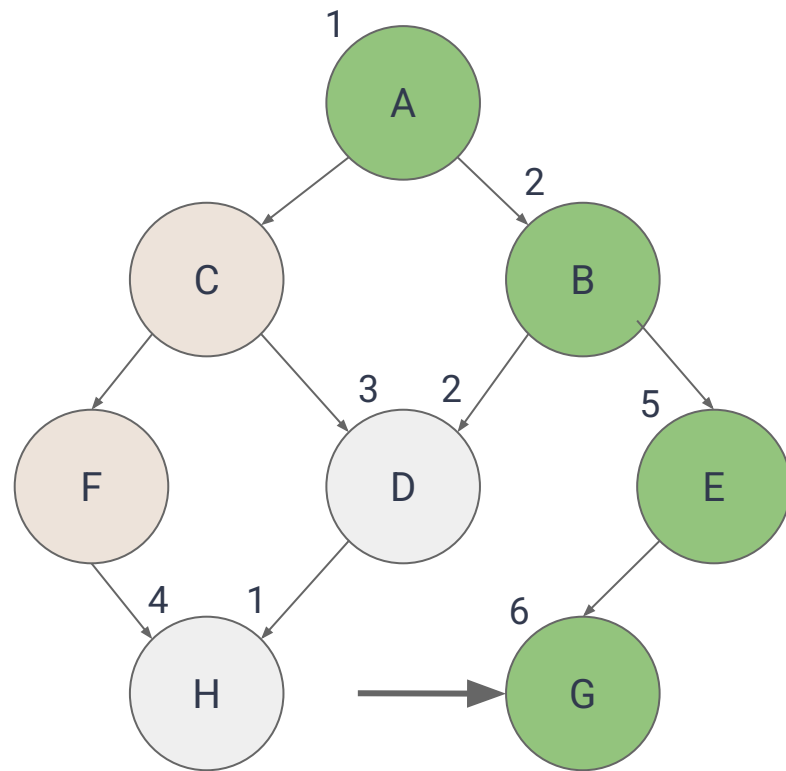
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

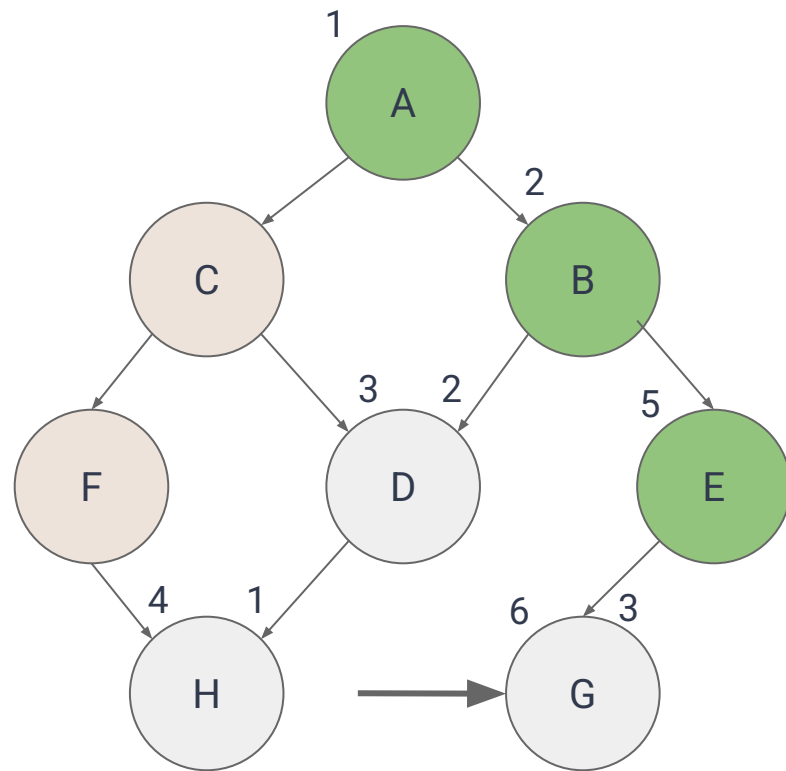
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

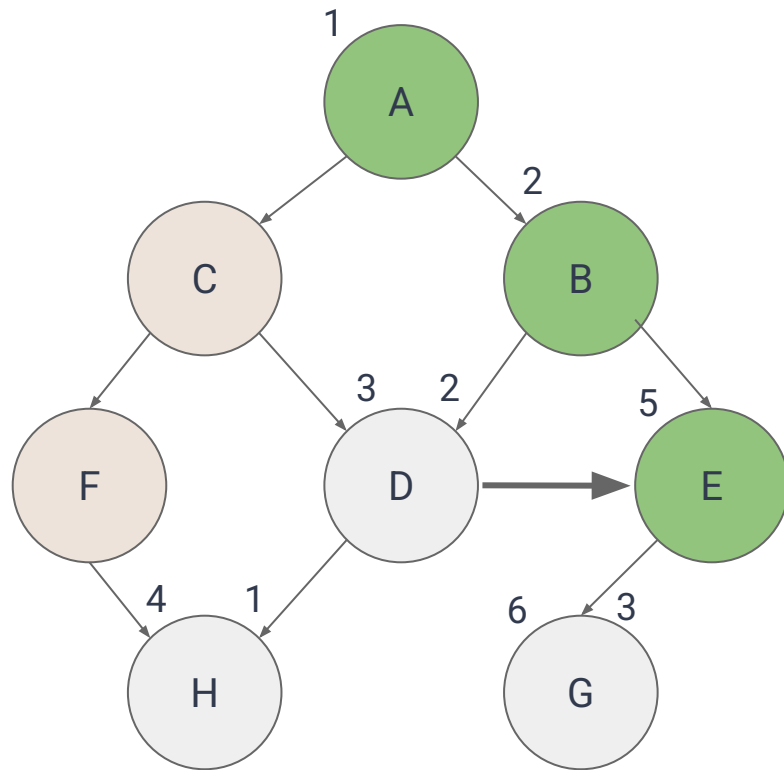
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

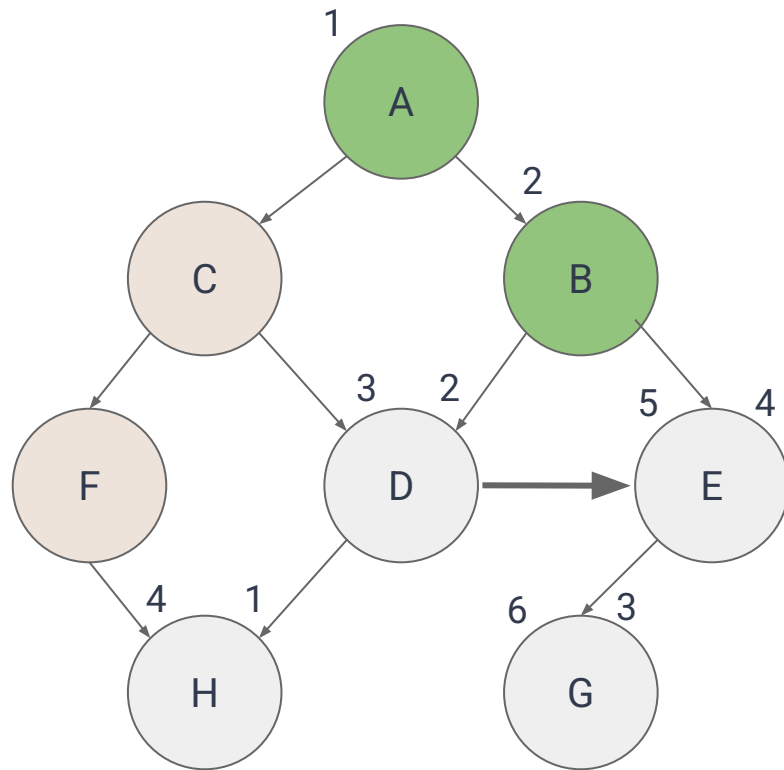
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

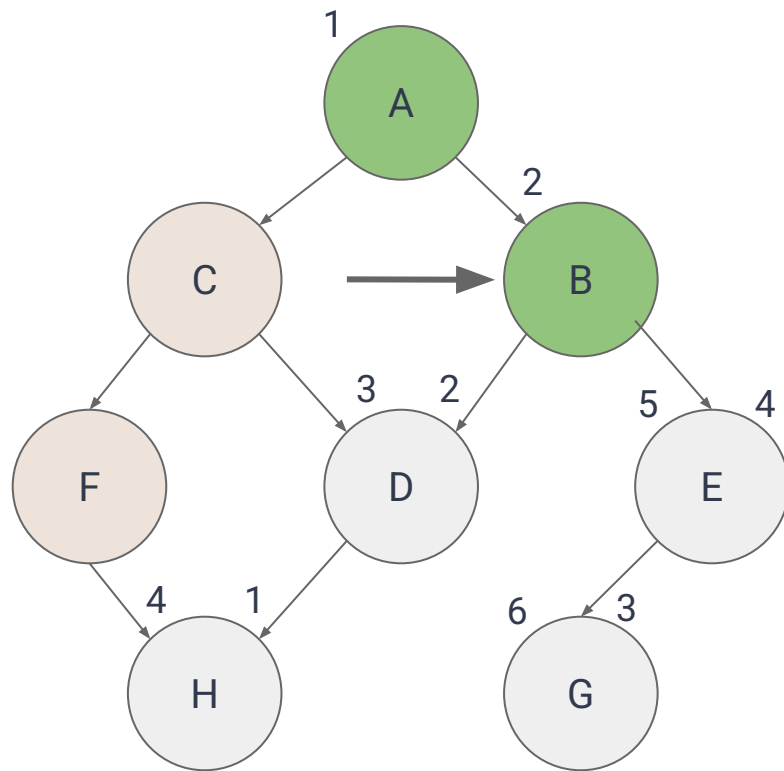
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

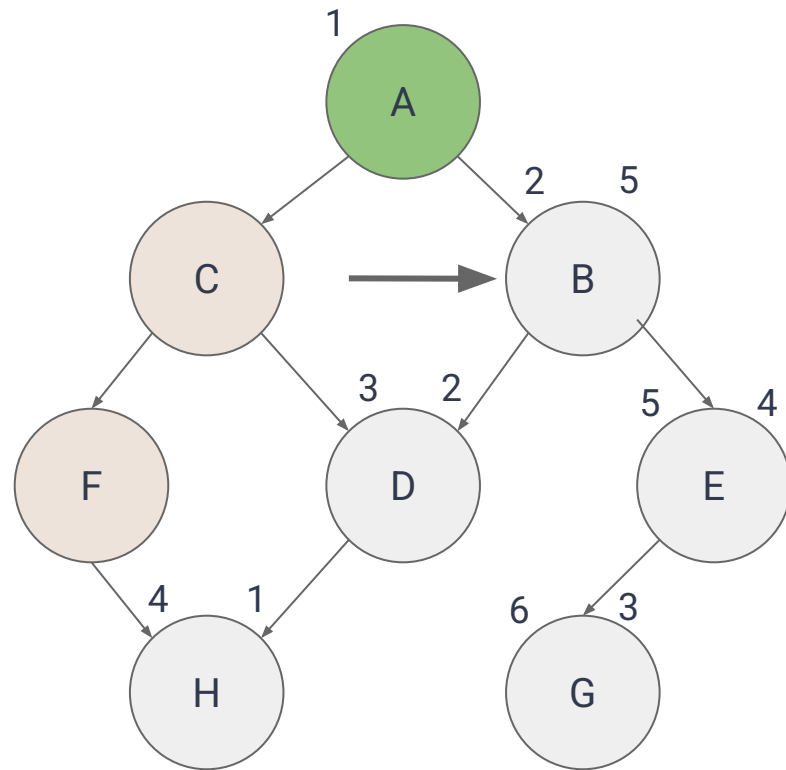
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

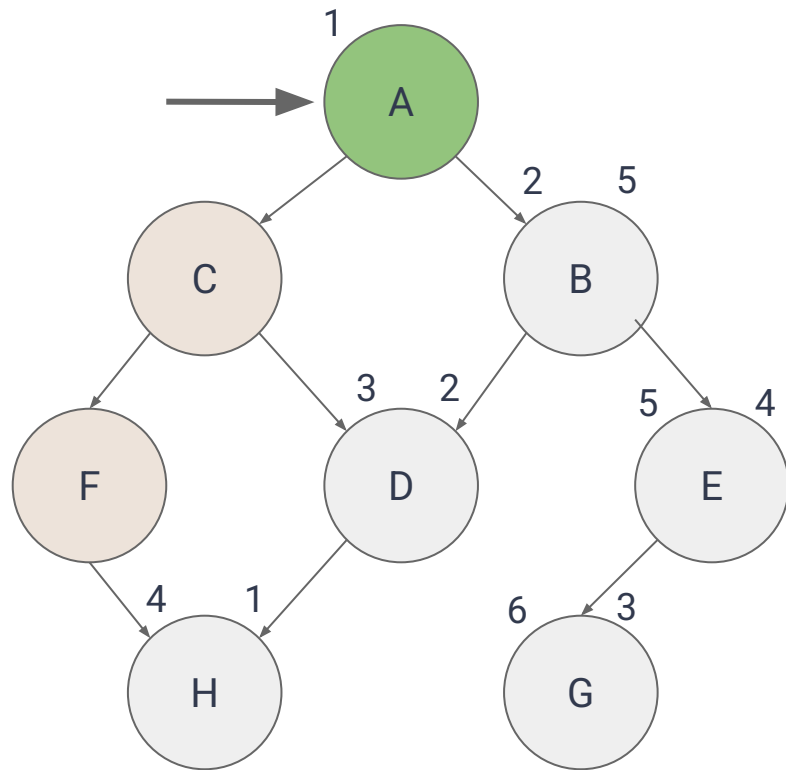
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

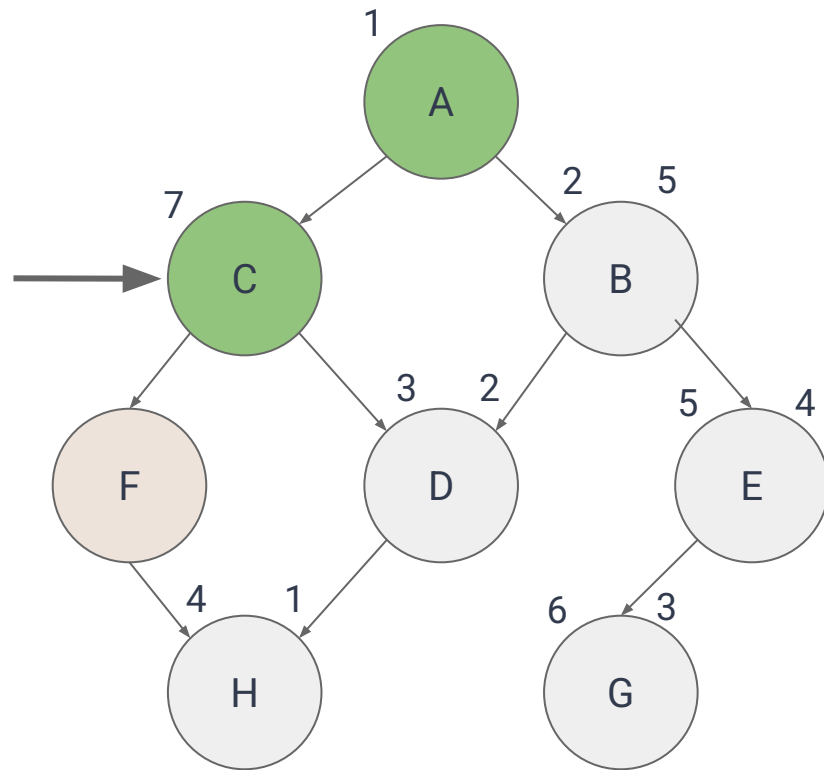
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

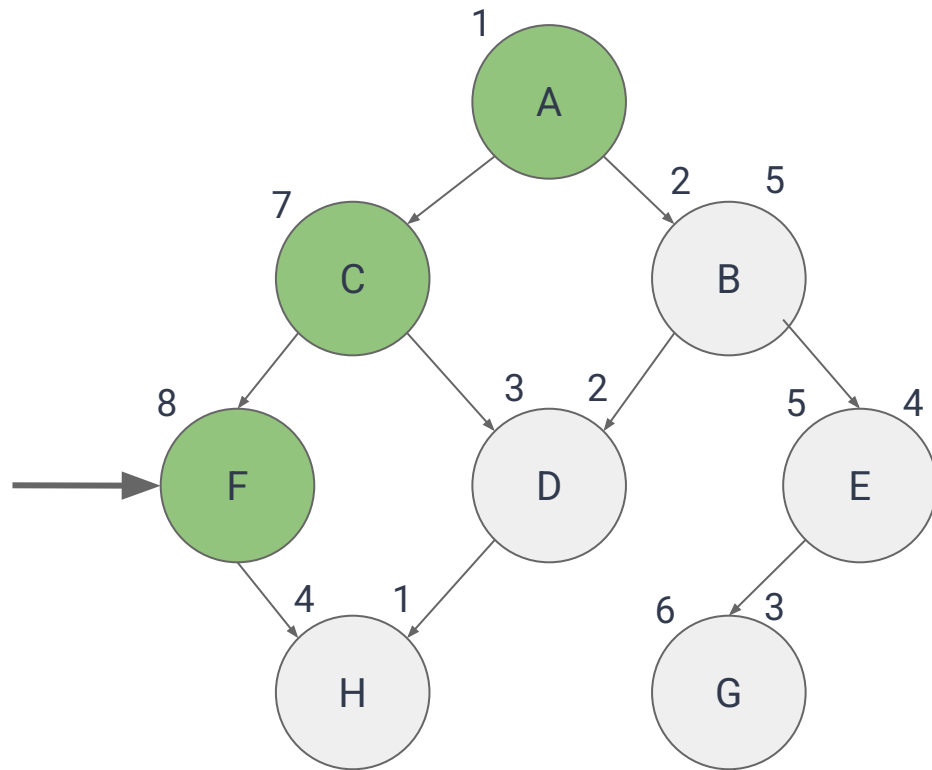
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

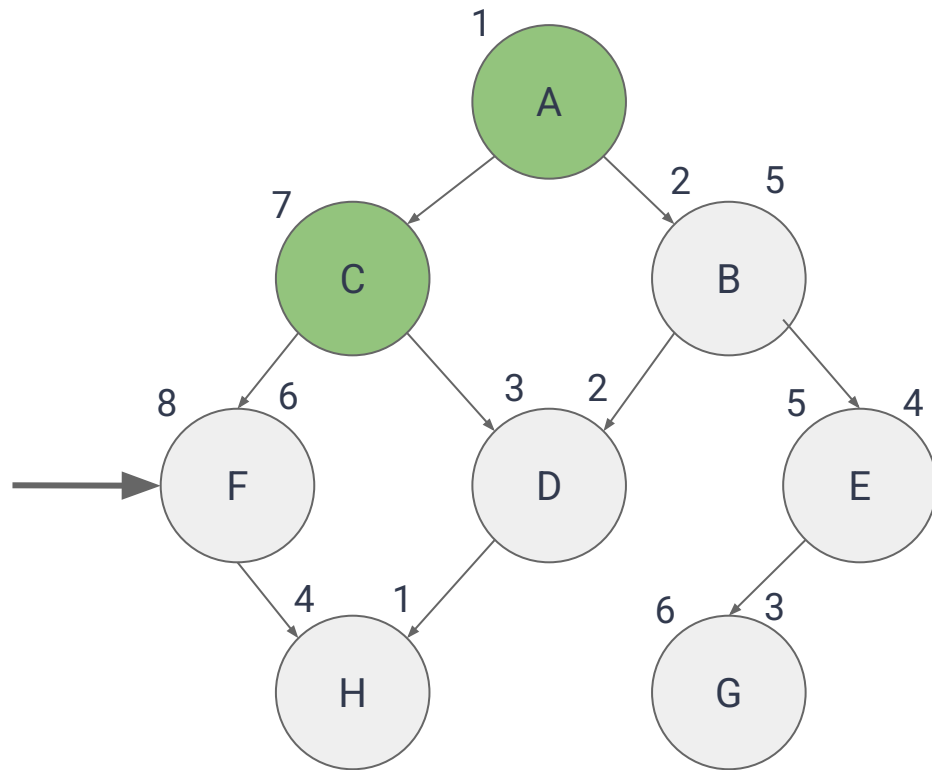
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

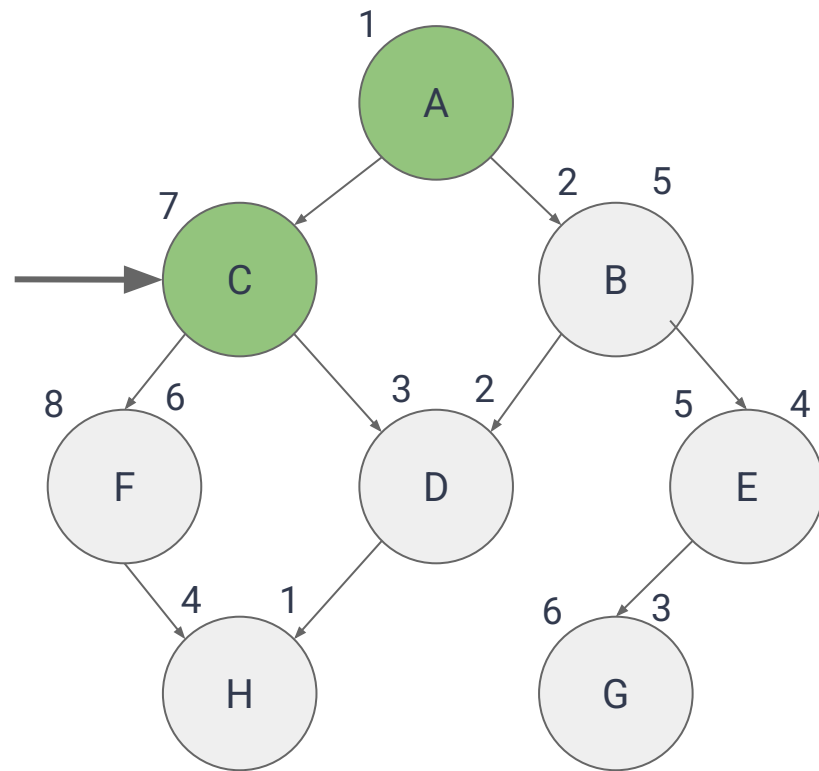
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex s):

previsit(s)

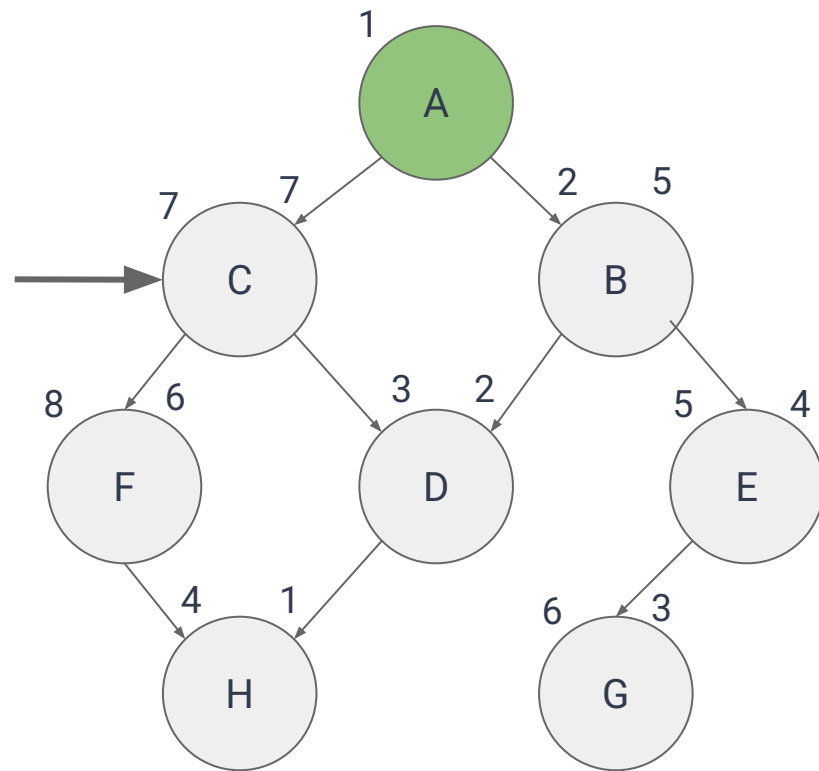
mark s as visited

for each edge (s, v) :

if v is unvisited:

DFS-Recursive(v)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex **s**):

previsit(s)

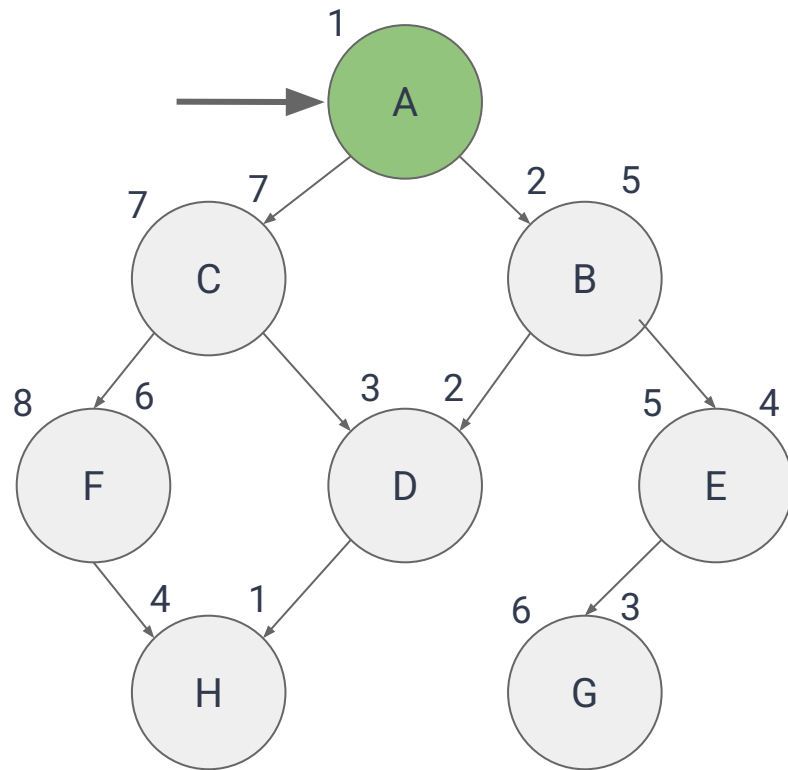
mark **s** as visited

for each edge (**s**, **v**):

if **v** is unvisited:

DFS-Recursive(**v**)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex **s**):

previsit(s)

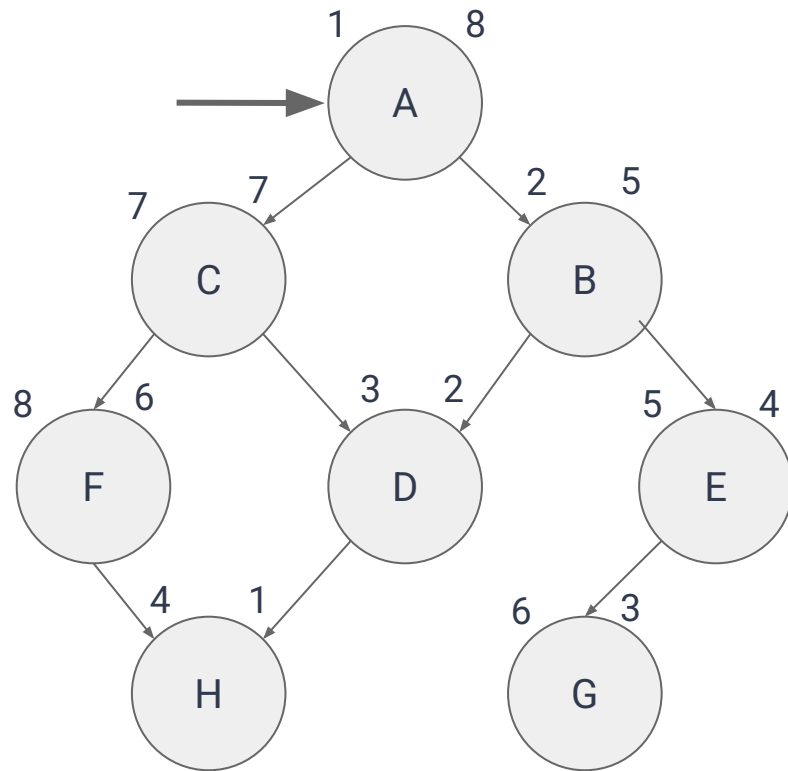
mark **s** as visited

for each edge (**s**, **v**):

if **v** is unvisited:

DFS-Recursive(**v**)

postvisit(s)



Visit Tracking for DFS (Worksheet)

DFS-Recursive(starting vertex **s**):

previsit(s)

mark **s** as visited

for each edge (**s**, **v**):

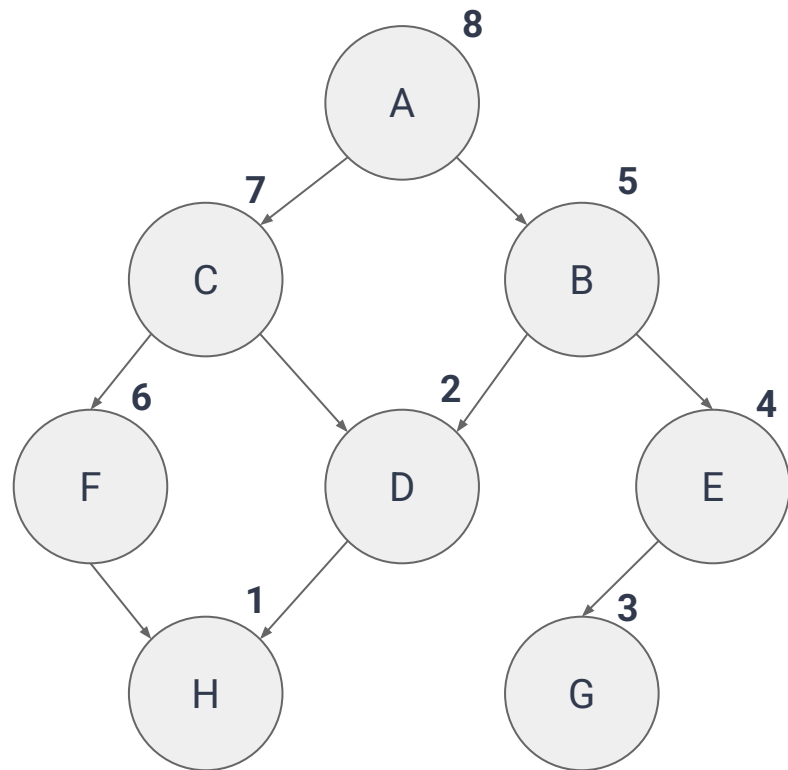
if **v** is unvisited:

DFS-Recursive(**v**)

postvisit(s)

In reverse post-order:

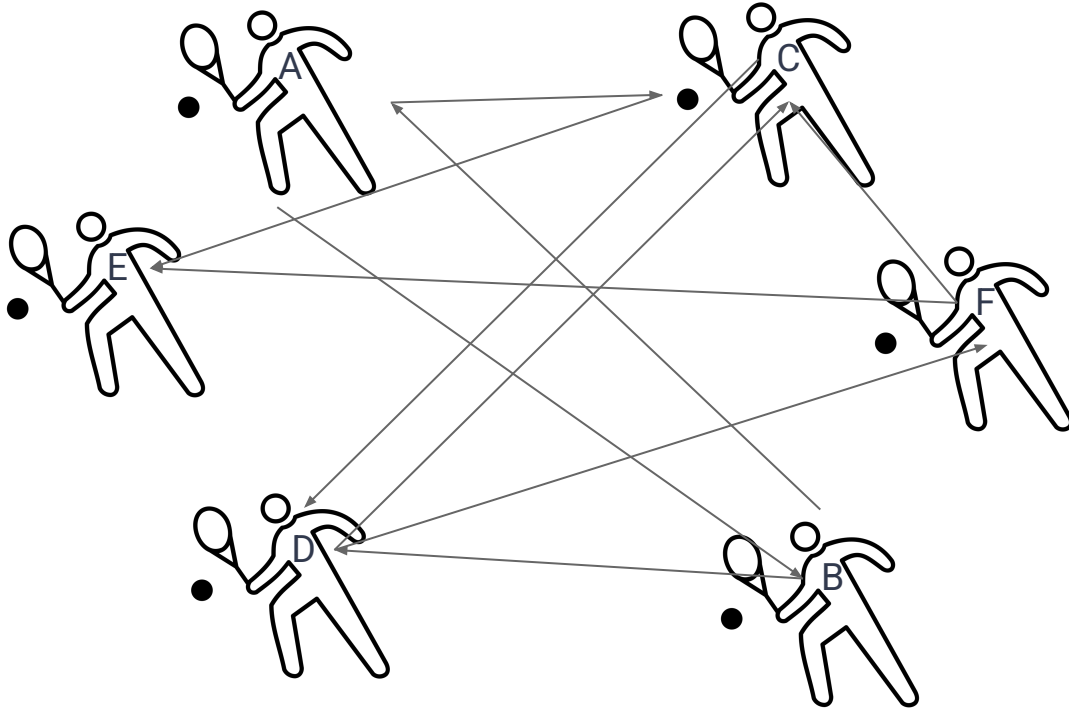
ACFBEGDH



What about non-acyclic graphs?

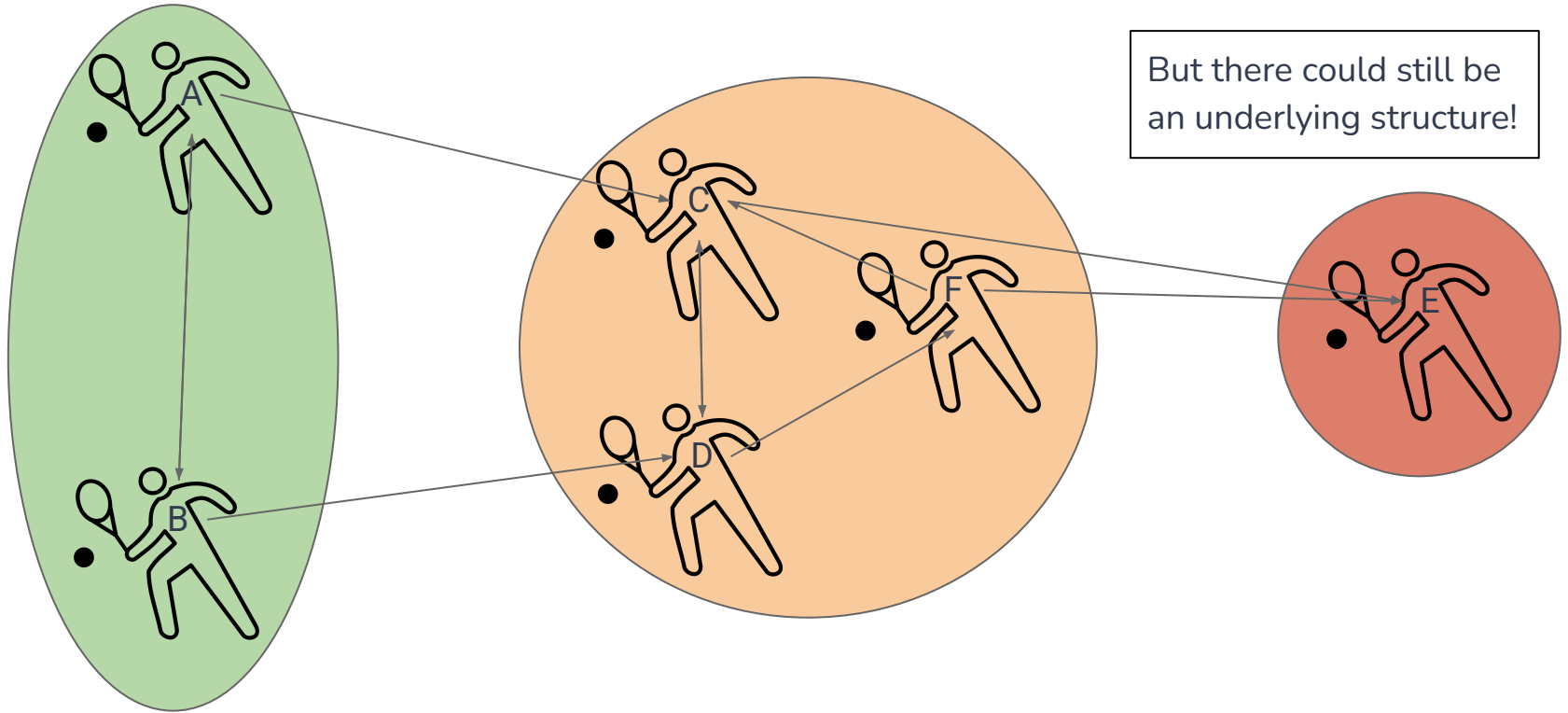


Not every directed graph is acyclic!

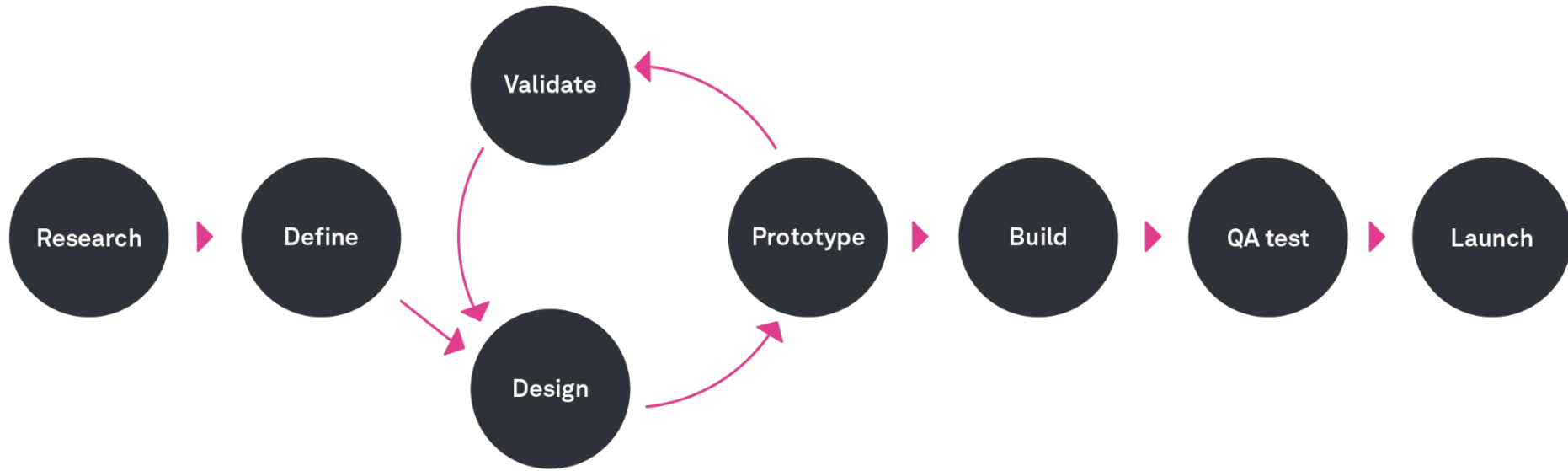


Who is the best player?

Not every directed graph is acyclic!



Not every directed graph is acyclic!



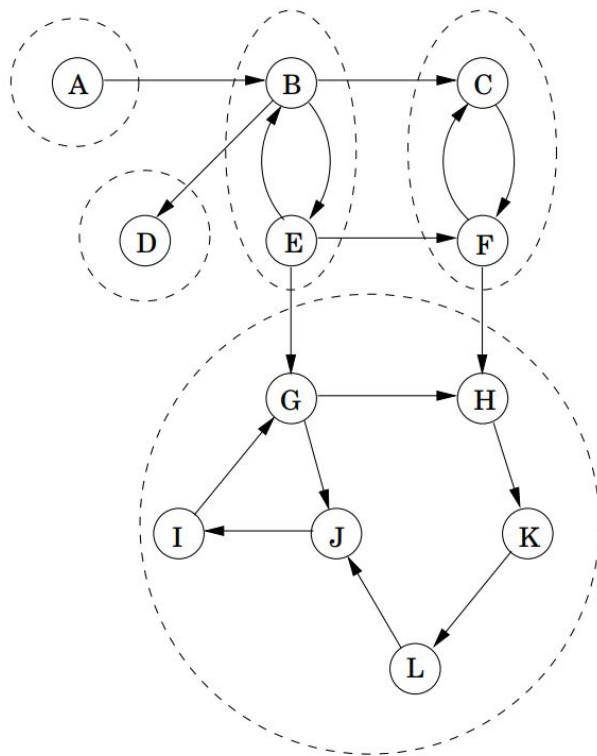
- The steps involved in developing a product

Strongly Connected Component

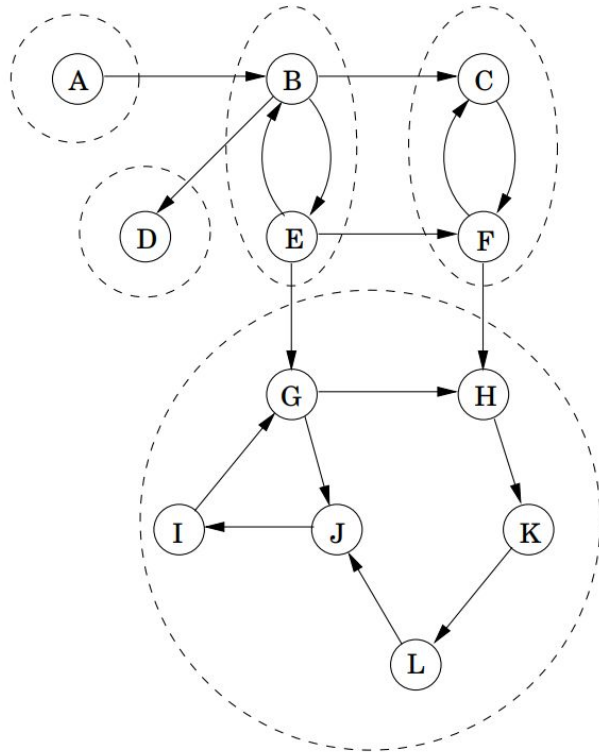
A **strongly connected component** of a directed graph is a *maximal* set of vertices where all vertices can reach all other vertices.

Maximal: No vertex can be added to the set and maintain its strong connectedness.

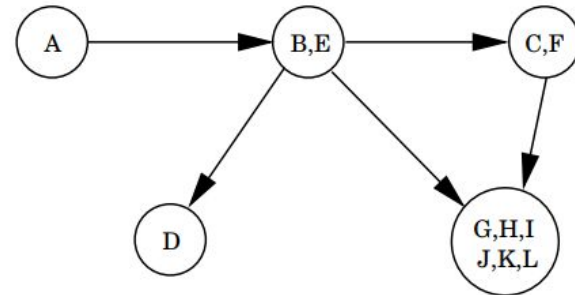
We want to know how to break up our graph into strongly connected components.



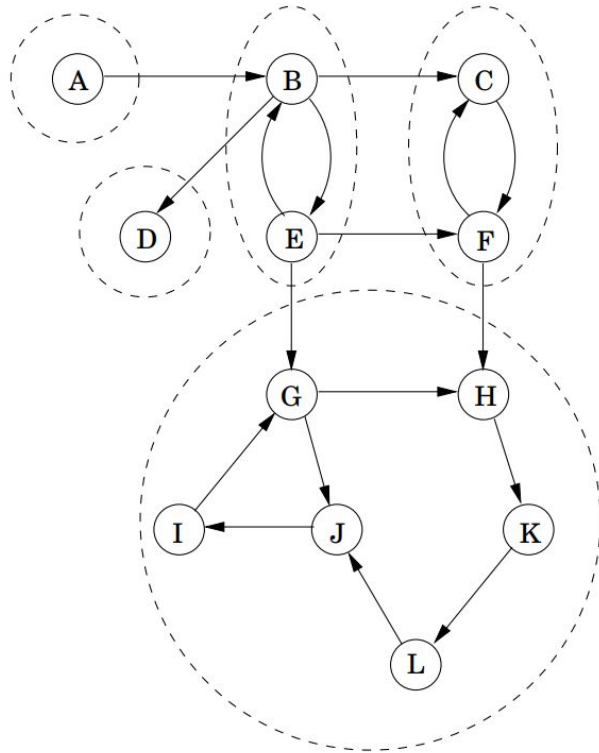
Problem Statement



Given a directed graph,
decompose it into a DAG of
strongly connected components.



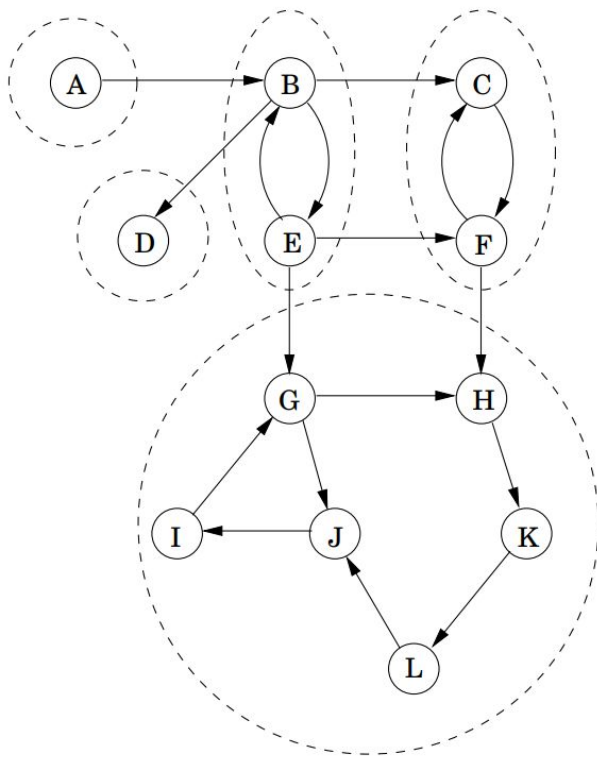
Problem Statement



Given a directed graph,
decompose it into a DAG of
strongly connected components.

Why don't BFS or DFS work?

Insight 1: Find a sink!



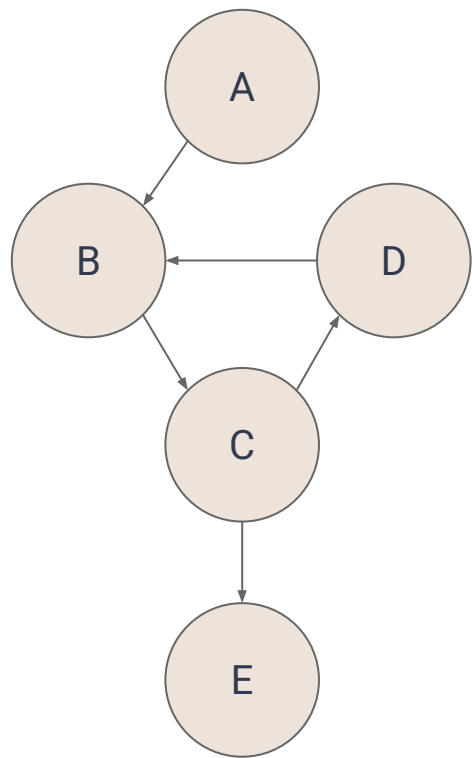
Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

A first useful insight: If we can identify a vertex in a sink, we can run DFS from there to find all vertices in that SCC.

But how do we find a sink?

What happens if we DFS on a graph with cycles?

Insight 1: Find a sink!



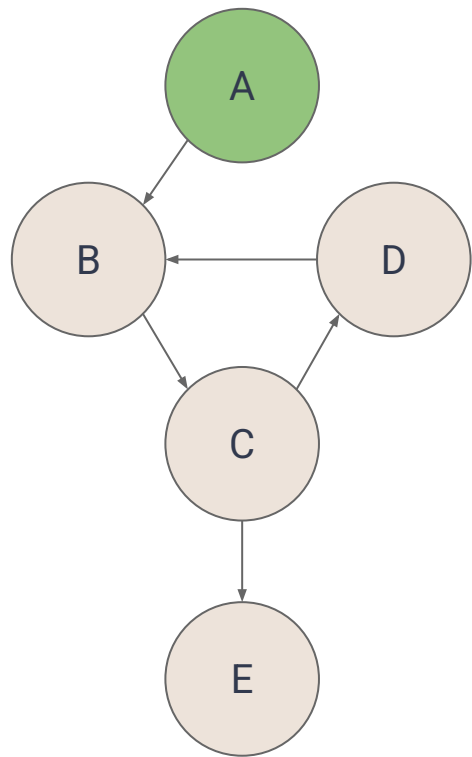
Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

A first useful insight: If we can identify a vertex in a sink, we can run DFS from there to find all vertices in that SCC.

But how do we find a sink...?

What happens if we DFS on a graph with cycles?

Insight 1: Find a sink!



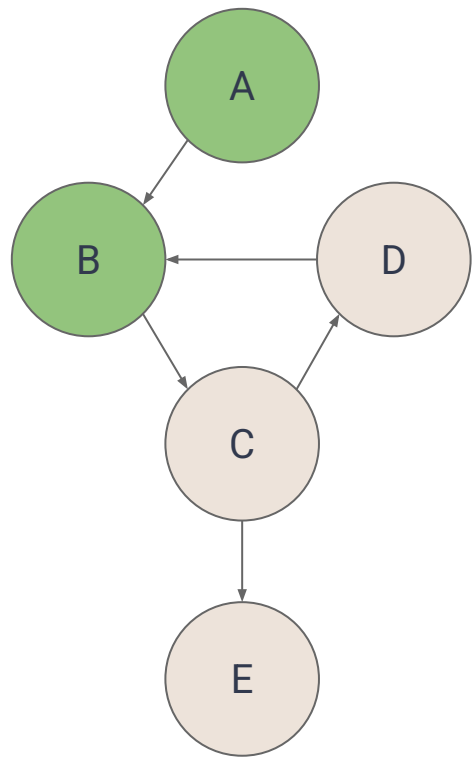
Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

A first useful insight: If we can identify a vertex in a sink, we can run DFS from there to find all vertices in that SCC.

But how do we find a sink...?

What happens if we DFS on a graph with cycles?

Insight 1: Find a sink!



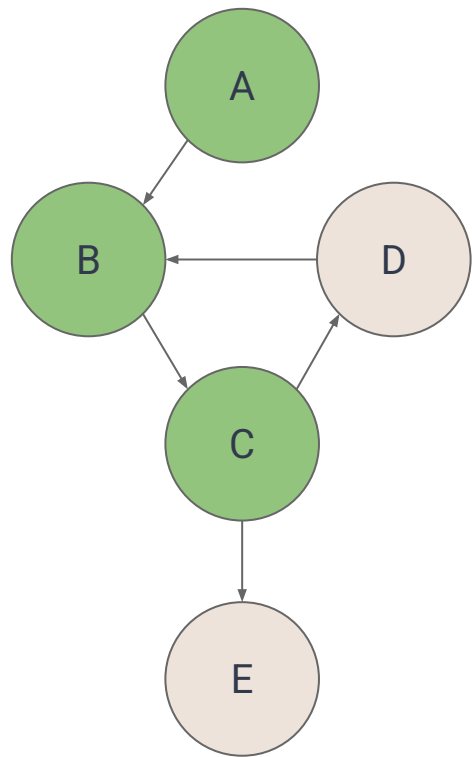
Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

A first useful insight: If we can identify a vertex in a sink, we can run DFS from there to find all vertices in that SCC.

But how do we find a sink...?

What happens if we DFS on a graph with cycles?

Insight 1: Find a sink!



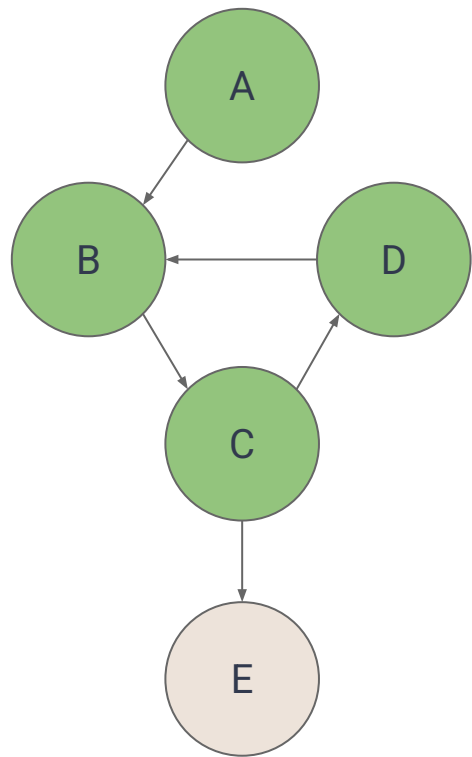
Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

A first useful insight: If we can identify a vertex in a sink, we can run DFS from there to find all vertices in that SCC.

But how do we find a sink...?

What happens if we DFS on a graph with cycles?

Insight 1: Find a sink!



Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

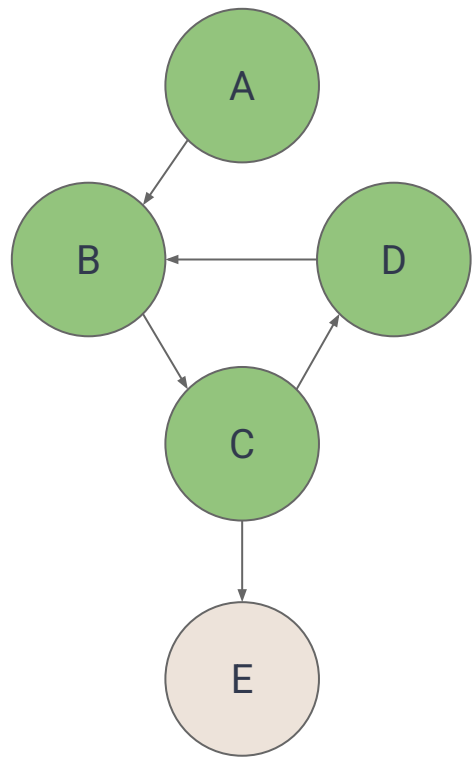
A first useful insight: If we can identify a vertex in a sink, we can run DFS from there to find all vertices in that SCC.

But how do we find a sink...?

What happens if we DFS on a graph with cycles?

D has post-order 1 here, despite not being in a sink!

Insight 1: Find a sink!



Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

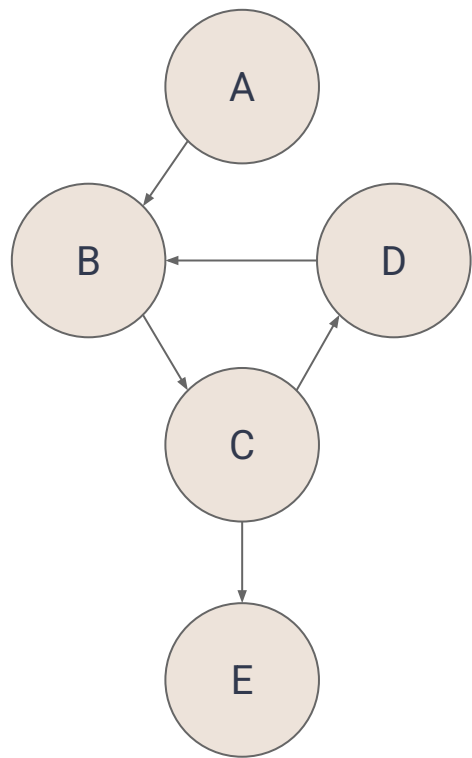
A first useful insight: If we can identify a vertex in a sink, we can run DFS from there to find all vertices in that SCC.

But how do we find a sink...?

What happens if we DFS on a graph with cycles?

But, a source always has highest post-order! Why?

Using DFS to find Sinks



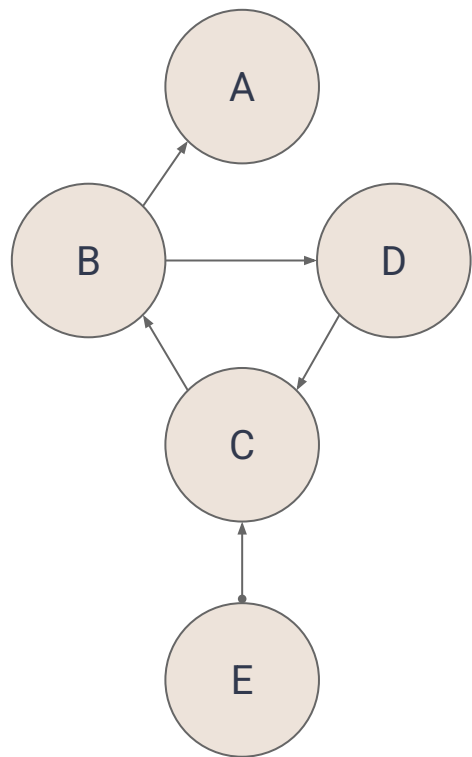
Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

Insight 1: If we can identify a vertex in a sink, we can run DFS from there to find all vertices in that SCC.

Insight 2: If we run DFS, a source will always have highest post-order.

How do we use that to find a sink?

Using DFS to find Sinks



Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

Insight 1: If we can identify a vertex in a sink, we can run DFS from there to find all vertices in that SCC.

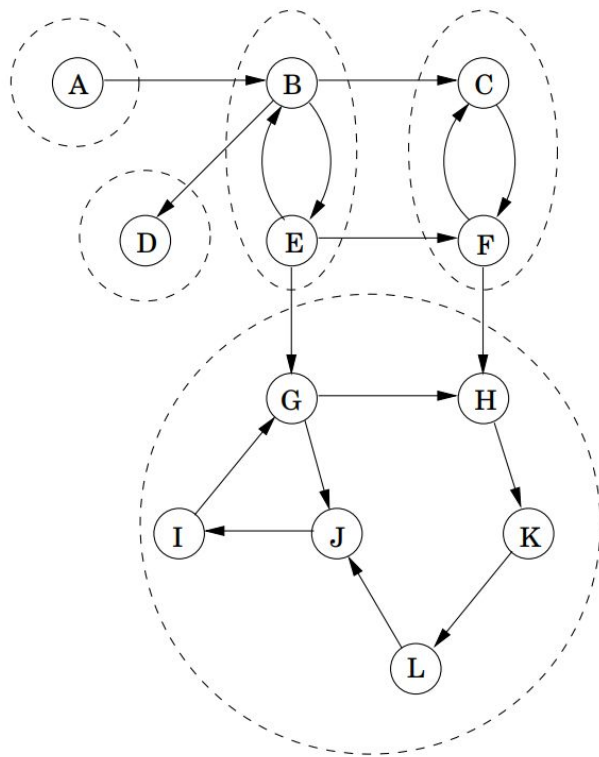
Insight 2: If we run DFS, a source will always have highest post-order.

How do we use that to find a sink?

If you reverse the graph, any sink becomes a source!

Plan: Reverse the graph and run DFS to find a source. These are sinks in the original graph.

Kosaraju's Algorithm



Problem: Given a directed graph, decompose it into a DAG of strongly connected components.

Step 1: Find an ordering that puts all sinks in front.

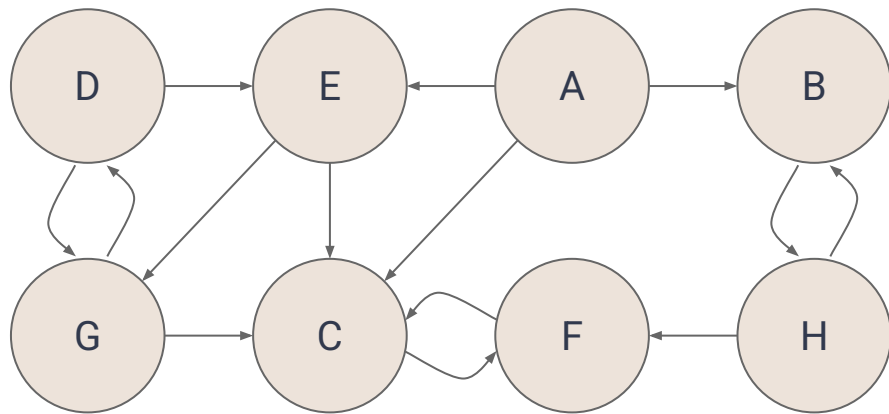
- Reverse all edges in the graph.
- Run DFS, keeping track of post-order.
 - DFS post-order gives sources the highest post-order. DFS on the reversed graph will give sinks in the original graph will get the highest post-order.

Step 2: Using the ordering, identify the SCCs one at a time.

- Take the vertex with highest post-order and run DFS from there to find its SCC.
- Once that SCC is found, remove it from the graph. Take the next vertex with highest post-order and repeat until all SCCs are found.

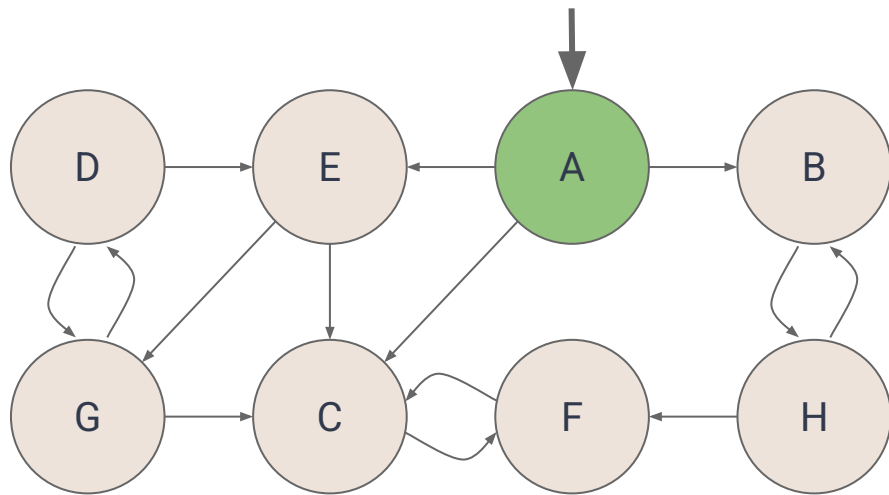
Kosaraju's Algorithm (Worksheet)

Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



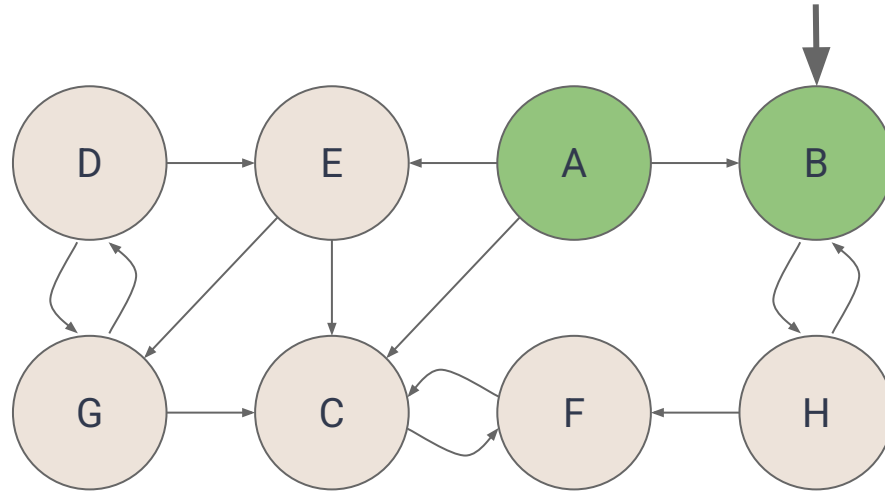
Kosaraju's Algorithm (Worksheet)

Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



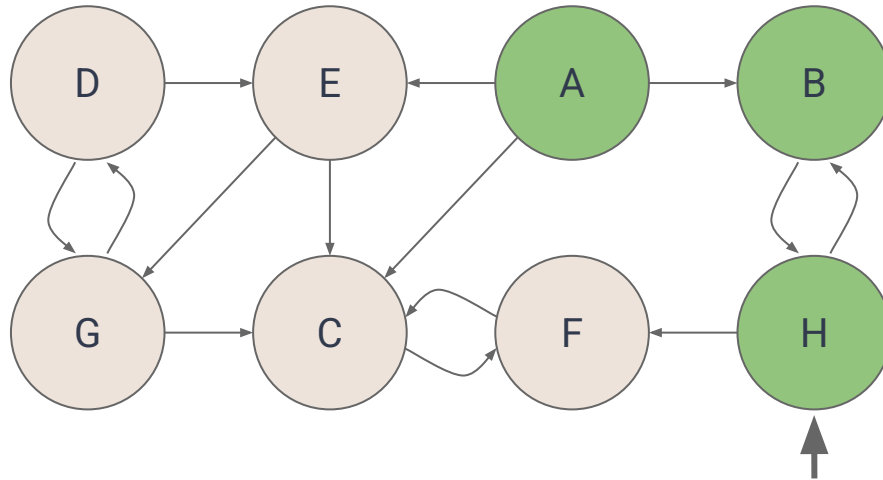
Kosaraju's Algorithm (Worksheet)

Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



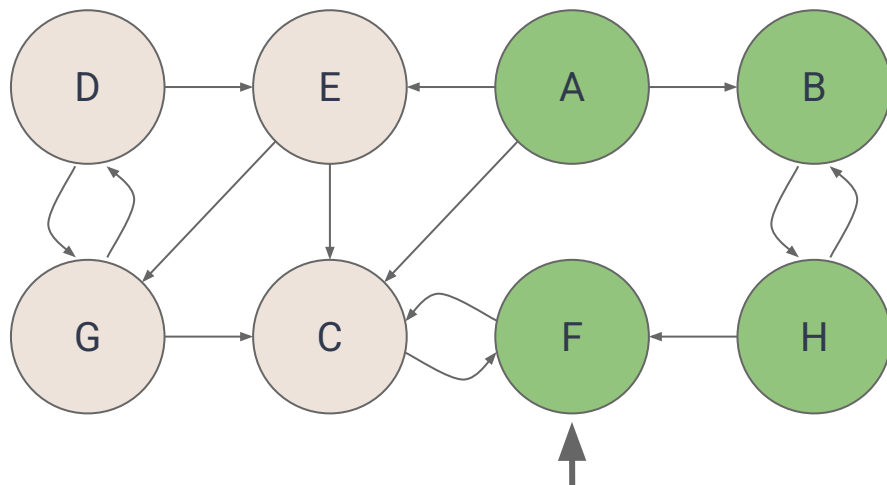
Kosaraju's Algorithm (Worksheet)

Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



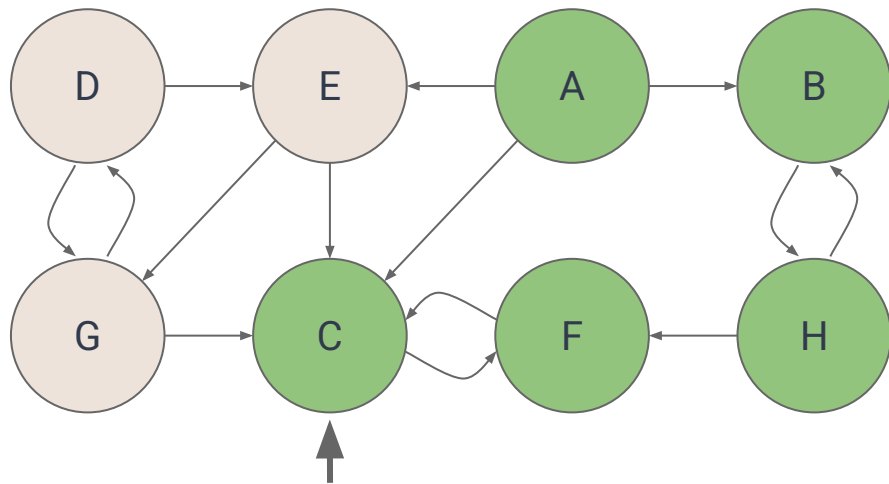
Kosaraju's Algorithm (Worksheet)

Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



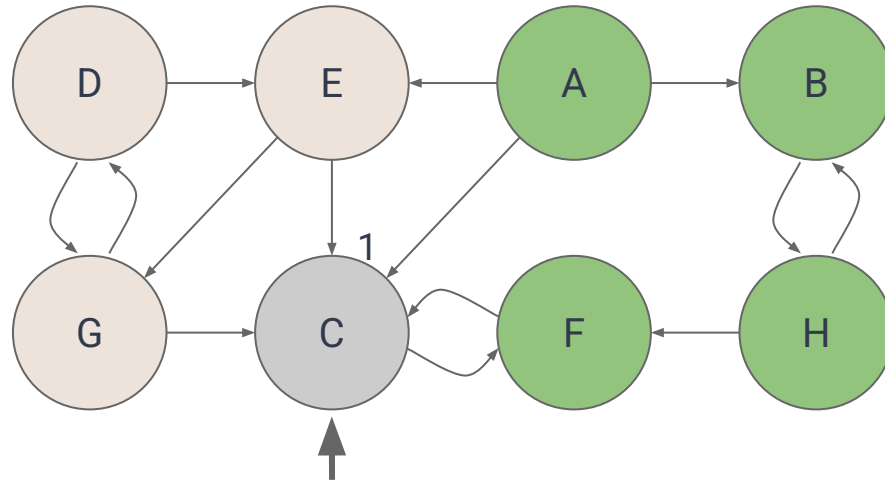
Kosaraju's Algorithm (Worksheet)

Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



Kosaraju's Algorithm (Worksheet)

Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.

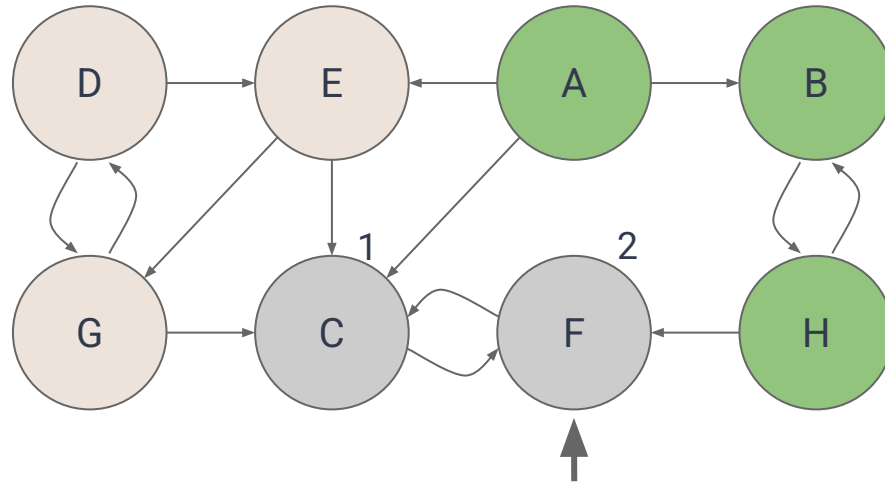


LIST OF VERTICES

C

Kosaraju's Algorithm (Worksheet)

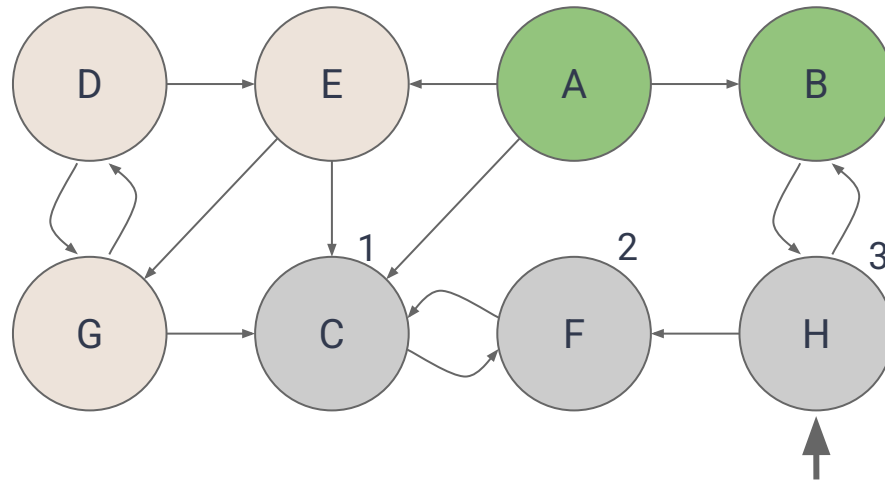
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CF

Kosaraju's Algorithm (Worksheet)

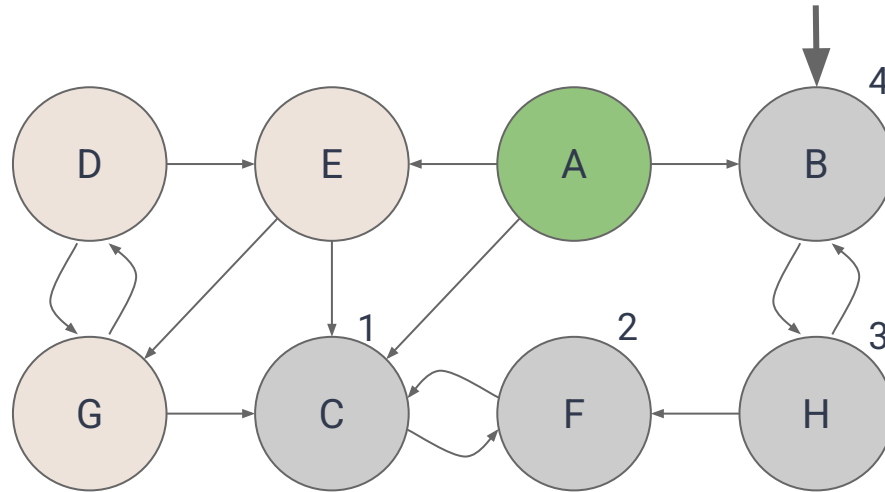
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFH

Kosaraju's Algorithm (Worksheet)

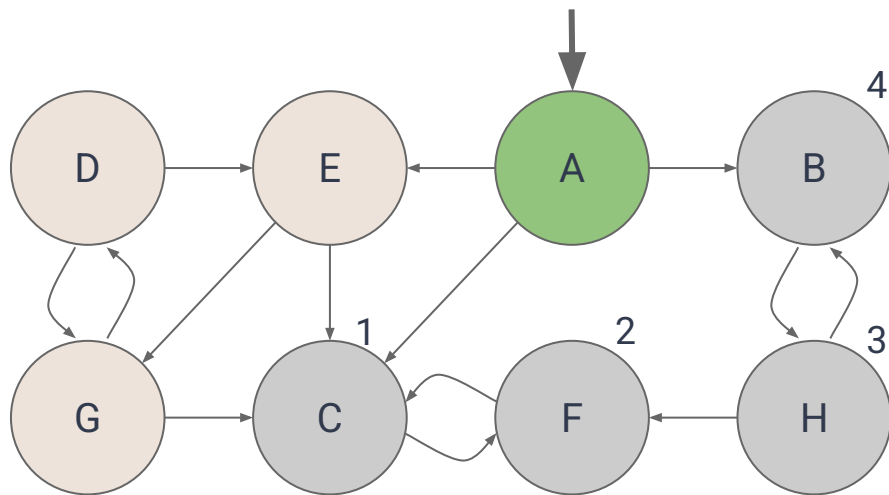
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFHB

Kosaraju's Algorithm (Worksheet)

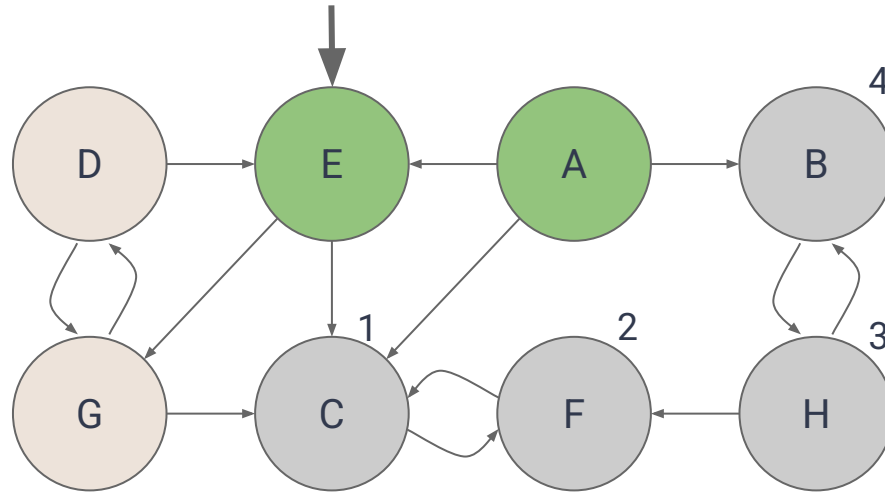
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFHB

Kosaraju's Algorithm (Worksheet)

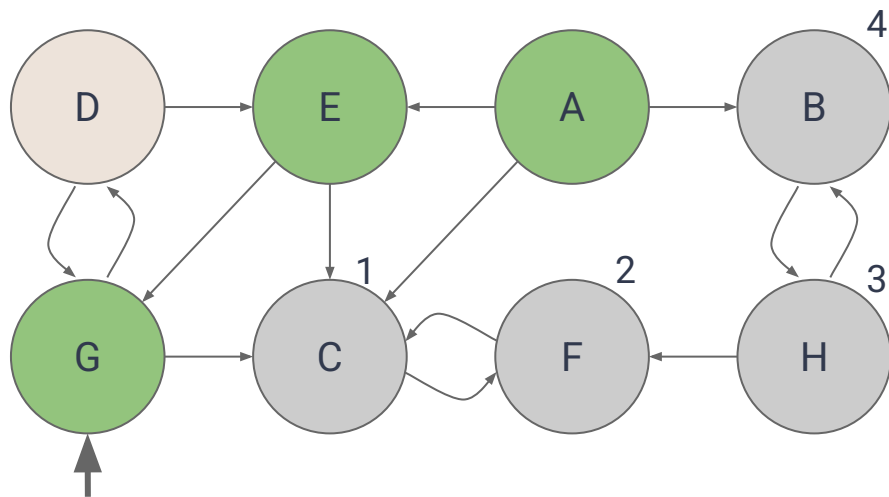
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFHB

Kosaraju's Algorithm (Worksheet)

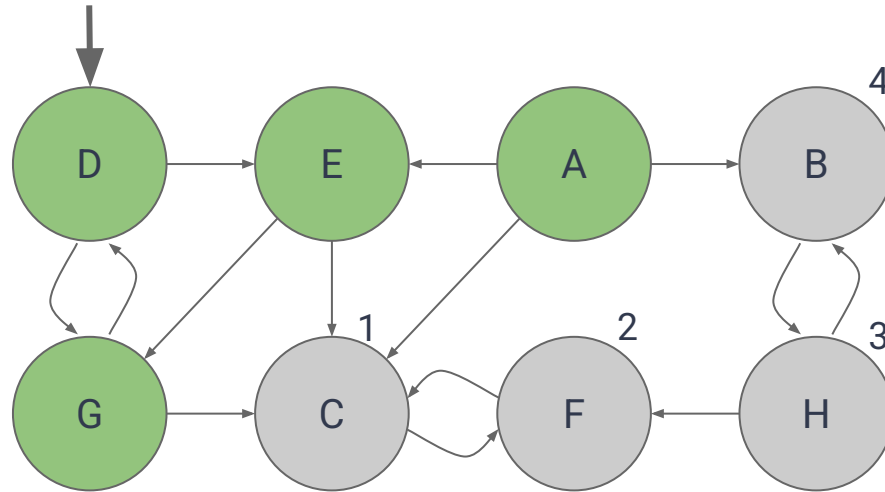
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFHB

Kosaraju's Algorithm (Worksheet)

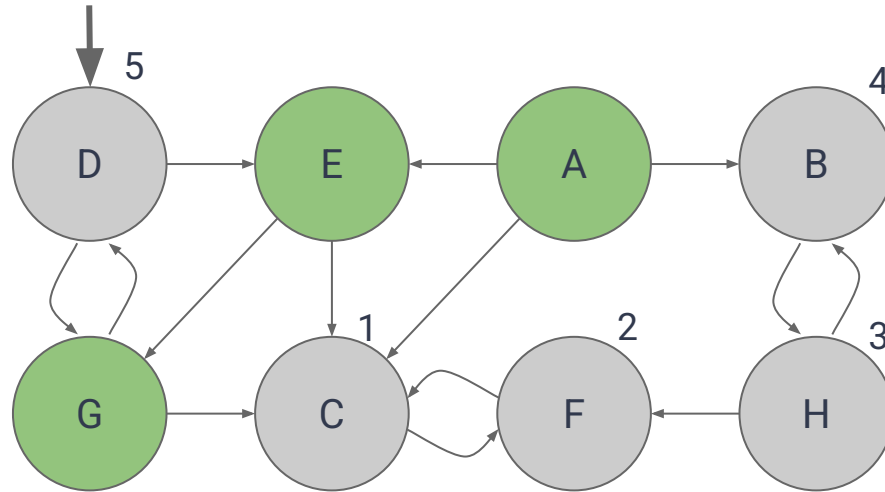
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFHB

Kosaraju's Algorithm (Worksheet)

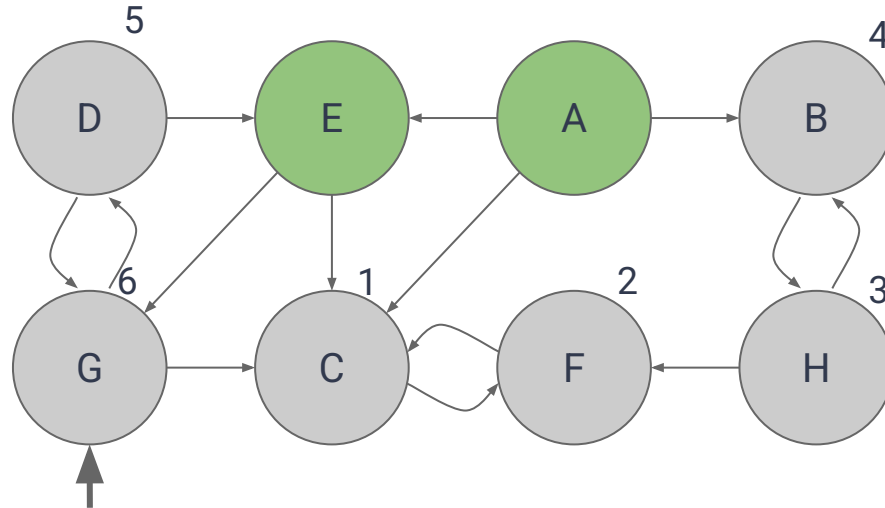
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFHBD

Kosaraju's Algorithm (Worksheet)

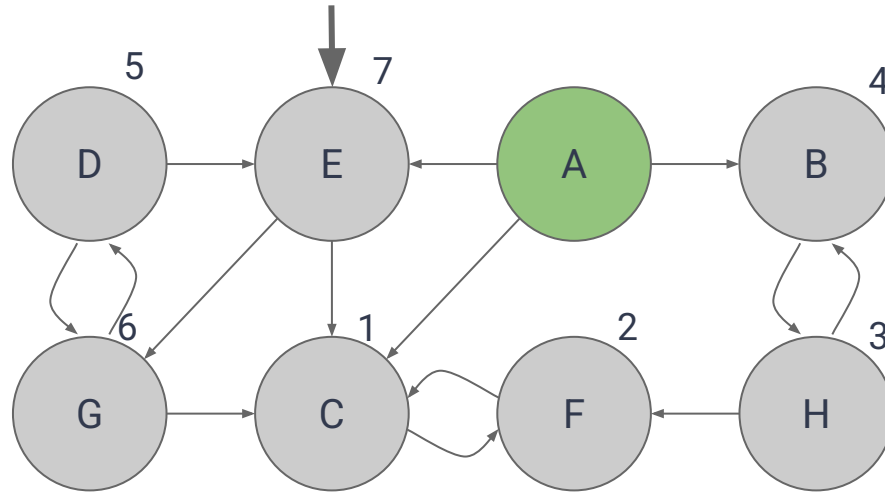
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFHBDG

Kosaraju's Algorithm (Worksheet)

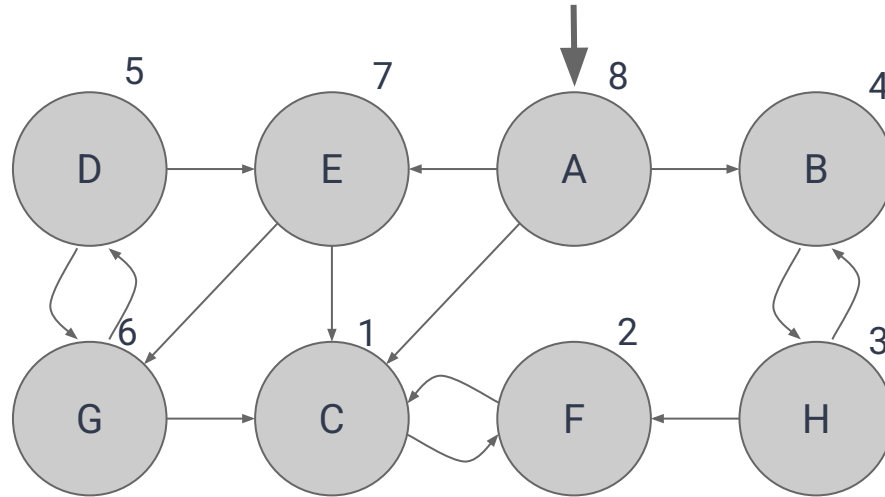
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFHBDGE

Kosaraju's Algorithm (Worksheet)

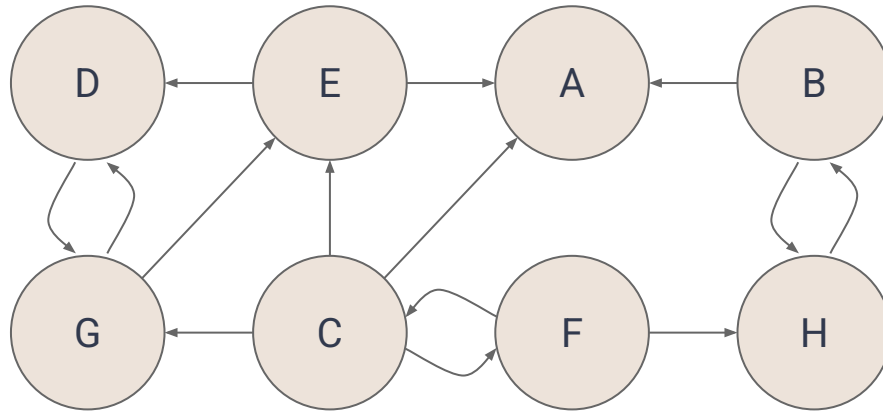
Step 1: Draw the graph with the edges reversed, and use it to run DFS, starting from A and breaking ties alphabetically.



LIST OF VERTICES
CFHBDGEA

Kosaraju's Algorithm (Worksheet)

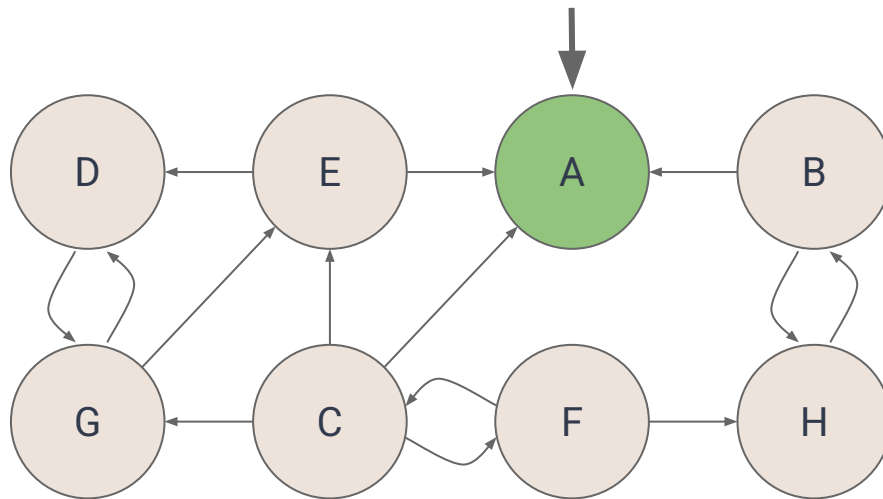
Step 2: Run DFS on the original graph, starting from the last vertex in the list. When the DFS pass ends, circle all vertices in that component.



LIST OF VERTICES
C F H B D G E A

Kosaraju's Algorithm (Worksheet)

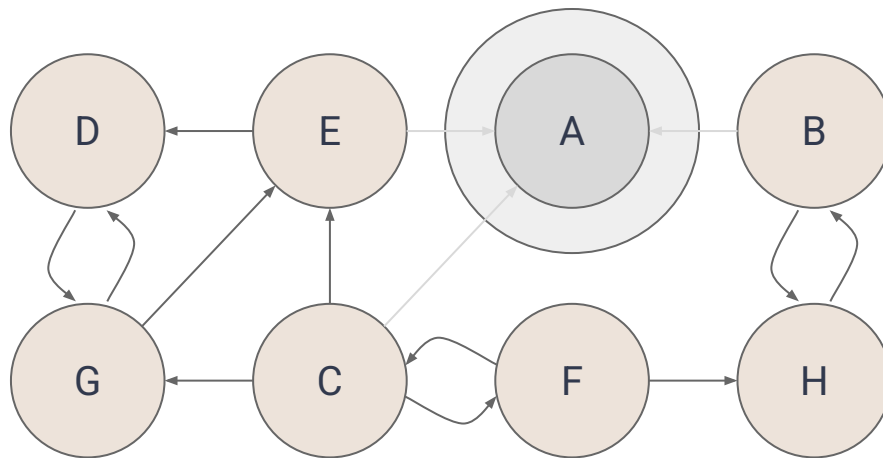
Step 2: Run DFS on the original graph, starting from the last vertex in the list. When the DFS pass ends, circle all vertices in that component.



LIST OF VERTICES
C F H B D G E A

Kosaraju's Algorithm (Worksheet)

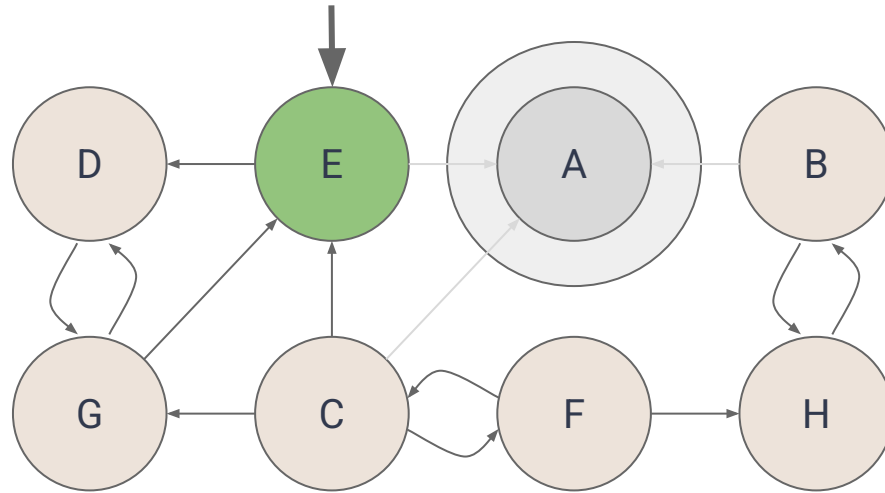
Step 2: Run DFS on the original graph, starting from the last vertex in the list. When the DFS pass ends, circle all vertices in that component.



LIST OF VERTICES
C F H B D G E A

Kosaraju's Algorithm (Worksheet)

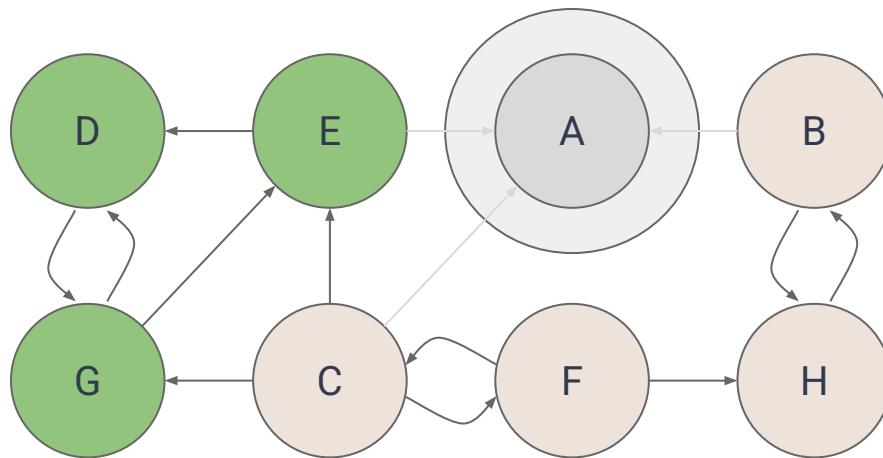
Step 2: Run DFS on the original graph, starting from the last vertex in the list. When the DFS pass ends, circle all vertices in that component.



LIST OF VERTICES
C F H B D G E A

Kosaraju's Algorithm (Worksheet)

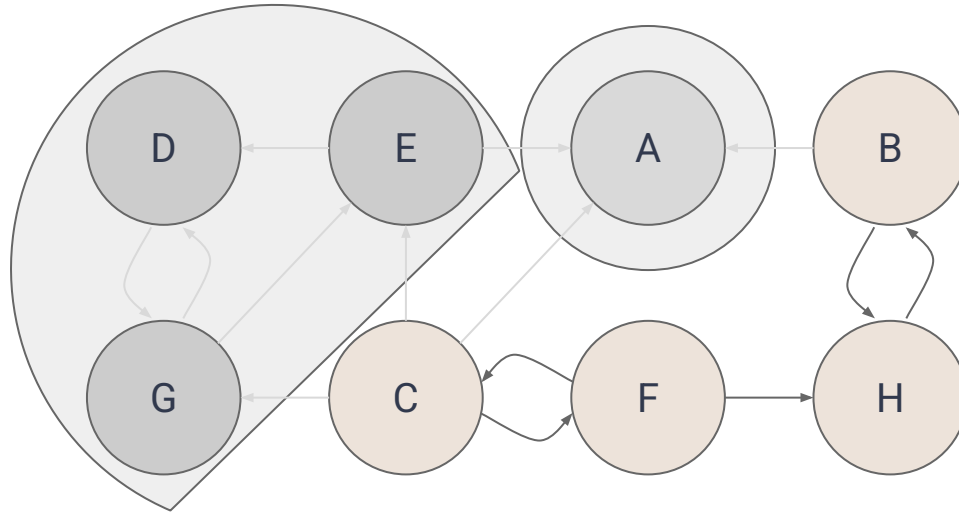
Step 2: Run DFS on the original graph, starting from the last vertex in the list. When the DFS pass ends, circle all vertices in that component.



LIST OF VERTICES
C F H B D G E A

Kosaraju's Algorithm (Worksheet)

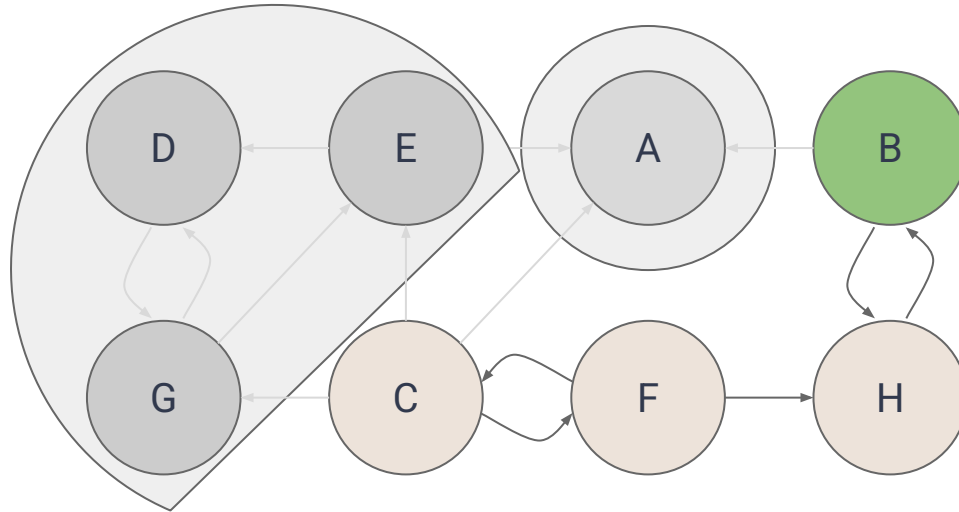
Step 2: Run DFS on the original graph, starting from the last vertex in the list. When the DFS pass ends, circle all vertices in that component.



LIST OF VERTICES
C F H B ~~D~~ ~~G~~ E A

Kosaraju's Algorithm (Worksheet)

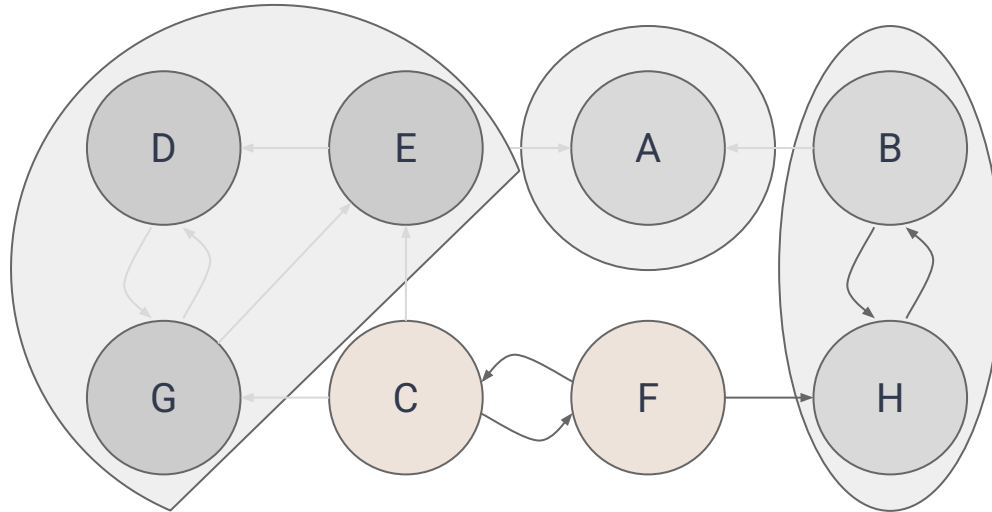
Step 2: Run DFS on the original graph, starting from the last vertex in the list. When the DFS pass ends, circle all vertices in that component.



LIST OF VERTICES
C F H **B** ~~D~~ ~~G~~ E A

Kosaraju's Algorithm (Worksheet)

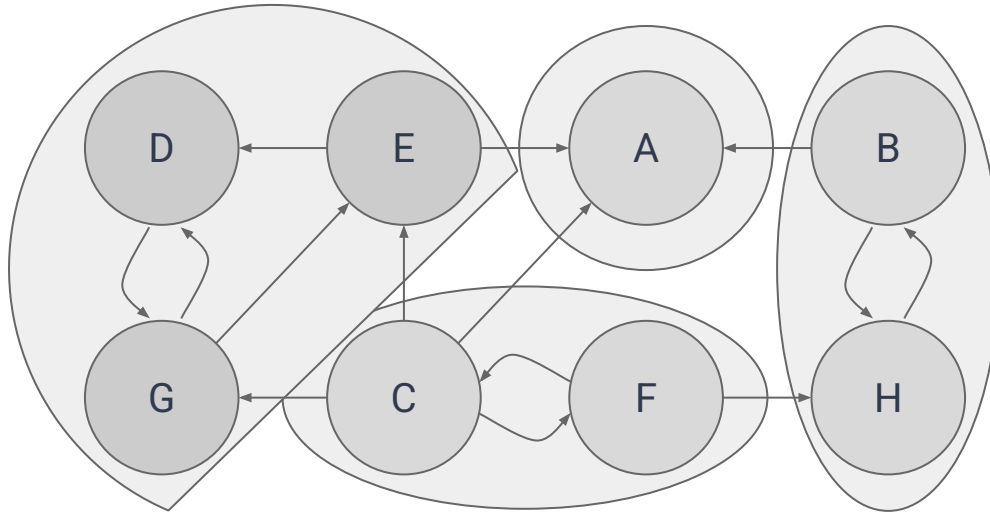
Step 2: Run DFS on the original graph, starting from the last vertex in the list. When the DFS pass ends, circle all vertices in that component.



LIST OF VERTICES
C F H ~~B~~ ~~D~~ ~~G~~ E A

Kosaraju's Algorithm (Worksheet)

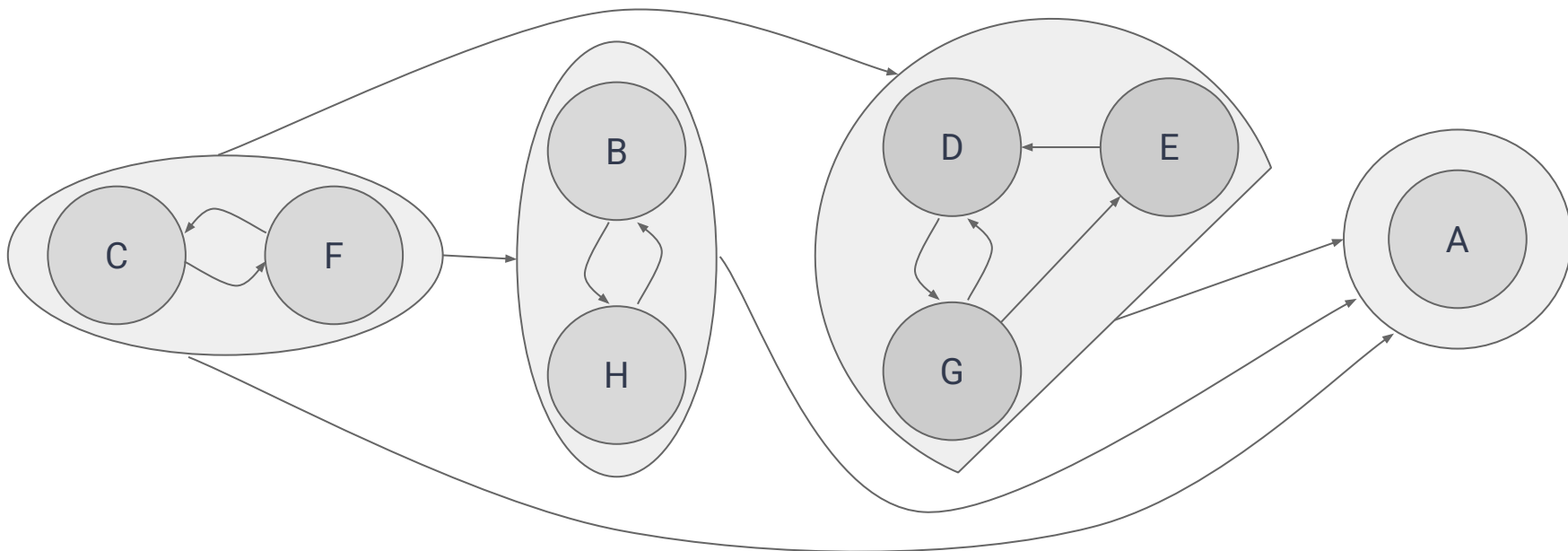
Step 3: Draw a linearized DAG of the graph, where each vertex in the DAG is a strongly connected component in the original graph.



LIST OF VERTICES
~~G F H B D G E A~~

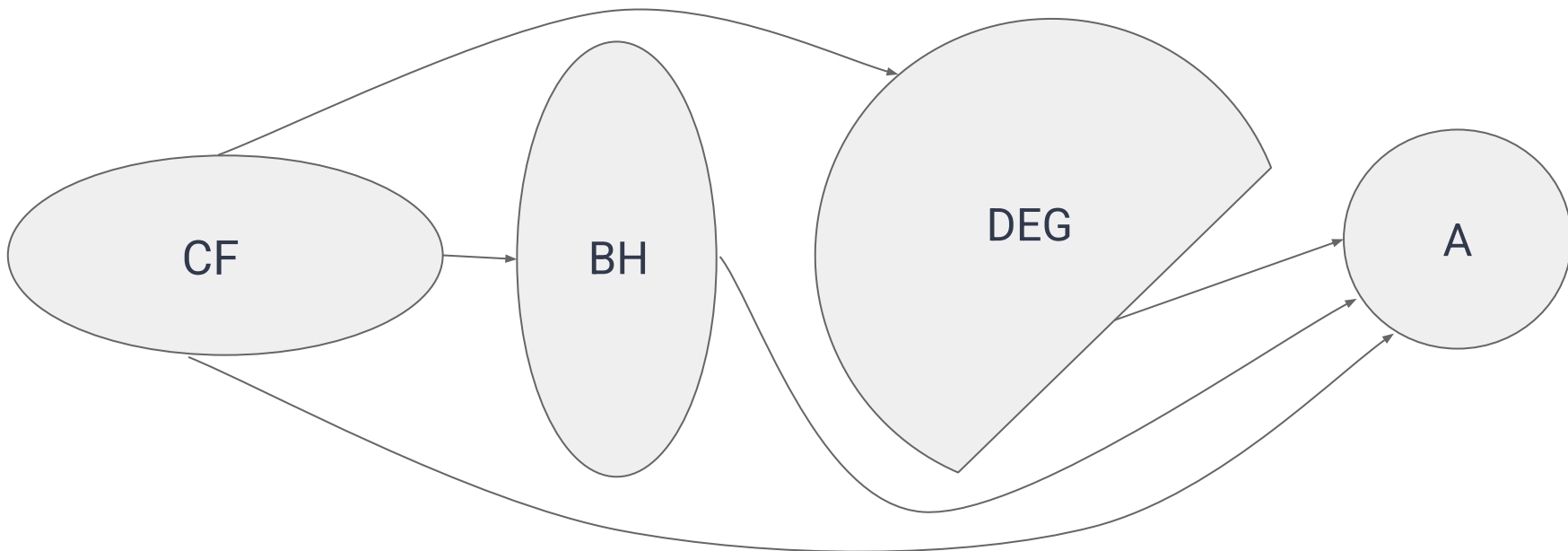
Kosaraju's Algorithm (Worksheet)

Step 3: Draw a linearized DAG of the graph, where each vertex in the DAG is a strongly connected component in the original graph.



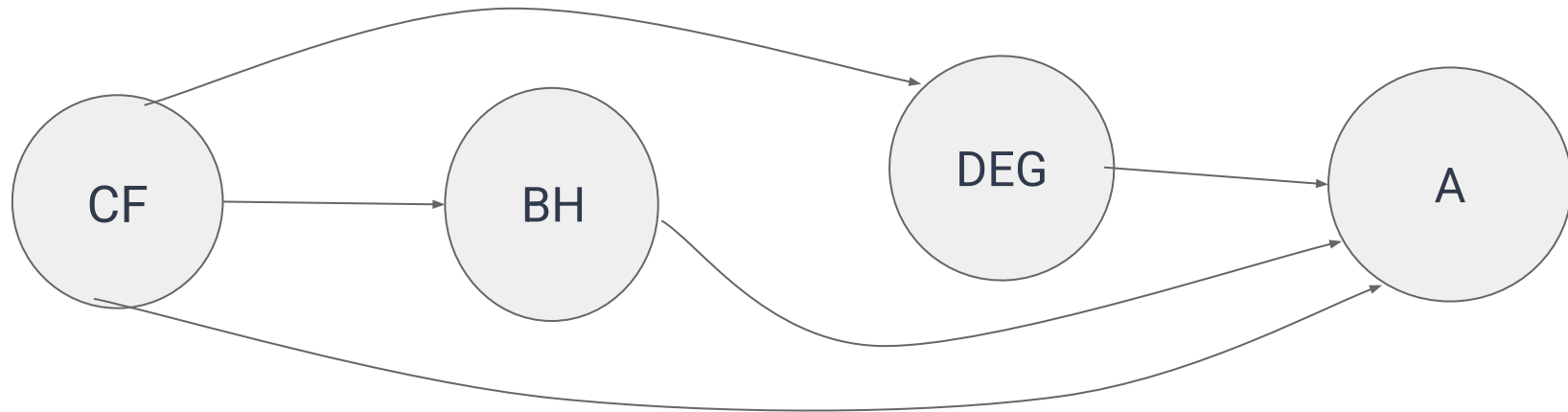
Kosaraju's Algorithm (Worksheet)

Step 3: Draw a linearized DAG of the graph, where each vertex in the DAG is a strongly connected component in the original graph.



Kosaraju's Algorithm (Worksheet)

Step 3: Draw a linearized DAG of the graph, where each vertex in the DAG is a strongly connected component in the original graph.



Summary

Today we were introduced to DFS, and saw a surprising application for it beyond just graph traversal: decomposing a graph into its strongly connected components.

This adds to the lesson from Tuesday, that the standard graph algorithms can do much more interesting tasks than they were designed for.

We also learned about pre-ordering and post-ordering, two useful metrics for graph traversal that help us understand the graph's structure better.