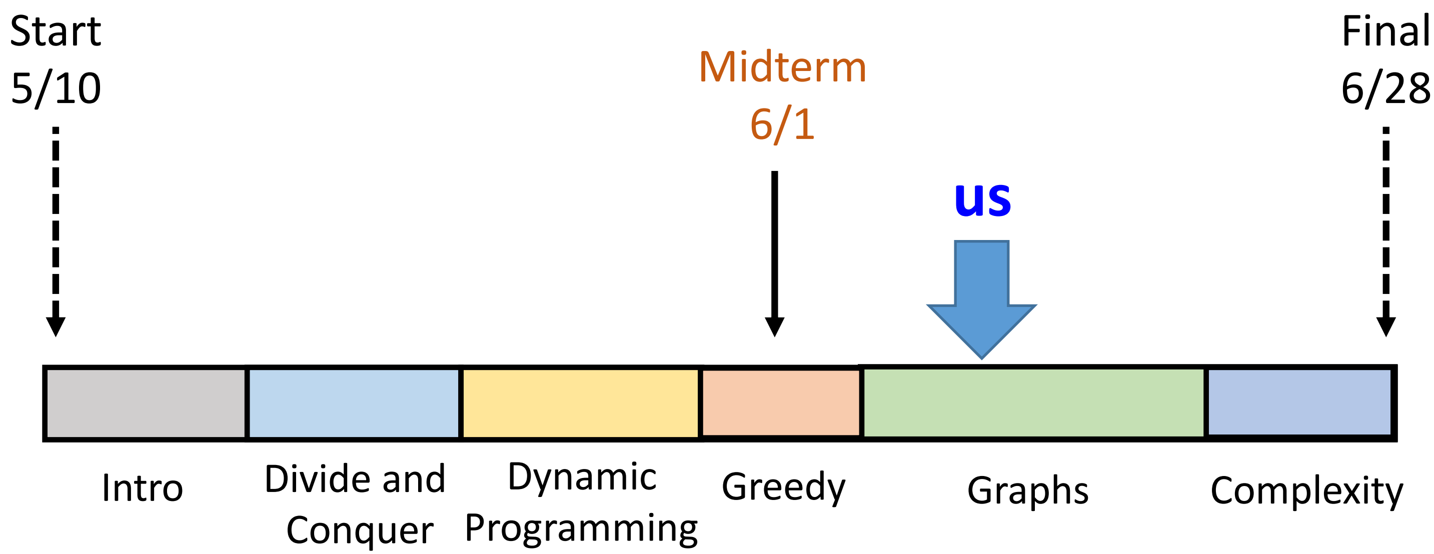
CS3000: Algorithms & Data
Drew van der Poel

Lecture 16

- Topological Orderings
- Strongly Connected Components

June 8, 2021

Outline



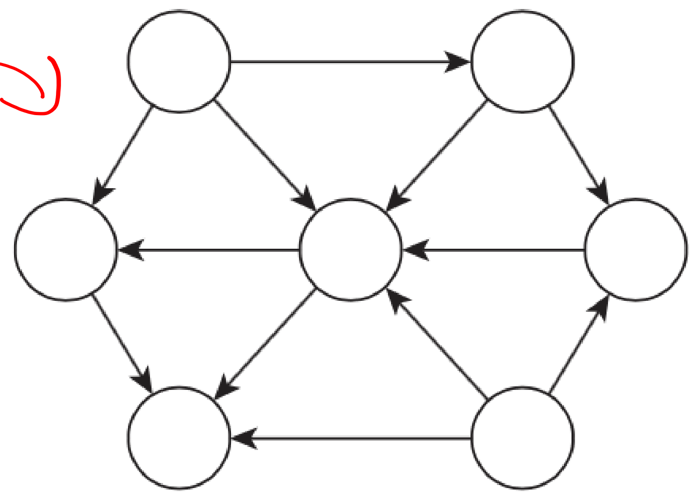
Last class: Graphs: DFS, Bipartiteness, Topological Orderings

Next class: Graphs: Dijkstra's

Directed Acyclic Graphs (DAGs)

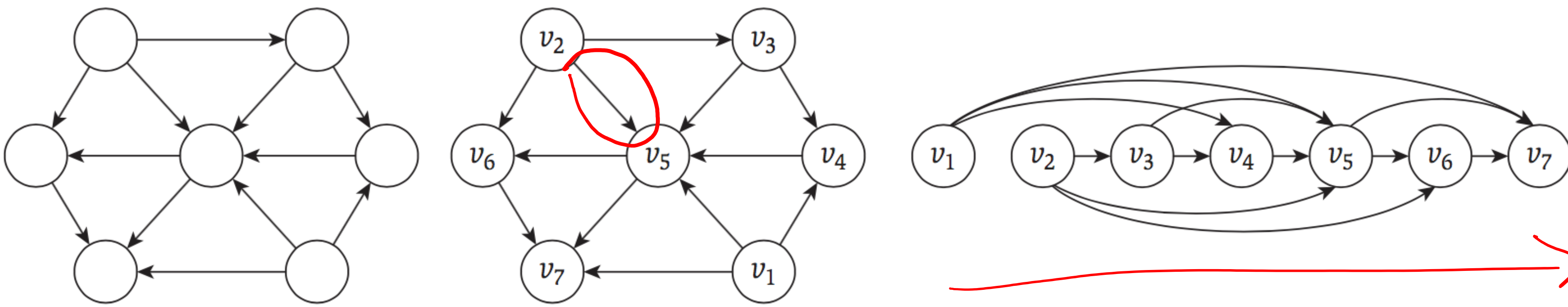
- **DAG:** A directed graph with no directed cycles
- Can be much more complex than a forest

DAG

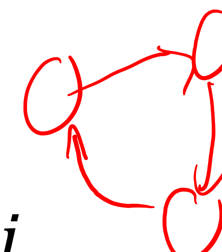


Directed Acyclic Graphs (DAGs)

- **DAG:** A directed graph with no directed cycles
- DAGs represent precedence relationships



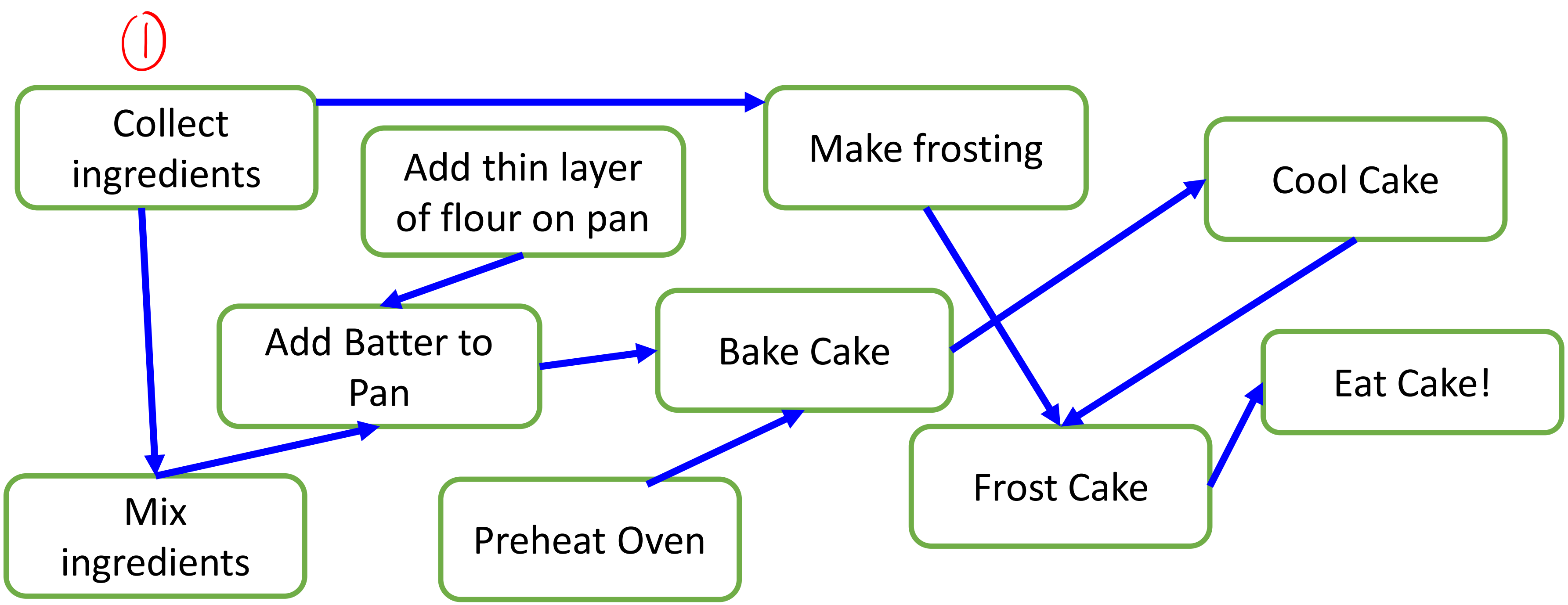
- A topological ordering of a directed graph is a labeling of the nodes from $v_1, \ldots, v_n$ so that all edges go "forwards", that is $(v_i, v_j) \in E \Rightarrow j > i$
  - G has a topological ordering $\Rightarrow$ G is a DAG



**G cannot be top. ordered if __it has a directed cycle__

Directed Acyclic Graphs (DAGs)

- **DAG:** A directed graph with no directed cycles
- DAGs represent precedence relationships



- Problems: counting students, stable matching, sorting, n-digit mulitiplication, array searching, selection, weighted interval scheduling, segmented least squares, knapsack, prefix-free encoding, graph exploration, bipartiteness, **topological sorting**

- Alg. techniques: divide & conquer, dynamic programming, greedy

- Analysis: asymptotic analysis, recursion trees, Master Thm., Graph Terminology/representations

- Proof techniques: (strong) induction, contradiction, greedy stays ahead, exchange argument

Directed Acyclic Graphs (DAGs)

- **Problem 1:** given a digraph G, is it a DAG?
- **Problem 2:** given a digraph G, can it be topologically ordered?

- **For given *G*, the answers to P1 and P2 are:**
  - Always the same
  - Sometimes different

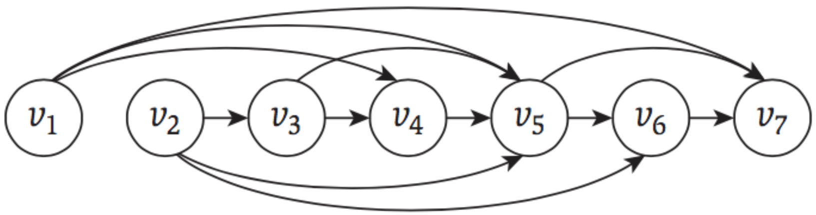
Directed Acyclic Graphs (DAGs)

- **Problem 1:** given a digraph G, is it a DAG?
- **Problem 2:** given a digraph G, can it be topologically ordered?

- **Thm:** G has a topological ordering $\Leftrightarrow$ G is a DAG
  - We will design one algorithm that either outputs a topological ordering or finds a directed cycle

Topological Ordering

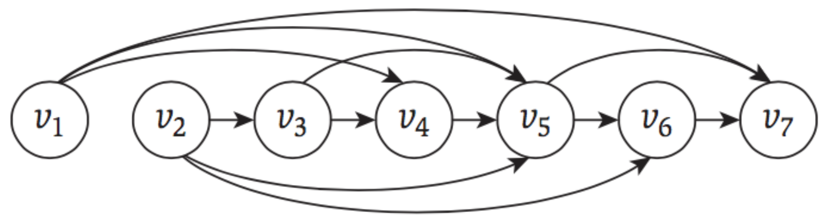
- **What can we say about the first node in the top. ordering?**



in-des of node 1 is always 0!

Topological Ordering

- **Observation:** the first node must have no in-edges



- **Observation:** In any DAG, there is always a node with no incoming edges

"proof by extremality"

Let P be a longest simple path in DAG

Claim: P_1 has in-deg $= 0$

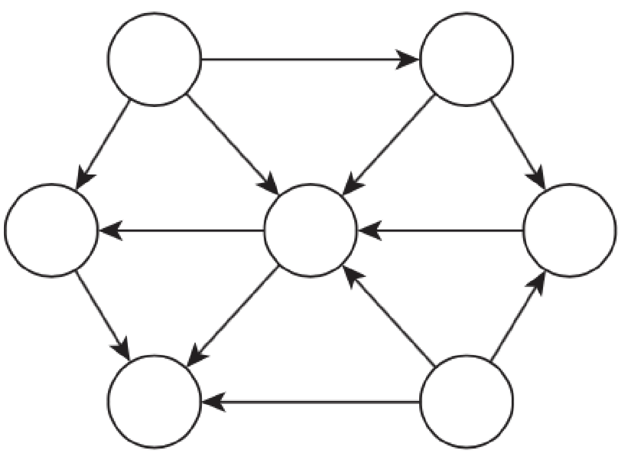
if $\exists\ (P_i, P_1) \in E\ (2 \le i \le |P|) \rightarrow$ cycle $\rightarrow\leftarrow$

$\square$ if $\exists\ (q, P_1) \in E\ \ q \notin P \rightarrow P$ is not longest $\rightarrow\leftarrow$

Topological Ordering

- **Fact:** In any DAG, there is a node with no incoming edges
- **Thm:** Every DAG has a topological ordering
- **Proof (Induction):**

H(k): every DAG w/ k nodes has a top. ordering

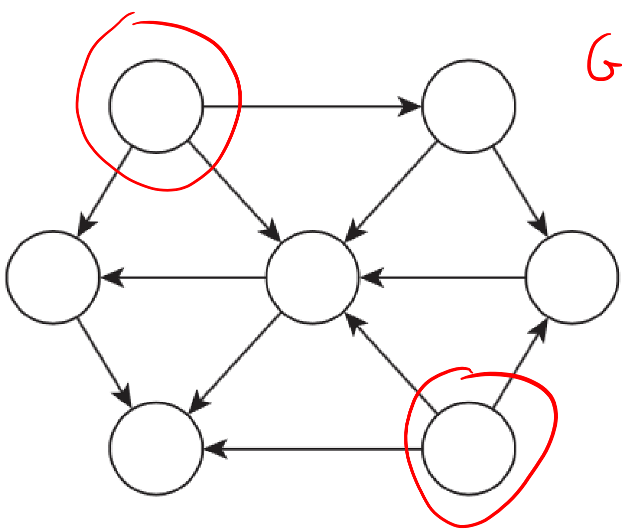


Base: H(1) - straightforward

1

Topological Ordering

- **Fact:** In any DAG, there is a node with no incoming edges
- **Thm:** Every DAG has a topological ordering
- **Proof (Induction):** Ind. $H(k-1) \rightarrow H(k)$

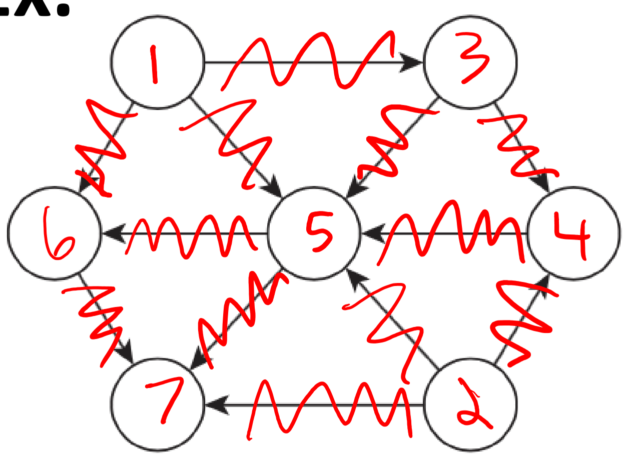


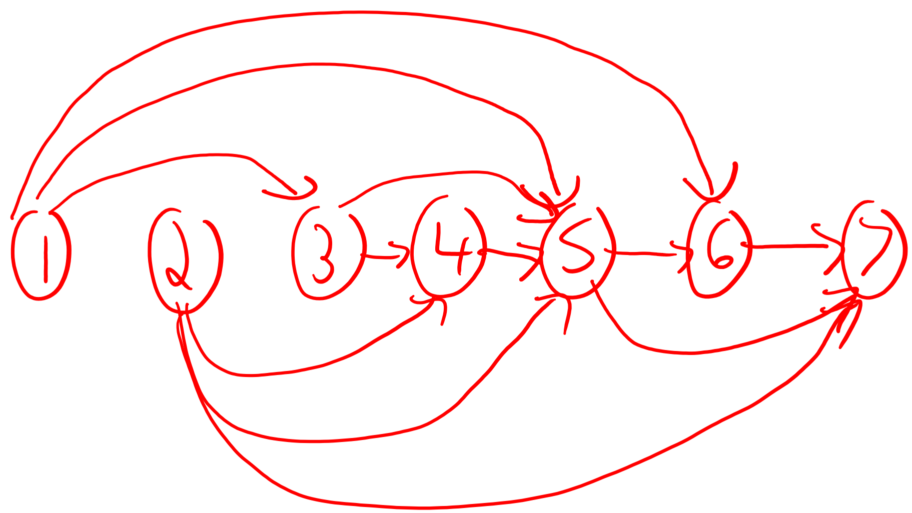
G is a DAG w/ k nodes. For any $v \in V$,
$G-v$ is still a DAG w/ $k-1$ nodes $\rightarrow$ $G-v$ has a TO.
Let v be a node w/ in-deg = 0.
Let T be the TO of $G-v$. (by IH)
Place v @ front of T $\rightarrow$ TO of G
$\square$

Implementing Topological Ordering

```
SimpleTopOrder(G):
  Set i ← 1
  Until (G has no unlabeled nodes):
    Find a node u with no incoming edges
    Label u as node i, increment i ← i+1
    Remove u's edges from G
```

Ex.





Implementing Topological Ordering

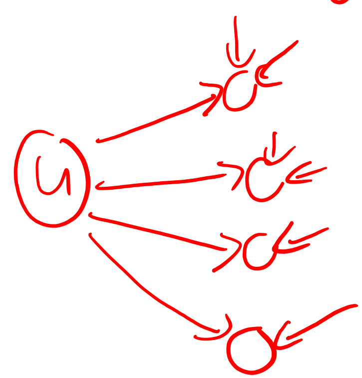
```
SimpleTopOrder(G):
O(1)  Set i  ← 1
      Until (G has no unlabeled nodes):        O(n) times
        Find a node u with no incoming edges   O(n)
        Label u as node i, increment i  ← i+1  O(1)
        Remove u's edges from G                O(m)
```

Runtime: (adj. list)

$$\sum_{v \in OUT[u]} \text{in-deg}[v] \leq m$$

$$n(n + 1 + m)$$

$$= O(n^2 + nm)$$



Fast Topological Ordering

```
DeleteNode(v):
  Label v as node i in the top. order
  i = i+1
  For every w in OUT-NEIGH[v]:
    Decrease w's mark by 1
  For every w in OUT-NEIGH[v]:
    If w's mark is 0:
      DeleteNode(w)

FastTopOrder(G):
  Mark all nodes with their # of in-edges
  Let i = 1        // i is a global variable

  Put all nodes w/ mark 0 in queue Q
  while Q is not empty:
    u <- Q.dequeue()
    DeleteNode(u)
```

$O(n)$ times in TOTAL

$O(1)$ each

each: $\text{out-deg}(v)$

total: $\sum_{v \in V} \text{out-deg}(v) = m = O(m)$

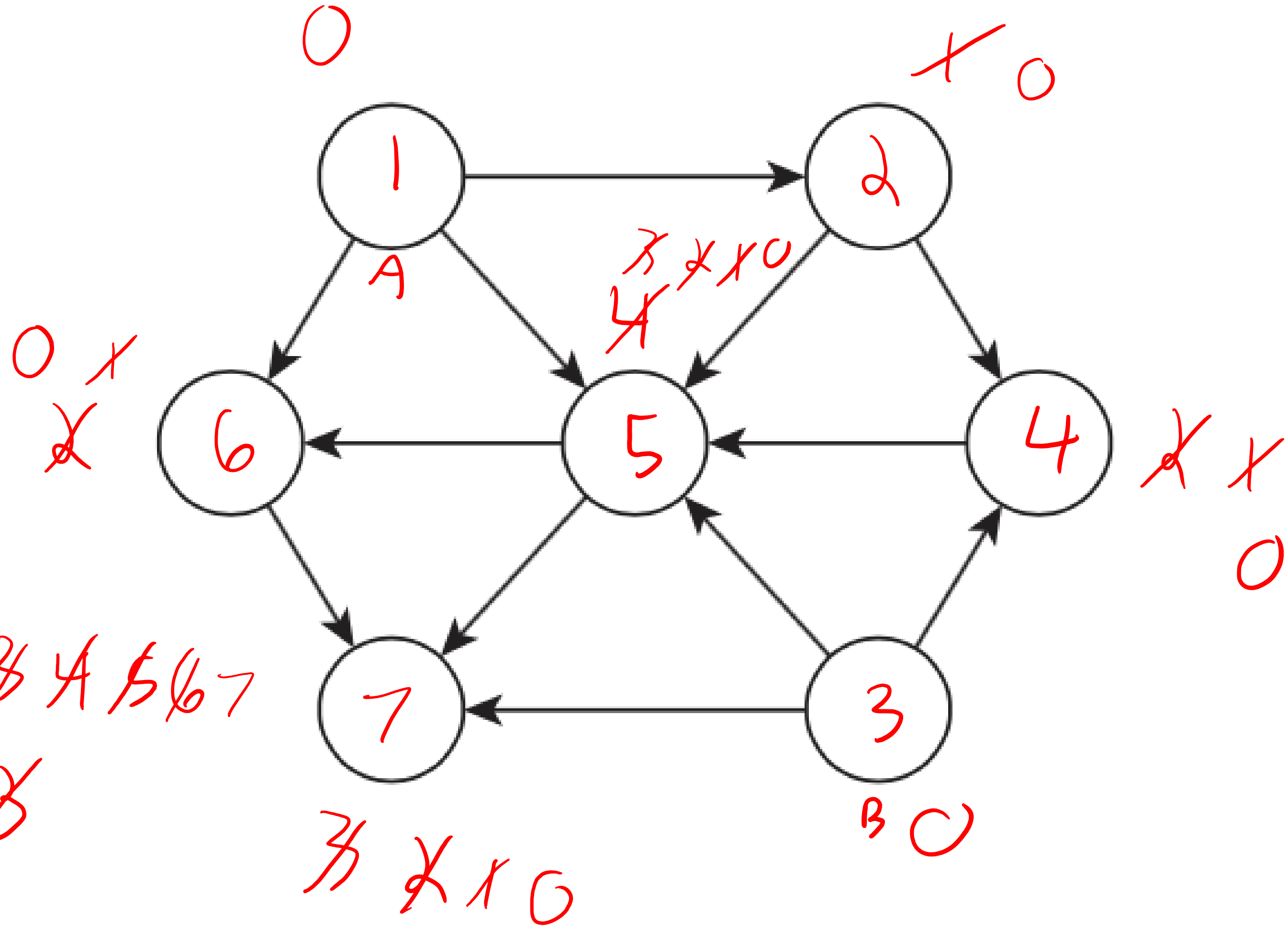
$O(m)$

$O(n)$

Total: $n + 3m$

$= O(n+m)$

Fast Topological Ordering Example

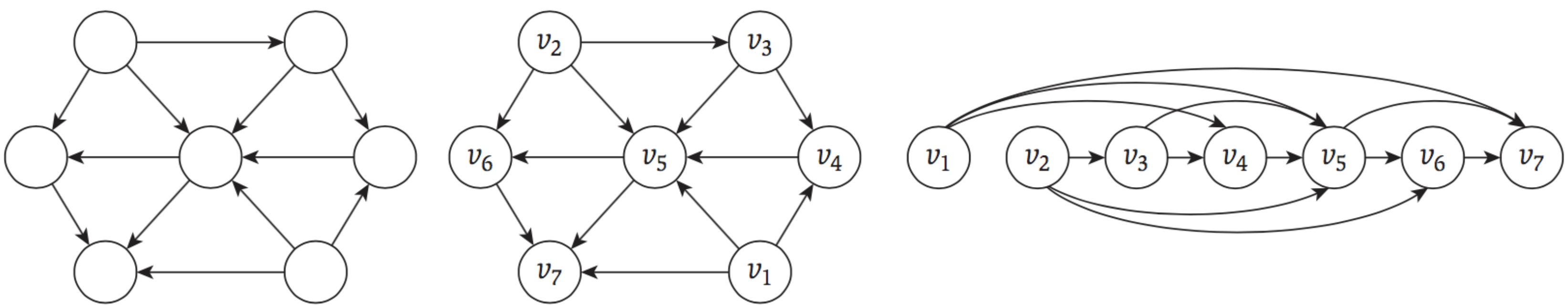


i = ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ 7

Q: ~~A~~, ~~B~~

Topological Ordering Summary

- **DAG:** A directed graph with no directed cycles
- Any DAG can be **topologically ordered**
  - There is an algorithm that either outputs a topological ordering or finds a directed cycle in time $O(n+m)$

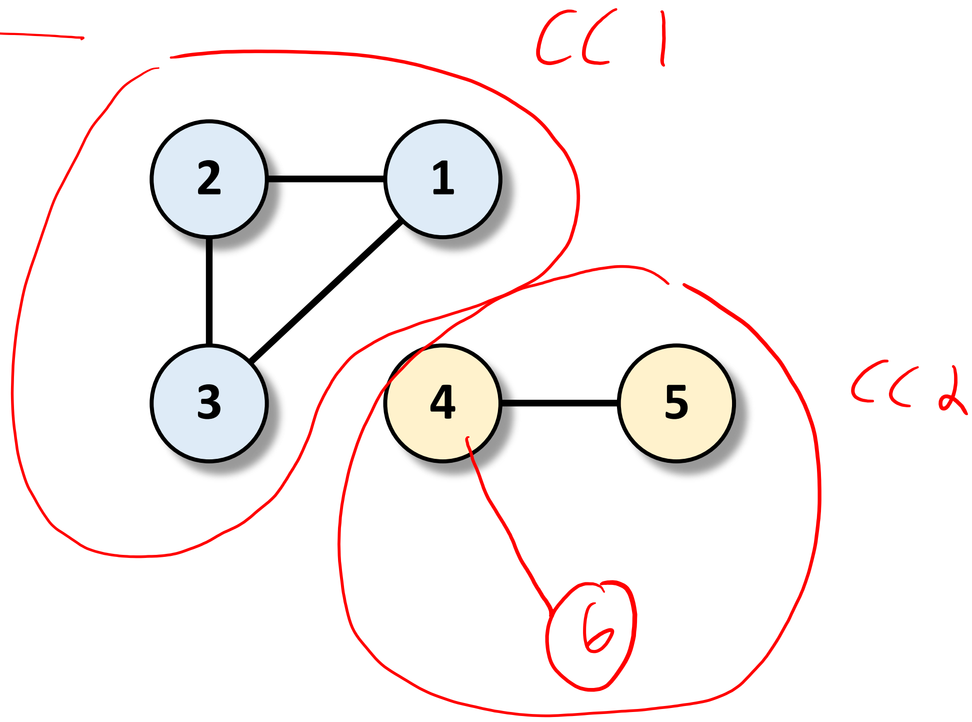


(Strongly) Connected Components

Connected Components

directed   undirected

- **(strongly) connected component:** a maximal subset of vertices which are all (strongly) connected in G

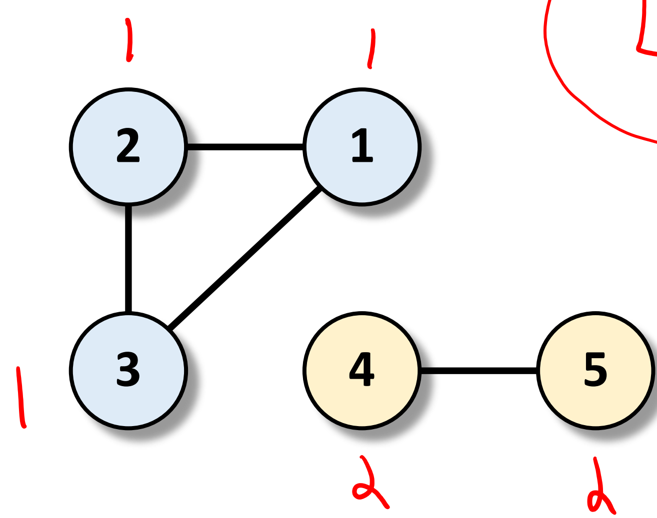


Connected Components (Undirected)

- **Problem:** Given an undirected graph G, split it into connected components
- **Input:** Undirected graph $G = (V, E)$
- **Output:** A labeling of the vertices by their connected component

OUTPUT:

1	1	1	2	2
1	2	3	4	5

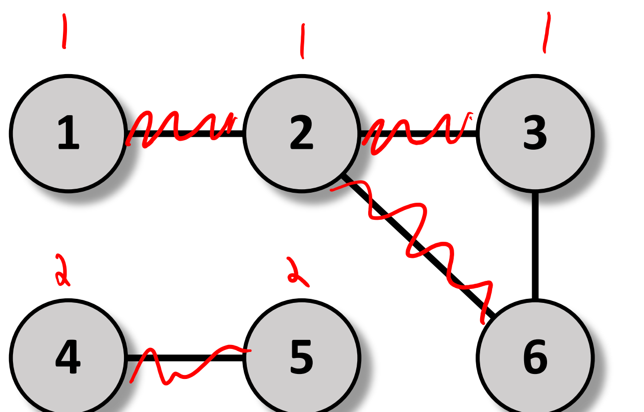


Connected Components (Undirected)

- **Algorithm:**
  - Pick a node v
  - Use DFS to find all nodes reachable from v
  - Labels those as one connected component
  - Repeat until all nodes are in some component

DFS(3)

DFS(4)



7 3

Connected Components (Undirected)

V, E

```
CC(G):
  // Initialize an empty array and a counter
  let comp[1:n] = ⊥, c = 1

  // Iterate through nodes
  for (u = 1,…,n):
    // Ignore this node if it already has a comp.
    // Otherwise, explore it using DFS
    if (comp[u] = ⊥):
      run DFS(G,u)
      let comp[v] = c for every v found by DFS
      let c = c + 1

  output comp[1:n]
```

Could modify DFS to include this

Running Time

TOTAL: O(n+m)

```
CC(G):
  let comp[1:n] = ⊥, c ← 1

  for (u = 1,…,n):
    if (comp[u] != ⊥):
      run DFS(G,u)
      let comp[v] = c for every v
          found by DFS
      let c = c + 1

  output comp[1:n]
```

← O(n)

← O(n)

each:

← O(# of nodes reachable from u + # of edges reachable from u)

$O(n_i + m_i)$ where i is u's CC

$\boxed{\text{TOTAL: } O(n+m)}$

when we run DFS:

$n_1 + m_1 + n_2 + m_2 + \ldots = n+m$

each CC DFSed exactly once

Connected Components (Undirected)

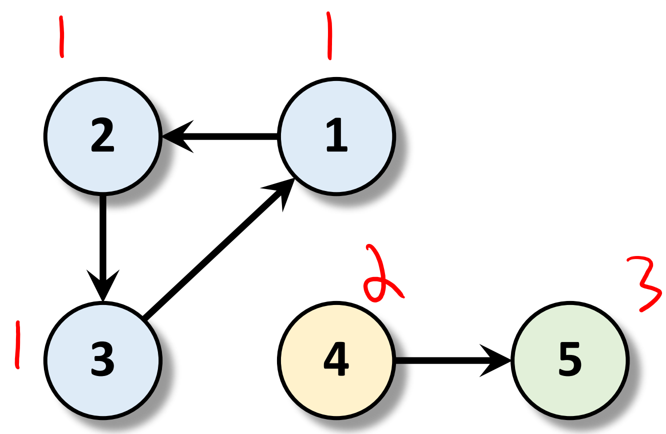
- **Problem:** Given an undirected graph G, split it into connected components
- **Algorithm:** Can split a graph into connected components in time $\Theta(n+m)$ using DFS
- **Punchline:** Usually assume graphs are connected
    - Implicitly assume that we have already broken the graph into CCs in $\Theta(n+m)$ time

Strongly Connected Components

- **Problem:** Given a directed graph G, split it into strongly connected components
- **Input:** Directed graph $G = (V, E)$
- **Output:** A labeling of the vertices by their strongly connected component

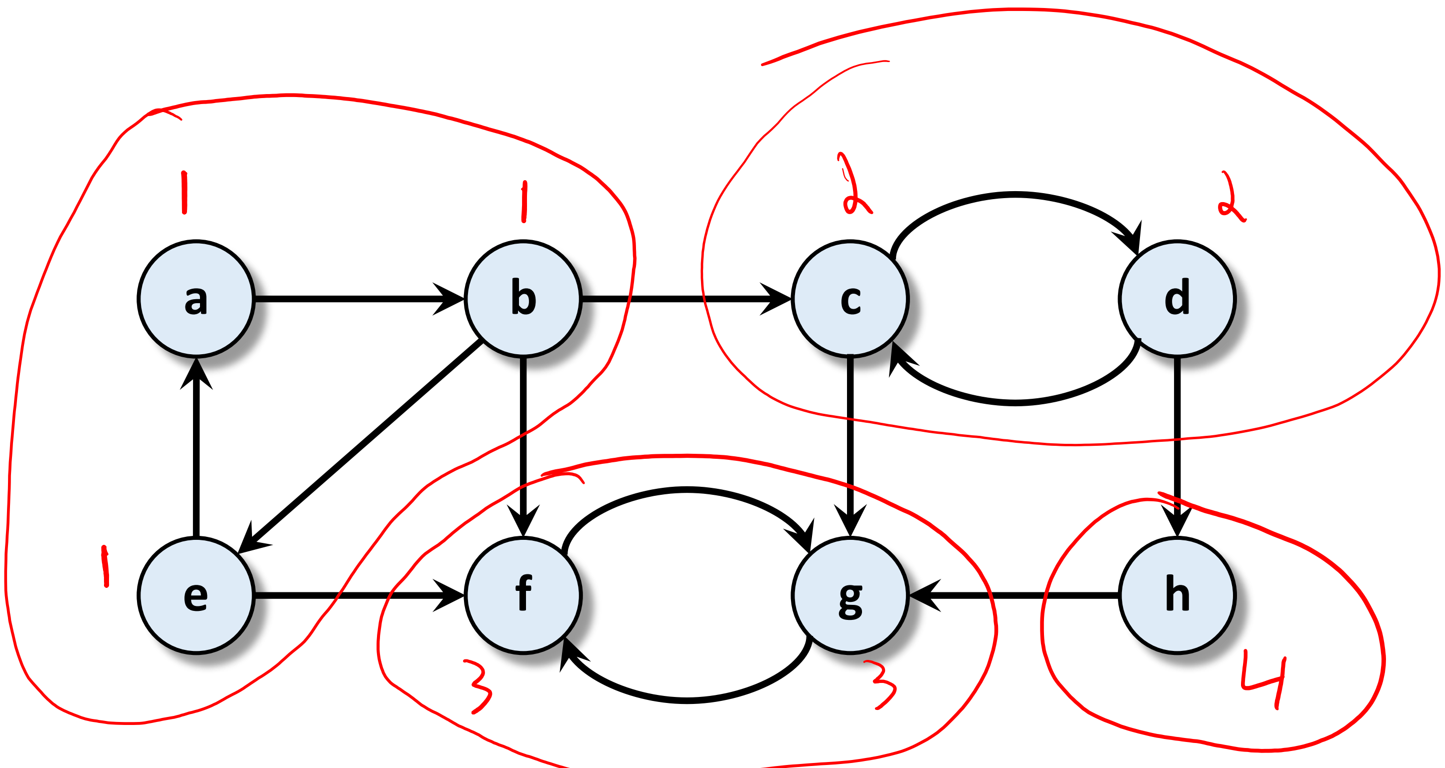
u and v are strongly connected if $u \rightsquigarrow v$ and $v \rightsquigarrow u$

1	1	1	2	3



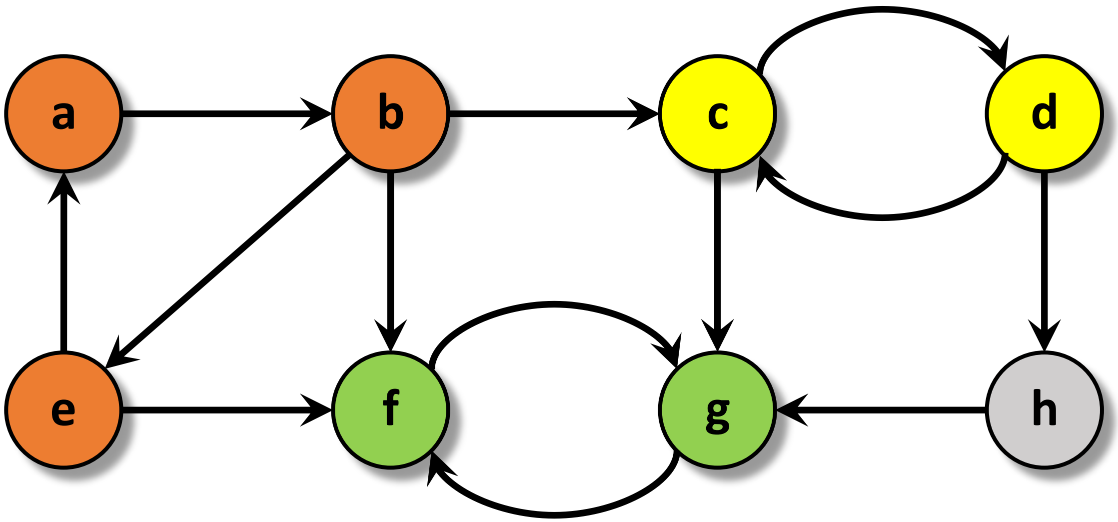
Ask the Audience

- Find all the strongly connected components (SCCs) of this directed graph



Ask the Audience

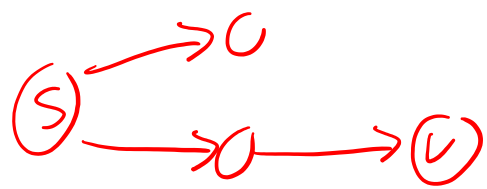
- Find all the strongly connected components (SCCs) of this directed graph



Strongly Connected Components

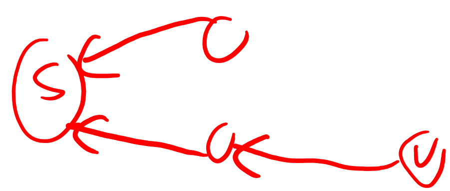
- **Observation:** $\text{SCC}(s)$ is all nodes $v \in V$ such that v is reachable from s and vice versa
  - Can find all nodes reachable from s using DFS
  - How do we find all nodes that *can reach* s?
    - DFS(s) in reverse of the graph!

DFS in G^R :



all nodes that can reach s in G

G :



path from s to v

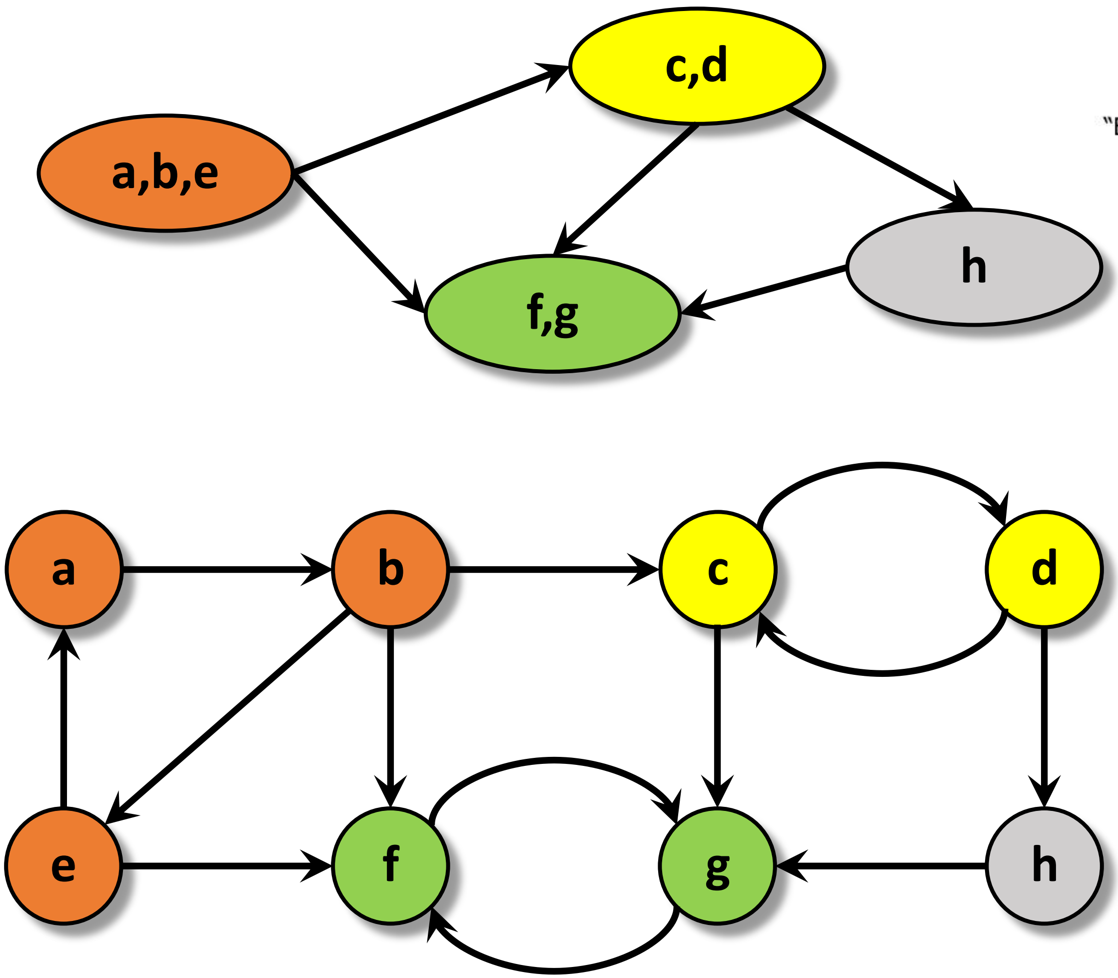
↓

path from v to s

SCCs by DFS: Take I

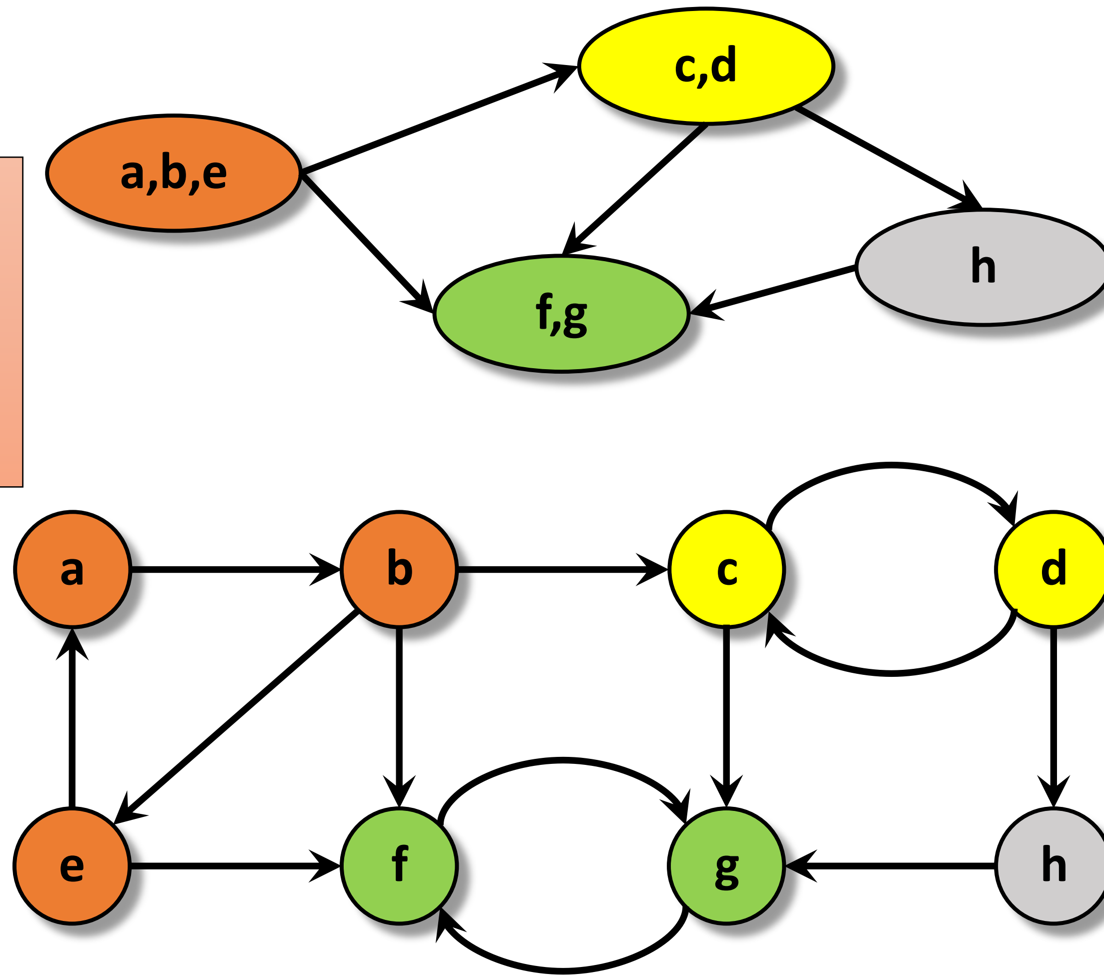
```
SCC-Slow():
  G^R = G with all edges "reversed"

  // Initialize an array and counter
  comp[1:n] = ⊥, c = 1

  for (u = 1,…,n):
    // If u has not been explored
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(G,u)
      T = set of nodes found by DFS(G^R,u)
      // S ∩ T contains SCC(u)
      label S ∩ T with c
      c = c + 1

  return comp
```

DFS: SCCs Form a DAG!



Clever use of DFS for SCC

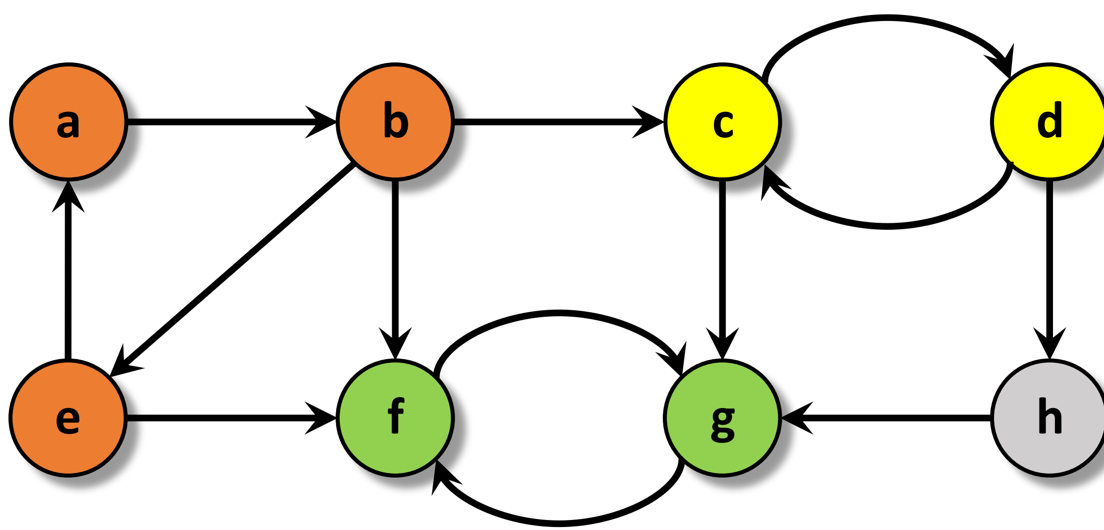
Observation: DFS from any node in a sink component finds that component



SCC Algorithm Template

- Repeat until all nodes marked:
  - Find a node in a sink component of G
  - Run DFS(u) to find SCC of u
  - Mark the nodes in SCC of u so not visited again
- How to find a node in a sink component?

Vertex	a	b	c	d	e	f	g	h
Finish f[]	16	15	12	11	14	8	9	10

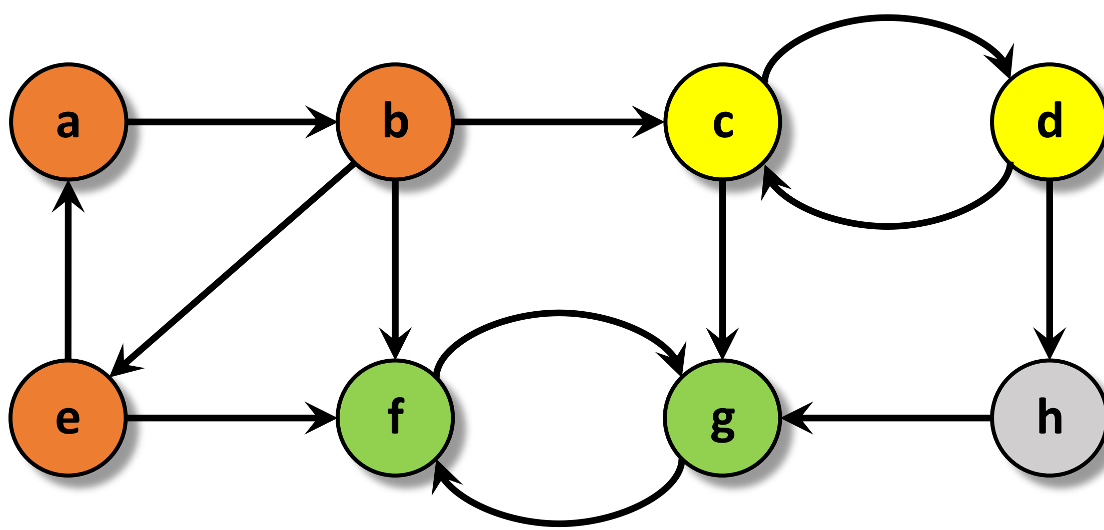


SCC Algorithm Template

- Repeat until all nodes marked:
  - Find a node in a sink component of G
  - Run DFS(u) to find SCC of u
  - Mark the nodes in SCC of u so not visited again
- How to find a node in a sink component?

Vertex	a	b	c	d	e	f	g	h
Finish f[]	16	15	12	11	14	8	9	10

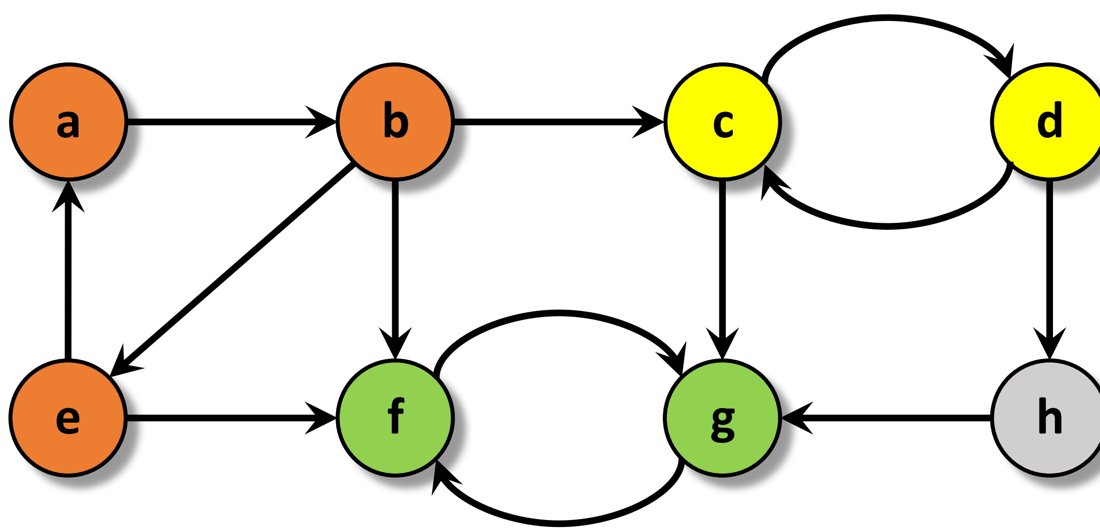
Fact: Node with largest finish time is in a *source* component



SCC Algorithm Template

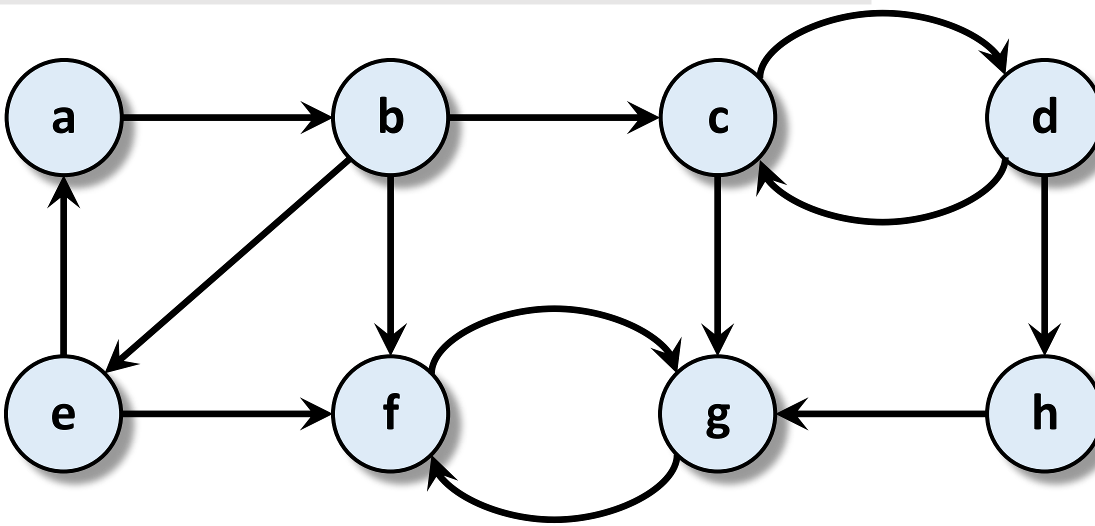
- Repeat until all nodes marked:
  - Find a node in a sink component of G
  - Run DFS(u) to find SCC of u
  - Mark the nodes in SCC of u so not visited again
- How to find a node in a sink component?
  - Node with largest finish time in reverse of G!

Fact: Node with largest finish time is in a *source* component



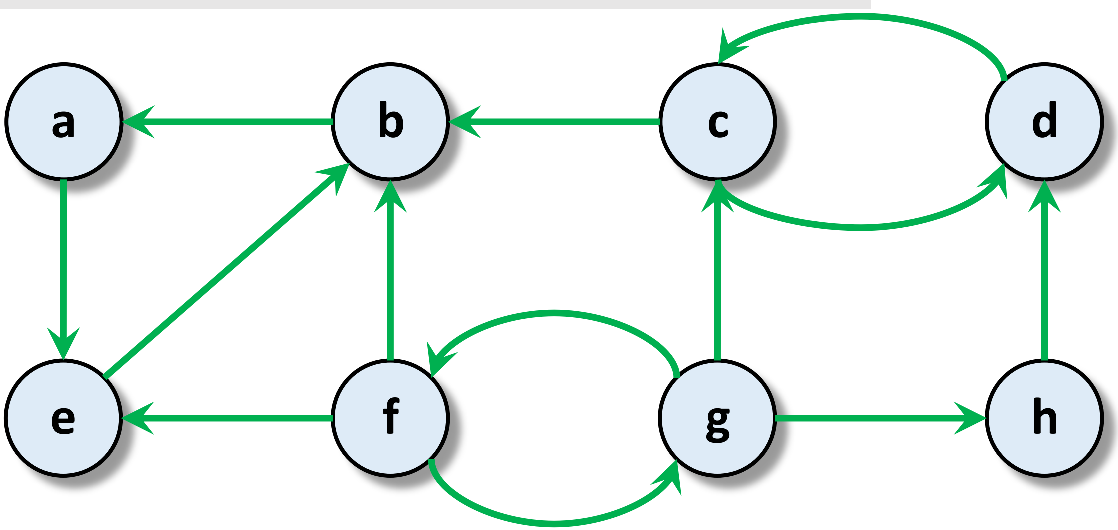
Linear-time algorithm for SCC

```
SCC(G):
  G^R = G with all edges "reversed"
  DFS of G^R to compute finish times f^R
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of f^R)
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



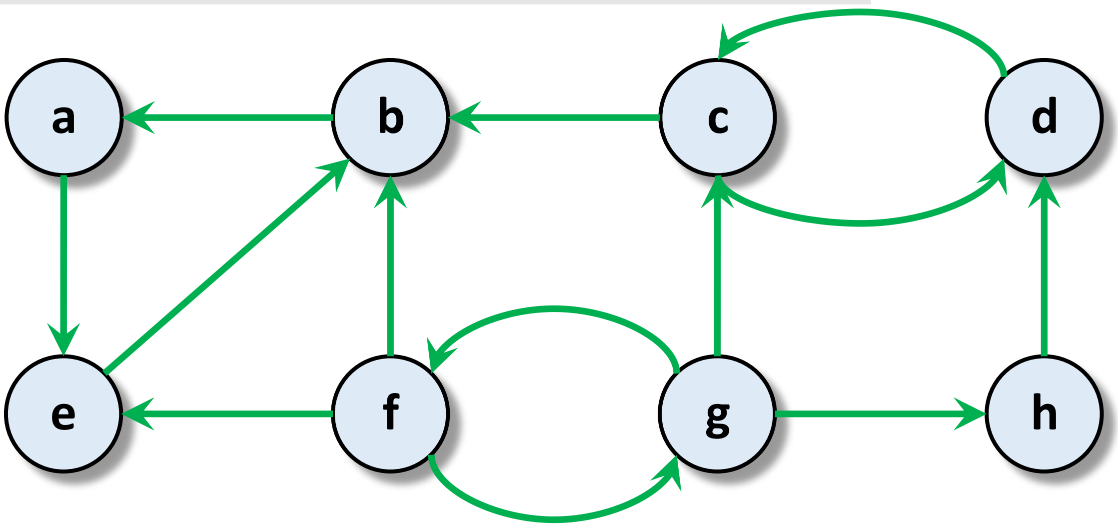
SCC Algorithm in Linear Time

```
SCC(G):
  Gᴿ = G with all edges "reversed"
  DFS of Gᴿ to compute finish times fᴿ
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of fᴿ)
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



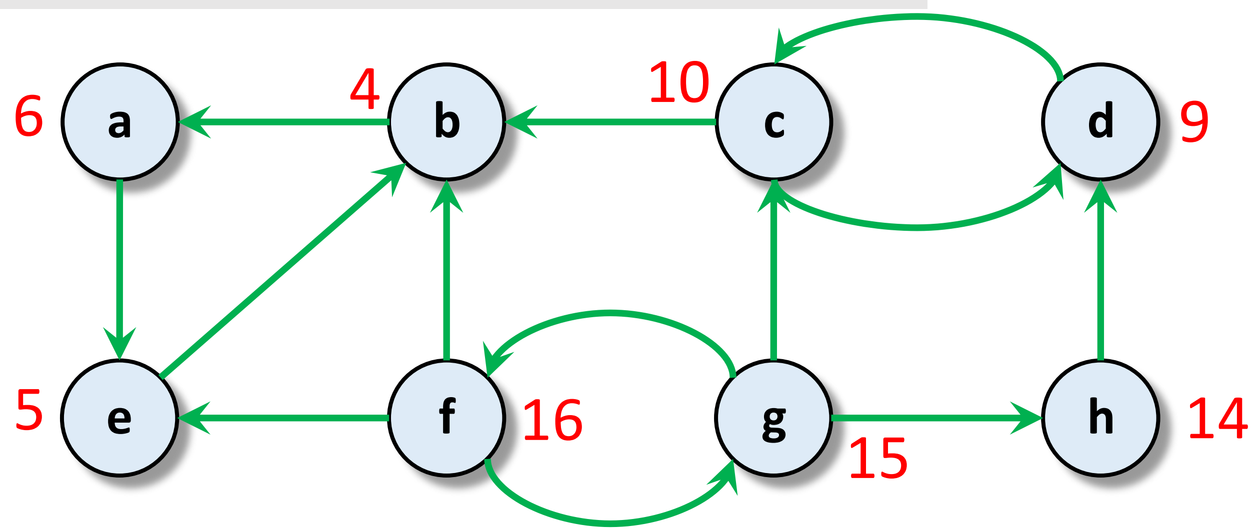
SCC Algorithm in Linear Time

```
SCC(G):
  Gᴿ = G with all edges "reversed"
  DFS of Gᴿ to compute finish times fᴿ
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of fᴿ)
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



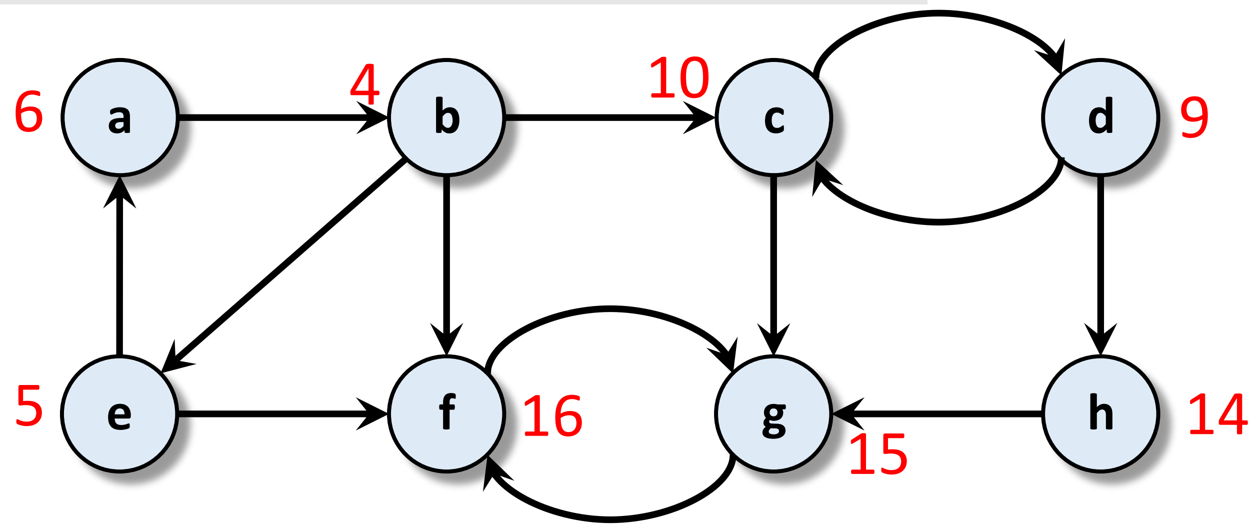
SCC Algorithm in Linear Time

```
SCC(G):
  Gᴿ = G with all edges "reversed"
  DFS of Gᴿ to compute finish times fᴿ
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of fᴿ)
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



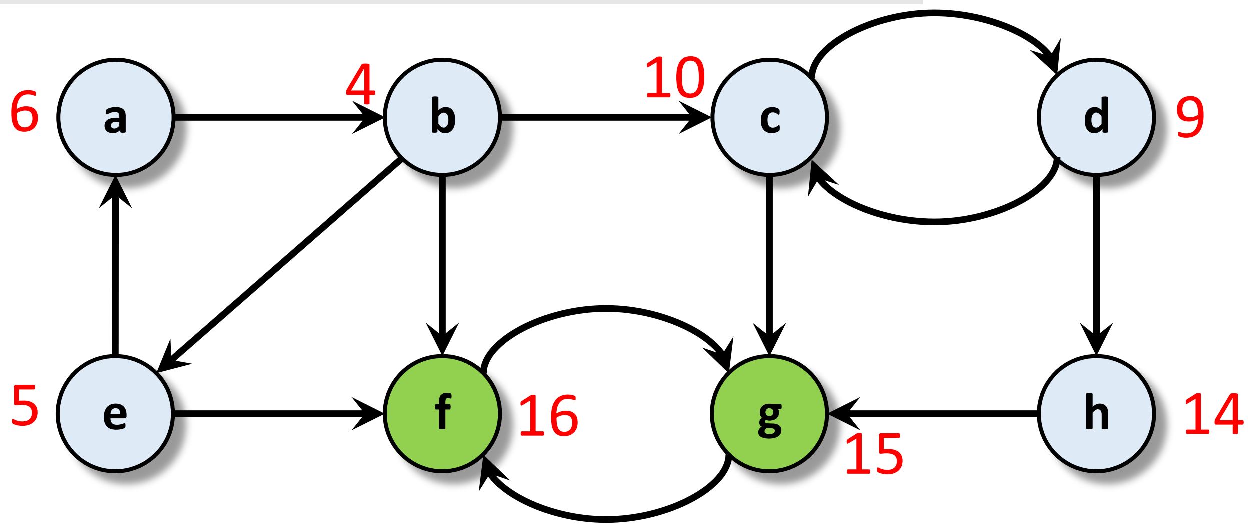
SCC Algorithm in Linear Time

```
SCC(G):
  G^R = G with all edges "reversed"
  DFS of G^R to compute finish times f^R
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of f^R)
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



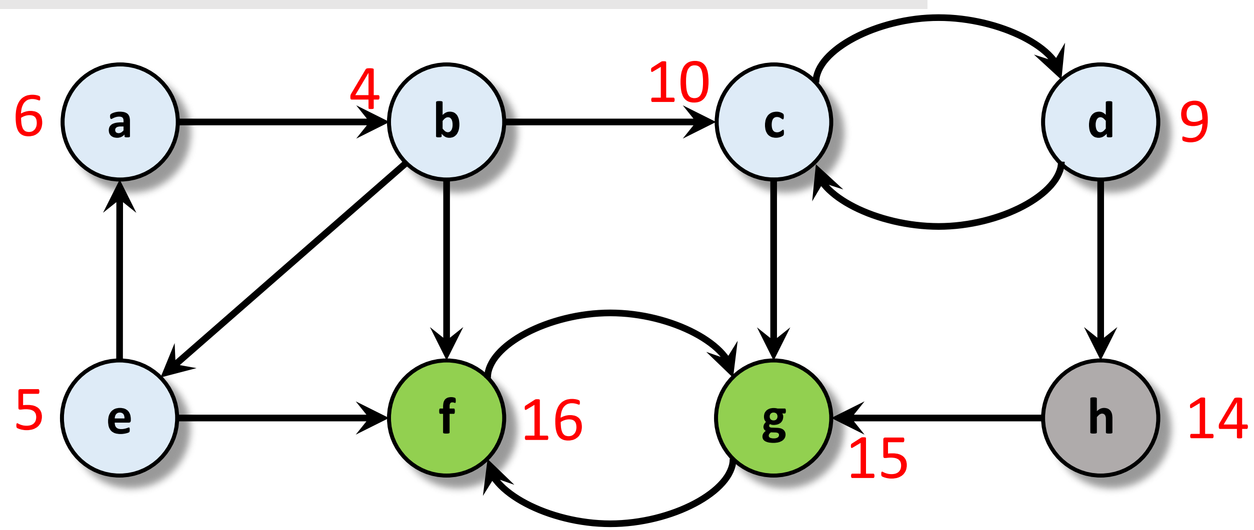
SCC Algorithm in Linear Time

```
SCC(G):
  G^R = G with all edges "reversed"
  DFS of G^R to compute finish times f^R
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of f^R)
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



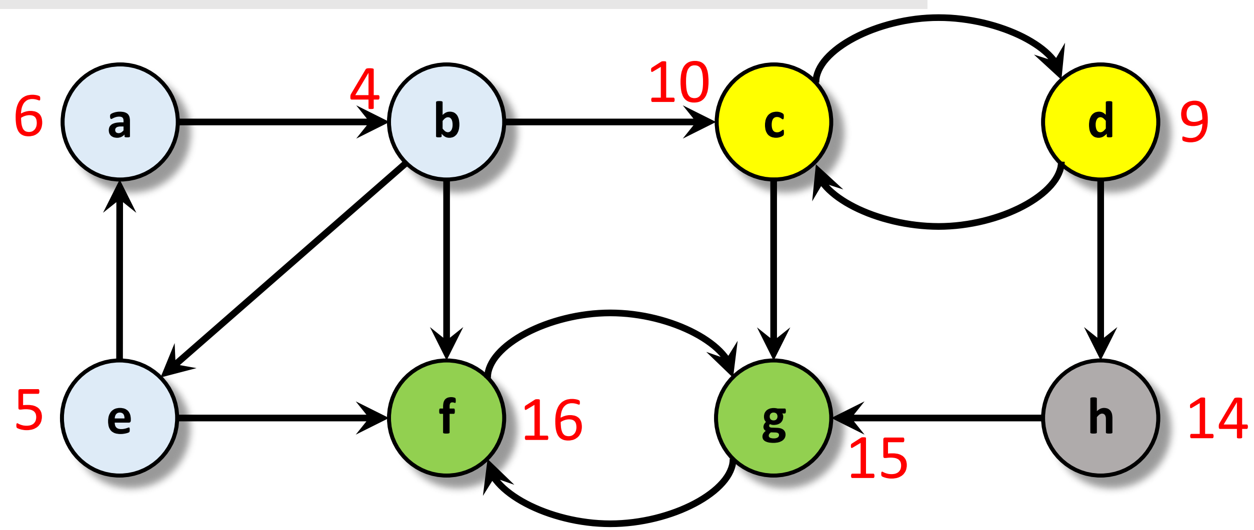
SCC Algorithm in Linear Time

```
SCC(G):
  G^R = G with all edges "reversed"
  DFS of G^R to compute finish times f^R
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of f^R)
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



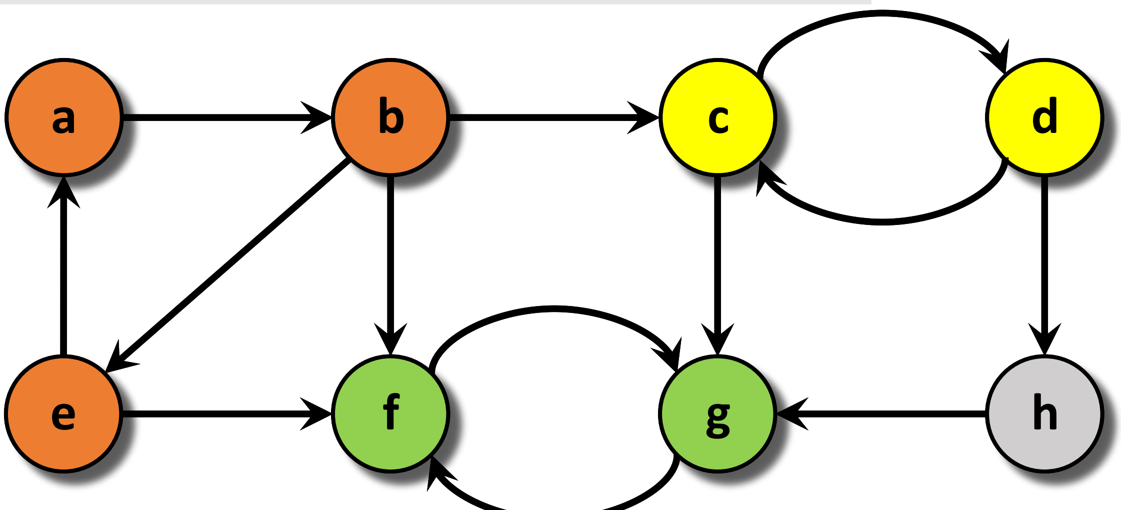
SCC Algorithm in Linear Time

```
SCC(G):
  Gᴿ = G with all edges "reversed"
  DFS of Gᴿ to compute finish times fᴿ
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of fᴿ)
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



SCC Algorithm in Linear Time

```
SCC(G):
  G^R = G with all edges "reversed"
  DFS of G^R to compute finish times f^R
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of f^R)
    if (comp[u] != ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



$\Theta(n + m)$ time

Strongly Connected Components Recap

- **Problem:** Given a directed graph G, split it into strongly connected components
- **Input:** Directed graph $G = (V, E)$
- **Output:** A labeling of the vertices by their strongly connected component
- **Punchline: $\boldsymbol{O(n+m)}$ time** algorithm for SCCs
  - Clever use of DFS on G and reverse of G
  - Can also compute the meta-graph DAG of SCCs
- Can be directly invoked in other algorithms