Exhaustive Algorithms on Graphs: Topological Sorting with DFS

Lecture 04.02

By Marina Barsky

Recap: Depth-First Search (Recursive)

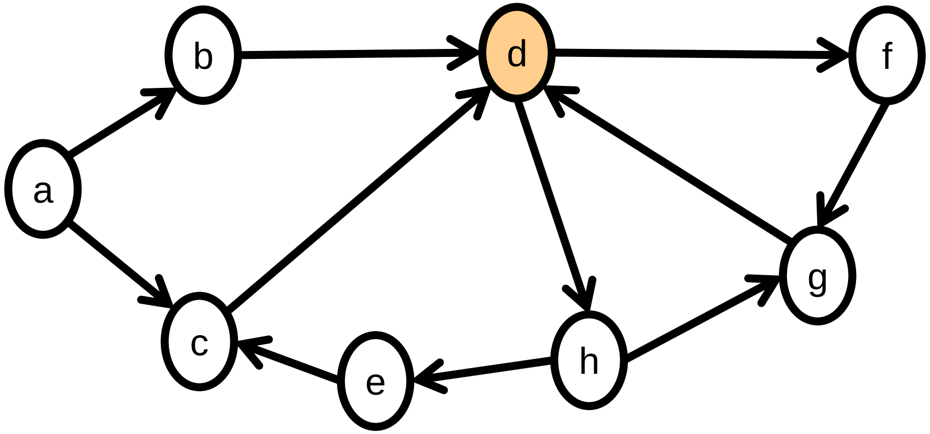
Recursive implementation implicitly replaces the `todo` stack with the call stack.

```
Algorithm DFS(G, current)

    current.state:= "discovered"
    for each u in neighbors(current)
        if u.state = "undiscovered" then
            DFS(G, u)
    current.state:="processed"

for each u in vertices of G
    u.state:= "undiscovered"
DFS(G, start)  // start is a vertex in G
```

This is an exhaustive algorithm, because it visits every node and every edge of graph G

It runs in time **O(n + m)** if implemented using adjacency list

DFS in Directed Graphs



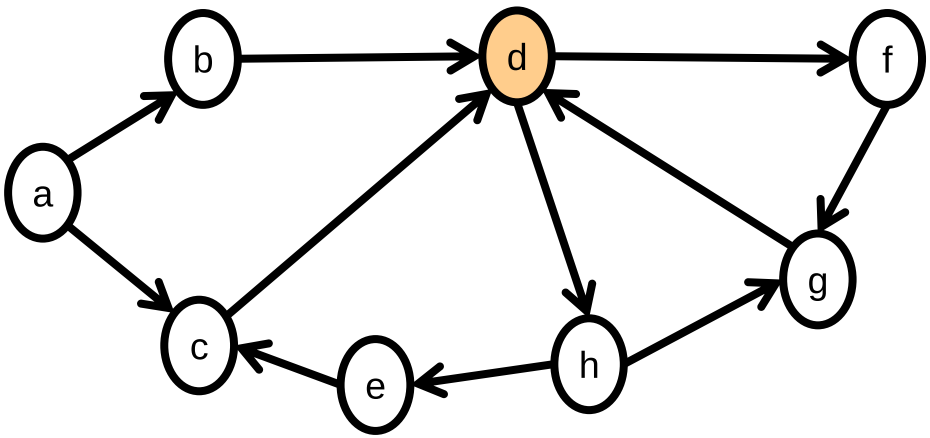
The algorithm for Directed Graphs is exactly the same

By the end we discover all the nodes in digraph G that are **reachable** from the source node *start*

```
Algorithm DFS(digraph G, current)

    current.state:= "discovered"
    for each u in out_arcs(current)
        if u.state = "undiscovered" then
            DFS(G, u)
    current.state:="processed"

for each u in vertices of G
    u.state:= "undiscovered"
DFS(digraph G, start)  // start is a vertex in G
```

The time of discovery and finishing time

- Unlike in BFS (with its removal from the front of a queue) the order in which we discover a new unprocessed vertex differs from the order in which we mark vertices as processed



- Imagine that we have a global clock, and before we begin: clock = 1

- The moment that we mark some node as processed, we also mark it with the current value of the clock, and we increment the clock value by 1

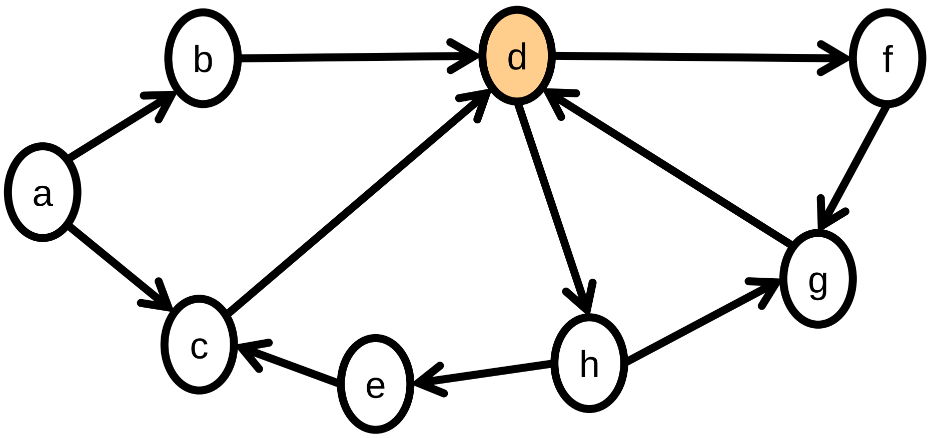
Definition

Let ***finishing time f(v)*** of node v be the value of *clock* variable at the moment that v was marked as **processed** by the DFS algorithm

In essence f(v) is the count of all the vertices processed before v

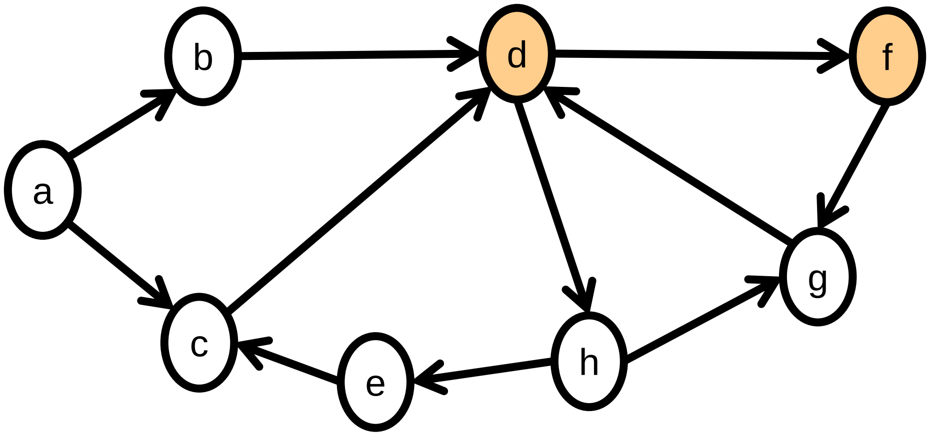
Example of computing finishing time

- Let's start DFS from an arbitrary vertex, say, vertex d
- We traverse the graph and recursively call DFS on all nodes reachable from d
- The node is marked as processed when there are no more undiscovered nodes that can be reached from it



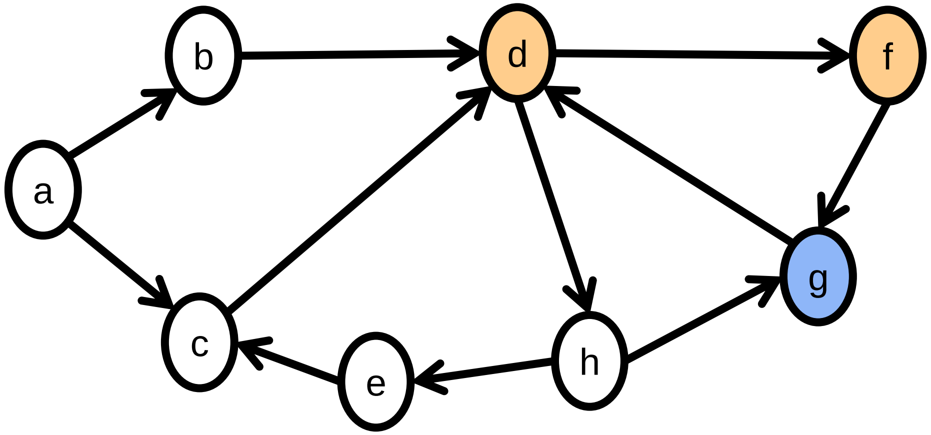
clock = 1

Example of computing finishing time



clock = 1

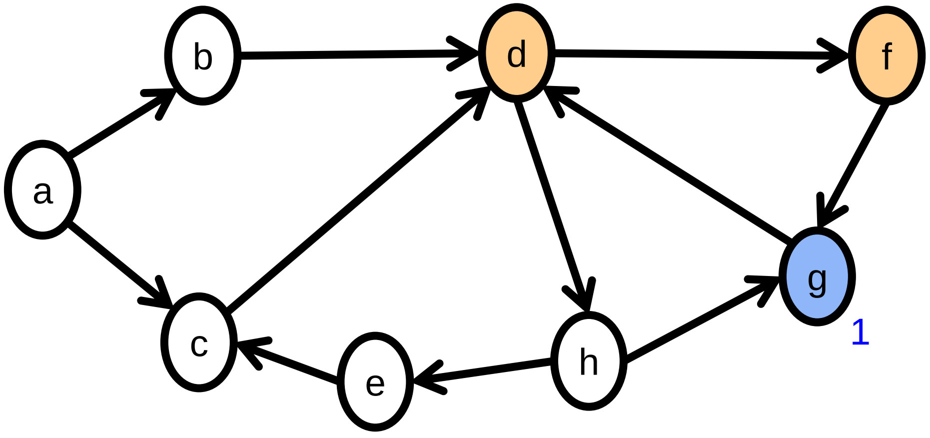
Example of computing finishing time



clock = 1

There is nowhere to go from g:
Node g is processed

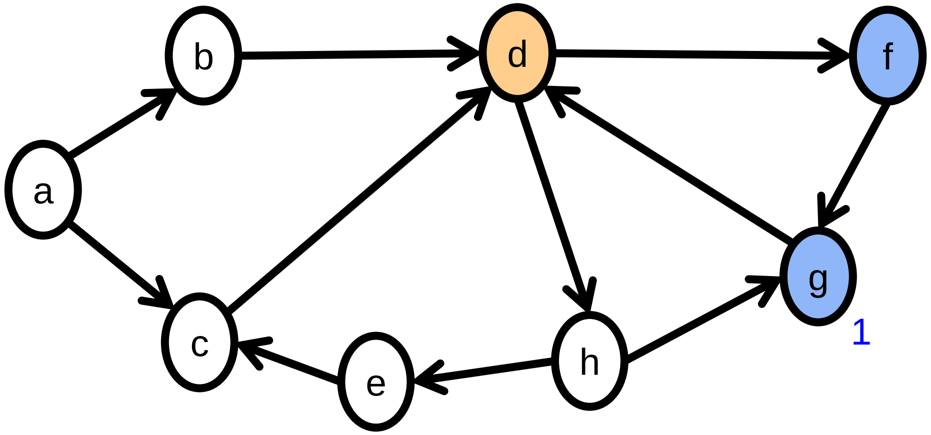
Example of computing finishing time



clock = 2

There is nowhere to go from g:
Node g is processed
Its finishing time is 1 (first to finish)

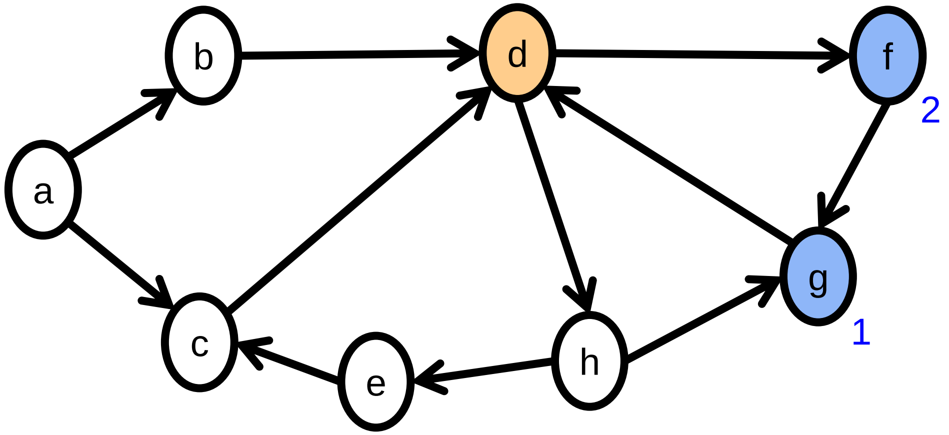
Example of computing finishing time



clock = 2

We return from the call stack and the next node marked as processed is node f

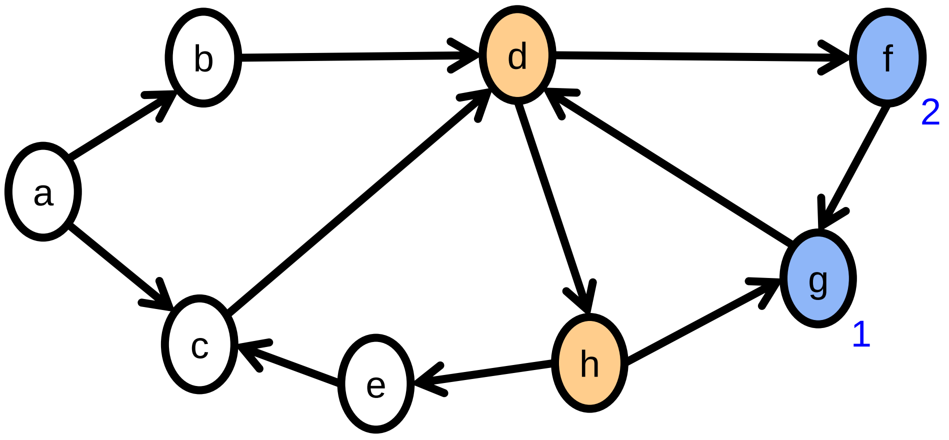
Example of computing finishing time



clock = 3

We return from the call stack and the next node marked as processed is node f
Its finishing time is 2

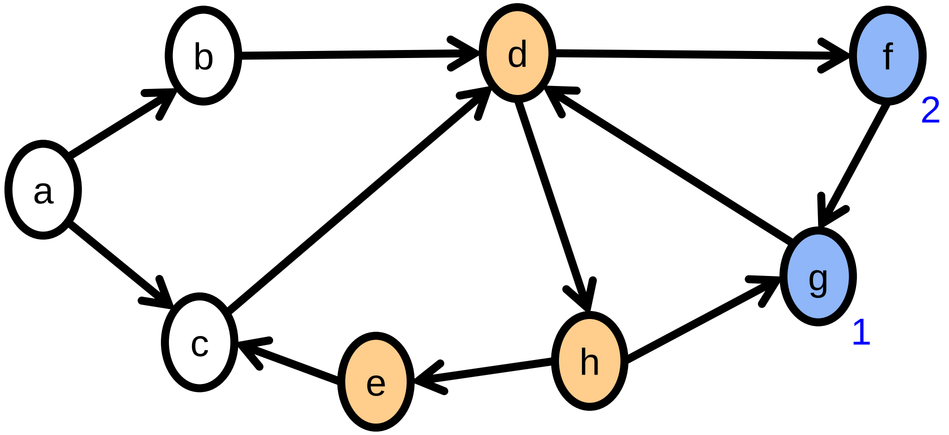
Example of computing finishing time



clock = 3

Node d is not done yet:
We move to its next neighbor h

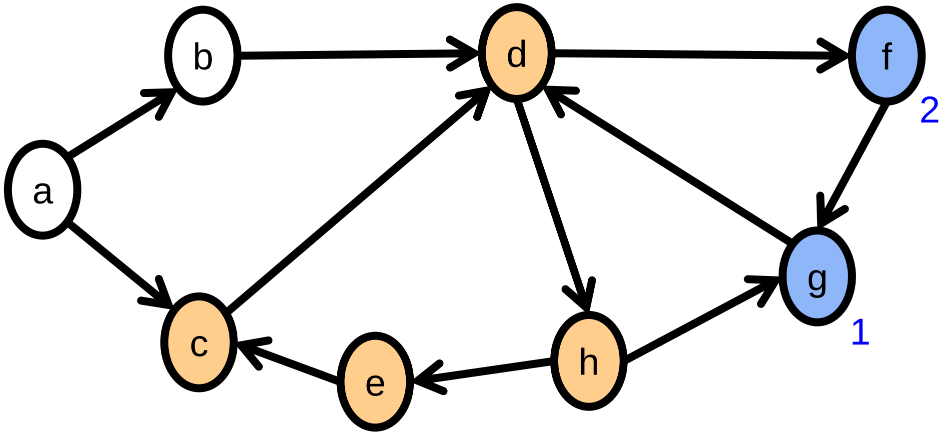
Example of computing finishing time



clock = 3

Node d is not done yet:
We move to its next neighbor h, and then to e

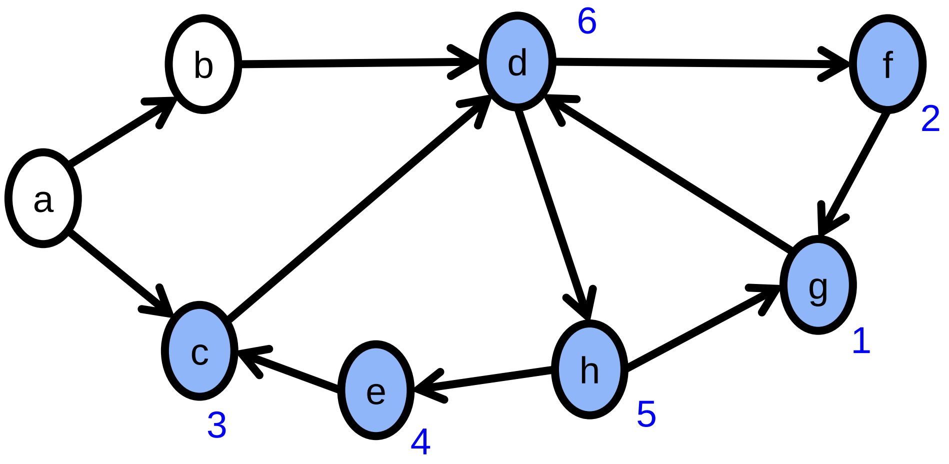
Example of computing finishing time



clock = 3

Node d is not done yet:
We move to its next neighbor h, and then to e, and then to c

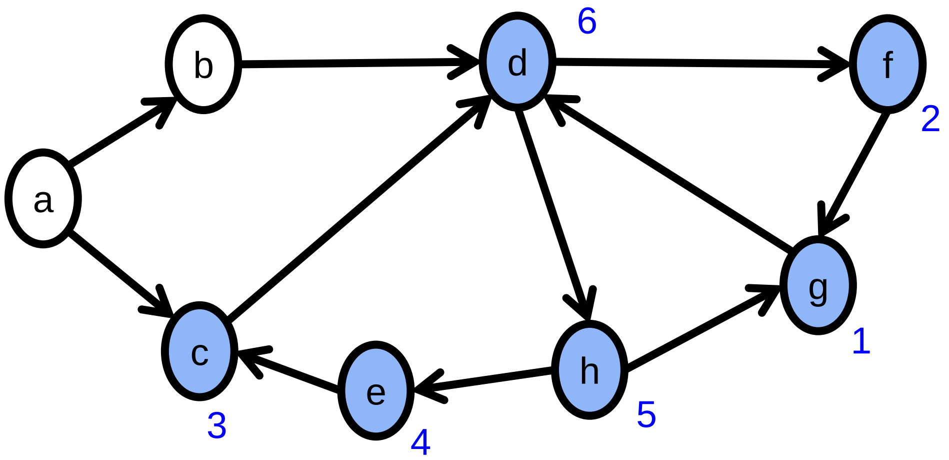
Example of computing finishing time



clock = 7

We mark every node with its finishing time

Example of computing finishing time

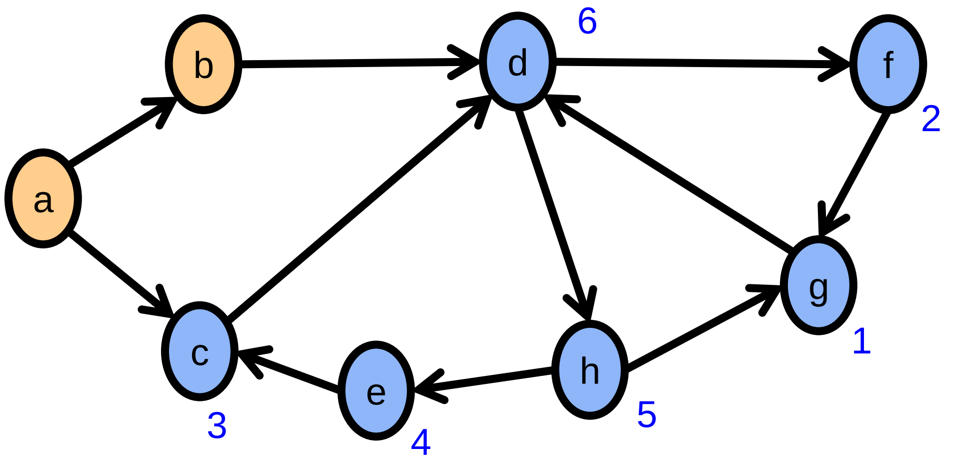


clock = 7

All nodes reachable from d have been processed

We can continue from any remaining unprocessed vertex, say, a

Example of computing finishing time

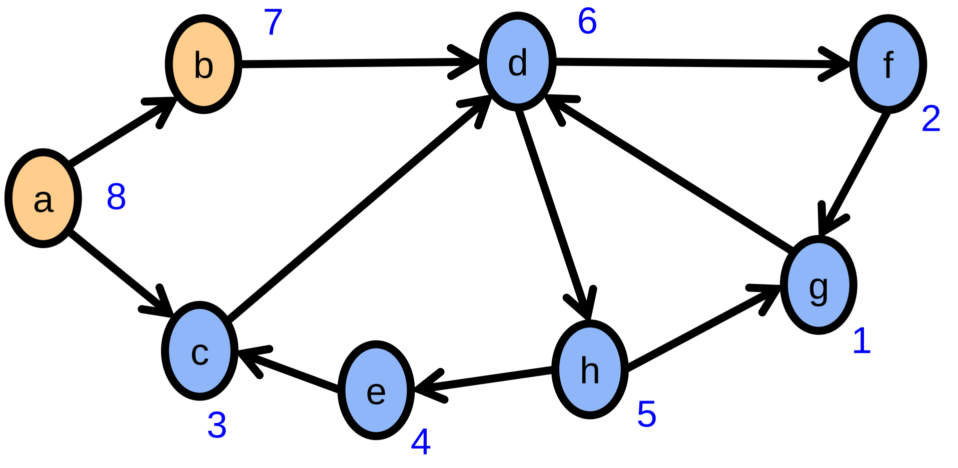


clock = 7

All nodes reachable from d have been processed

We can continue from any remaining unprocessed vertex, say, a

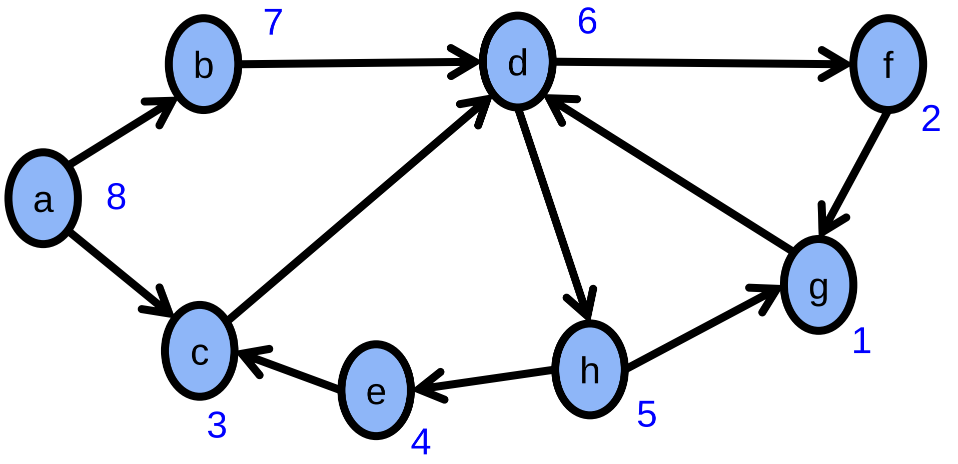
Example of computing finishing time



All nodes reachable from a are now processed

Mark remaining finishing time.

Finishing time for all vertices



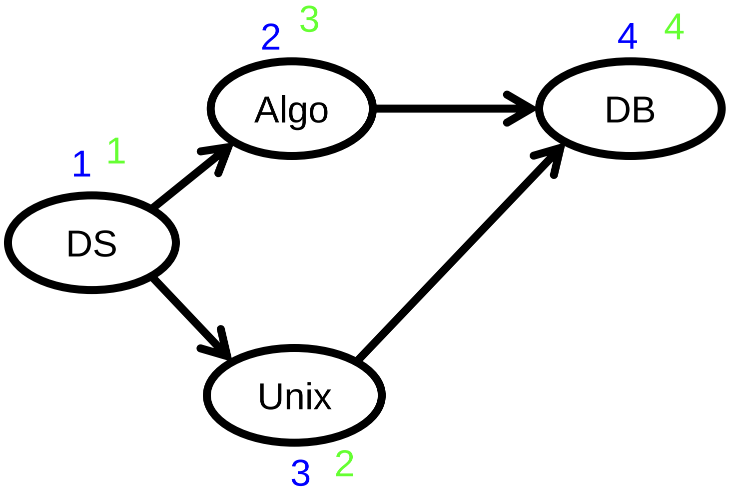
a	b	c	d	e	f	g	h	e
8	7	3	6	4	2	1	5	4

Note that this order of processing is not unique, because we selected the next starting vertex arbitrarily (try to start from vertex h)

Modeling order constraints with DAG

Directed graphs can model ordering constrains:

- Clothes: we cannot wear boots before socks, and a coat before dress
- Course prerequisite structure at universities: some courses must be taken before others



A directed edge v → w indicates that course *v* must be completed before course *w*

Such ordering of vertices can only be modeled with a Directed **Acyclic** Graph [**DAG**]

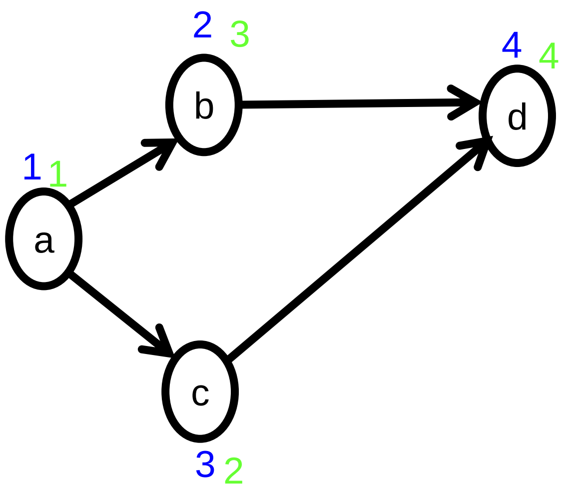
Topological Order

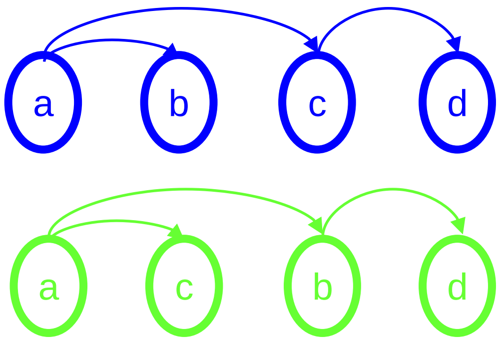
- *Topological sorting* is an ordering of vertices in a Directed Acyclic Graph [DAG] in which *each node comes before all nodes to which it has outgoing edges.*

- Each node is assigned a label $t(v)$:
  - $t(v)$ is a unique order of node v from 1 to n
  - If there is a directed edge $u \rightarrow v$, then $t(u)<t(v)$

  For example, topological ordering for courses is the sequence which does not violate the prerequisite requirement

- **Topological sorting is not possible if the graph has a cycle**, since for two vertices u and v on the cycle, it is not possible to create a sequence where $t(u)<t(v)$ and at the same time $t(v)<t(u)$





Topological Order is not unique

Computing Topological Order with DFS

The topological order is exactly opposite to the finishing time:

- The finishing time of the vertex indicates that all nodes reachable from it have been processed, that means it is not a prerequisite for any one of them
- Thus the node without prerequisites (with the smallest t(v)) finishes last (has the largest f(v))

Algorithms:

- We can compute finishing time (as before) and sort vertices in descending order of finishing time
- We also can generate topological ordering during the DFS directly, by adding a processed node in front of a Linked List (see next slide)
- There is an alternative algorithm which uses in-degree of vertices (read the textbook Chapter 13.4)

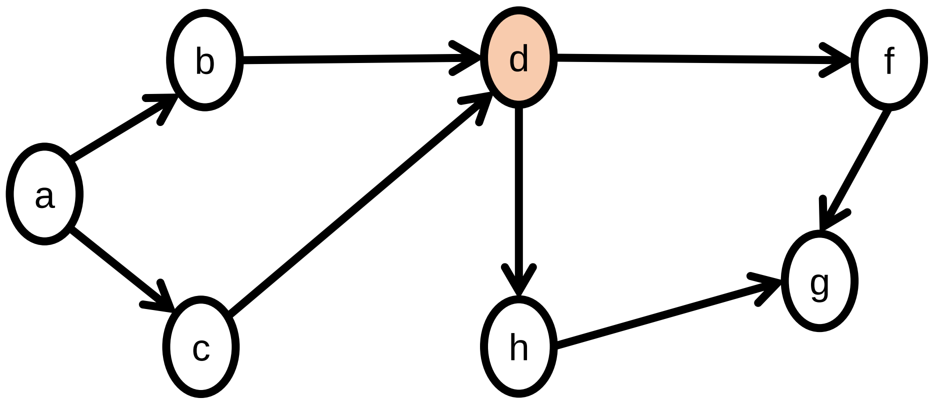
Topological Sort with DFS

```
global sorted_nodes:= empty linked list
global clock: = 1
```

```
Algorithm DFS(DAG G, current)

    current.state:= "discovered"
    for each u in out_arcs(current)
        if u.state = "undiscovered" then
            DFS(G, u)
    current.state:="processed"
    current.finishing_time: = clock
    clock: = clock + 1
    sorted_nodes.add_in_front(current)
```

```
Algorithm DFS_loop(DAG G)

    mark all nodes of G as "undiscovered"
    for each u in vertices of G
        if u.state = "undiscovered"
            DFS(DAG G, u)
```

Example



clock = 1

current_vertex

Recursion stack:

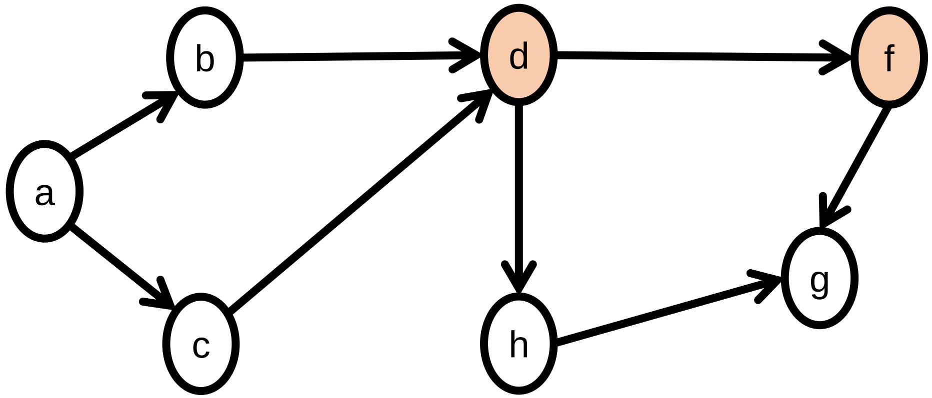
d					

Finishing time

Sorted list

Example

clock = 1



current_vertex

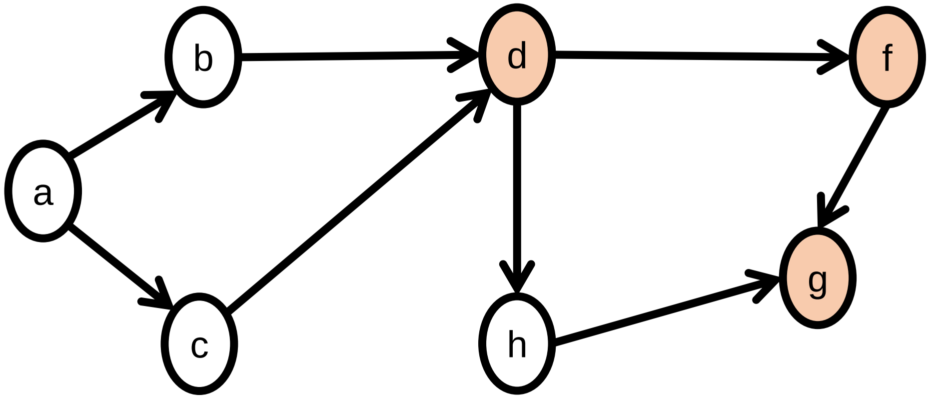
Recursion stack:

d	f				

Finishing time

Sorted list

Example



clock = 1

current_vertex

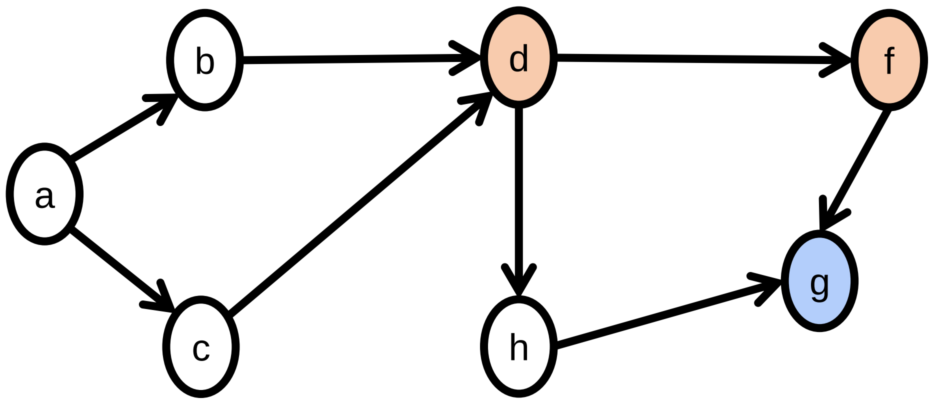
Recursion stack:

d	f	g			

Finishing time							
Sorted list							

Example

clock = 1



current_vertex	g

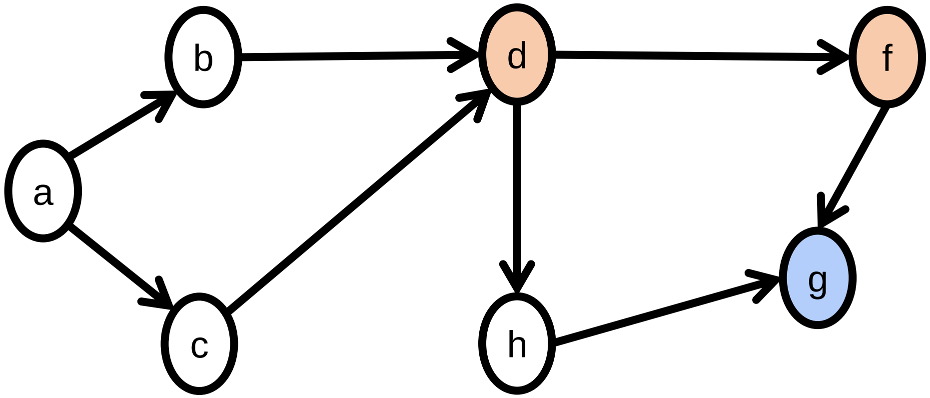
Recursion stack:

d	f				

Finishing time							
Sorted list							

Example

clock = 2



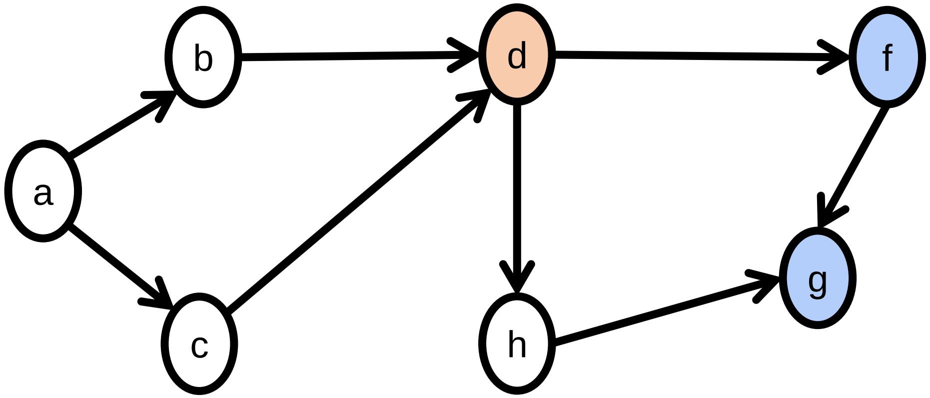
current_vertex	g

Recursion stack:	d	f				

Finishing time	1						
Sorted list	g						

Example

clock = 2



current_vertex	f

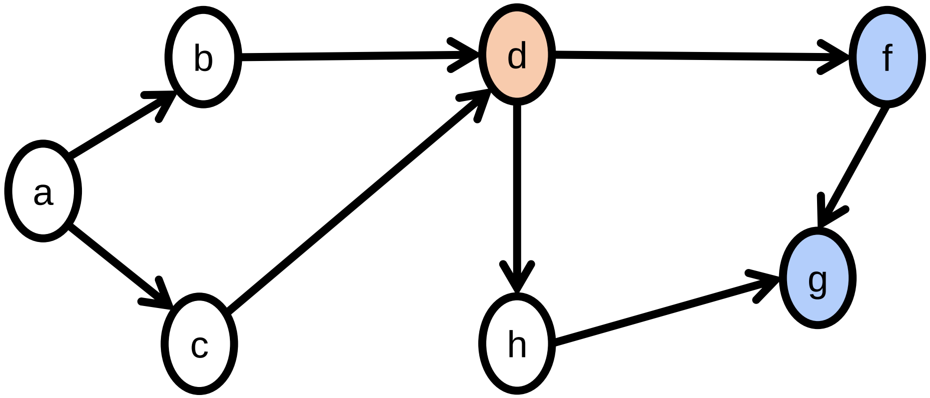
Recursion stack:

d					

Finishing time	1						
Sorted list	g						

Example

clock = 3



current_vertex: f

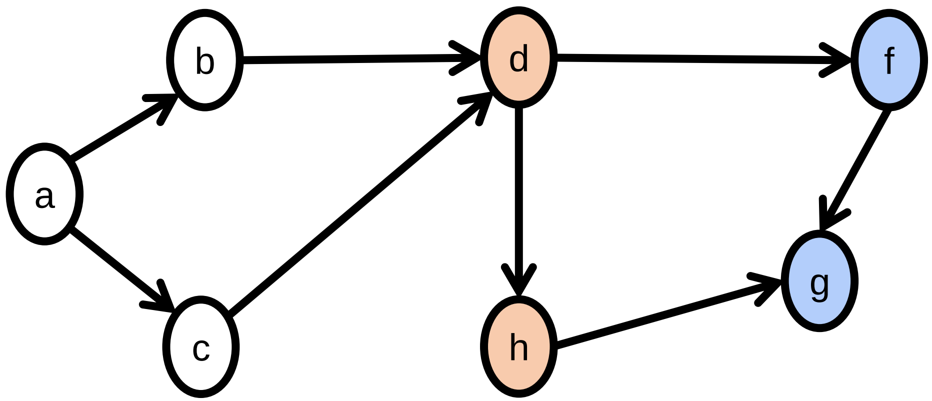
Recursion stack:

d					

Finishing time	2	1					
Sorted list	f	g					

Example

clock = 3



current_vertex

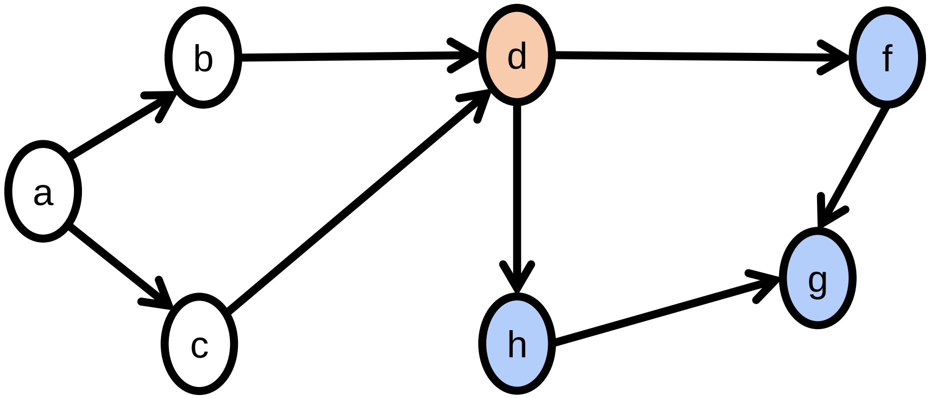
Recursion stack:

d	h				

Finishing time	2	1					
Sorted list	f	g					

Example

clock = 3



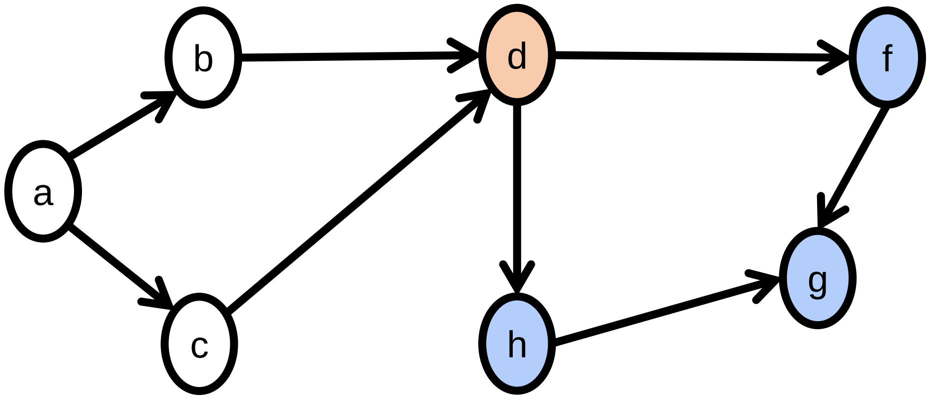
current_vertex: h

Recursion stack: d

Finishing time	2	1					
Sorted list	f	g					

Example

clock = 4



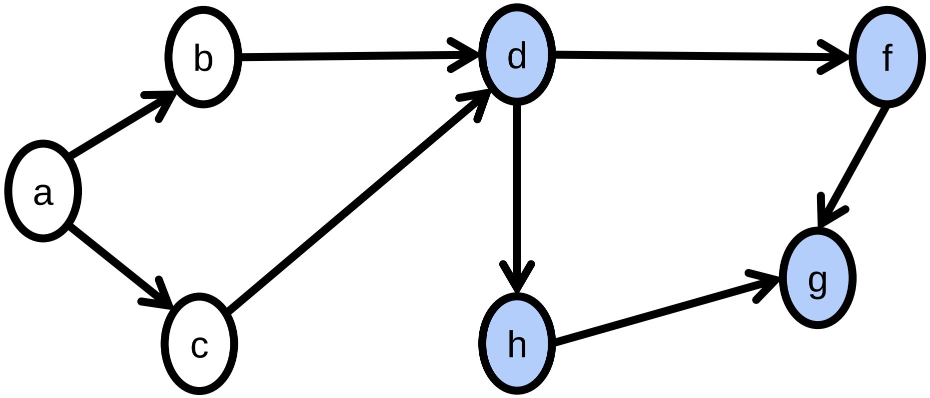
current_vertex: h

Recursion stack: d

Finishing time	3	2	1				
Sorted list	h	f	g				

Example

clock = 4

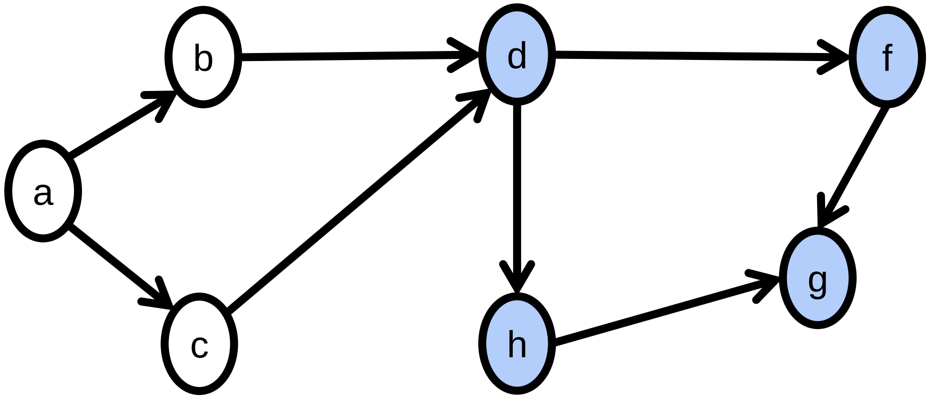


current_vertex: d

Recursion stack:

Finishing time	3	2	1				
Sorted list	h	f	g				

Example

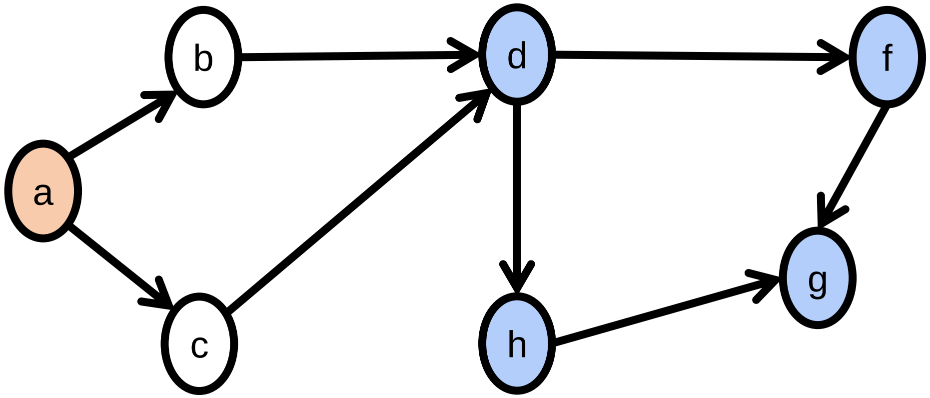


current_vertex	d

Recursion stack:

Finishing time	4	3	2	1			
Sorted list	d	h	f	g			

Example



clock = 5

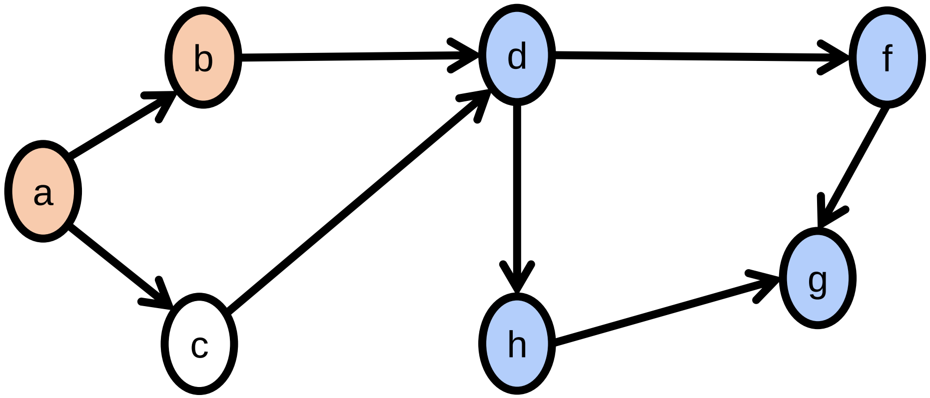
current_vertex

Recursion stack:

a					

Finishing time	4	3	2	1			
Sorted list	d	h	f	g			

Example



clock = 5

current_vertex

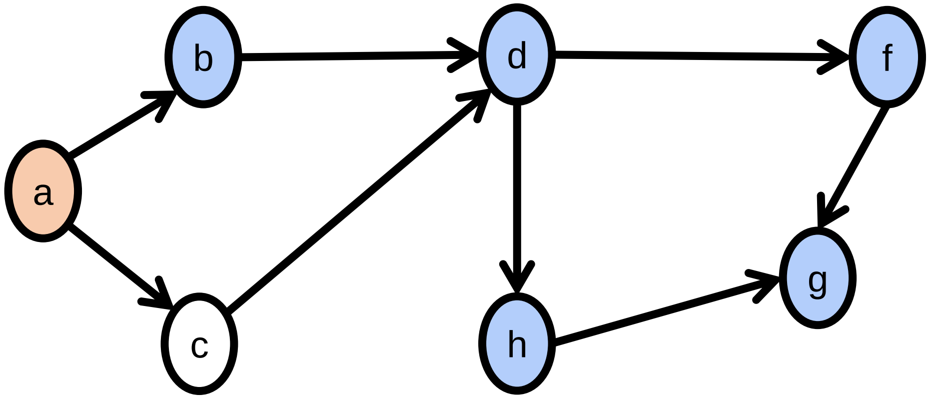
Recursion stack:

a	b				

Finishing time	4	3	2	1			
Sorted list	d	h	f	g			

Example

clock = 5



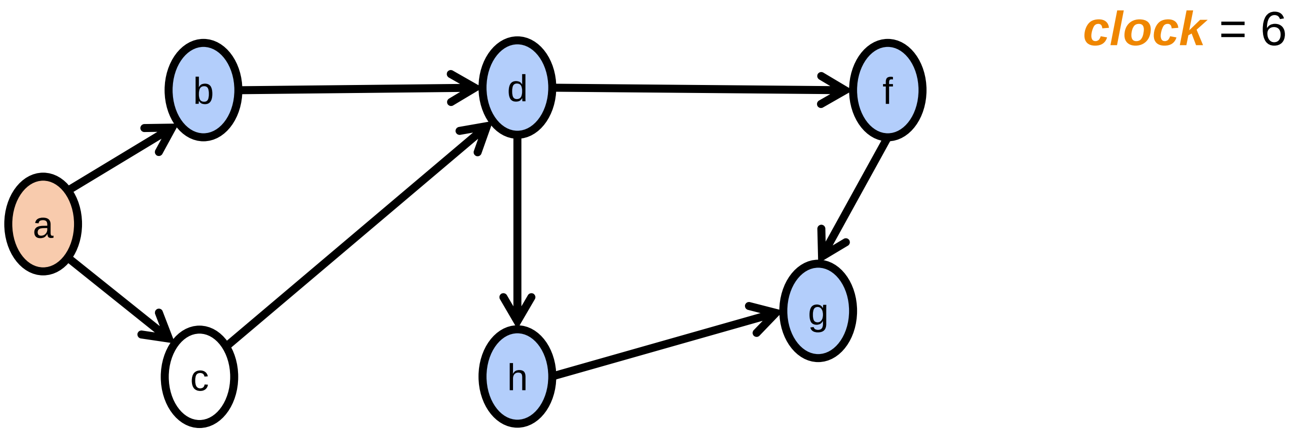
current_vertex: b

Recursion stack:

a					

Finishing time	4	3	2	1			
Sorted list	d	h	f	g			

Example



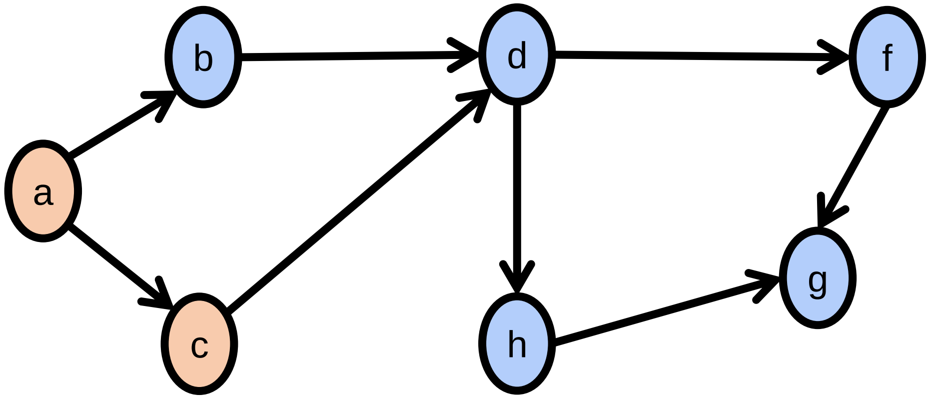
current_vertex

Recursion stack:

a					

Finishing time	5	4	3	2	1		
Sorted list	b	d	h	f	g		

Example



current_vertex

current_vertex

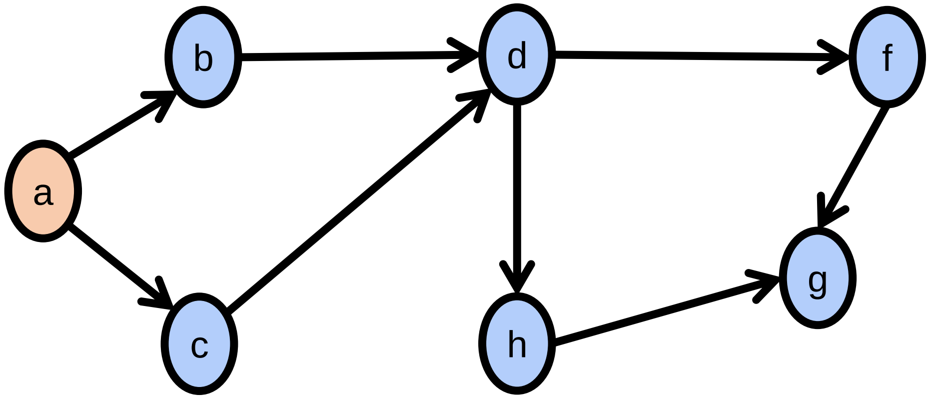
Recursion stack:

a	c				

Finishing time	5	4	3	2	1		
Sorted list	b	d	h	f	g		

Example

clock = 6



current_vertex: c

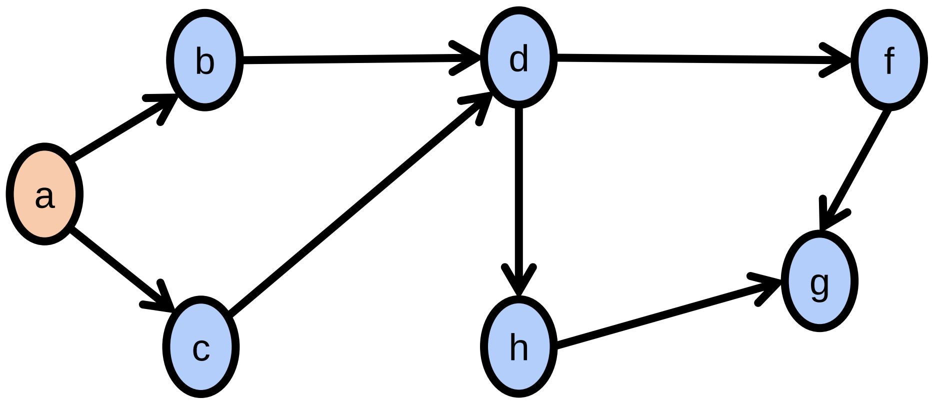
Recursion stack:

a					

Finishing time	5	4	3	2	1		
Sorted list	b	d	h	f	g		

Example

clock = 7



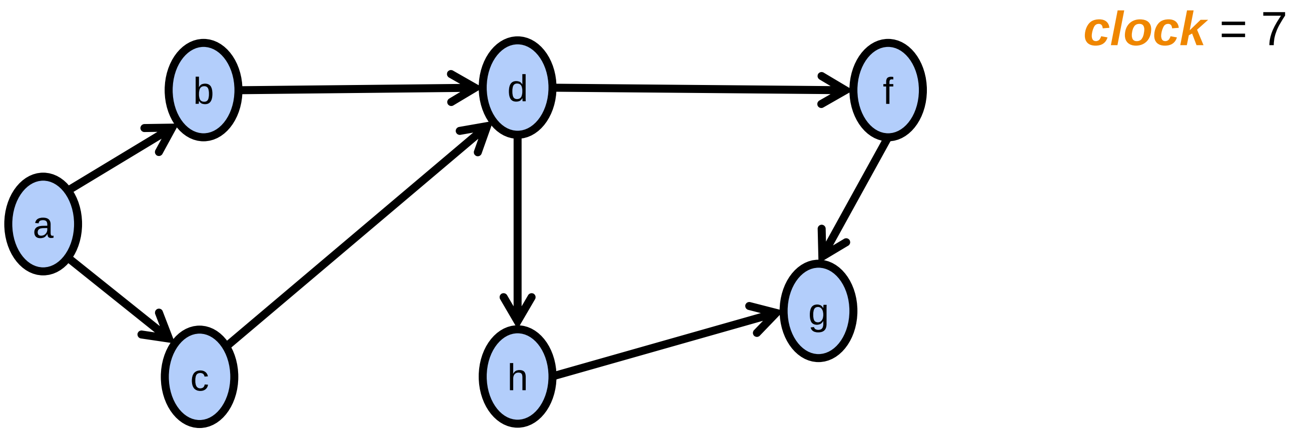
current_vertex

Recursion stack:

a					

Finishing time	6	5	4	3	2	1	
Sorted list	c	b	d	h	f	g	

Example

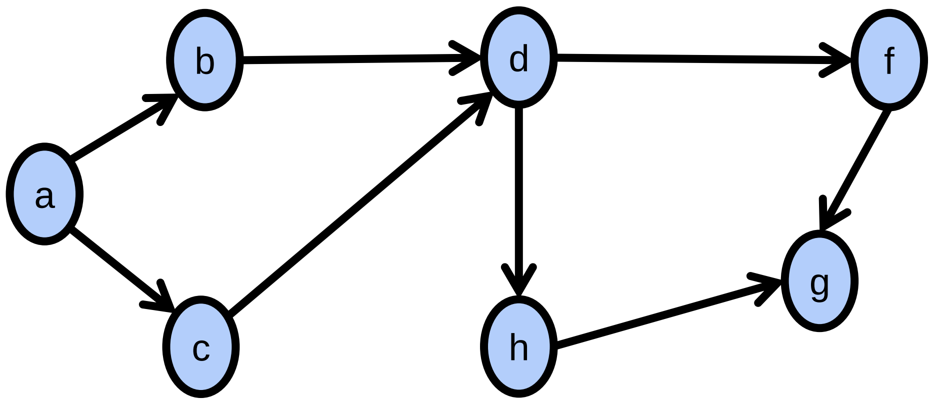


current_vertex: a

Recursion stack:

Finishing time	6	5	4	3	2	1	
Sorted list	c	b	d	h	f	g	

Example



current_vertex | a

Recursion stack:

Sorted list	a	c	b	d	h	f	g
Finishing time	7	6	5	4	3	2	1

Question

- How can we use the same DFS loop to determine if the graph is cycle-free?