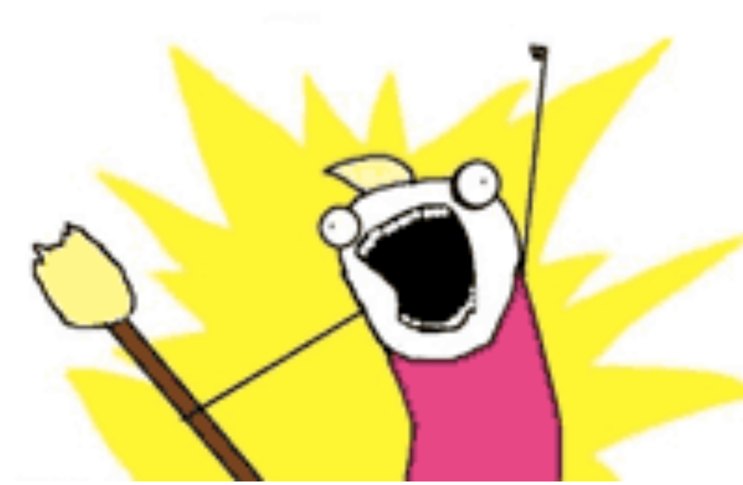
# CSC263 Week 9

# Announcements

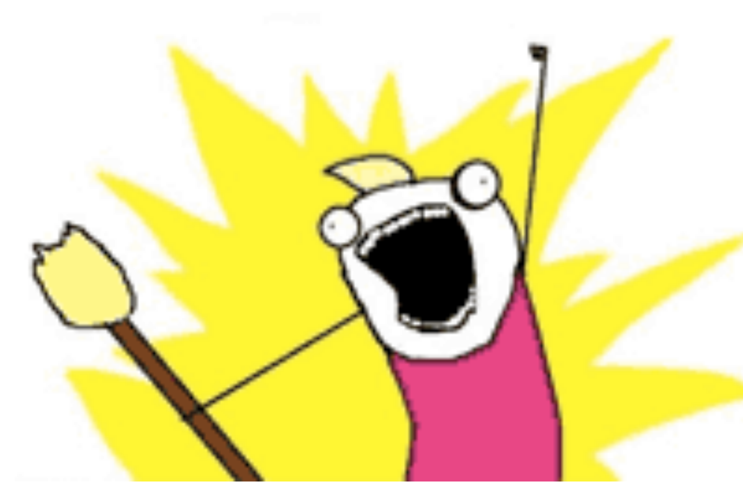
**HW3 is graded. Average is 81%**



# Announcements

**Problem Set 4 is due this Tuesday!**

Due Tuesday (Nov 17)



# Recap

- ➔The Graph ADT
  - ◆definition and data structures
- ➔BFS
  - ◆gives us single-source **shortest** path
  - ◆Let **δ(s, v)** denote the length of shortest path from **s** to **v**...
  - ◆then after performing a BFS starting from **s**, we have, for all vertices v

$$\mathbf{d[v] = \delta(s, v)}$$

**We can prove it.**

# Idea of the proof

There is no way $d[v] < \delta(s, v)$, according to Lemma 22.2

Use contradiction: suppose there exist v s.t. $d[v] > \delta(s, v)$, let v be the one with the **minimum** $\delta(s, v)$.

Then on a shortest path between s and v, pick vertex u which is immediately before v...
then we have $\mathbf{d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1}$

Must be equal because u is on the shortest path from s to v.

Must be equal because v is the minimum $\delta(s, v)$ that violates $d[v] > \delta(s, v)$, so u must not be violating.

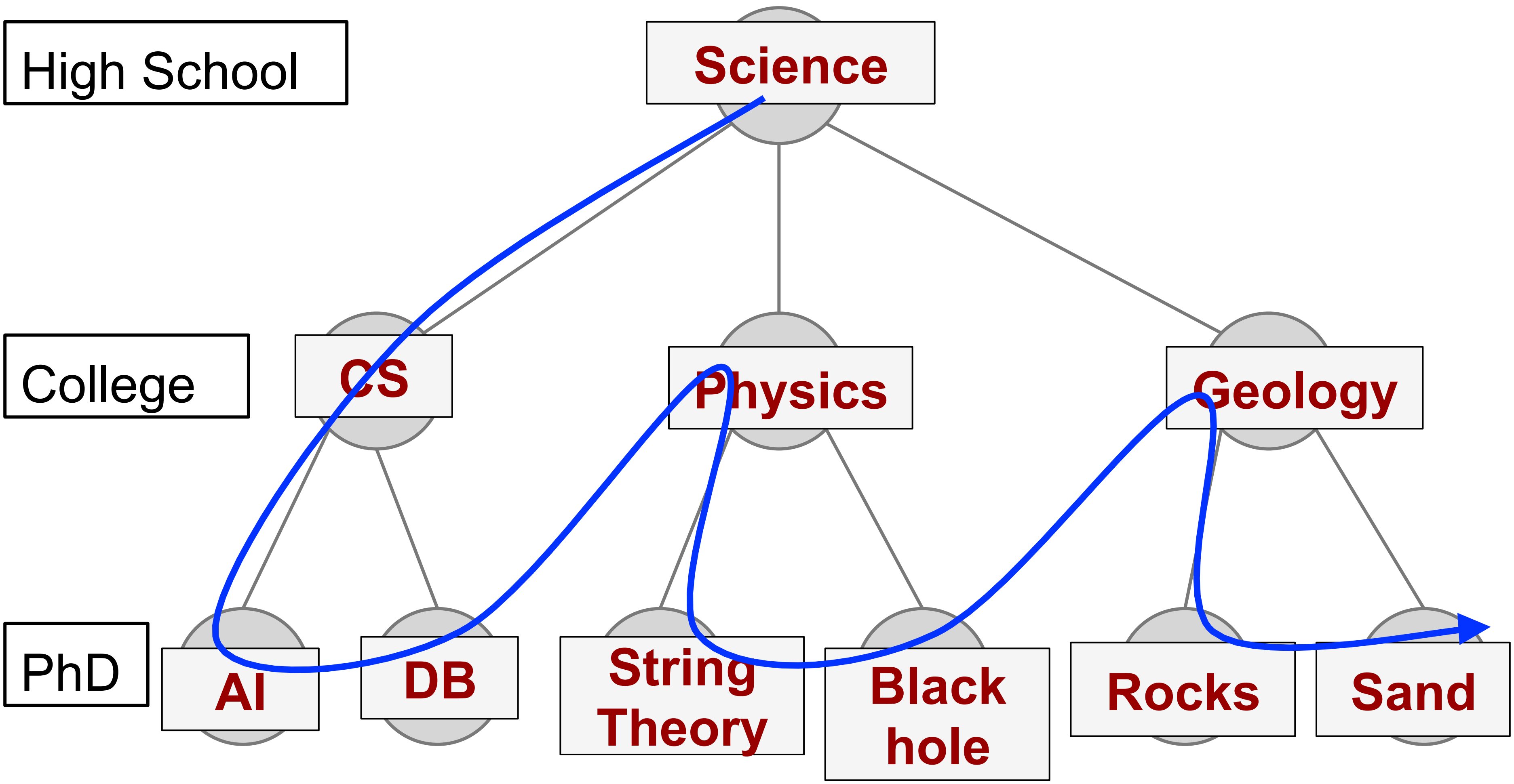
Think about the moment after dequeue u (checking u's neighbours)

- ➔ if v is white, $d[v] = d[u] + 1$ (how BFS works), **contradiction**!
- ➔ if v is black, $d[v] <= d[u]$ (coz v is dequeued before u), **contradiction**!
- ➔ if v is gray, then it is coloured gray by some other vertex w, then $d[v] = d[w] + 1$ and $d[w] <= d[u]$, therefore $d[v] <= d[u] + 1$, **contradiction**!

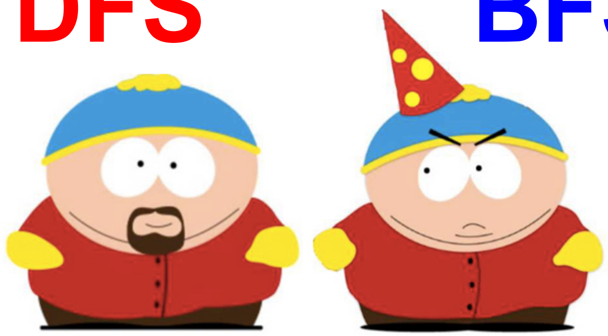
# Depth-First Search

**The Depth-First way of learning these subjects**

➔ Go towards PhD whenever possible; only start learning physics after finishing everything in CS.



**DFS** **BFS**



```
NOT_YET_BFS(root):
  Q ← Queue()
  Enqueue(Q, root)
  while Q not empty:
    x ← Dequeue(Q)
    print x
    for each child c of x:
      Enqueue(Q, c)
```

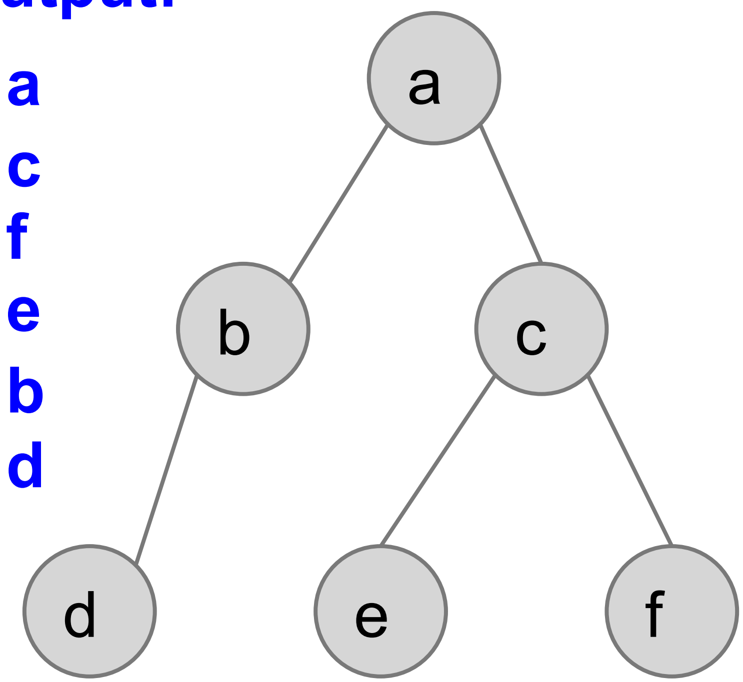
```
NOT_YET_DFS(root):
  Q ← Stack()
  Push(Q, root)
  while Q not empty:
    x ← Pop(Q)
    print x
    for each child c of x:
      Push(Q, c)
```

**Why they are twins!**

# DFS in a tree

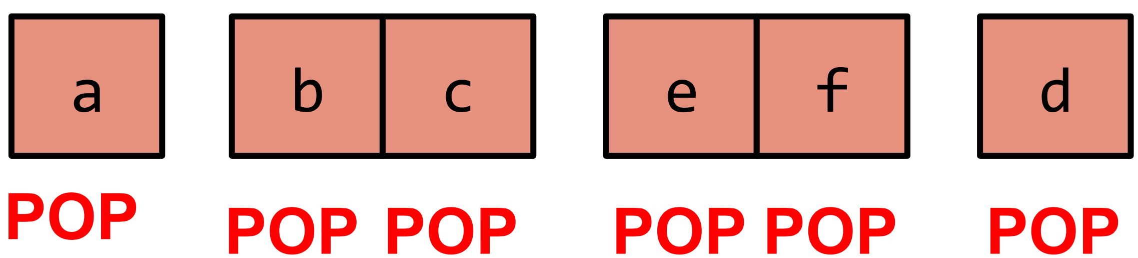
**Output:**

**a**
**c**
**f**
**e**
**b**
**d**




```
NOT_YET_DFS(root):
  Q ← Stack()
  Push(Q, root)
  while Q not empty:
    x ← Pop(Q)
    print x
    for each child c of x:
      Push(Q, c)
```

**Stack:**



# A nicer way to write this code?

The use of stack is basically implementing **recursion**________.

```
NOT_YET_DFS(root):
  Q ← Stack()
  Push(Q, root)
  while Q not empty:
    x ← Pop(Q)
    print x
    for each child c of x:
      Push(Q, c)
```

```
NOT_YET_DFS(root):
  print root
  for each child c of x:
      NOT_YET_DFS(c)
```

Exercise: Try this code on the tree in the previous slide.

# Avoid visiting a vertex twice, same as BFS

Remember you visited it by **labelling** it using **colours**.

→**White**: "unvisited"

→**Gray**: "encountered"

→**Black**: "explored"


- → Initially all vertices are **white**
- → Colour a vertex **gray** the **first** time visiting it
- → Colour a vertex **black** when **all** its **neighbours** have been encountered
- → Avoid visiting **gray** or **black** vertices
- → In the end, all vertices are **black**

# Other values to remember, some are same as BFS

➔ **pi[v]**: the vertex from which v is encountered

- "I was introduced as **whose** neighbour?"

# Other values to remember, different from BFS

- ➔There is a **clock** ticking, incremented whenever someone's colour is changed
- ➔For each vertex v, remember two **timestamps**
  - ◆**d[v]**: "discovery time", when the vertex is first encountered
  - ◆**f[v]**: "finishing time", when all the vertex's neighbours have been visited.

**Note : this d[v] is totally different from that distance value d[v] in BFS!**

# The pseudo-code (incomplete)

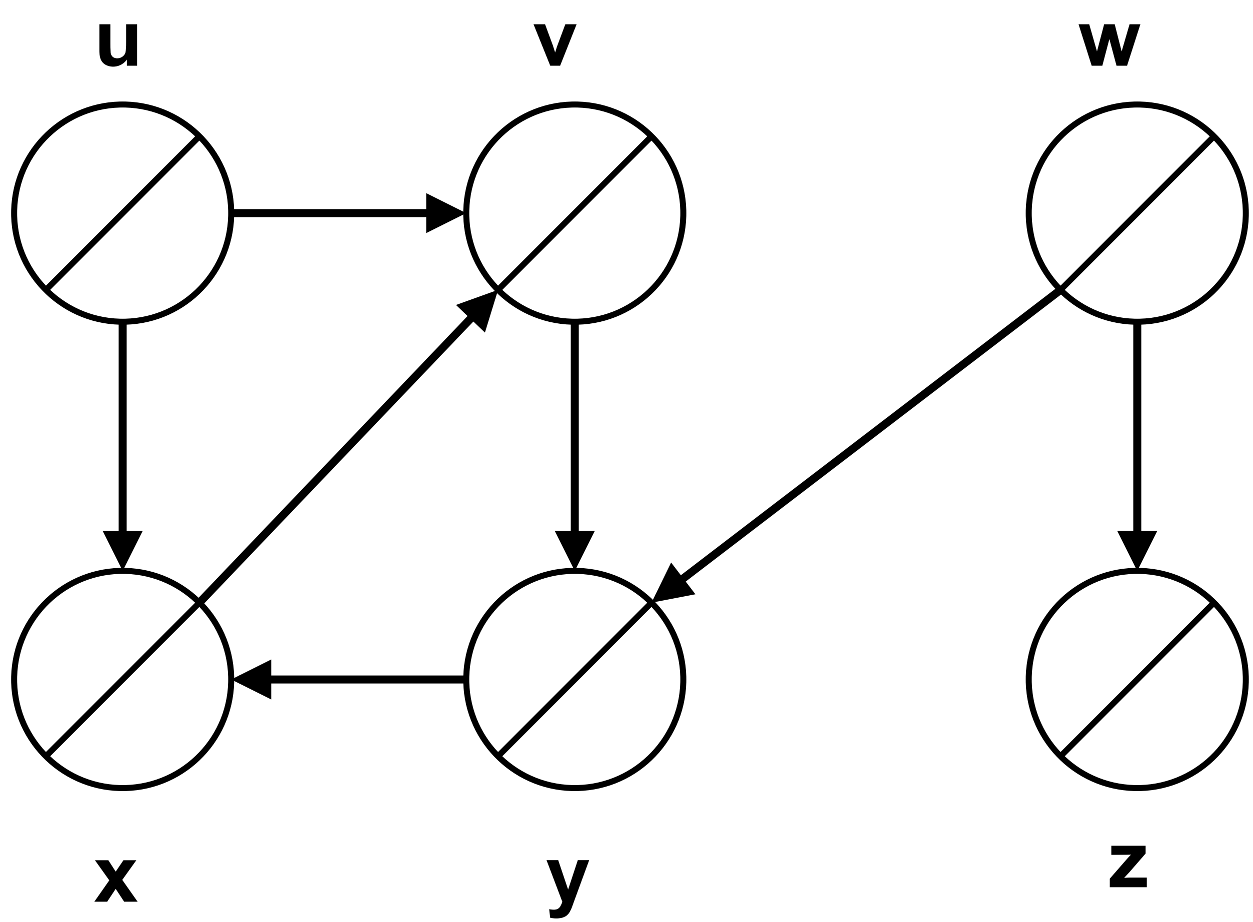
```
DFS_VISIT(G, u):
    colour[u] ← gray
    time ← time + 1
    d[u] ← time        # keep discovery time on first encounter
    for each neighbour v of u:
        if colour[v] = white:
            pi[v] ← u
            DFS_VISIT(G, v)
    colour[u] ← black
    time ← time + 1
    f[u] ← time        # keep finishing time after exploring all neighbours
```

The red part is the same as NOT_YET_DFS

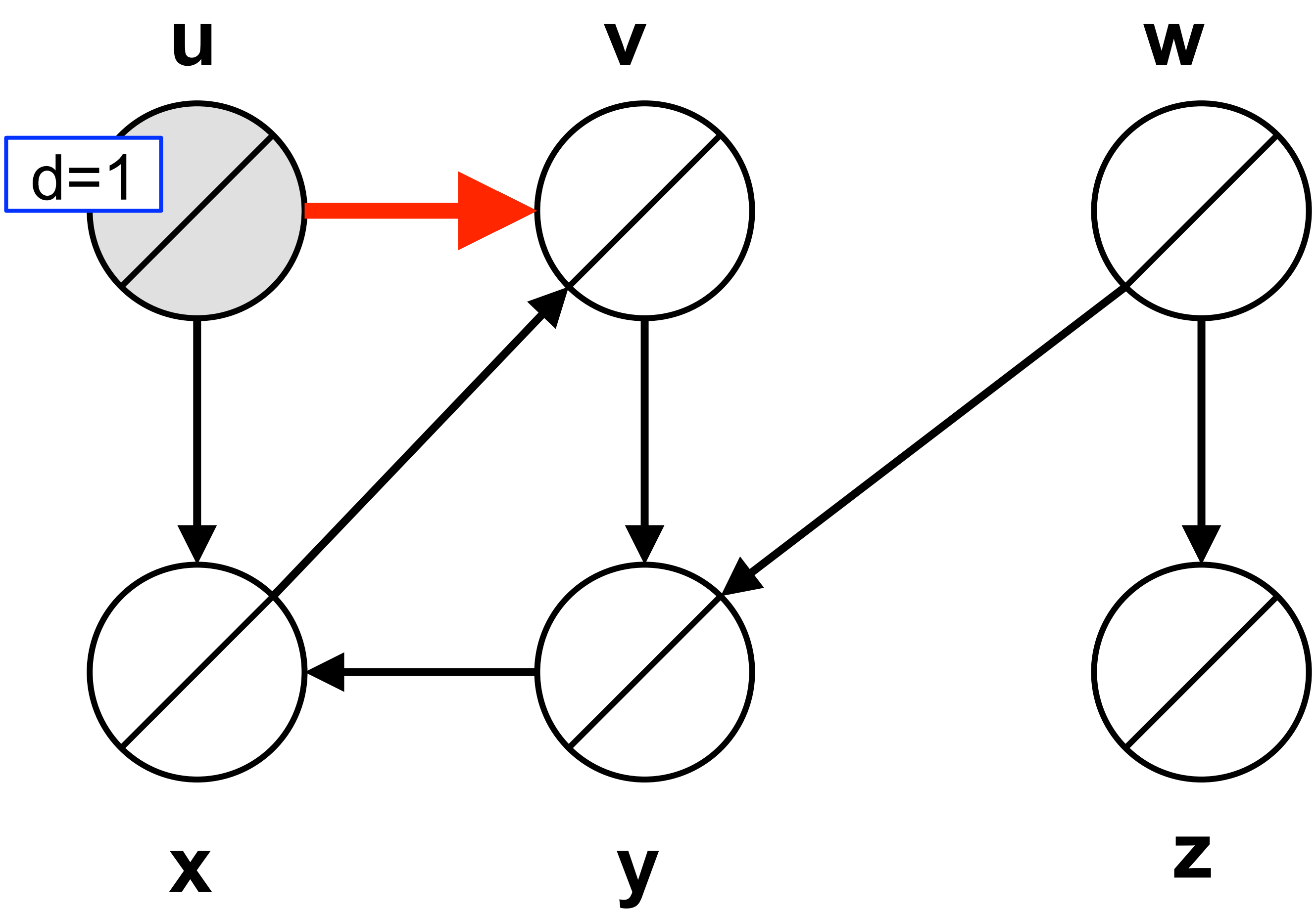
Why **DFS_VISIT** instead of **DFS**? We will see...

# Let’s run an example!

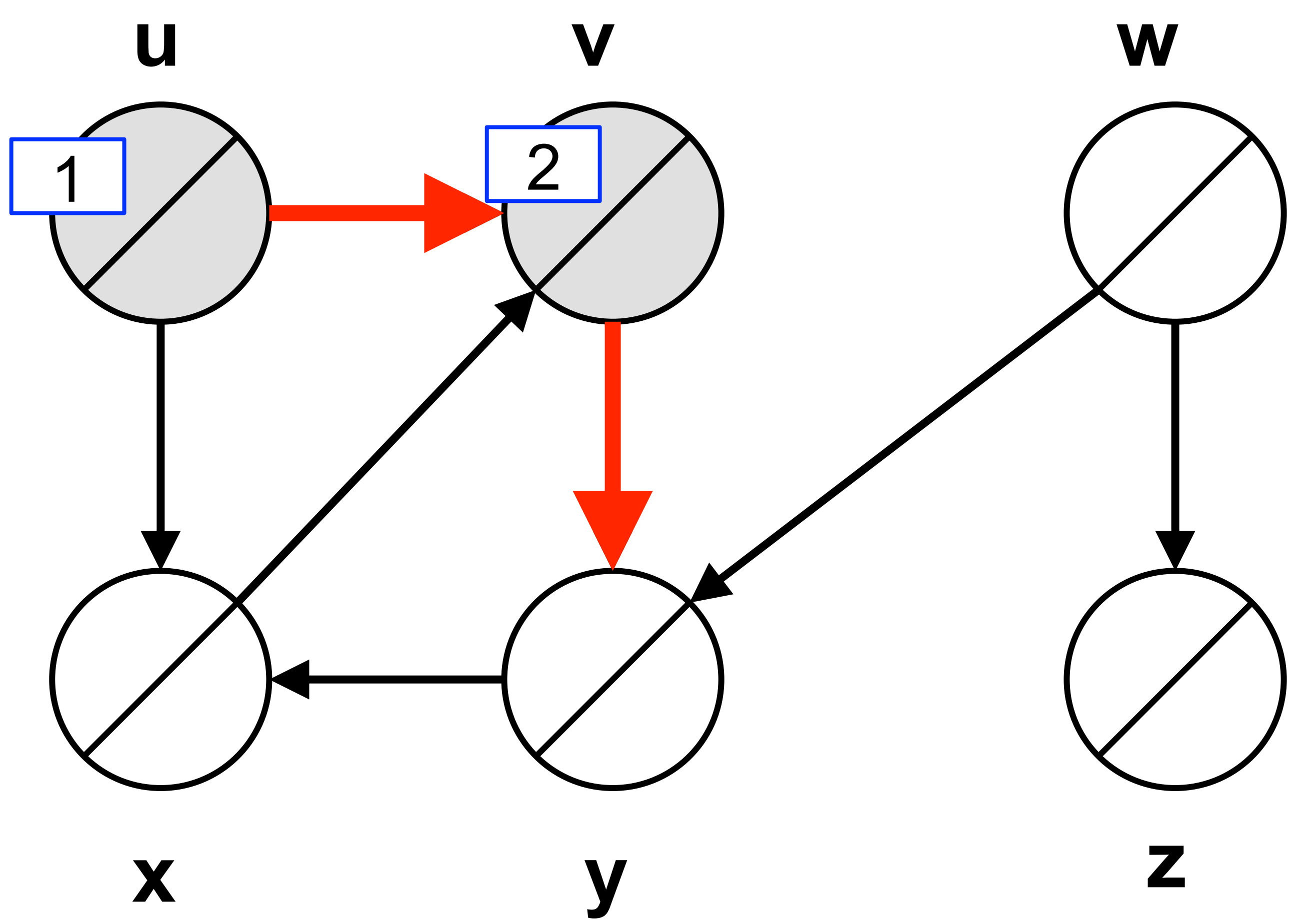
DFS_VISIT(G, u)



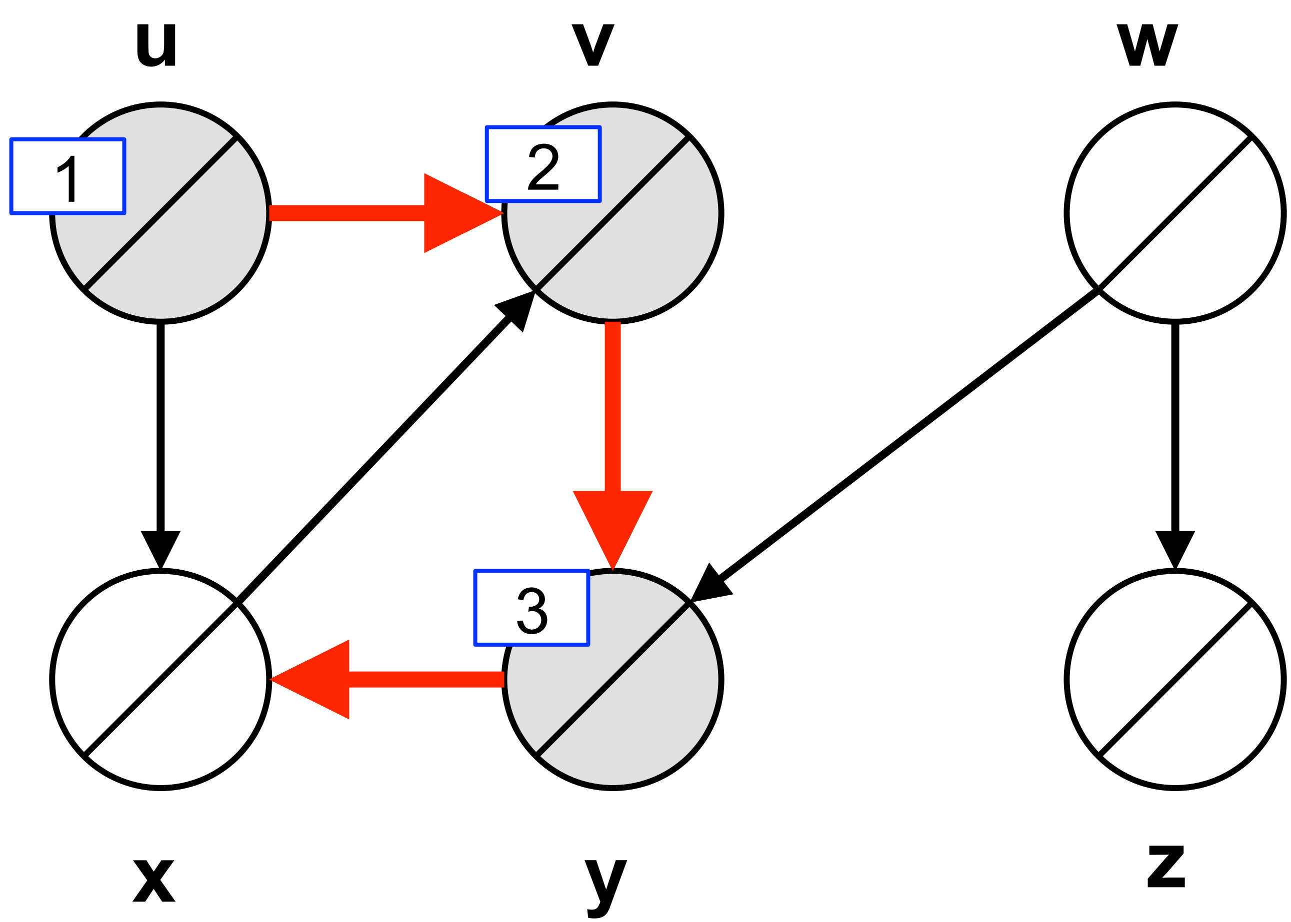
**time = 1,** encounter the source vertex



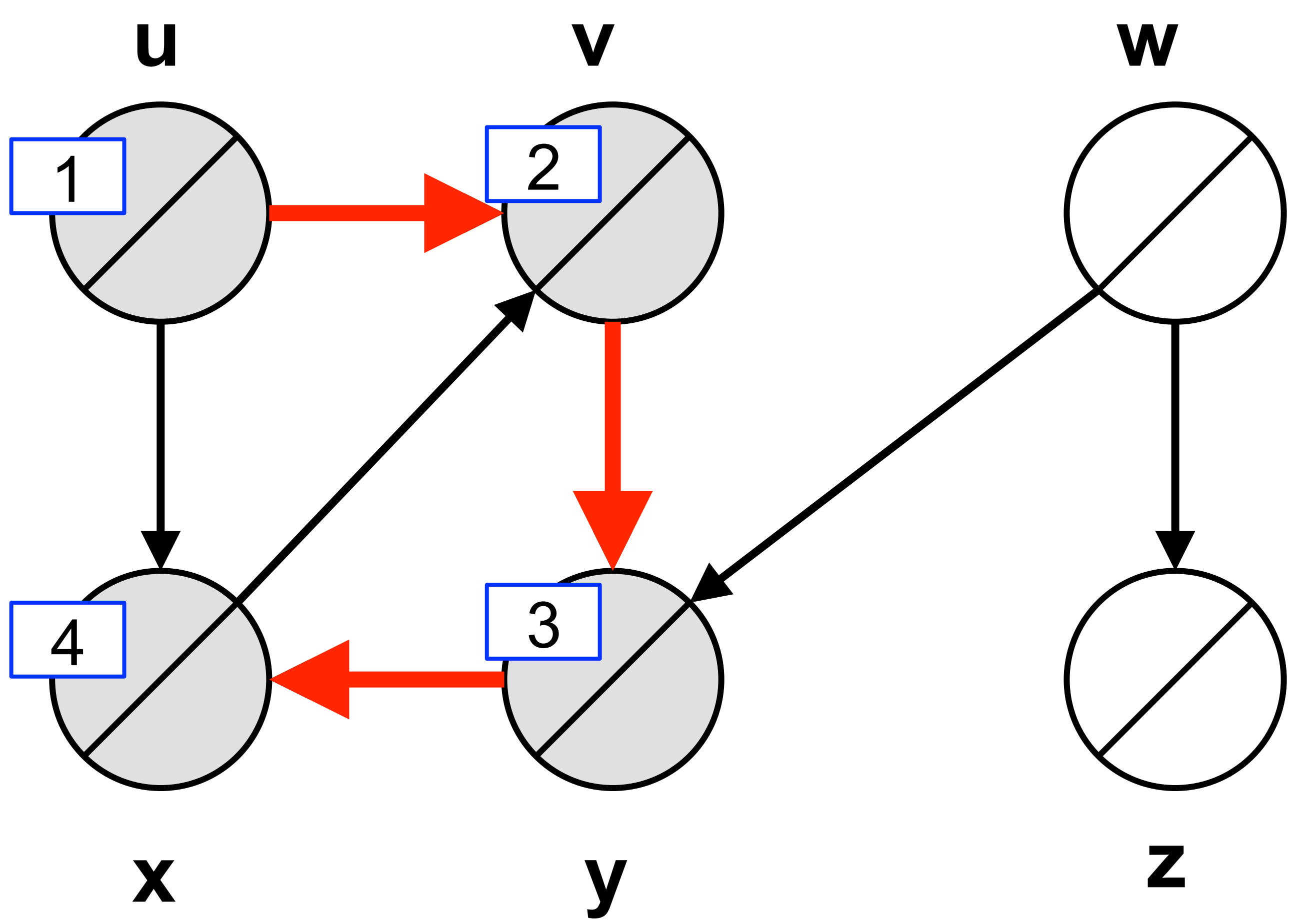
**time = 2,** recursive call, level 2



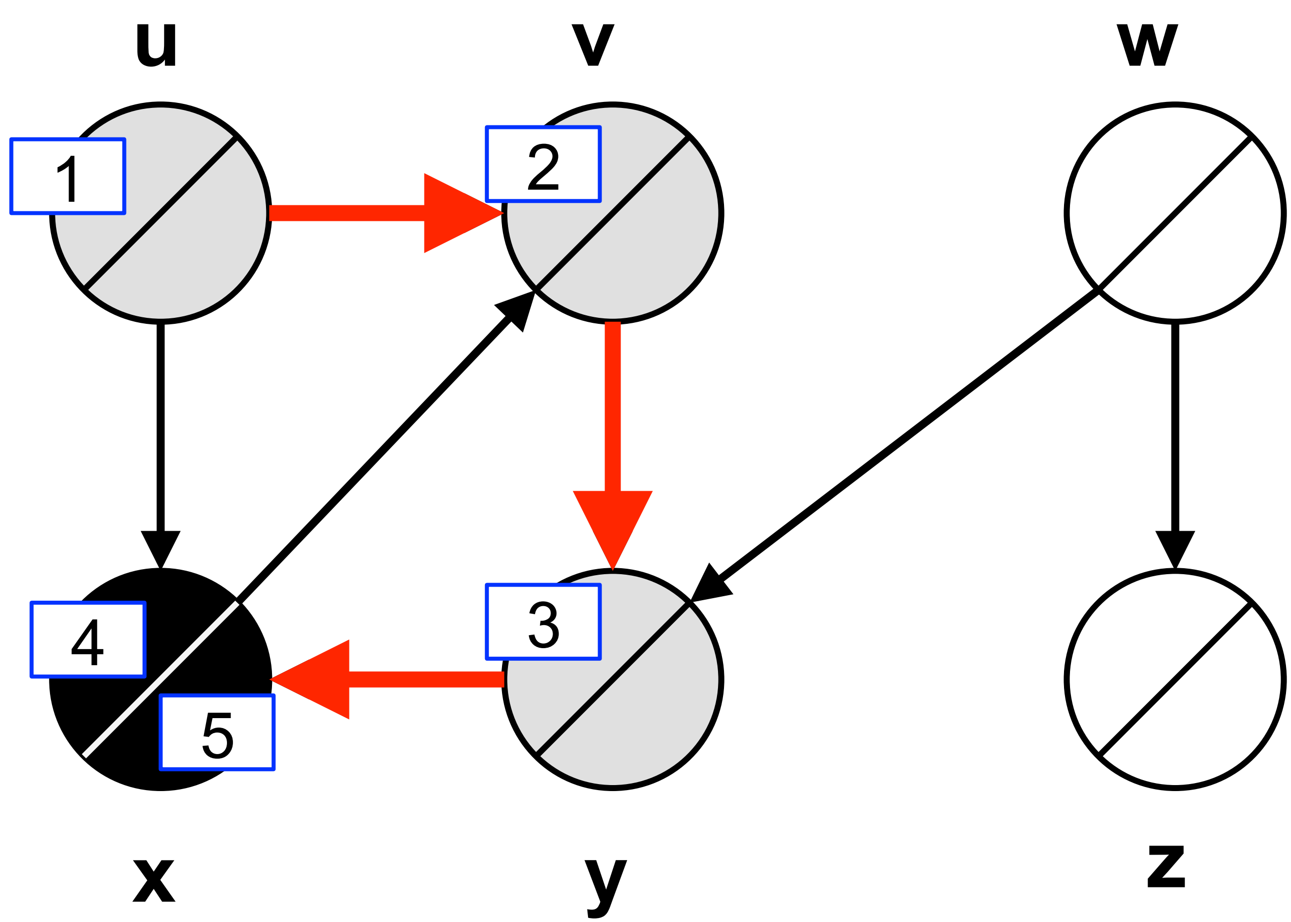
**time = 3,** recursive call, level 3



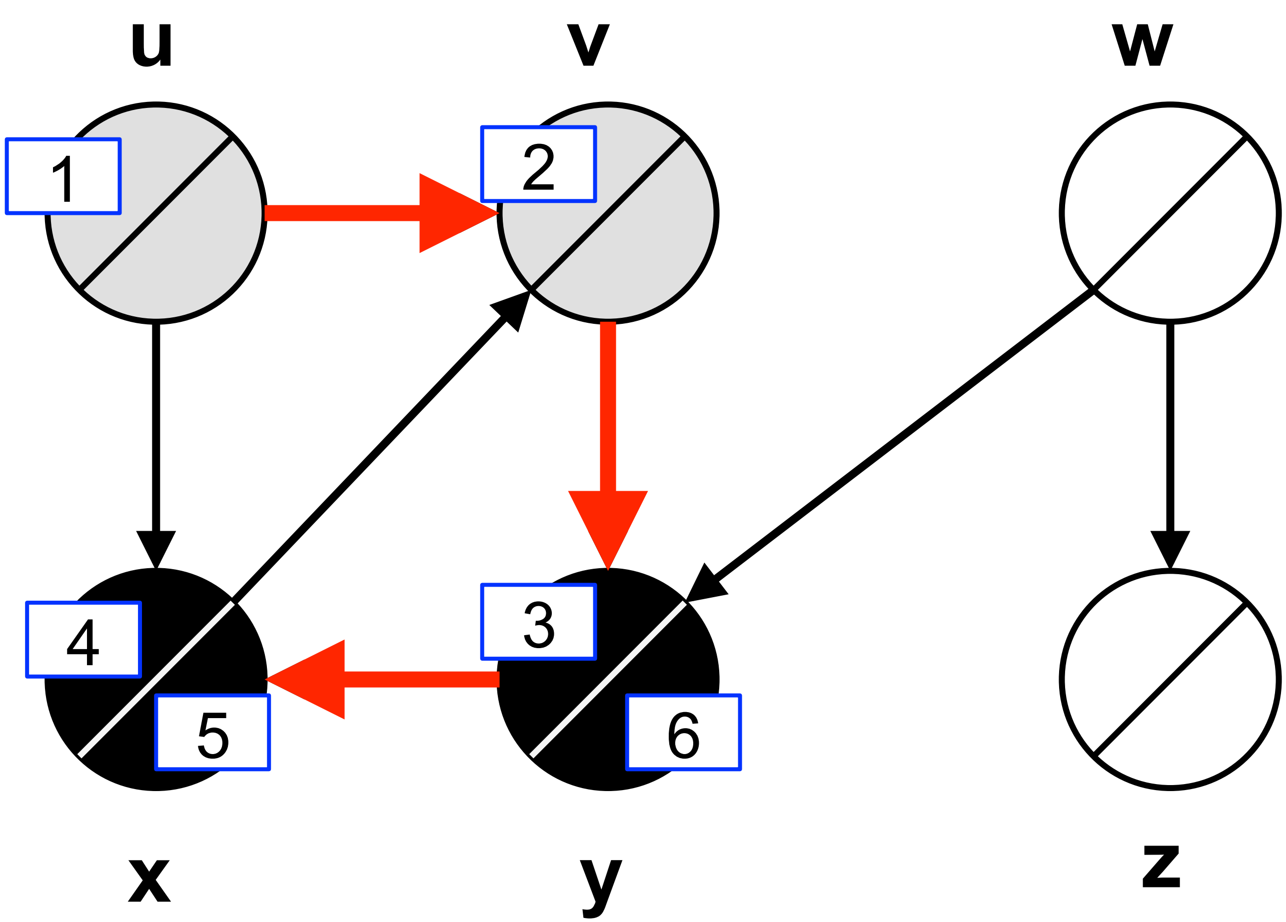
**time = 4,** recursive call, level 4



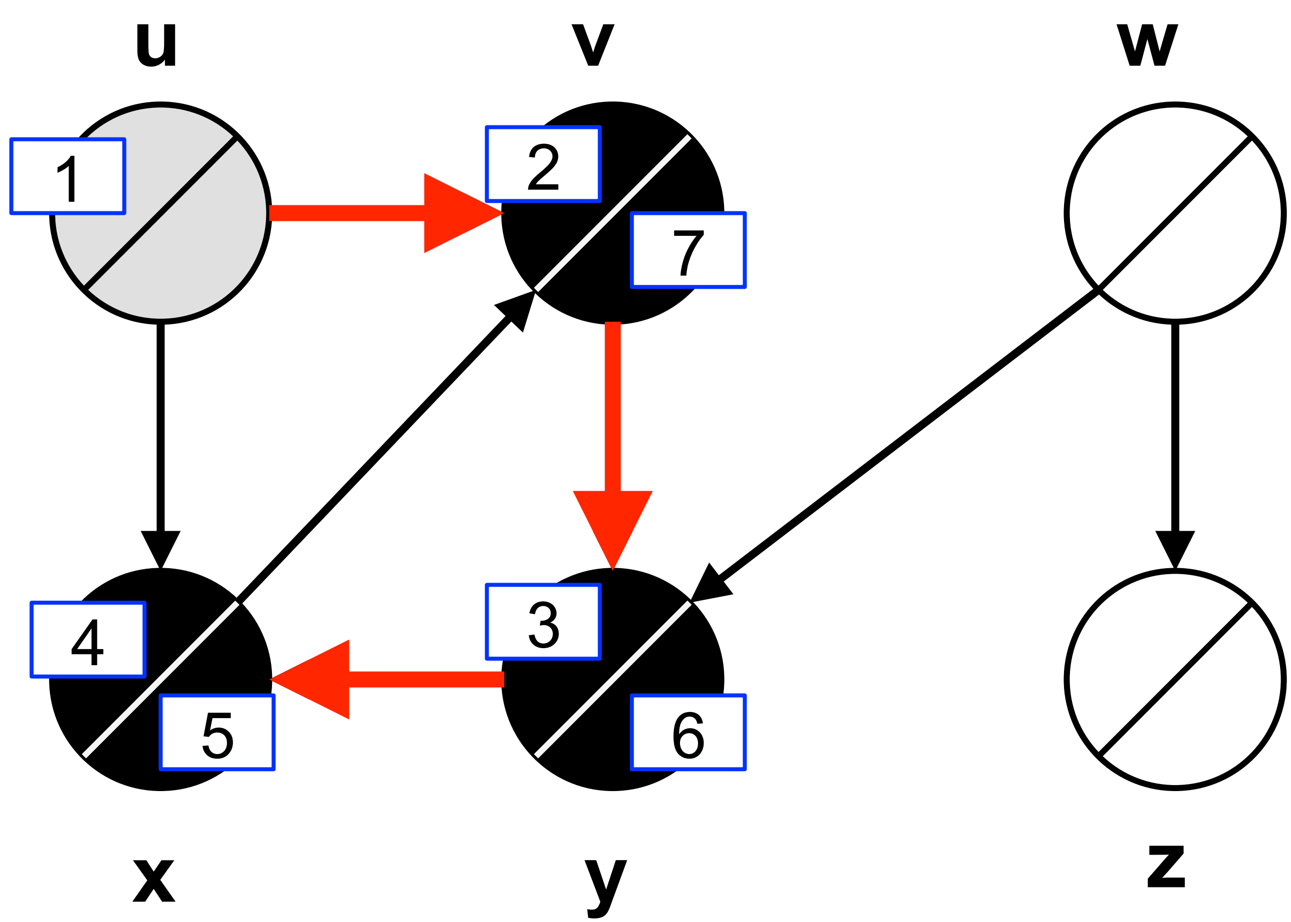
**time = 5,** vertex x finished



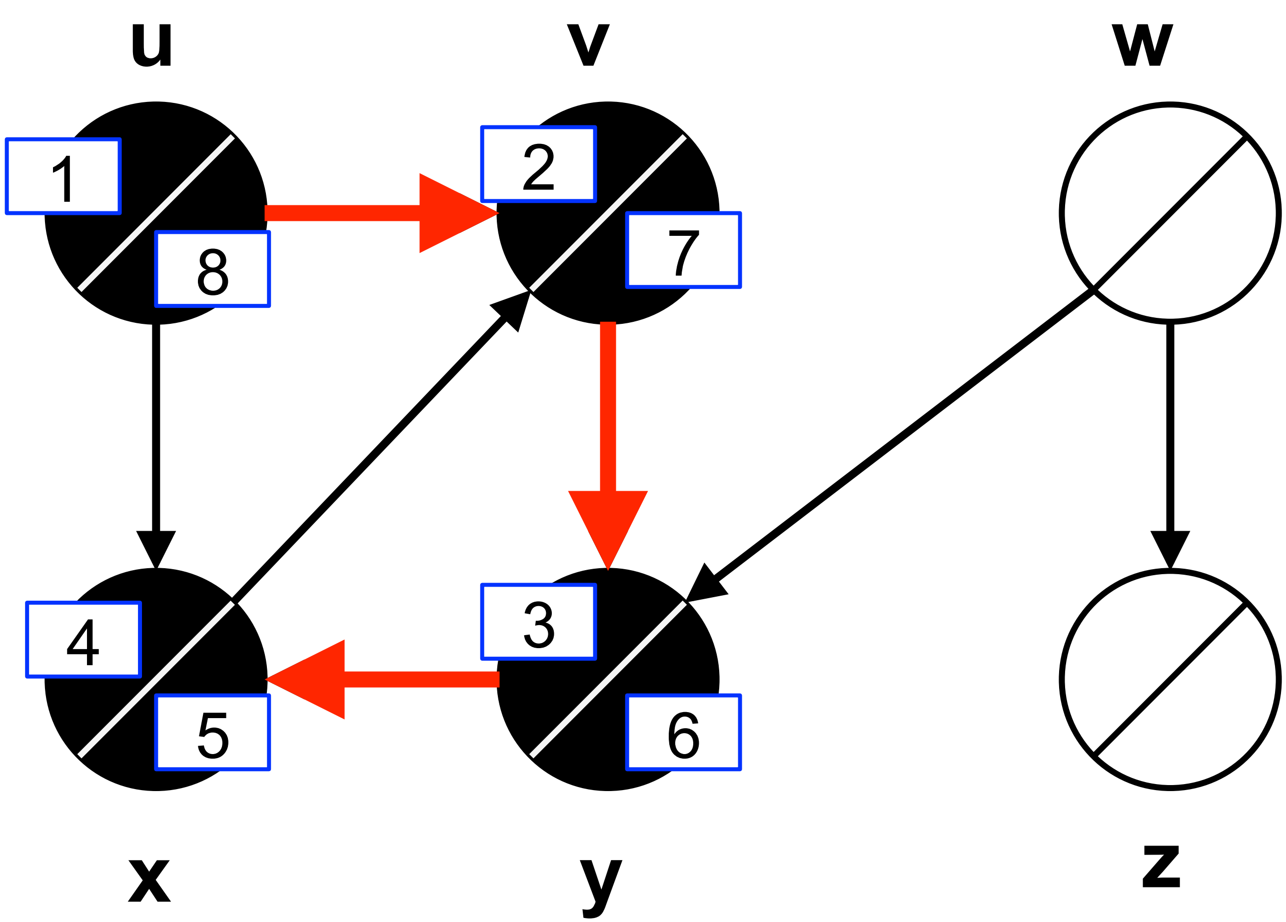
**time = 6,** recursion back to level 3, finish y



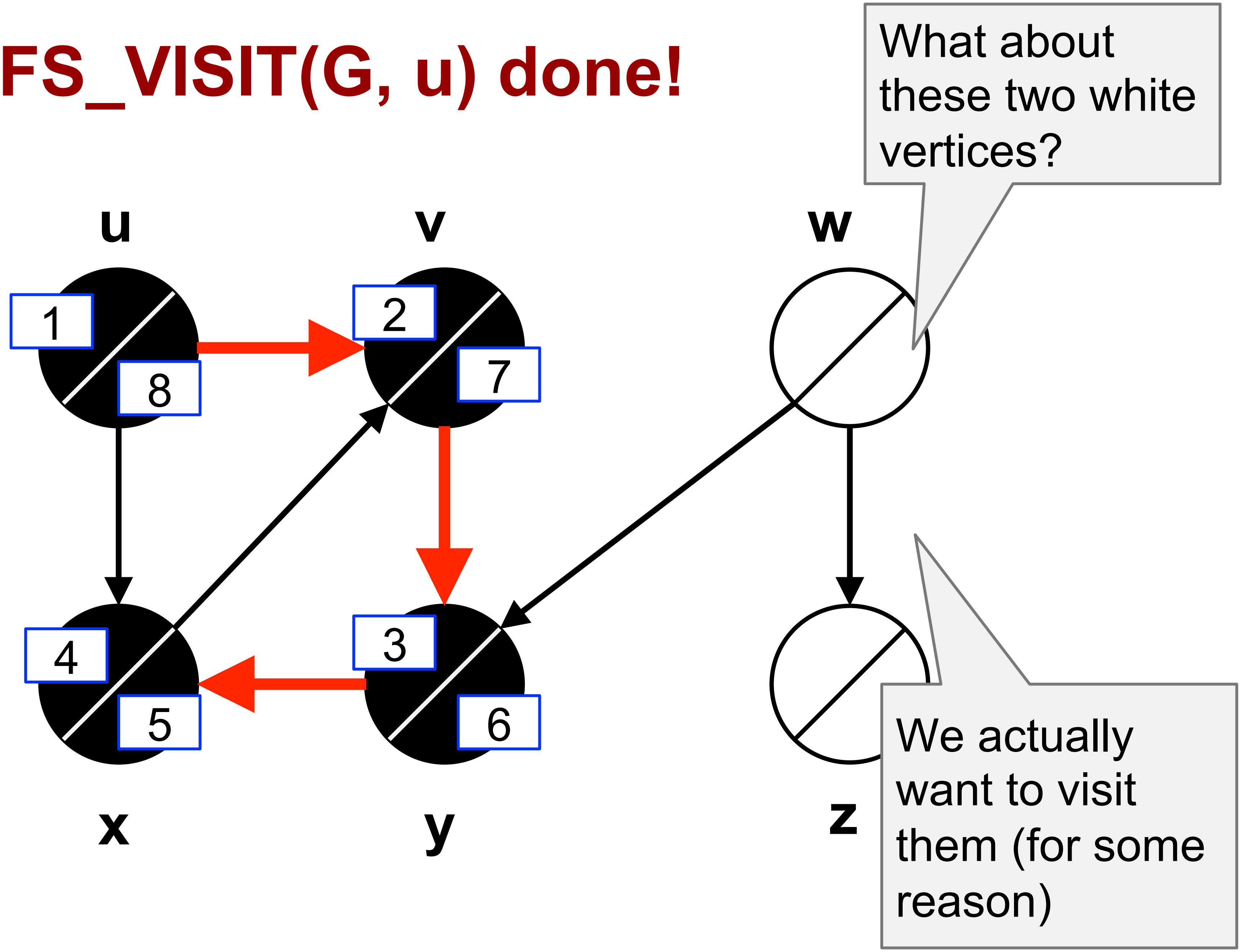
**time = 7,** recursive back to level 2, finish v



**time = 8,** recursion back to level 1, finish u



# DFS_VISIT(G, u) done!



# The pseudo-code for visiting everyone

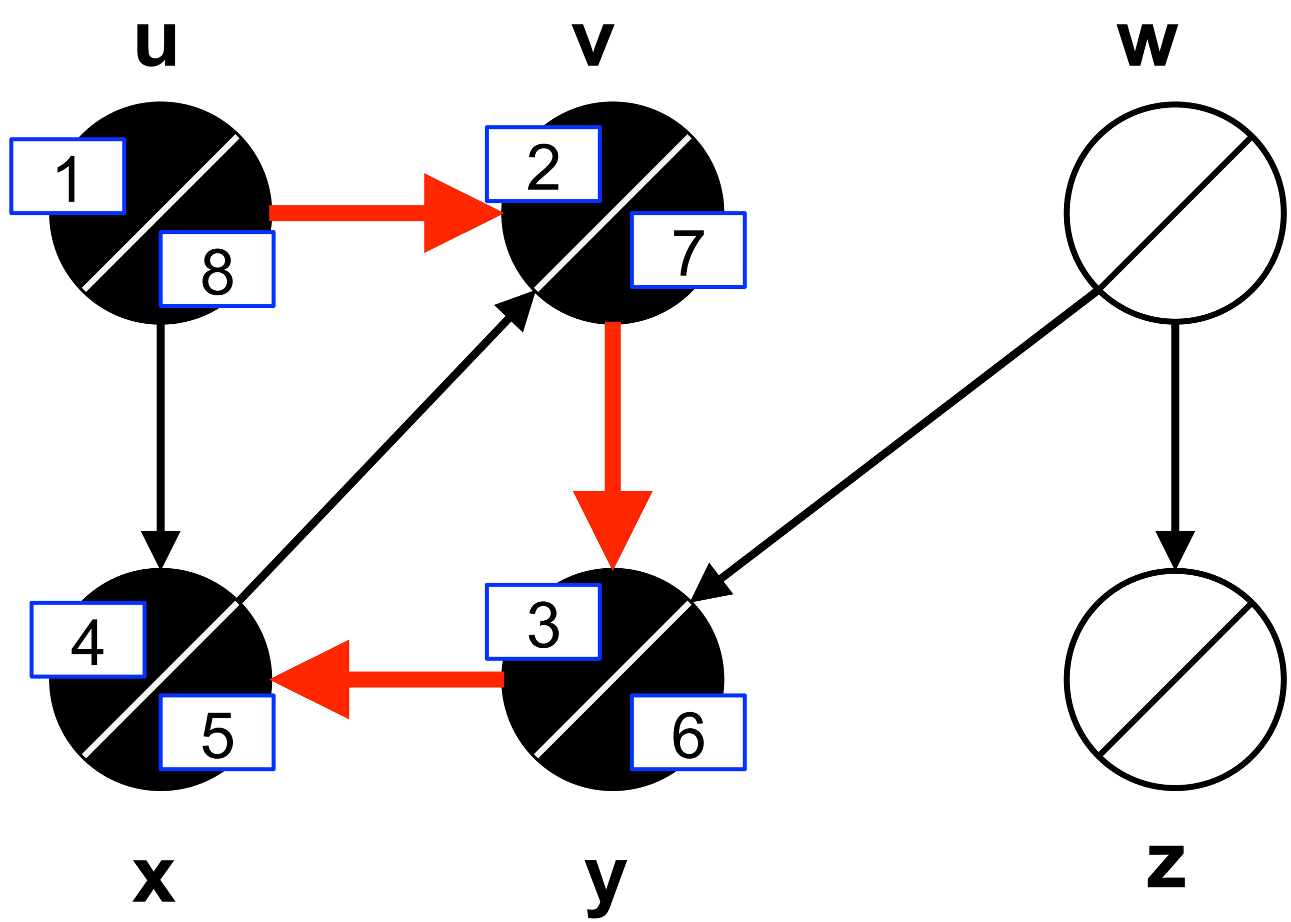
```
DFS(G):
    for each v in G.V:
        colour[v] ← white
        f[v] ← d[v] ← ∞
        pi[v] ← NIL
    time ← 0
    for each v in G.V:
        if colour[v] = white:
            DFS_VISIT(G, v)
```

Initialization

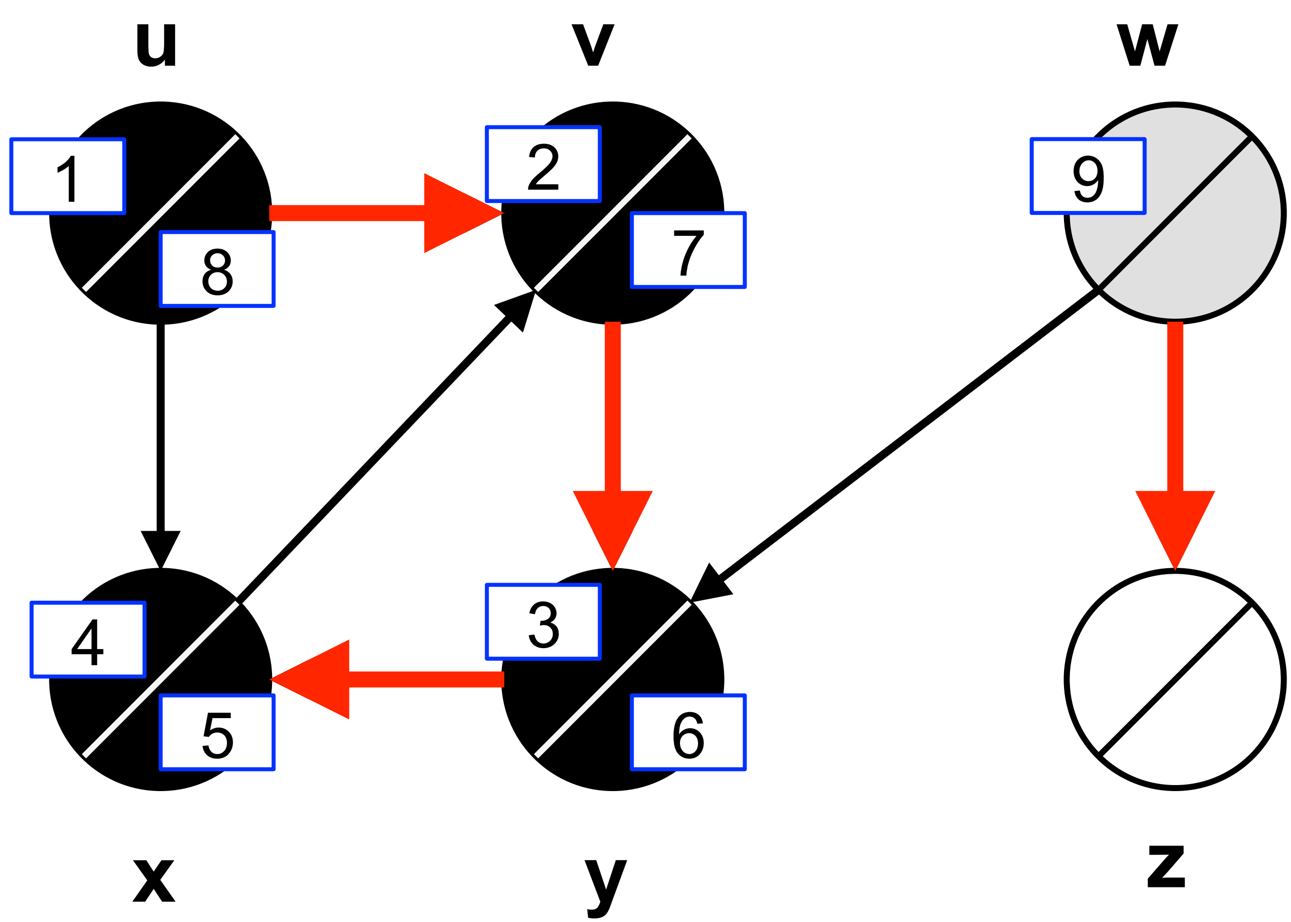
Make sure NO vertex is left with white colour.

```
DFS_VISIT(G, u):
    colour[u] ← gray
    time ← time + 1
    d[u] ← time
    for each neighbour v of u:
        if colour[v] = white:
            pi[v] ← u
            DFS_VISIT(G, v)
    colour[u] ← black
    time ← time + 1
    f[u] ← time
```

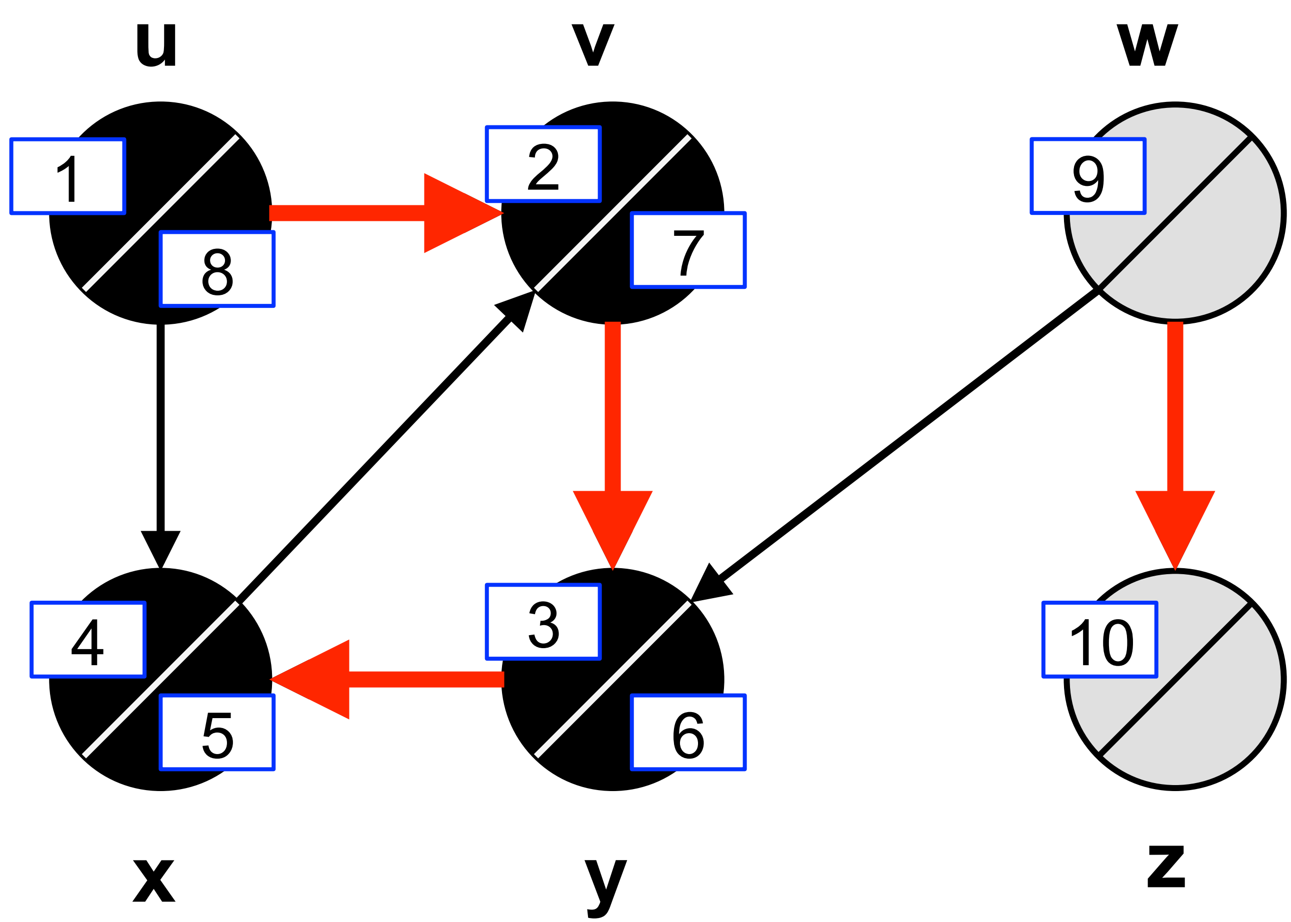
# So, let's finish this DFS



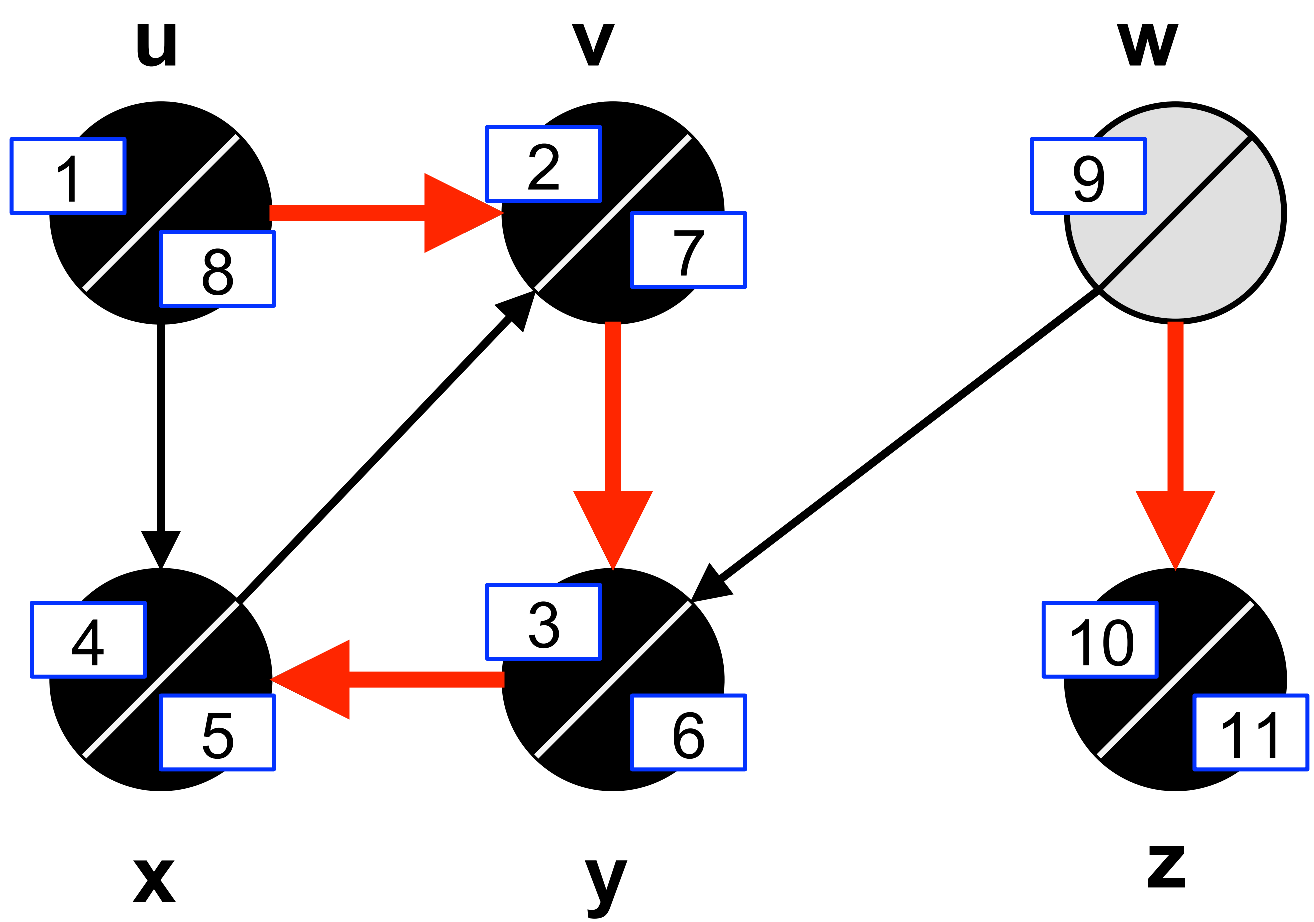
# time = 9, DFS_VISIT(G, w)



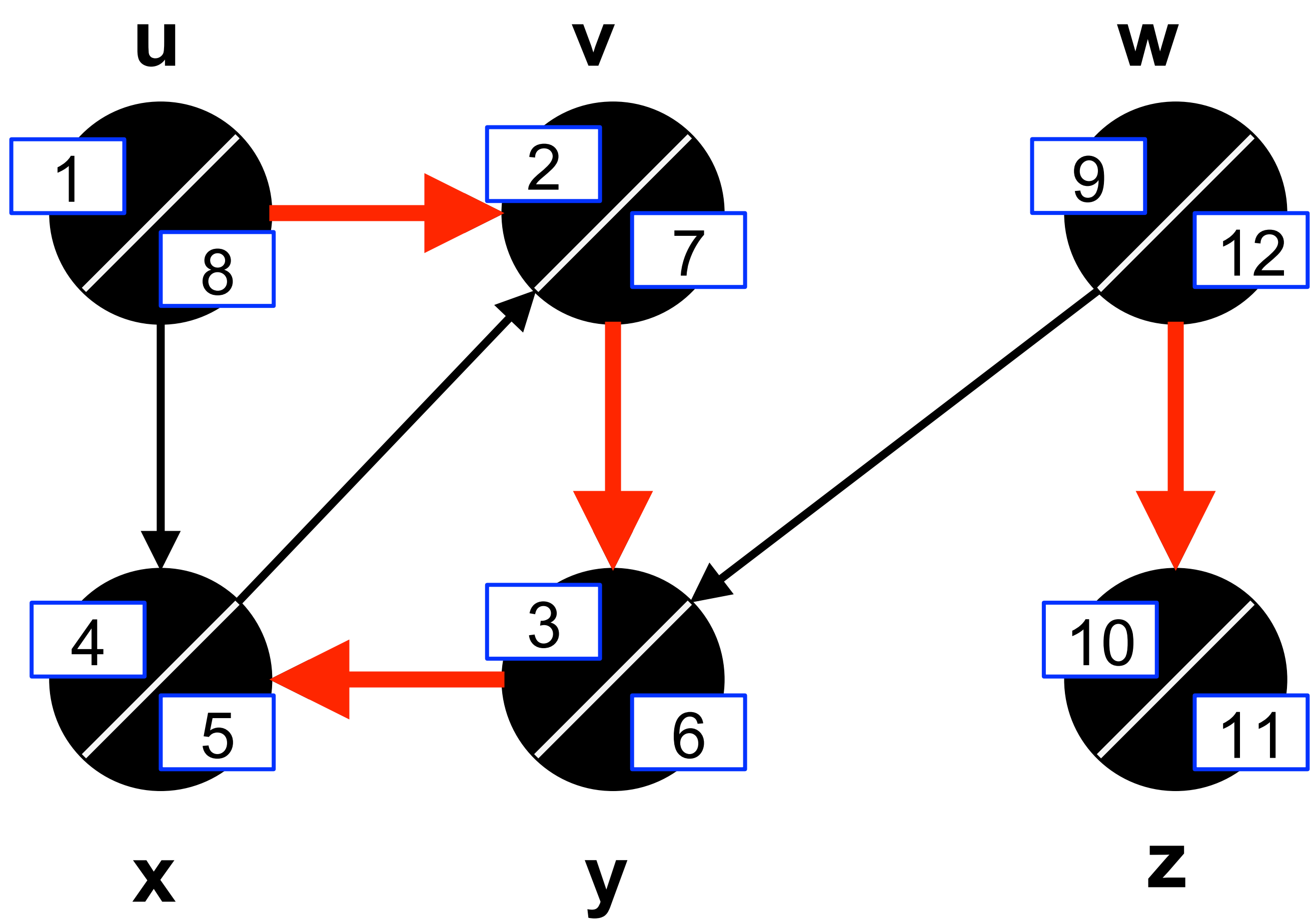
# time = 10



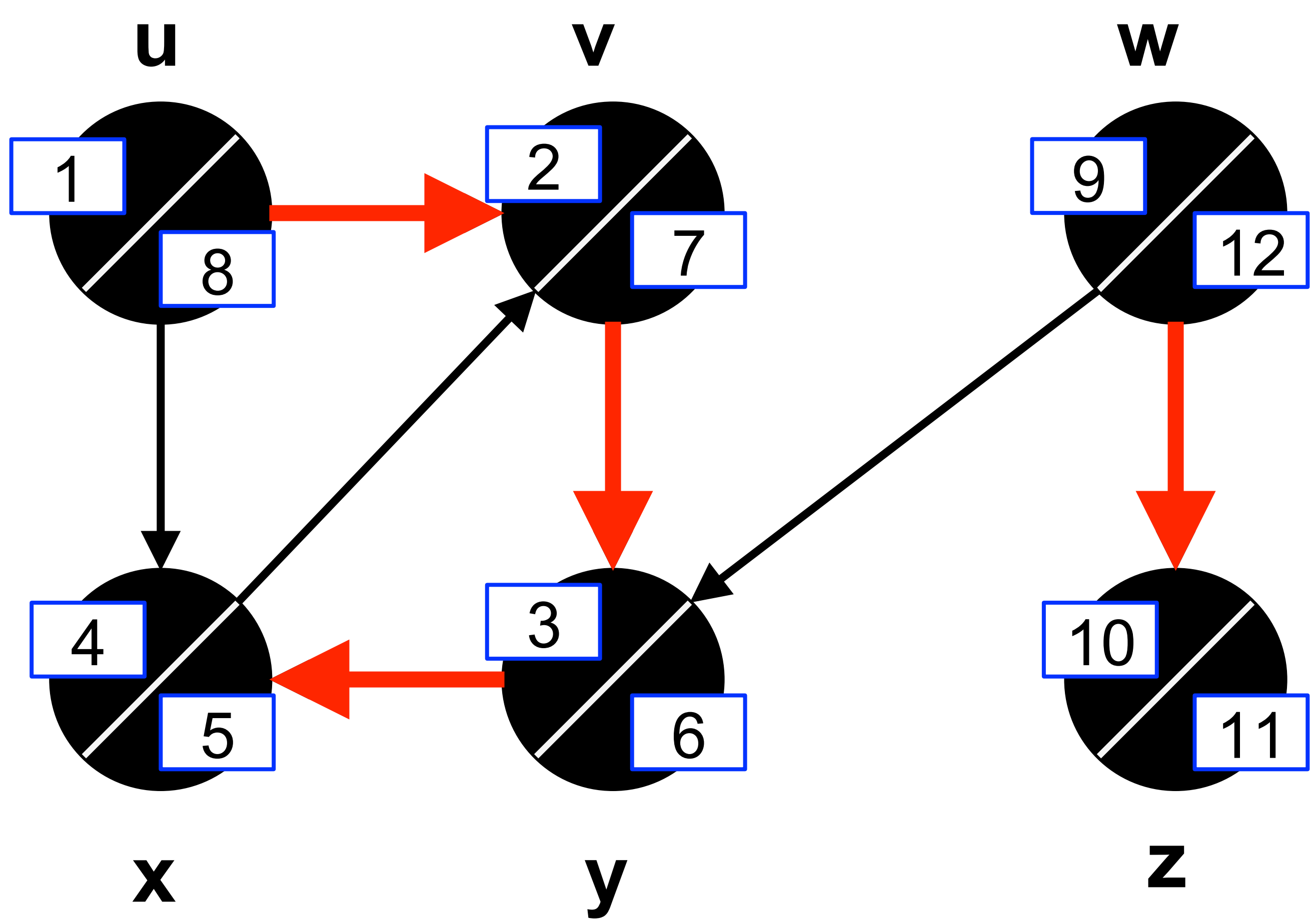
# time = 11



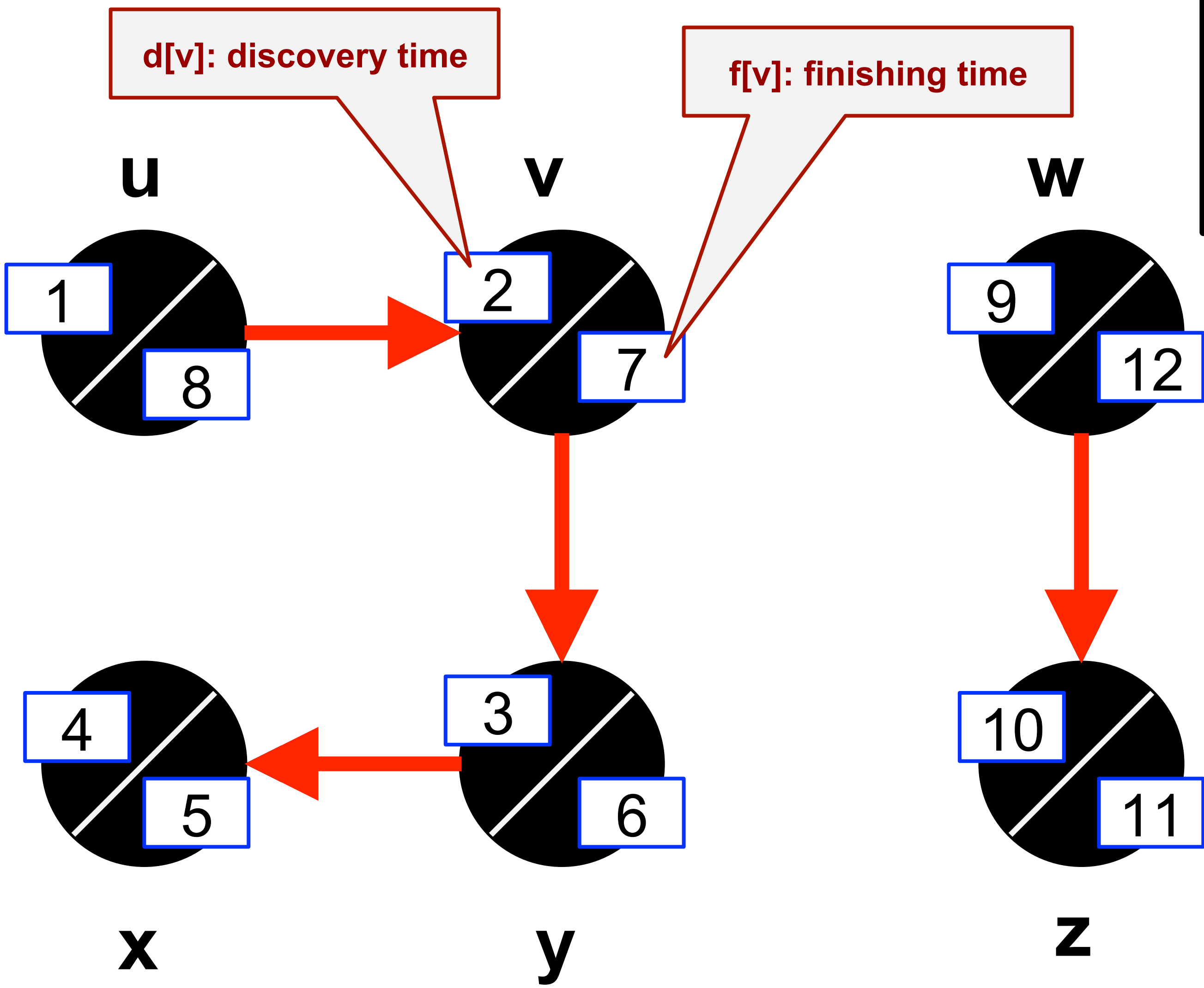
# time = 12



# DFS(G) done!



# Recap




We get a DFS **forest** ( a set of disjoint trees)

# Runtime analysis!

The total amount of work (use **adjacency list**):

- ➔ Visit each vertex once
  - constant work per vertex
  - in total: **O(|V|)**
- ➔ At each vertex, check all its neighbours (all its **incident edges**)
  - Each edge is checked **once** (in a directed graph)
  - in total: **O(|E|)**

**Same as BFS**

**Total runtime:**
**O(|V|+|E|)**

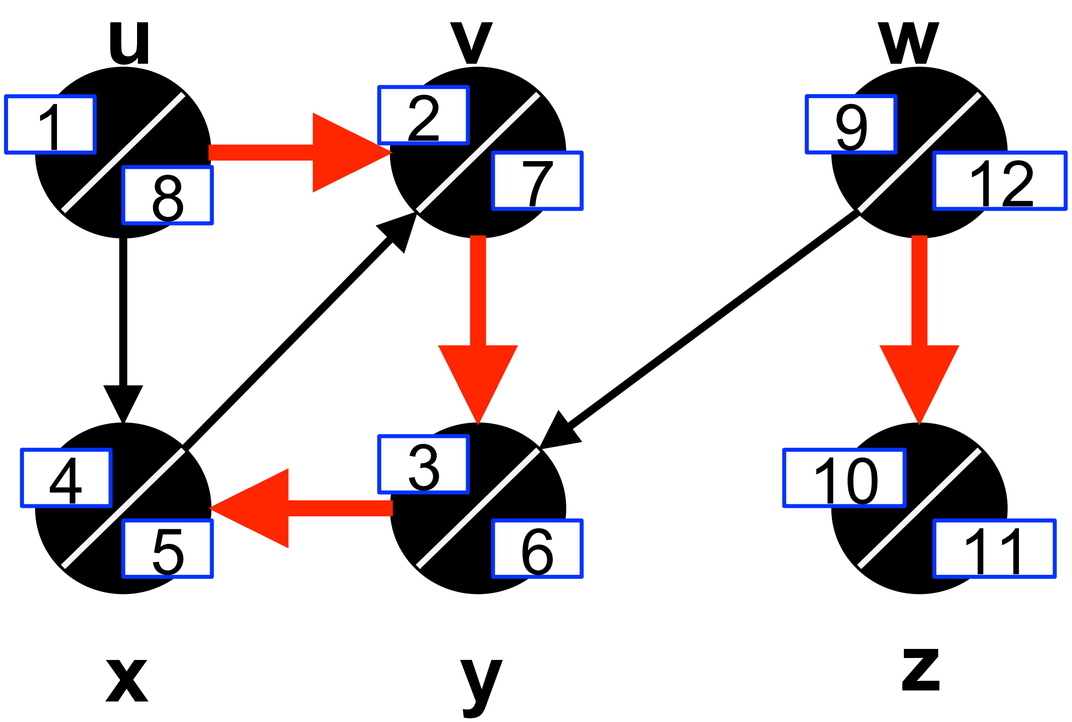
# What do we get from DFS?

- ➔Detect whether a graph has a cycle.
  - ◆That's why we wanted to visit all vertices -- if you want to be sure whether a graph has a cycle or not, you'd better check **everywhere**.
  - ◆Why didn't we do the similar thing for BFS?
- ➔How exactly do we detect a cycle?

**determine descendant / ancestor relationship in the DFS forest**

How to decide whether **y** is a **descendant** of **u** in the DFS forest?

**Idea #1**: trace back the **pi[v]** pointers (the red edges) starting from **y**, see whether you can get to **u**.
Worst-case takes **O(n)** steps.




# the “parenthesis structure”

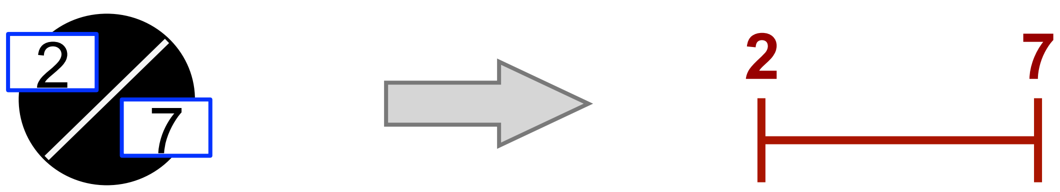
( ( ( ) ) ) ( ) ( ( ) )

➔ Either one pair **contains** the another pair.
➔ Or one pair is **disjoint** from another

( ( ) )

This (overlapping)
never happens!

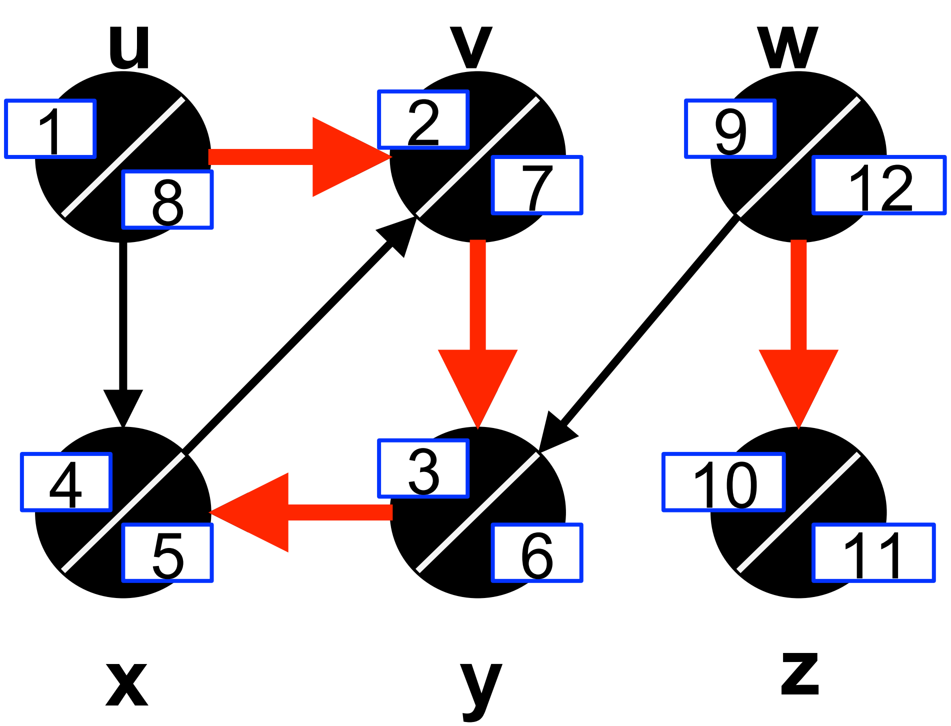
**Visualize d[v], f[v] as interval [ d[v], f[v] ]**

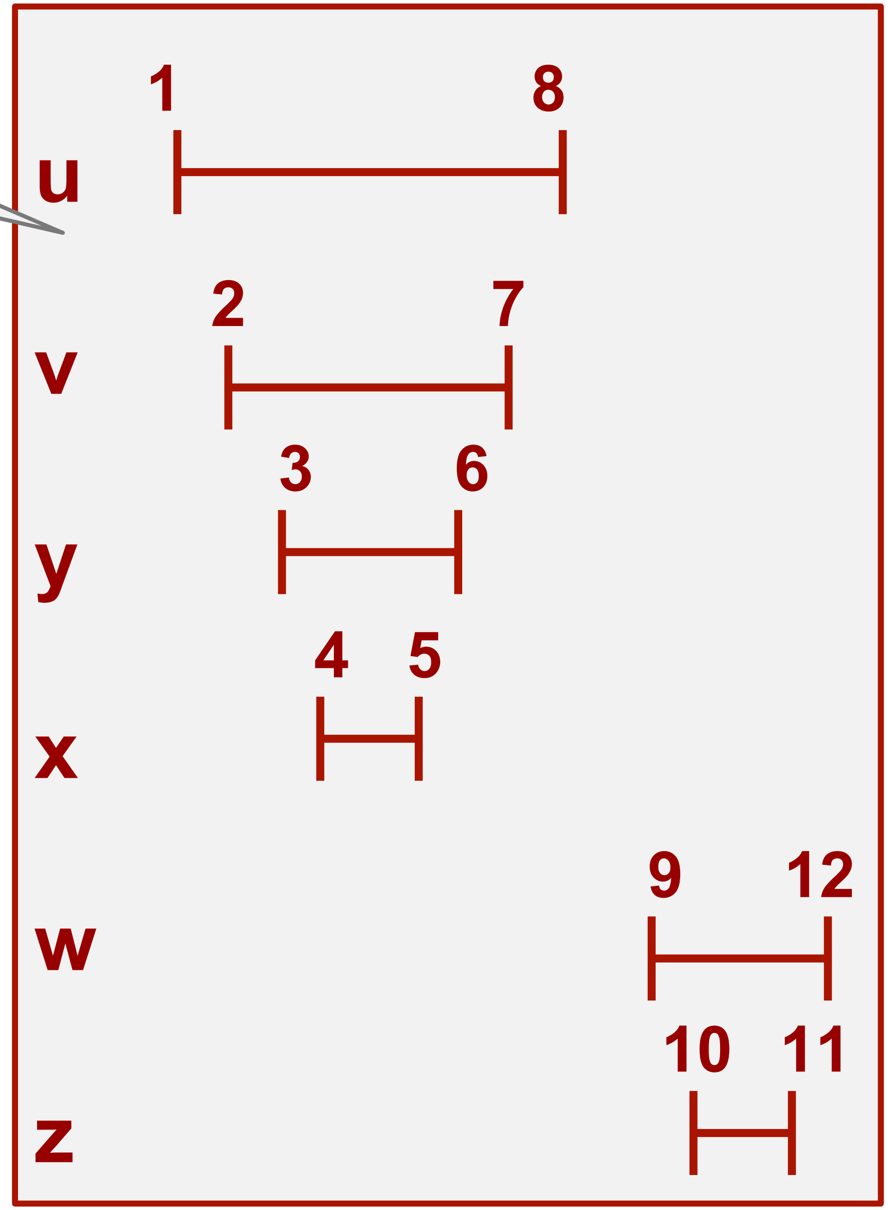


# Now, visualize all the intervals!

What do you see in this?

**Parenthesis structure!**



The **[ d[v], f[v] ]** intervals that we got from DFS follow the parenthesis structure, i.e.,

- ➔Either one interval **contains** another
- ➔Or one is **disjoint** from another

Moreover,

- ➔**Iff** interval of **u** contains interval of **v**, then **u** is an **ancestor** of **v** in the DFS forest.
- ➔If interval of **u** is disjoint from interval of **v**, then they are **not** ancestors of each other.

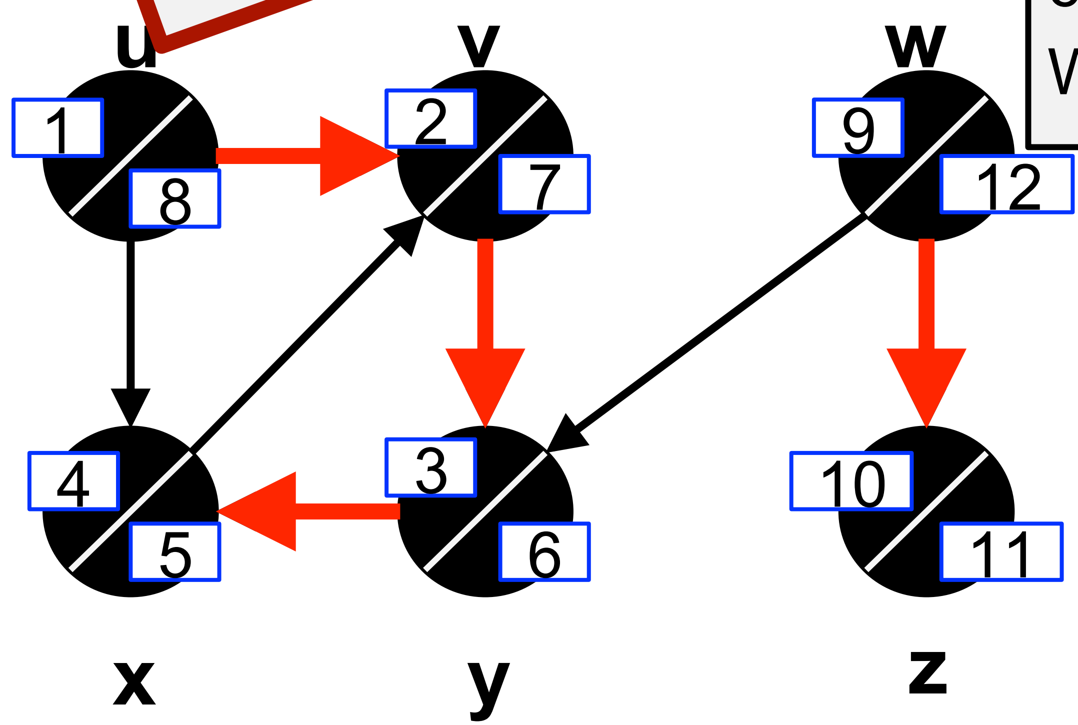
How to decide whether **y** is a **descendant** of **u** in the DFS forest?

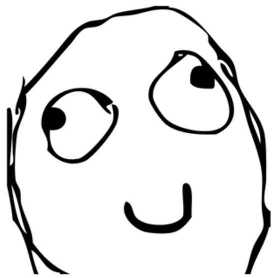
**Idea #1**: trace b... ters (the r... y, see ... to **u**.

W... akes **O(n)** steps.

FORGET ABOUT IT

**Idea #2**: see if **[d[u], f[u]]** contains **[d[y], f[y]].** Worst-case: **1 step**!





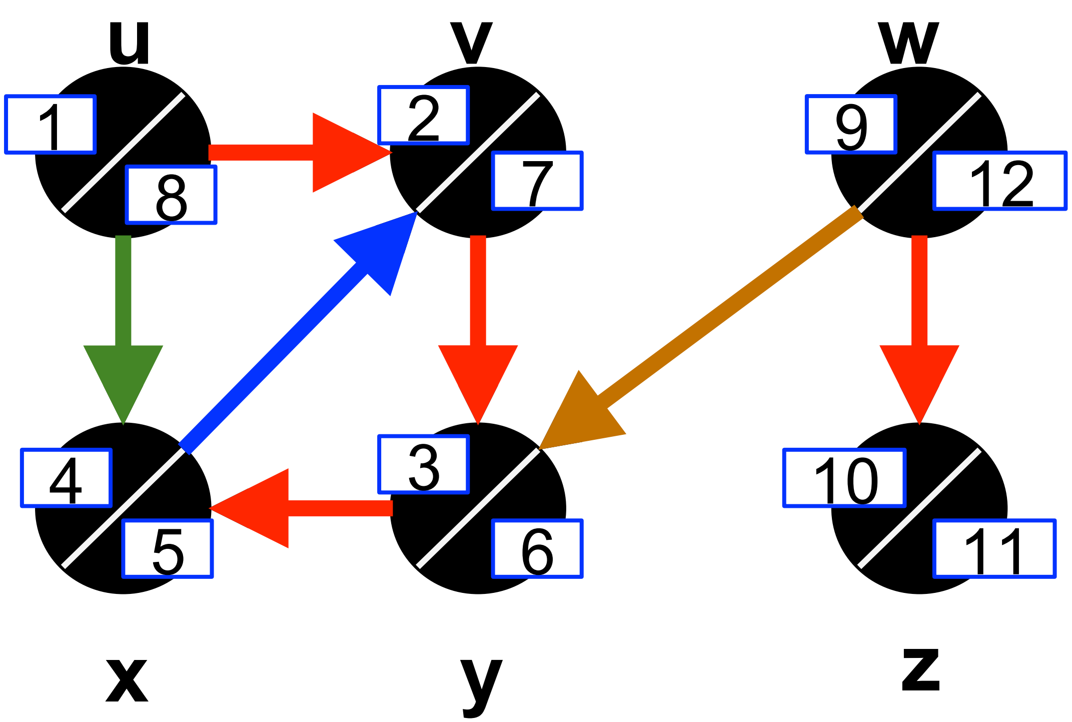
We can efficiently check whether a vertex is an ancestor of another vertex in the DFS forest.

so what...

# Classifying Edges

# 4 types of edges in a graph after a DFS

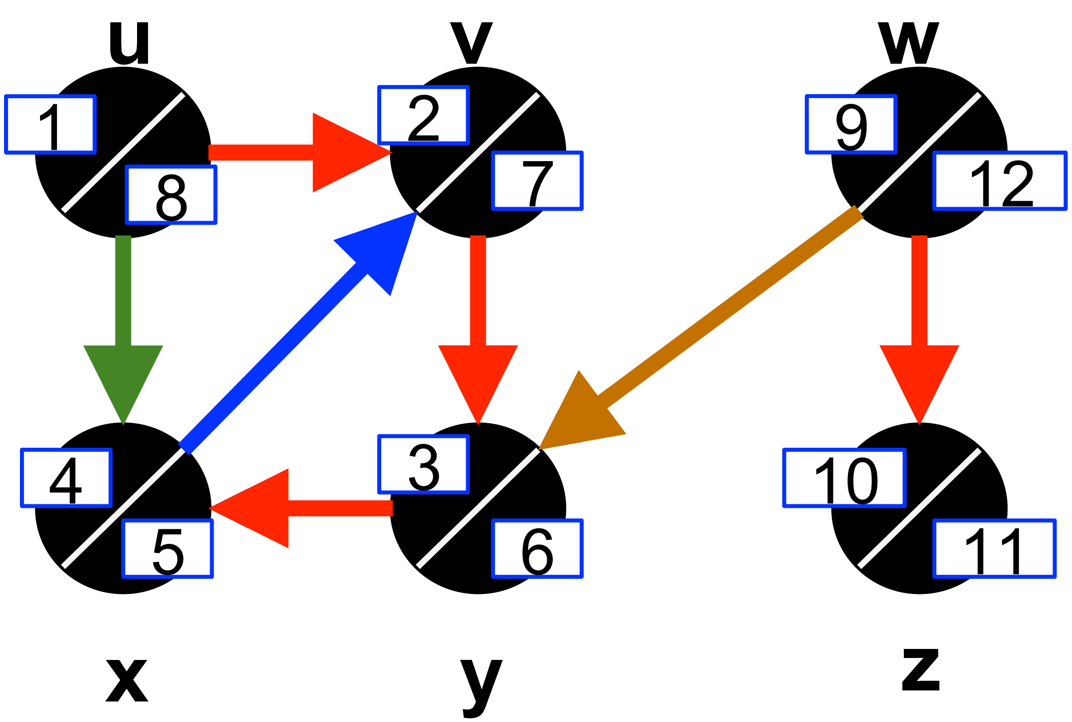
- ➔ **Tree edge:** an edge in the DFS-forest
- ➔ **Back edge:** a non-tree edge pointing from a vertex to its **ancestor** in the DFS forest.
- ➔ **Forward edge:** a non-tree edge pointing from a vertex to its **descendant** in the DFS forest

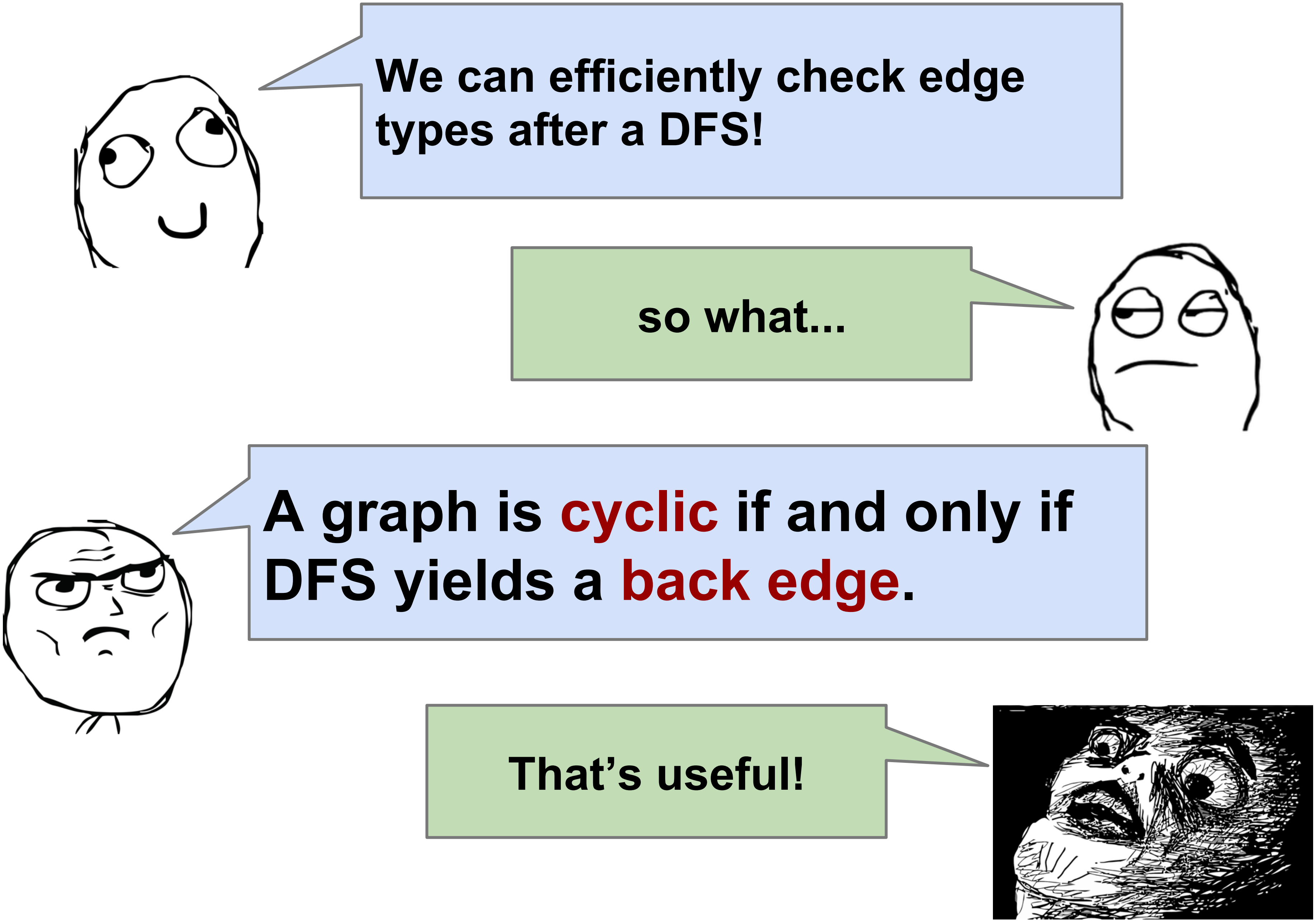



- ➔ **Cross edge**: all other edges

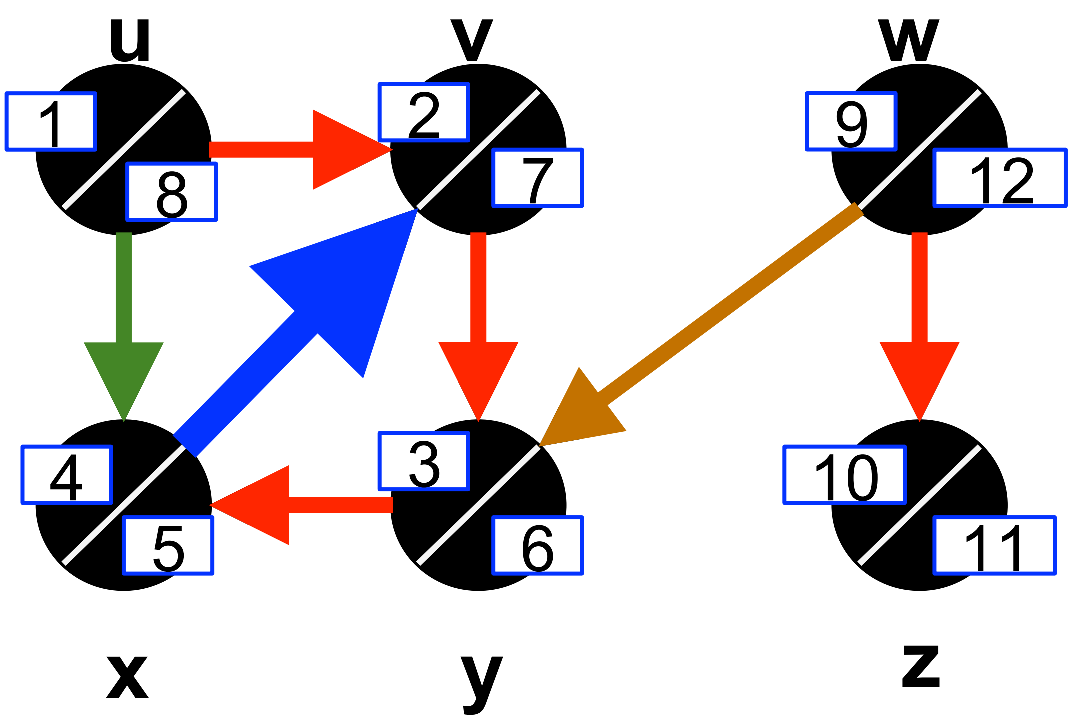
# Checking edge types

We can efficiently check edge types, because…
we can efficiently check whether a vertex is an **ancestor / descendant** of another vertex using…
the **parenthesis structure** of **[ d[v], f[v] ]** intervals!




We can efficiently check edge types after a DFS!
so what...
A graph is cyclic if and only if DFS yields a back edge.
That's useful!

**A (directed) graph contains a cycle if and only if DFS yields a back edge**



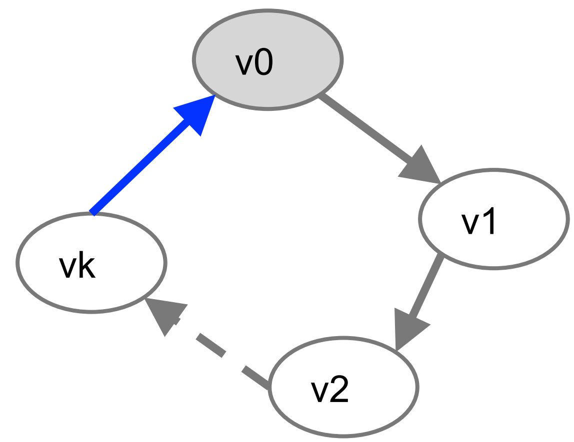
# A (directed) graph contains a cycle if and only if DFS yields a back edge

**Proof of "if":**

Let the edge be (u, v),
then by definition of back edge, v is an ancestor of u in the DFS tree,
then their is a path from v to u, i.e., v → … → u,
plus the back edge u → v,
BOOM! Cycle.

**Proof of "only if":**

Let the cycle be...,

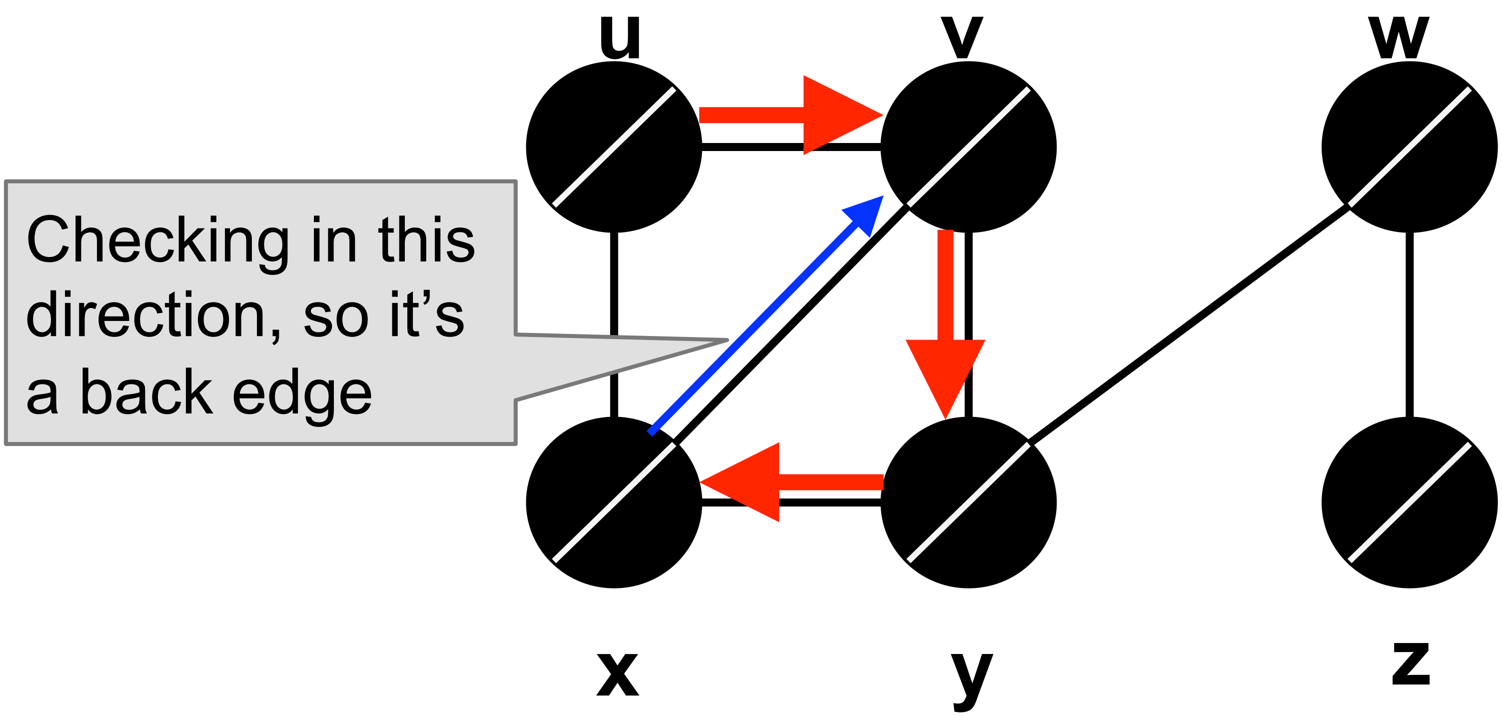



Let v0 be the first one that turns gray, when all others in the cycle are white, then vk must be a descendant of v0. (Read "White Path Theorem" in Text)

# How about undirected graph?

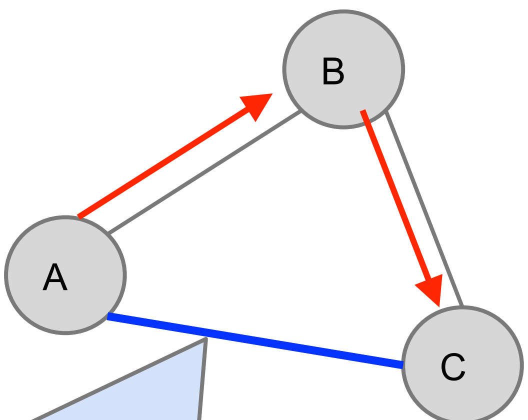
Should **back** and **forward** edges be the same thing?

➔ No, because although the edges are undirected, **neighbour checking** still has a "direction".

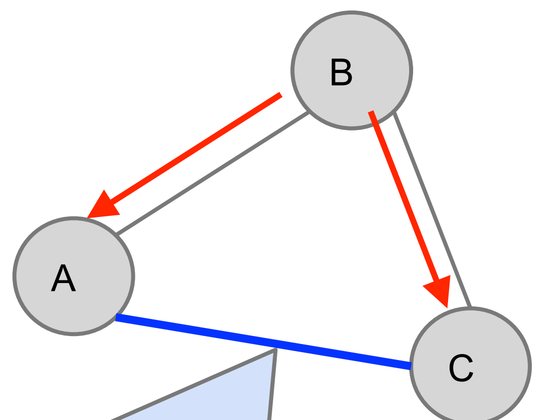


# More about undirected graph

After a DFS on a undirected graph, **every** edge is either a **tree edge** or a **back edge**, i.e., **no** forward edge or cross edge.




If this were a forward edge, it would violate the DFS algorithm (not checking at C but tracing back and check at A)

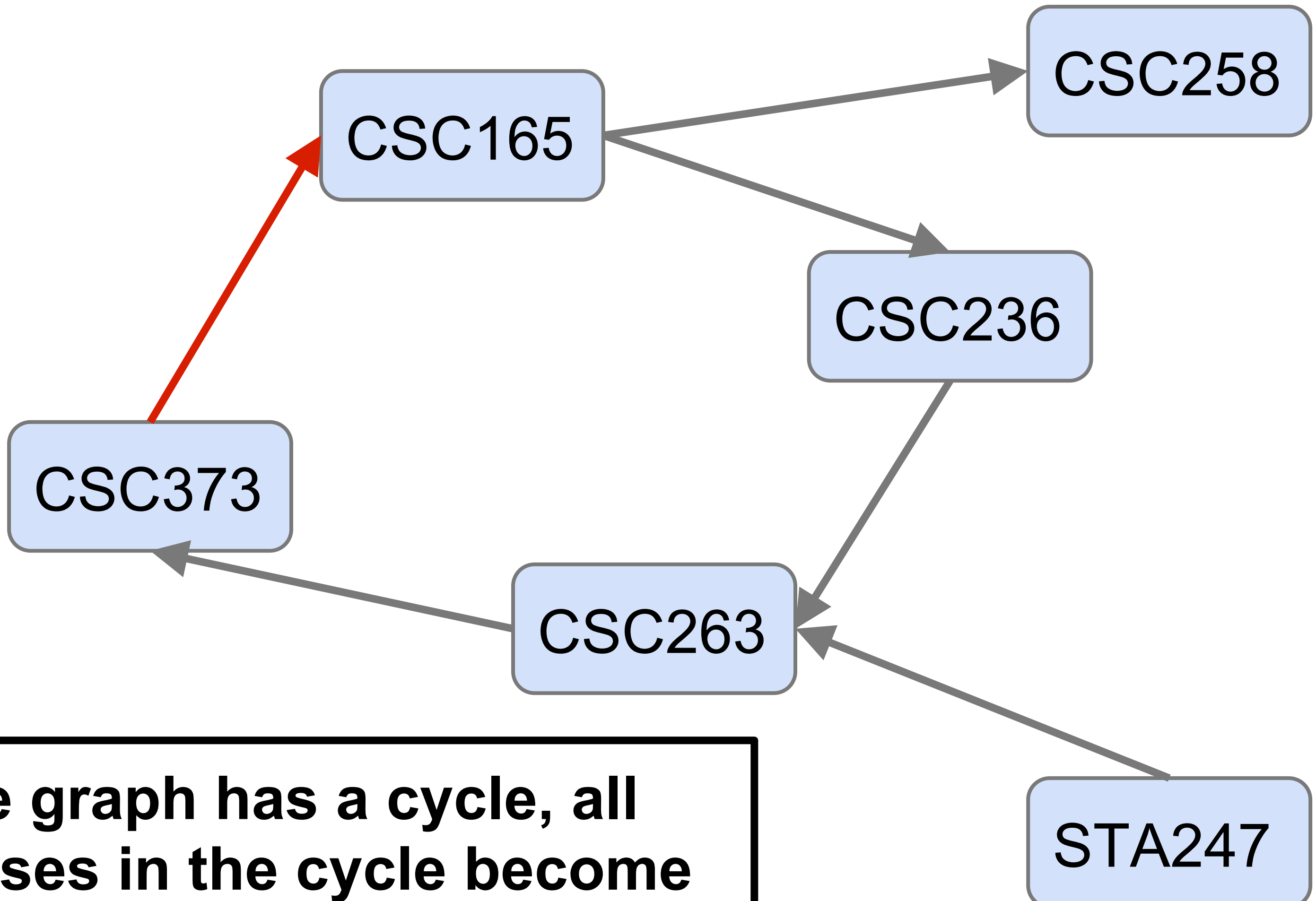



If this were a cross edge, it violets DFS again (should have checked (A, C) when reached A, but instead wait until C is visited.)

# Why do we care about cycles in a graph?

Because cycles have meaningful implication in real applications.

# Example:
# a course prerequisite graph




**If the graph has a cycle, all courses in the cycle become impossible to take!**

# Applications of DFS

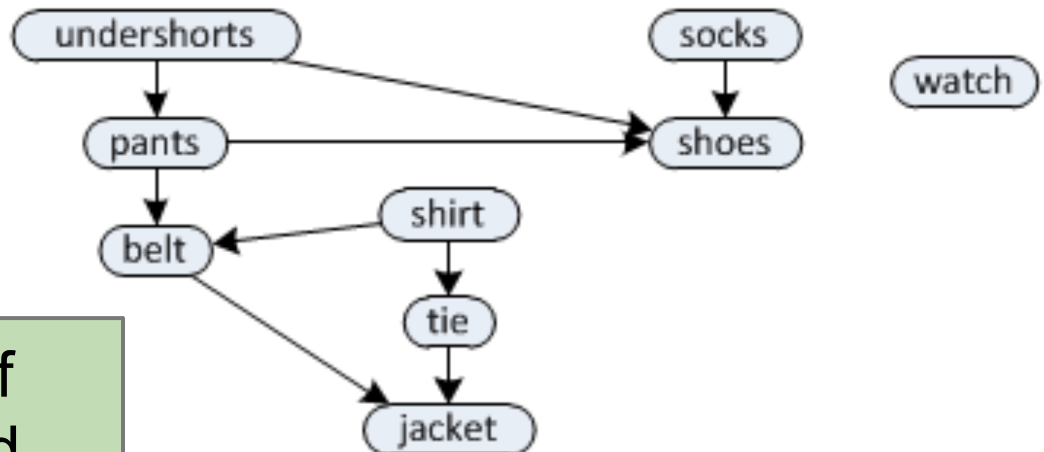
➔Detect cycles in a graph

➔Topological sort

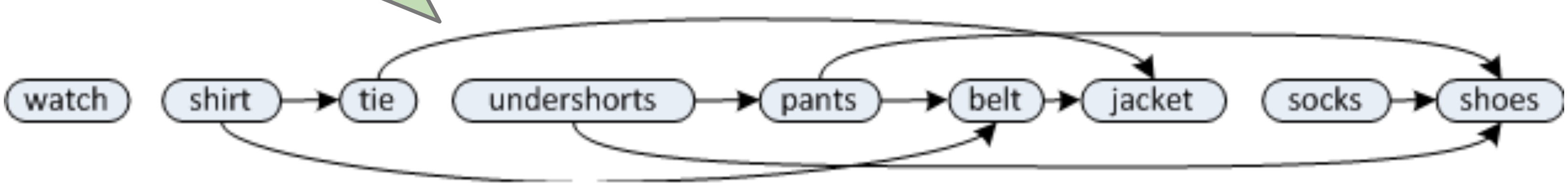
➔Strongly connected components

# Topological Sort

➔Place the vertices in such an order that all edges are pointing to the right side.




A valid order of getting dressed.



# Topological sort more formally

Suppose that in a **directed** graph **G = (V, E)** vertices **V** represent tasks, and each edge (**u**, **v**)∈**E** means that task **u** must be done before task **v**

What is an ordering of vertices 1, ..., |**V**| such that for every edge (**u**, **v**), **u** appears before **v** in the ordering?

Such an ordering is called a **topological sort of G**

Note: there can be multiple topological sorts of G

# Topological sort more formally

Is it possible to execute all the tasks in **G** in an order that respects all the precedence requirements given by the graph edges?

The answer is "**yes**" *if and only if* the directed graph **G** has **no cycle**!

(otherwise we have a **deadlock**)

Such a **G** is called a Directed Acyclic Graph, or just a **DAG**

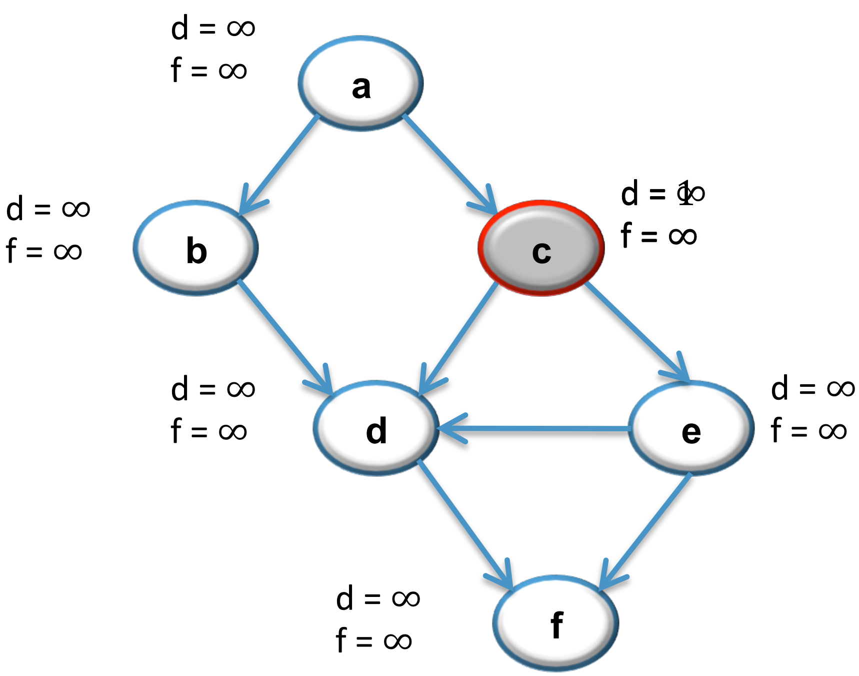
# Algorithm for TS

- TOPOLOGICAL-SORT(**G**):
  1) call DFS(G) to compute **finishing** times **f**[**v**] for each vertex **v**
  2) as each vertex is finished, insert it onto the **front** of a linked list
  3) return the linked list of vertices

- Note that the result is just a list of vertices in order of **decreasing** finish times **f**[]

# Topological sort

**Time = 2**



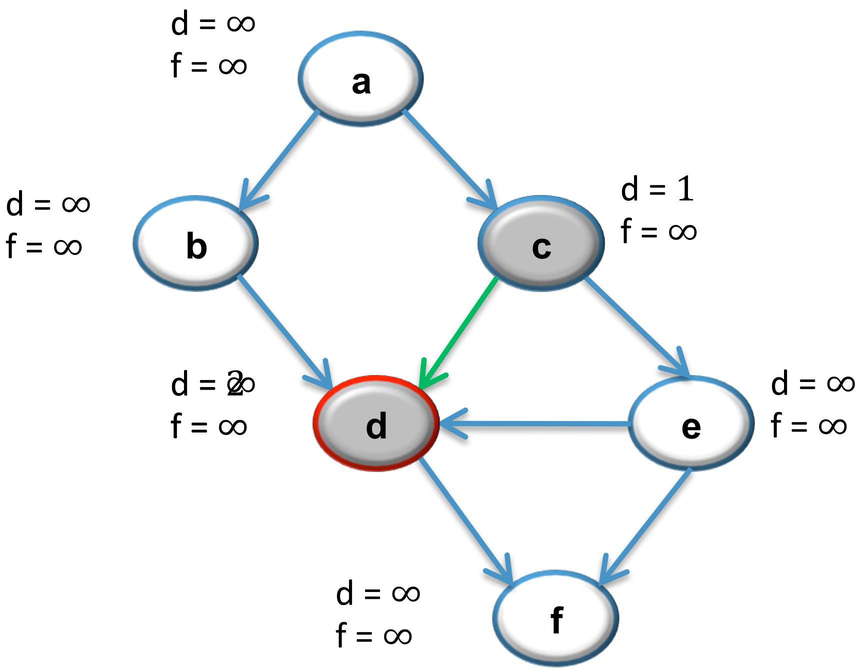

1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

# Topological sort

Time = 3

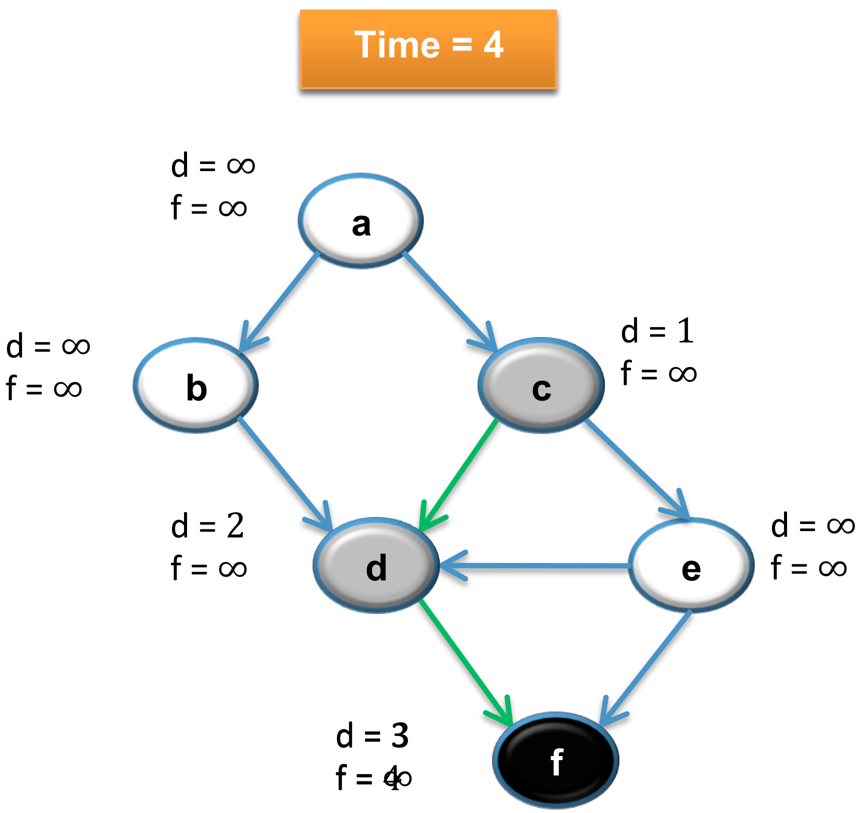



1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

# Topological sort

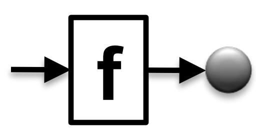



1) Call DFS(**G**) to compute the finishing times **f[v]**

2) as each vertex is finished, insert it onto the **front** of a linked list

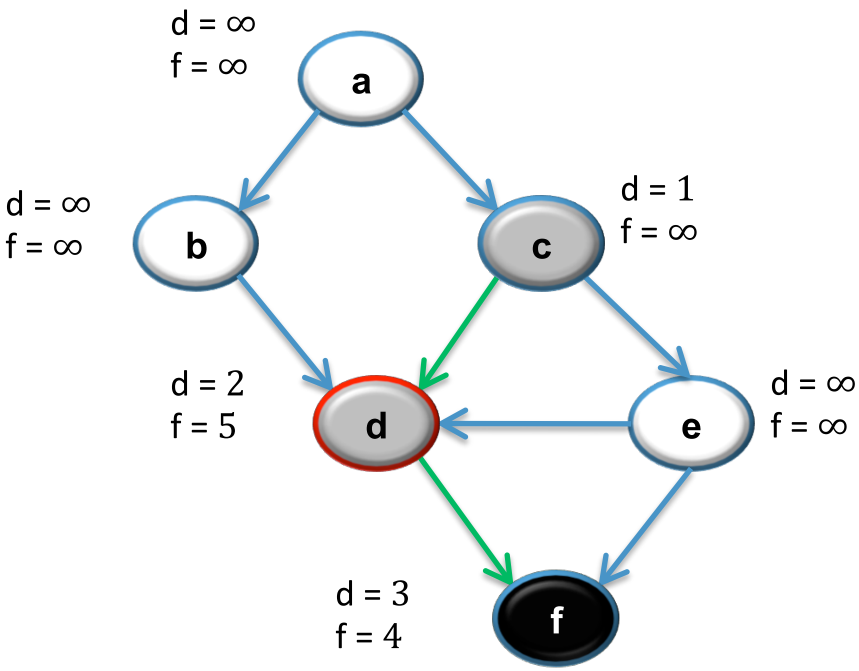
Next we discover the

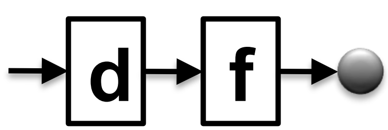
**f** is done, move back to **d**



# Topological sort

**Time = 5**




1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

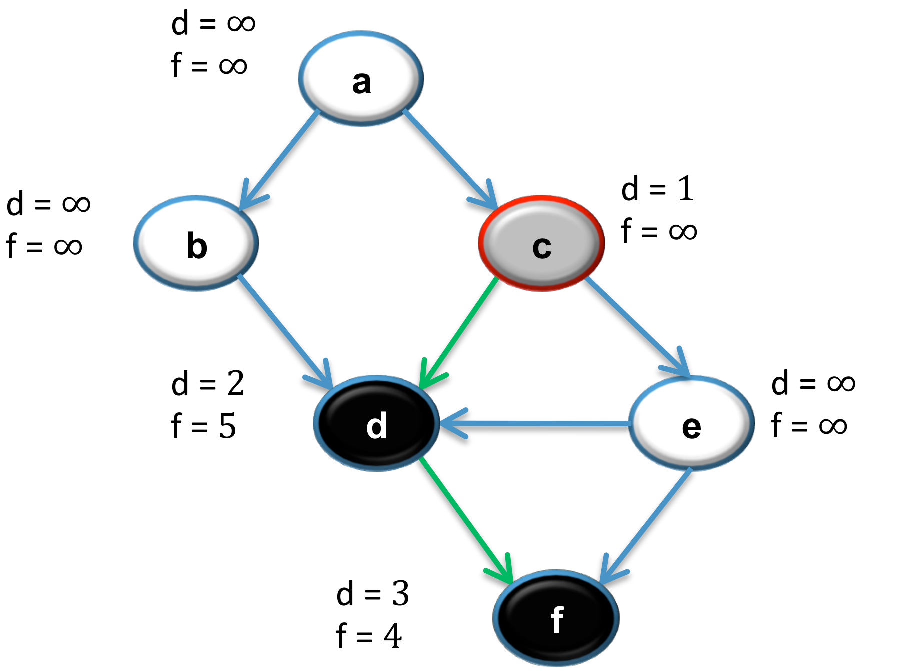
Next we discover the vertex **f**

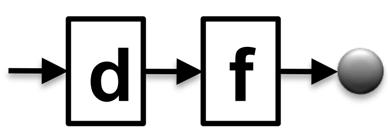
**f** is done, move back to **d**

**d** is done, move back to **c**

# Topological sort

**Time = 6**




1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

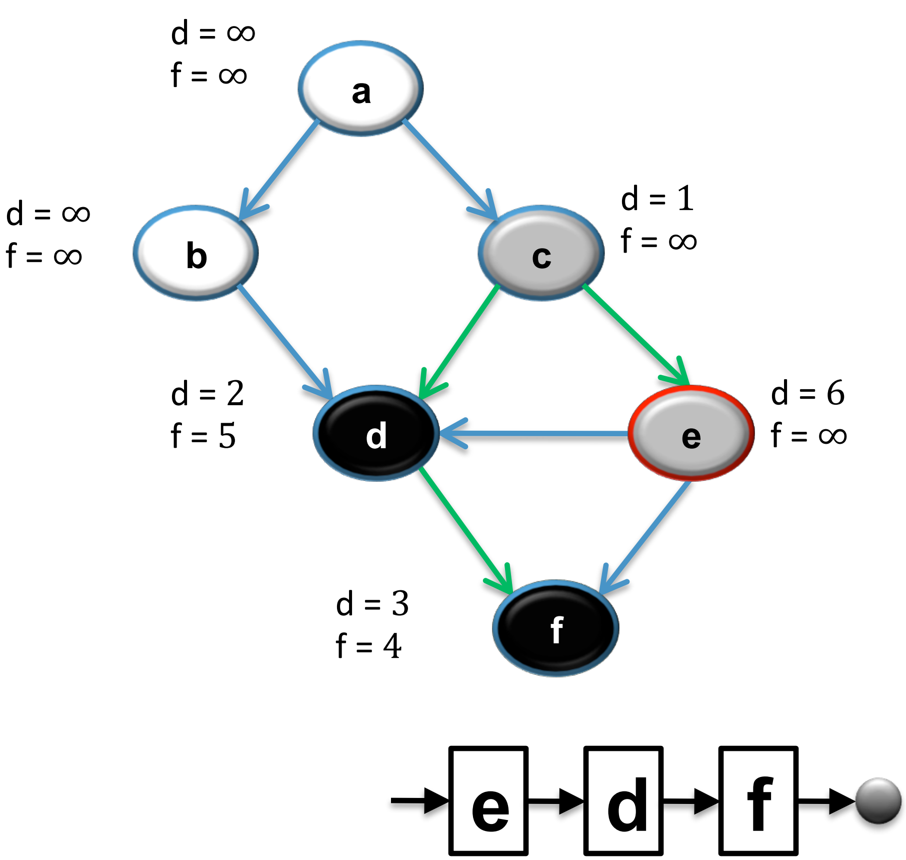
**f** is done, move back to **d**

**d** is done, move back to **c**

Next we discover the vertex **e**

# Topological sort

Time = 7




1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's say we start the DFS from the vertex **c**

Next we discover the

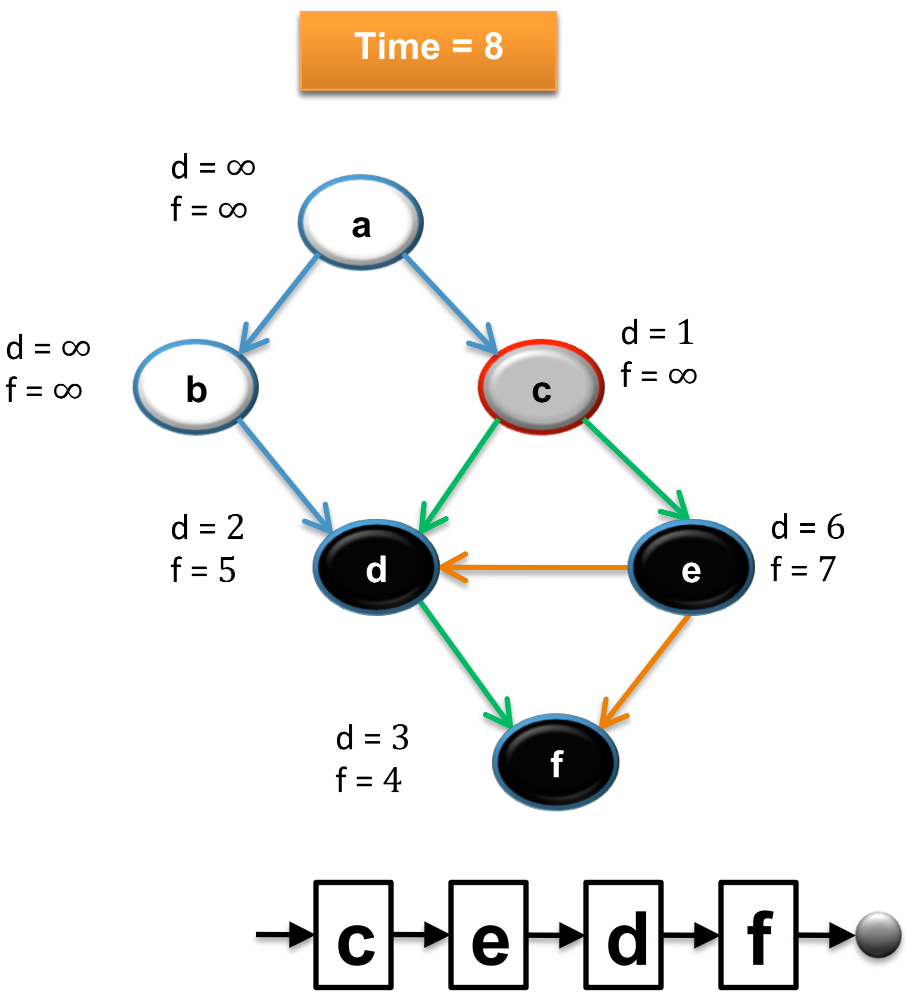
**Both edges from e are cross edges**

**d** is done, move back to **c**

Next we discover the

**e** is done, move back to **c**

# Topological sort




1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's say we start the DFS from the vertex **c**

Just a note: If there was (**c**,**f**) edge in the graph, it would be classified as a **forward edge**
(in this particular DFS run)

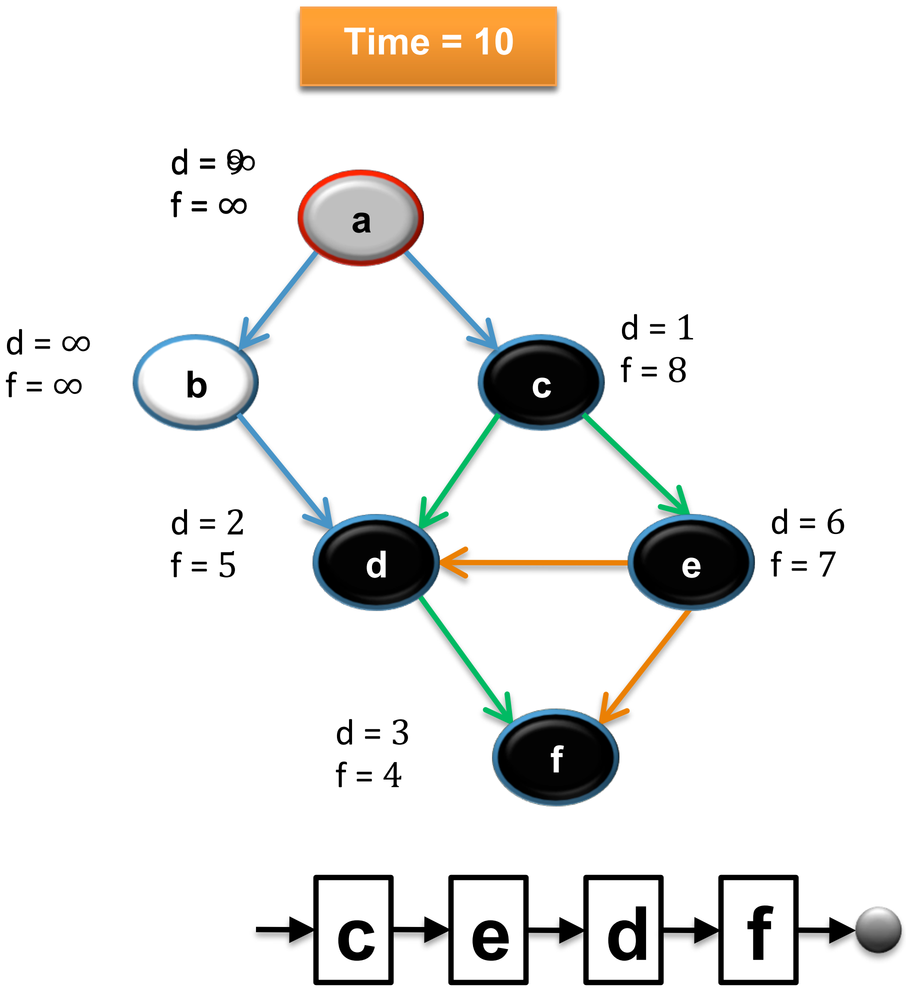
**d** is done, move back to **c**

Next we discover the vertex **e**

**e** is done, move back to **c**

**c** is done as well

# Topological sort




1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's now call DFS visit from the vertex **a**

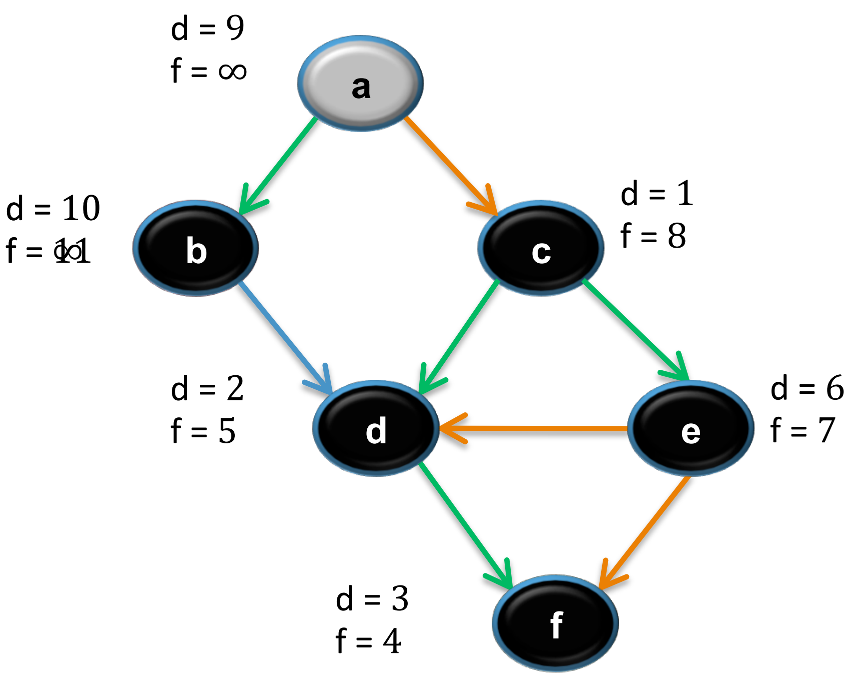
Next we discover the vertex **c**, but **c** was already

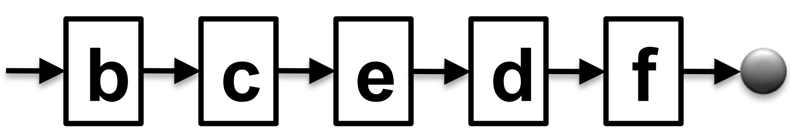
Next we discover the vertex **b**

# Topological sort

1) Call DFS(**G**) to compute the finishing times **f[v]**

Time = 11




Let’s now call DFS visit from the vertex **a**

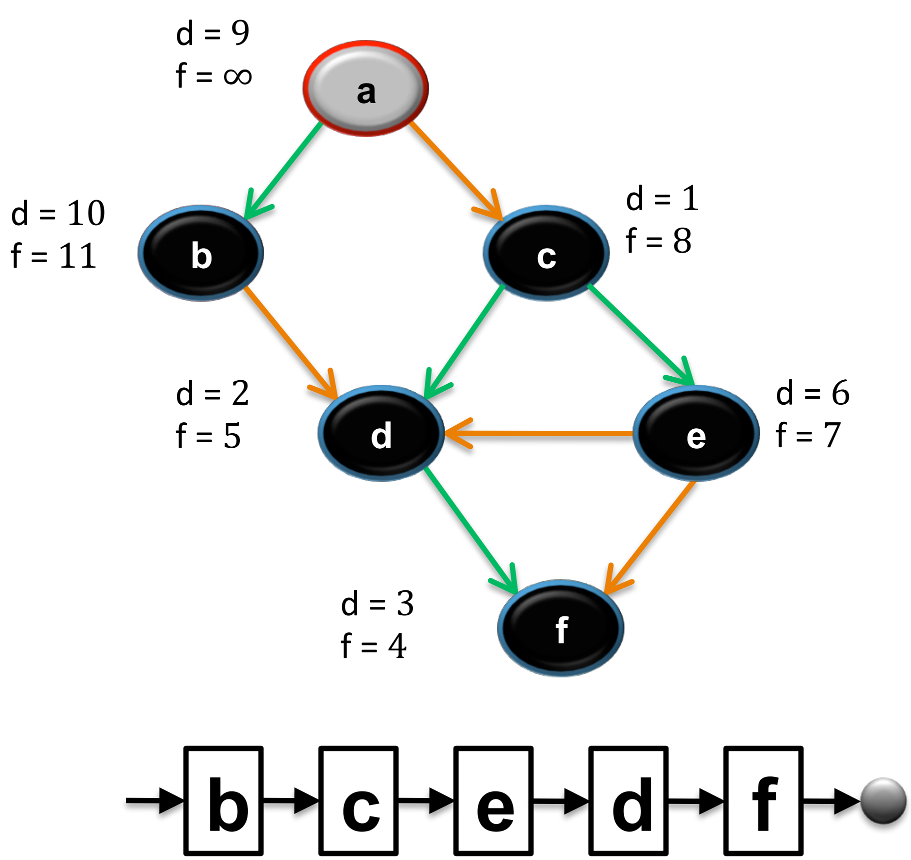
Next we discover the vertex **c**,
but **c** was already

Next we discover the

**b** is done as (**b**,**d**) is a cross edge => now move back to **c**

# Topological sort

Time = 12




1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's now call DFS visit from the vertex **a**

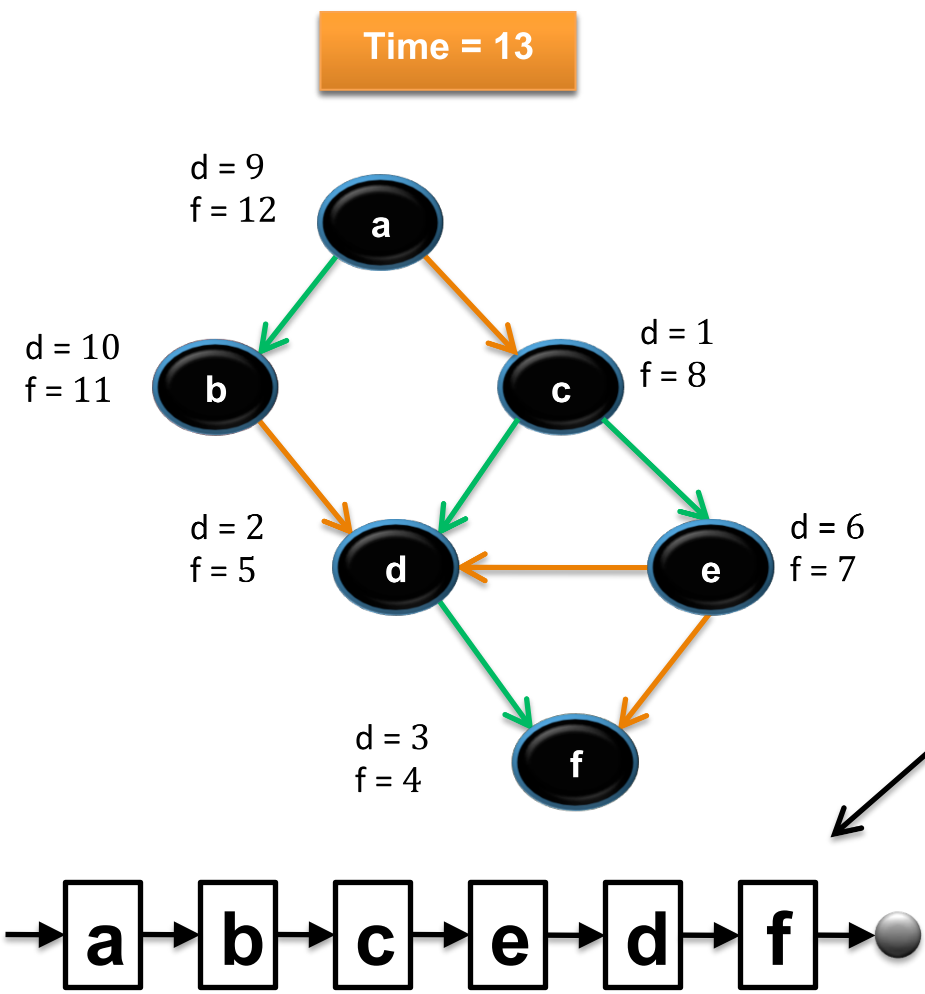
Next we discover the vertex **c**, but **c** was already

Next we discover the

**b** is done as (**b**,**d**) is a cross edge => now move

**a** is done as well

# Topological sort




1) Call DFS(**G**) to compute the finishing times **f[v]**

**WE HAVE THE RESULT!**

3) return the linked list of vertices

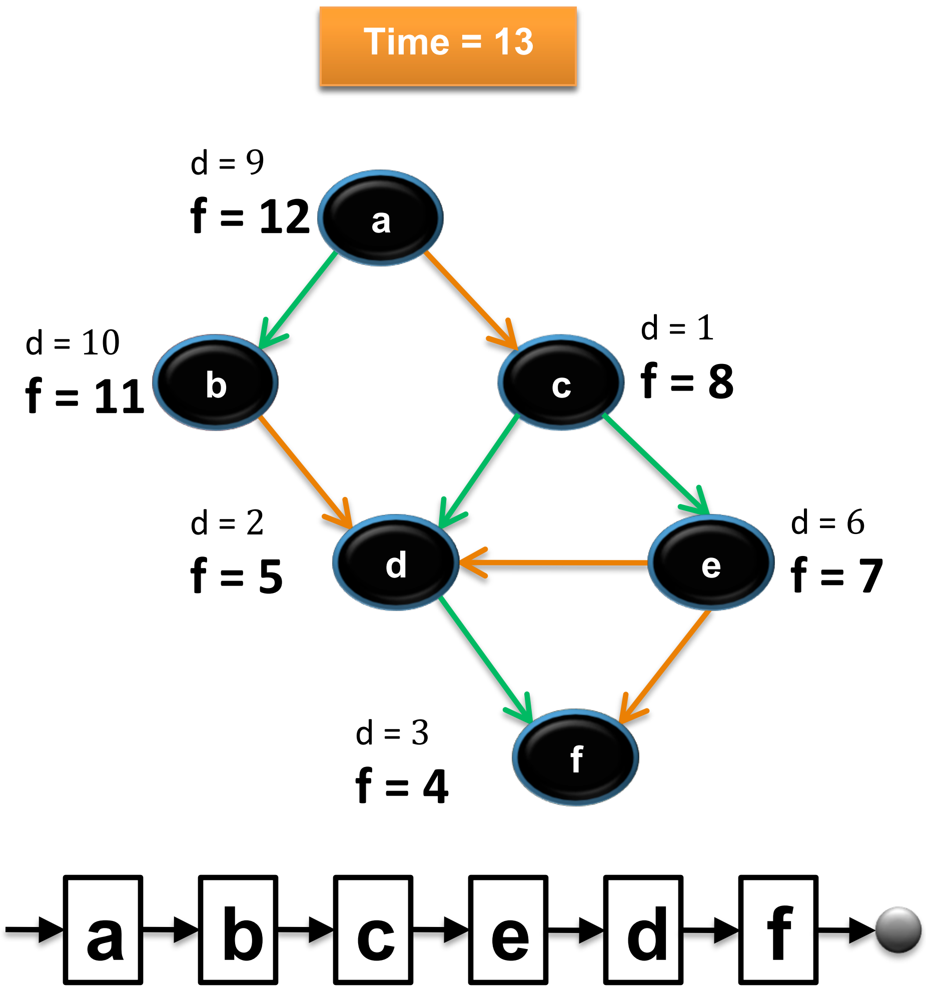
but **c** was already

Next we discover the

**b** is done as (**b**,**d**) is a cross edge => now move

**a** is done as well

# Topological sort




The linked list is sorted in **decreasing** order of finishing times **f[]**

Try yourself with different vertex order for DFS visit

Note: If you redraw the graph so that all vertices are in a line ordered by a valid topological sort, then all edges point **„from left to right"**

# Time complexity of TS(G)

Running time of topological sort:

$$\mathbf{\Theta(n + m)}$$

where **n**=|**V**| and **m**=|**E**|

Why? Depth first search takes **Θ(n + m)** time in the worst case, and inserting into the front of a linked list takes **Θ(1)** time

## Proof of correctness

- **Theorem**: TOPOLOGICAL-SORT(**G**) produces a topological sort of a DAG **G**

- The TOPOLOGICAL-SORT(**G**) algorithm does a DFS on the DAG **G**, and it lists the nodes of **G** in order of decreasing finish times **f**[]
- We must show that this list satisfies the topological sort property, namely, that for every edge (**u**,**v**) of **G**, **u** appears before **v** in the list
- **Claim**: For every edge (**u**,**v**) of **G**: **f**[**v**] < **f**[**u**] in DFS

# Proof of correctness

"For every edge (**u**,**v**) of **G**, **f**[**v**] < **f**[**u**] in this DFS"

The DFS classifies (**u**,**v**) as a **tree edge**, a **forward edge** or a **cross-edge** (it cannot be a back-edge since **G** has no cycles):

i. If (**u**,**v**) is a **tree** or a **forward edge** ⇒ **v** is a descendant of **u** ⇒ **f**[**v**] < **f**[**u**]
ii. If (**u**,**v**) is a **cross-edge**

# Proof of correctness

"For every edge (**u**,**v**) of **G**: $\mathbf{f}[\mathbf{v}] < \mathbf{f}[\mathbf{u}]$ in this DFS"

Q.E.D. of Claim

ii. If (**u**,**v**) is a **cross-edge**:

as (**u**,**v**) is a cross-edge, by definition, neither **u** is a descendant of **v** nor **v** is a descendant of **u:**

$d[\mathbf{u}] < f[\mathbf{u}] < d[\mathbf{v}] < f[\mathbf{v}]$

**or**

$d[\mathbf{v}] < f[\mathbf{v}] < d[\mathbf{u}] < f[\mathbf{u}]$

since (**u**,**v**) is an edge, **v** is surely discovered **before u**'s exploration completes

$\mathbf{f[v] < f[u]}$

# Proof of correctness

TOPOLOGICAL-SORT(G) lists the nodes of G from highest to lowest finishing times

By the **Claim**, for every edge (**u**,**v**) of **G**:

$$\mathbf{f}[\mathbf{v}] < \mathbf{f}[\mathbf{u}]$$

⇒ **u** will be before **v** in the algorithm's list

Q.E.D of **Theorem**

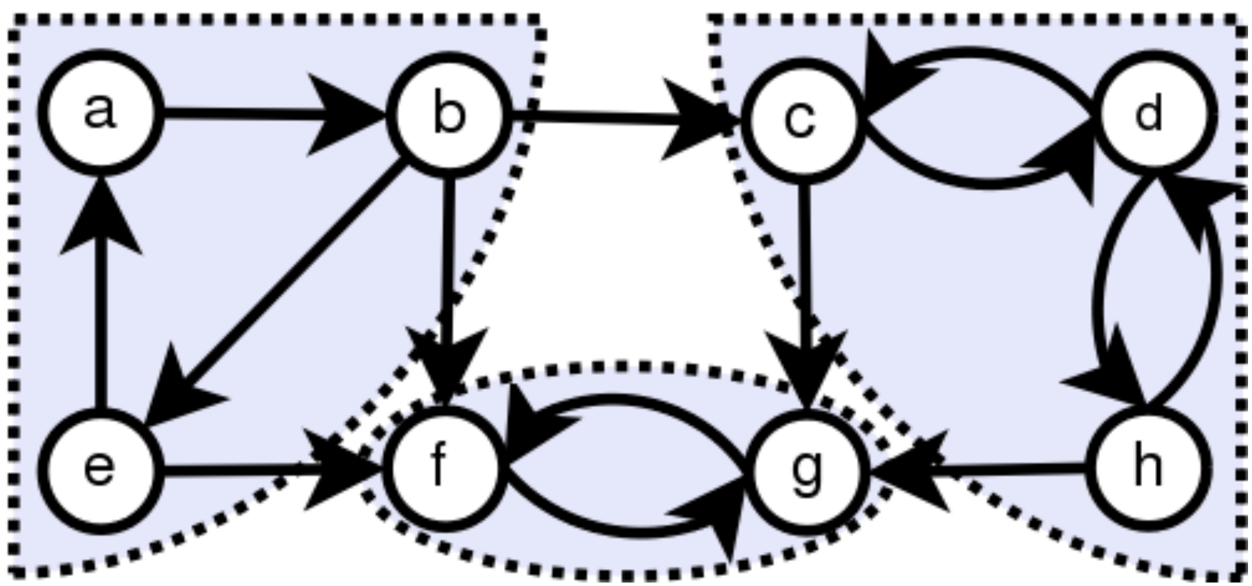
# Recap: topological sorting

1. Do a **DFS**

2. Order vertices according to their **finishing times f[v]**

# Strongly connected components
(covered in this week's tutorial)

➔Subgraphs with strong connectivity (any pair of vertices can reach each other)



# Summary of DFS

- ➔It's the twin of BFS (Queue vs Stack)
- ➔Keeps two timestamps: d[v] and f[v]
- ➔Has same runtime as BFS
- ➔Does NOT give us shortest-path
- ➔Give us cycle detection (back edge)
- ➔For real problems, choose BFS and DFS wisely.

# Next week

➔Minimum Spanning Tree