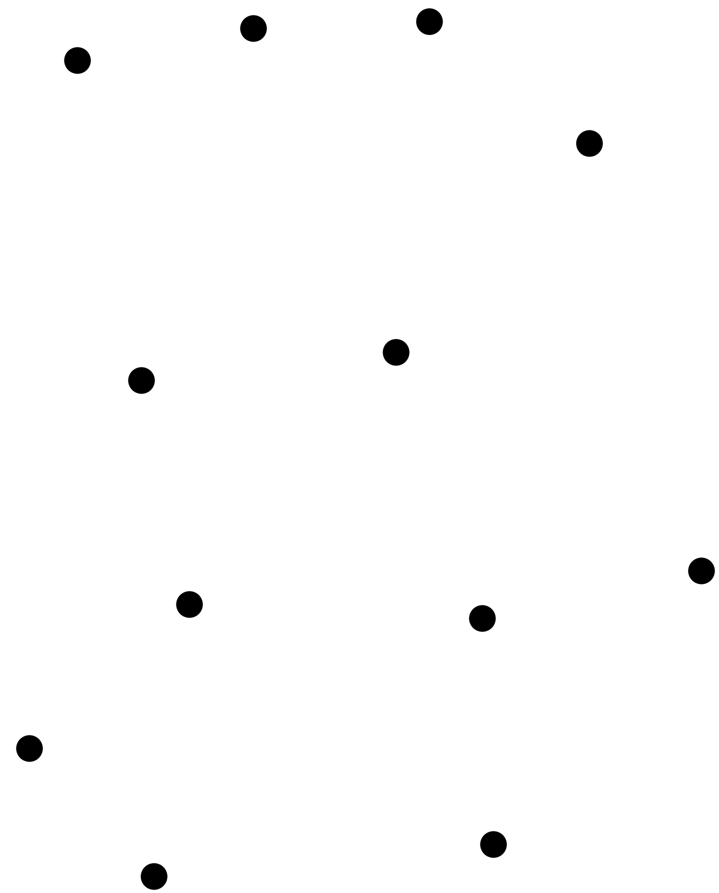
Lecture 17
Dynamic set partitions and the union-find algorithm

EECS-214

Set partitions

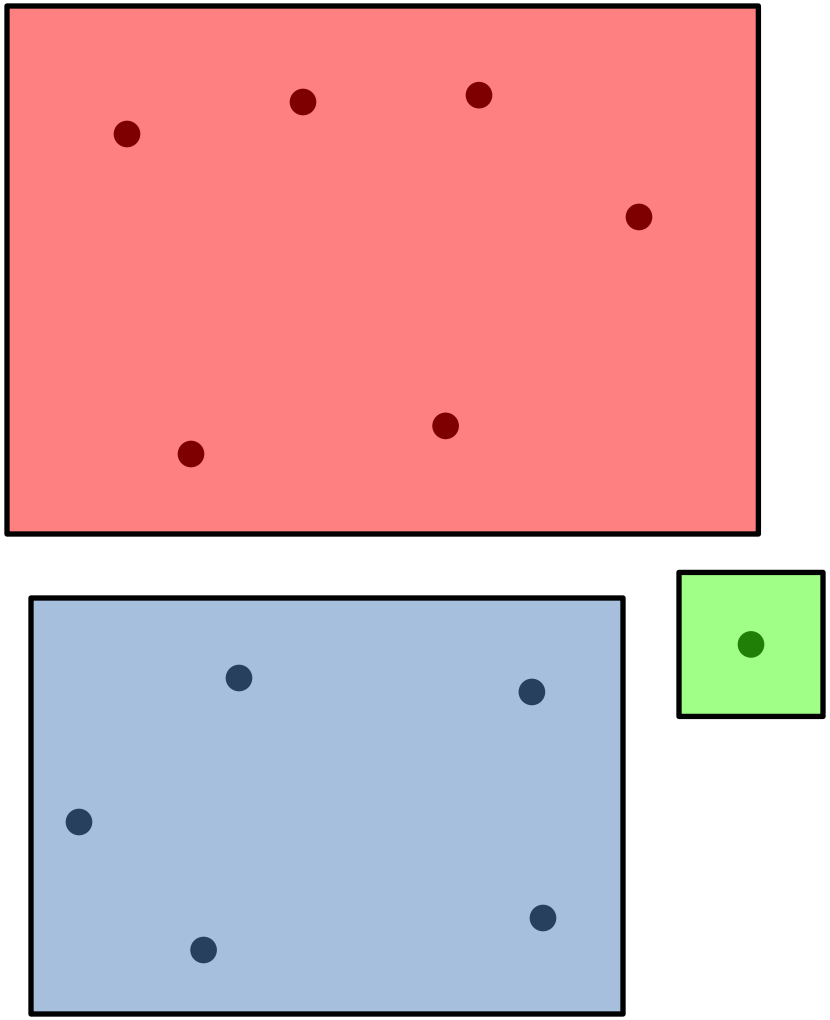
- **Division** of elements of a set into **disjoint groups**
- Formally, given a set S, a partition $P = \{ P_1, \dots, P_n \}$ is
  - A set of non-empty subsets of S
    - i.e. for all i, $\{\} \neq P_i \subseteq S$
  - Such that
    - Every element of S is a member of **exactly one element** of P
    - Or equivalently, the P_i are
      - **Disjoint**
        $P_i \cap P_j = \{\}$ for all $i \neq j$
      - **Cover** S
        $$S = \bigcup_{P_i \in P} P_i$$



Set partitions

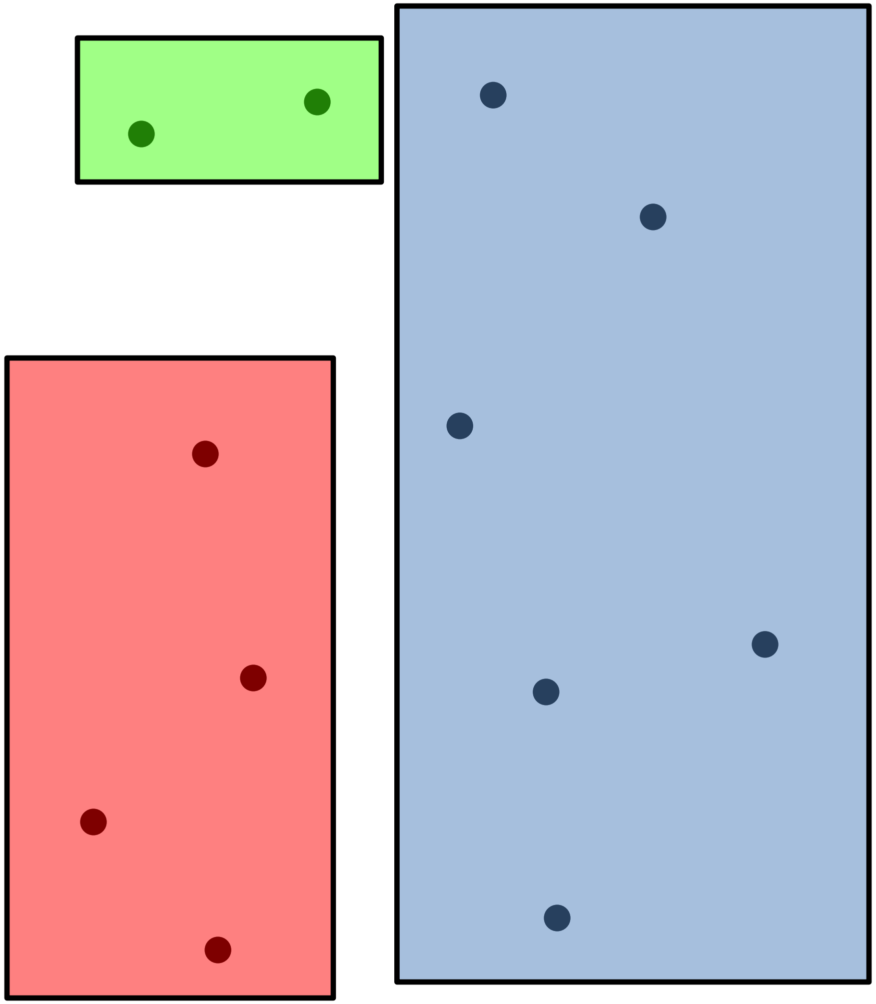
- **Division** of elements of a set into **disjoint groups**
- Formally, given a set S, a partition $P = \{ P_1, \ldots, P_n \}$ is
  - A set of non-empty subsets of S
    - i.e. for all i, $\{\,\} \neq P_i \subseteq S$
  - Such that
    - Every element of S is a member of **exactly one element** of P
    - Or equivalently, the P_i are
      - **Disjoint**
        $P_i \cap P_j = \{\,\}$ for all $i \neq j$
      - **Cover** S

$$S = \bigcup_{P_i \in P} P_i$$



Set partitions

- **Division** of elements of a set into **disjoint groups**
- Formally, given a set S, a partition $P = \{ P_1, \ldots, P_n \}$ is
  - A set of non-empty subsets of S
    - i.e. for all i, $\{\} \neq P_i \subseteq S$
  - Such that
    - Every element of S is a member of **exactly one element** of P
    - Or equivalently, the P_i are
      - **Disjoint**
        $P_i \cap P_j = \{\}$ for all $i \neq j$
      - **Cover** S
        $$S = \bigcup_{P_i \in P} P_i$$



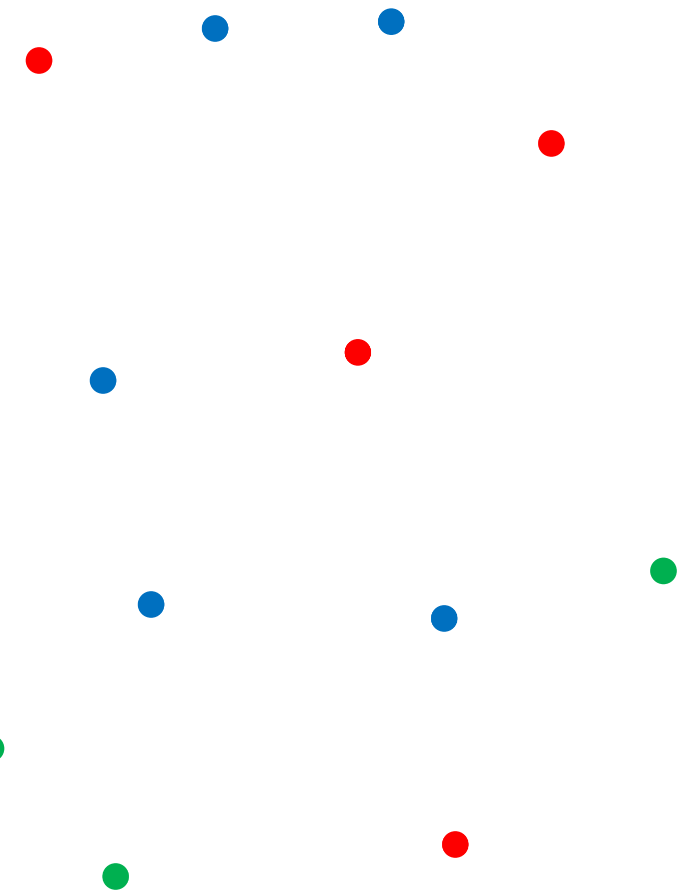
Set partitions

- **Division** of elements of a set into **disjoint groups**

- Formally, given a set S, a partition $P = \{ P_1, \dots, P_n \}$ is
  - A set of non-empty subsets of S
    - i.e. for all i, $\{\} \neq P_i \subseteq S$
  - Such that
    - Every element of S is a member of **exactly one element** of P
    - Or equivalently, the P_i are
      - **Disjoint**

        $P_i \cap P_j = \{\}$ for all $i \neq j$
      - **Cover** S

        $$S = \bigcup_{P_i \in P} P_i$$



Partitions and equivalence relations

Remember equivalence relations?

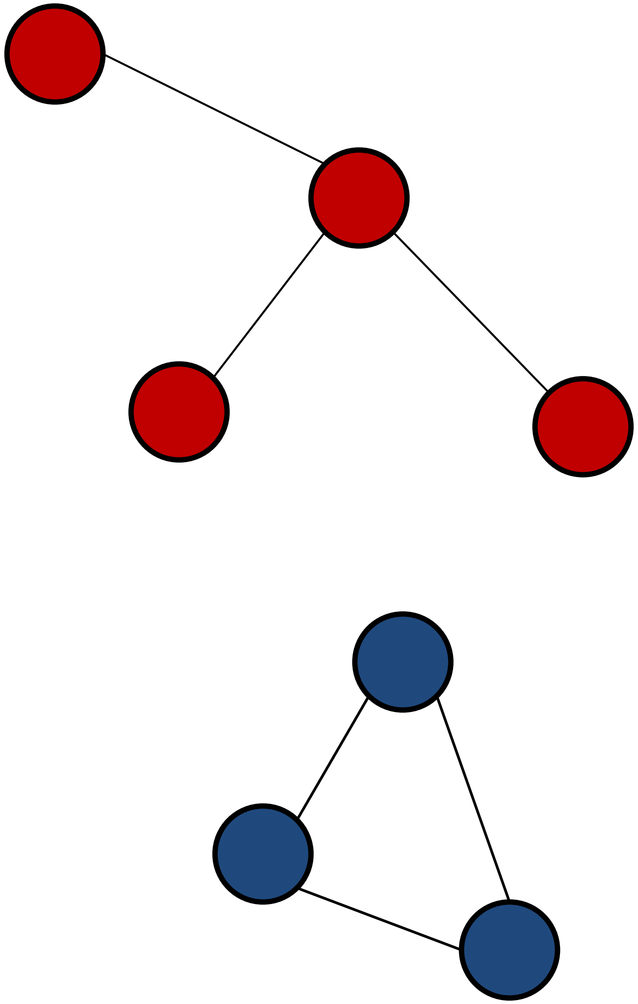
- Relation (call it $\equiv$)
  - Over some set S
- That has the **properties** of
  - **Reflexivity**
    - $x \equiv x$
  - **Transitivity**
    - If $x \equiv y$ and $y \equiv z$
    - Then $x \equiv z$
  - **Symmetry**
    - If $x \equiv y$, then $y \equiv x$
- Divide elements into **equivalence classes**
  - Two elements, $x, y \in S$, are in the same equivalence class iff $x \equiv y$
  - Equivalence class of x is $[x] = \{\, y \mid x \equiv y \,\}$

Partitions and equivalence relations are **interchangeable** ideas

- **Equivalence classes form a partition** of S
- **Given a partition**, we can **construct an equivalence relation** from it
  - $a \equiv b$ iff a and b are in the same set within the partition

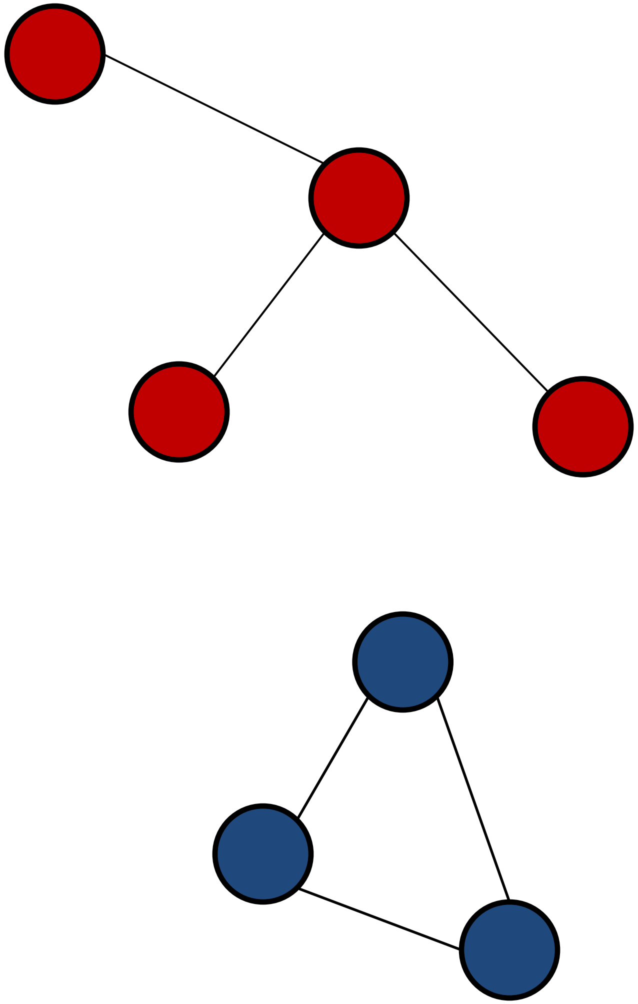
Connected components

- **Connectivity** in an undirected graph is an **equivalence relation**
  - $a \equiv b$ iff there is a **path** between a and b
- Its **equivalence classes** are essentially its **connected components**
- They form a **partition** of the nodes
  - Every node belongs to **exactly one connected component**



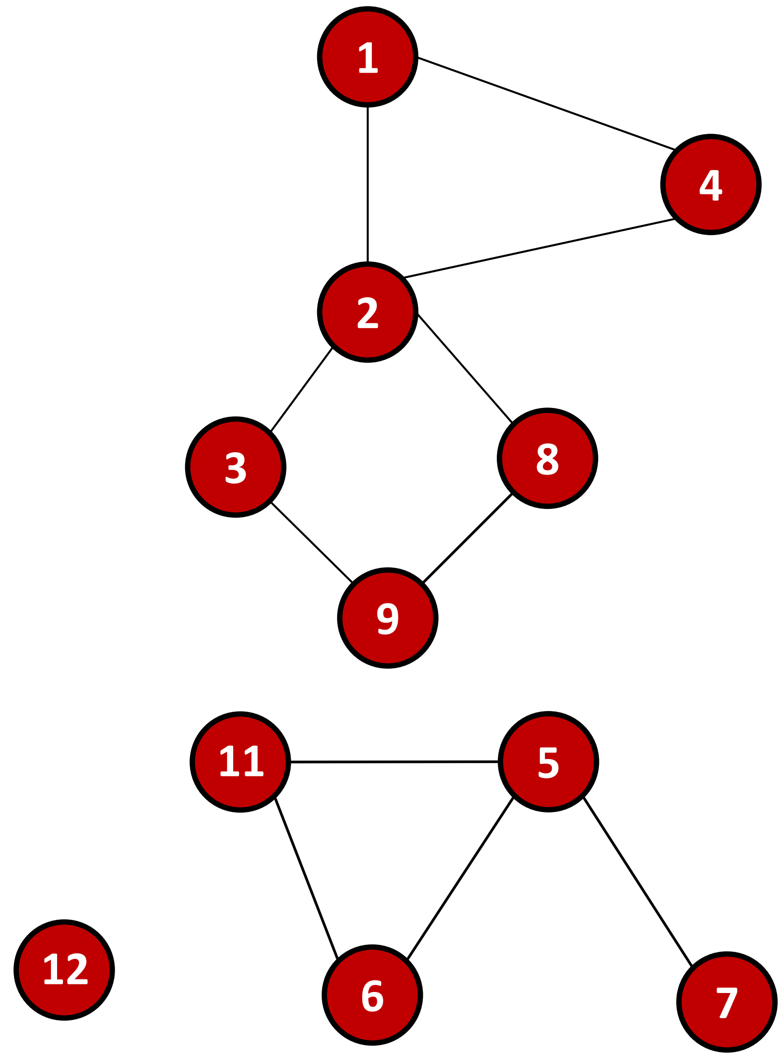
Connected components

- How do we **compute** the partition?



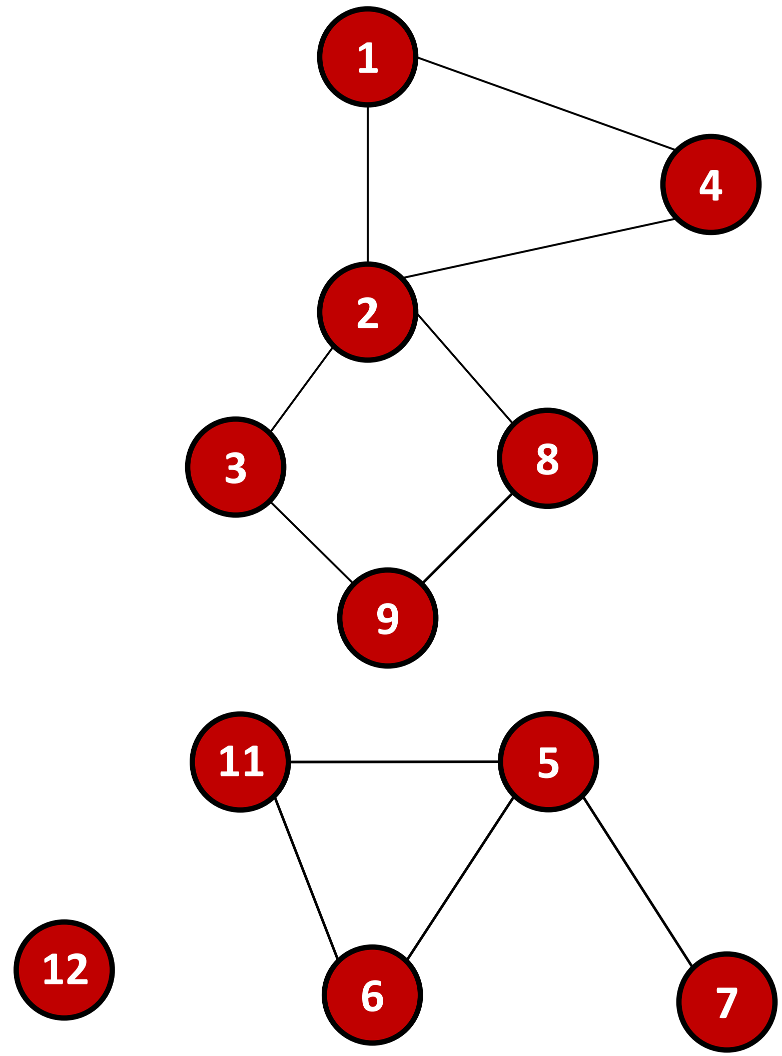
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
     if node not visited
        LCCVisit(node, component)
        component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
           neighbor, n, of node
    LCCVisit(n, c)
```



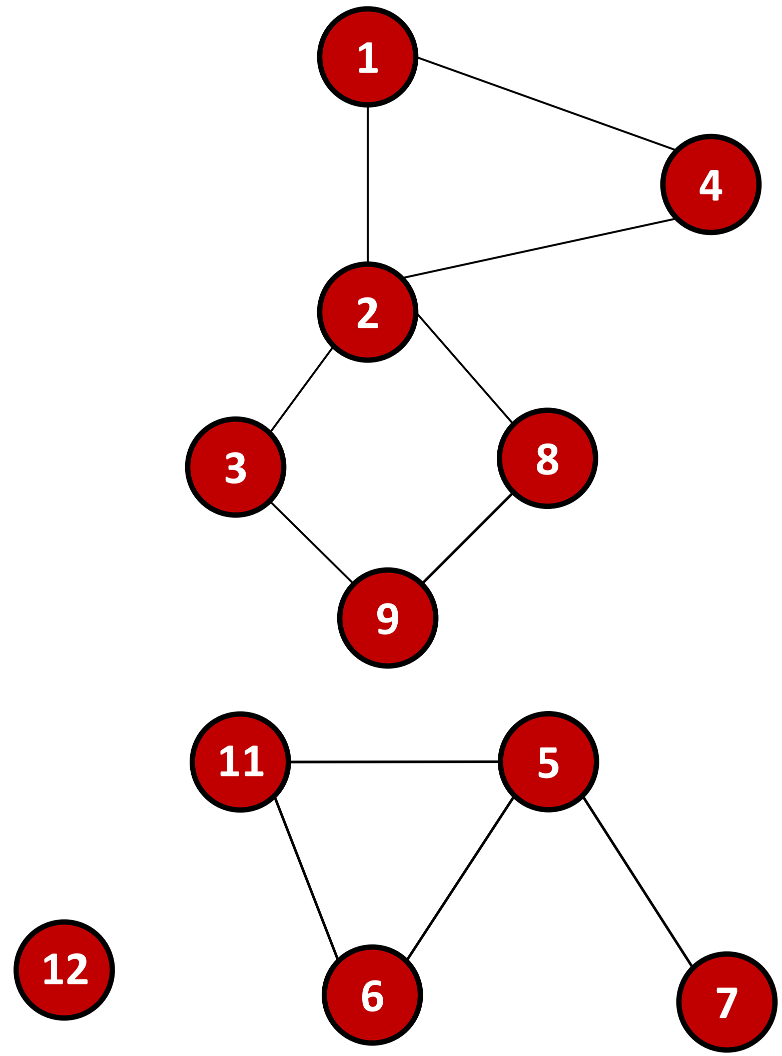
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
     if node not visited
        LCCVisit(node, component)
        component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
           neighbor, n, of node
    LCCVisit(n, c)
```



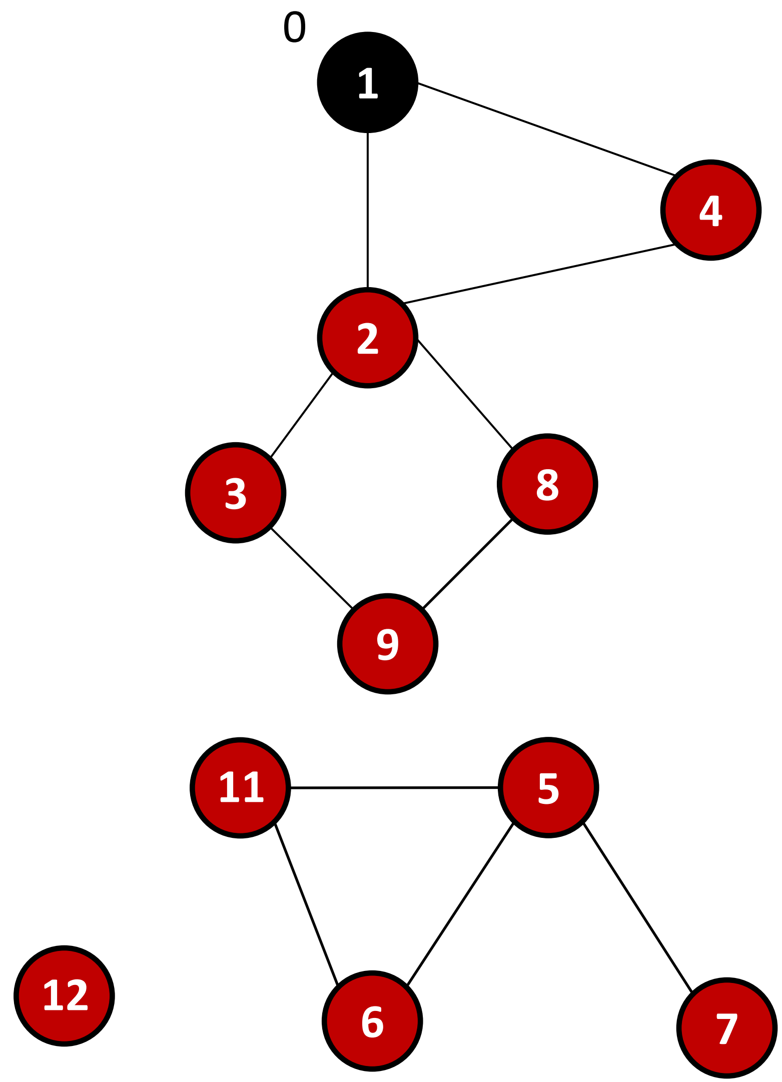
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
          LCCVisit(node, component)
          component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
              neighbor, n, of node
     LCCVisit(n, c)
```



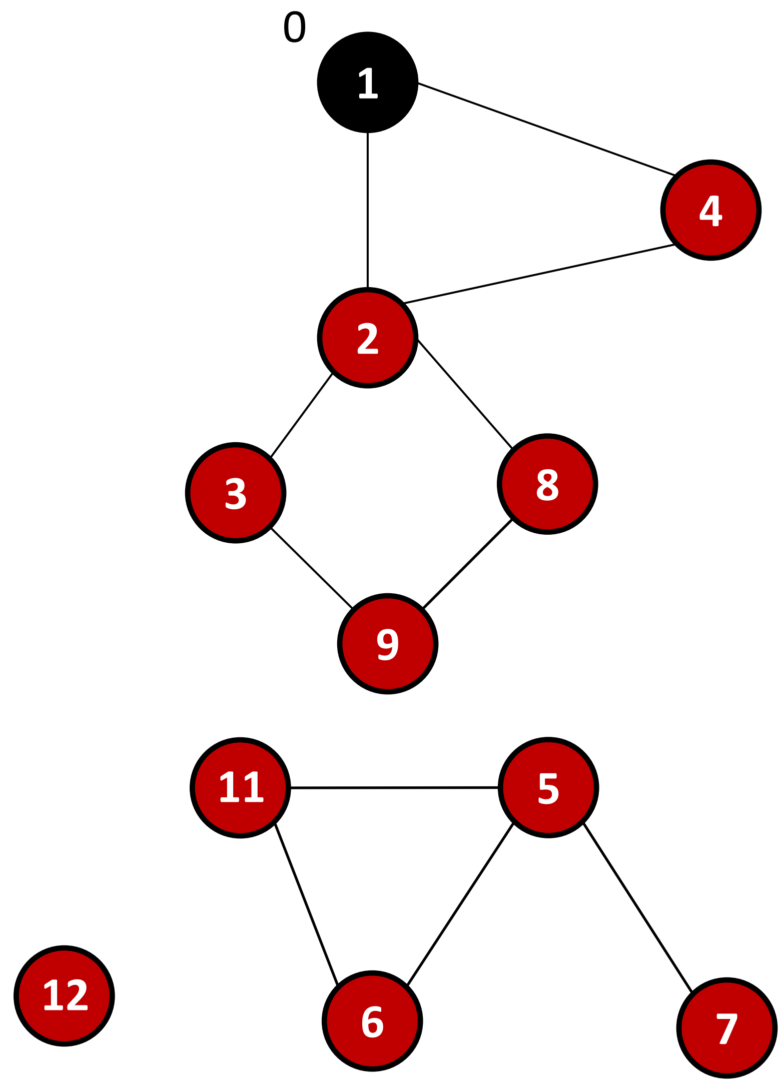
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
             neighbor, n, of node
    LCCVisit(n, c)
```



Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
          LCCVisit(node, component)
          component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
              neighbor, n, of node
     LCCVisit(n, c)
```



Connected component analysis using depth-first search

LabelConnectedComponents()

  component = 0

  for each node in graph

    if node not visited

      LCCVisit(node, component)

      component++

LCCVisit(node, c)

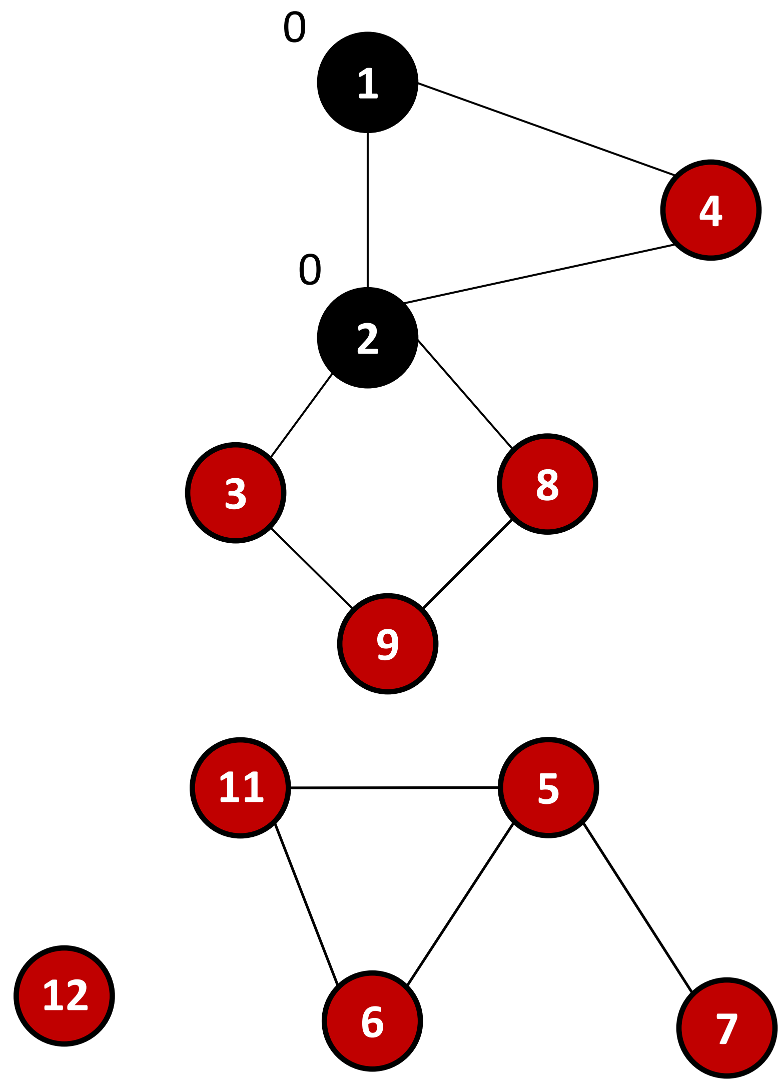
  mark node visited

  node.component = c

  for each unvisited

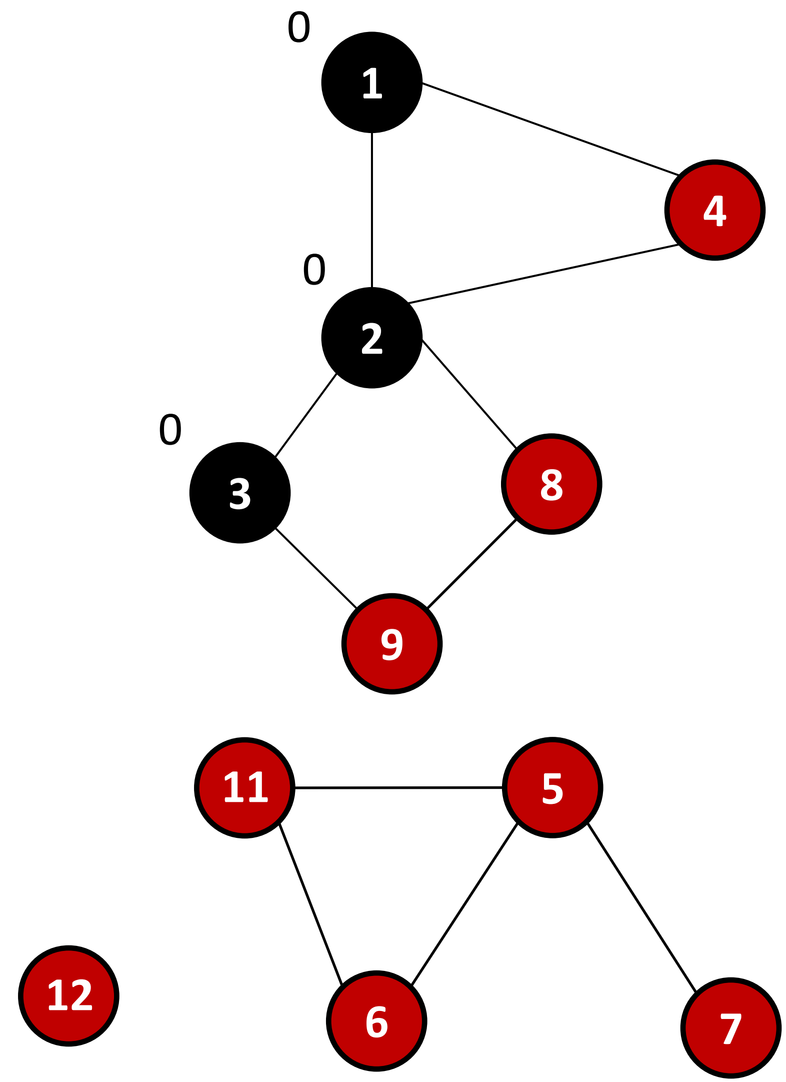
      neighbor, n, of node

    LCCVisit(n, c)



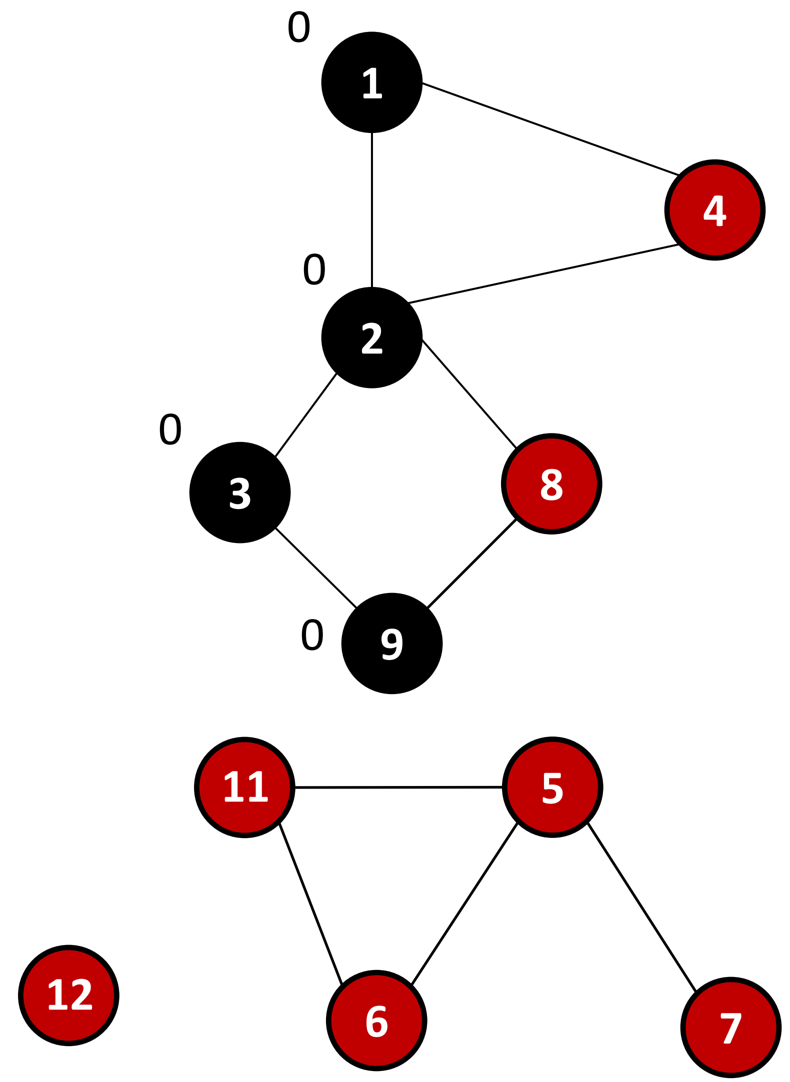
Connected component analysis using depth-first search

```
LabelConnectedComponents()
   component = 0
   for each node in graph
       if node not visited
           LCCVisit(node, component)
           component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
            neighbor, n, of node
     LCCVisit(n, c)
```



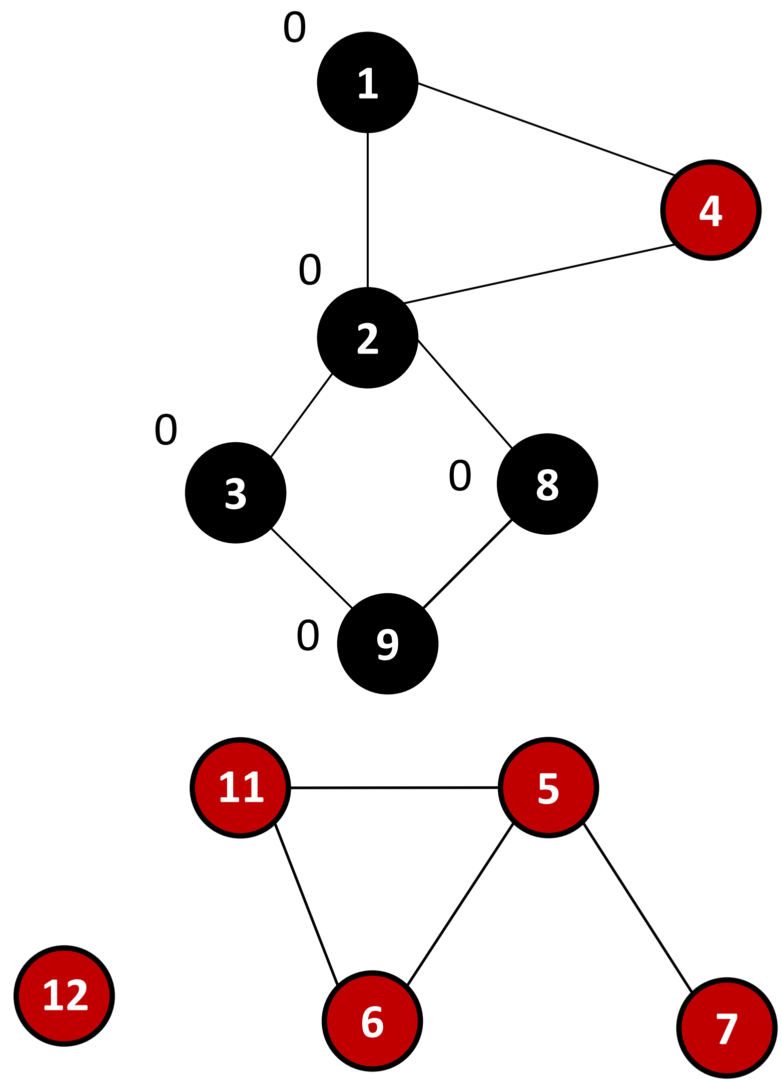
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
           neighbor, n, of node
    LCCVisit(n, c)
```



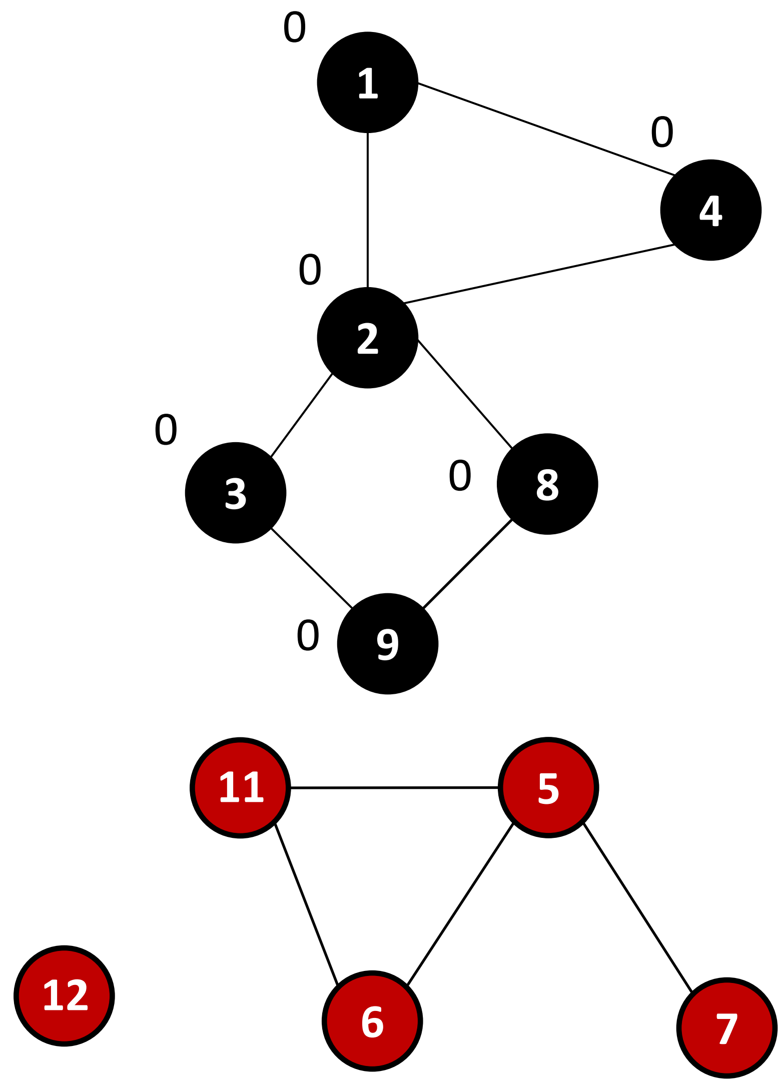
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
           neighbor, n, of node
    LCCVisit(n, c)
```



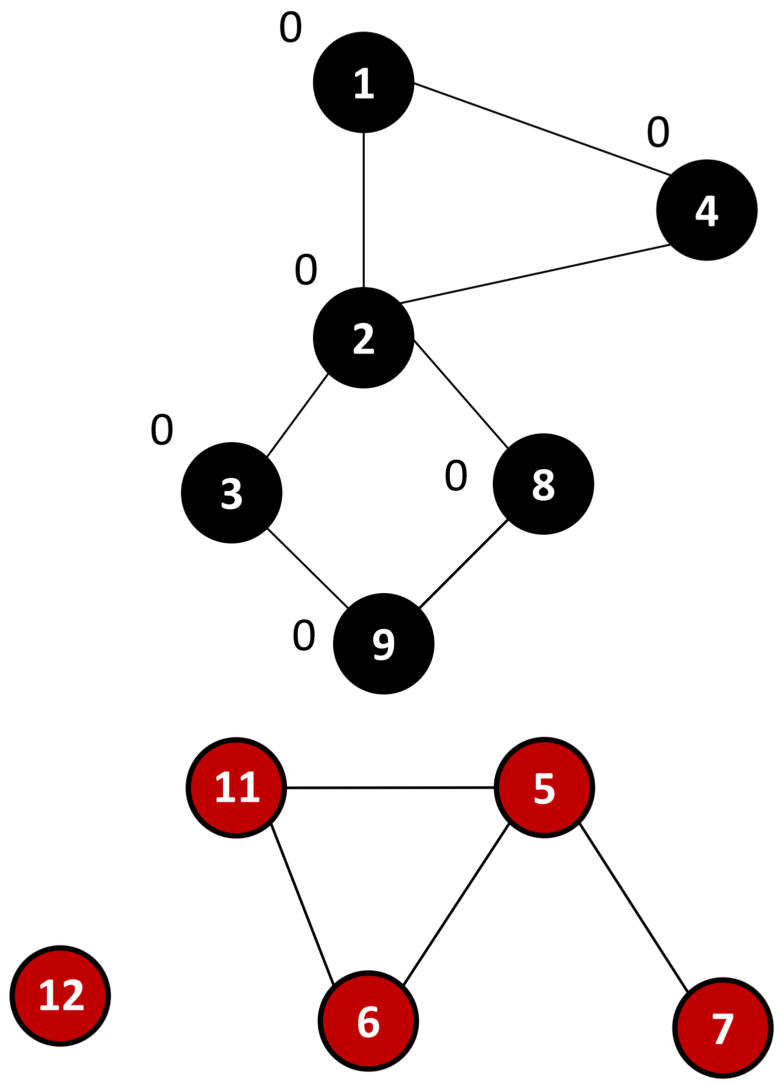
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
          neighbor, n, of node
    LCCVisit(n, c)
```



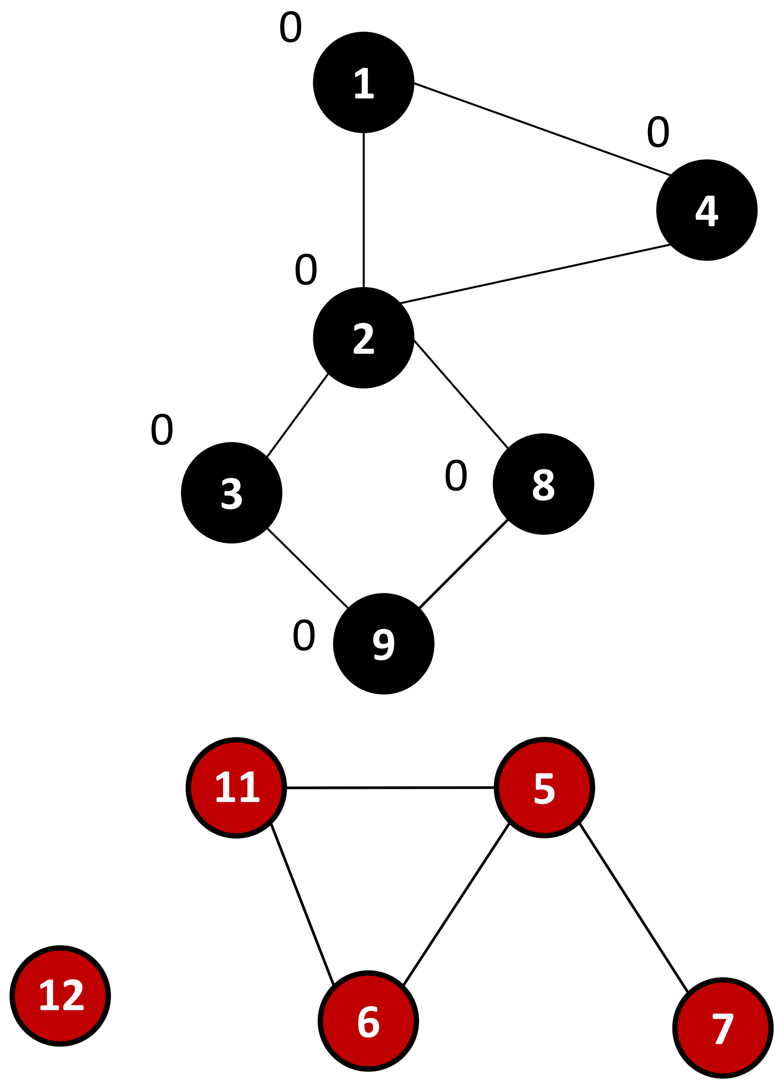
Connected component analysis using depth-first search

```
LabelConnectedComponents()
   component = 0
   for each node in graph
        if node not visited
            LCCVisit(node, component)
            component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
                neighbor, n, of node
      LCCVisit(n, c)
```



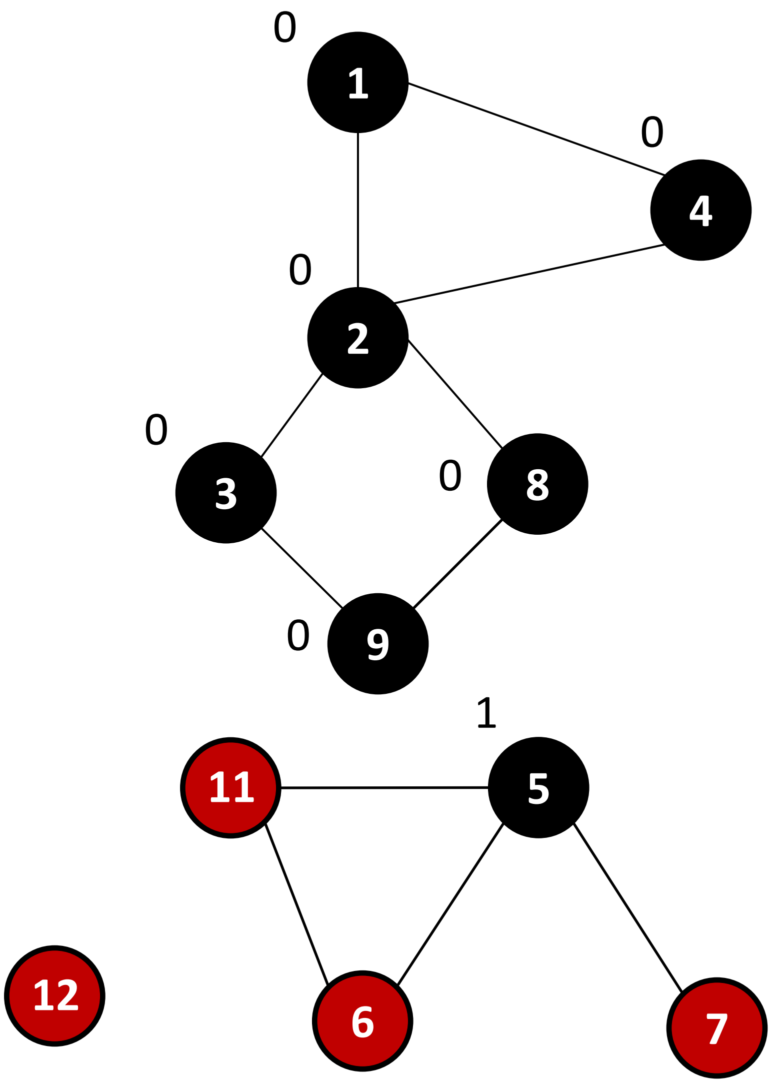
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
     if node not visited
        LCCVisit(node, component)
        component++


LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
           neighbor, n, of node
     LCCVisit(n, c)
```



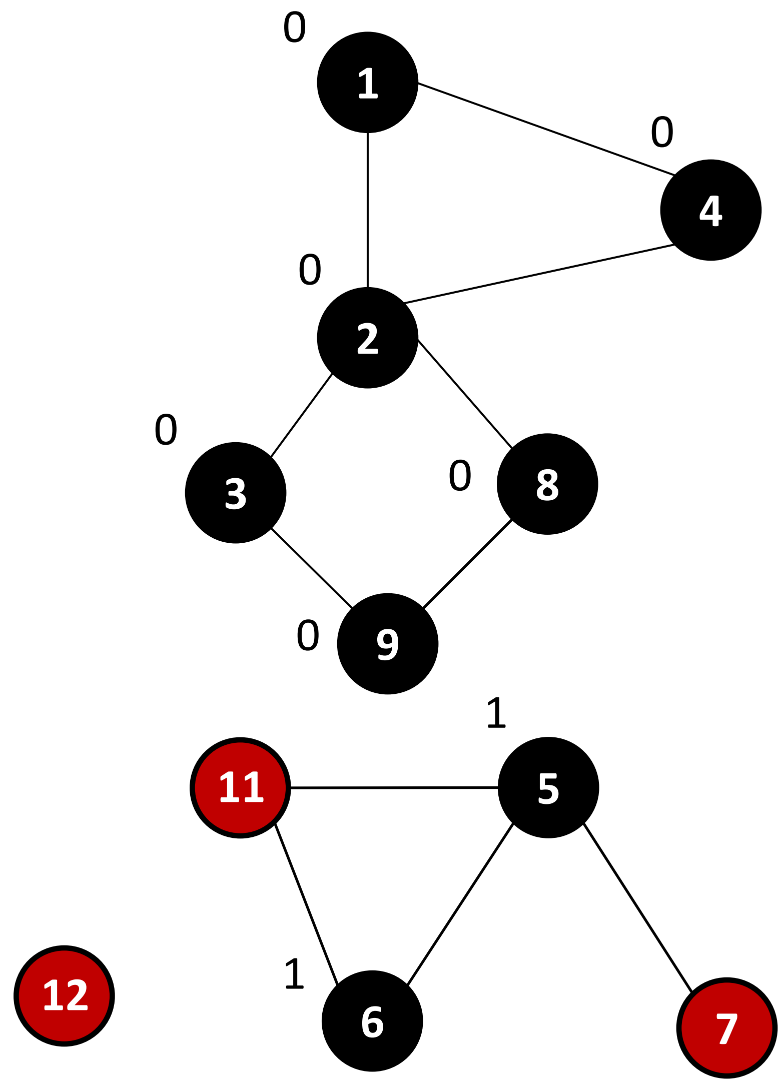
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
     if node not visited
        LCCVisit(node, component)
        component++


LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
           neighbor, n, of node
    LCCVisit(n, c)
```



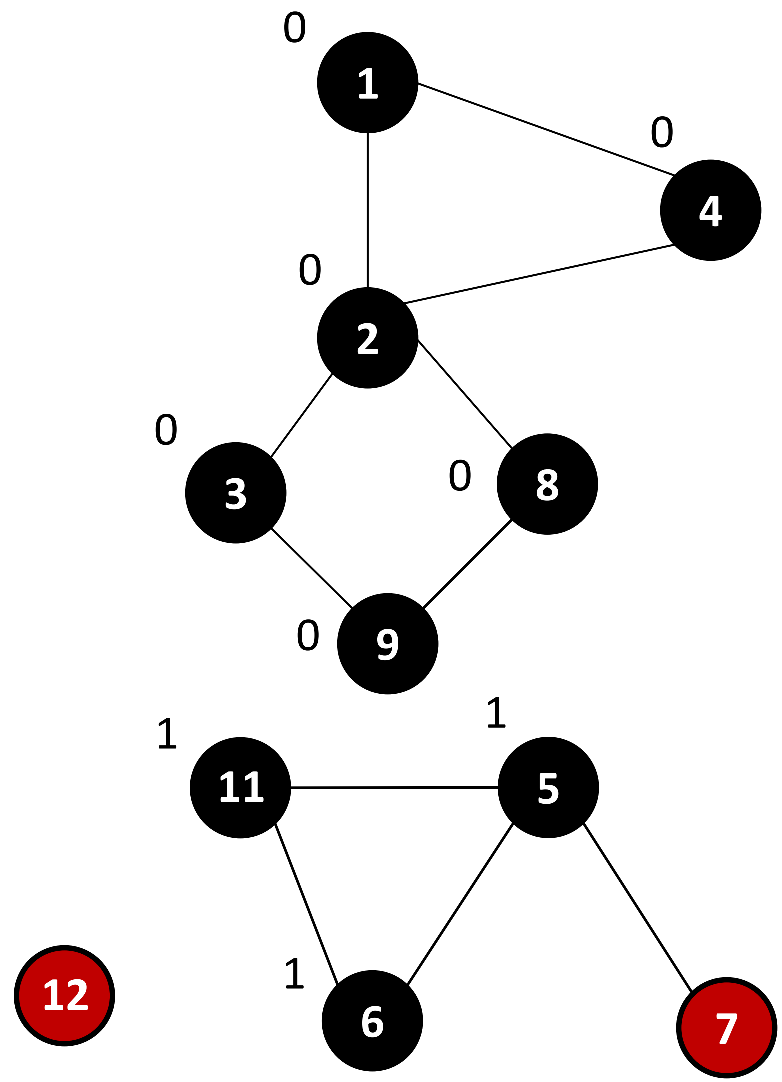
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
             neighbor, n, of node
    LCCVisit(n, c)
```



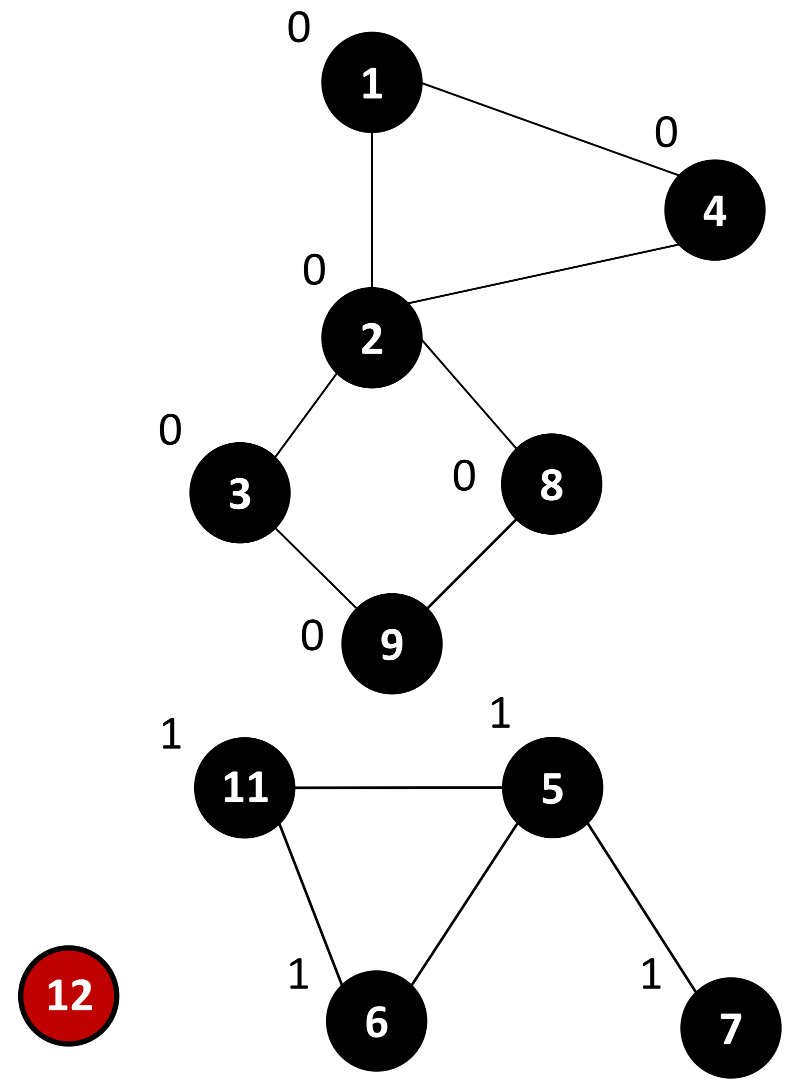
Connected component analysis using depth-first search

```
LabelConnectedComponents()
   component = 0
   for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
            neighbor, n, of node
      LCCVisit(n, c)
```



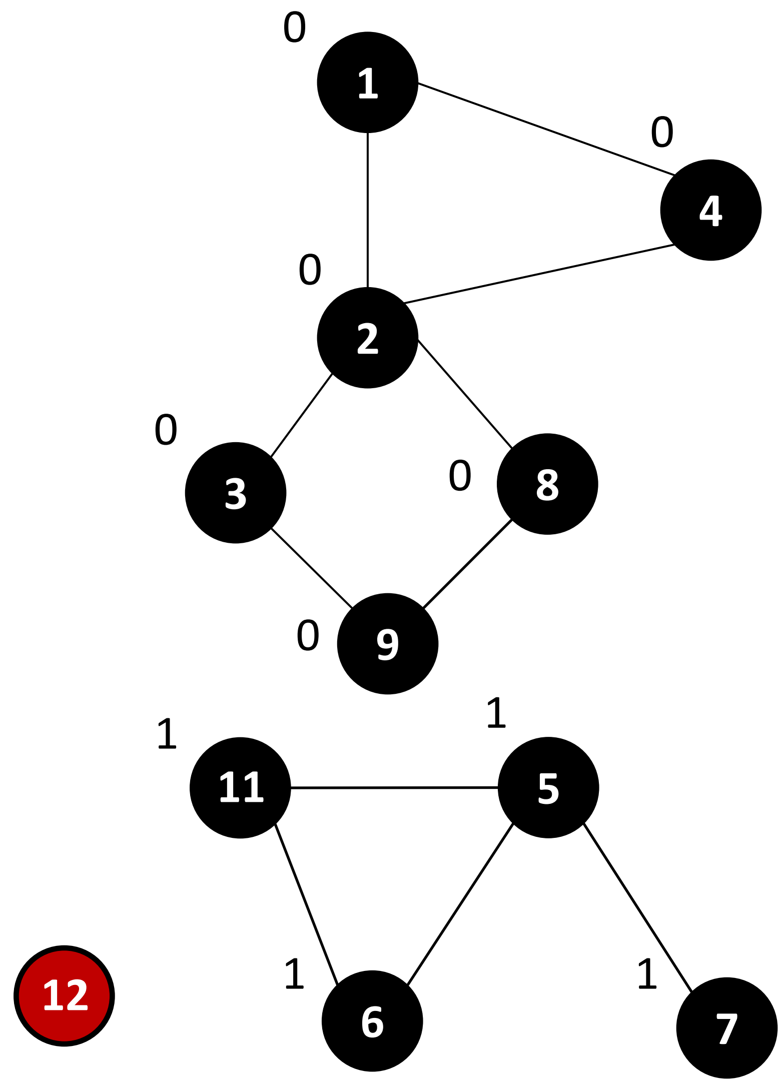
Connected component analysis using depth-first search

```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
            neighbor, n, of node
    LCCVisit(n, c)
```



Connected component analysis using depth-first search

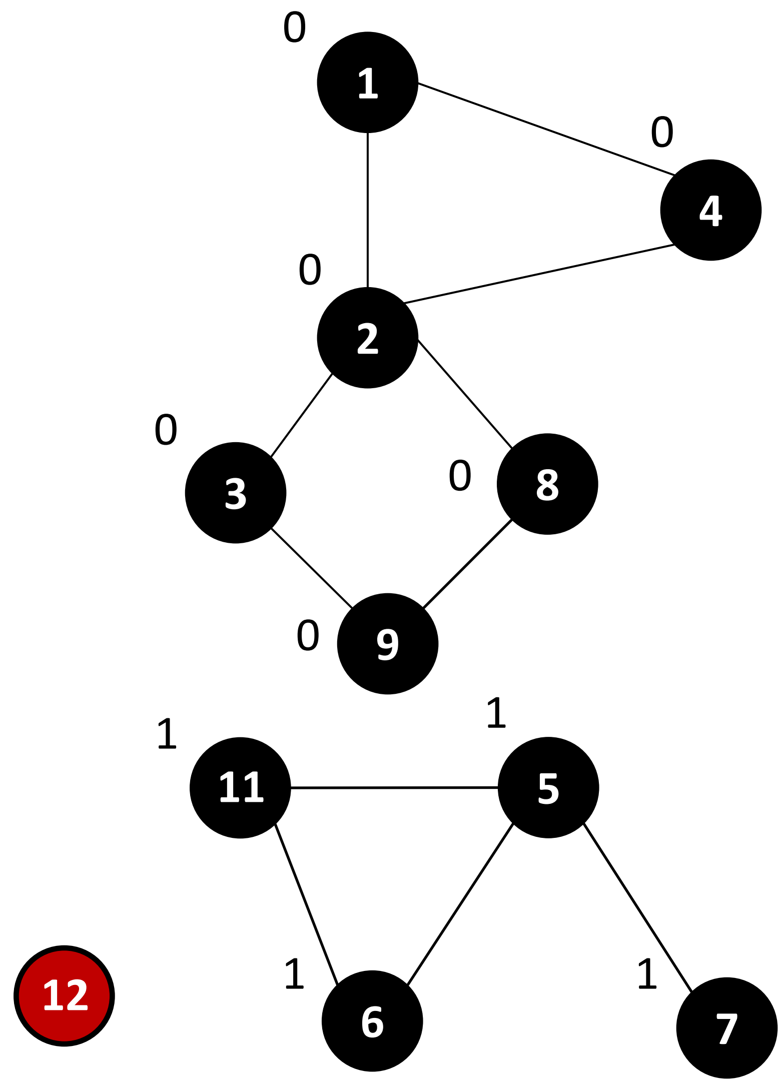
```
LabelConnectedComponents()
   component = 0
   for each node in graph
      if node not visited
         LCCVisit(node, component)
         component++

LCCVisit(node, c)
   mark node visited
   node.component = c
   for each unvisited
            neighbor, n, of node
      LCCVisit(n, c)
```



Connected component analysis using depth-first search

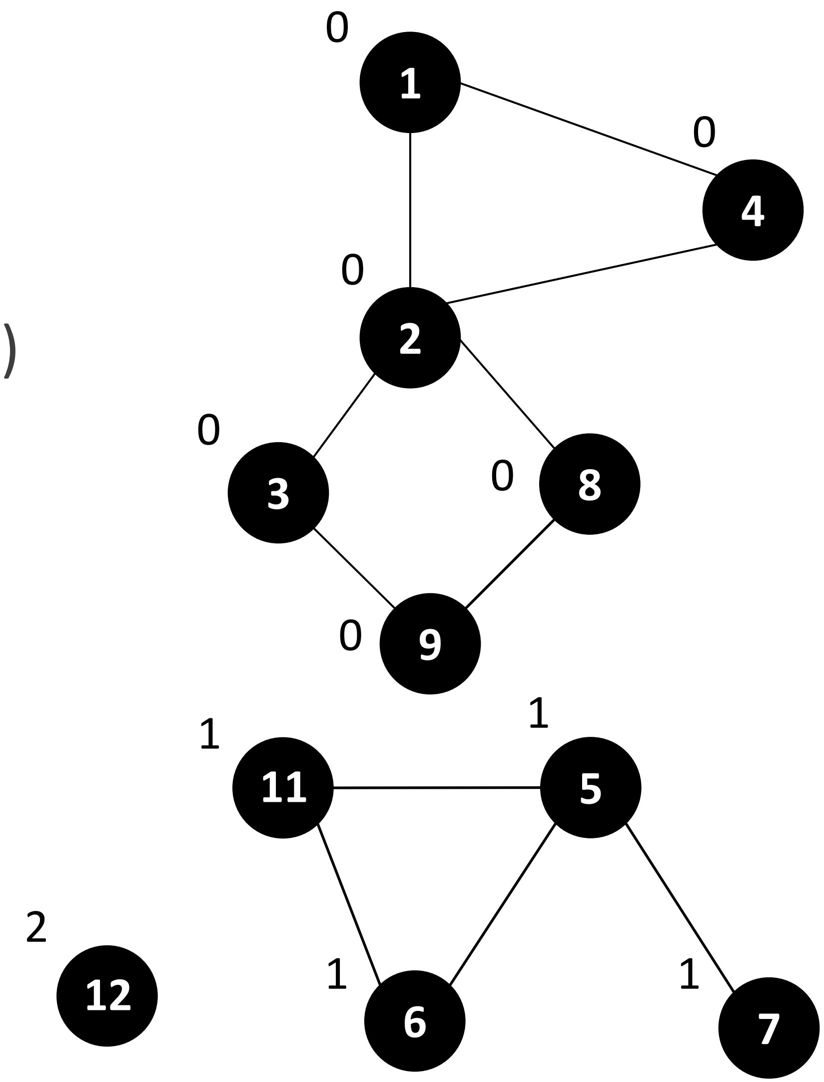
LabelConnectedComponents()
  component = 0
  for each node in graph
    if node not visited
      LCCVisit(node, component)
      component++

LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
        neighbor, n, of node
    LCCVisit(n, c)



Connected component analysis using depth-first search

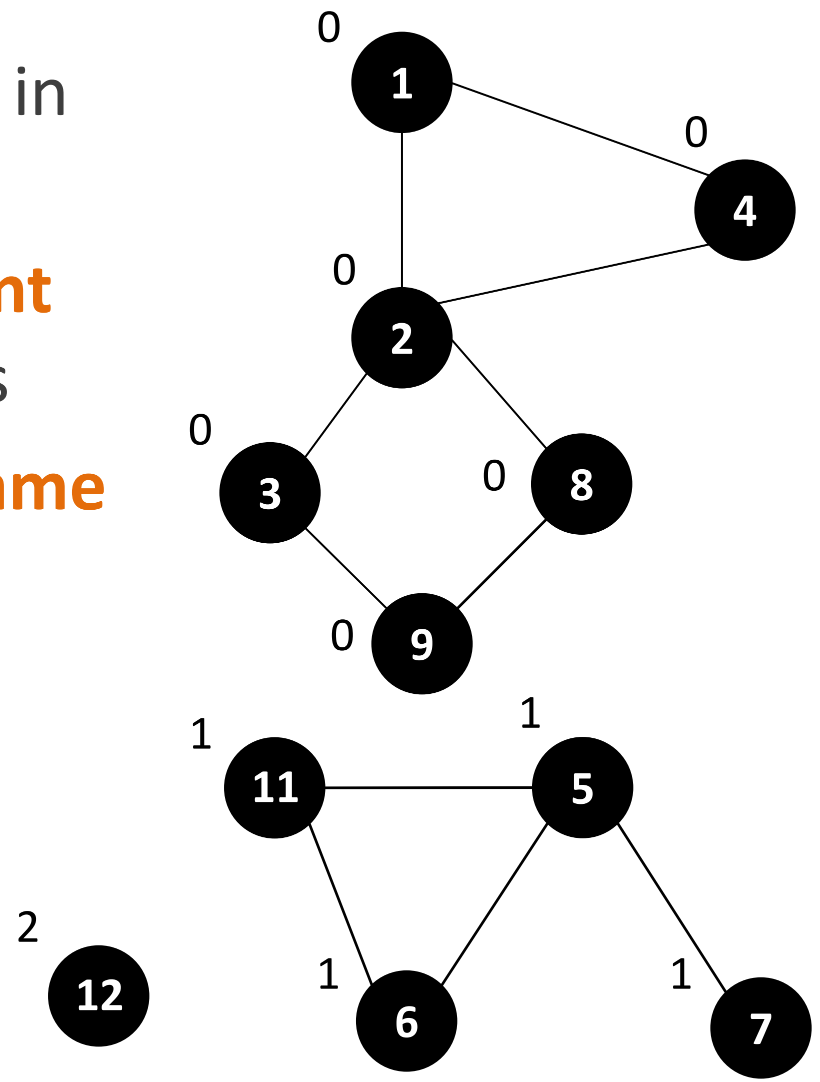
```
LabelConnectedComponents()
  component = 0
  for each node in graph
      if node not visited
          LCCVisit(node, component)
          component++


LCCVisit(node, c)
  mark node visited
  node.component = c
  for each unvisited
            neighbor, n, of node
    LCCVisit(n, c)
```



Testing if two nodes are connected

This lets us **test connectivity** in **constant time**

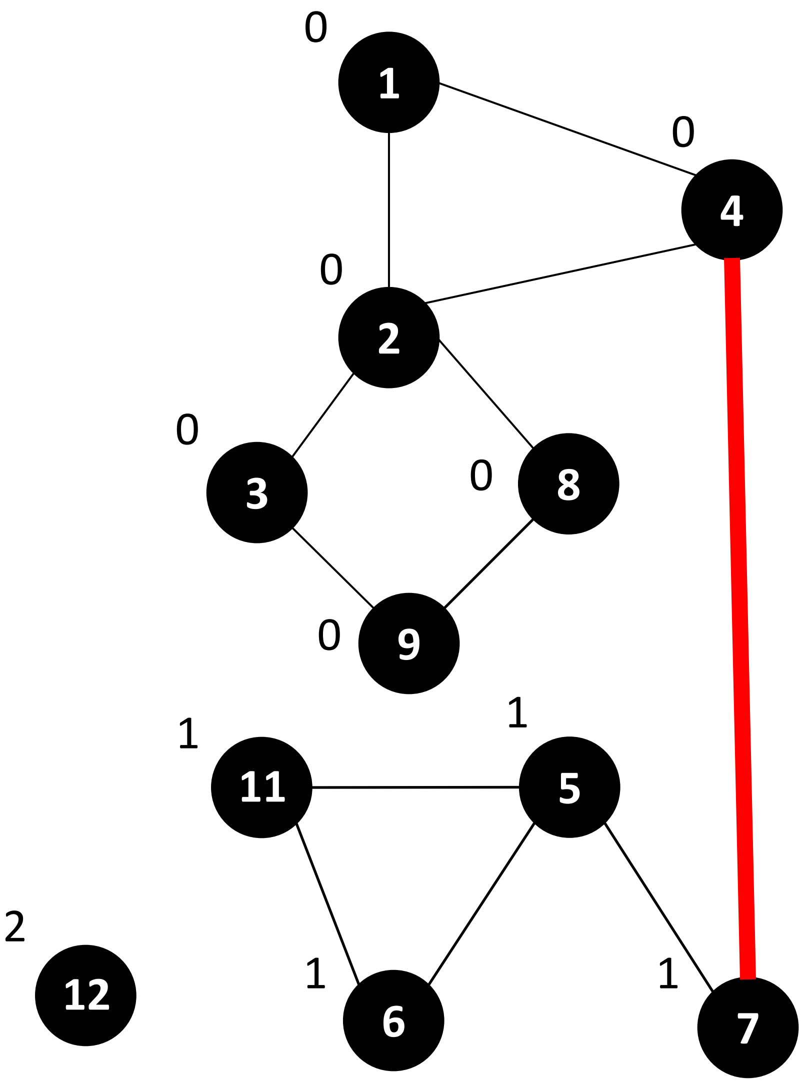
- Just look up the **component numbers** of the two nodes
- And check if they're the **same**



Testing if two nodes are connected

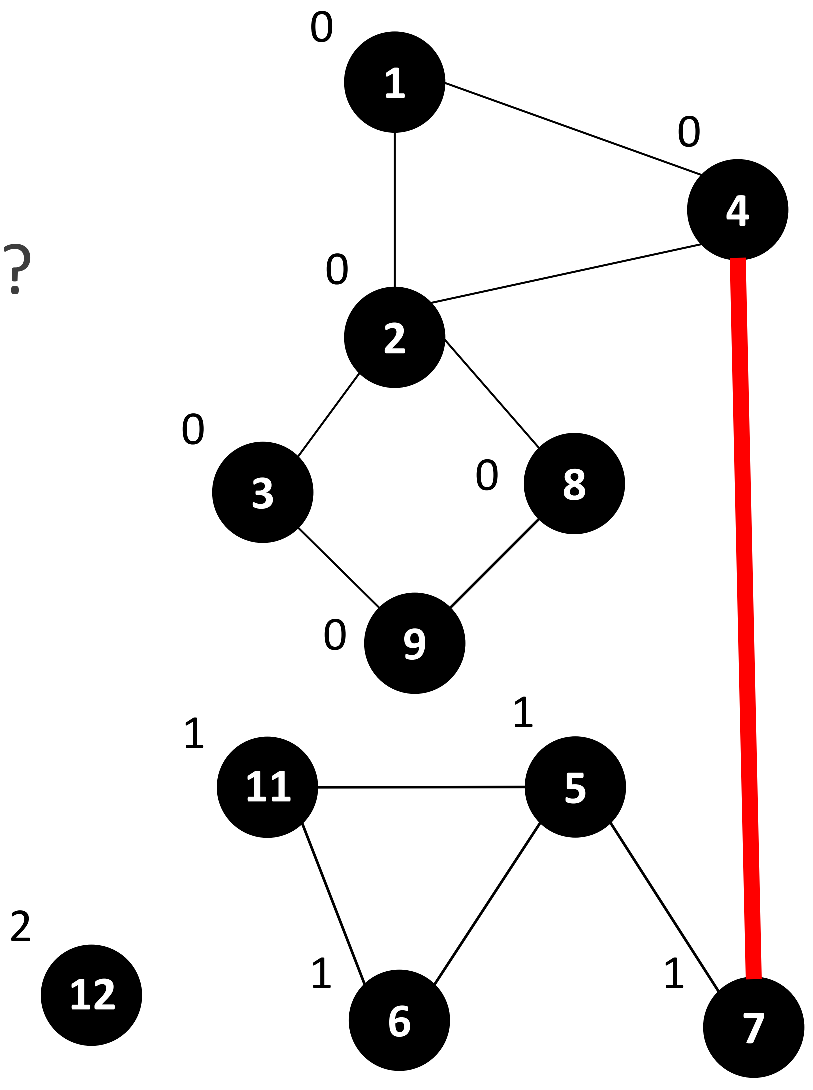
- Oh, **your boss called**
- They want to **add an edge**

- You **don't mind do you**?



Dynamic set partition algorithms

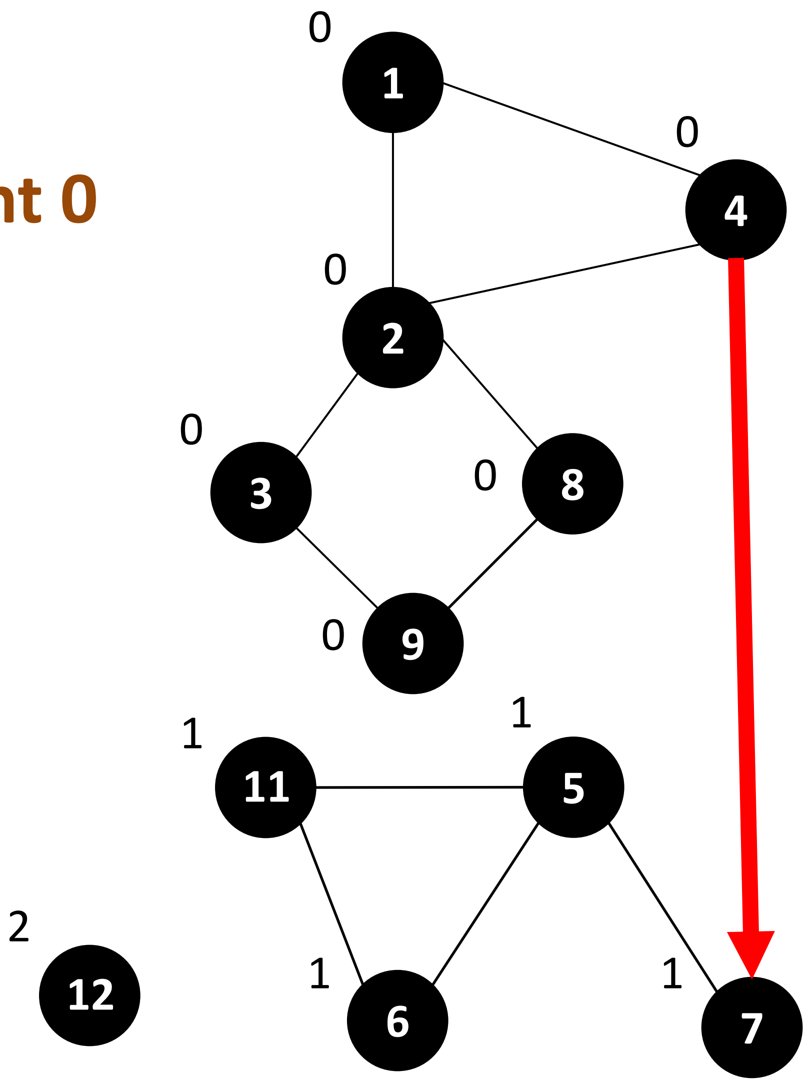
- How do we modify this to allow us to **incrementally merge** equivalence classes?



Dynamic set partition algorithms

Basic idea:

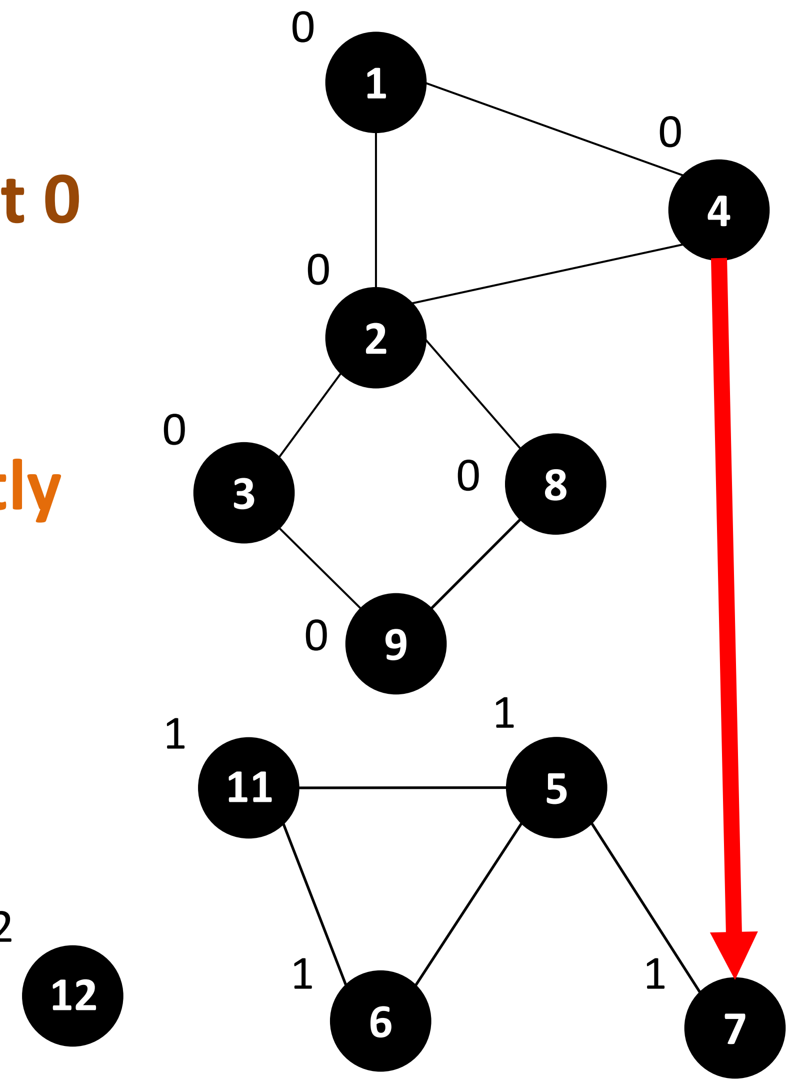
- Remember that **component 0 is “really” component 1**



Dynamic set partition algorithms

Basic idea:

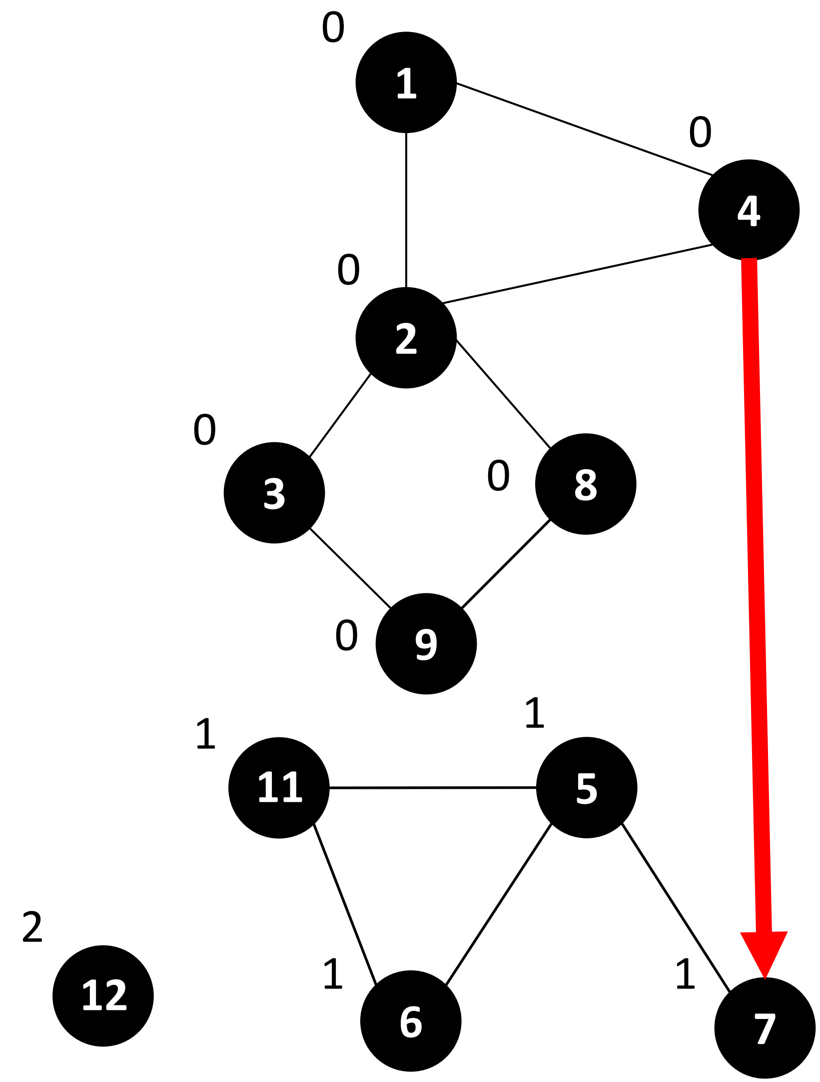
- Remember that **component 0 is "really" component 1**
- Then **don't compare** the component numbers **directly**
- Compare the "**real**" component numbers



Dynamic set partition algorithms

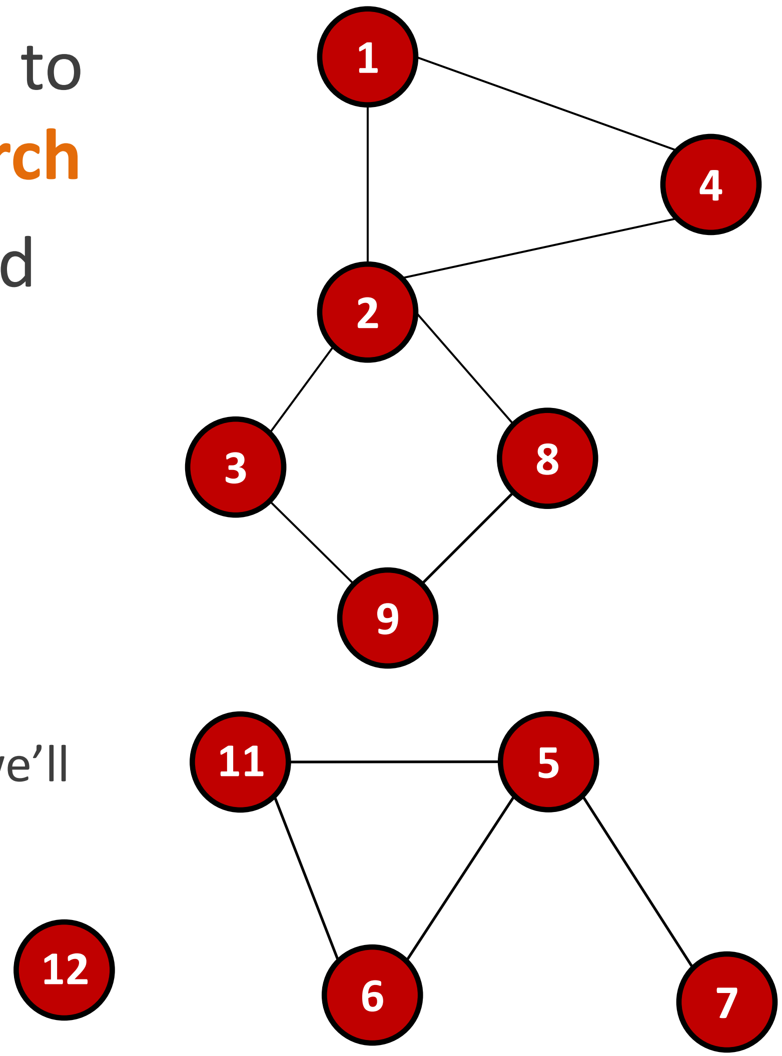
Example:

- Node 0 and node 11 are **connected**
  - Because node 0 is in **component 0**
  - But component 0 is **really component 1**
  - And **node 11 is also in component 1**



Computing connected components

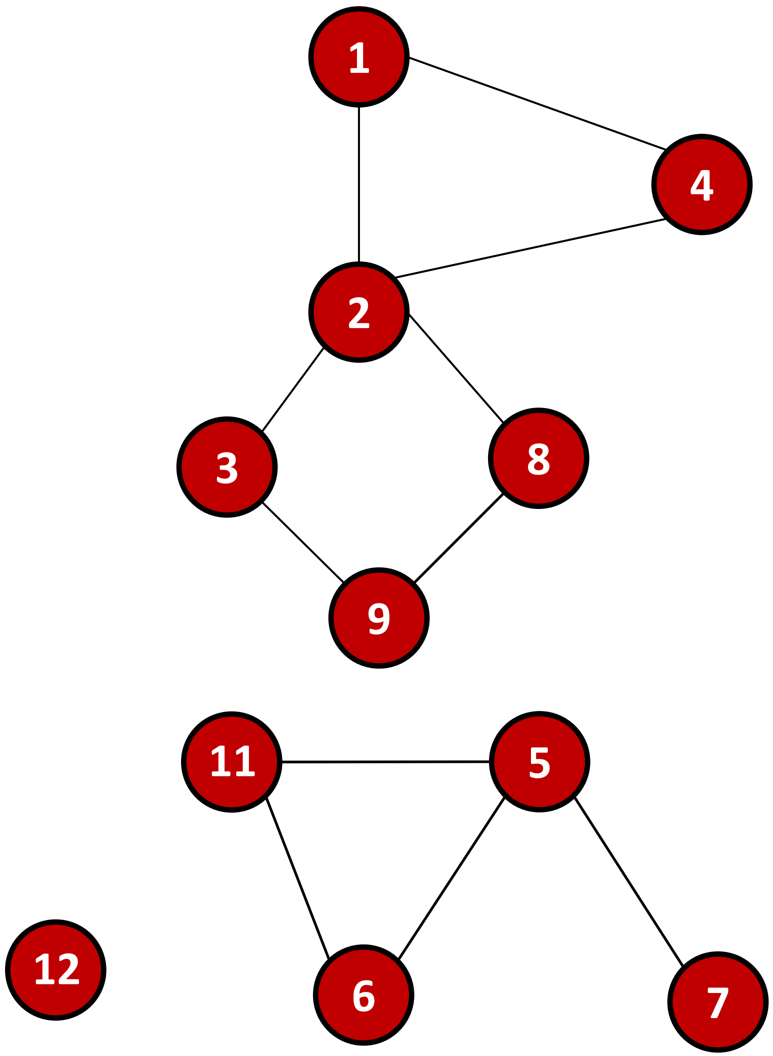
- In fact, we can use this idea to **replace the depth-first search**
- And **compute** the connected components **directly**



Note: this is going to be hand-wavy, but we'll make the algorithms precise later

Computing connected components

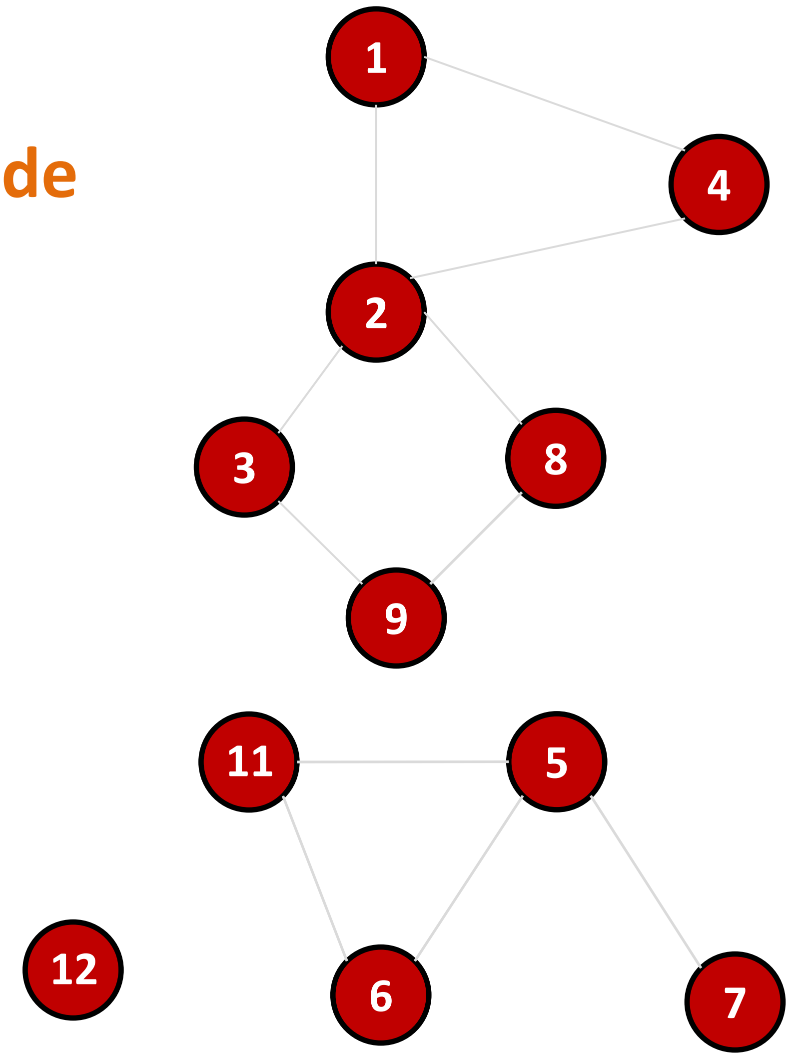
Here's the idea



Computing connected components

Here's the idea

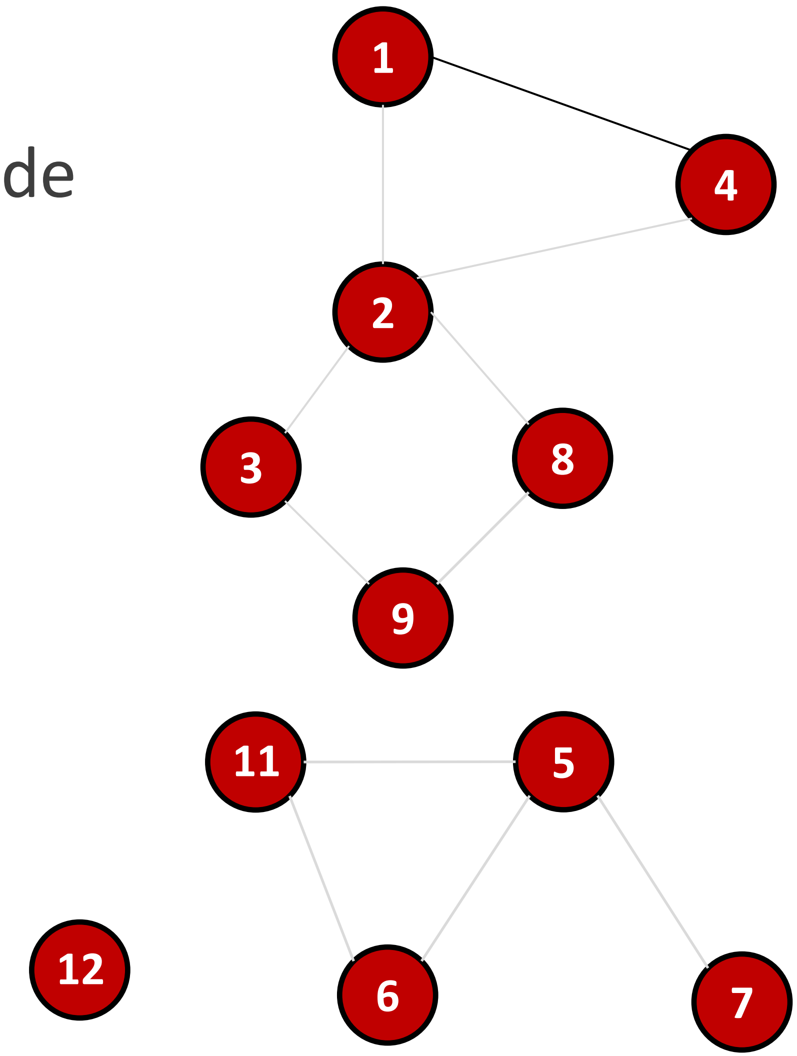
- Start by assuming **every node** is in its **own** connected **component**



Computing connected components

Here's the idea

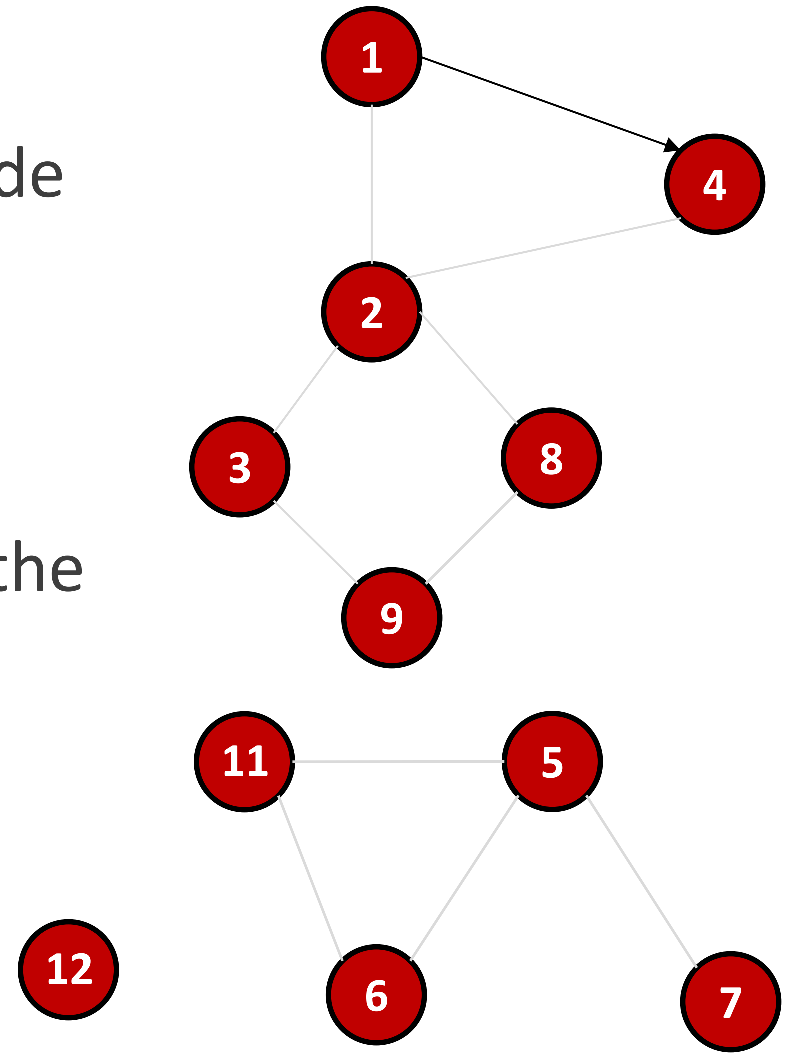
- Start by assuming every node is in its own connected component
- **Pick an edge**



Computing connected components

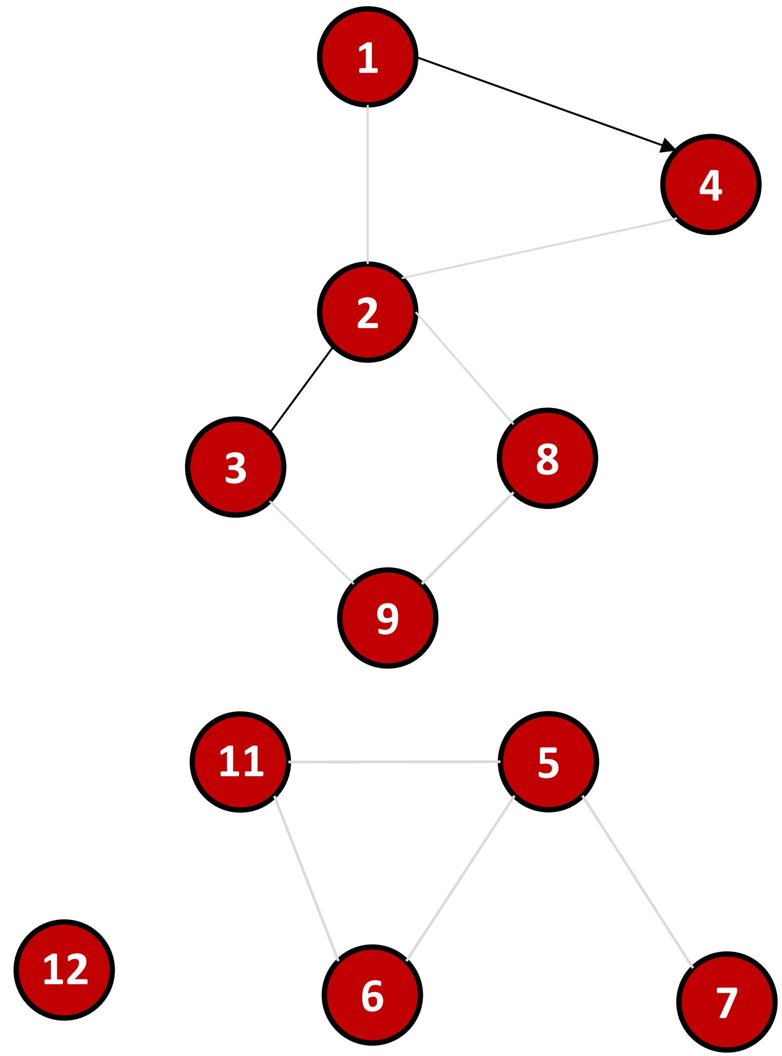
Here's the idea

- Start by assuming every node is in its own connected component
- Pick an edge
- **Merge** the components of the two nodes
  - By remembering that one
  - Is in the component of the other



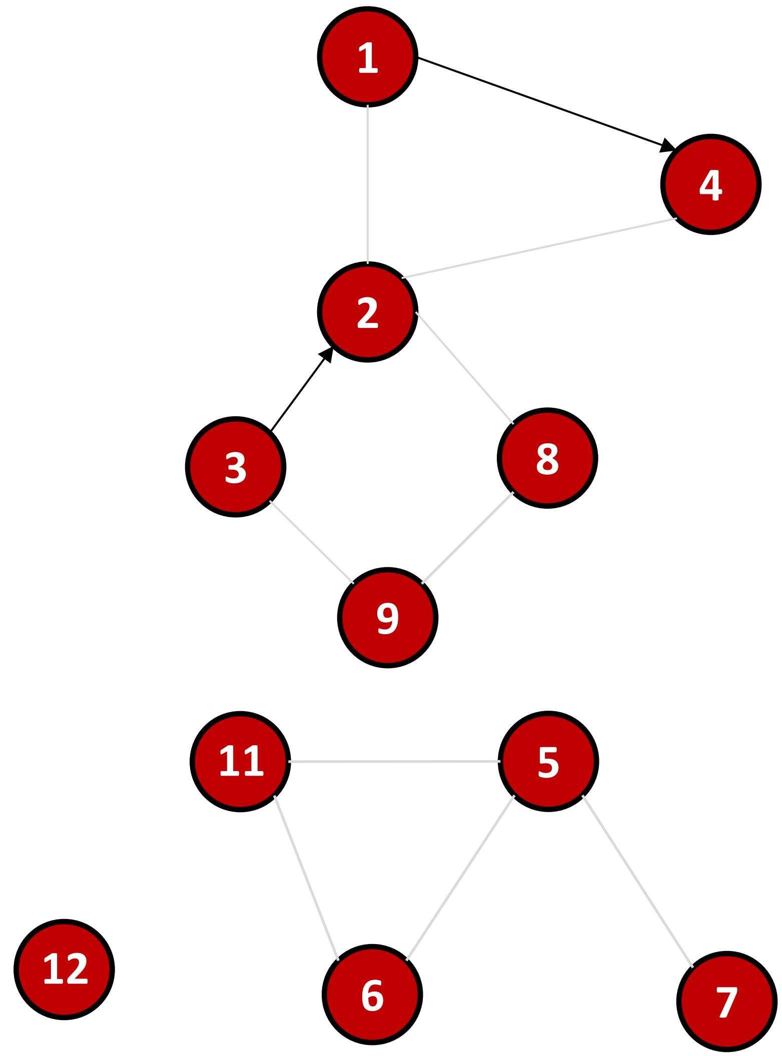
Computing connected components

- Pick **another edge**



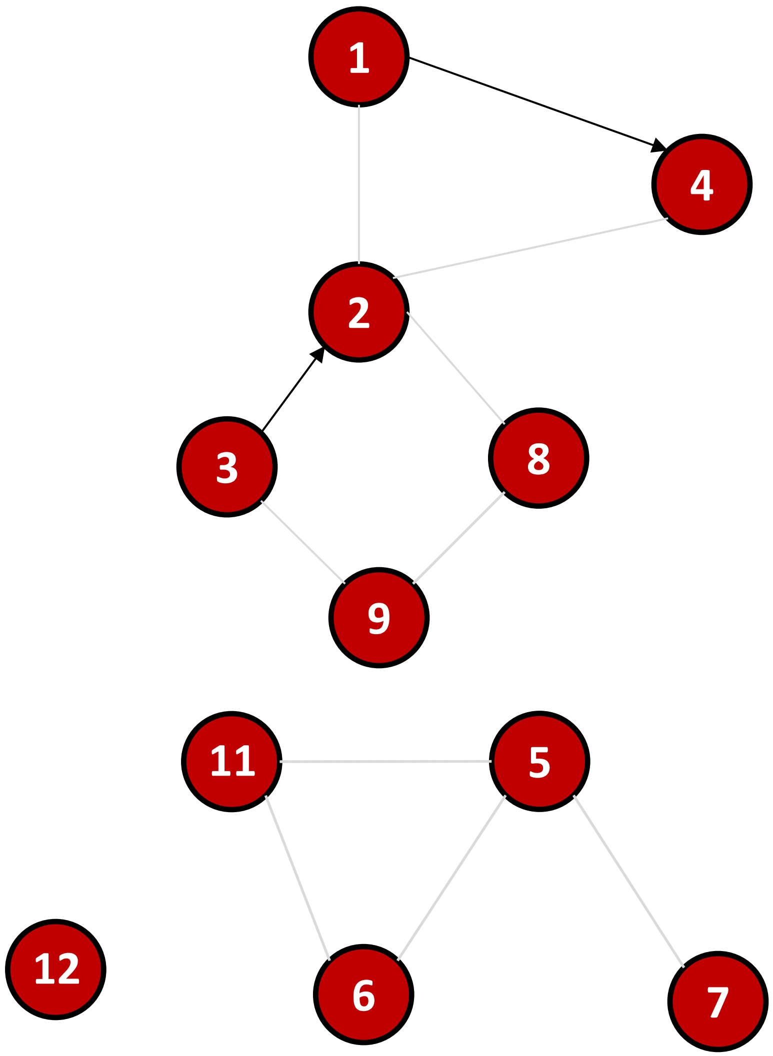
Computing connected components

- Pick another edge
- **Merge** again



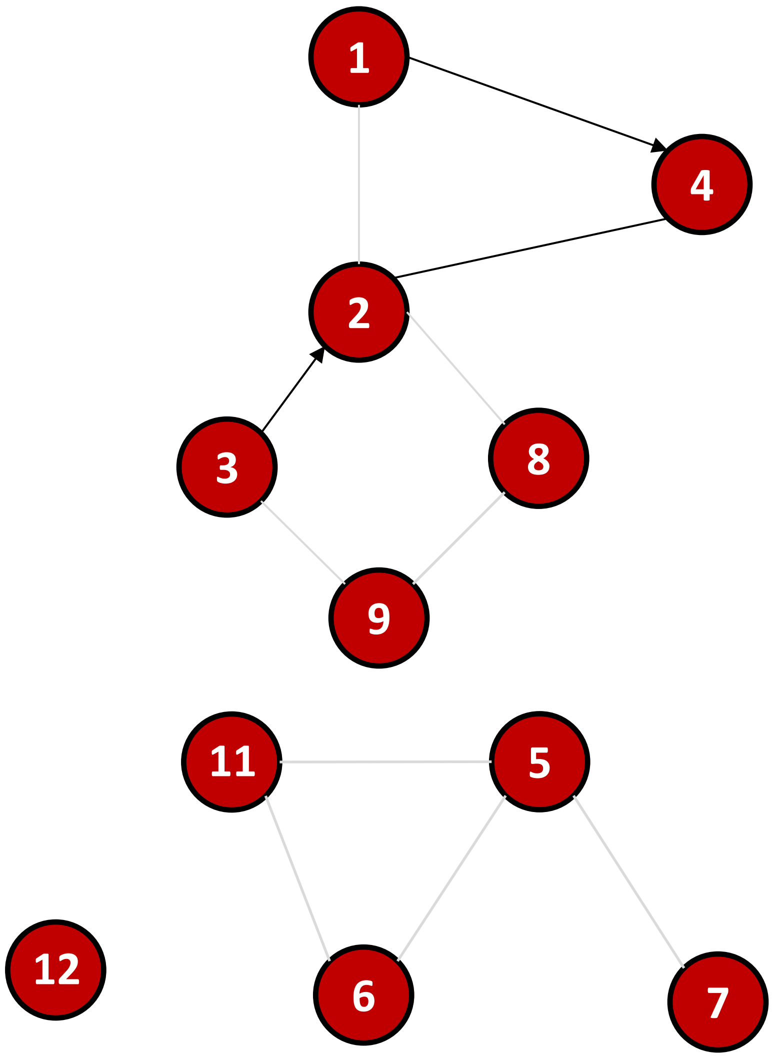
Computing connected components

- Now we have that
  - 1 is really in 4's component
  - 3 is really in 2's component



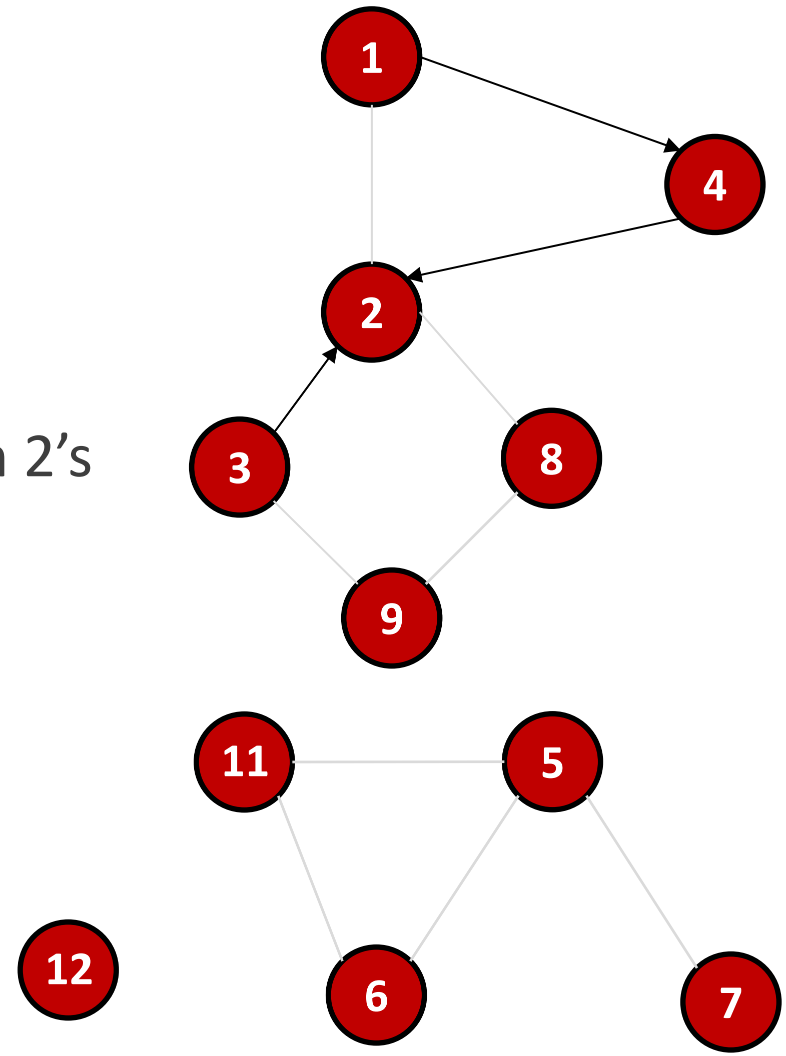
Computing connected components

- **Pick another** edge



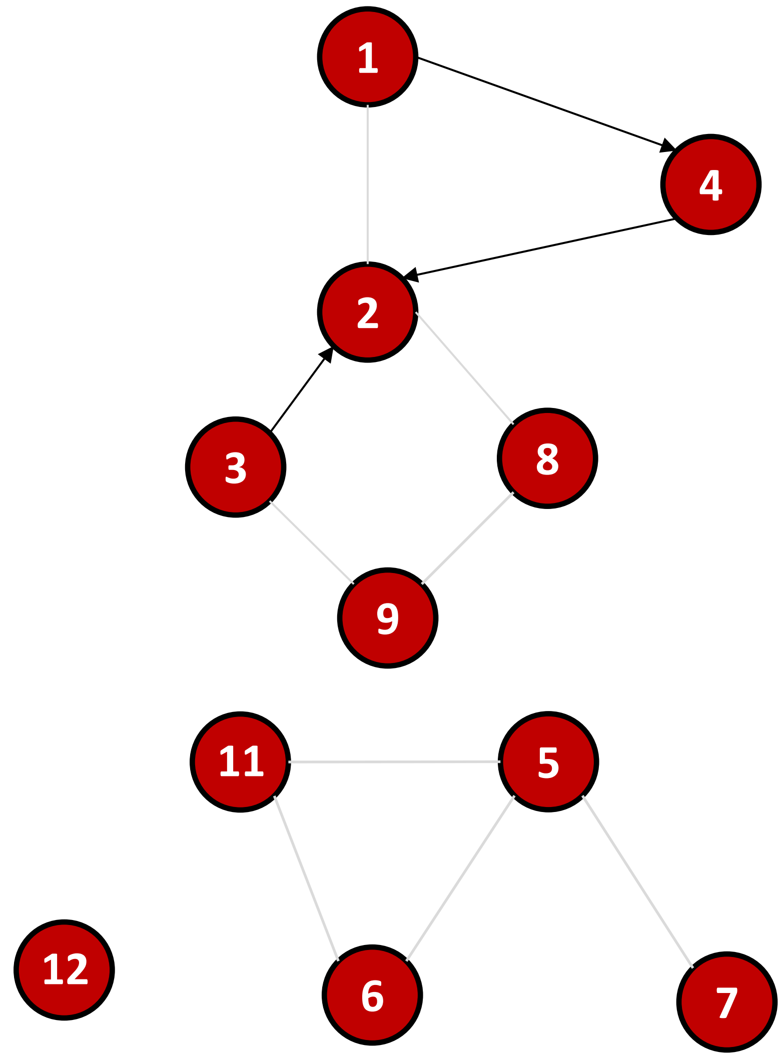
Computing connected components

- Pick another edge
- **Merge** again
  - 4 is now in 2's component
  - But 1 is in 4's component
  - So really, 1, 3, and 4 are all in 2's component



Computing connected components

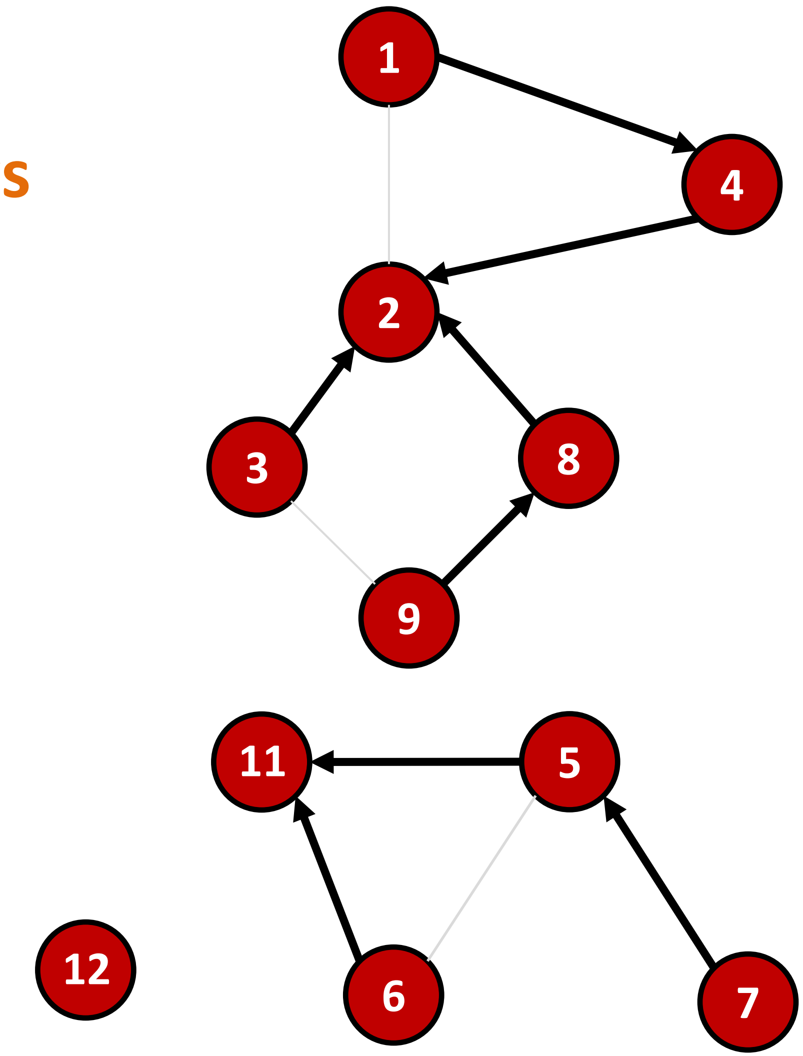
Keep doing this



Computing connected components

Keep doing this

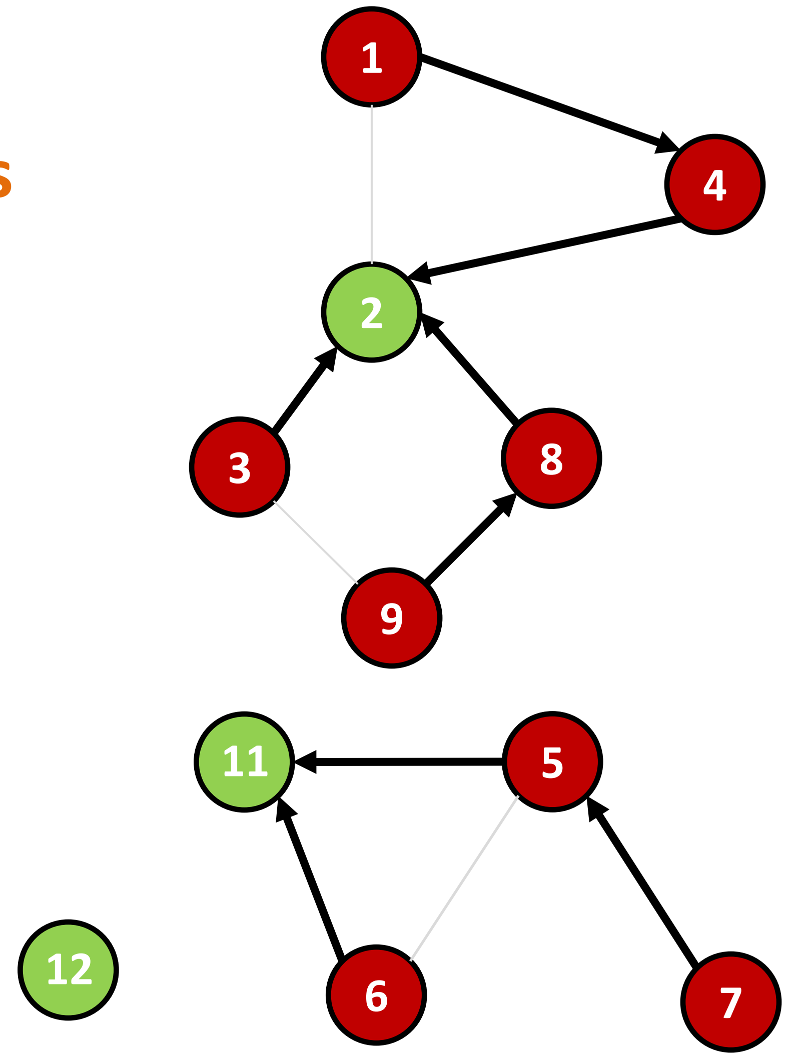
- And we end up with **arrows**
- From all the **nodes in a connected component**



Computing connected components

Keep doing this

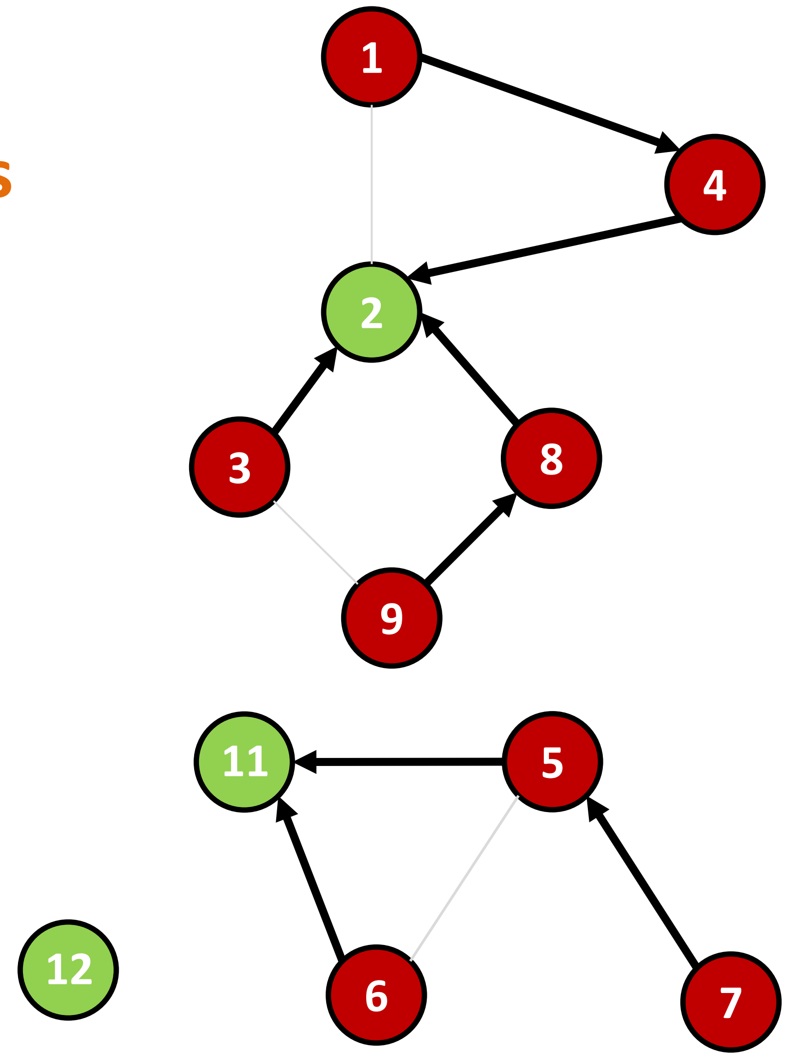
- And we end up with **arrows**
- From all the **nodes in a connected component**
- **Flowing to a single node** in each component



Computing connected components

Keep doing this

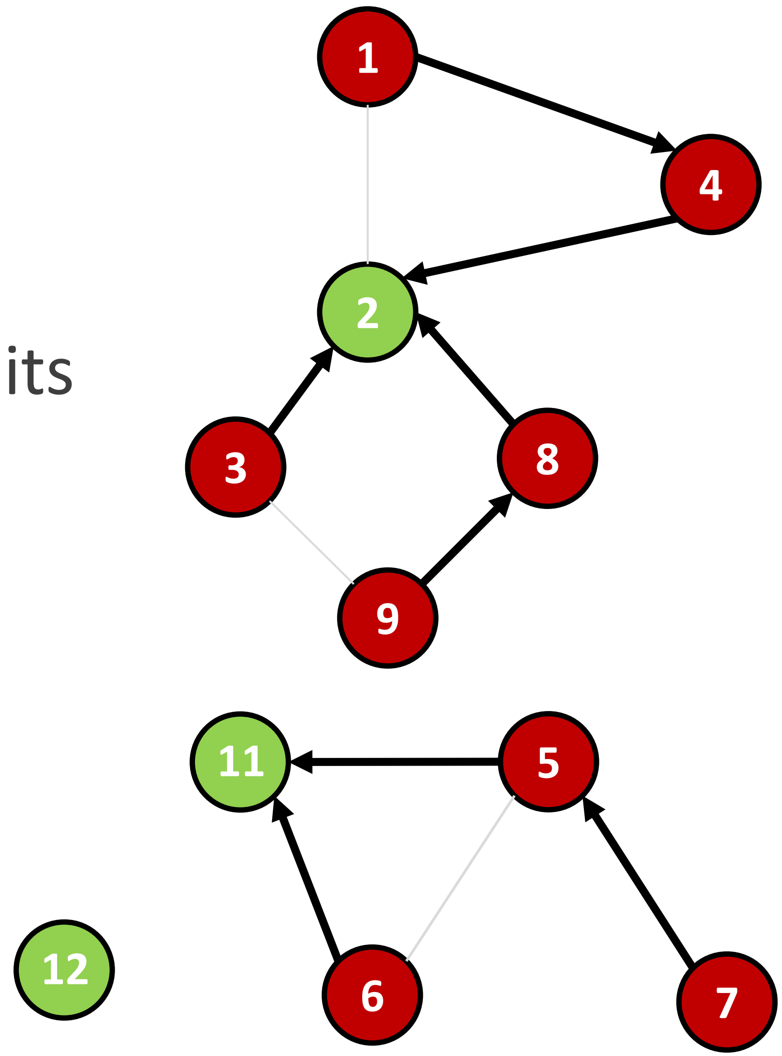
- And we end up with **arrows**
- From all the **nodes in a connected component**
- **Flowing to a single node** in each component
- These are called the **representatives** of the components



Computing connected components

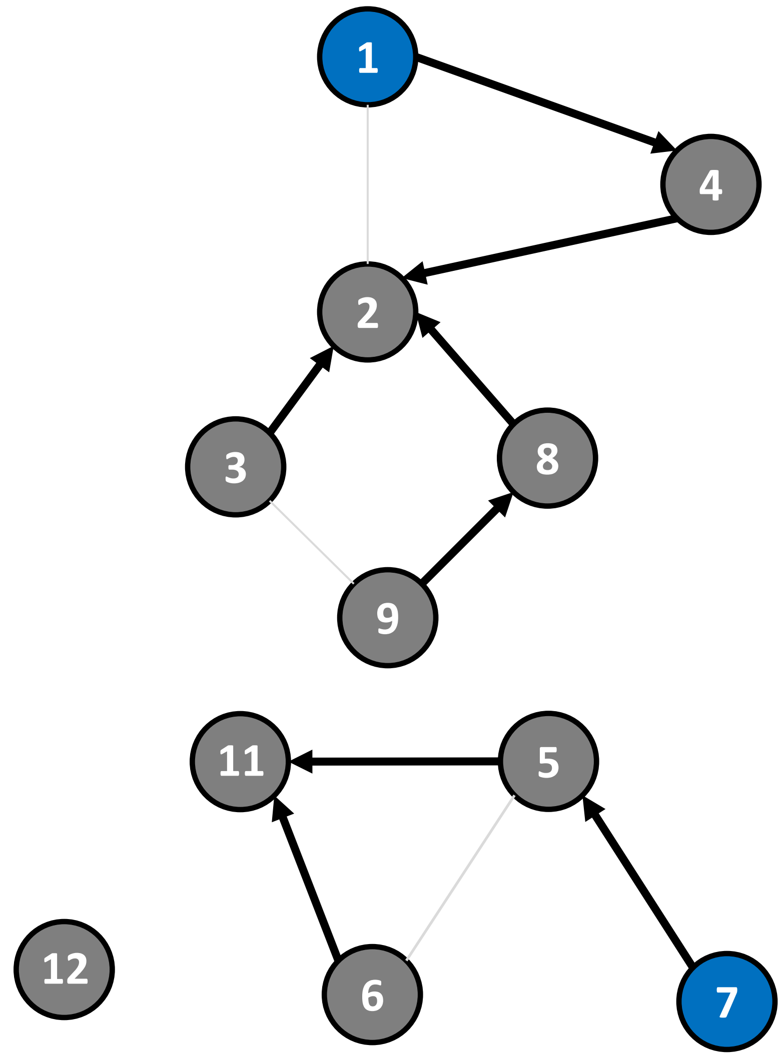
Each component forms a **tree**

- With arrows pointing **from children to parents**
- With the **representative** as its **root**
- So they collectively form a **forest**



Computing connected components

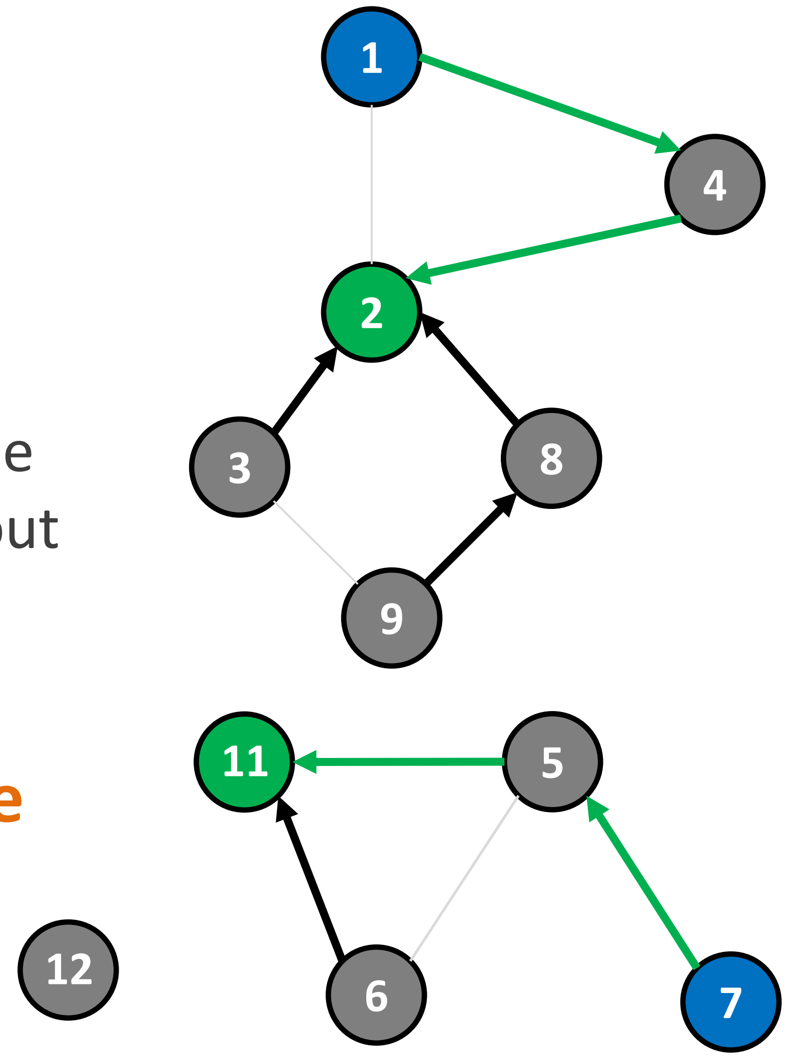
To decide if two nodes are connected



Computing connected components

To decide if two nodes are connected

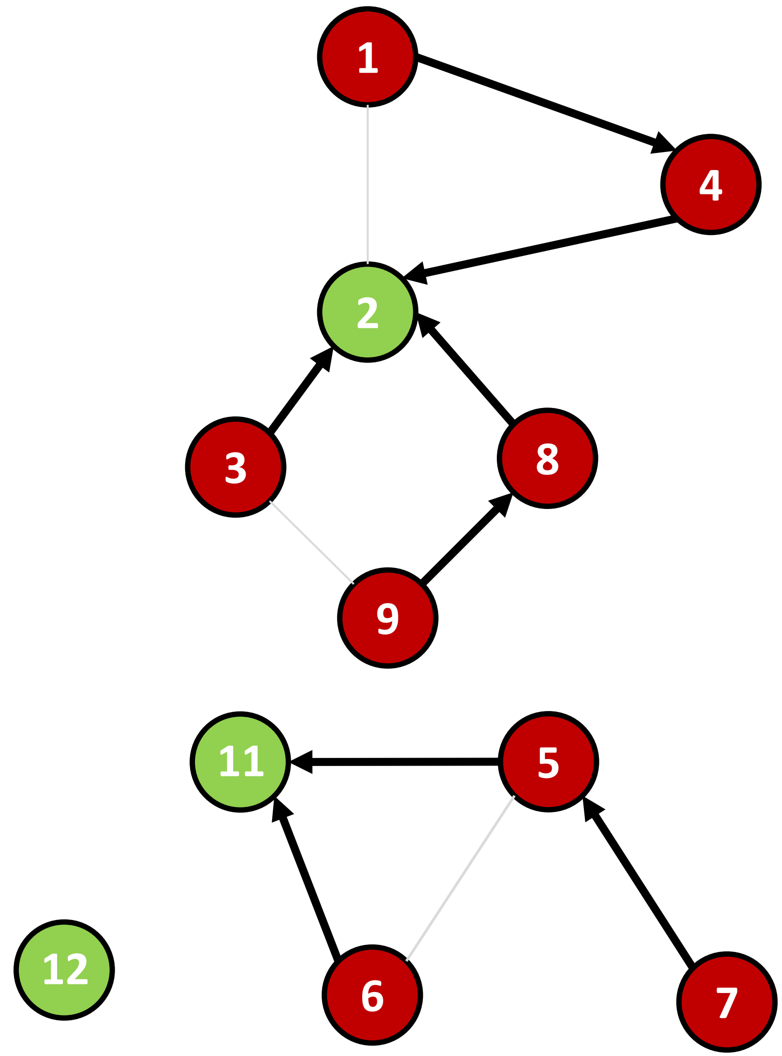
- **Follow the arrows** to their representatives
  - Easy to do because each node only has one edge pointing out
  - Except representatives, who have none
- Check if they have the **same representative**



Computing connected components

The cool thing is

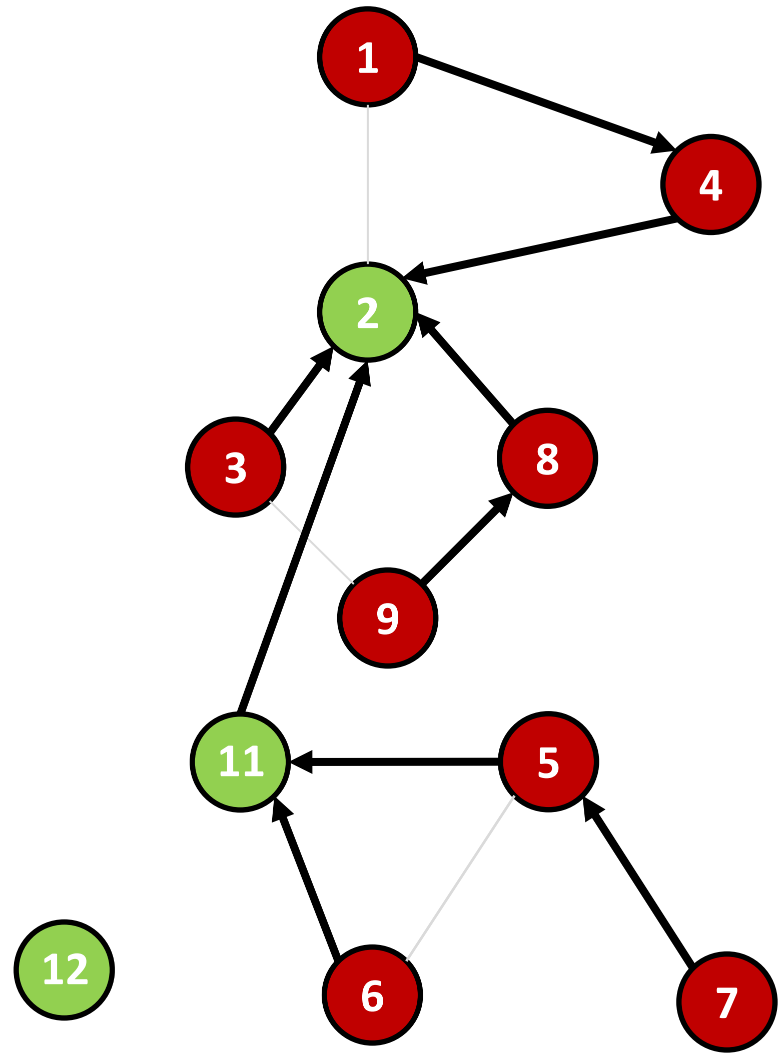
- To **merge two components**



Computing connected components

The cool thing is

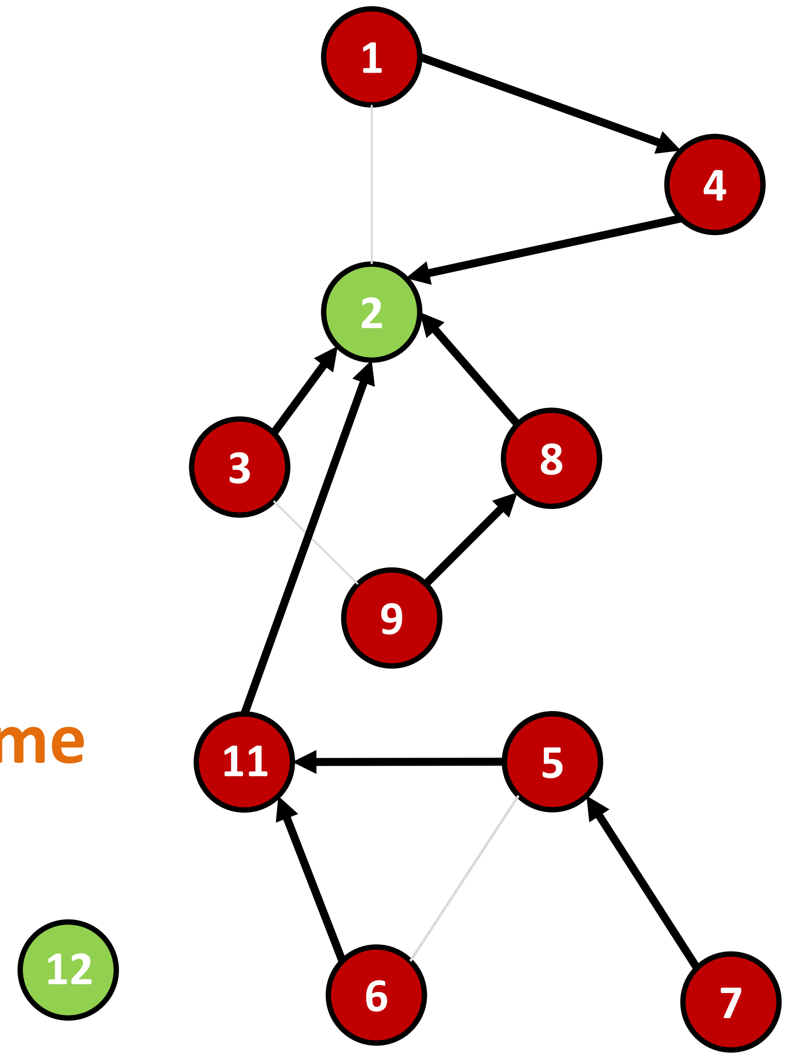
- To **merge two components**
- All we have to do is **add a pointer**
  - From one component's representative
  - To the other's



Computing connected components

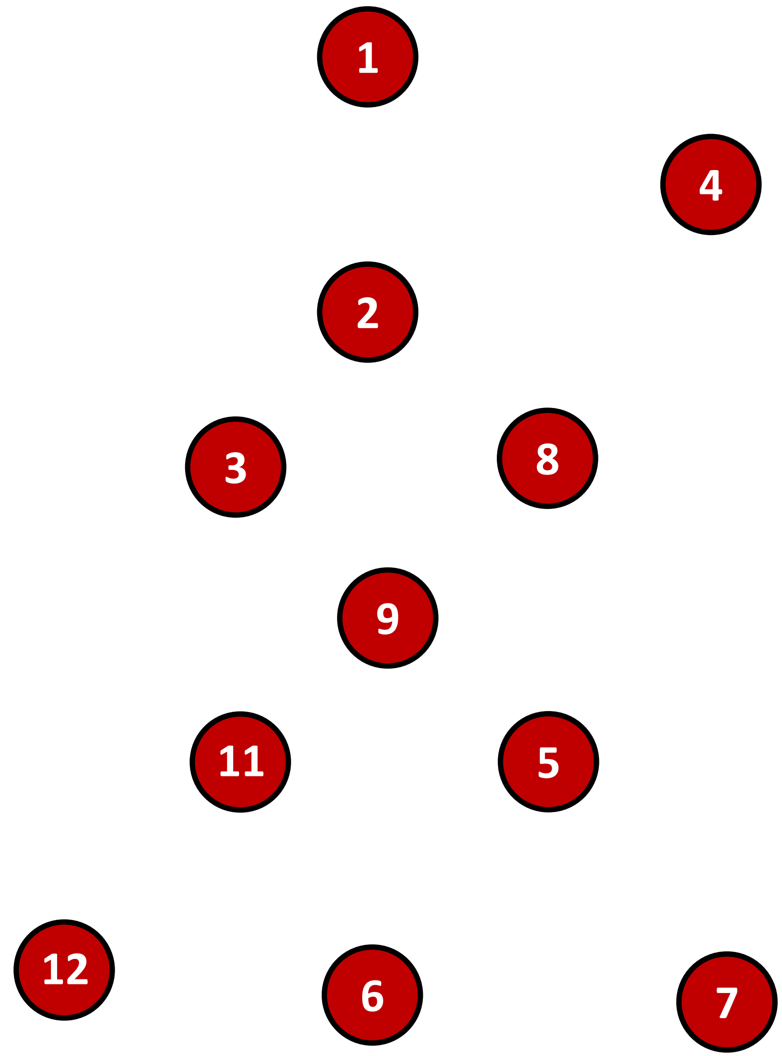
The cool thing is

- To **merge two components**
- All we have to do is **add a pointer**
  - From one component's representative
  - To the other's
- Now they both have the **same representative**



The union-find algorithm

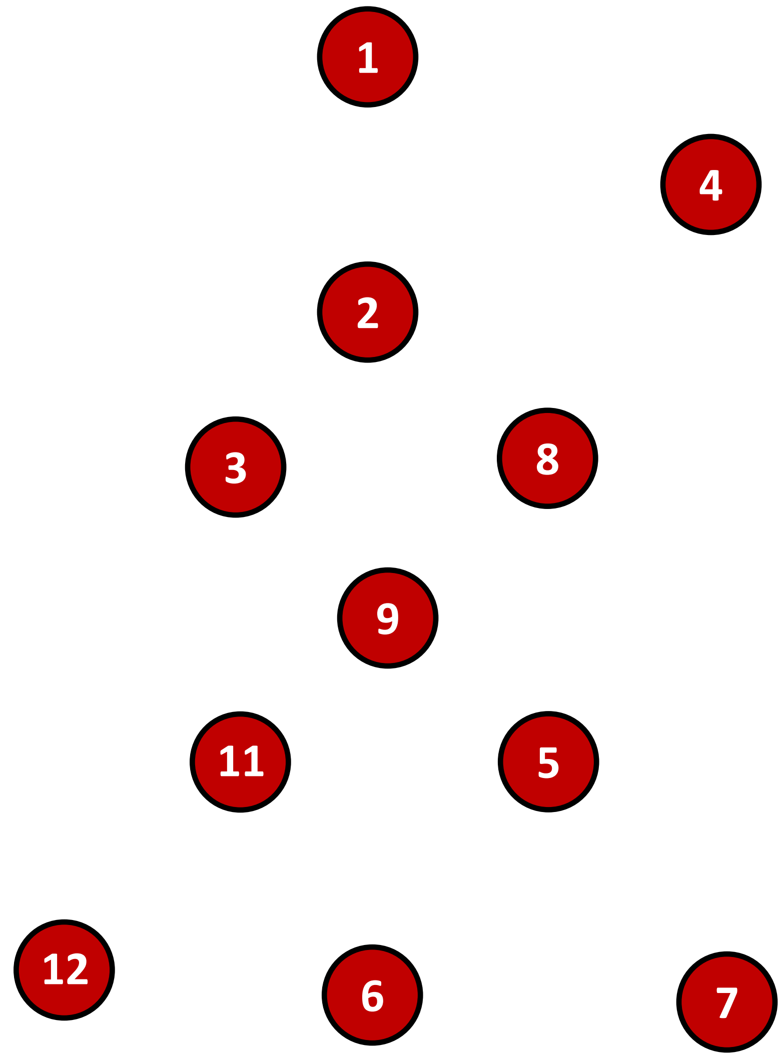
- This is a handwavy version of a **general algorithm**
- Called the **union-find** algorithm
- For working with **partitions** of sets



The union-find algorithm

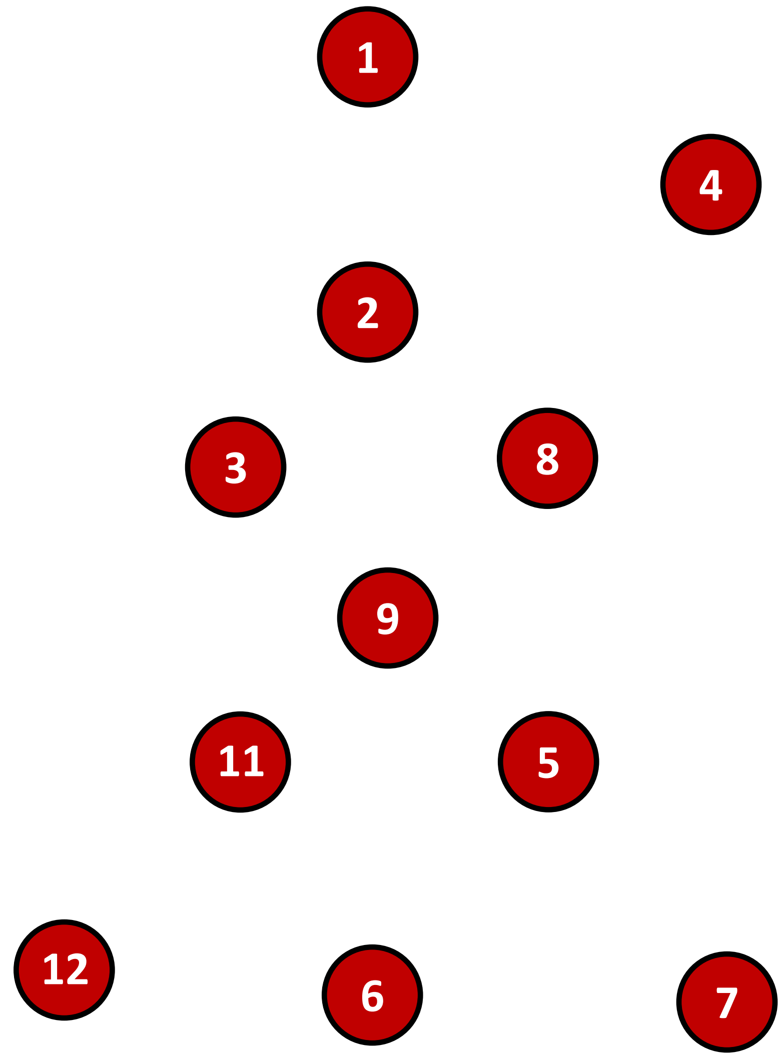
```
class SetElement {
  SetElement parent;
}
```

- Keep a **parent pointer** for each element of the set
  - Points to its parent in the tree
  - **Representatives** are roots, so parent is **null**
    - Or it's often **set to itself**
    - Which saves some work later



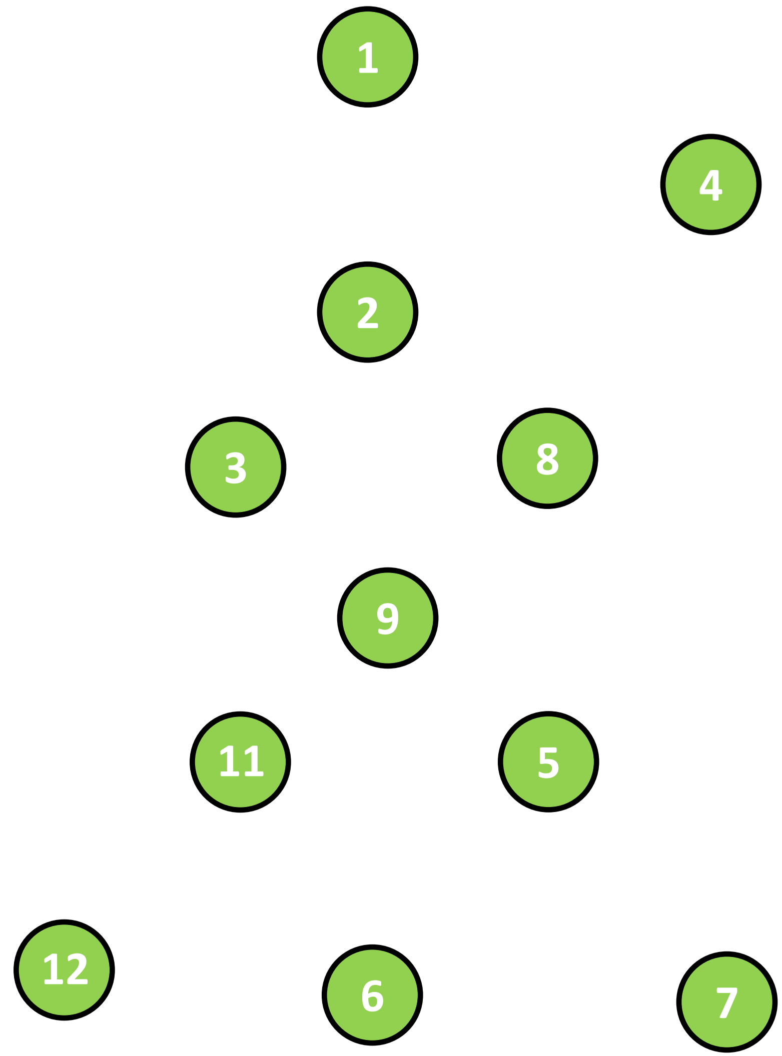
The union-find algorithm

- We start with all the elements in their **own groups** in the partition



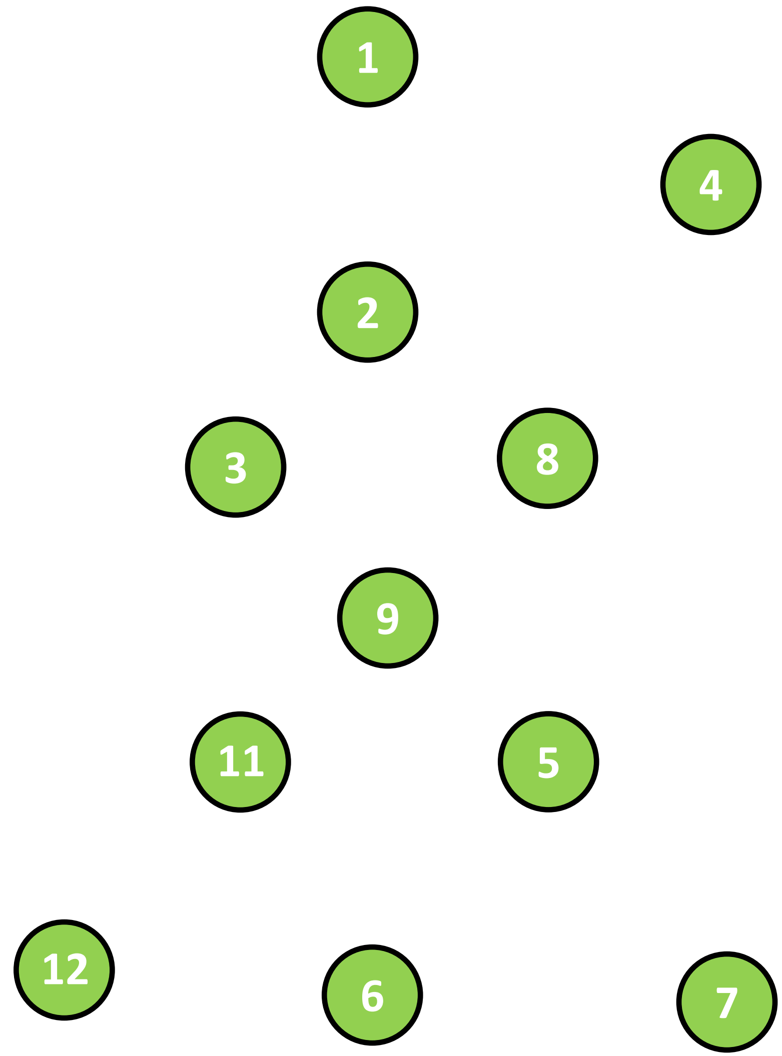
The union-find algorithm

- We start with all the elements in their **own groups** in the partition
- So they're all their **own representatives**



The union-find algorithm

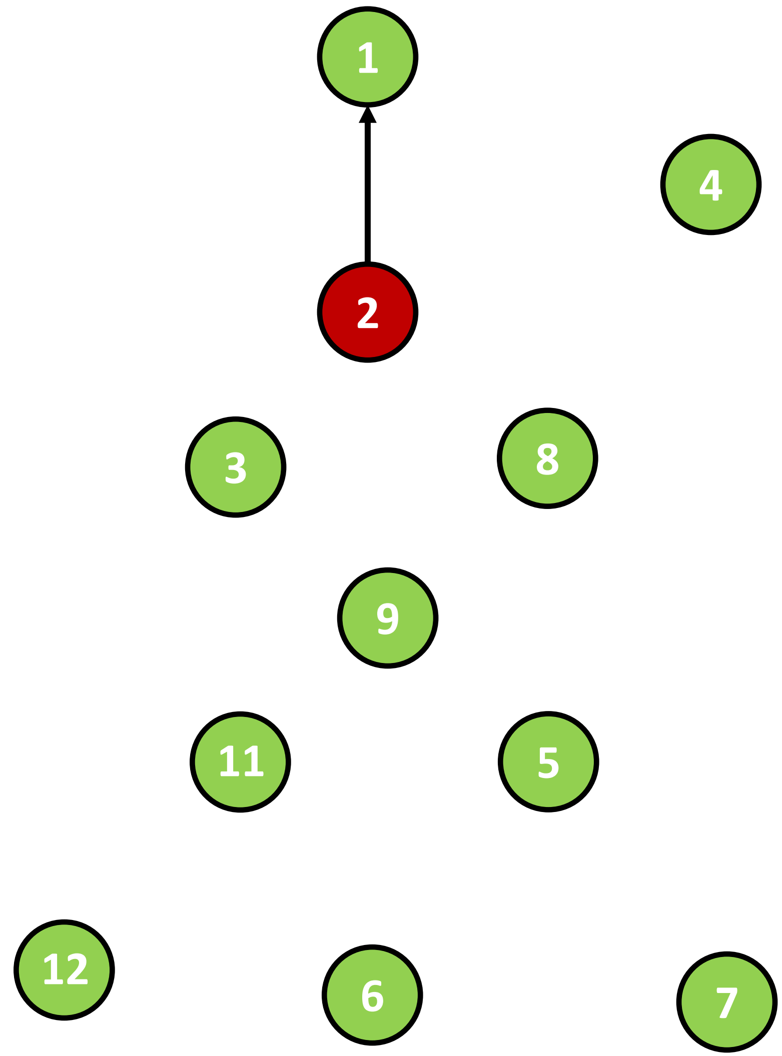
To **union** (merge) two groups together



The union-find algorithm

To **union** (merge) two groups together

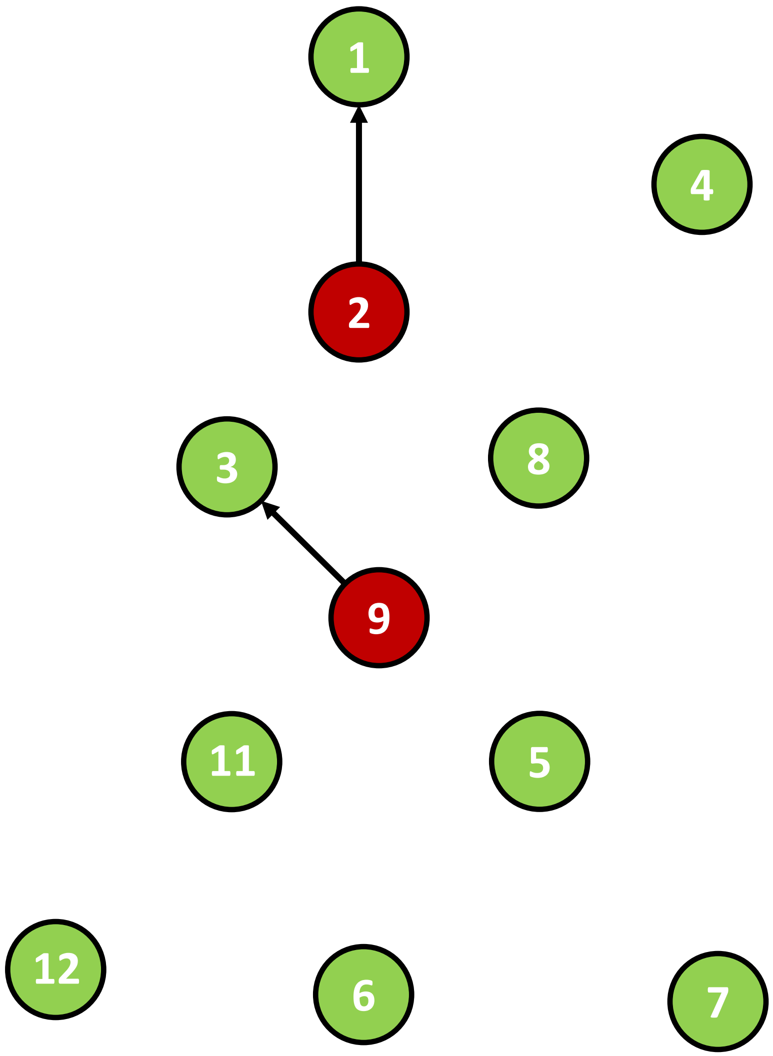
- Just make the **representative of one**
- Be the **parent** of the **representative** of the other



The union-find algorithm

To **union** (merge) two groups together

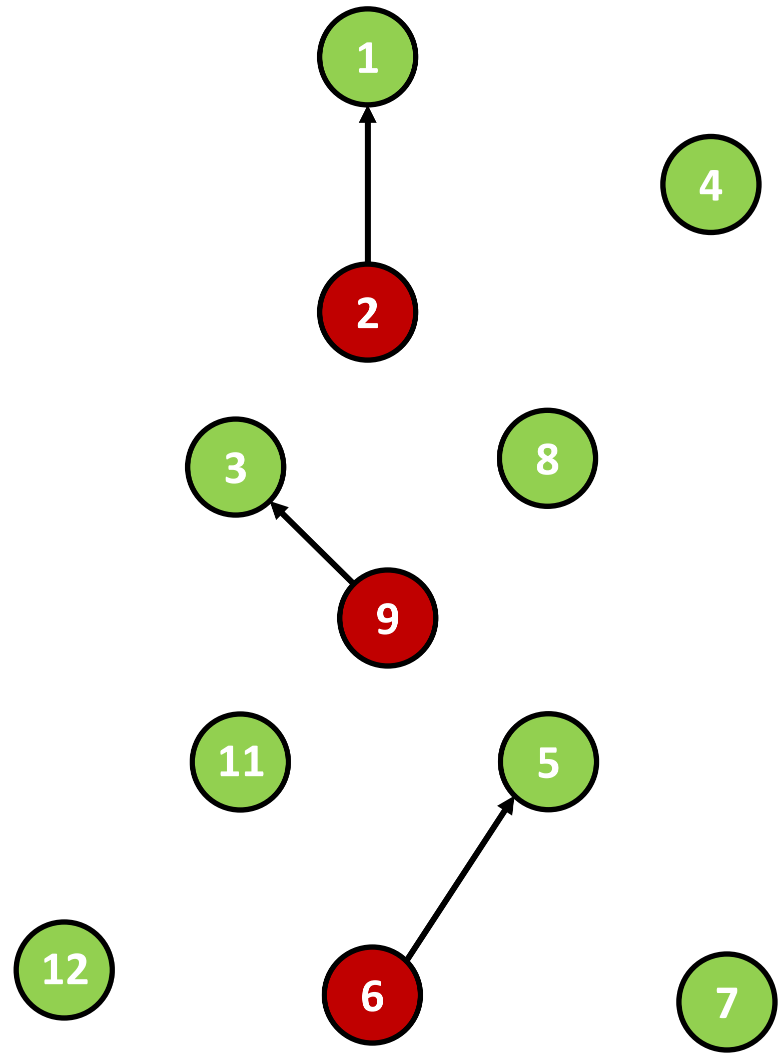
- Just make the **representative of one**
- Be the **parent** of the **representative** of the other



The union-find algorithm

To **union** (merge) two groups together

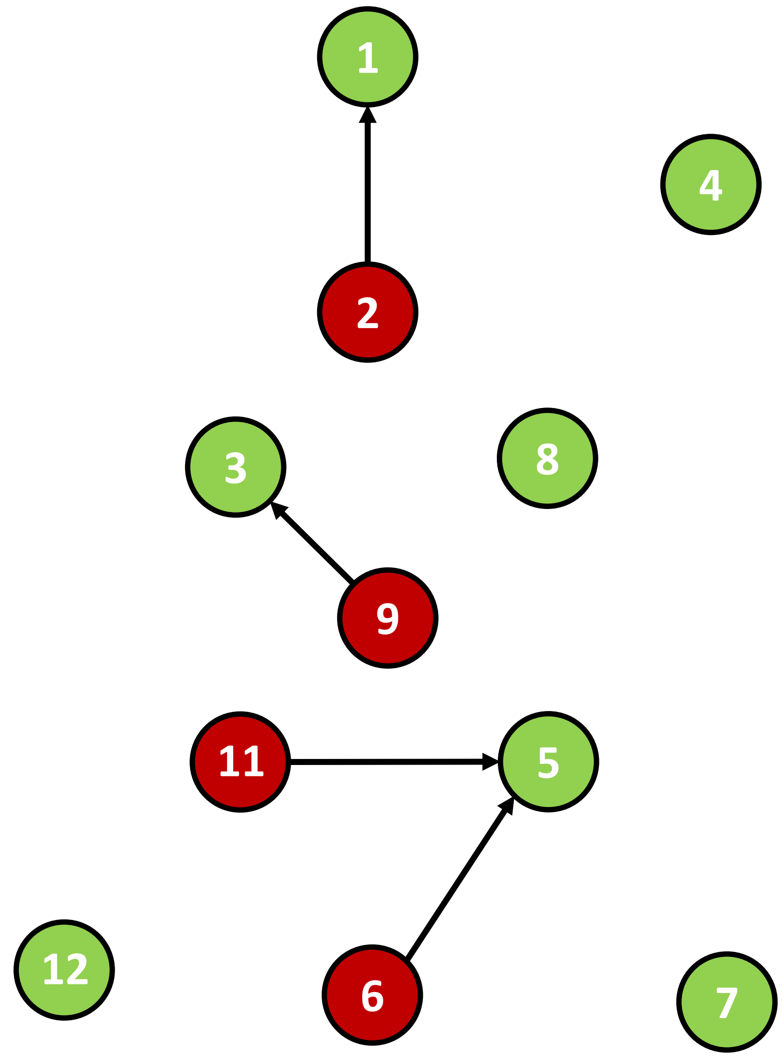
- Just make the **representative of one**
- Be the **parent** of the **representative** of the other



The union-find algorithm

To **union** (merge) two groups together

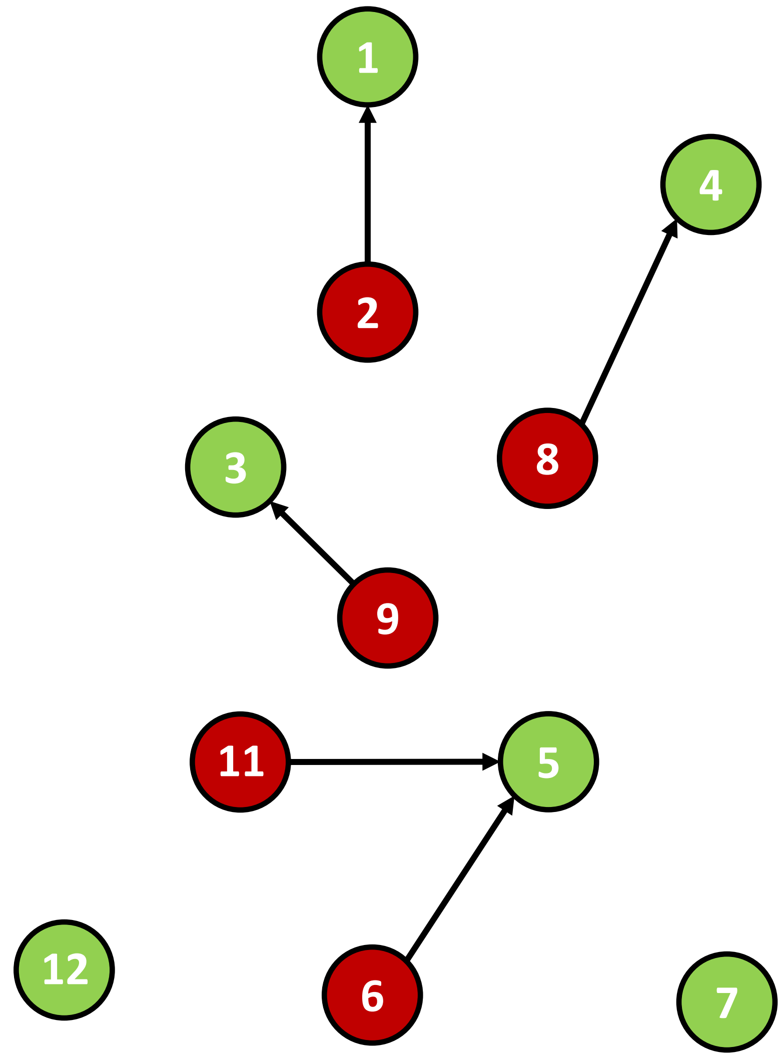
- Just make the **representative of one**
- Be the **parent** of the **representative** of the other



The union-find algorithm

To **union** (merge) two groups together

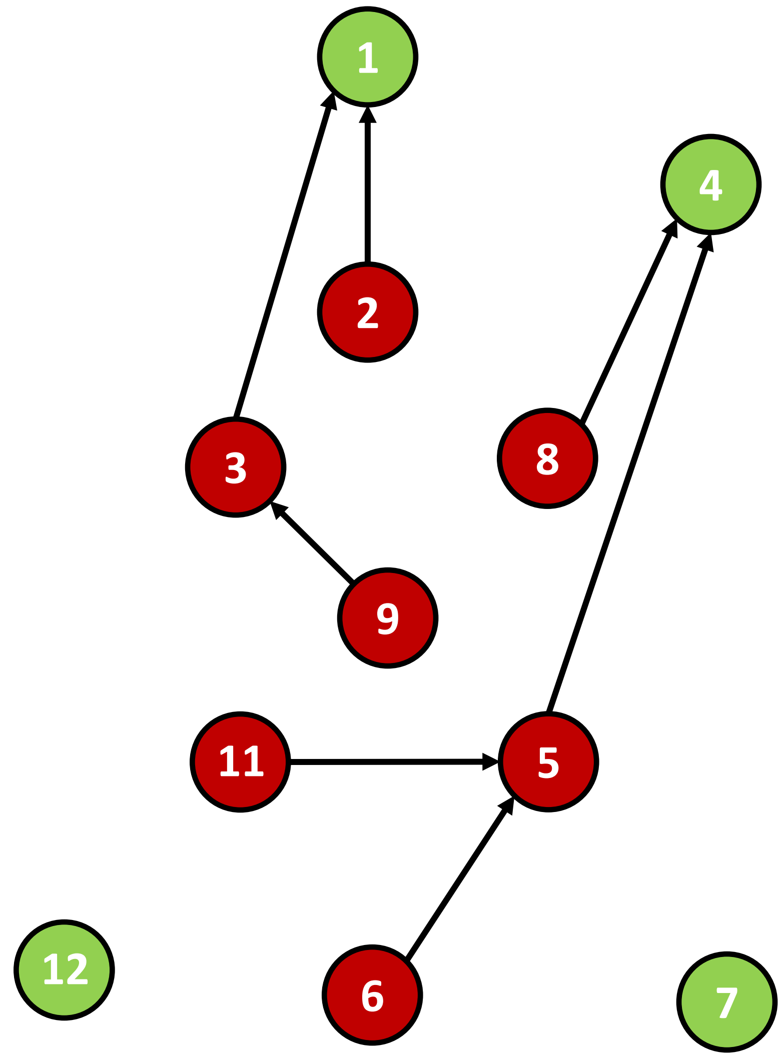
- Just make the **representative of one**
- Be the **parent** of the **representative** of the other



The union-find algorithm

To **union** (merge) two groups together

- Just make the **representative of one**
- Be the **parent** of the **representative** of the other



The union-find algorithm

(slow version)

```
// Find the representative
// for e's group

Find(SetElement e) {
  while (e.parent != null)
    e = e.parent;
  return e;
}
```

```
// Merge the groups of a
// and b together

Union(SetElement a, SetElement b)
{
  ar = Find(a);
  br = Find(b);
  if (ar != br)
    ar.parent = br;
}
```

Complexity analysis

```
// Find the representative
// for e's group
Find(SetElement e) {
  while (e.parent != null)
    e = e.parent;
  return e;
}
```

```
// Merge the groups of a
// and b together
Union(SetElement a, SetElement b)
{
  ar = Find(a);
  br = Find(b);
  if (ar != br)
    ar.parent = br;
}
```

Complexity analysis

// Find the representative
// for e's group

```
Find(SetElement e) {
  while (e.parent != null)
    e = e.parent;
  return e;
}
```

$O(h)$

// Merge the groups of a
// and b together

```
Union(SetElement a, SetElement b)
{
  ar = Find(a);
  br = Find(b);
  if (ar != br)
    ar.parent = br;
}
```

$O(h)$

The worst-case running time of each algorithm is the height of the tree

Complexity analysis

// Find the representative
// for e's group

```
Find(SetElement e) {
  while (e.parent != null)
    e = e.parent;
  return e;
}
```

$O(n)$

// Merge the groups of a
// and b together

```
Union(SetElement a, SetElement b)
{
  ar = Find(a);
  br = Find(b);
  if (ar != br)
    ar.parent = br;
}
```

$O(n)$

But the worst-case height of the tree is the number of nodes :-(

How do we make it better?

How do we make it better?

- **Balance** the tree!
- Keep **track** of the **heights** of the trees
- Always **add the shorter one to the bigger one**

Union by rank (height)

```
class SetElement {
  SetElement parent;
  int rank;
}
```

```
Union(SetElement a, SetElement b)
{
  ar = Find(a);
  br = Find(b);

  if (ar == br) return;

  if (ar.rank < br.rank)
    ar.parent = br;
  else if (ar.rank > br.rank)
    br.parent = ar;
  else {
    br.parent = ar;
    ar.rank = ar.rank + 1;
}
```

Complexity

- Union by rank guarantees the trees are **balanced**
- So their **heights** are $\boldsymbol{O(\log n)}$
- And so the **complexities** are Union and Find are also $O(\log n)$

But it turns out we can actually do **better**

Saving redundant work

- If we call **Find** on the same node **twice**
- We run up the chain twice
- Why not **cache** the original result?

Path compression

- Just **update** the **parent** of the element
- To point directly at the **representative**
- And while you're at it, update **all the nodes on the path** to the representative too!

```
Find(SetElement a) {
  if (a.parent == null)
    return a;
  a.parent = Find(a.parent);
  return a.parent;
}
```

Complexity analysis

- The bad news is that Find is **still $O(\log n)$** in the **worst case**

- So what's the good news?

Amortized complexity analysis

- It's only $O(\log n)$ the **first time** you do a lookup on a node
- After that, it's **constant time**!
  - At least **until** you do **another merge**

- Tarjan proved that the **amortized complexity** of both union and find is $\boldsymbol{O(\alpha(n))}$
- Great! **What's $\boldsymbol{\alpha(n, n)}$**?

The Ackermann function

There's this function called the **Ackermann function**:

$$A(m,n) = \begin{cases} n+1, & \text{if } m = 0 \\ A(m-1,1), & \text{if } m > 0 \text{ and } n = 0 \\ A\big(m-1, A(m,n-1)\big), & \text{otherwise} \end{cases}$$

The Ackermann function

Here's Wikipedia's table of **values** for Ackermann's function:

Values of *A*(*m*, *n*)

*m**n*	0	1	2	3	4	*n*
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=2+(n+3)-3$
2	3	5	7	9	11	$2n+3=2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^2}-3$	65533 $=2^{2^{2^2}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^2}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^2}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^2}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3}-3$

It grows really really **fast**.

The inverse Ackermann function

- $\alpha(n)$ is the **inverse** of $A(n, n)$
- It grows really really **slowly**

Values of $A(m, n)$

$m \backslash n$	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2 = 2 + (n+3) - 3$
2	3	5	7	9	11	$2n+3 = 2 \cdot (n+3) - 3$
3	5	13	29	61	125	$2^{(n+3)} - 3$
4	13 $=2^{2^2} - 3$	65533 $=2^{2^{2^2}} - 3$	$2^{65536} - 3$ $=2^{2^{2^{2^2}}} - 3$	$2^{2^{65536}} - 3$ $=2^{2^{2^{2^{2^2}}}} - 3$	$2^{2^{2^{65536}}} - 3$ $=2^{2^{2^{2^{2^{2^2}}}}} - 3$	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3} - 3$

The inverse Ackermann function

- It grows **comically slowly**
- $\boldsymbol{\alpha}(\text{the number of subatomic particles in the universe}) < 4$

Values of $A(m, n)$

$m \backslash n$	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=2+(n+3)-3$
2	3	5	7	9	11	$2n+3=2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^2}-3$	65533 $=2^{2^{2^2}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^2}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^2}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^2}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3}-3$

Estimated number of subatomic particles $\cong 10^{80} \cong 2^{265}$

The inverse Ackermann function

- Did I mention **it grows slowly**?

Values of *A*(*m*, *n*)

*m**n*	0	1	2	3	4	*n*
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=2+(n+3)-3$
2	3	5	7	9	11	$2n+3=2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^{2}}-3$	65533 $=2^{2^{2^{2}}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^{2}}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^{2}}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^{2}}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3}-3$

The inverse Ackermann function

- So slowly that it **might as well be constant**

Values of *A*(*m*, *n*)

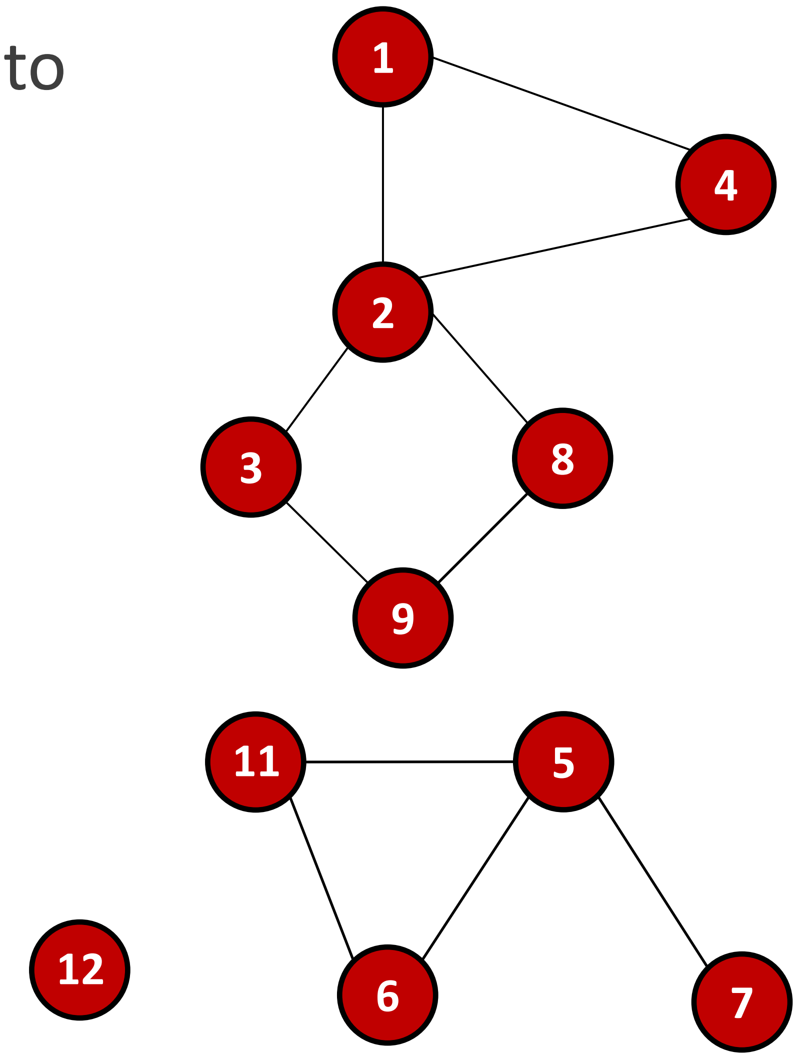
*m**n*	0	1	2	3	4	*n*
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=2+(n+3)-3$
2	3	5	7	9	11	$2n+3=2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^2}-3$	65533 $=2^{2^{2^2}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^2}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^2}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^2}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3}-3$

Amortized complexity analysis

- Tarjan proved that the **amortized complexity** of both union and find is $O(\alpha(n))$
- So for all practical purposes, it's $\boldsymbol{O(1)}$
- That's pretty **cool**

Incremental connected components

- Add parent and rank fields to each node
- For each edge (u,v)
  Union(u, v)
- To test whether two nodes are connected check
  Find(u) == Find(v)



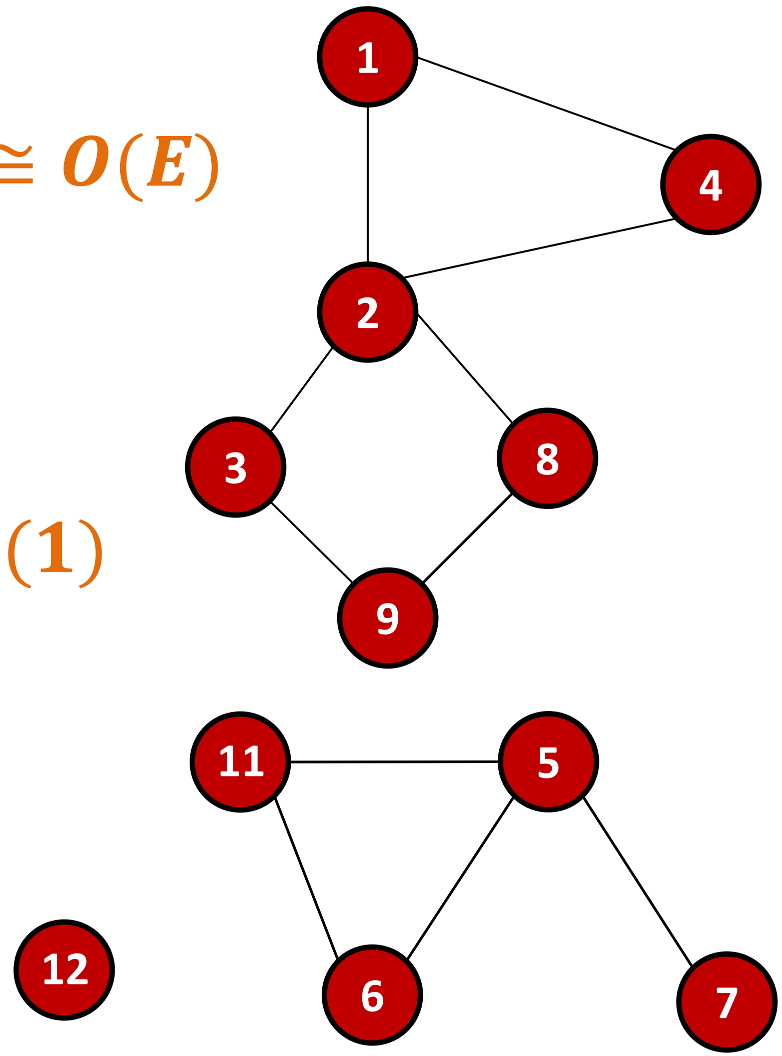
Amortized complexity

- For each edge (u,v)

  Union(u, v) $\boldsymbol{O(\alpha(V)E) \cong O(E)}$
- To test whether two nodes are connected check

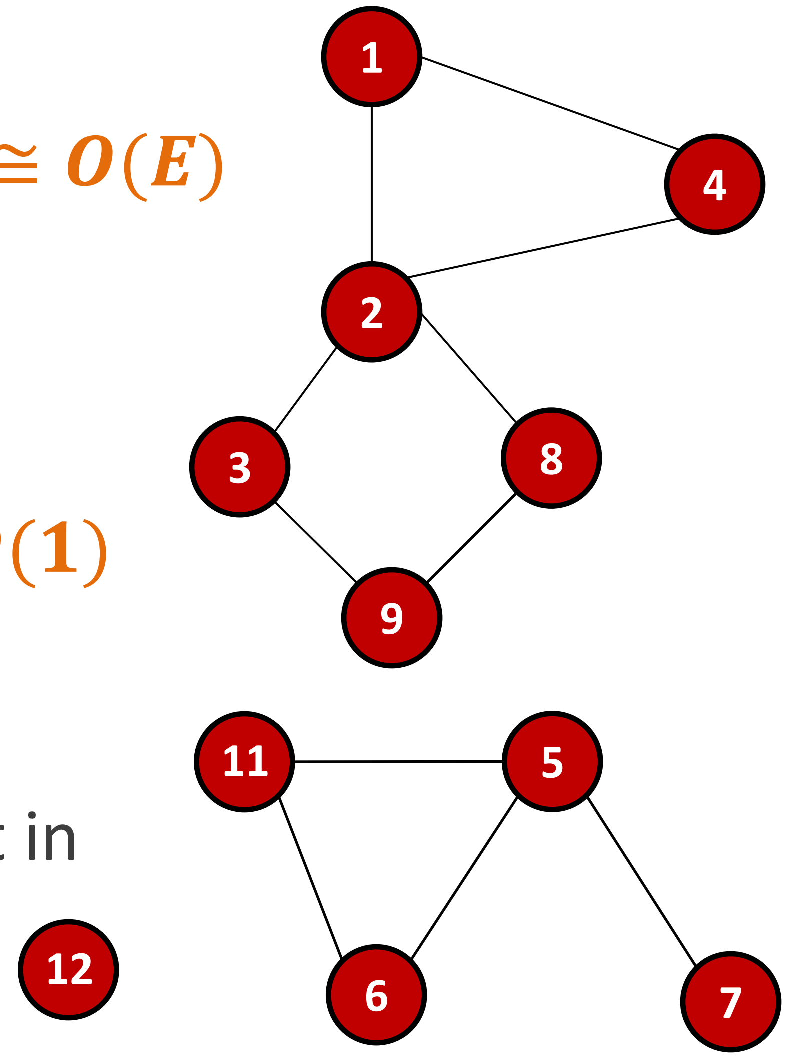
  Find(u) == Find(v)

  $\boldsymbol{O(\alpha(V)) \cong O(1)}$



Amortized complexity

- For each edge (u,v)
  Union(u, v) $\boldsymbol{O(\alpha(V)E) \cong O(E)}$
- To test whether two nodes are connected check
  Find(u) == Find(v)
  $\boldsymbol{O(\alpha(V)) \cong O(1)}$
- So we can compute connected component **incrementally** and still do it in **linear time**



Oh crap! I don't understand the Ackermann function at all! Do I need to know this for the quiz??!?!?

- No
- You just need to understand
  - The **union-find algorithm** with union by **rank** and **path compression**
  - That it's **effectively $O(1)$** amortized time
  - **Why** you might want to keep track of **partitions** on a set
    - E.g. to compute connected components or equivalence classes

Reading

- Read Chapter 21, sections 1-3