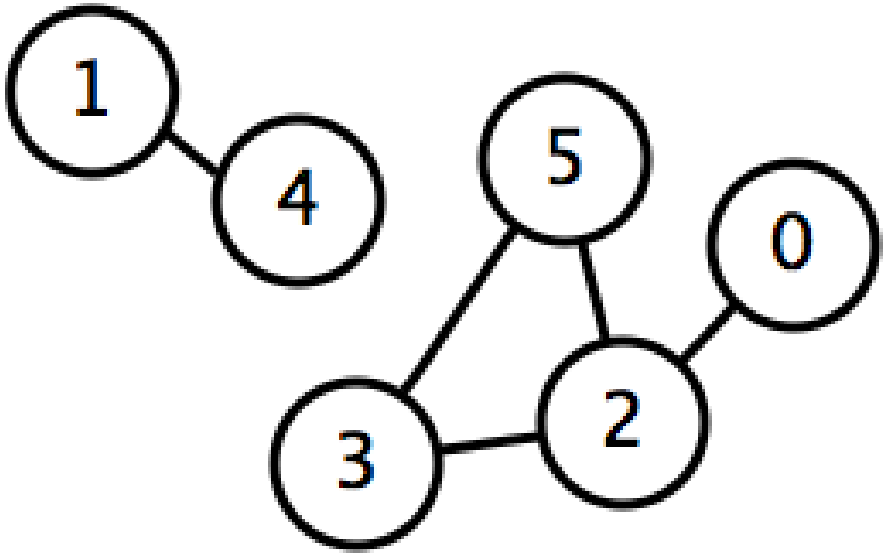
Strongly Connected Components (SCCs) and 2-SAT

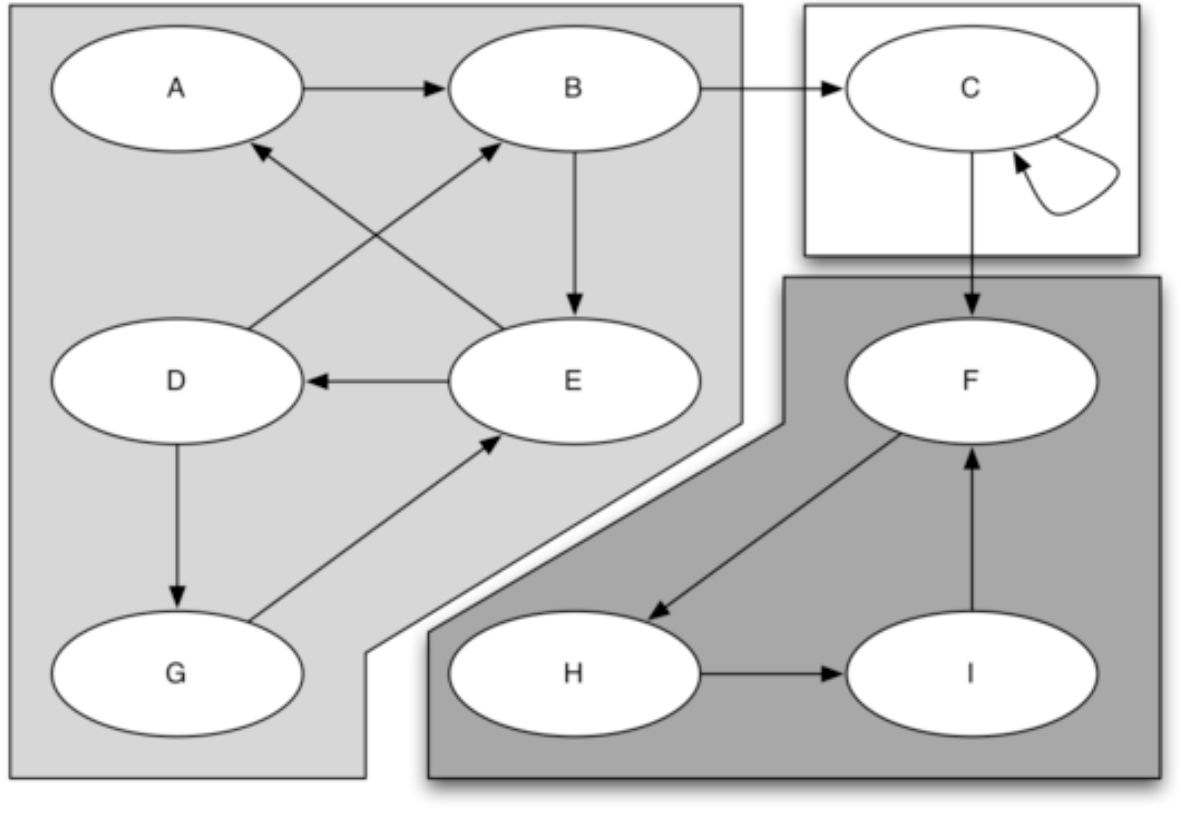
Michael Bradet-Legris

Table of Contents

Strongly Connected Components

- Clause: Each node in the component has a path to every other node in the component
- Connected Components: Undirected Graph
- Strongly Connected Components: Directed Graph





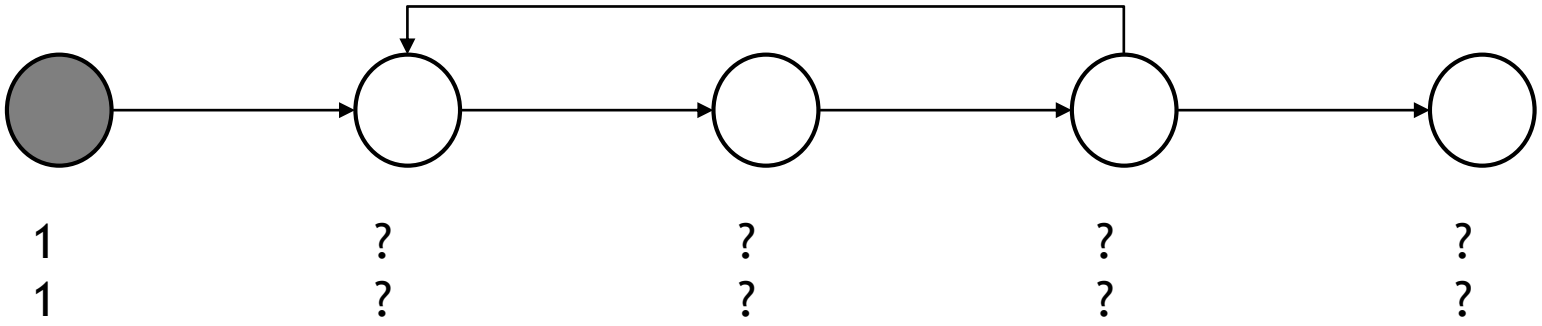
Tarjan's Algorithm: Rundown

- Linear Time algorithm: O(V + E)
- Perform a single modified DFS (may need to do multiple if one DFS doesn't reach all nodes)
- Tarjan(node v): //Basic idea, extremely rudimentary
  - Index[v] = lowIndex[v] = next available index
  - lowIndex[v] = Min(index[v], lowIndex of all adjacent nodes not in a SCC)
    - If lowIndex is undefined for an adjacent node, call Tarjan on that node
  - If Index[v] = lowIndex[v], found a SCC.
    - The SCC consists of all nodes u with lowIndex[u] = lowIndex[v]

- *Kosaraju's Algorithm: Another linear solution, but more complex to code and more overhead, and (probably) higher constant factor for complexity.

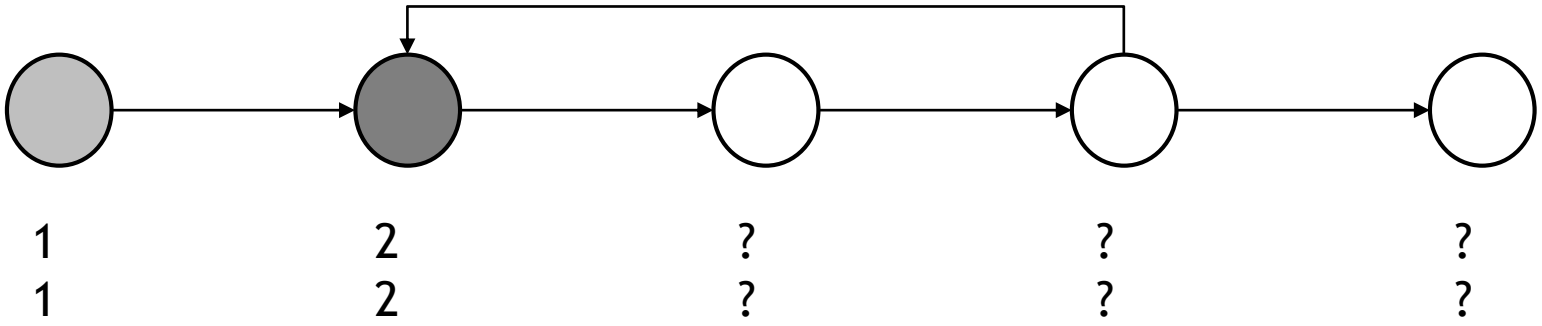
Tarjan's Algorithm: Example 1

- Top Value: Index
- Bottom Value: LowIndex
- Adjacent Node has undefined lowIndex. Recurse.



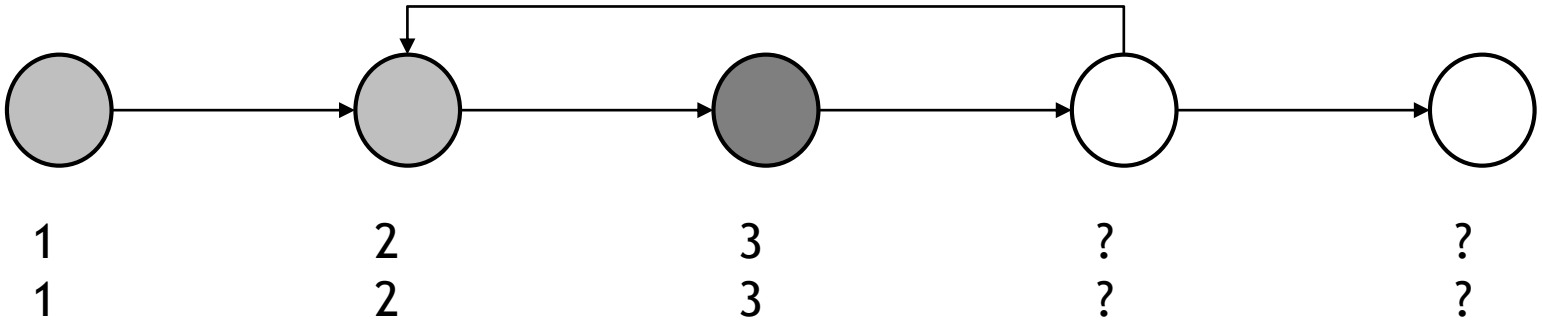
Tarjan's Algorithm: Example 1

- Adjacent Node has undefined lowIndex. Recurse.



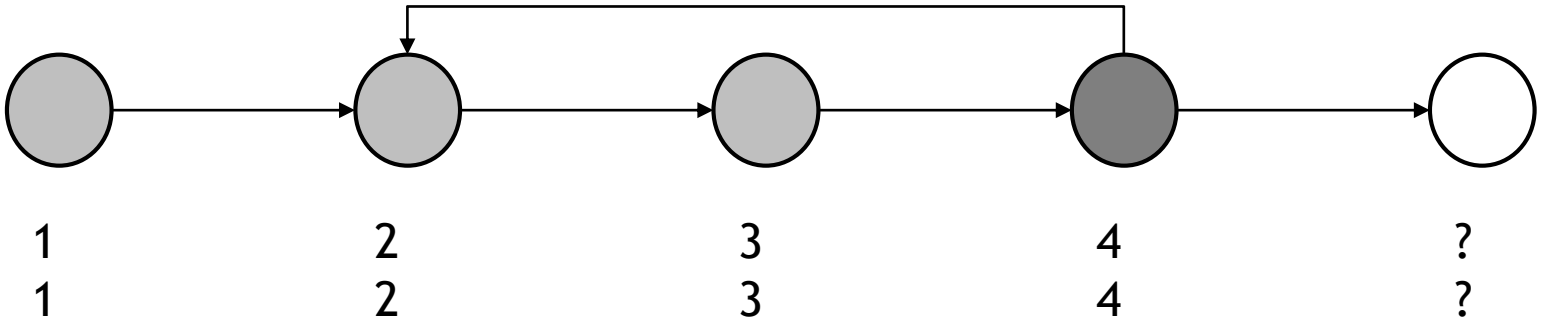
Tarjan's Algorithm: Example 1

- Adjacent Node has undefined lowIndex. Recurse.



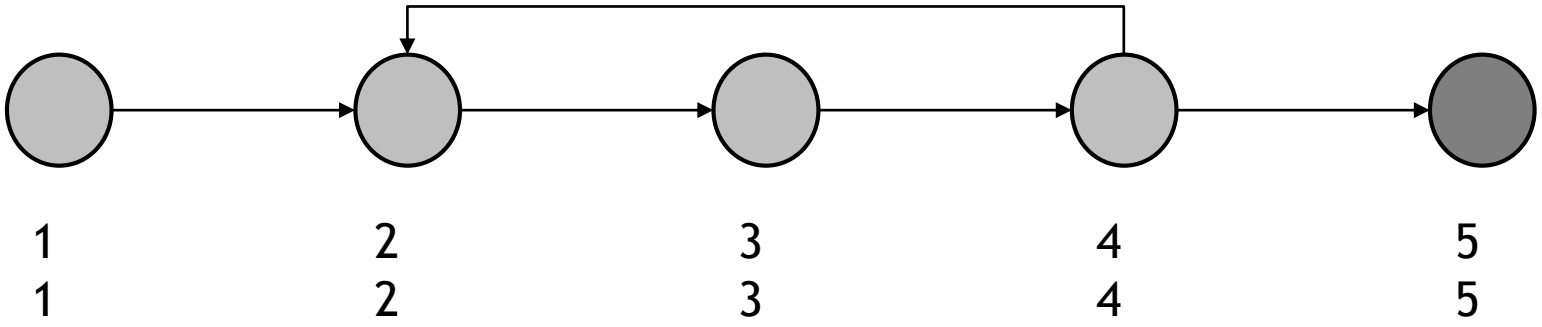
Tarjan's Algorithm: Example 1

- Adjacent Node has undefined lowIndex. Recurse.



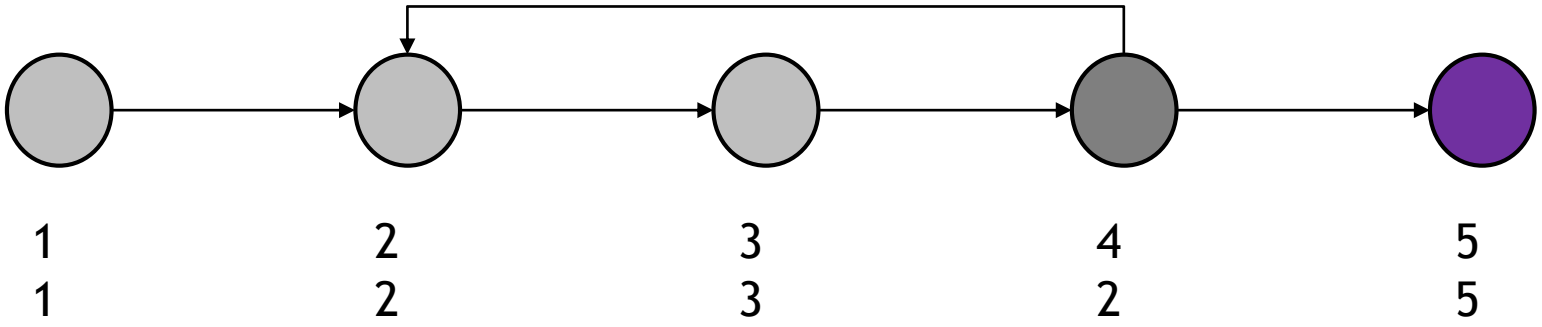
Tarjan's Algorithm: Example 1

- We have lowIndex = Index = 5 so create the SCC with all nodes with lowIndex = 5, then recurse back.



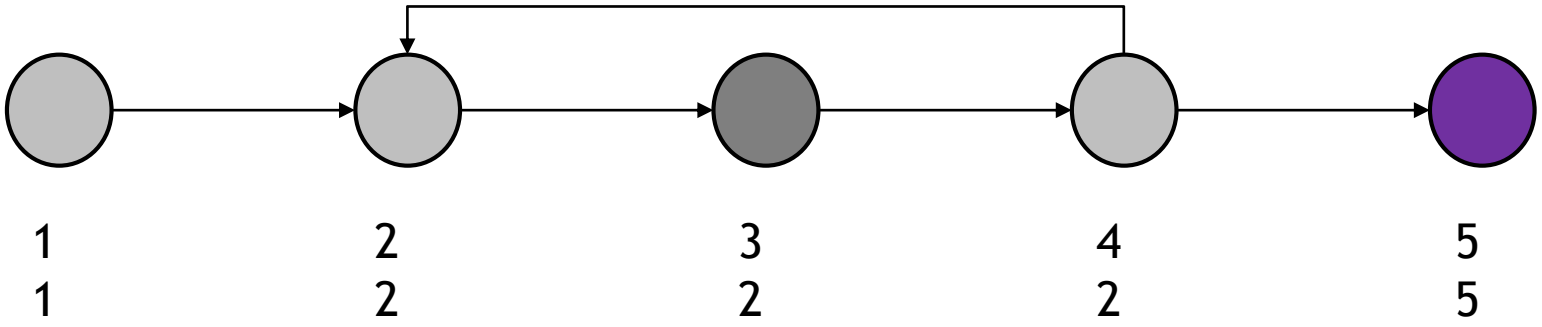
Tarjan's Algorithm: Example 1

- Created one SCC (purple)
- Update lowIndex of current Node, then recurse back.



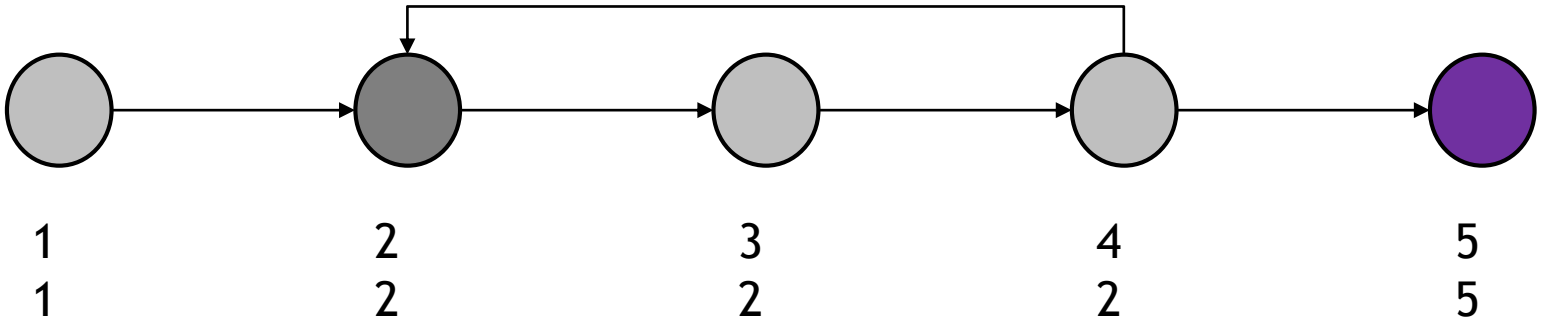
Tarjan's Algorithm: Example 1

- Update lowIndex of current Node, then recurse back.



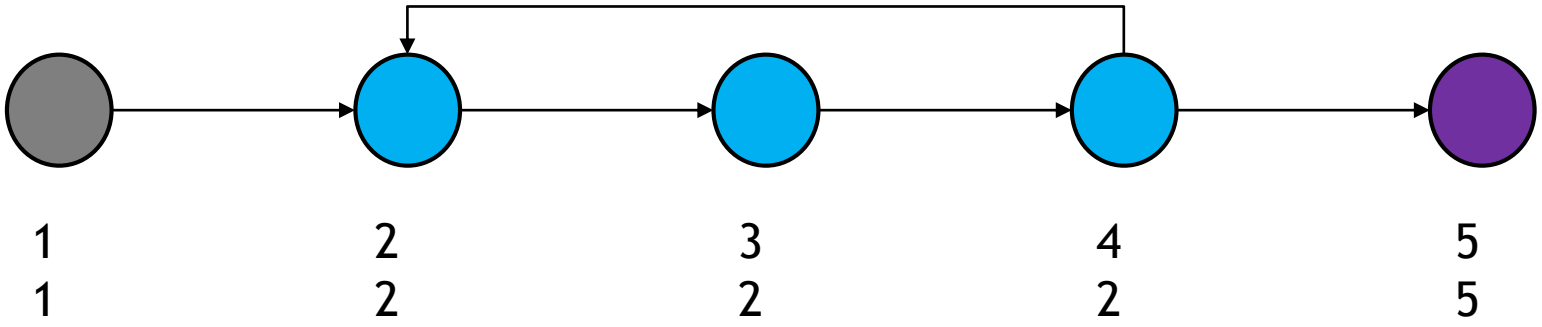
Tarjan's Algorithm: Example 1

- We have lowIndex = Index = 2 so create the SCC with all nodes with lowIndex = 2, then recurse back.



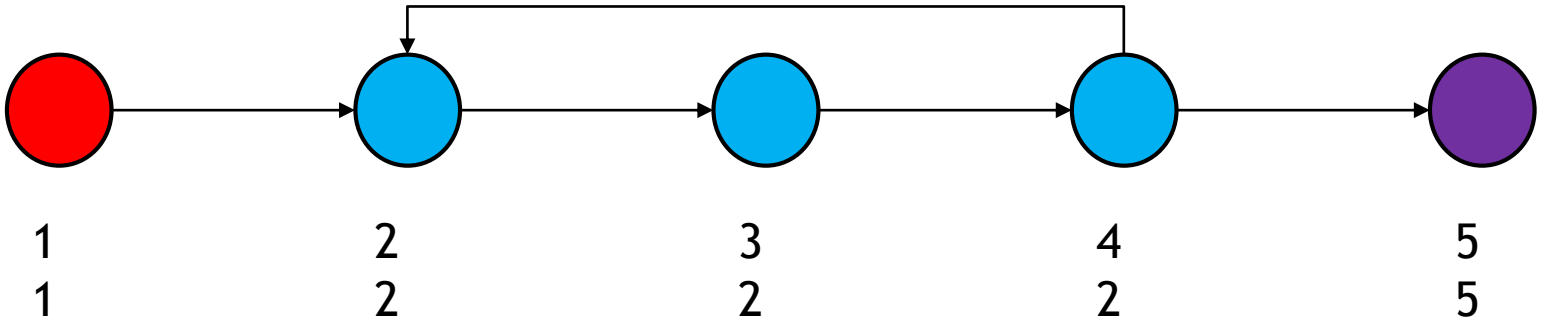
Tarjan's Algorithm: Example 1

- Created one SSC (blue)
- We have lowIndex = Index = 1 so create the SCC with all nodes with lowIndex = 1, then recurse back.



Tarjan's Algorithm: Example 1

- Created one SCC (red)
- Algorithm terminates. We now have 3 SCCs.



Tarjan's Algorithm: Pseudocode

```
algorithm tarjan is
 input: graph G = (V, E)
 output: set of strongly connected components (sets of vertices)

 index := 0
 S := empty array
 for each v in V do
   if (v.index is undefined) then
     strongconnect(v)
   end if
 end for
```

Tarjan's Algorithm: Pseudocode

```
function strongconnect(v)
  // Set the depth index for v to the smallest unused index
  v.index := index
  v.lowlink := index
  index := index + 1
  S.push(v)
  v.onStack := true
```

Tarjan's Algorithm: Pseudocode

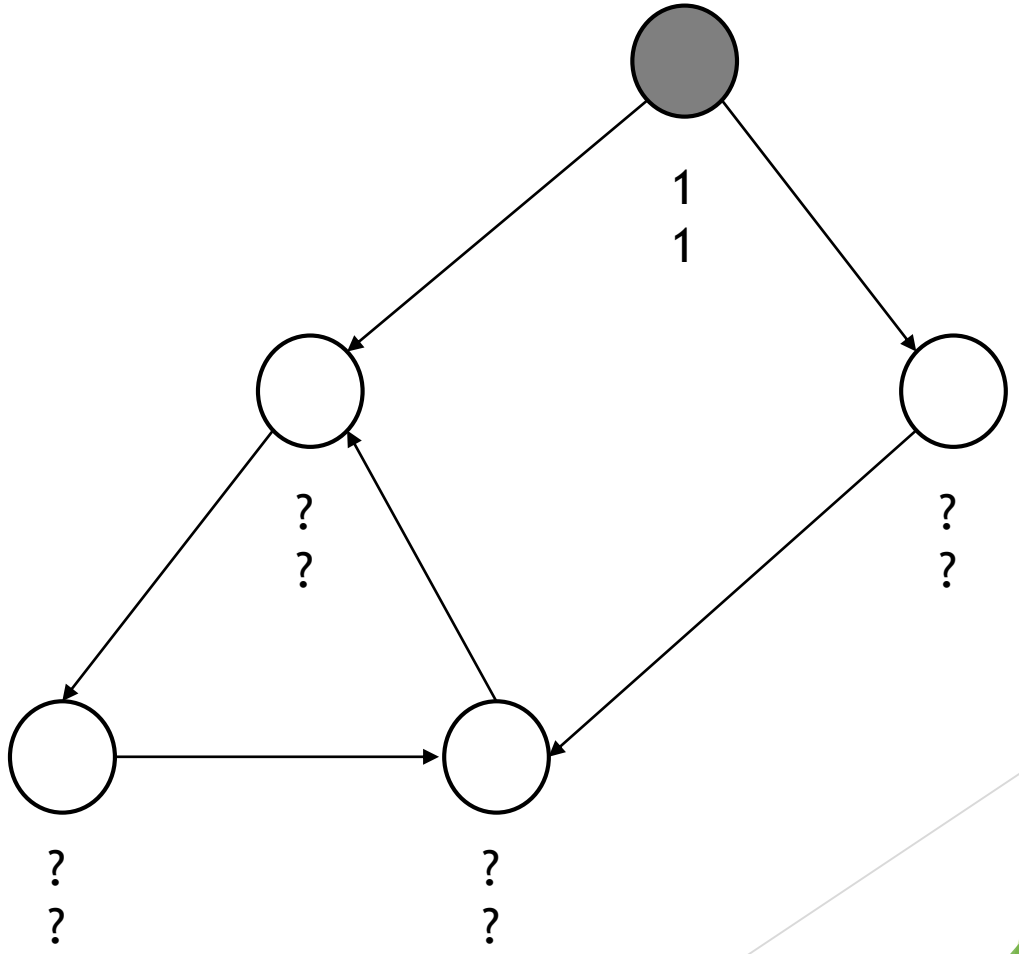
```
// Consider successors of v
for each (v, w) in E do
  if (w.index is undefined) then
    // Successor w has not yet been visited; recurse on it
    strongconnect(w)
    v.lowlink  := min(v.lowlink, w.lowlink)
  else if (w.onStack) then
    // Successor w is in stack S and hence in the current SCC
    v.lowlink  := min(v.lowlink, w.lowlink)
  end if
end for
```

Tarjan's Algorithm: Pseudocode

```
  // If v is a root node, pop the stack and generate an SCC
  if (v.lowlink = v.index) then
    start a new strongly connected component
    repeat
      w := S.pop()
      w.onStack := false
      add w to current strongly connected component
    while (w != v)
    output the current strongly connected component
  end if
end function
```

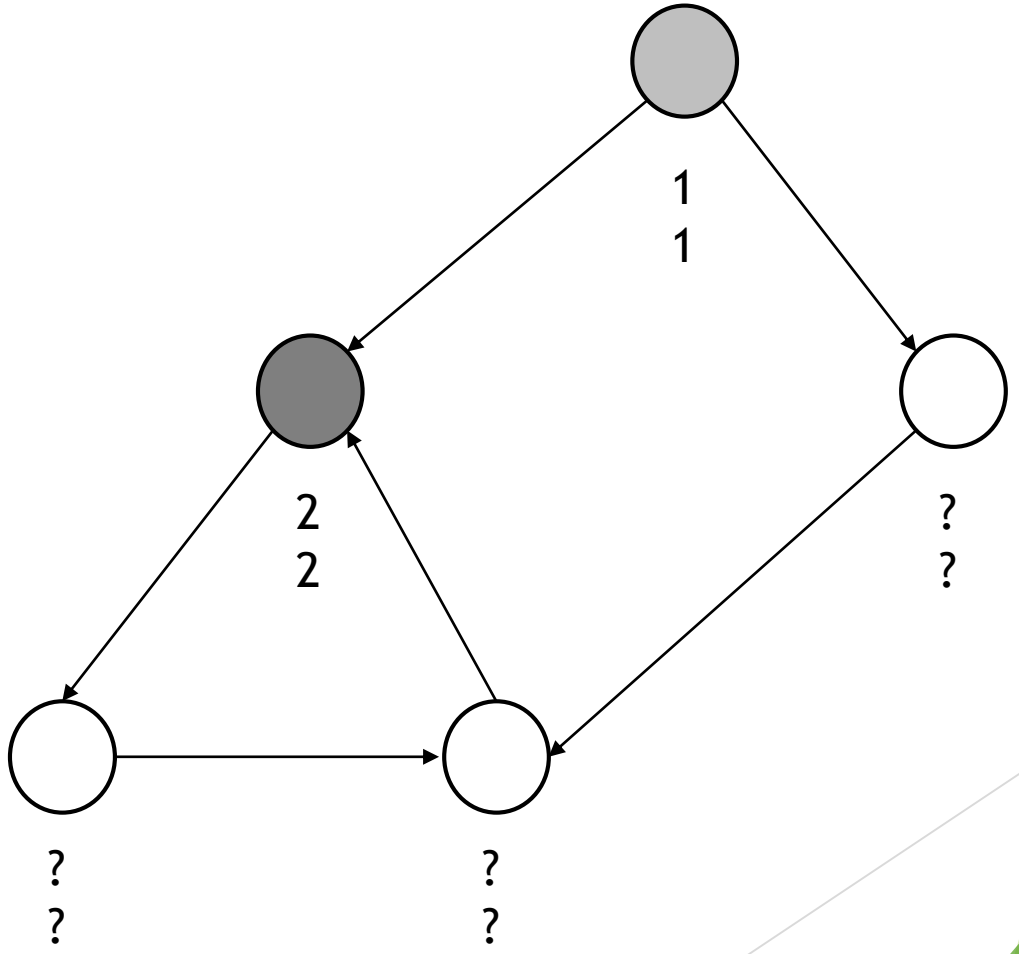
Tarjan's Algorithm: Example 2

- Top Value: Index
- Bottom Value: LowIndex
- Adjacent Node has undefined lowIndex. Recurse.



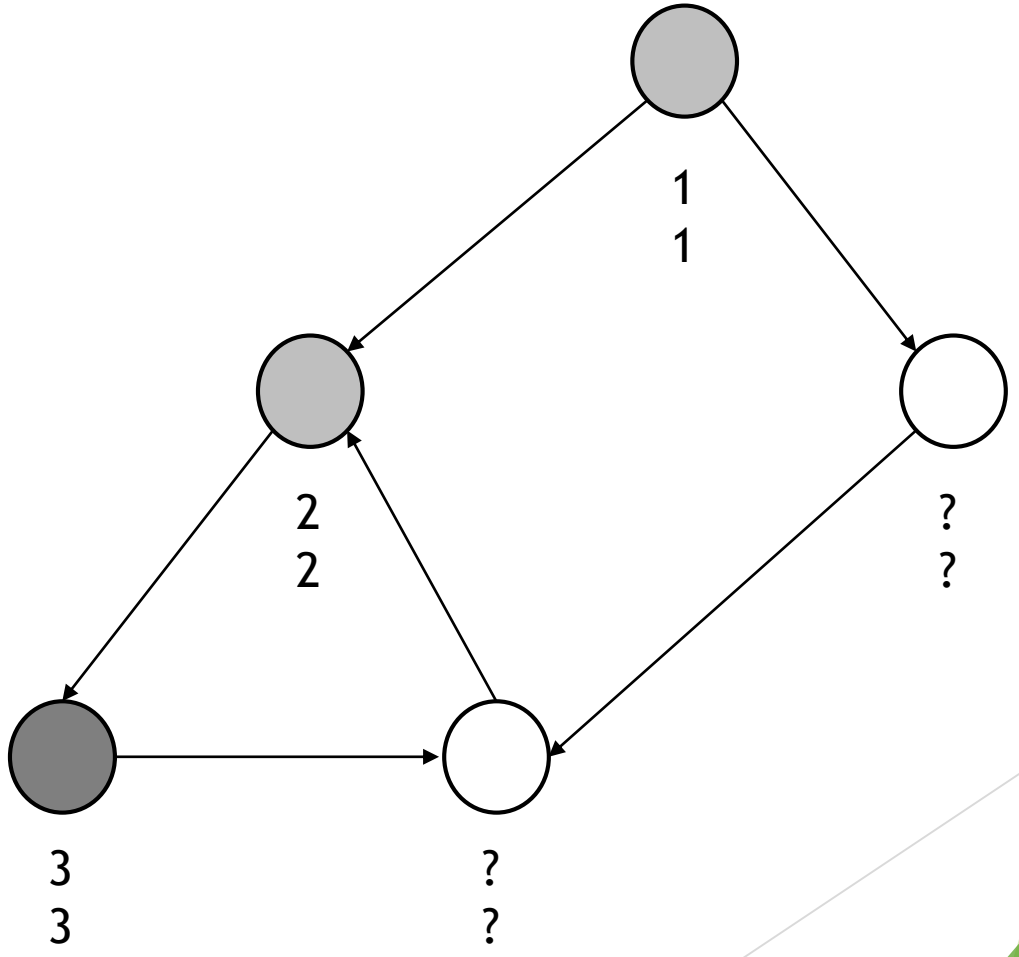
Tarjan's Algorithm: Example 2

- Adjacent Node has undefined lowIndex. Recurse.



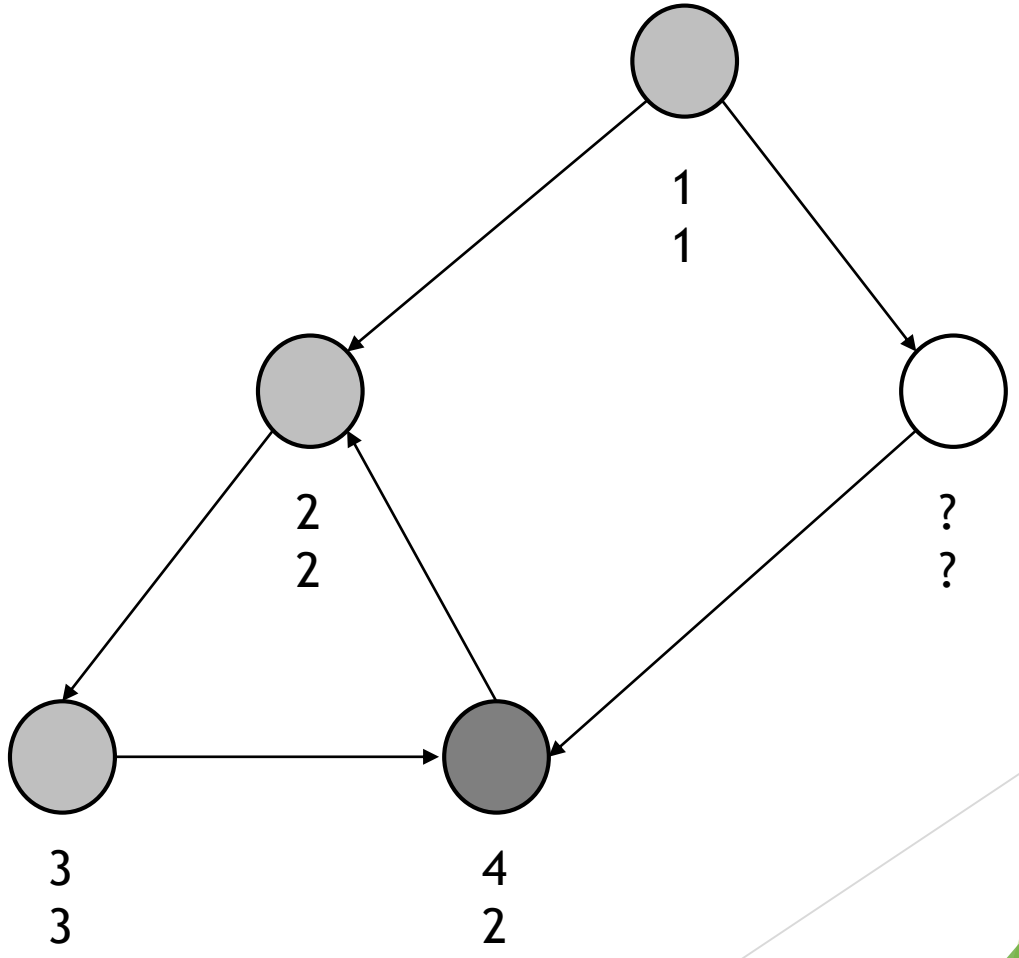
Tarjan's Algorithm: Example 2

- Adjacent Node has undefined lowIndex. Recurse.



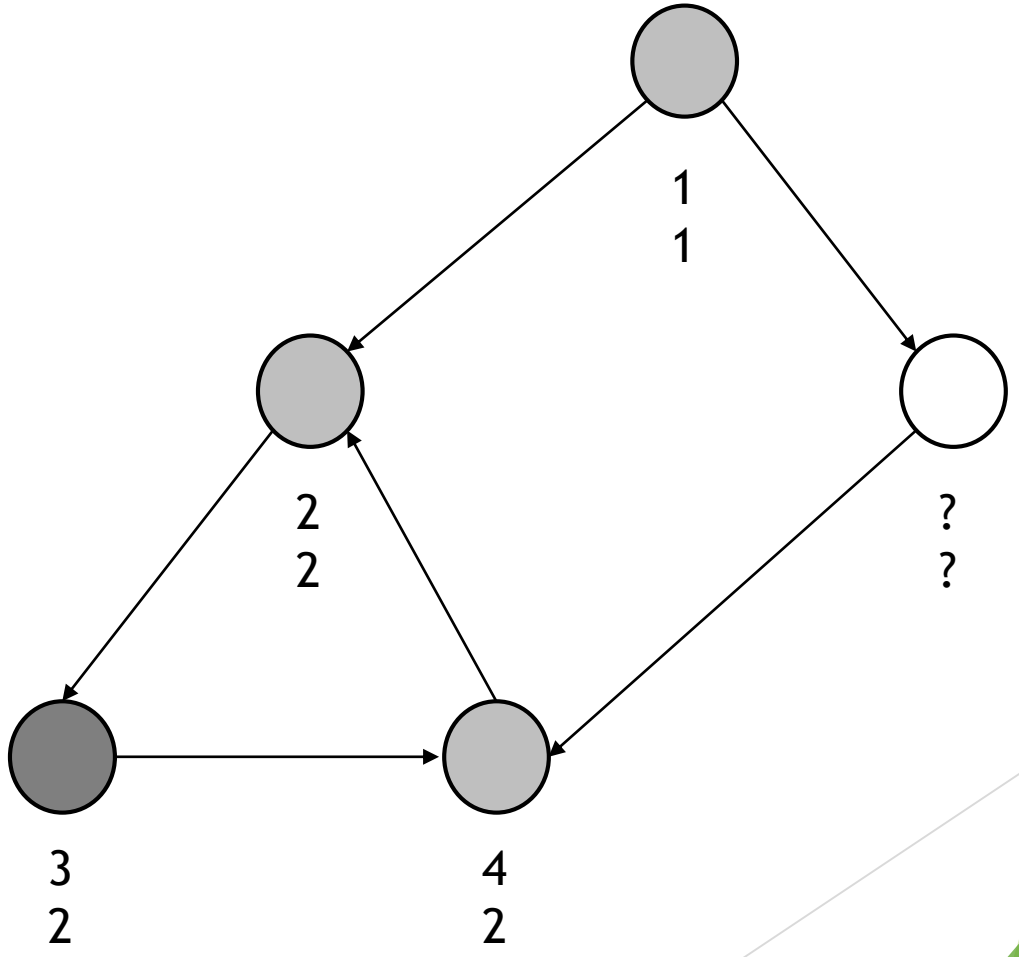
Tarjan's Algorithm: Example 2

- No adjacent nodes that have not been processed.
- Set LowIndex of current node to 2 since it is the minimum of 4 and the lowIndex of adjacent nodes.
- Recurse back.



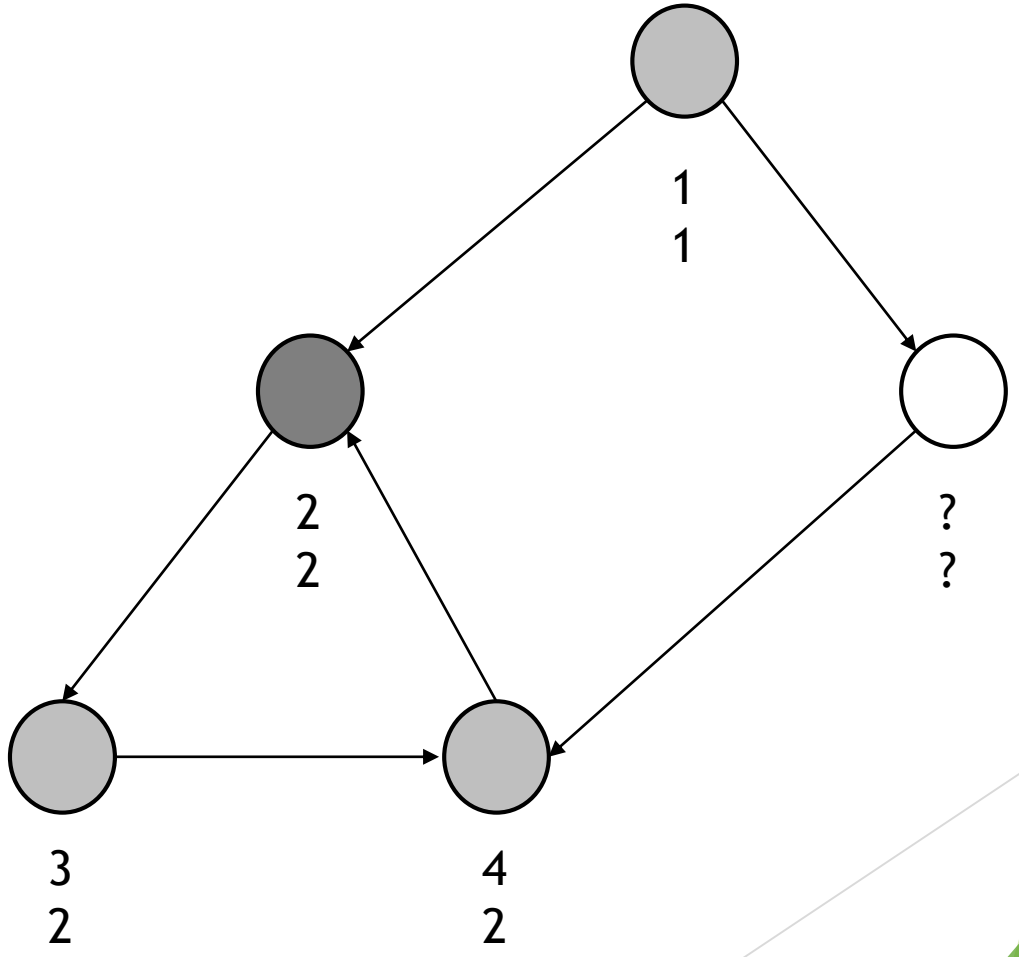
Tarjan's Algorithm: Example 2

- No adjacent nodes that have not been processed.
- Set LowIndex of current node to 2 since it is the minimum of 3 and the lowIndex of adjacent nodes.
- Recurse back.



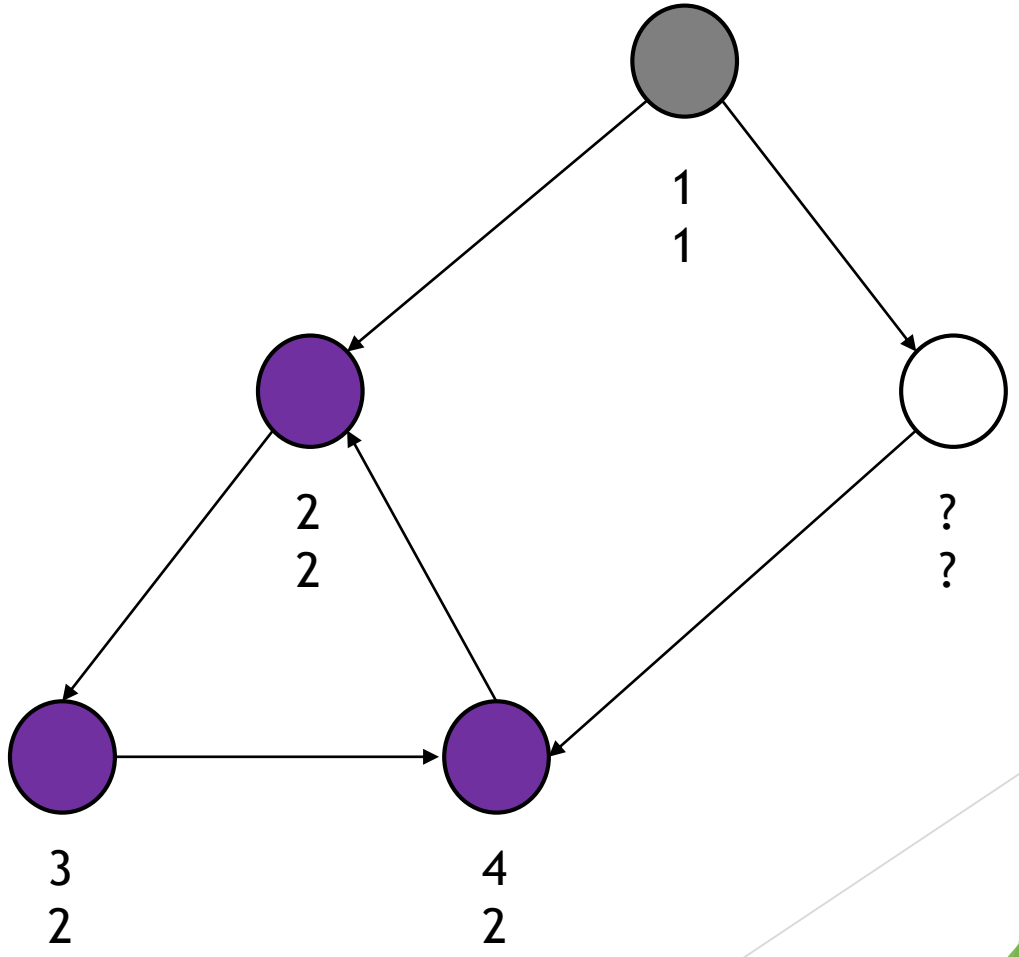
Tarjan's Algorithm: Example 2

- No adjacent nodes that have not been processed.
- LowIndex is the lowest of all adjacent nodes.
- Have LowIndex = Index = 2. Found a SCC. Create the SCC with all nodes having lowIndex = 2.
- Recurse back.



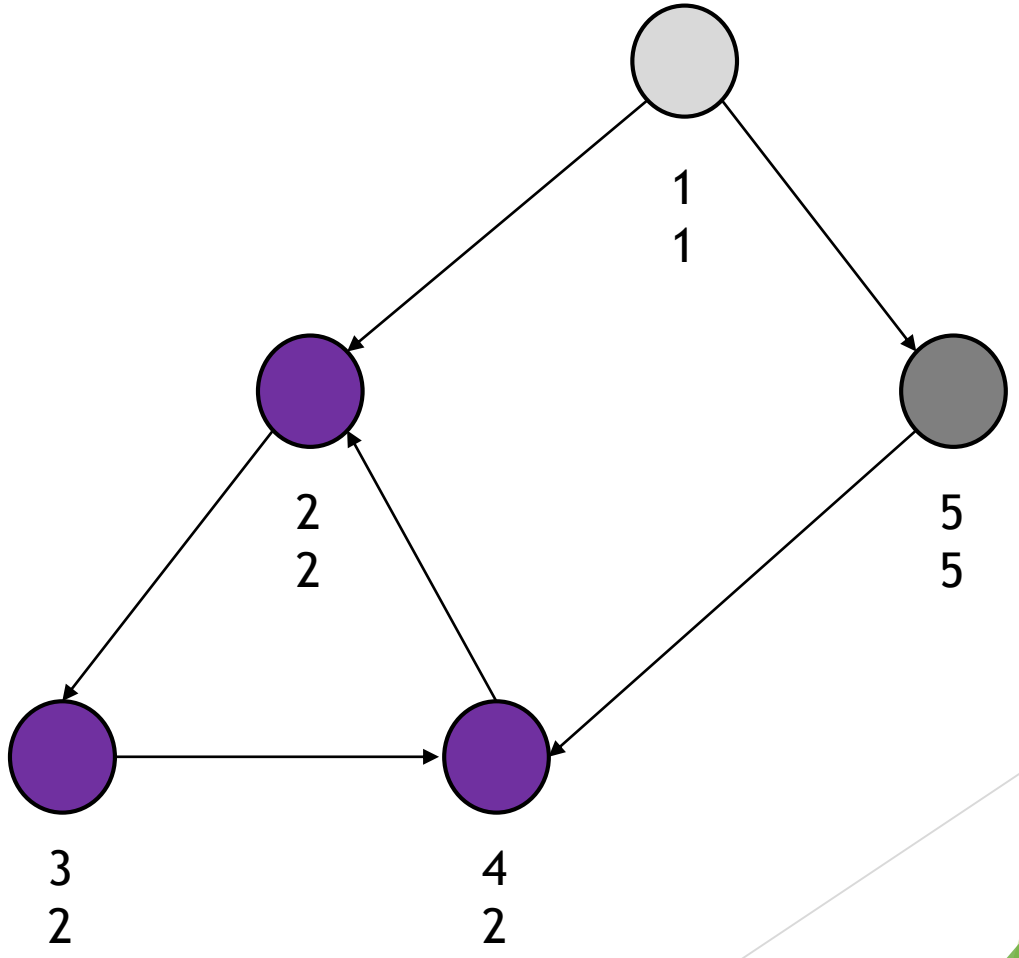
Tarjan's Algorithm: Example 2

- Created one SCC (purple)
- Adjacent Node has undefined lowIndex. Recurse.



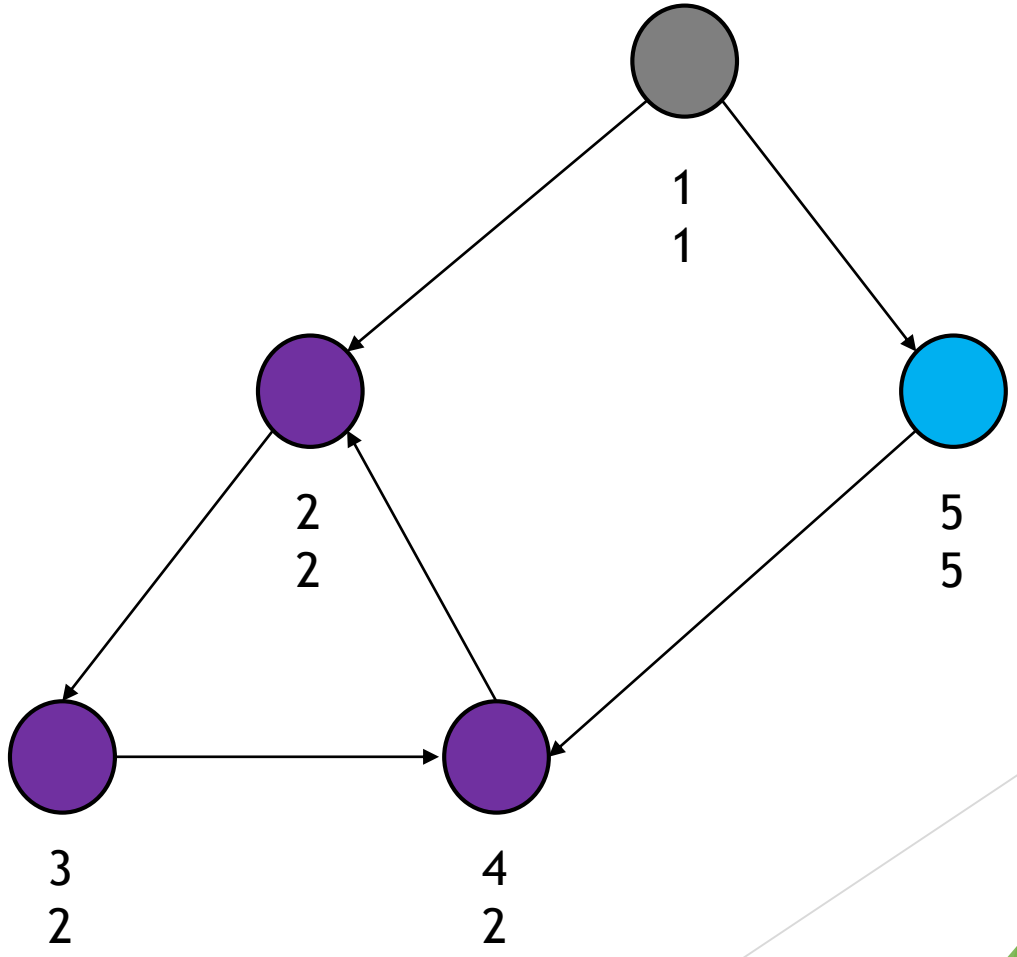
Tarjan's Algorithm: Example 2

- Note here that we cannot make lowIndex of the current node 2, since the adjacent purple node with index 4 is already in an SCC. If node 5 was in the purple SCC, then it would have been found in the DFS of one of the purple nodes, but it wasn't.
- Have LowIndex = Index = 5. Found a SCC. Create the SCC with all nodes having lowIndex = 5.
- Recurse back.



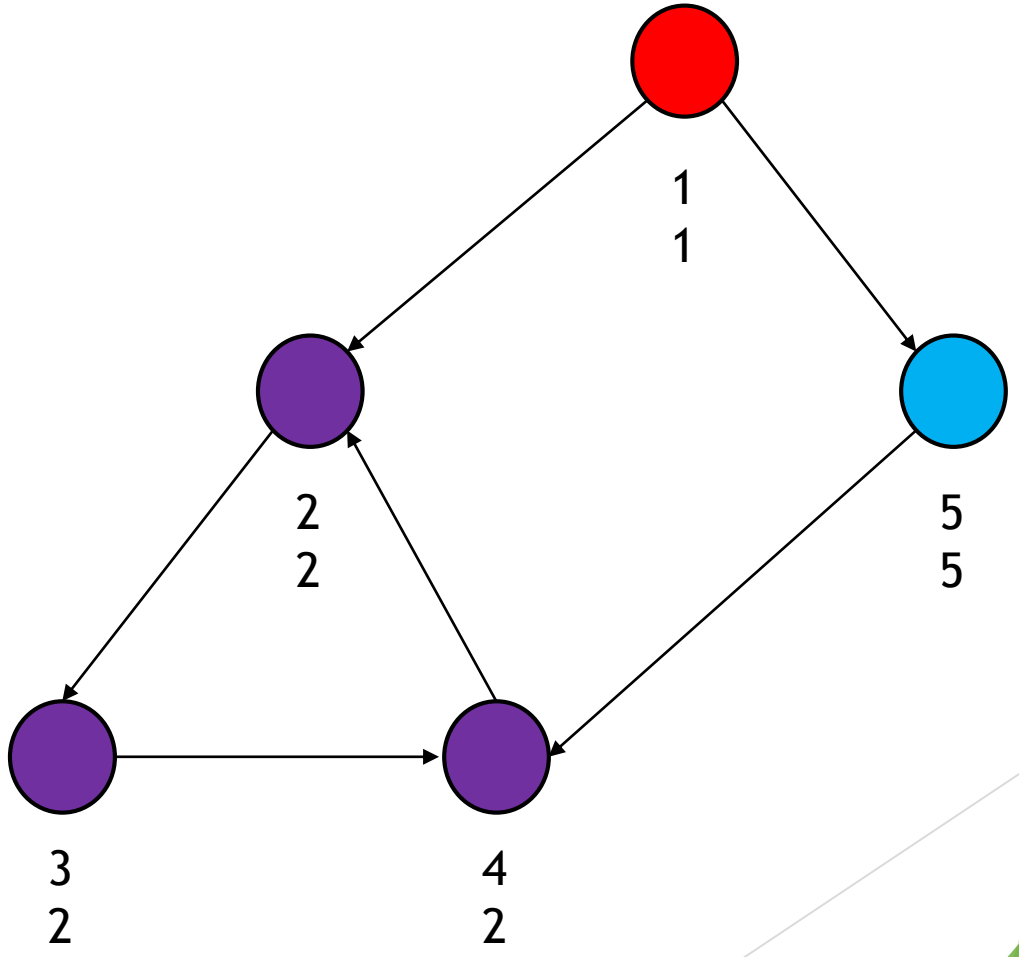
Tarjan's Algorithm: Example 2

- Created one SCC (blue)
- LowIndex is the lowest of all adjacent nodes.
- Have LowIndex = Index = 1. Found a SCC. Create the SCC with all nodes having lowIndex = 1.

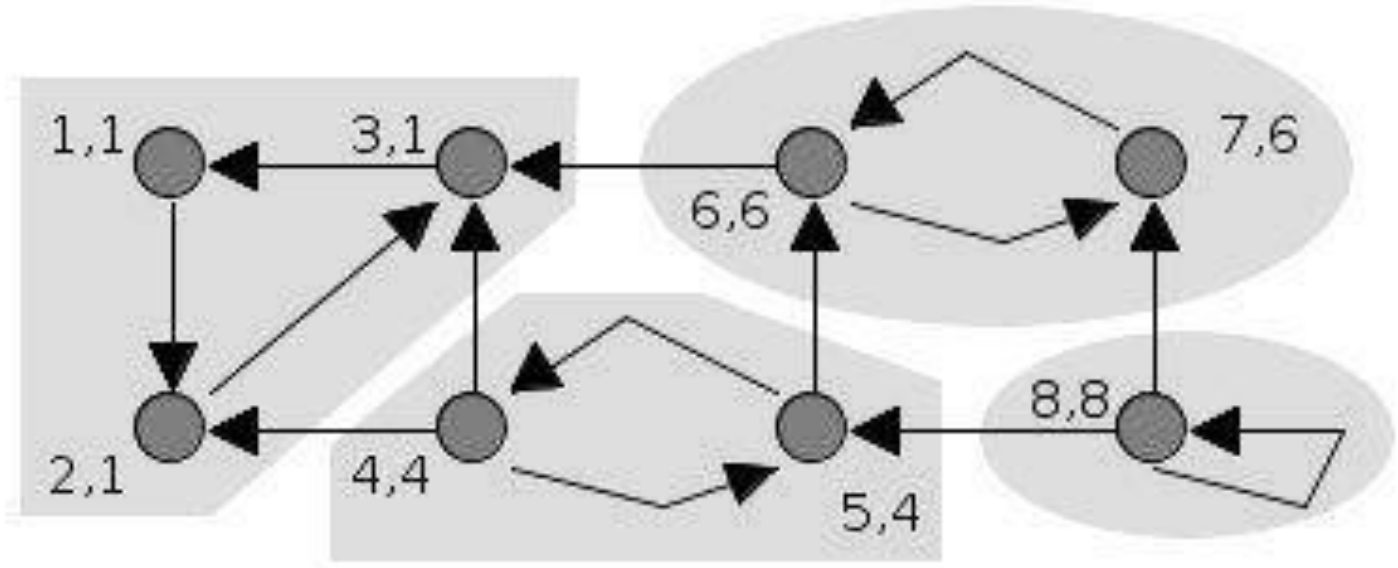


Tarjan's Algorithm: Example 2

- Created one SCC (red)
- Algorithm terminates.



Tarjan's Algorithm: Example 3



2-SAT: Description

- Set of “or” expressions of two Boolean variables connected together by “and” operators.
  - How do we represent this problem so that we can solve it easily?
  - Does a solution exist?
  - What is a solution?

$$(x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge (x_2 \vee \neg x_4) \wedge (x_0 \vee \neg x_5) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_5) \wedge (x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6).$$

2-SAT: Solution Part 1 – Visualization

- Note that we can convert this problem into a graph problem!

$$(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$$

- For each Boolean variable v, create one node for v and one for **v**. Add in all the edges created by the implications.

- Note: for the rest of the presentation, I will be using **boldface** + underscore to represent negation (ie: **v** is equivalent to "not v")

2-SAT: Solution Part 1 – Visualization

Example 1: $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$

Example 2: $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$

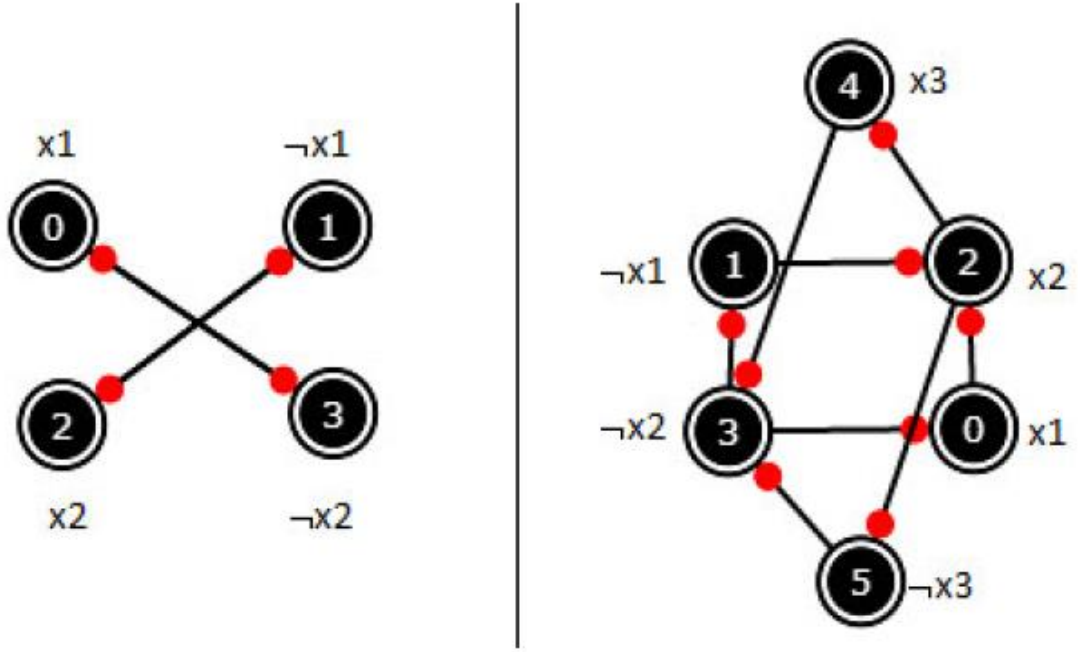


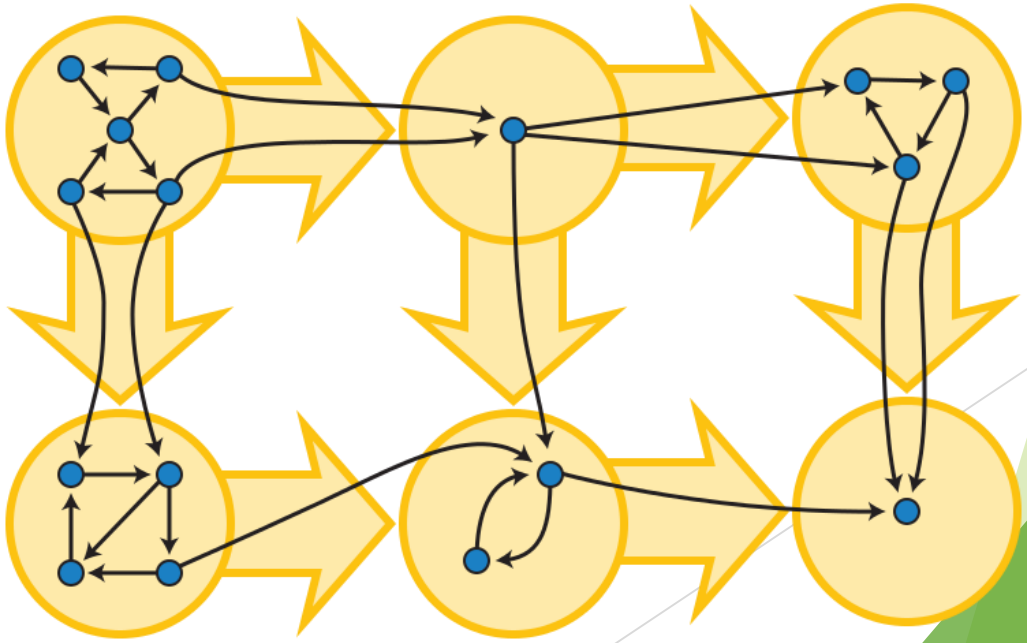
Figure 9.1: The Implication Graph of Example 1 (Left) and Example 2 (Right)

2-SAT: Solution Part 2 – Existence

- How do we know if there is a solution?
- Intuitively, if there is a variable v where v implies $\underline{\mathbf{v}}$ and $\underline{\mathbf{v}}$ implies v, there is no solution.
  - In our case, this means that if some v has a path to $\underline{\mathbf{v}}$ and $\underline{\mathbf{v}}$ has a path to v, there is no solution.
- It turns out that there is a tighter constraint that fully characterizes this problem: there is a solution if and only if each pair of nodes v and $\underline{\mathbf{v}}$ are not in the same SCC.
  - Can find SCCs using Tarjan's Algorithm
- Note: There are other ways to solve 2-SATs, but this is the one we will be covering.

2-SAT: Solution Part 3 – Solution

- This is assuming existence of a solution was shown with previous step.
- Construct the condensation of the graph (with “Super nodes” being SCCs)
- Topological sort the strongly connected components
- Iterate through the reverse order of the topological sort. Assign to each variable the truth value of the first Boolean variable representing it (ie: if **v** comes before v in this reverse topological sort, then assign the value of v to be false).



Kattis Problem: Running MoM

- https://open.kattis.com/problems/runningmom

- Abridged: Given up to 5000 cities and (one directional) flights between cities each day.
- Up to 1000 queries: given some starting city, is there an infinite path starting from that city?

Kattis Problem: Running MoM

...

```
public static boolean modifiedTarjan(int v){
  onStack[v] = true;
  checked[v] = true; //instead of Index.
  for (Integer i: edges.get(v)){
    if (checked[i]){
      if (onStack[i])
        return true;
    } else {
      if (modifiedTarjan(i))
        return true;
    }
  }
  onStack[v] = false;
  return false;
}
```

Shortest Path Faster Algorithm

- Uses a queue to remove redundant checks in the Bellman-Ford algorithm.
  - Is an optimization of Bellman-Ford, but feels more like Dijkstra's
- Complexity is O(kE), where k depends on the graph and $k \leq V$.
  - Worst case is fully connected graph: no optimization on BF, but extra overhead.
- In practice (according to the book Competitive Programming 3), this algorithm is as fast as Dijkstra's, but can also handle negative edges.

Recall: Bellman-Ford

```
/* Distance is initialized to +infinity
 * for all vertices exept the start node. */
static double distance[];
static ArrayList<SimpleEdge> edges;

                    ...


for (int i = 0; i < numVertices; i++){
  for(SimpleEdge e: edges){
    if (distance[e.v1] + e.w < distance[e.v2])
      distance[e.v2] = distance[e.v1] + e.w;
  }
}
```

Shortest Path Faster Algorithm

```
/* Distance is initialized to +infinity
 * for all vertices exept the start node. */
static double distance[];
static ArrayList<ArrayList<FastEdge>> edges;
static boolean[] inQueue;
static LinkedList<Integer> queue = new LinkedList<Integer>();
```

•••

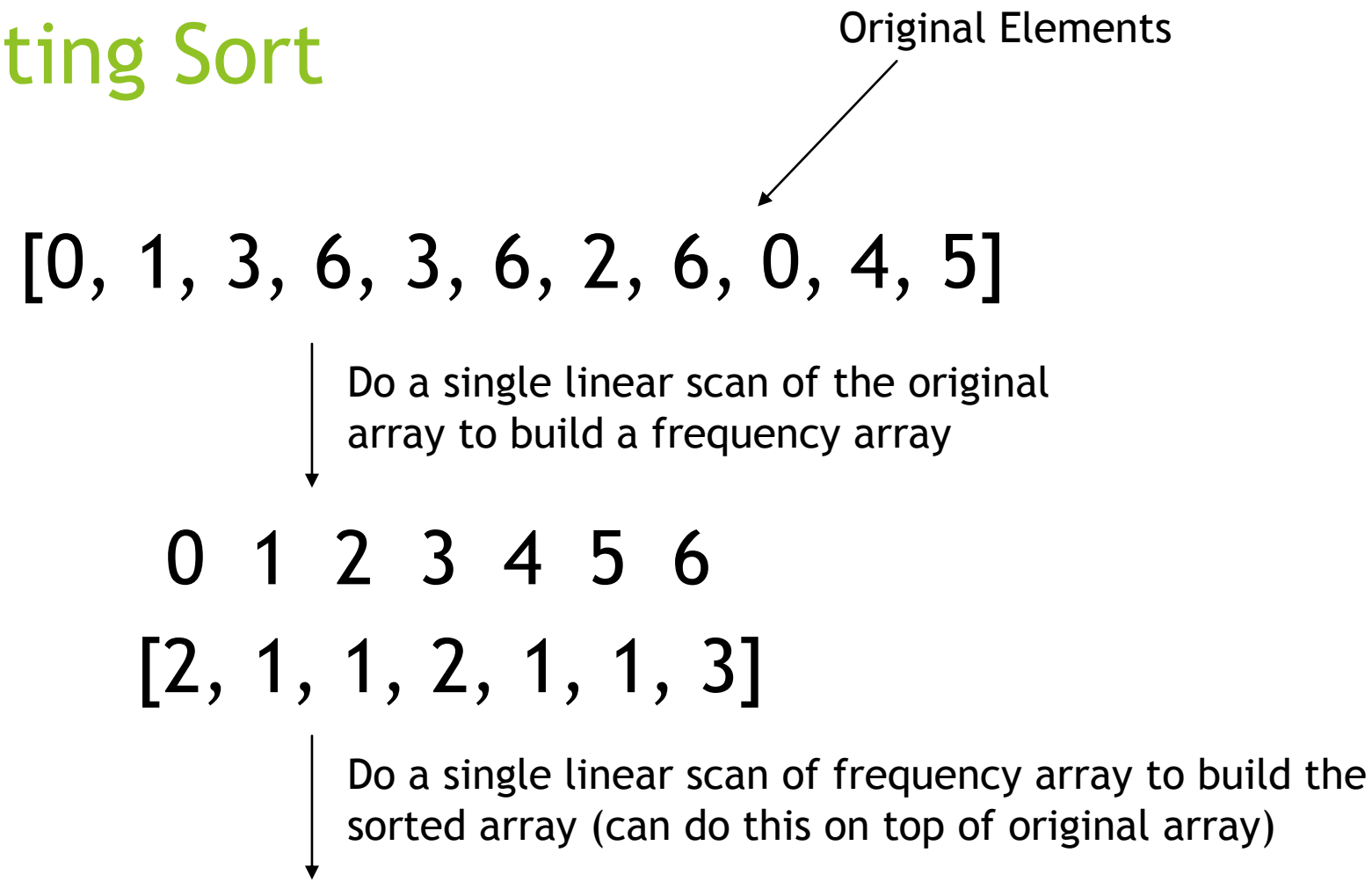
```
//run Shortest Path Faster Alg
queue.add(startNode);
while(!queue.isEmpty()){
  int vertex = queue.removeFirst();
  inQueue[vertex] = false;

  for(FastEdge e: edges.get(vertex)){
    if (distance[e.v1] + e.w < distance[e.v2]){
      distance[e.v2] = distance[e.v1] + e.w;
      if (!inQueue[e.v2]){
        inQueue[e.v2] = true;
        queue.addLast(e.v2);
      }
    }
  }
}
```

Counting Sort

- Linear Time!
- Let N be the number of elements to sort, and K be the set of possible values of the N numbers.
- Complexity: O(N + K)
- Can only be done if there is a small range of possible values (otherwise O(NlogN) might be better)

Counting Sort



Radix Sort

- What if the range of numbers is high?
- If the numbers are non-negative and the number of digits of the highest values is low (say D) , we can do radix sort! Complexity: O(D x N)
- First step: Pad each number with zeroes so they have the same number of digits
  - Note: we don't actually need to do this when coding. For example, if we want the fourth digit of 332, we do $\frac{332}{1000} = 0$
- Second step: Perform counting sort on each of the digits, starting with the least significant digit, while maintaining order to break ties.

Radix Sort

Input d = 4	Append Zeroes	Sort by the fourth digit	Sort by the third digit	Sort by the second digit	Sort by the first digit
323	0323	032(2)	00(1)3	0(0)13	(0)013
1257	1257	032(3)	03(2)2	1(2)57	(0)322
13	0013	001(3)	03(2)3	0(3)22	(0)323
322	0322	125(7)	12(5)7	0(3)23	(1)257

Which sort is better?

- Example: Suppose range of number is [0, 10^9].
- Note: here we are referring to log base 2

Number of entries to sort	NlogN sort operations	Counting Sort operations	Radix Sort operations	Best Algorithm
1000	1000 x log(1000) → 10^4	1000 + 10^9 → 10^9	1000*9 → 10^4	NlogN (Radix has a lot of overhead)
10^6	10^6 x log(10^6) → 2 x 10^7	10^6 + 10^9 → 10^9	10^6 x 9 → 10^7	Radix Sort
10^9	10^9 x log(10^9) → 3 x 10^{10}	10^9 + 10^9 → 2 x 10^9	10^9 x 9 →10^{10}	Counting Sort

References

- Halim, S., & Halim, F. (2013). *Competitive Programming 3: the new lower bound of programming contests*. Singapore: Lulu.com.

- The one and only most accurate resource on the internet, Wikipedia.