CSE 421 Winter 2025
Lecture 4: Asymptotics and Graph Search

Nathan Brunelle

http://www.cs.uw.edu/421

Complexity analysis

- Problem size n
  - **Worst-case complexity**:
    maximum # steps algorithm takes on any input of size n
  - **Best-case complexity:**
    minimum # steps algorithm takes on any input of size n
  - **Average-case complexity**:
    Expected # steps algorithm takes on inputs of size n

Complexity

- The complexity of an algorithm associates a number $\boldsymbol{T(n)}$, the worst/average-case/best time the algorithm takes, with each problem size **n**.

- Mathematically,
  - $\boldsymbol{T}$ is a function that maps positive integers giving problem size to positive real numbers giving number of steps.

- Sometimes we have more than one size parameter
  - e.g. $\boldsymbol{n}$=# of vertices, $\boldsymbol{m}$=# of edges in a graph.

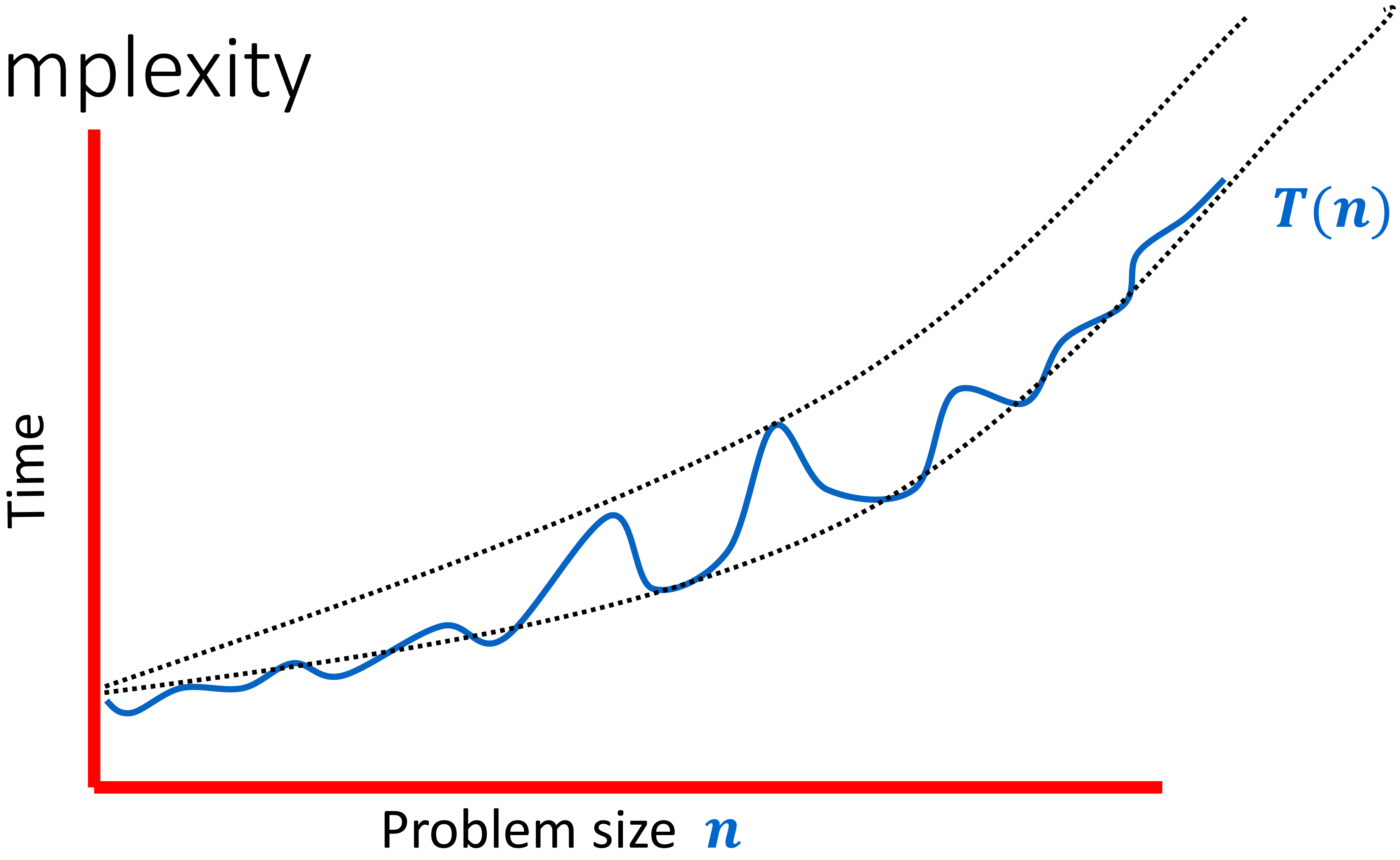
Efficient = Polynomial Time

- Polynomial time
  - Running time $T(n) \le cn^k + d$ for some $c, d, k \geq 0$

- Why polynomial time?
  - If problem size grows by at most a constant factor then so does the running time
    - e.g. $T(2n) \le c\,(2n)^k + d = 2^k cn^k + d \le 2^k\left(cn^k + d\right) = 2^k\,T(n)$
    - polynomial-time is exactly the set of running times that have this property

  - Typical running times are small degree polynomials, mostly less than n^3, at worst n^6, not n^{100}

Complexity



O-notation etc

- Given two positive functions $\boldsymbol{f}$ and $\boldsymbol{g}$
  - $\boldsymbol{f(n)}$ is $\boldsymbol{O(g(n))}$ iff there is a constant $\boldsymbol{c > 0}$ so that $\boldsymbol{f(n)}$ is eventually always $\le \boldsymbol{c \cdot g(n)}$
  - $\boldsymbol{f(n)}$ is $\boldsymbol{o(g(n))}$ iff for every constant $\boldsymbol{c > 0}$ $\boldsymbol{f(n)}$ is eventually always $\le \boldsymbol{c \cdot g(n)}$
  - $\boldsymbol{f(n)}$ is $\boldsymbol{\Omega(g(n))}$ iff there is a constant $\boldsymbol{\varepsilon > 0}$ so that $\boldsymbol{f(n) \ge \varepsilon \cdot g(n)}$ for infinitely many values of $\boldsymbol{n}$
  - $\boldsymbol{f(n)}$ is $\boldsymbol{\omega(g(n))}$ iff for every constant $\boldsymbol{c > 0}$ $\boldsymbol{f(n)}$ is eventually always $\ge \boldsymbol{c \cdot g(n)}$
  - $\boldsymbol{f(n)}$ is $\boldsymbol{\Theta(g(n))}$ iff $\boldsymbol{f(n)}$ is $\boldsymbol{O(g(n))}$ and $\boldsymbol{f(n)}$ is $\boldsymbol{\Omega(g(n))}$

Note: The definition of "$\boldsymbol{f(n)}$ is $\boldsymbol{\Omega(g(n))}$" is *almost* the same as "$\boldsymbol{f(n)}$ is **not** $\boldsymbol{o(g(n))}$"

The definition of "$\boldsymbol{f(n)}$ is $\boldsymbol{\Omega(g(n))}$" is *almost* the same as "$\boldsymbol{f(n)}$ is **not** $\boldsymbol{o(g(n))}$"

Asymptotic Notation intuition

$f(n)$ is...

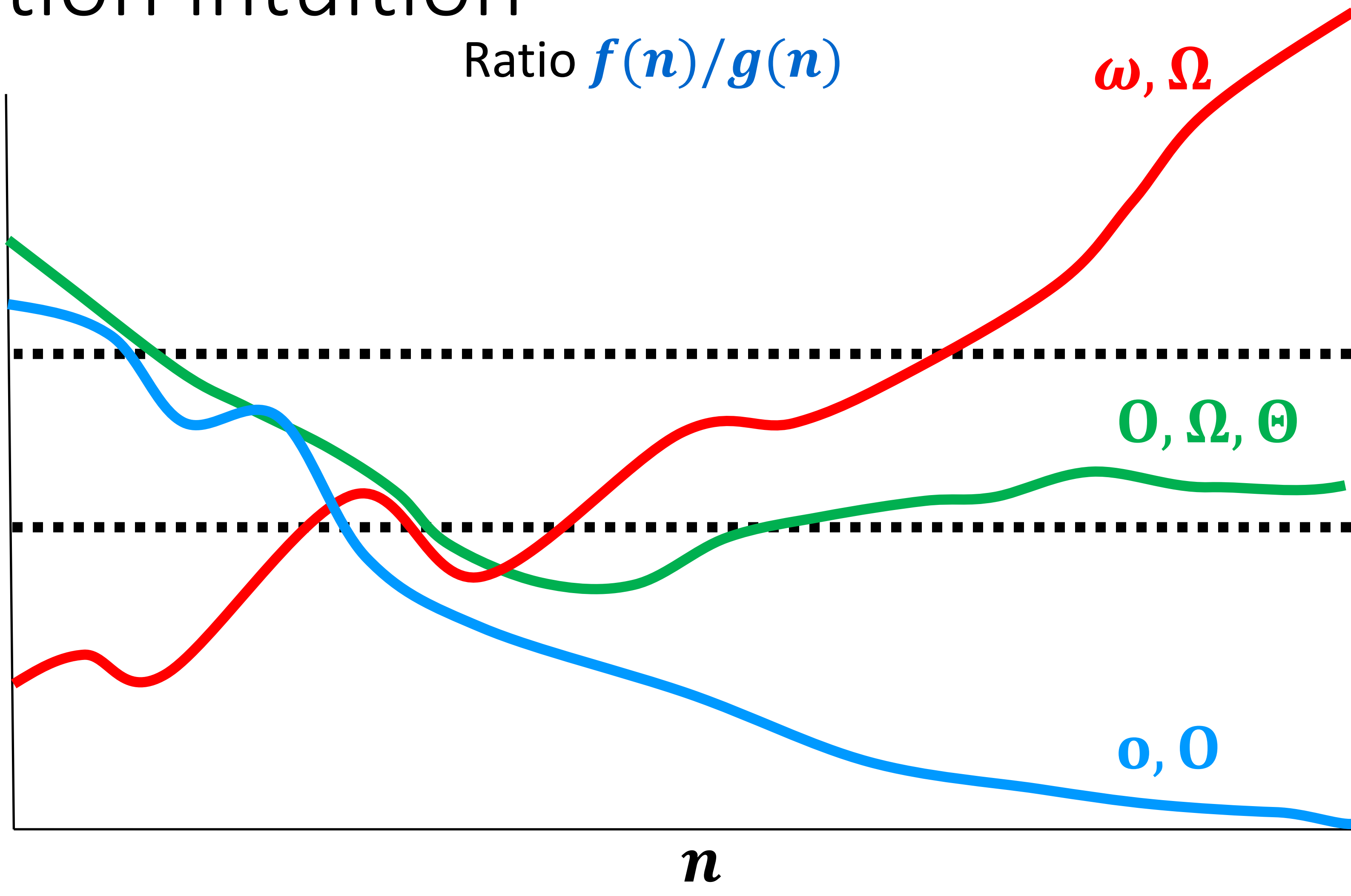
$O(g(n))$: ratio eventually below some line forever

$o(g(n))$: ratio eventually below every line forever

$\Omega(g(n))$: ratio eventually above some line forever

$\omega(g(n))$: ratio eventually above every line forever

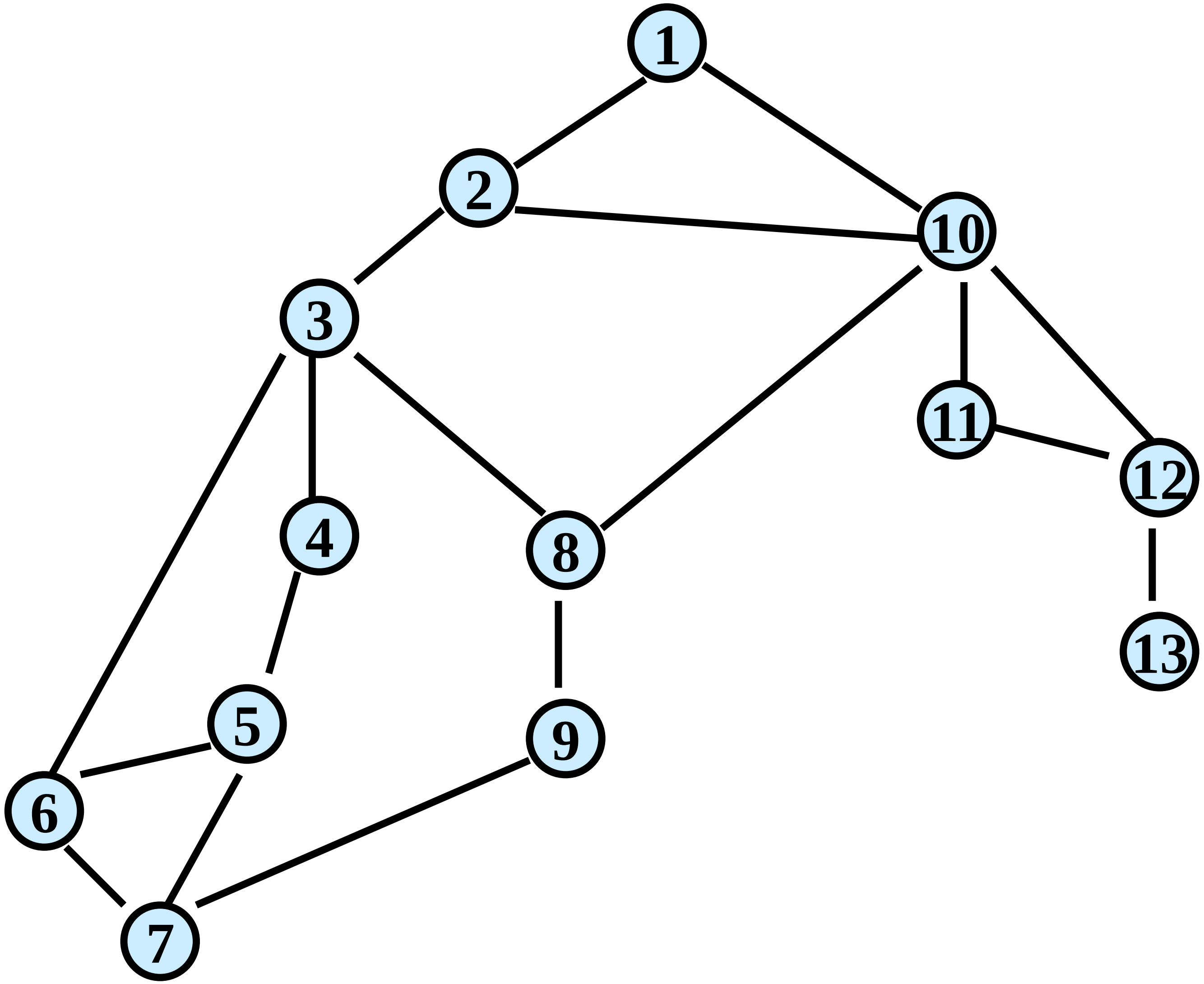
$\Theta(g(n))$: both O and Ω



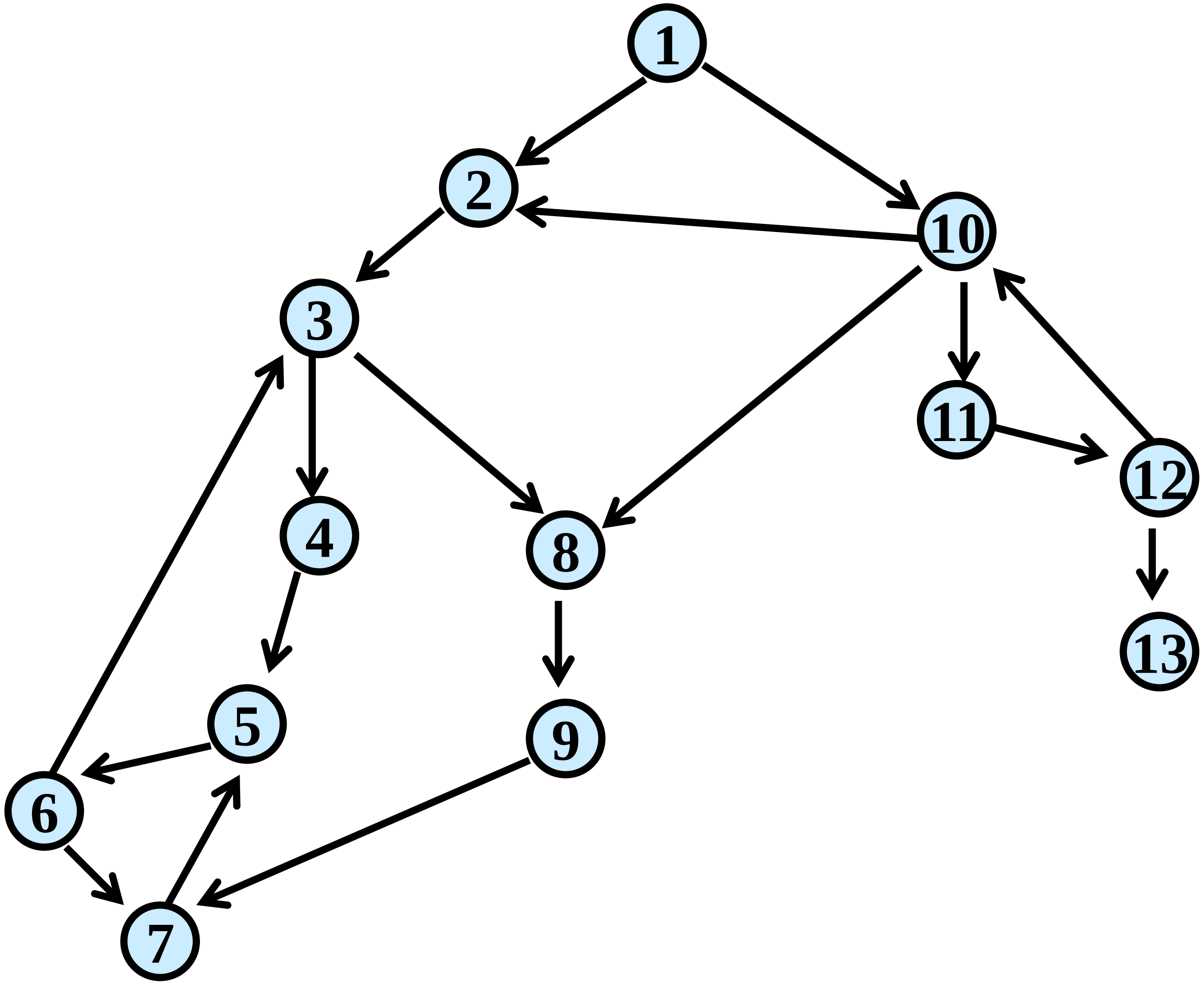
Introduction to Algorithms

- **Graph Search/Traversal**

Undirected Graph G = (V,E)



Directed Graph G = (V,E)



Graph Traversal

Learn the basic structure of a graph

Walk from a fixed starting vertex $\boldsymbol{s}$ to find all vertices reachable from $\boldsymbol{s}$

Generic Graph Traversal Algorithm

Given: Graph graph $\boldsymbol{G = (V, E)}$ vertex $\boldsymbol{s \in V}$

Find: set $\boldsymbol{R}$ of vertices reachable from $\boldsymbol{s \in V}$

Reachable($\boldsymbol{s}$):

Add $\boldsymbol{s}$ to $\boldsymbol{R}$

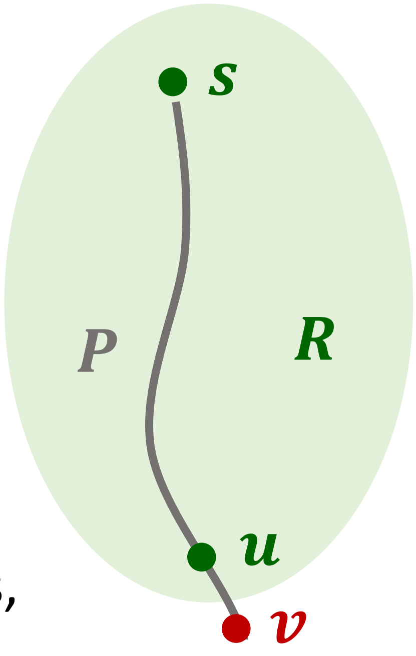
while there is a $\boldsymbol{(u, v) \in E}$ where $\boldsymbol{u \in R}$ and $\boldsymbol{v \notin R}$

Add $\boldsymbol{v}$ to $\boldsymbol{R}$

return $\boldsymbol{R}$

Generic Traversal Always Works

Claim: At termination, R is the set of nodes reachable from s



Proof

$\subseteq$: For every node $v \in R$ there is a path from s to v

- Induction based on edges found.
  - Base case: s is reachable from s
  - Inductive step: If there is a path to every member of R after i iterations, then there is a path to every member of R after $i + 1$ iterations

$\supseteq$: Suppose there is a node $w \notin R$ reachable from s via a path P

- Take first node v on P such that $v \notin R$
- Predecessor u of v in P satisfies
  - $u \in R$
  - $(u, v) \in E$
- But this contradicts the fact that the algorithm exited the while loop.

Graph Traversal

Learn the basic structure of a graph

Walk from a fixed starting vertex $\boldsymbol{s}$ to find all vertices reachable from $\boldsymbol{s}$

Three states of vertices

- **unvisited**
- **visited/discovered** (in $\boldsymbol{R}$)
- **fully-explored** (in $\boldsymbol{R}$ and all neighbors have been visited)

Breadth-First Search

Completely explore the vertices in order of their distance from s

Naturally implemented using a queue

BFS(s)

Global initialization: mark all vertices "unvisited"

BFS(s)

    Mark s "visited"
    Add s to Q
    i = 0
    Mark s as "layer i"
    while Q not empty
        u = next item removed from Q
        i = "layer of u"
        for each edge (u, v)
            if (v is "unvisited")
                mark v "visited"
                mark s as "layer $i + 1$"
        mark u "fully-explored"

Properties of BFS

BFS(s) visits x iff there is a path in G from s to x.

Edges followed to undiscovered vertices define a breadth first spanning tree of G

Layer i in this tree:

L_i = set of vertices u with shortest path in G from root s of length i.

Properties of BFS

Claim: For undirected graphs:

All edges join vertices on the same or adjacent layers of BFS tree

Proof: Suppose not...

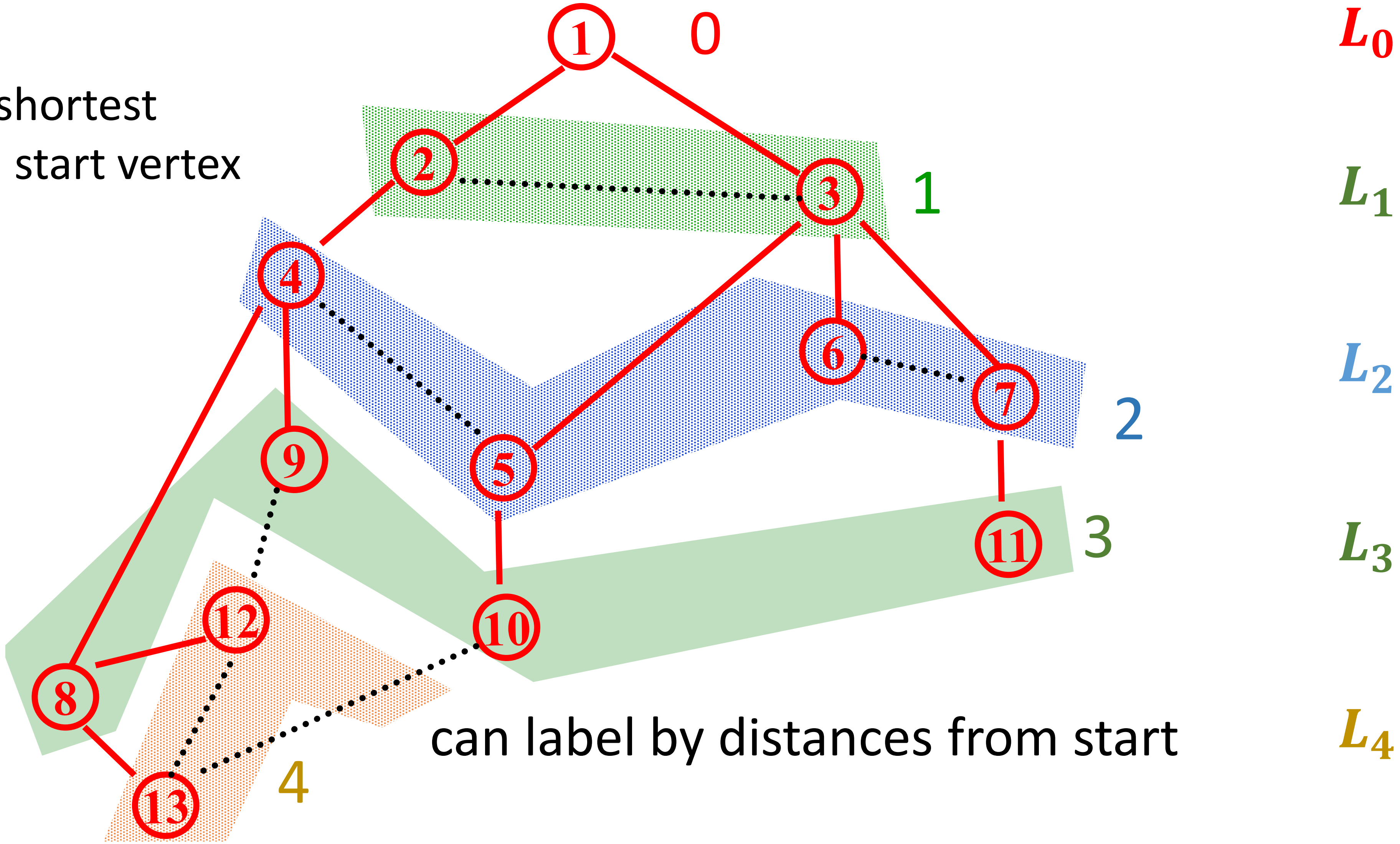
Then there would be vertices (x, y) s.t. $x \in L_i$ and $y \in L_j$ and $j > i + 1$.

Then, when vertices adjacent to x are considered in BFS,
y would be added with layer $i + 1$ and not layer j.

Contradiction.

BFS Application: Shortest Paths

Tree gives shortest paths from start vertex



can label by distances from start

Undirected Graph Search Application: Connected Components

Want to answer questions of the form:

Given: vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\boldsymbol{G}$

Is there a path from $\boldsymbol{u}$ to $\boldsymbol{v}$?

Idea: create array $\boldsymbol{A}$ s.t

$\boldsymbol{A}[\boldsymbol{u}]$ = smallest numbered vertex connected to $\boldsymbol{u}$

Answer is yes iff $\boldsymbol{A}[\boldsymbol{u}] = \boldsymbol{A}[\boldsymbol{v}]$

Q: Why is this better than an array **Path**$[\boldsymbol{u}, \boldsymbol{v}]$?

Undirected Graph Search Application: Connected Components

Initial state: all v unvisited
for s from 1 to n do:
    if state(s) $\neq$ fully-explored then
        BFS(s): setting $A[u] = s$ for each u found
            (and marking u visited/fully-explored)

Total cost: $O(n+m)$

- Each vertex is touched once in outer procedure and edges examined in different BFS runs are disjoint
- Works also with Depth First Search ...

DFS($\boldsymbol{u}$) – Recursive Procedure

Global Initialization: mark all vertices "unvisited"

DFS($\boldsymbol{u}$)

mark $\boldsymbol{u}$ “visited” and add $\boldsymbol{u}$ to $\boldsymbol{R}$

for each edge $(\boldsymbol{u}, \boldsymbol{v})$

if ($\boldsymbol{v}$ is “unvisited”)

DFS($\boldsymbol{v}$)

mark $\boldsymbol{u}$ “fully-explored”

Properties of DFS(s)

Like BFS(s):

- DFS(s) visits x iff there is a path in G from s to x
- Edges into undiscovered vertices define depth-first spanning tree of G

Unlike the BFS tree:

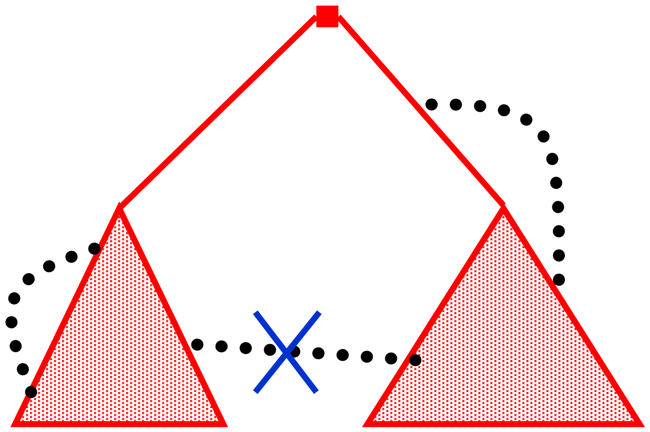
- the DFS spanning tree *isn't* minimum depth
- its levels *don't* reflect min distance from the root
- non-tree edges *never* join vertices on the same or adjacent levels

BUT…

Non-tree edges in DFS tree of undirected graphs

Claim: All non-tree edges join a vertex and one of its descendents/ancestors in the DFS tree

- In other words ... No "cross edges".



No cross edges in DFS on undirected graphs

Claim: During DFS($\boldsymbol{x}$) every vertex marked "visited" is a descendant of $\boldsymbol{x}$ in the DFS tree $\boldsymbol{T}$

Claim: For every $\boldsymbol{x}, \boldsymbol{y}$ in the DFS tree $\boldsymbol{T}$, if $(\boldsymbol{x}, \boldsymbol{y})$ is an edge *not* in $\boldsymbol{T}$ then one of $\boldsymbol{x}$ or $\boldsymbol{y}$ is an ancestor of the other in $\boldsymbol{T}$

Proof:

- One of DFS($\boldsymbol{x}$) or DFS($\boldsymbol{y}$) is called first, suppose WLOG that DFS($\boldsymbol{x}$) was called before DFS($\boldsymbol{y}$)
- During DFS($\boldsymbol{x}$), the edge $(\boldsymbol{x}, \boldsymbol{y})$ is examined
- Since $(\boldsymbol{x}, \boldsymbol{y})$ is a *not* an edge of $\boldsymbol{T}$, $\boldsymbol{y}$ was already visited when edge $(\boldsymbol{x}, \boldsymbol{y})$ was examined during DFS($\boldsymbol{x}$)
- Therefore $\boldsymbol{y}$ was visited during the call to DFS($\boldsymbol{x}$) so $\boldsymbol{y}$ is a descendant of $\boldsymbol{x}$.

Applications of Graph Traversal: Bipartiteness Testing

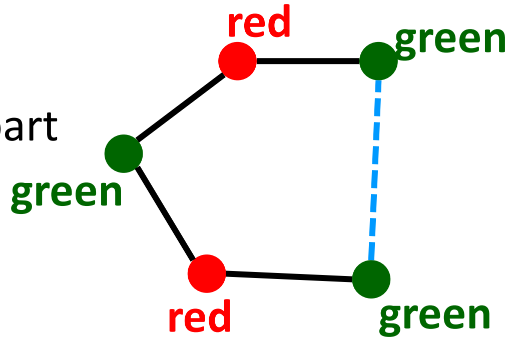
Definition: An undirected graph G is bipartite iff we can color its vertices **red** and **green** so each edge has different color endpoints

Input: Undirected graph G

Goal: If G is bipartite, output a coloring;
otherwise, output "NOT Bipartite".

Fact: Graph G contains an odd-length cycle $\Rightarrow$ it is not bipartite

Just coloring the cycle part of G is impossible



On a cycle the two colors must alternate, so

- **green** every 2nd vertex
- **red** every 2nd vertex

Can't have either if length is not divisible by 2.

Applications of Graph Traversal: Bipartiteness Testing

WLOG ("without loss of generality"): Can assume that G is connected

- Otherwise run on each component

Simple idea: start coloring nodes starting at a given node s

- Color s **red**
- Color all neighbors of s **green**
- Color all their neighbors **red**, etc.
- If you ever hit a node that was already colored
  - the **same** color as you want to color it, ignore it
  - the **opposite** color, output "NOT Bipartite" and halt

BFS gives Bipartiteness

Run BFS assigning all vertices from layer L_i the color $i \bmod 2$

- i.e., **red** if they are in an even layer, **green** if in an odd layer

- if no edge joining two vertices of the same color
  - then it is a good coloring
- otherwise
  - there is a bad edge; output “Not Bipartite”

Why is that “Not Bipartite” output correct?

Why does BFS work for Bipartiteness?

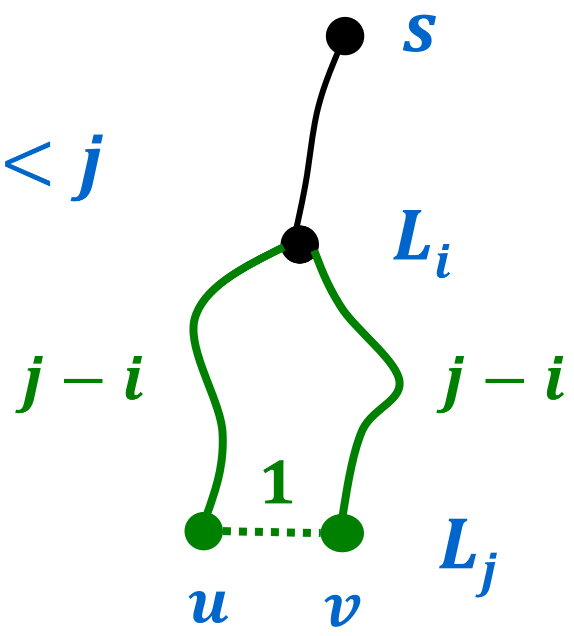
Recall: All edges join vertices on the same or adjacent BFS layers

⇒ Any "bad" edge must join two vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ in the same layer

Say the layer with $\boldsymbol{u}$ and $\boldsymbol{v}$ is $\boldsymbol{L_j}$

$\boldsymbol{u}$ and $\boldsymbol{v}$ have common ancestor at some level $\boldsymbol{L_i}$ for $\boldsymbol{i < j}$

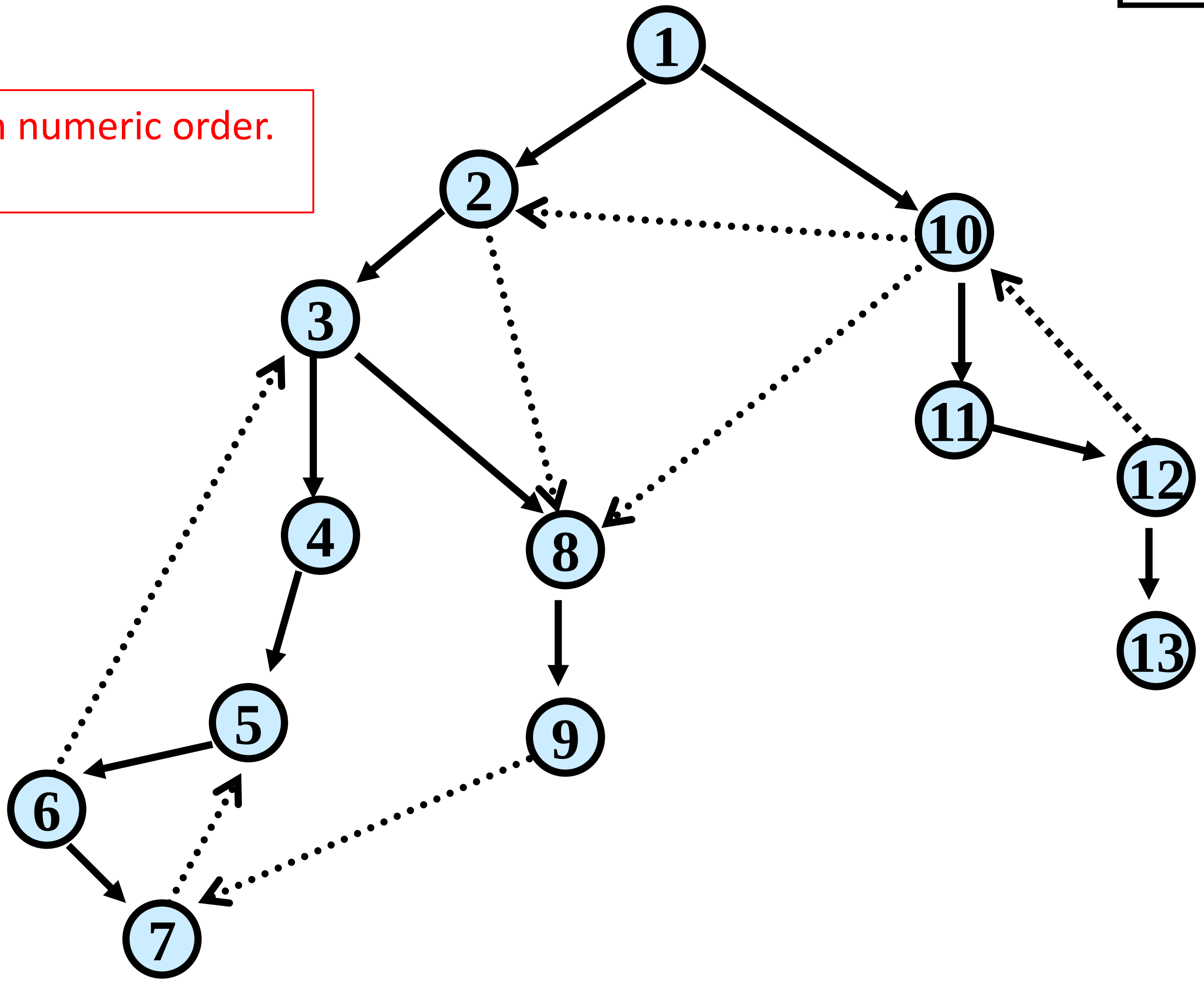
Odd cycle of length $\mathbf{2}(\boldsymbol{j - i}) + \mathbf{1}$
⇒ Not Bipartite



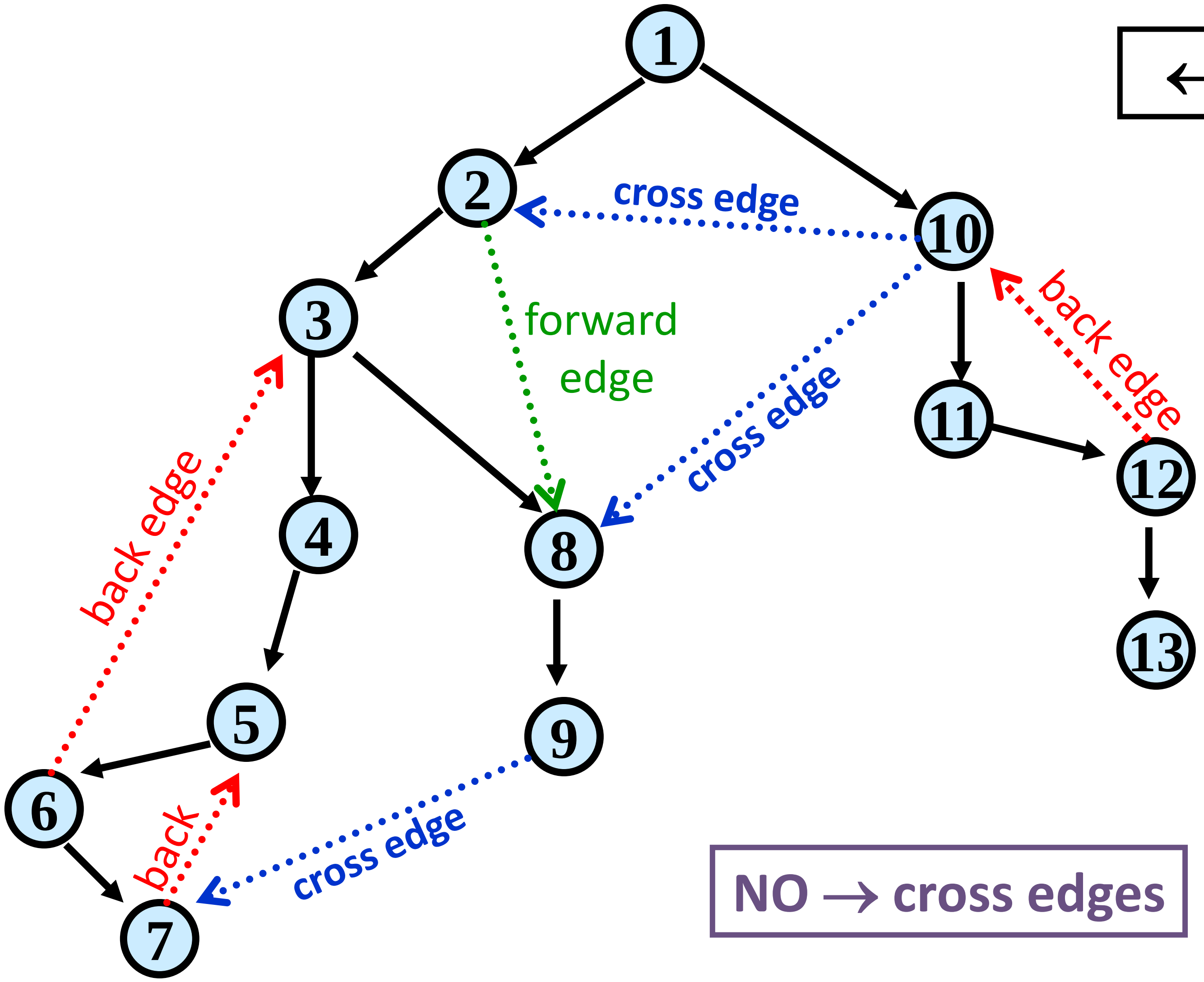
DFS($\boldsymbol{v}$) for a directed graph

← **tree edges**

Nodes were visited in numeric order.
How can we tell?



DFS($\boldsymbol{v}$)



← tree edges

NO → cross edges

Properties of Directed DFS

- Before DFS(s) returns, it visits all previously unvisited vertices reachable via directed paths from s

- Every cycle contains a back edge in the DFS tree

Directed Acyclic Graphs

A directed graph $G = (V, E)$ is acyclic iff it has no directed cycles

Terminology: A directed acyclic graph is also called a DAG

After shrinking the strongly connected components of a directed graph to single vertices, the result is a DAG

Topological Sort

Given: a directed acyclic graph (DAG) $G = (V, E)$

Output: numbering of the vertices of G with distinct numbers from 1 to n so that edges only go from lower numbered to higher numbered vertices

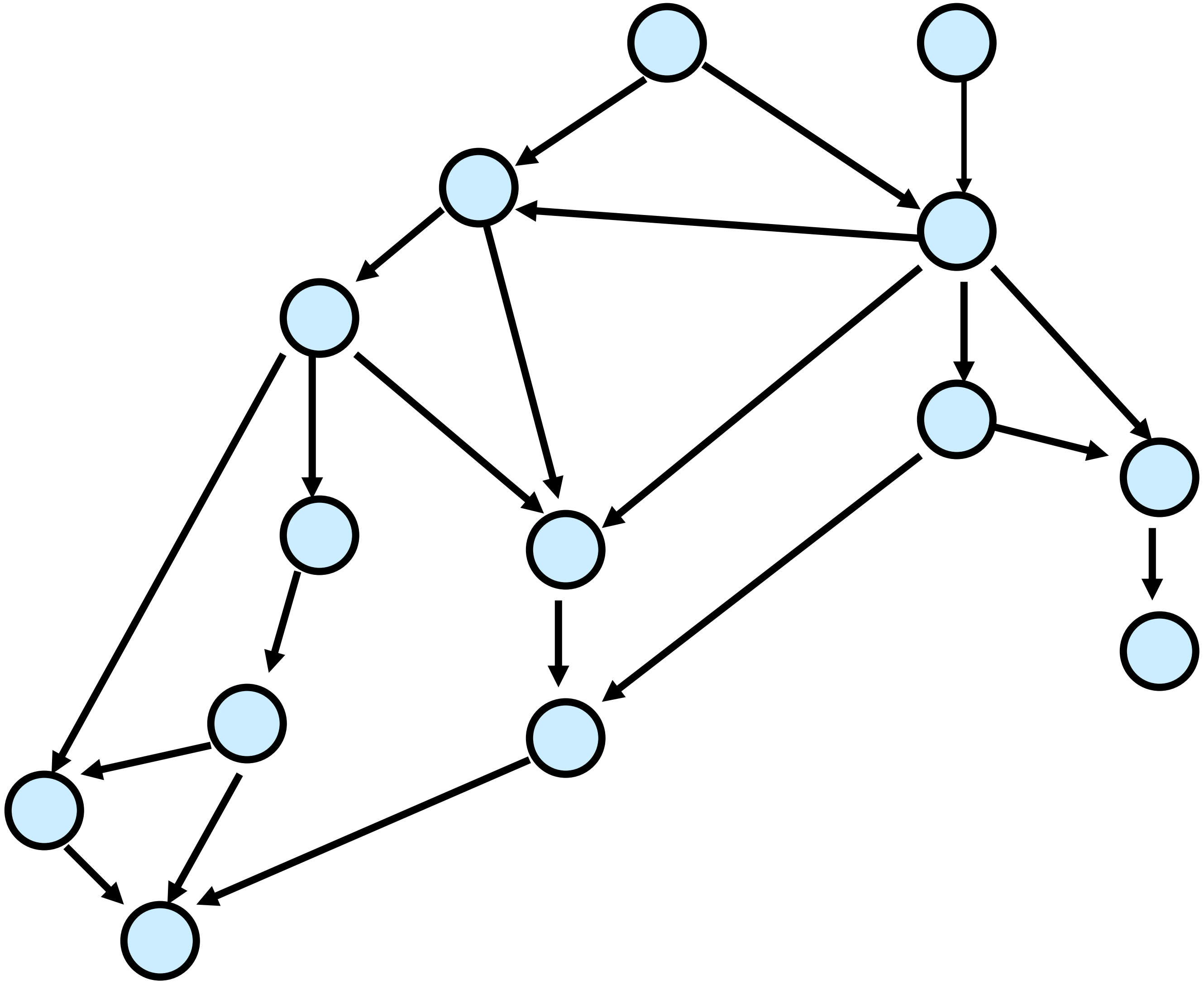
Applications:

- nodes represent tasks
- edges represent precedence between tasks
- topological sort gives a sequential schedule for solving them

Nice algorithmic paradigm for general directed graphs:

- Process strongly connected components one-by-one in the order given by topological sort of the DAG you get from shrinking them.

Directed Acyclic Graph



In-degree 0 vertices

Claim: Every DAG has a vertex of in-degree 0

Proof: By contradiction

Suppose every vertex has some incoming edge

Consider following procedure:

while (true) do

$\boldsymbol{v}$ = some predecessor of $\boldsymbol{v}$

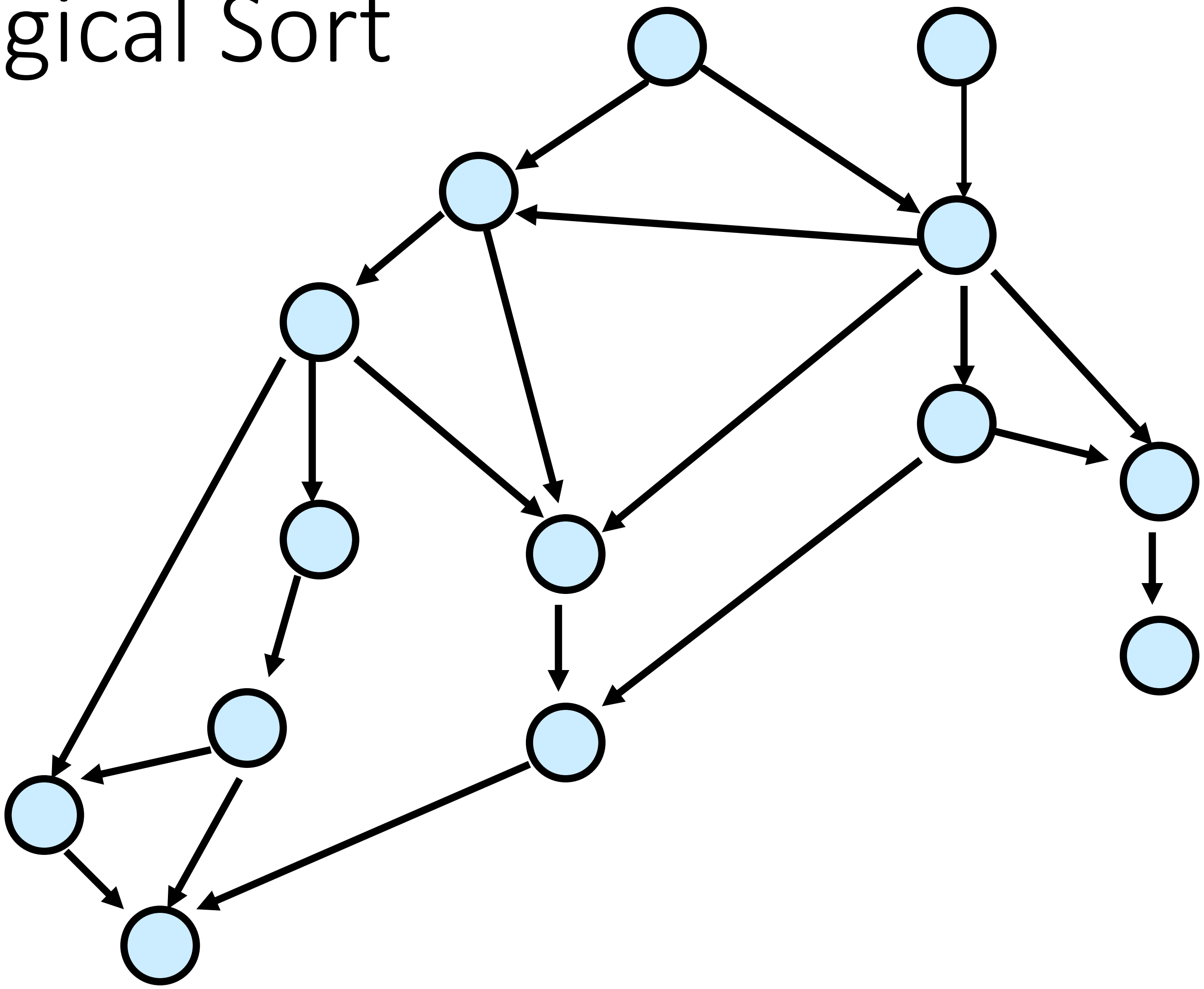
- After $\boldsymbol{n+1}$ steps where $\boldsymbol{n = |V|}$ there will be a repeated vertex
  - This yields a cycle, contradicting that it is a DAG.

Topological Sort

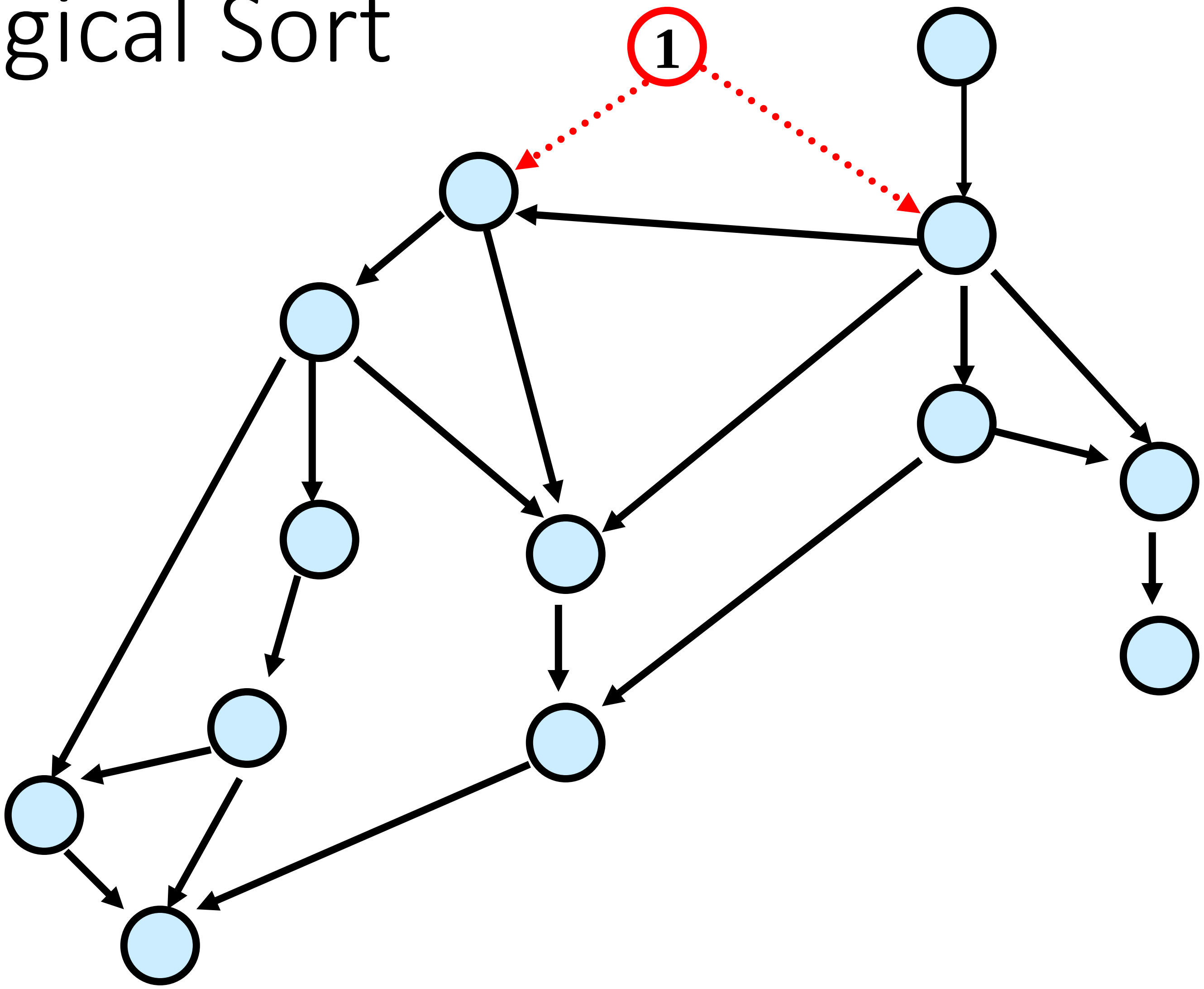
- Can do using DFS

- Alternative simpler idea:
  - Any vertex of in-degree 0 can be given number 1 to start
  - Remove it from the graph
  - Then give a vertex of in-degree 0 number 2
  - Etc.

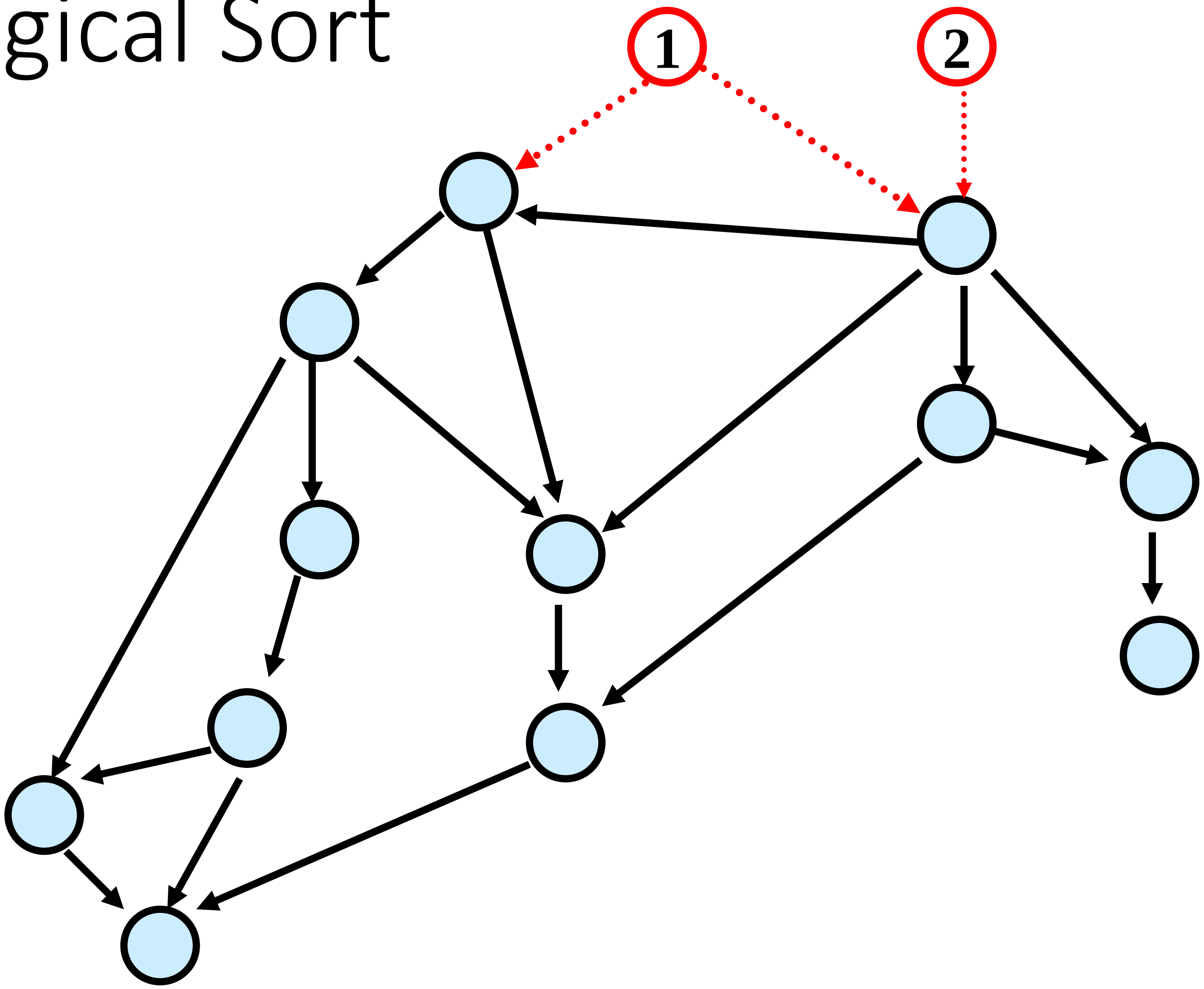
Topological Sort



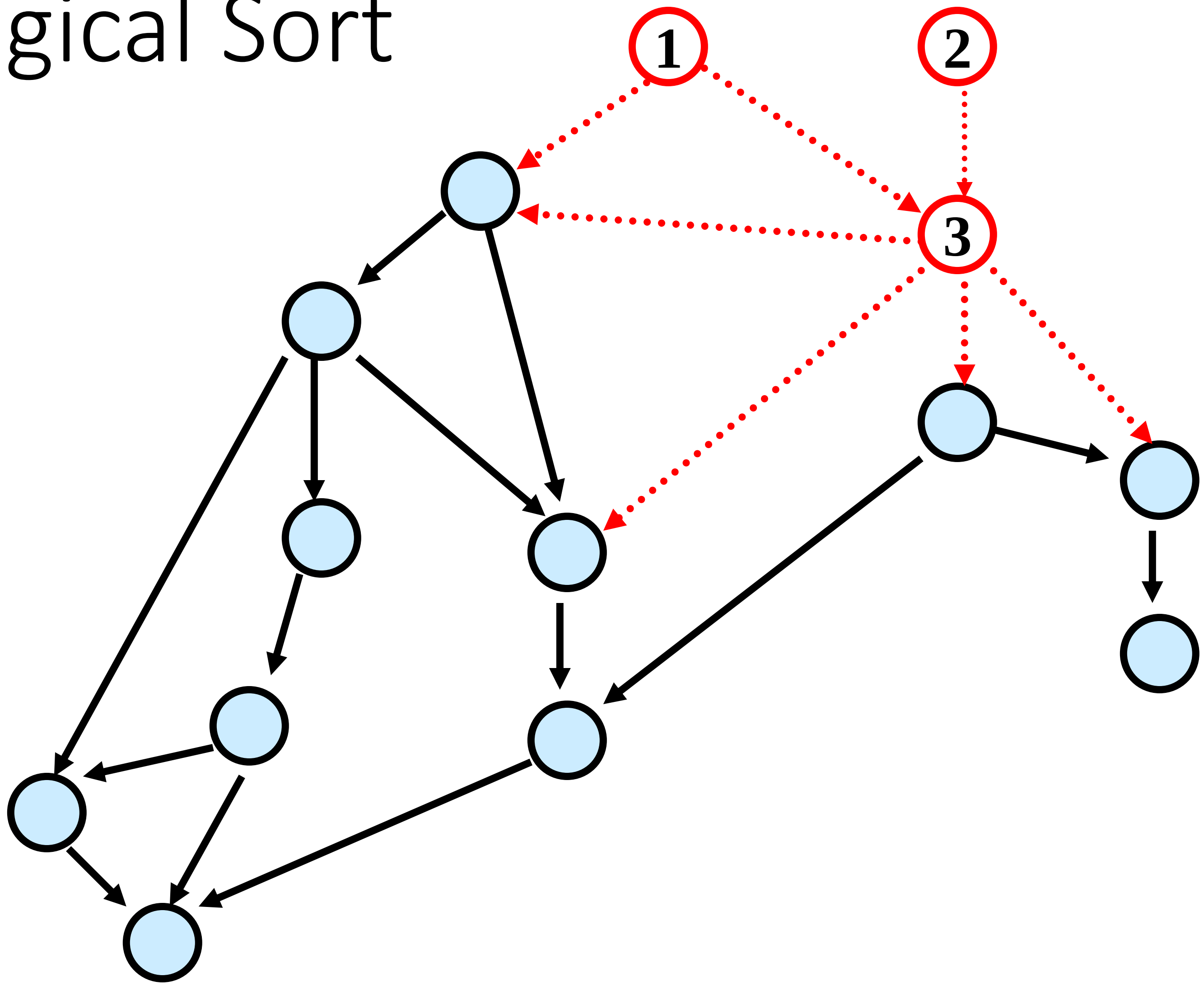
Topological Sort



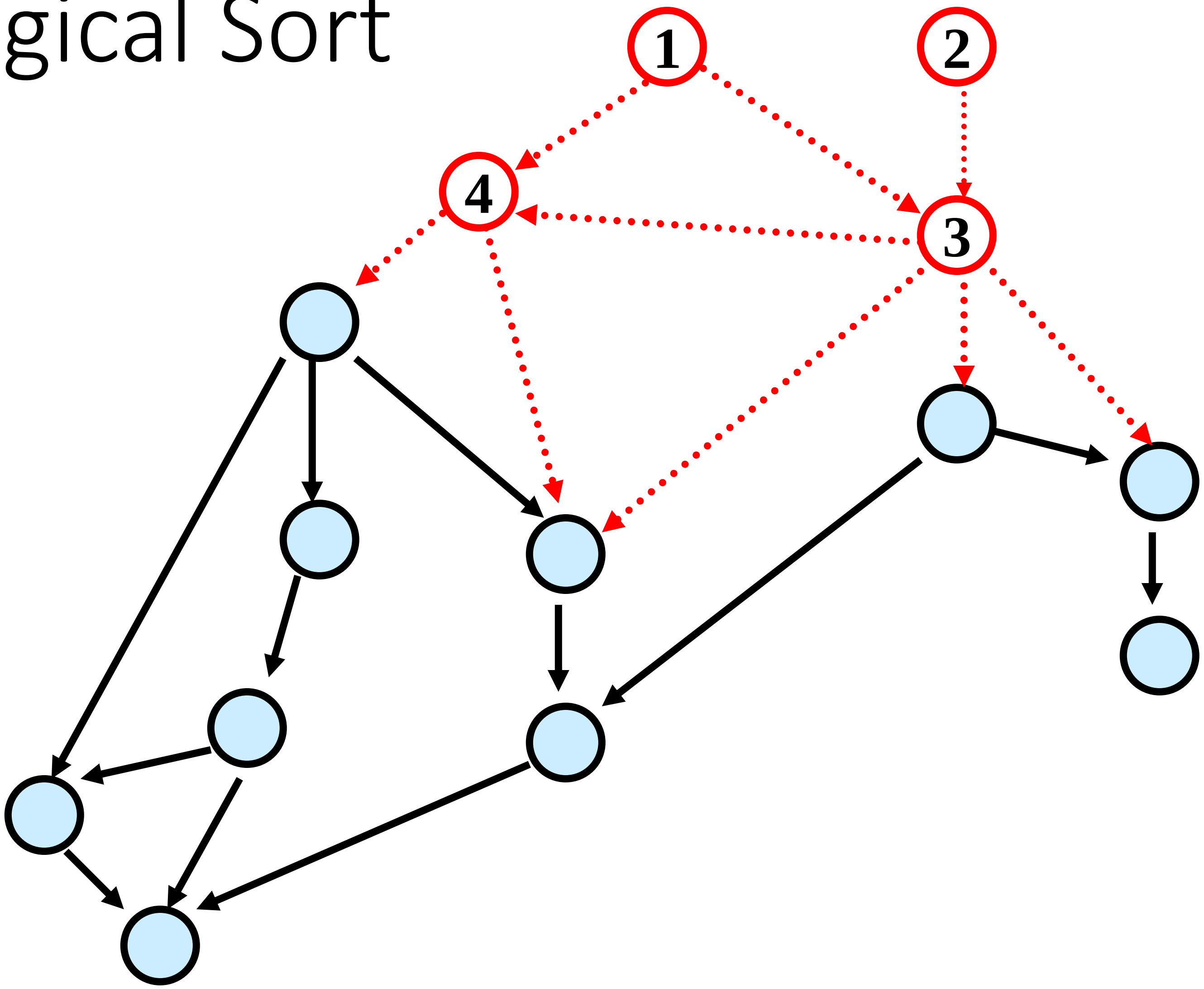
Topological Sort



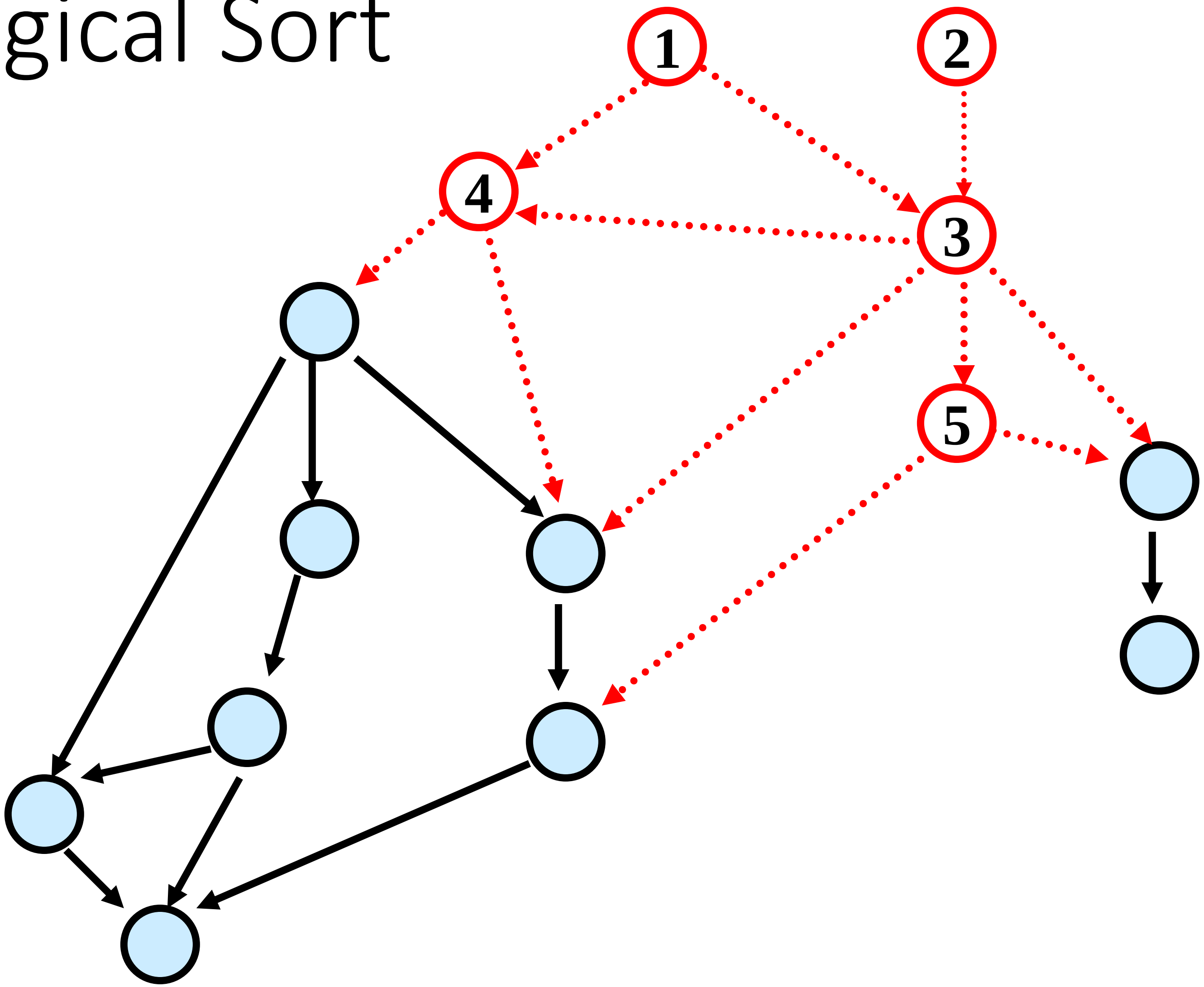
Topological Sort



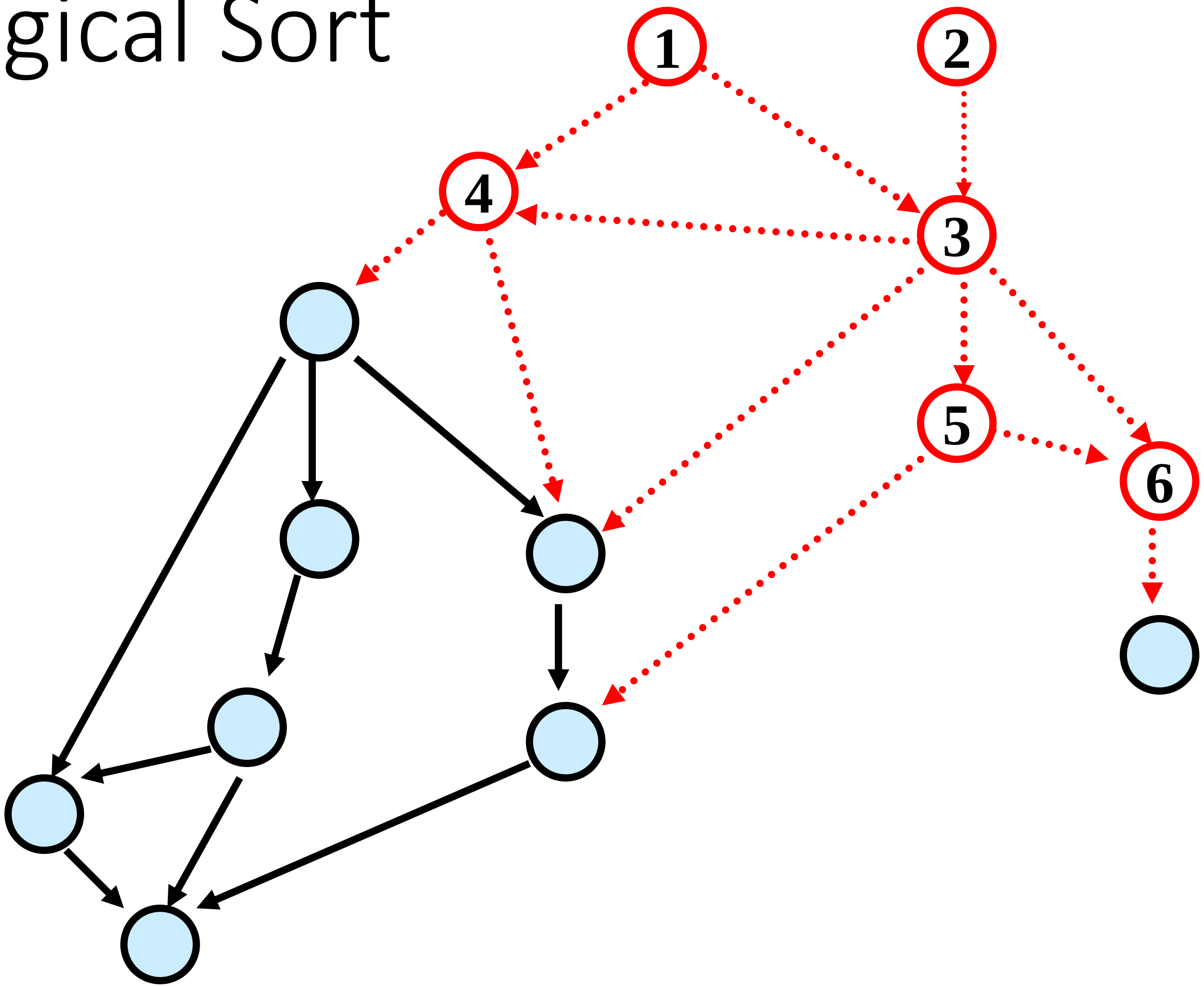
Topological Sort



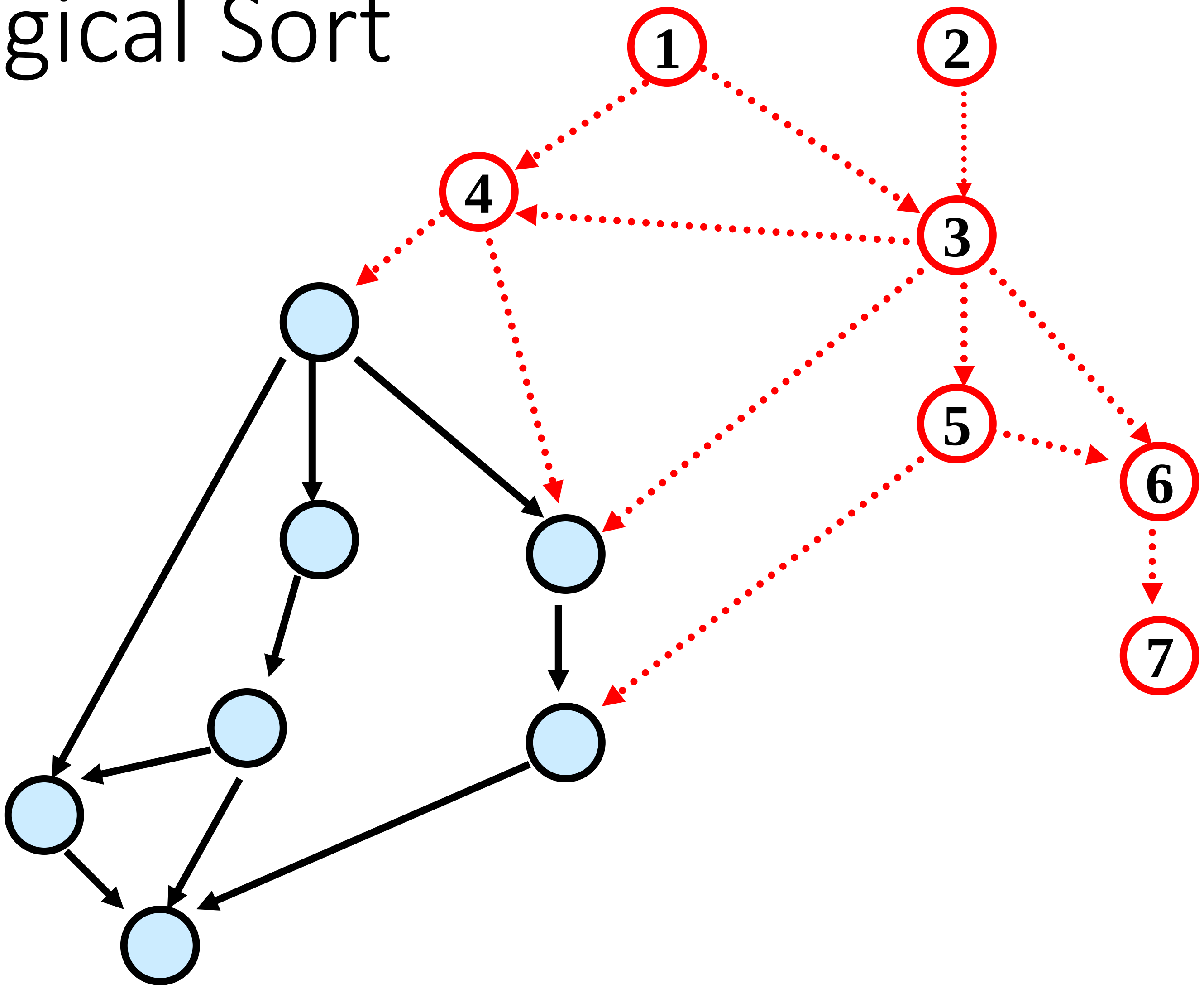
Topological Sort



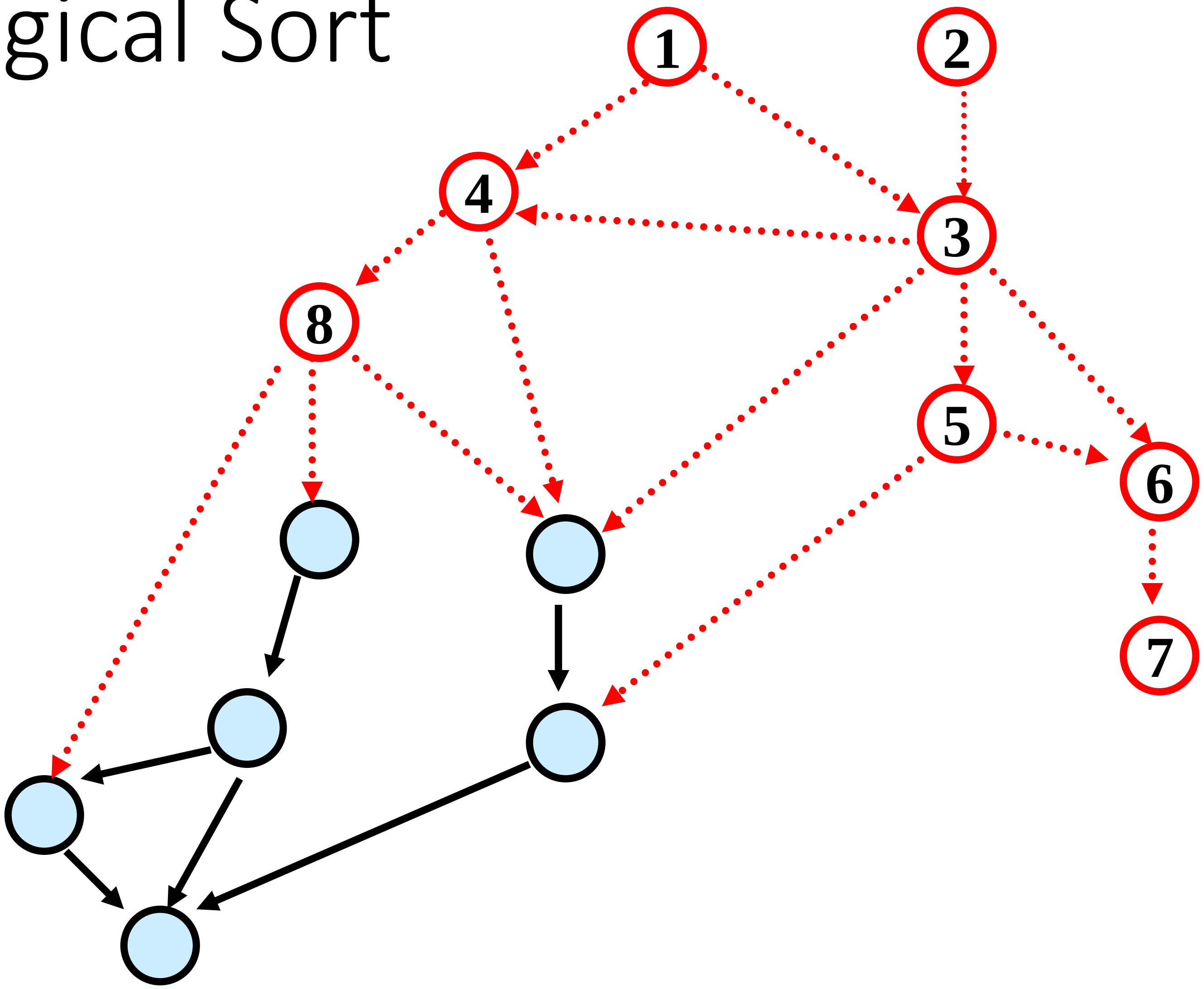
Topological Sort



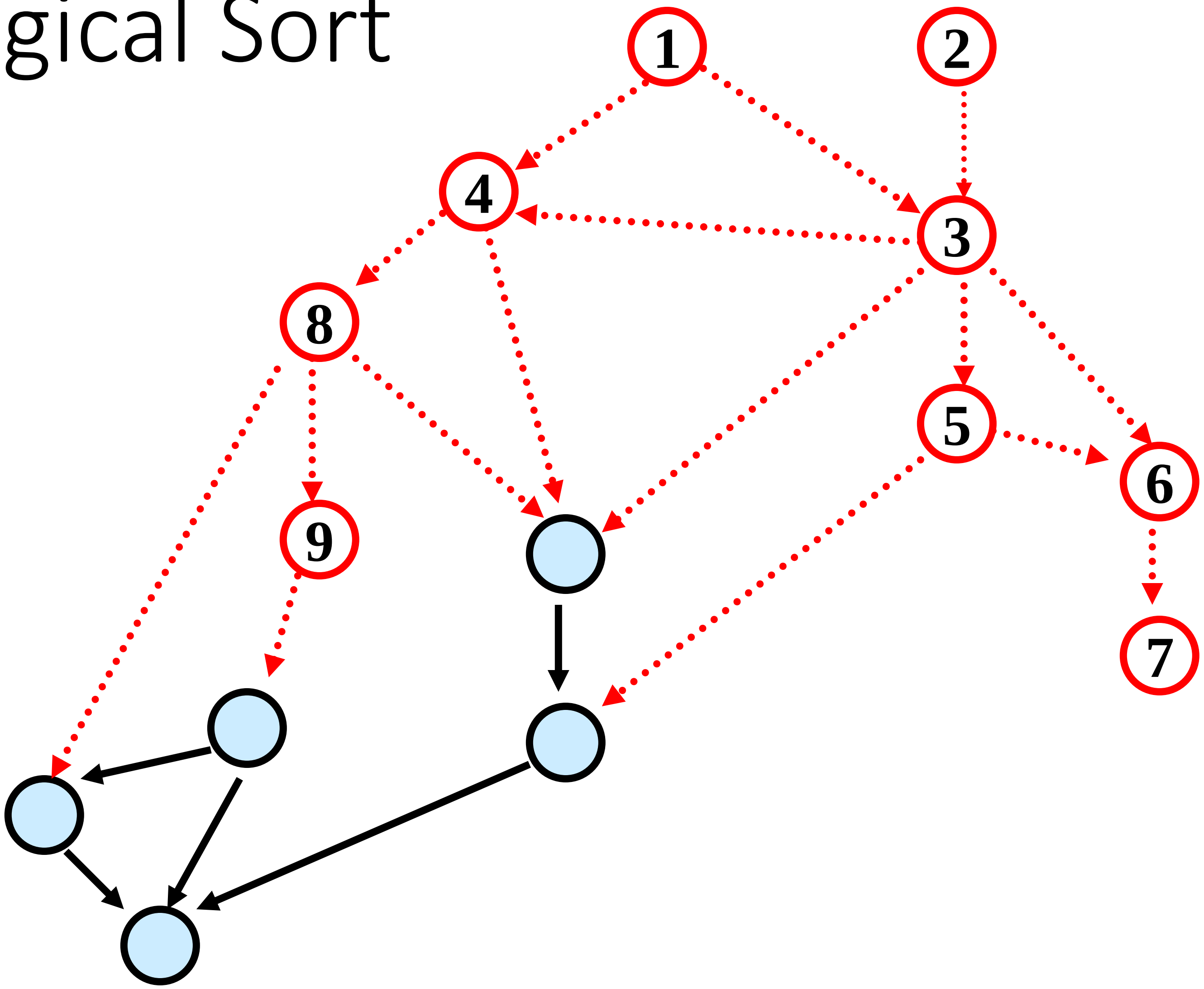
Topological Sort



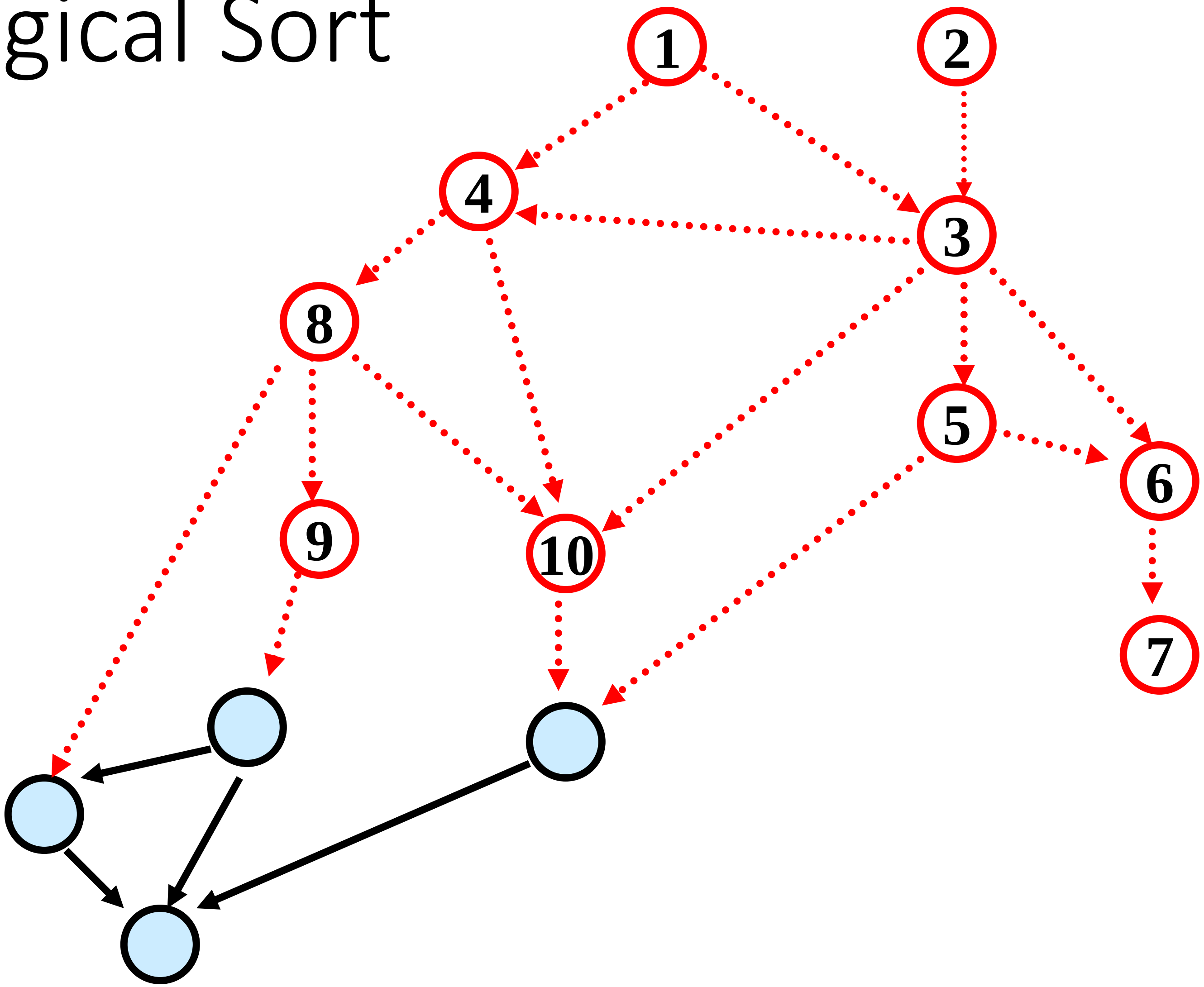
Topological Sort



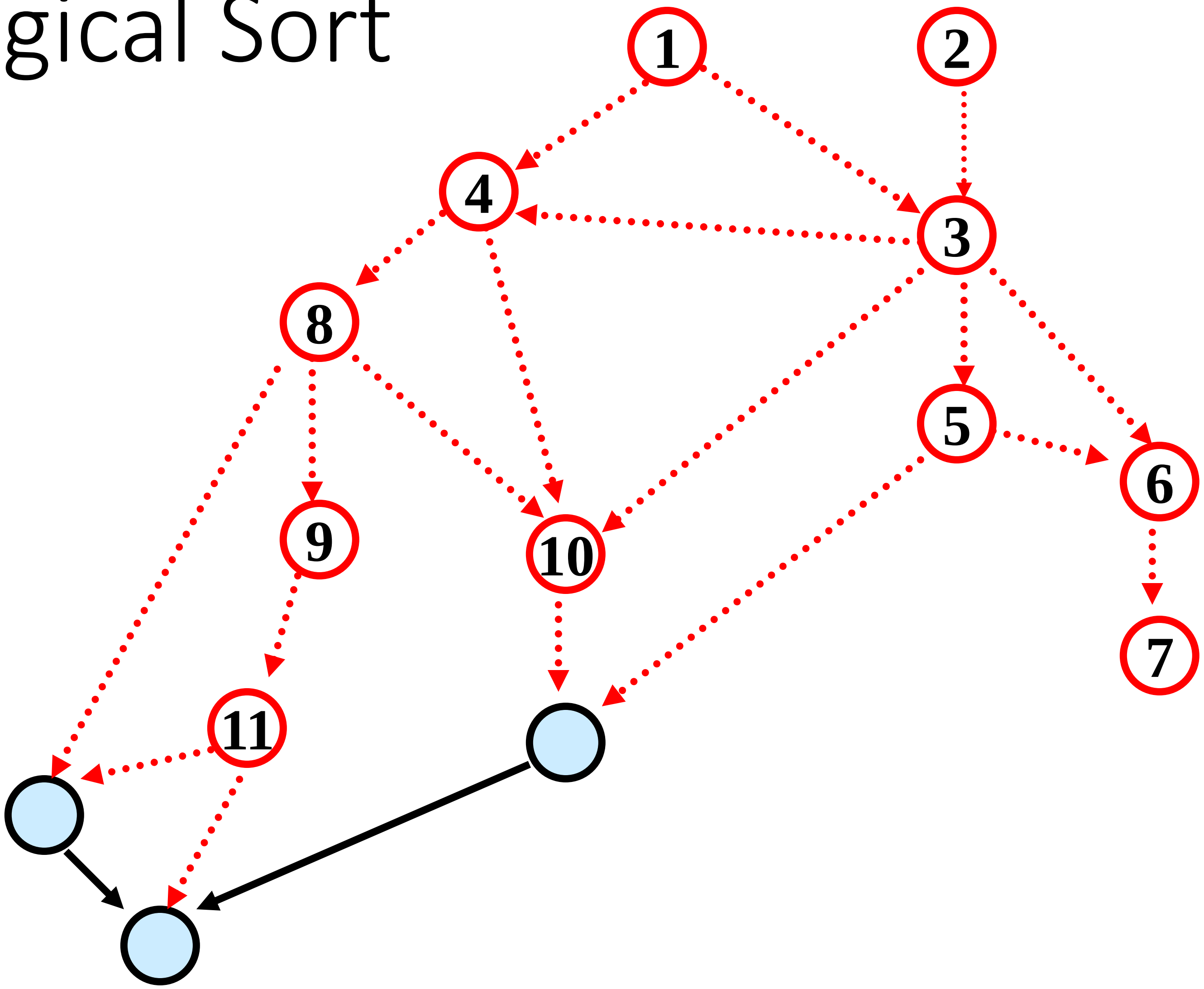
Topological Sort



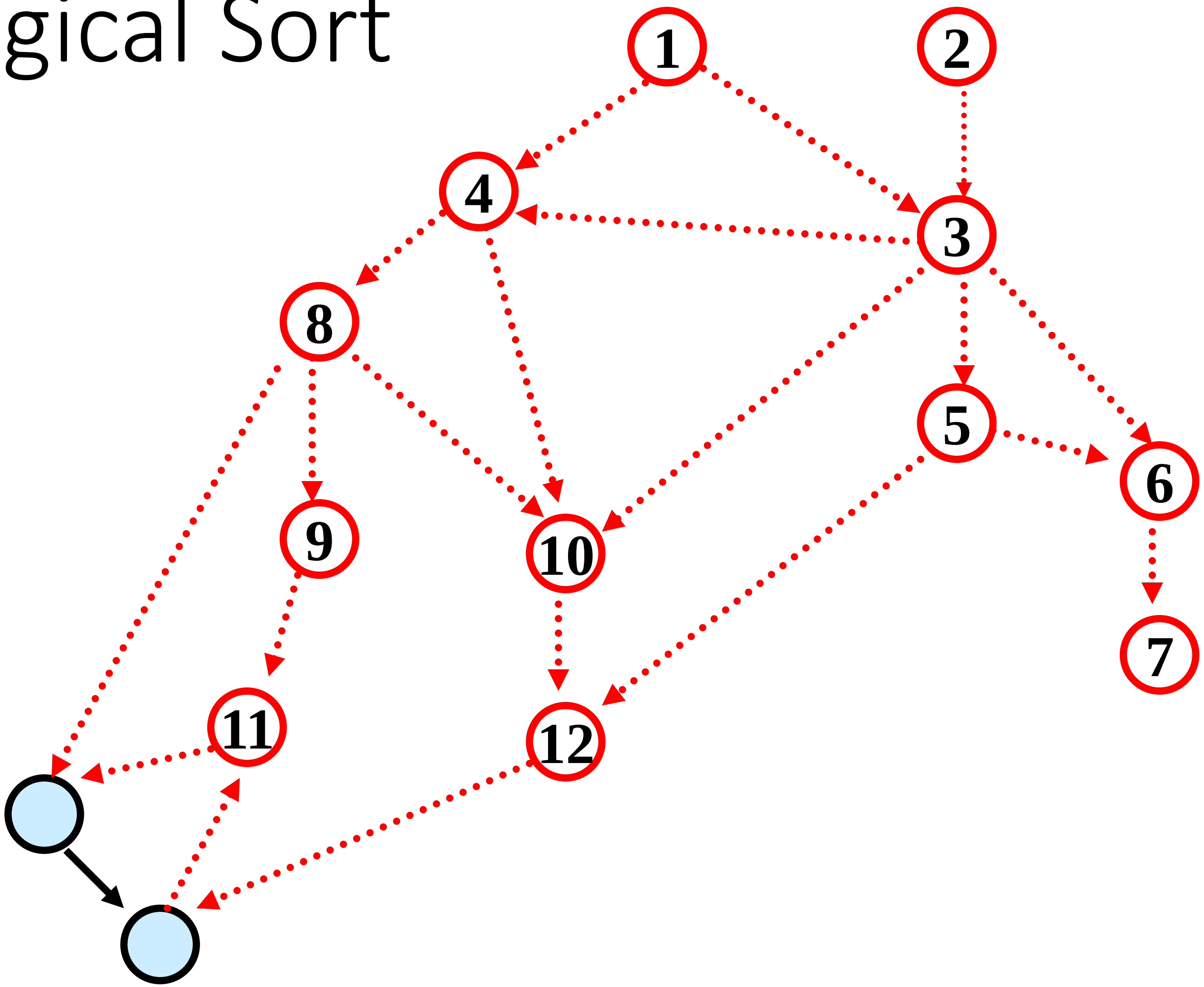
Topological Sort



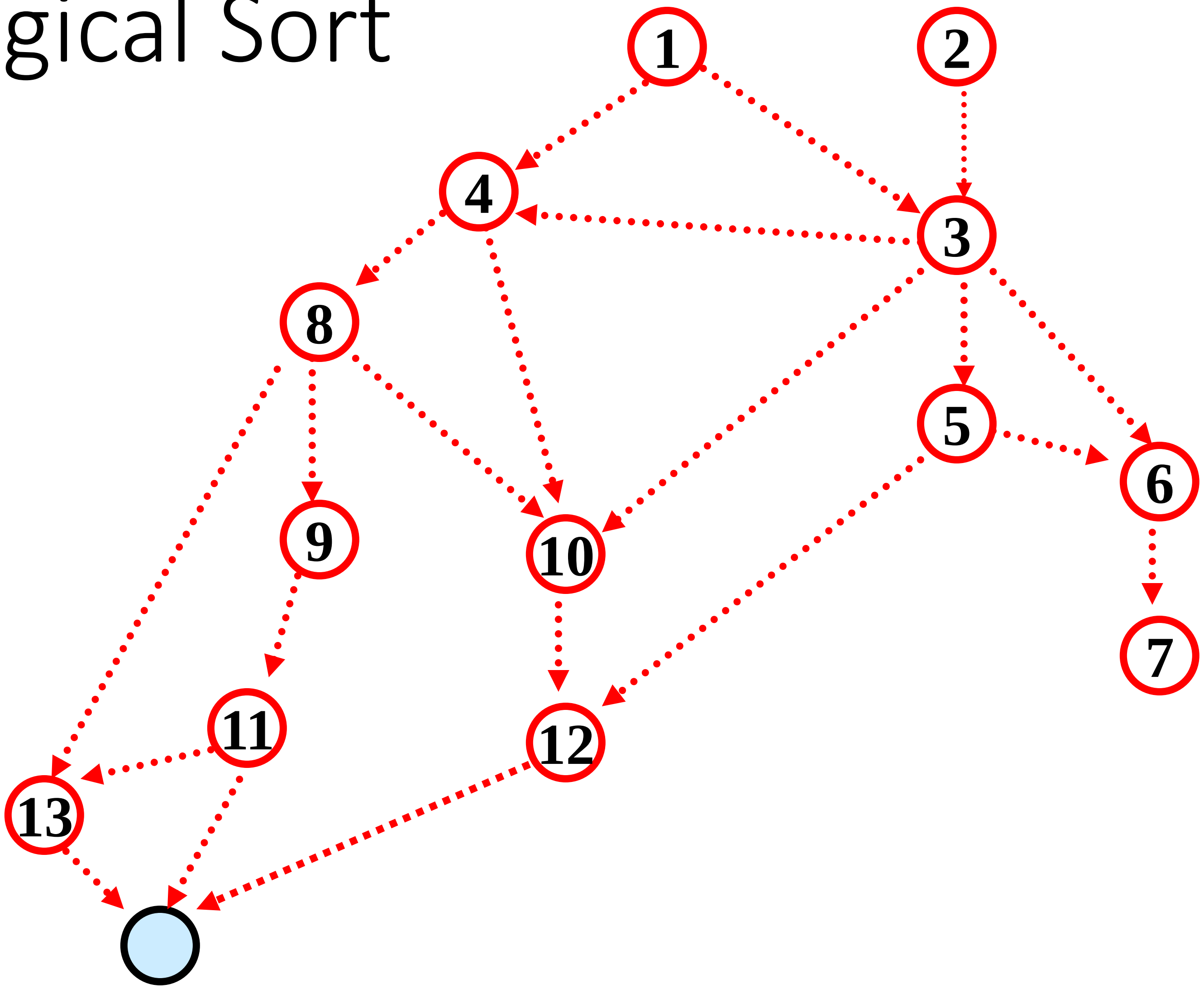
Topological Sort



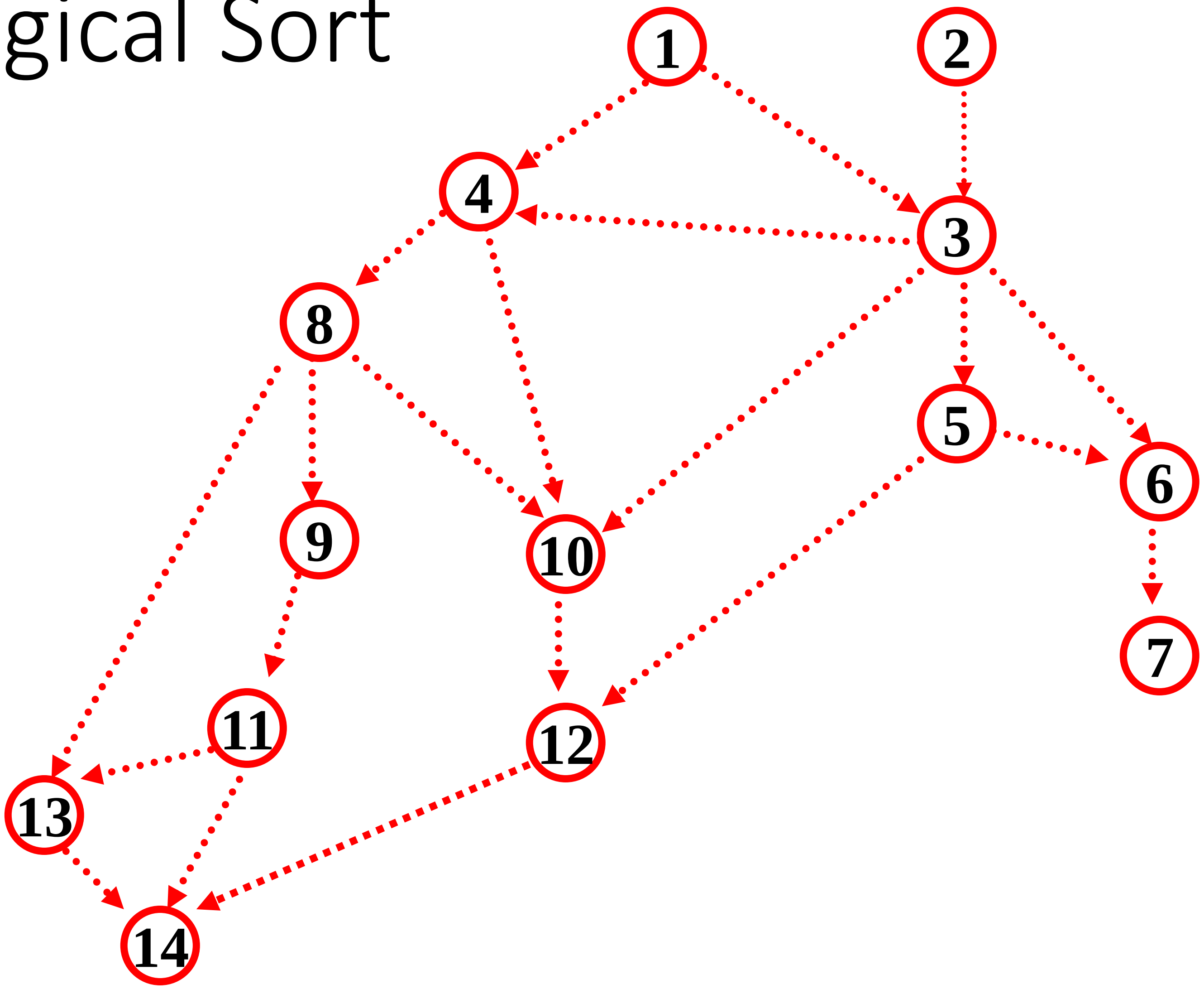
Topological Sort



Topological Sort



Topological Sort



Implementing Topological Sort

- Go through all edges, computing array with in-degree for each vertex
  $O(m+n)$
- Maintain a list of vertices of in-degree $\mathbf{0}$
- Remove any vertex in list and number it
- When a vertex is removed, decrease in-degree of each neighbor by $\mathbf{1}$ and add them to the list if their degree drops to $\mathbf{0}$

Total cost: $O(m+n)$

Strongly Connected Components of Directed Graphs

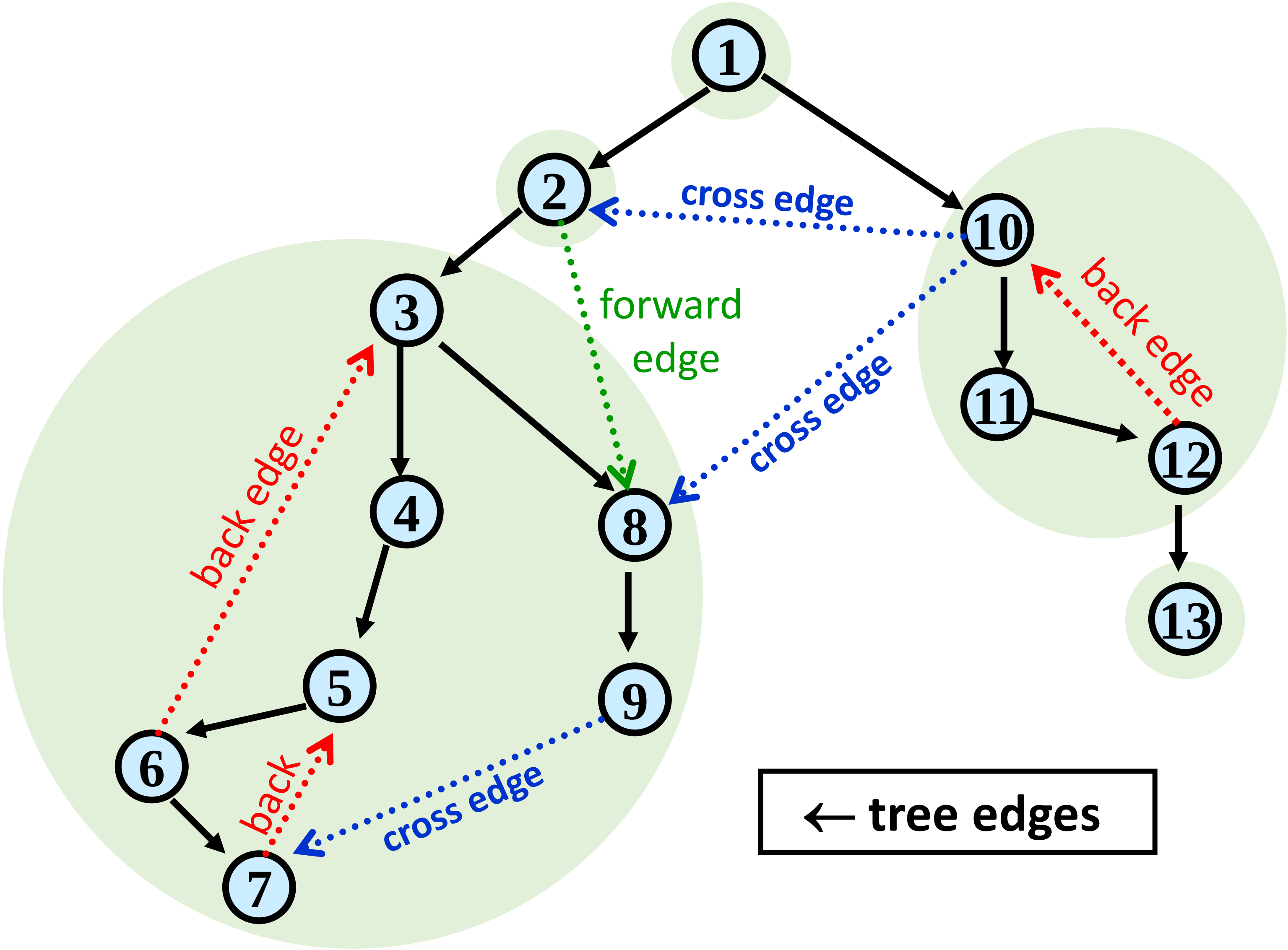
Defn: Vertices u and v are strongly connected iff they are on a directed cycle (there are paths from u to v and from v to u).

Defn: Can partition vertices of any directed graph into strongly connected components:

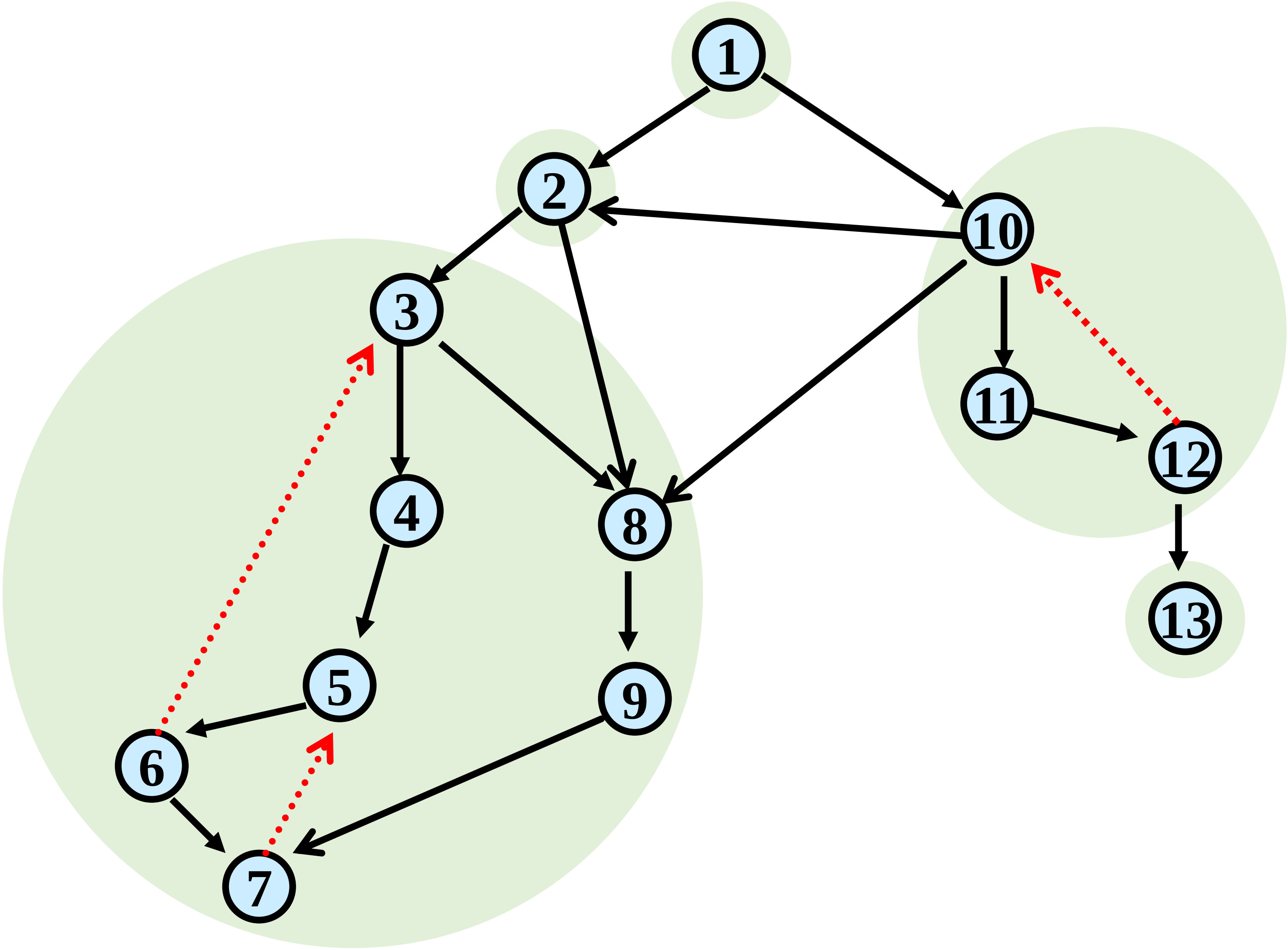
1. all pairs of vertices in the same component are strongly connected
2. can't merge components and keep property 1

- Strongly connected components can be stored efficiently just like connected components
- Can be found in $O(n+m)$ time using a DFS then a BFS
  - Do a depth-first sort, keeping track of the order nodes are marked "fully-explored"
  - Going in order from least recent to most recent, run connected components

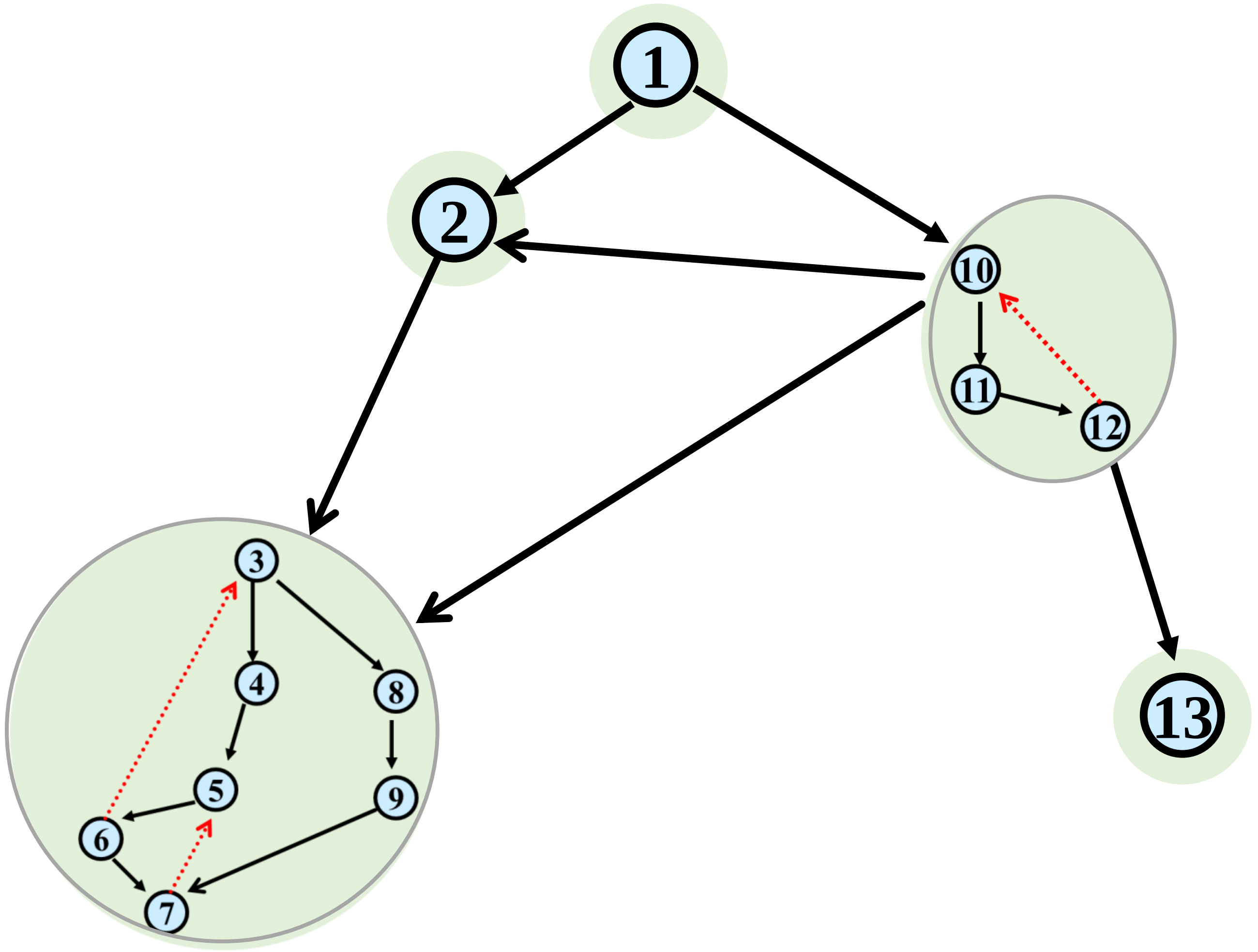
Strongly Connected Components



Strongly Connected Components



Strongly Connected Components



Strongly-Connected Components Usage

Common algorithmic paradigm for general directed graphs:

- Process strongly connected components one-by-one in the order given by topological sort of the DAG you get from shrinking them.