CS 106X, Lecture 25
Topological Sort

reading:

Programming Abstractions in C++, Chapter 18

Plan For This Week

- Graphs: Topological Sort (HW8)
- Classes: Inheritance and Polymorphism (HW8)
- Sorting Algorithms

Plan For Today

- Topological Sort
  - Kahn's Algorithm
  - Recursive DFS
  - Spreadsheets
- Announcements

- **Learning Goal:** understand how to implement topological sort, and why it and dependency graphs are useful.

Plan for Today

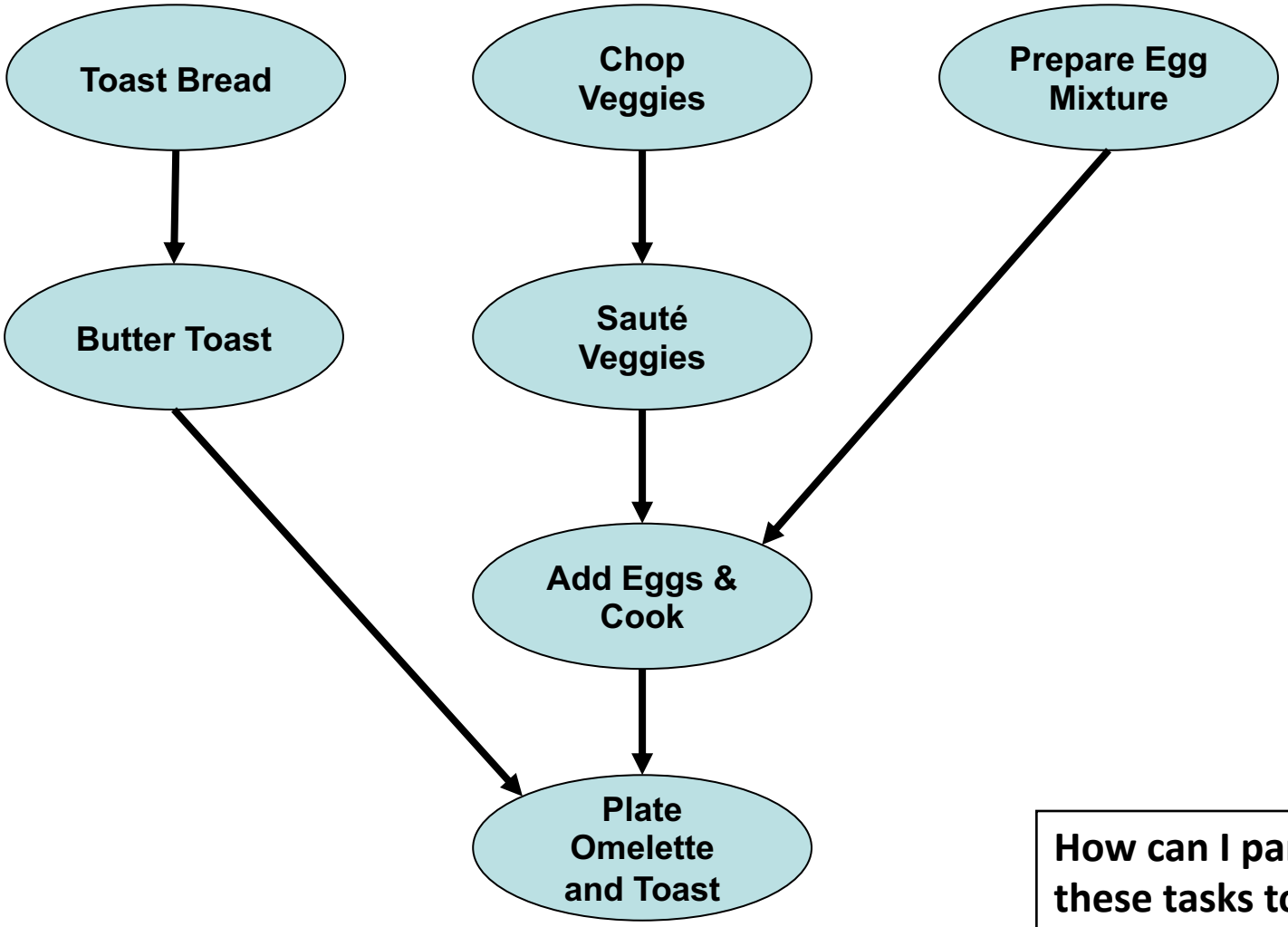
- **Topological Sort**
  - Kahn's Algorithm
  - Recursive DFS
  - Spreadsheets
- Announcements

Graphs So Far

- Graphs as Data Structures
  - Adjacency List
  - Adjacency Matrix
  - Edge List
- Searching
  - DFS
  - BFS
  - Dijkstra's Algorithm
  - A* Search

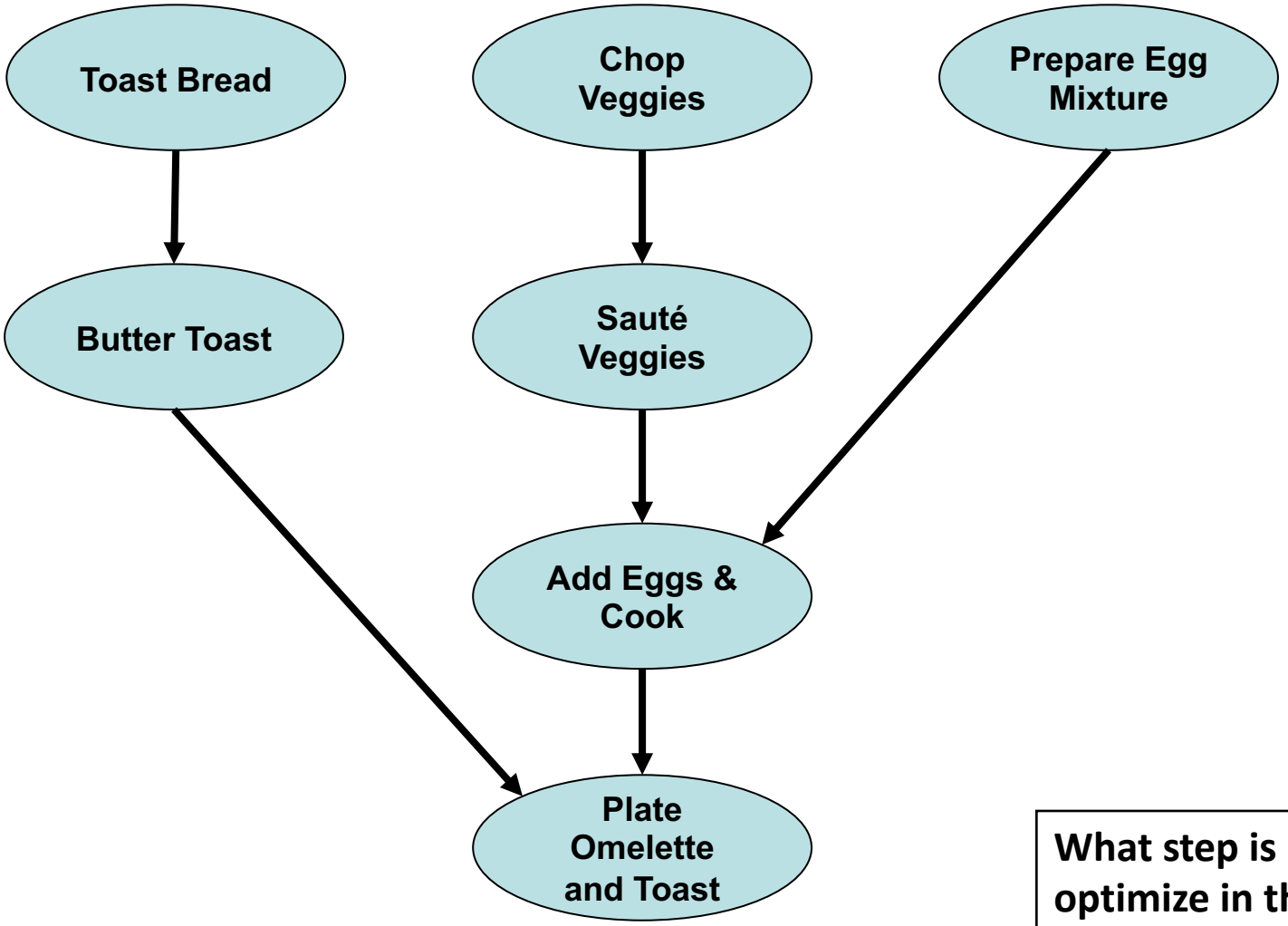
Dependency Graphs

Dependency Graphs: Tasks



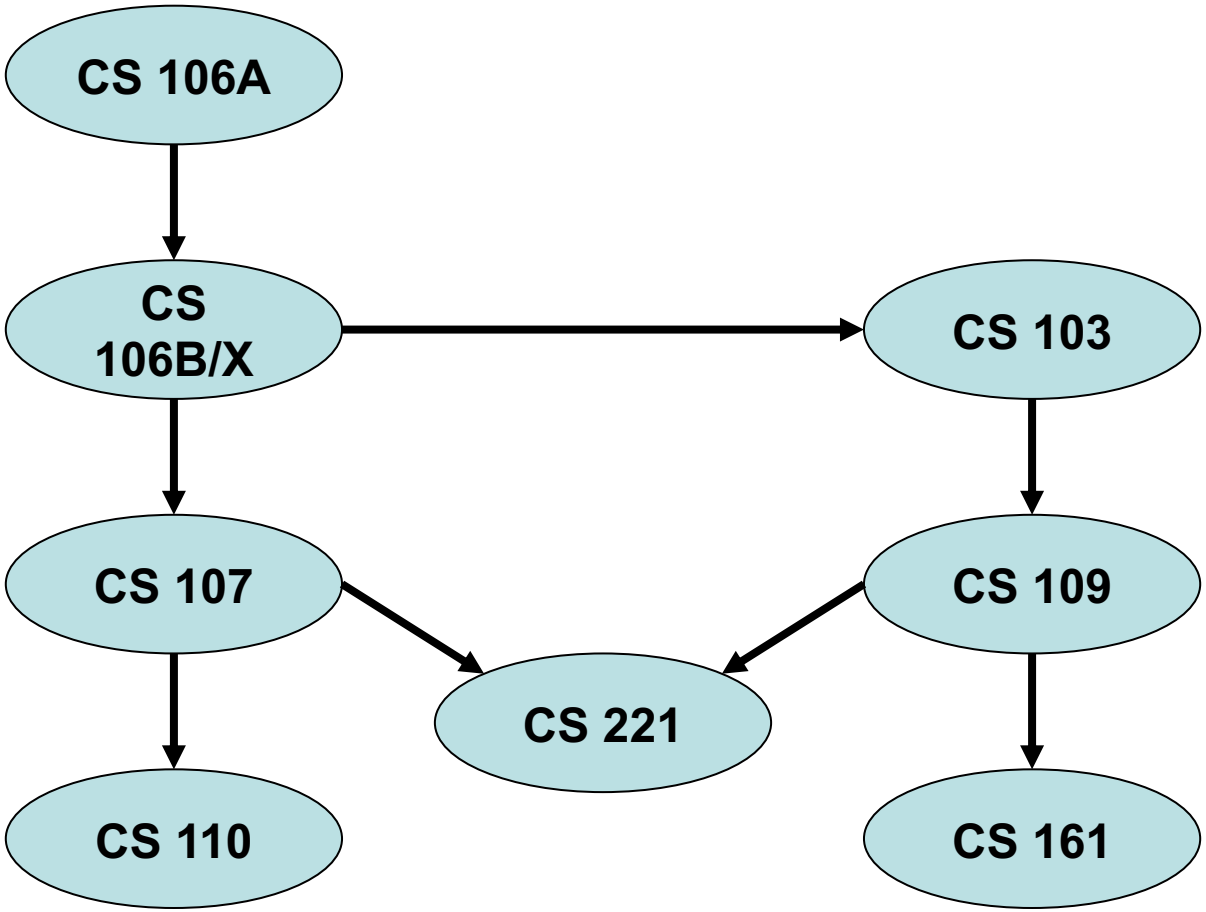
How can I parallelize these tasks to finish more quickly?

Dependency Graphs: Tasks



What step is best to optimize in this production process?

Dependency Graphs: Orderings

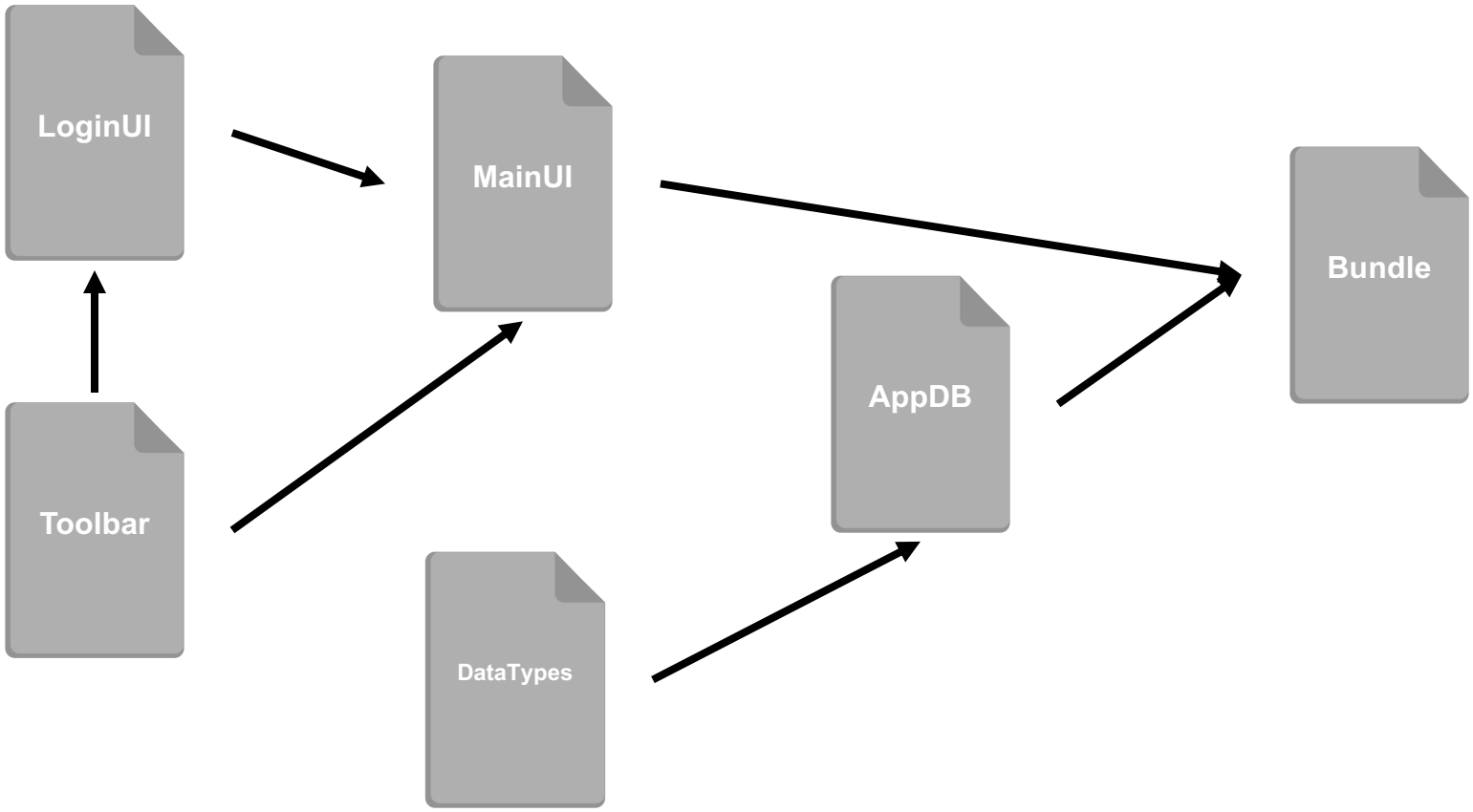


What are valid orderings of these tasks?

Dependency Graphs: Games

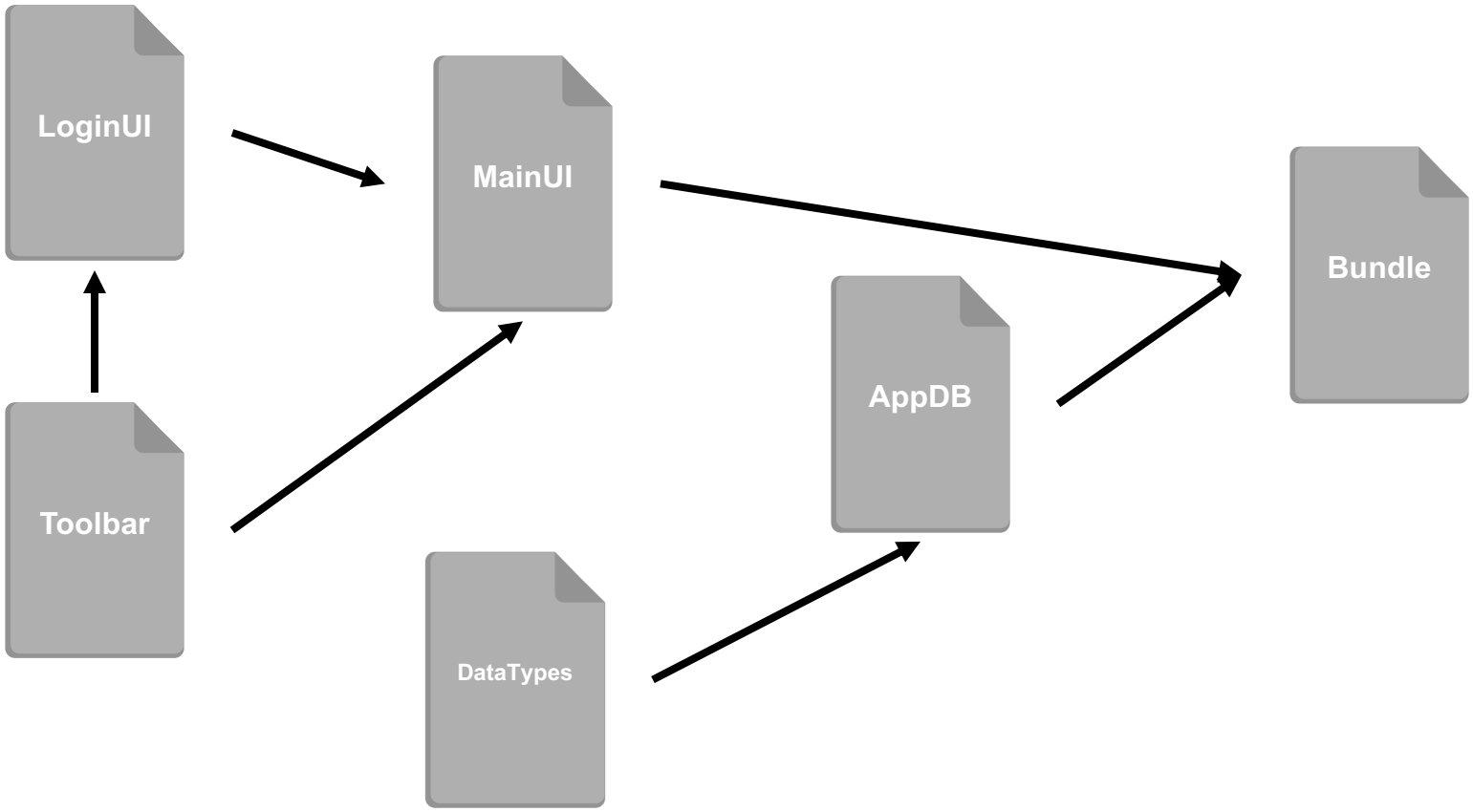
What is an effective AI strategy for resource management games (like Civilization)?

Dependency Graphs: Compilers



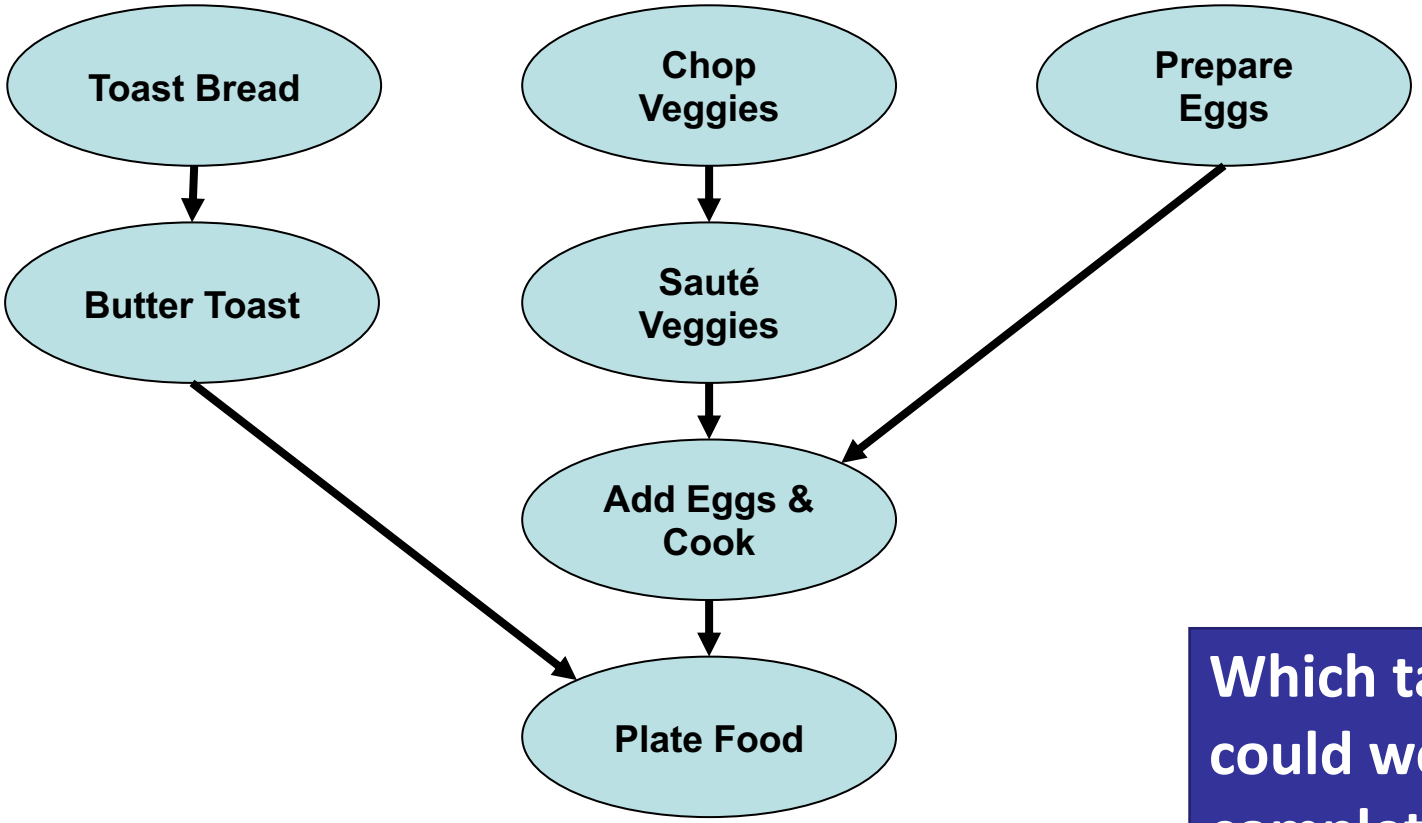
In what order should we recompile source files with dependencies?

Dependency Graphs: Web



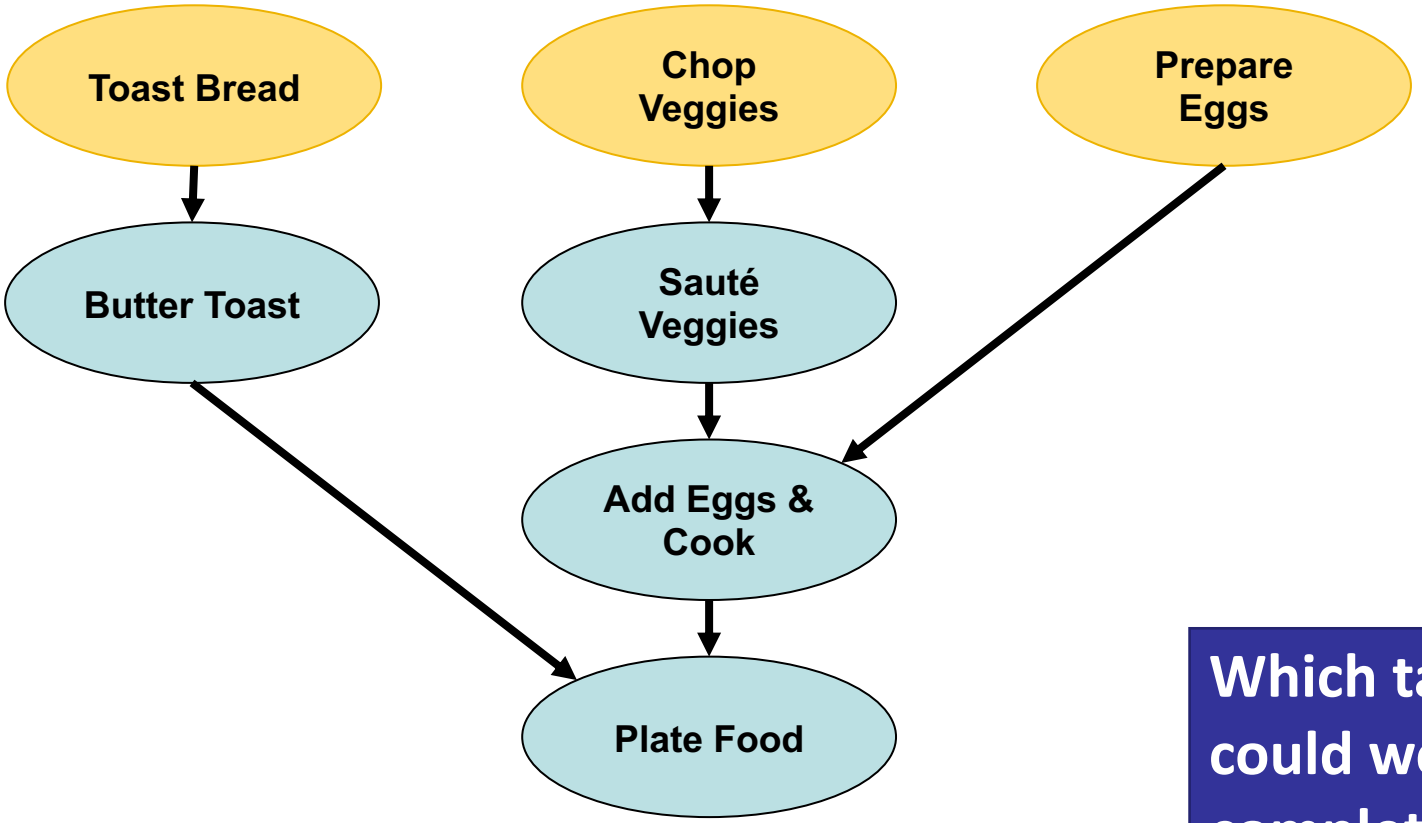
How can I efficiently package up and transmit web dependencies?

Topological Sort



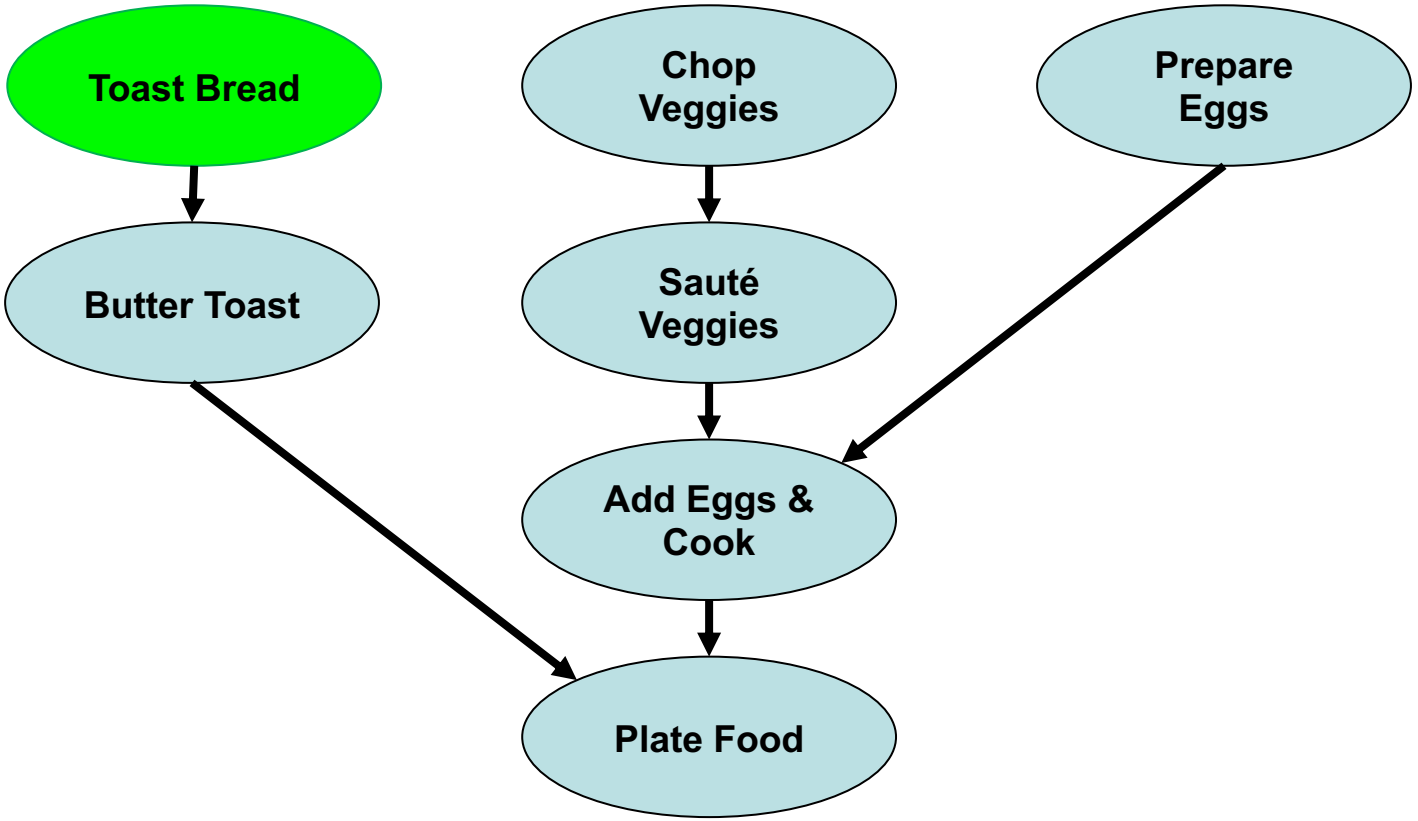
Which tasks could we complete first?

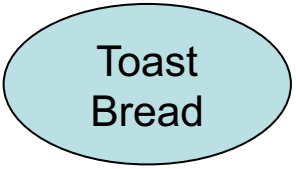
Topological Sort



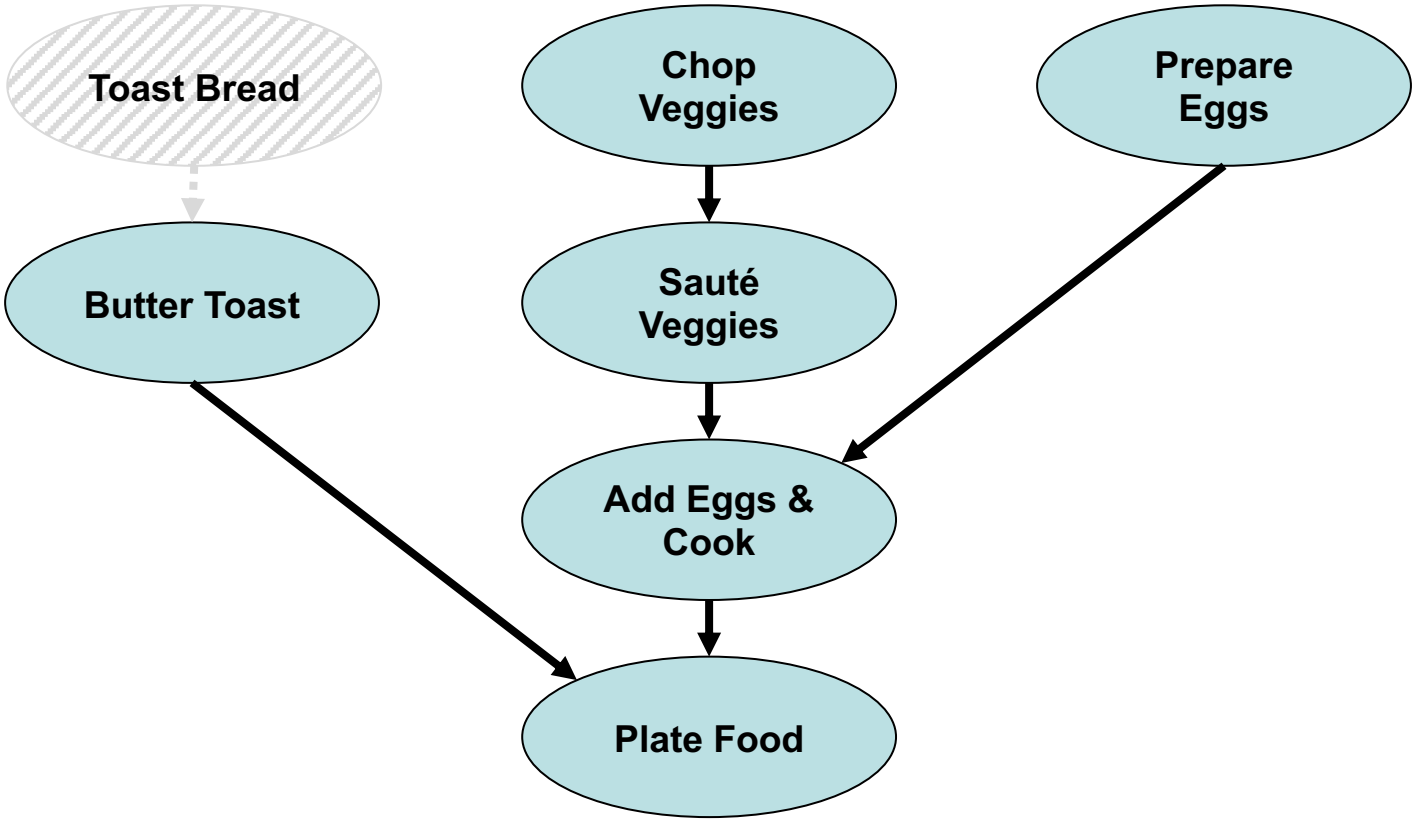
Which tasks could we complete first?

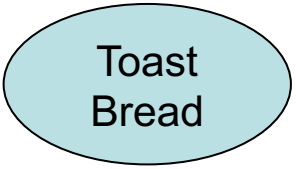
Topological Sort



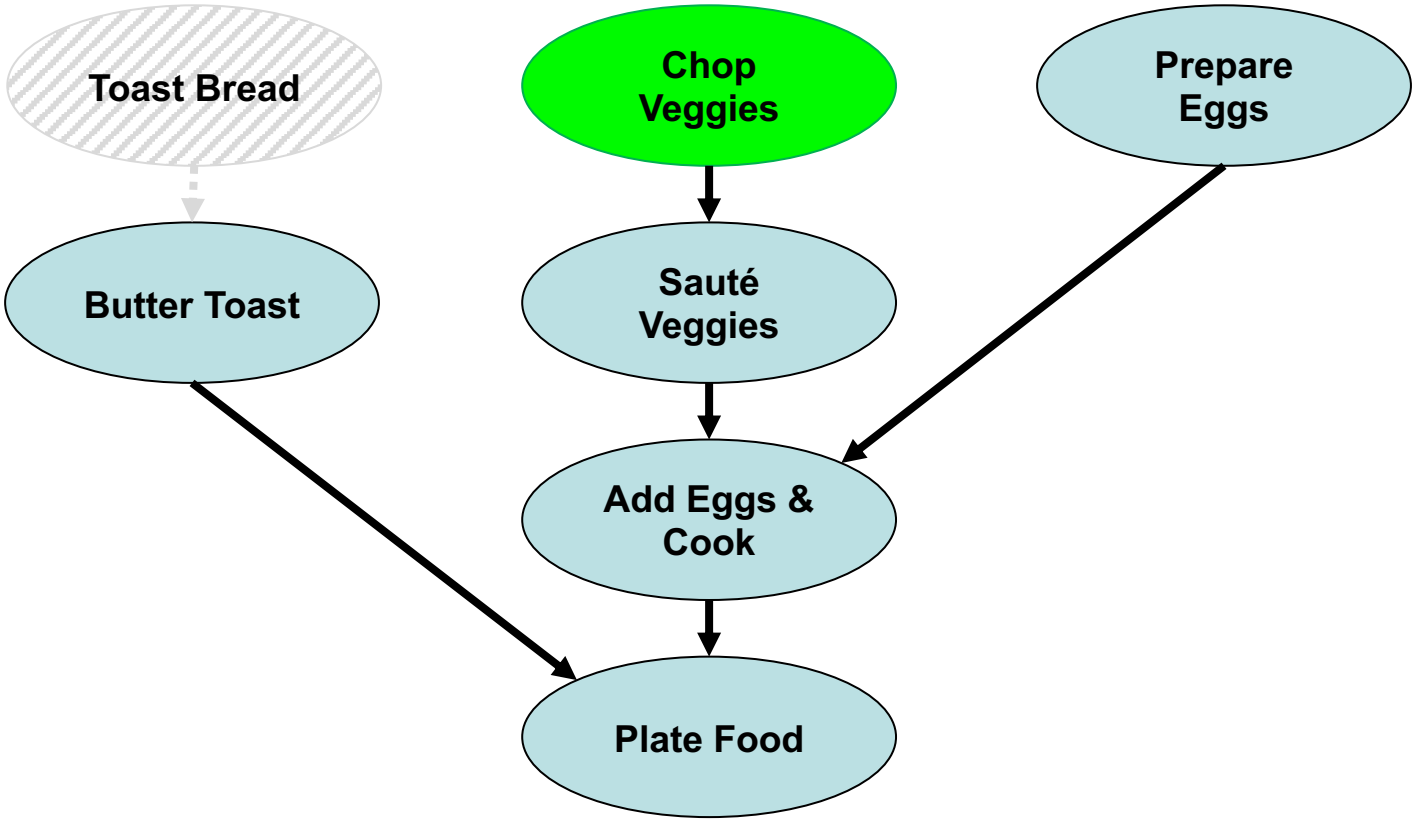


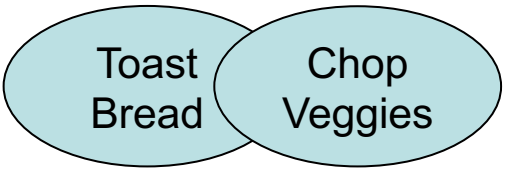
Topological Sort



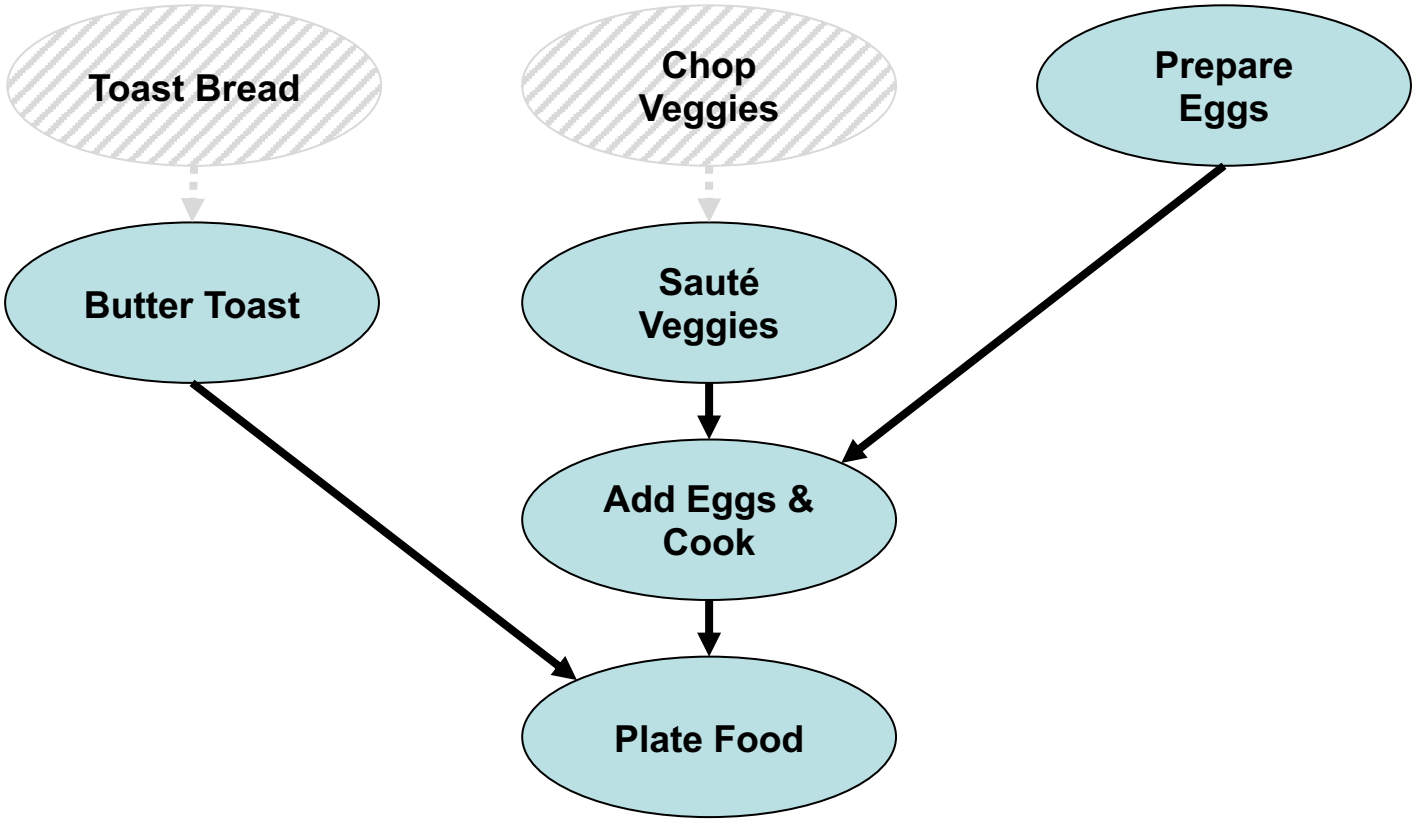


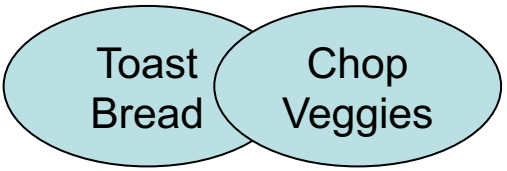
Topological Sort



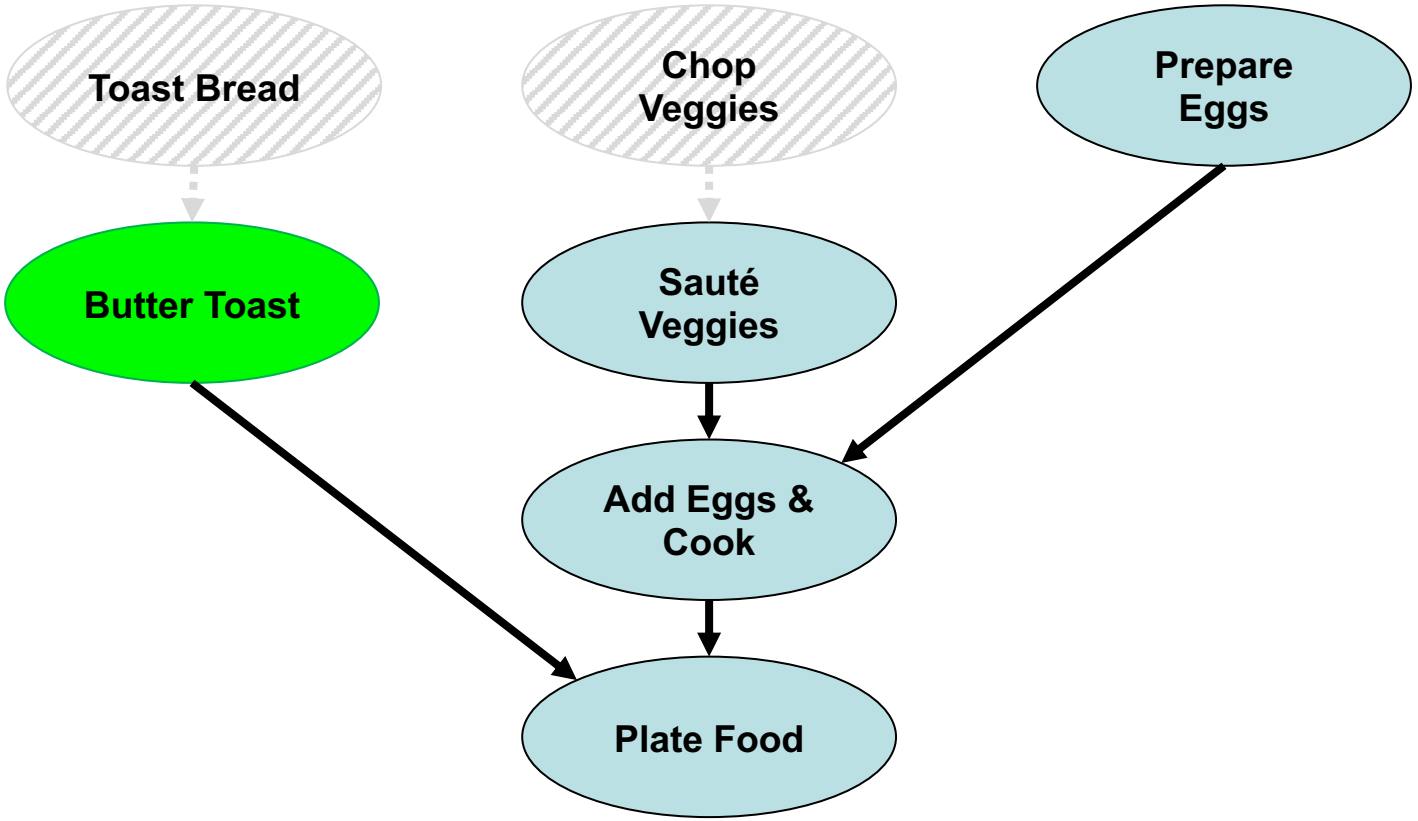


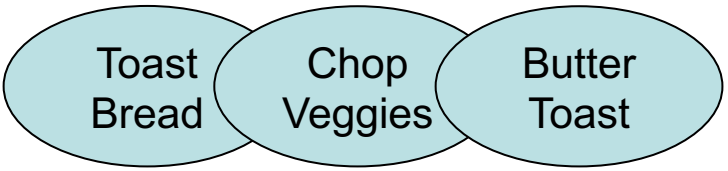
Topological Sort



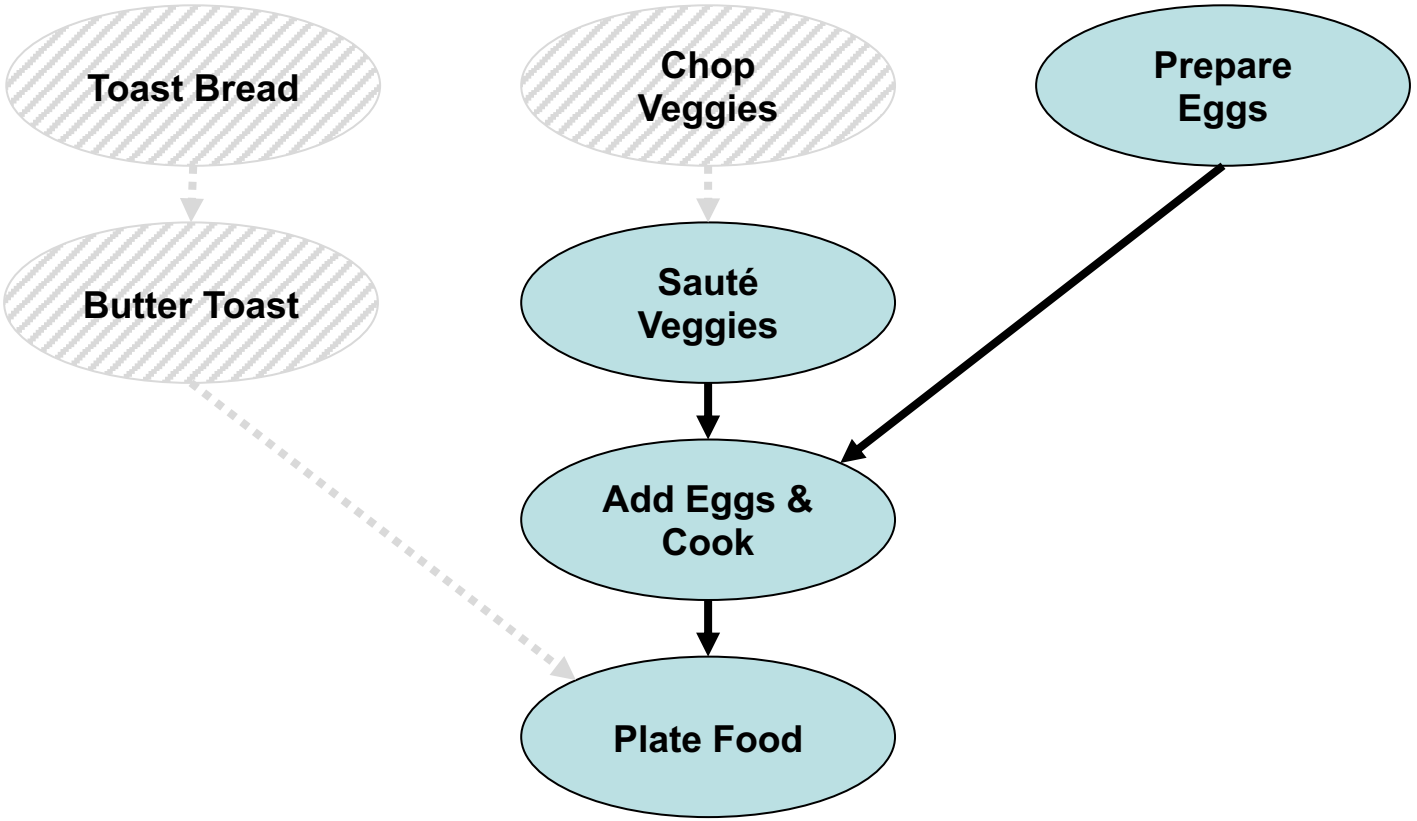


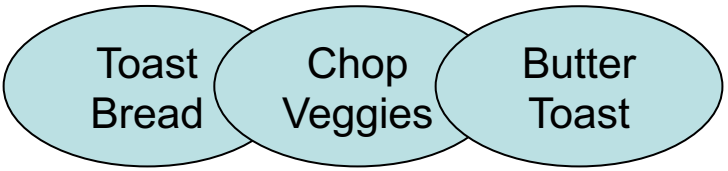
Topological Sort



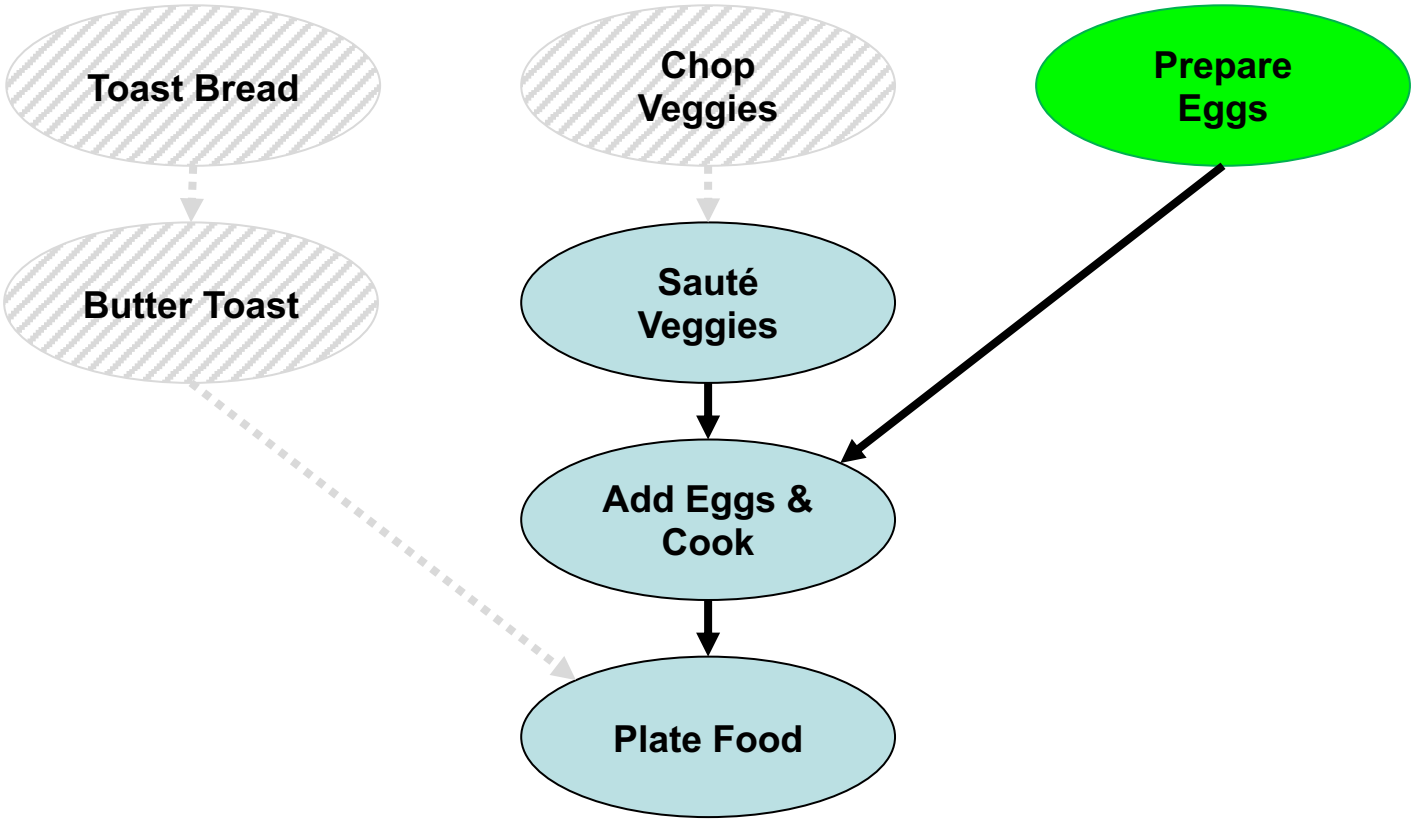


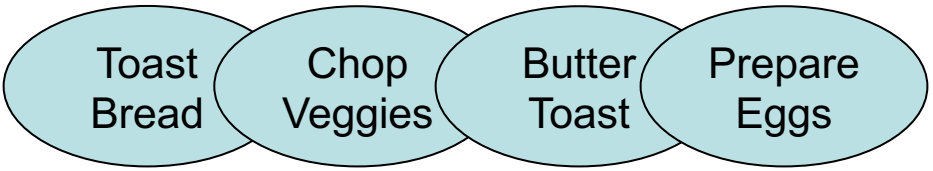
Topological Sort



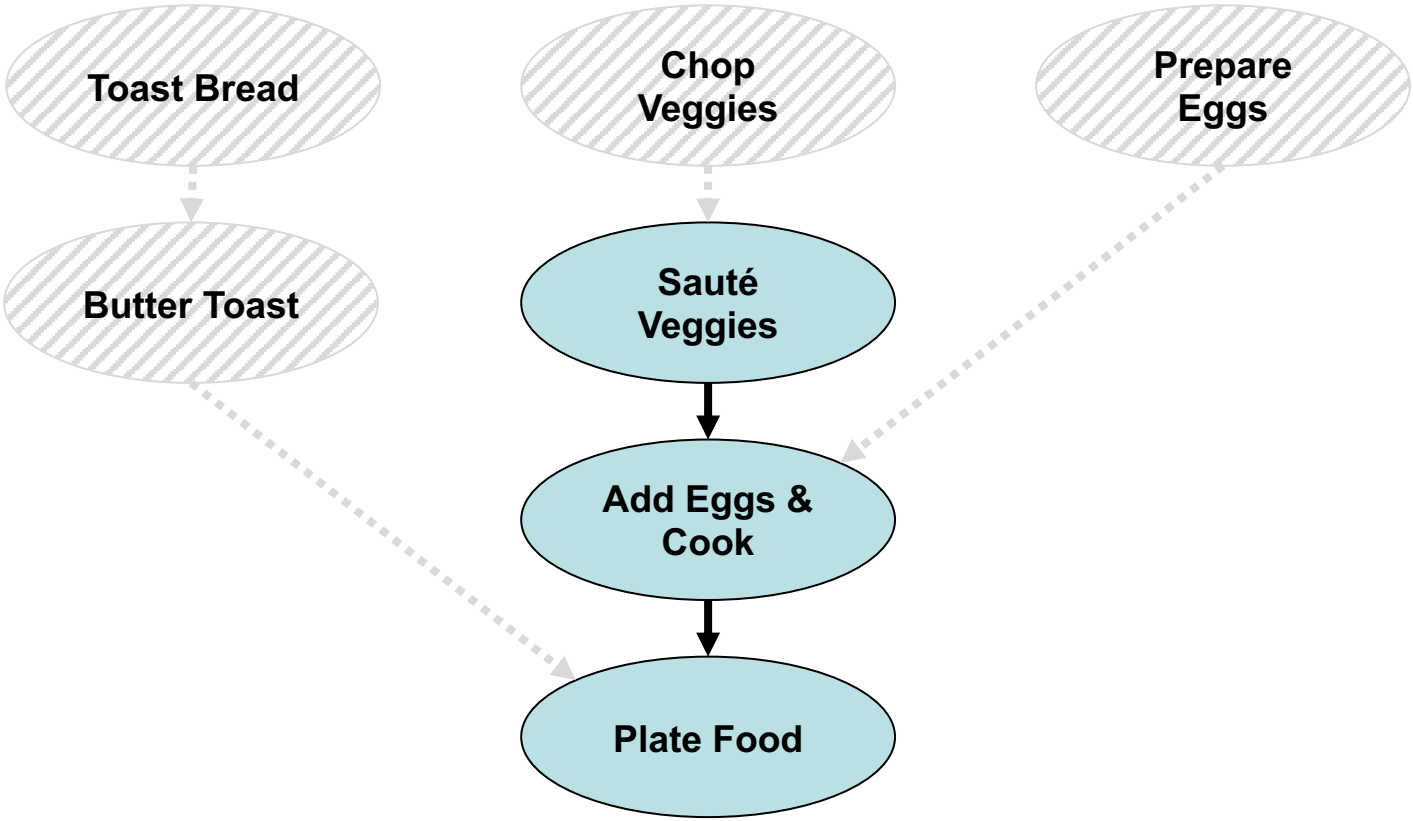


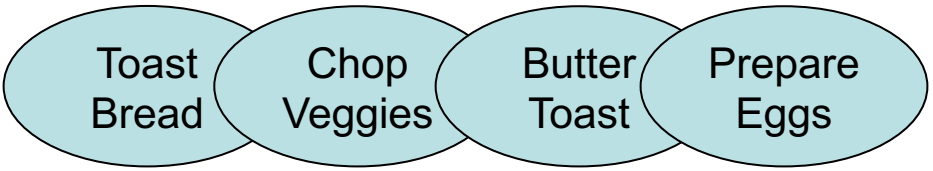
Topological Sort



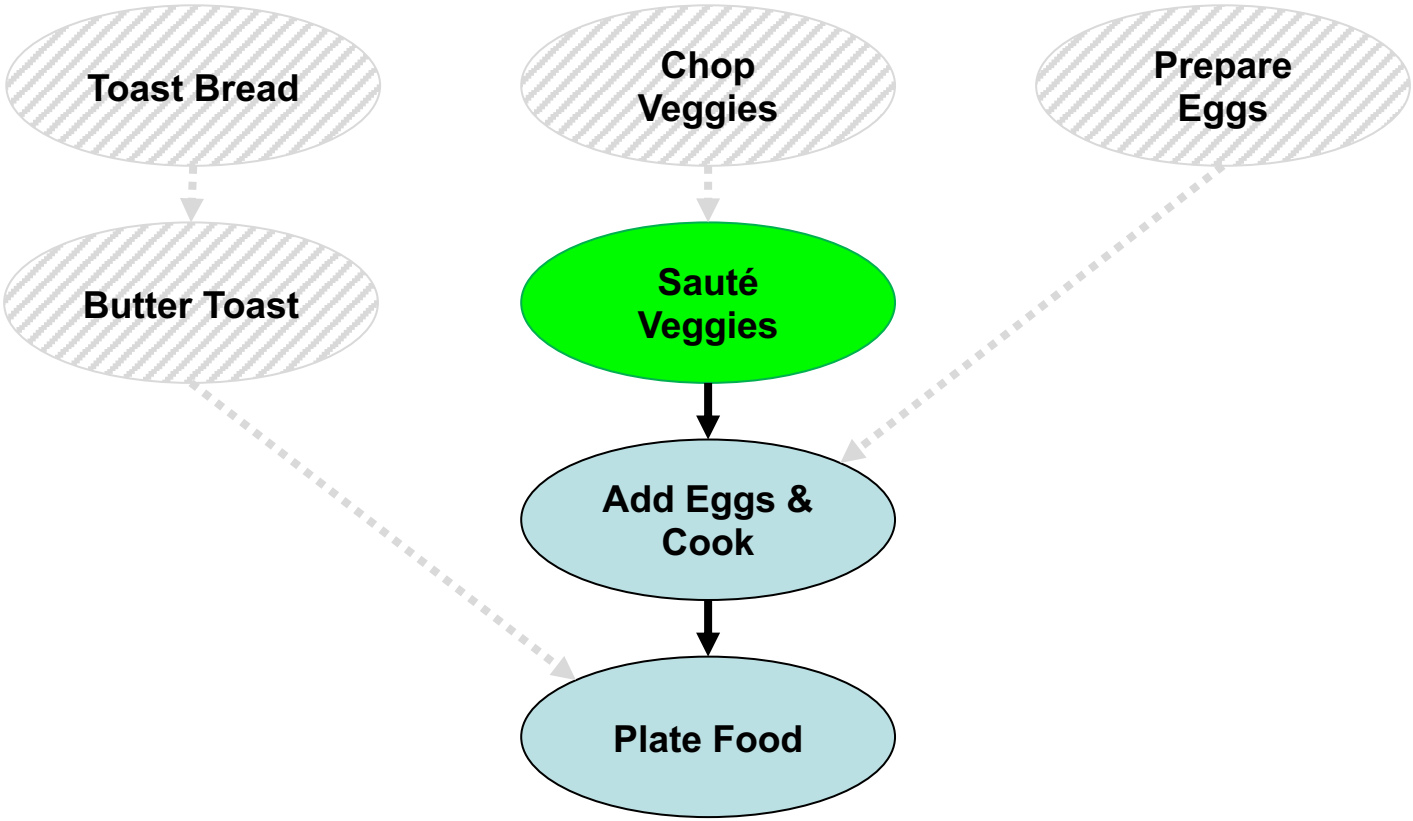


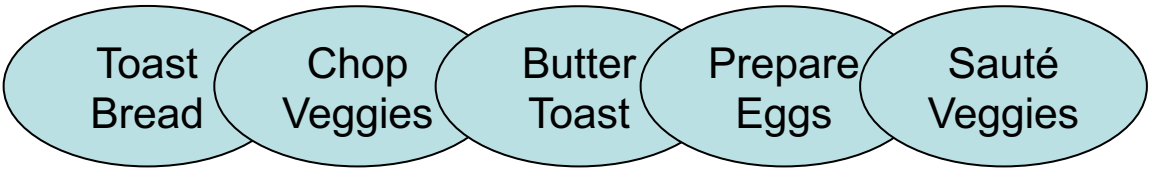
Topological Sort



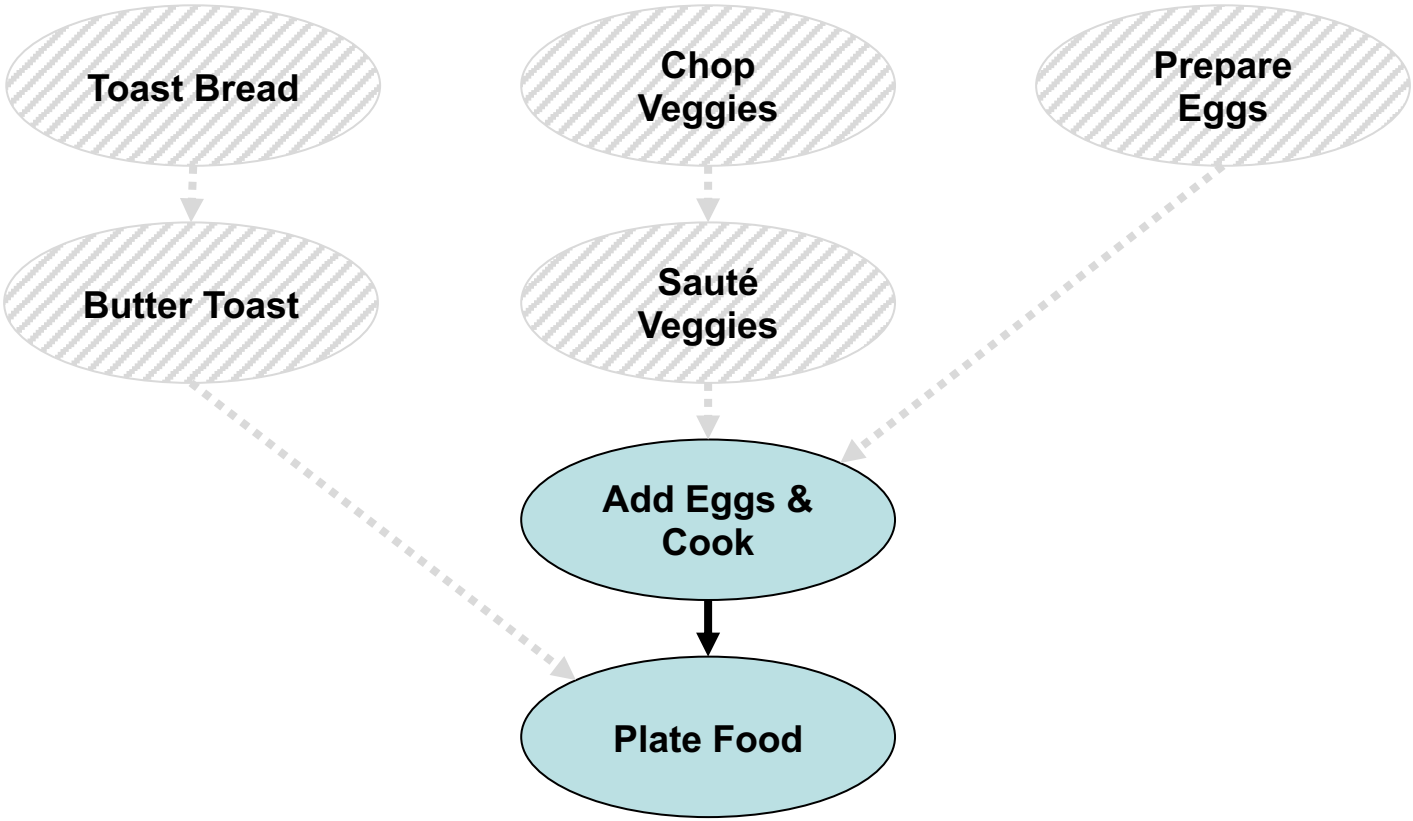


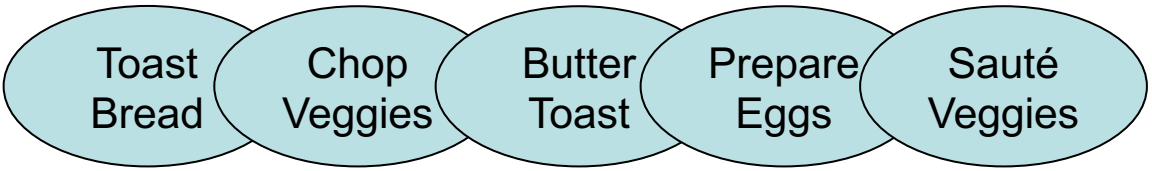
Topological Sort



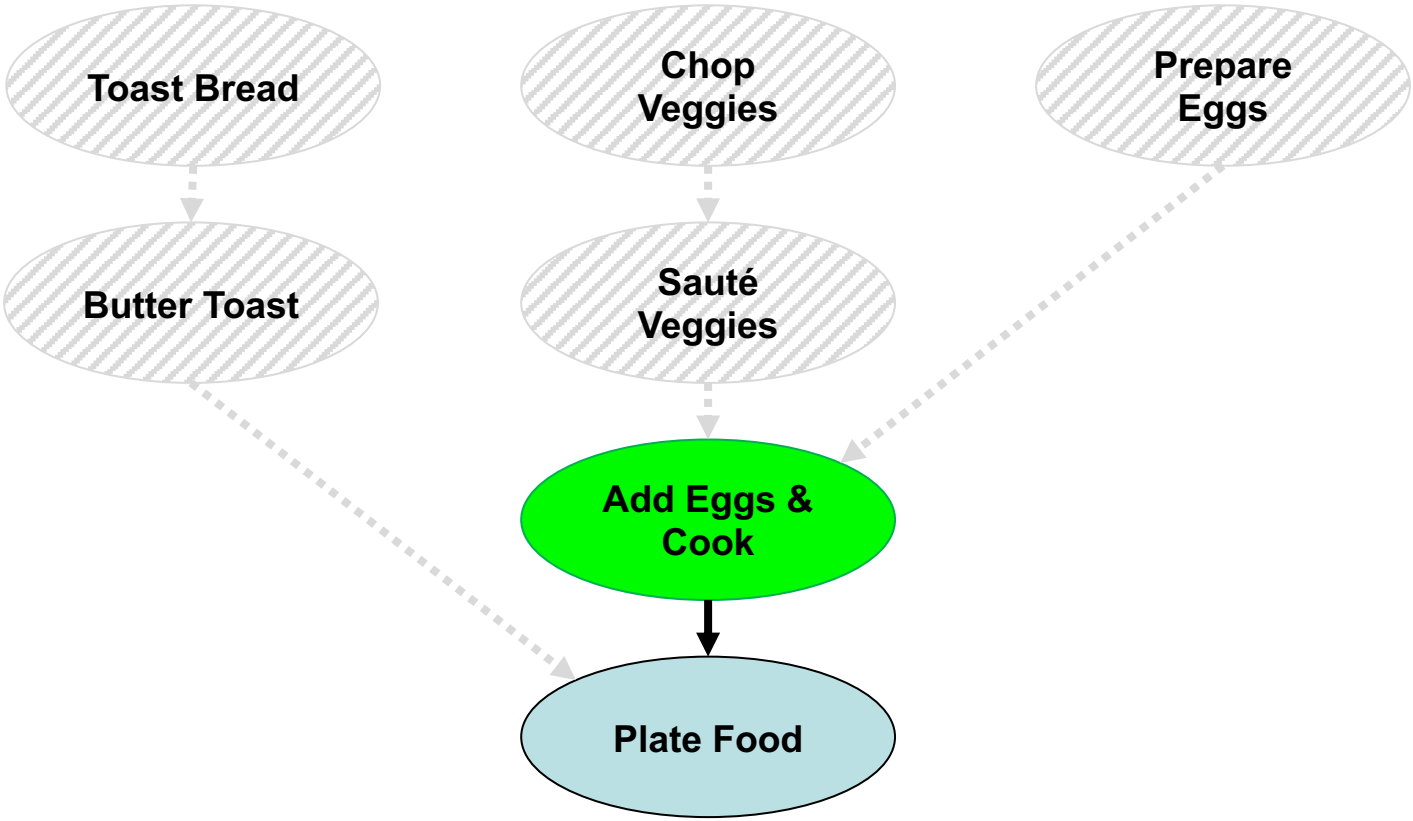


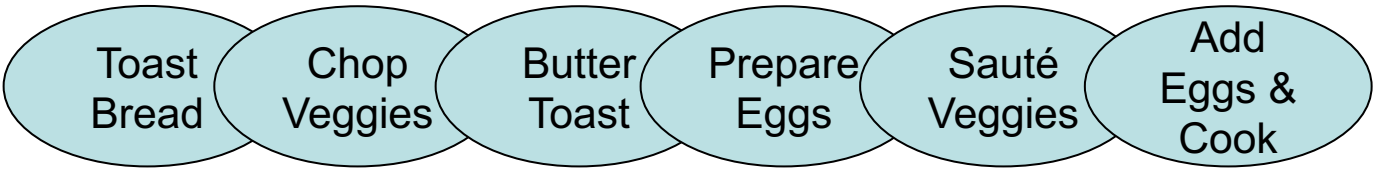
Topological Sort



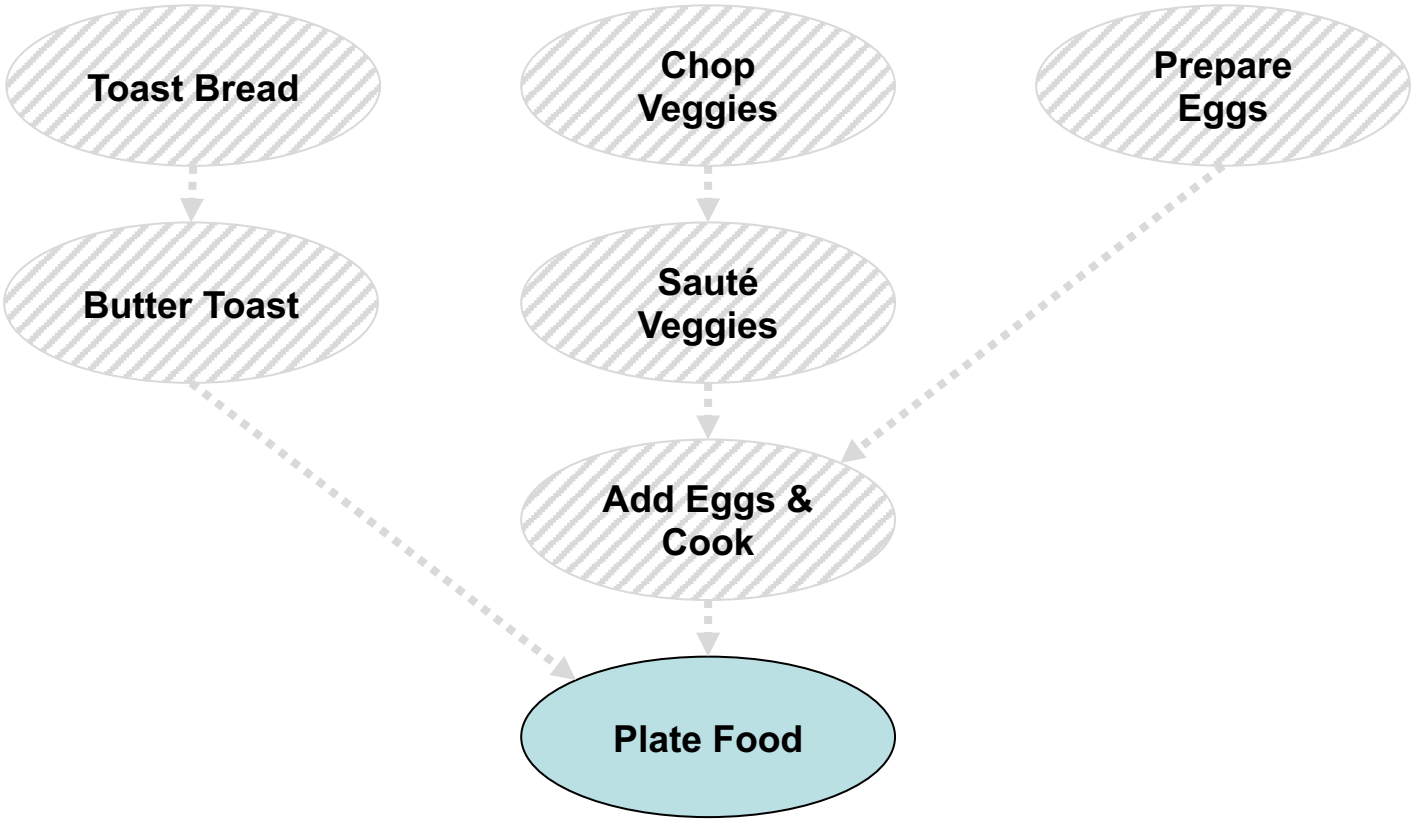


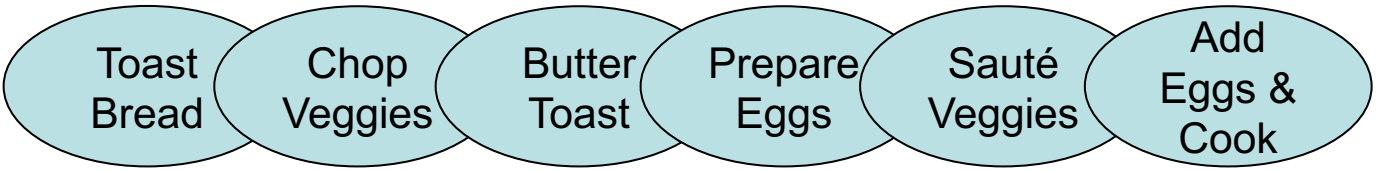
Topological Sort



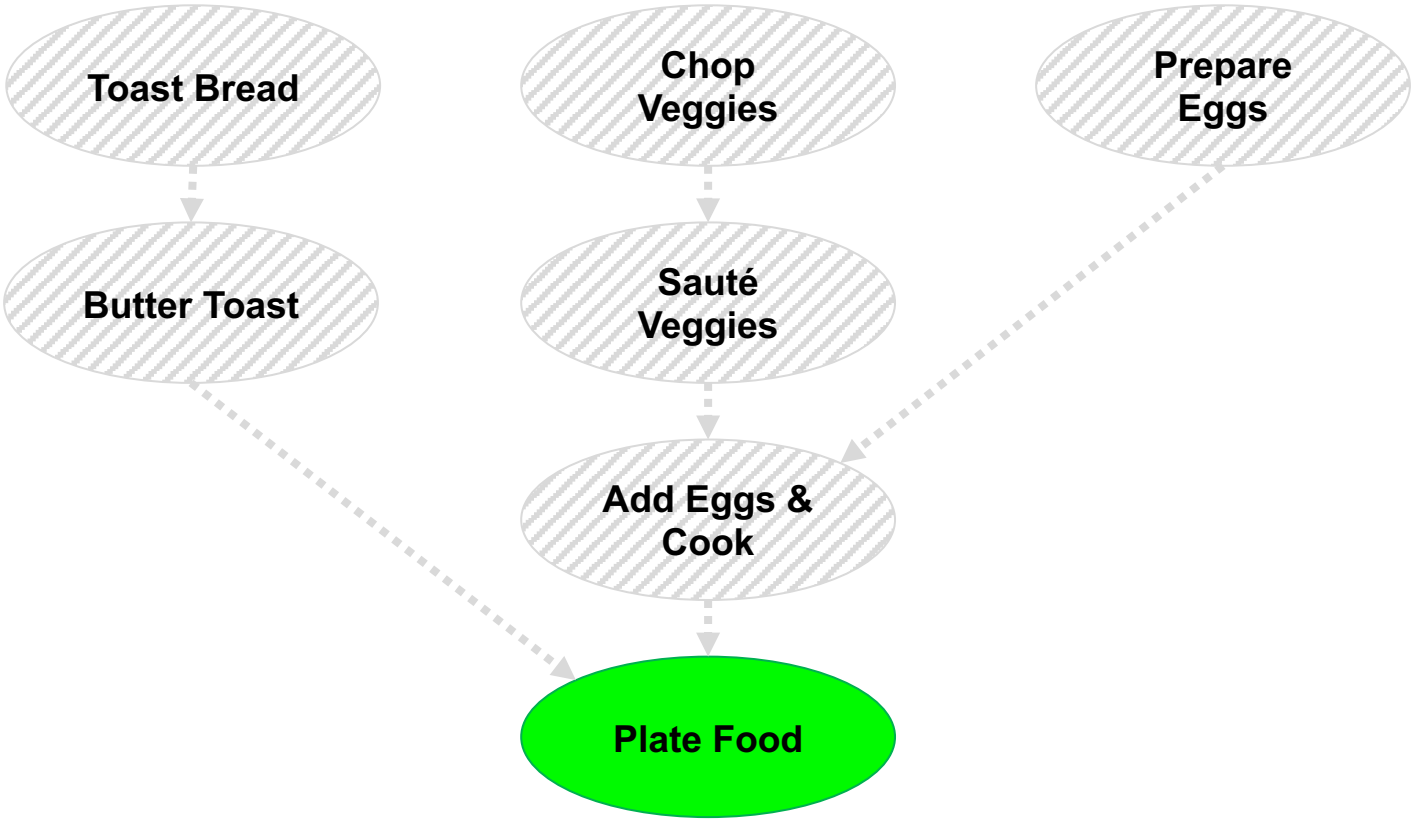


Topological Sort



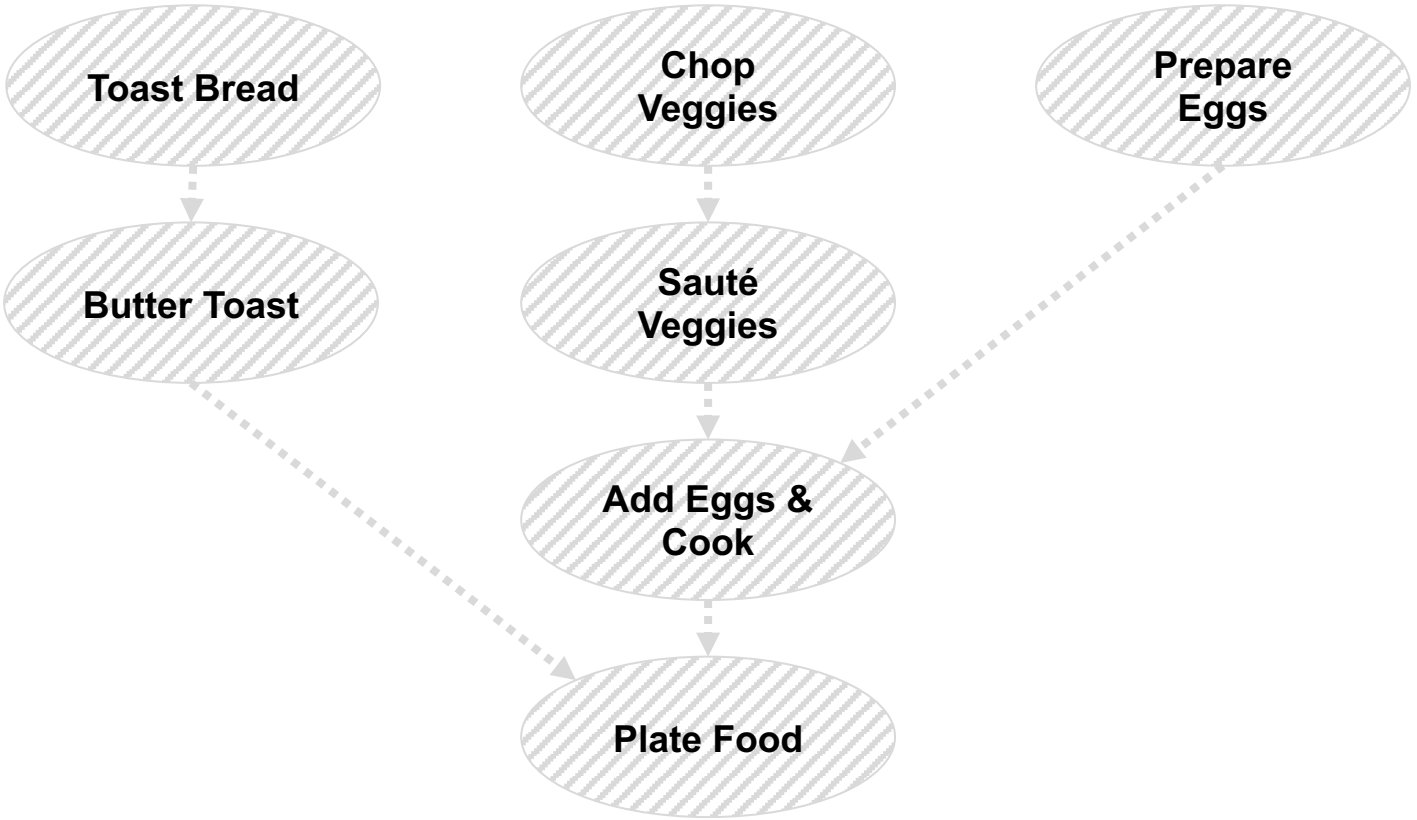


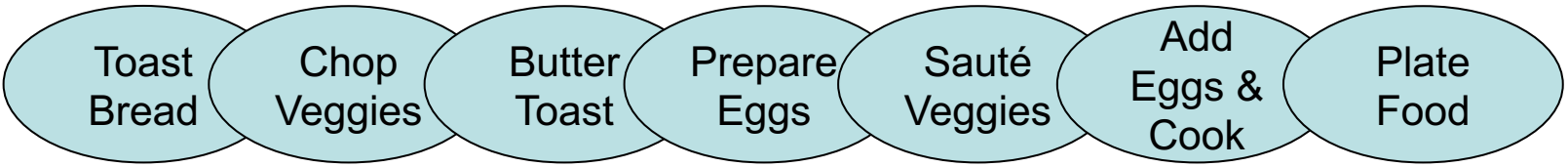
Topological Sort



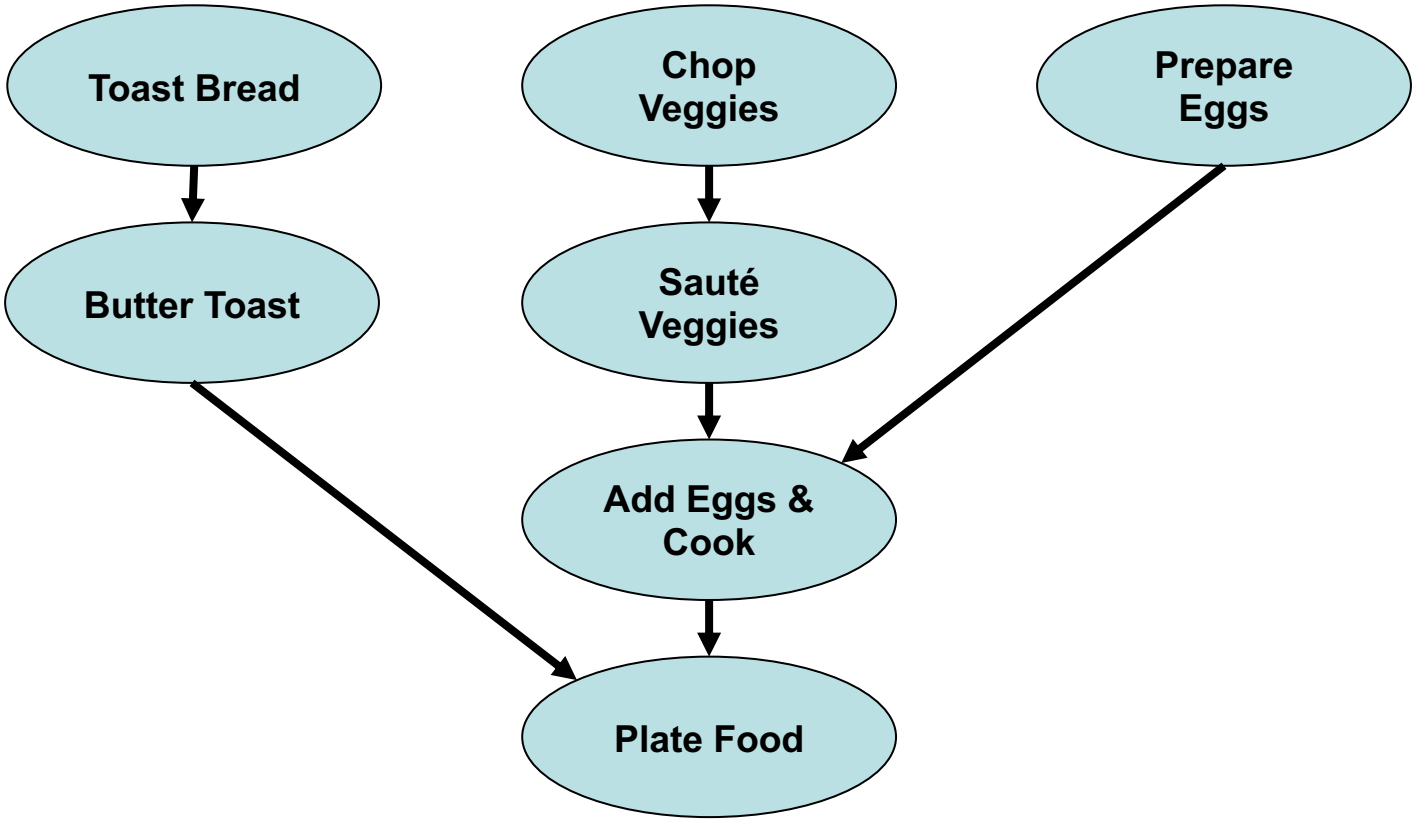
Toast Bread, Chop Veggies, Butter Toast, Prepare Eggs, Sauté Veggies, Add Eggs & Cook, Plate Food

Topological Sort





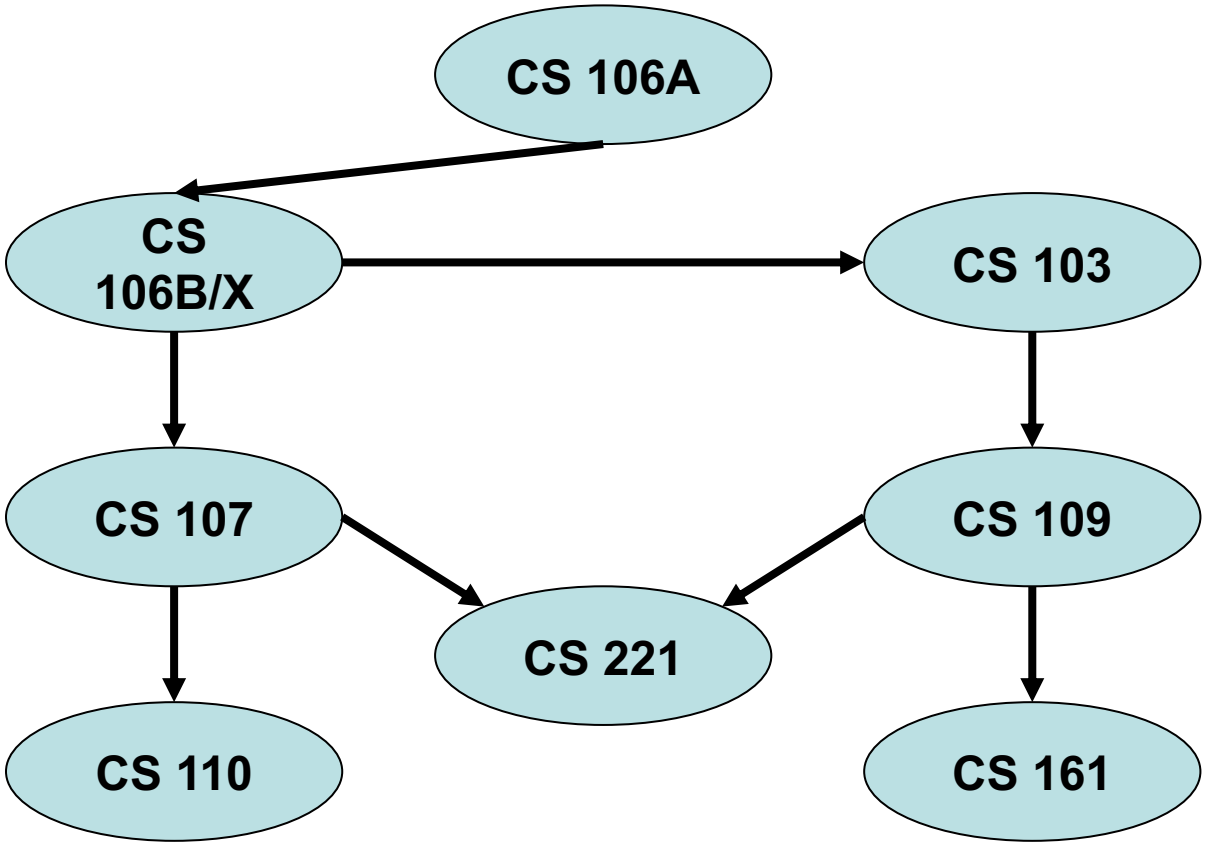
Topological Sort



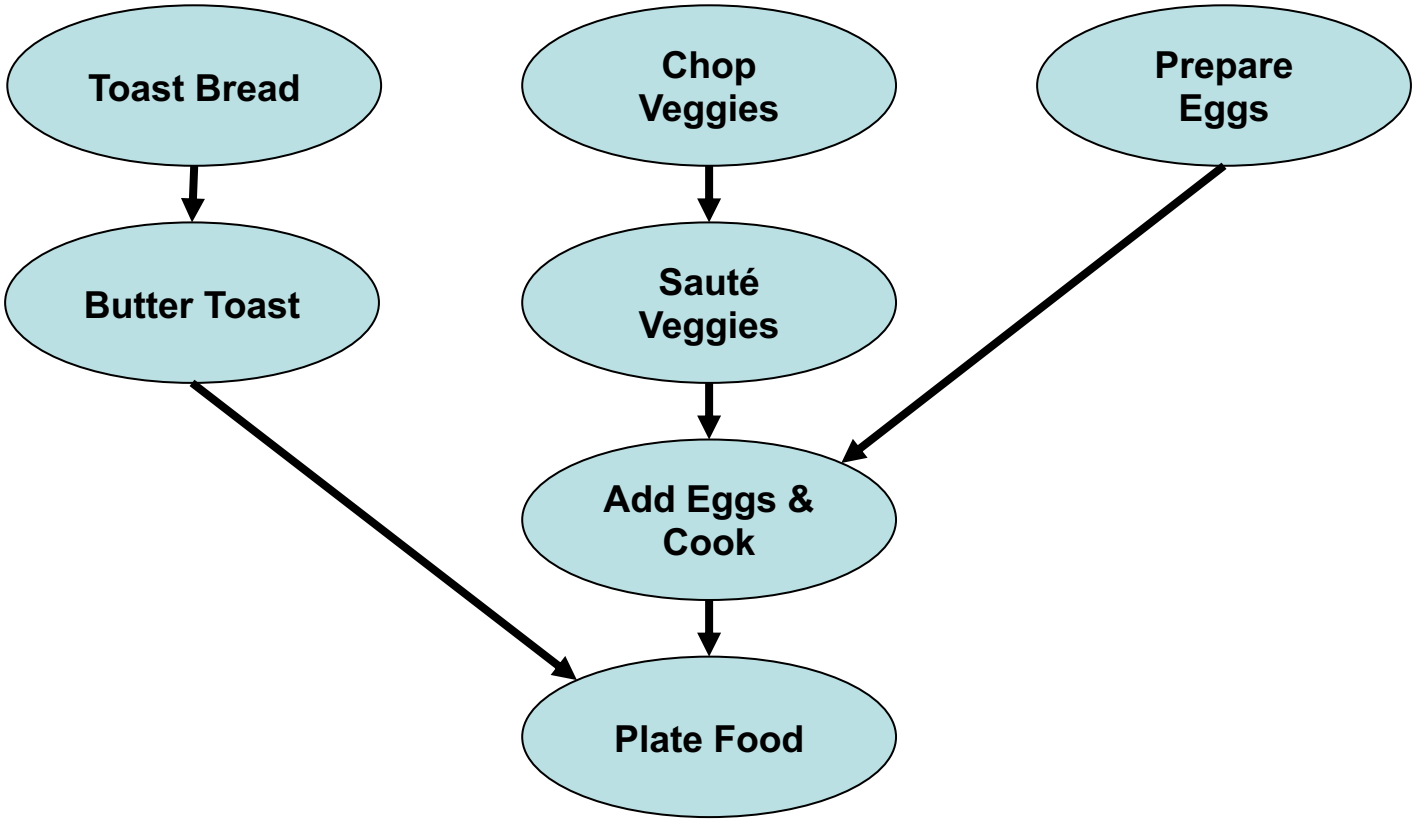
Toast Bread, Chop Veggies, Butter Toast, Prepare Eggs, Sauté Veggies, Add Eggs & Cook, Plate Food

Topological Sort

Given a directed (acyclic!) graph $G = (V, E)$, a topological sort is a total ordering of G's vertices such that for every edge (v, w) in E, vertex v precedes w in the ordering.

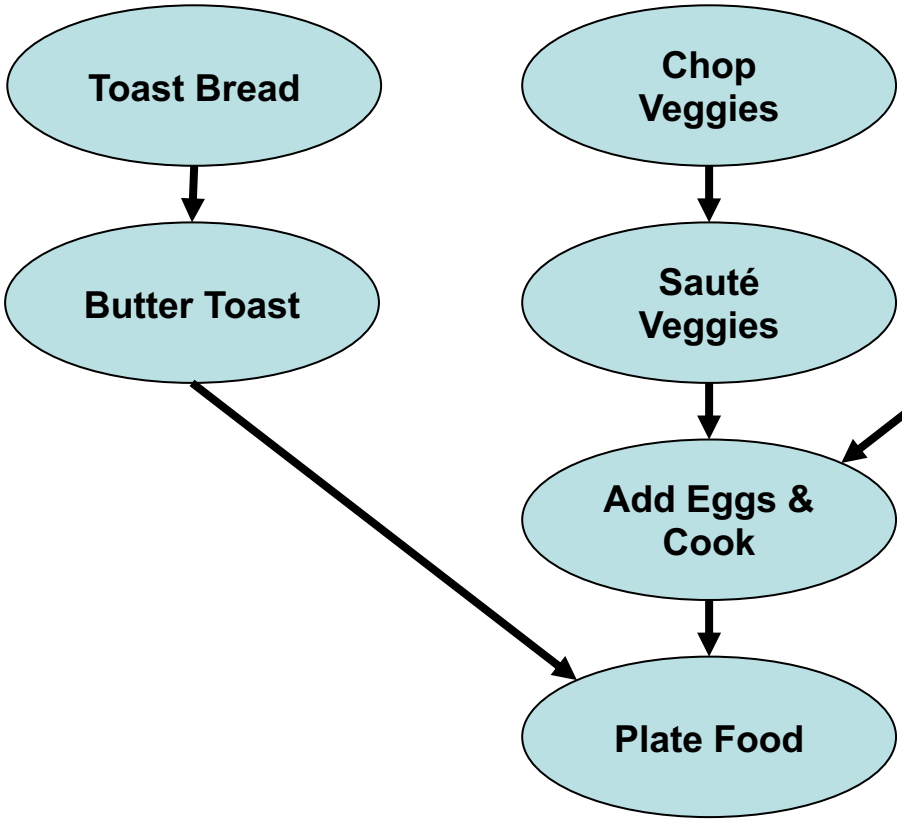


Topological Sort

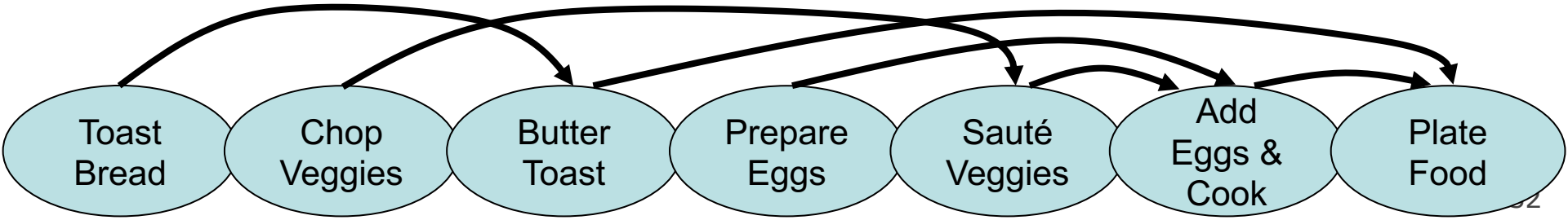


Toast Bread, Chop Veggies, Butter Toast, Prepare Eggs, Sauté Veggies, Add Eggs & Cook, Plate Food

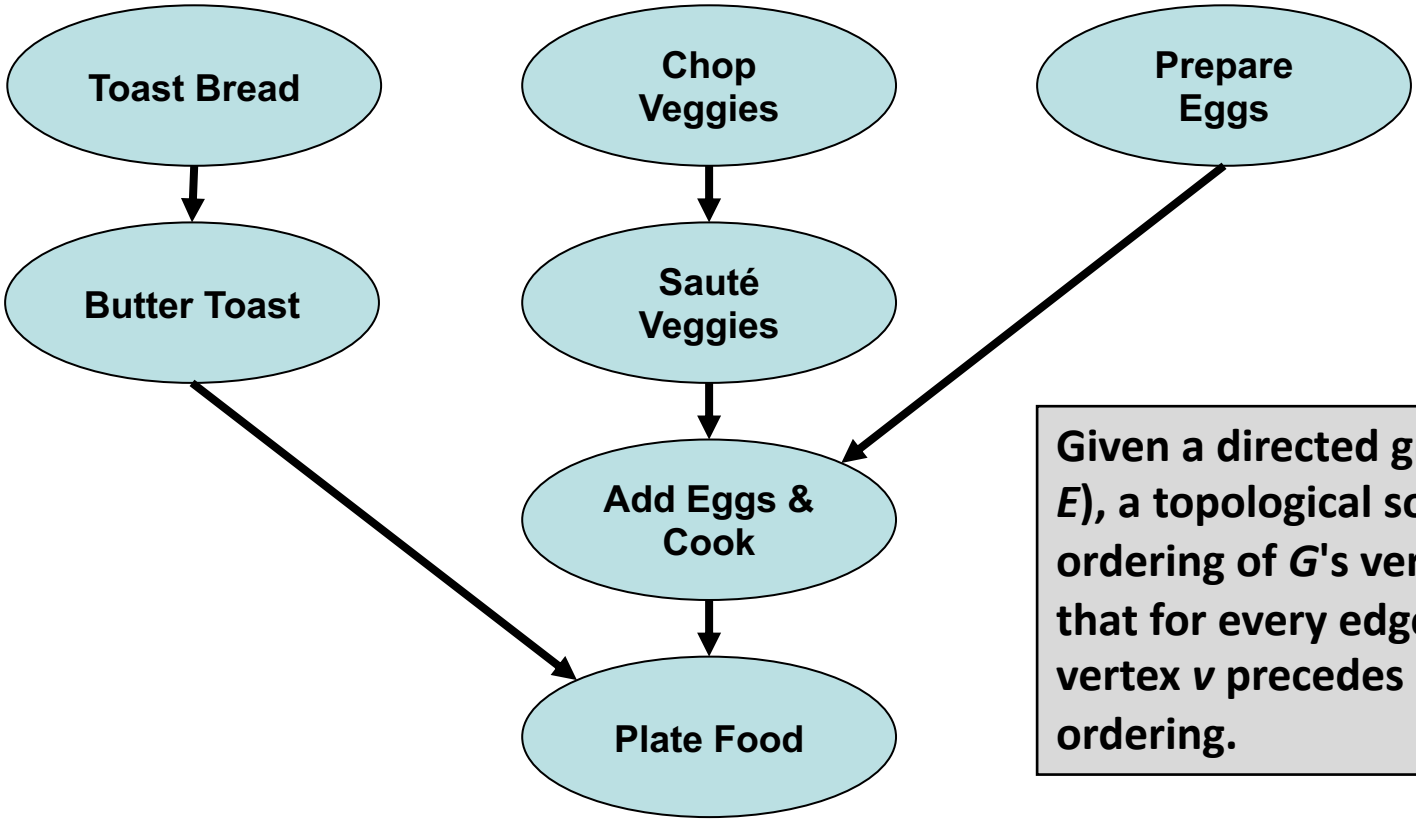
Topological Sort



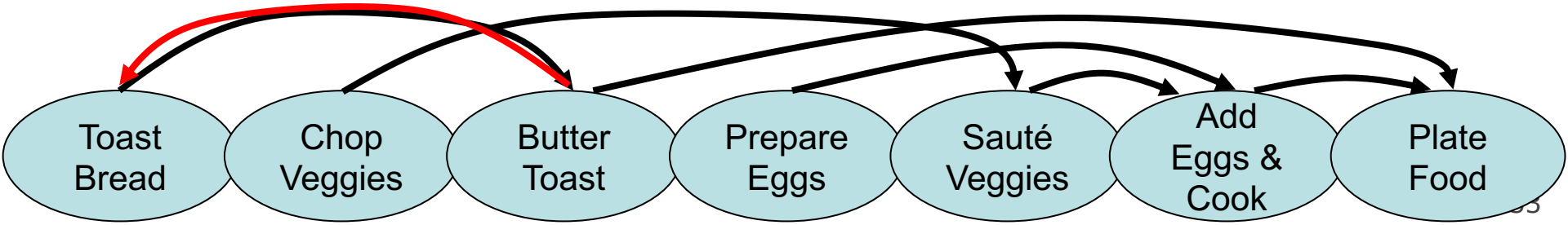
Given a directed graph *G* = (*V*, *E*), a topological sort is a total ordering of *G*'s vertices such that for every edge (*v*, *w*) in *E*, vertex *v* precedes *w* in the ordering.



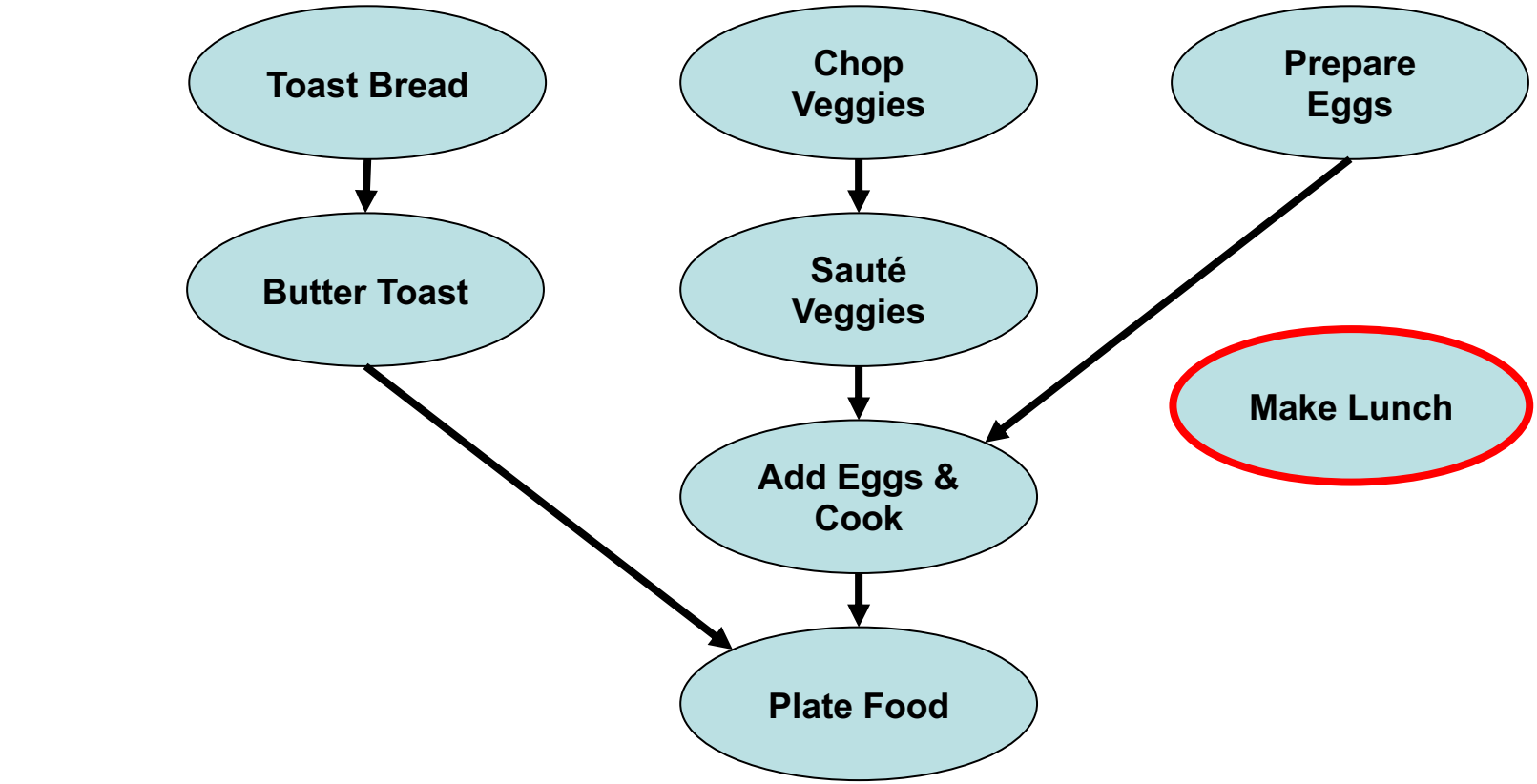
Topological Sort

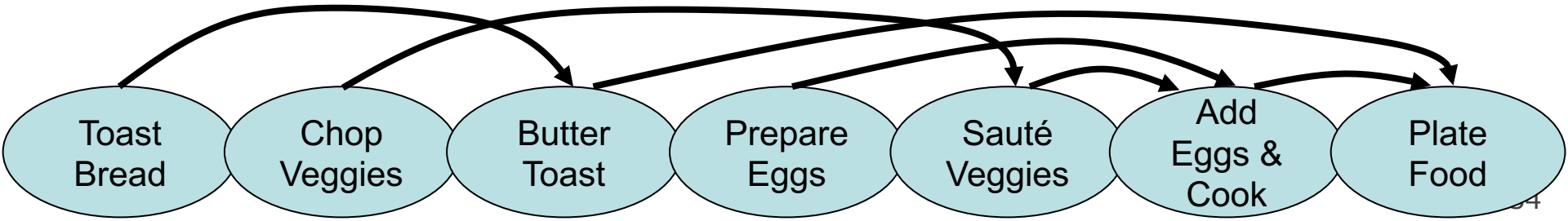


Given a directed graph *G* = (*V*, *E*), a topological sort is a total ordering of *G*'s vertices such that for every edge (*v*, *w*) in *E*, vertex *v* precedes *w* in the ordering.



Topological Sort





Topological Sort: Idea

1. Find node with in-degree of 0
2. Add it at the end of our topological ordering so far, and remove it and its edges from the graph
3. Repeat until no nodes left

Topological Sort Algorithm

```
ordering := { }.     // (empty list)
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```

- This algorithm modifies the passed-in graph ☹
- Have we handled all edge cases?
- What is the runtime of this algorithm?

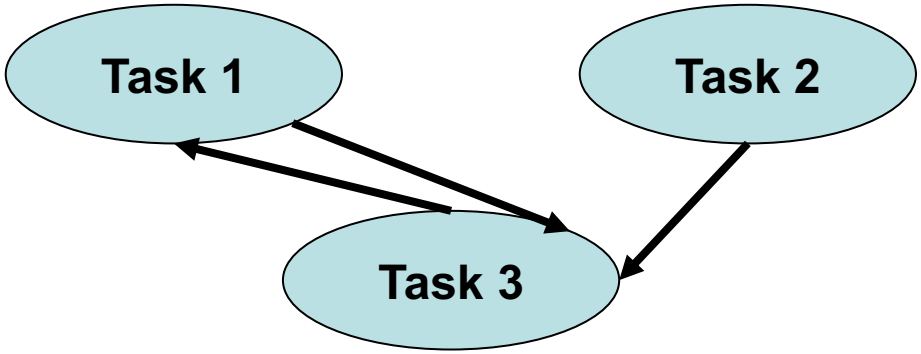
Topological Sort Algorithm

```
ordering := { }.     // (empty list)
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```

- This algorithm modifies the passed-in graph ☹
- Have we handled all edge cases?
- What is the runtime of this algorithm?

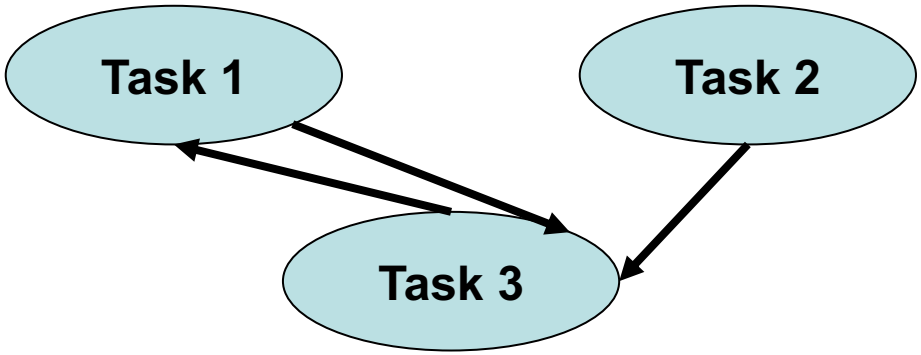
Topological Sort Algorithm: Cycles

```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```



Topological Sort Algorithm: Cycles

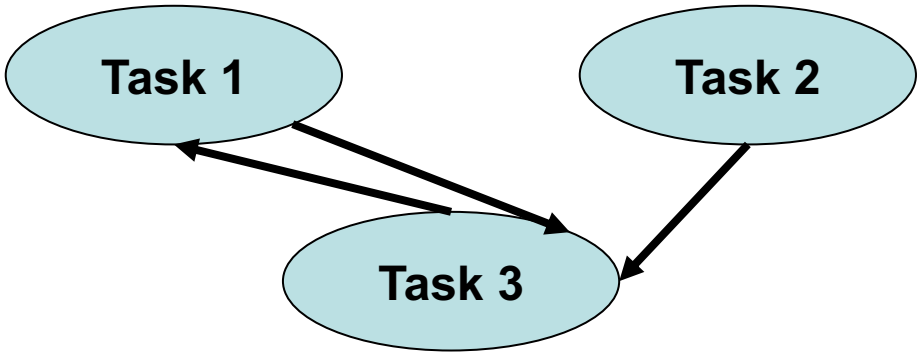
```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```



Ordering: { }

Topological Sort Algorithm: Cycles

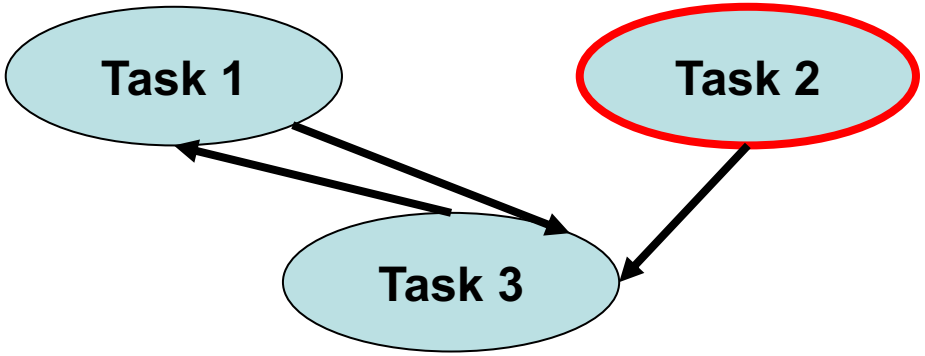
```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```



Ordering: { }

Topological Sort Algorithm: Cycles

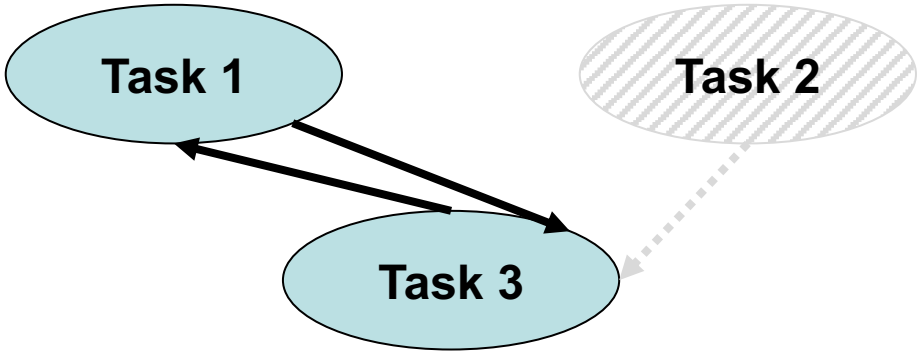
```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```



Ordering: { }

Topological Sort Algorithm: Cycles

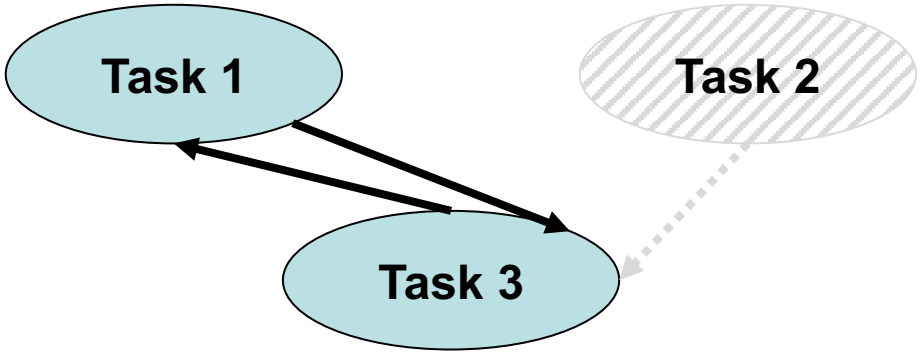
```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```



Ordering: { }

Topological Sort Algorithm: Cycles

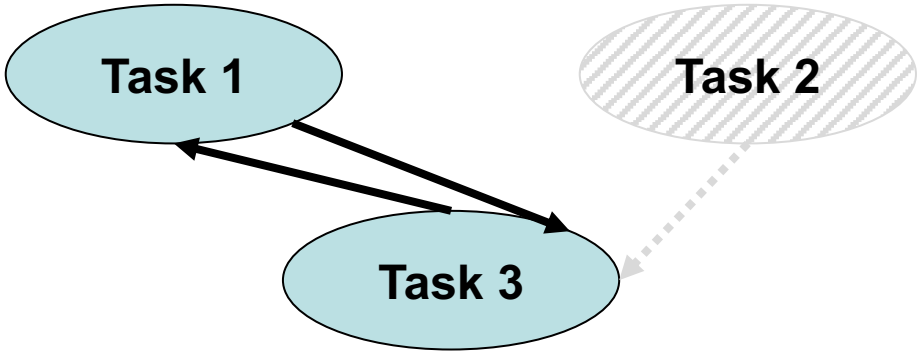
```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```



Ordering: { “Task 2” }

Topological Sort Algorithm: Cycles

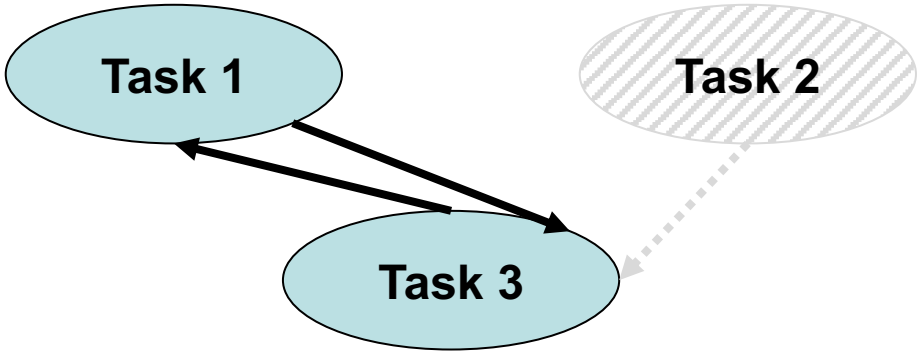
```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```



Ordering: { “Task 2” }

Topological Sort Algorithm: Cycles

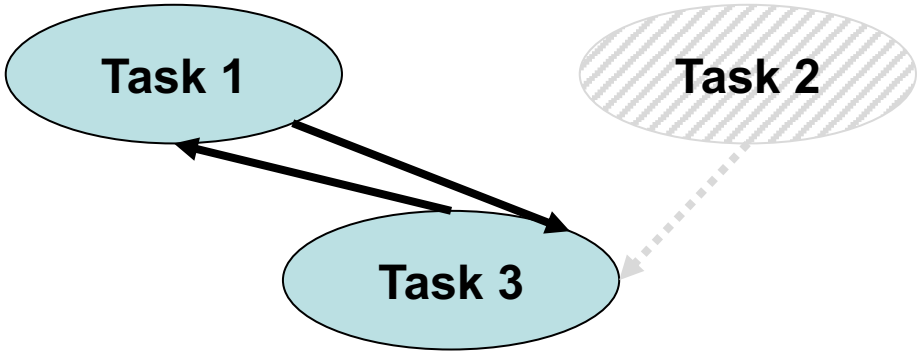
```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
   ordering += v
```



Ordering: { “Task 2” }

Topological Sort Algorithm: Cycles

```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
      - if none, no valid ordering possible
   Delete v and its outgoing edges from graph
   ordering += v
```



Ordering: { “Task 2” }

Topological Sort Algorithm: Cycles

```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
      - if none, no valid ordering possible
   Delete v and its outgoing edges from graph
   ordering += v
```

- This algorithm modifies the passed-in graph ☹
- Have we handled all edge cases?
- What is the runtime of this algorithm?

Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
      - if none, no valid ordering possible
   Delete v and its outgoing edges from graph
   ordering += v
```

Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

O(1)

```
ordering := { }.
```

O(V)

```
Repeat until graph is empty:
```

O(V)

```
    Find a vertex v with in-degree of 0
        - if none, no valid ordering possible
```

O(E)

```
    Delete v and its outgoing edges from graph
```

O(1)

```
    ordering += v
```

O(V(V+E))

Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

O(1)
```
ordering := { }.
```

O(V)
```
Repeat until graph is empty:
```

O(V)
```
Find a vertex v with in-degree of 0
   - if none, no valid ordering possible
```

O(E)
```
Delete v and its outgoing edges from graph
```

Is the worst case here really O(E) *every time?* For example, if one node has *all* the outgoing edges, then we'll have an O(E) operation that time, but a no-op every other time.

Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

O(1)
```
ordering := { }.
```

O(V)
```
Repeat until graph is empty:
```

O(V)
```
Find a vertex v with in-degree of 0
```
```
   - if none, no valid ordering possible
```

O(E)

```
Delete v and its outgoing edges from graph
```

Key Idea: every edge can be deleted *at most once.* Therefore, the total runtime of this operation is O(E), not O(V*E).

Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

O(1)

```
ordering := { }.
```

O(V)

```
Repeat until graph is empty:
```

O(V)

```
    Find a vertex v with in-degree of 0
        - if none, no valid ordering possible
```

O(E) *total*

```
    Delete v and its outgoing edges from graph
```

O(1)

```
    ordering += v
```

Tighter bound: $O(V^2 + E)$

Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

O(1)
ordering := { }.

O(V)
Repeat until graph is empty:

O(V)
Find a vertex *v* with in-degree of 0

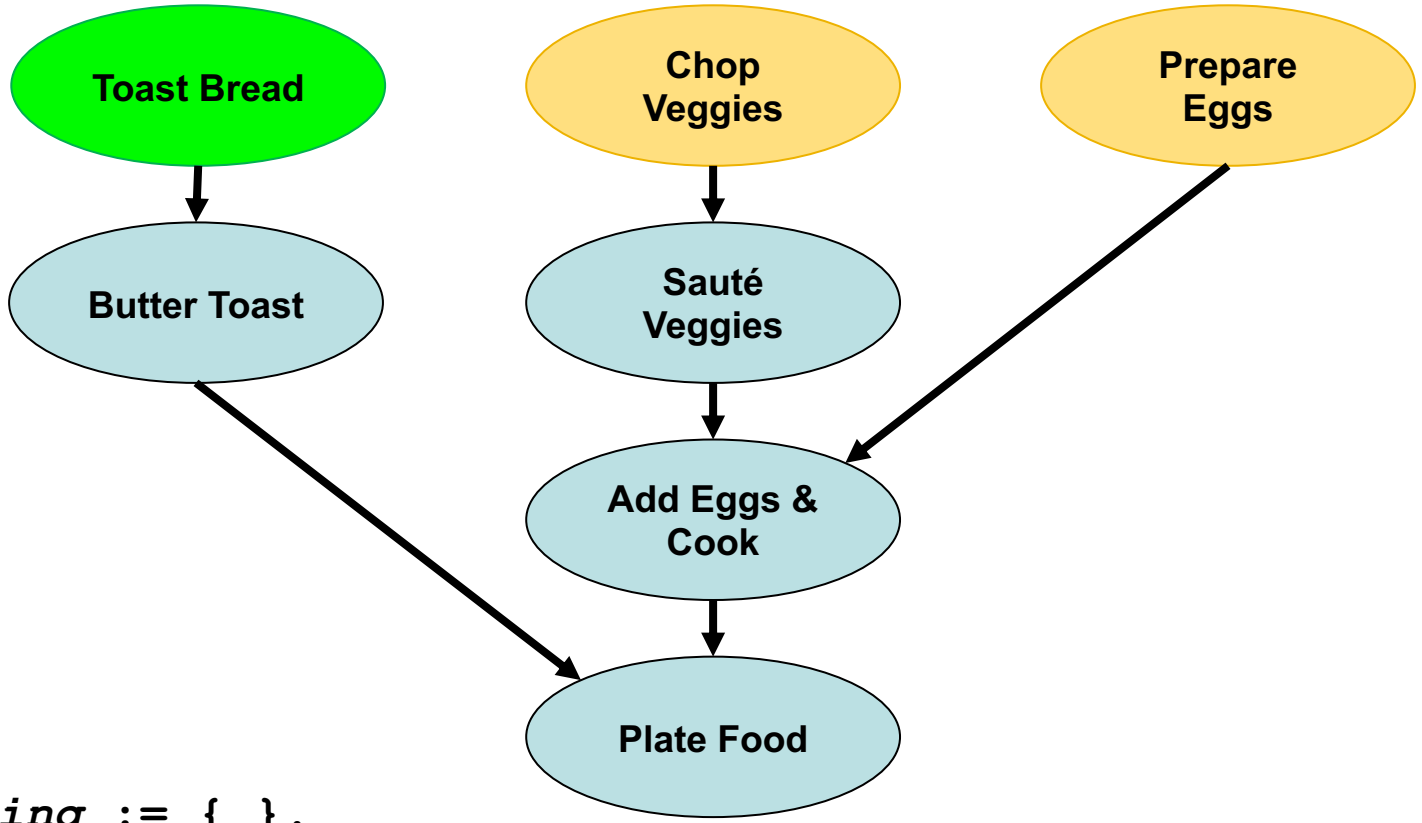
- if n… possible

Can we make this more efficient?

O(E) *total*
Delete *v* and its outgoing edges from graph

O(1)
ordering += v

Topological Sort: Take 2

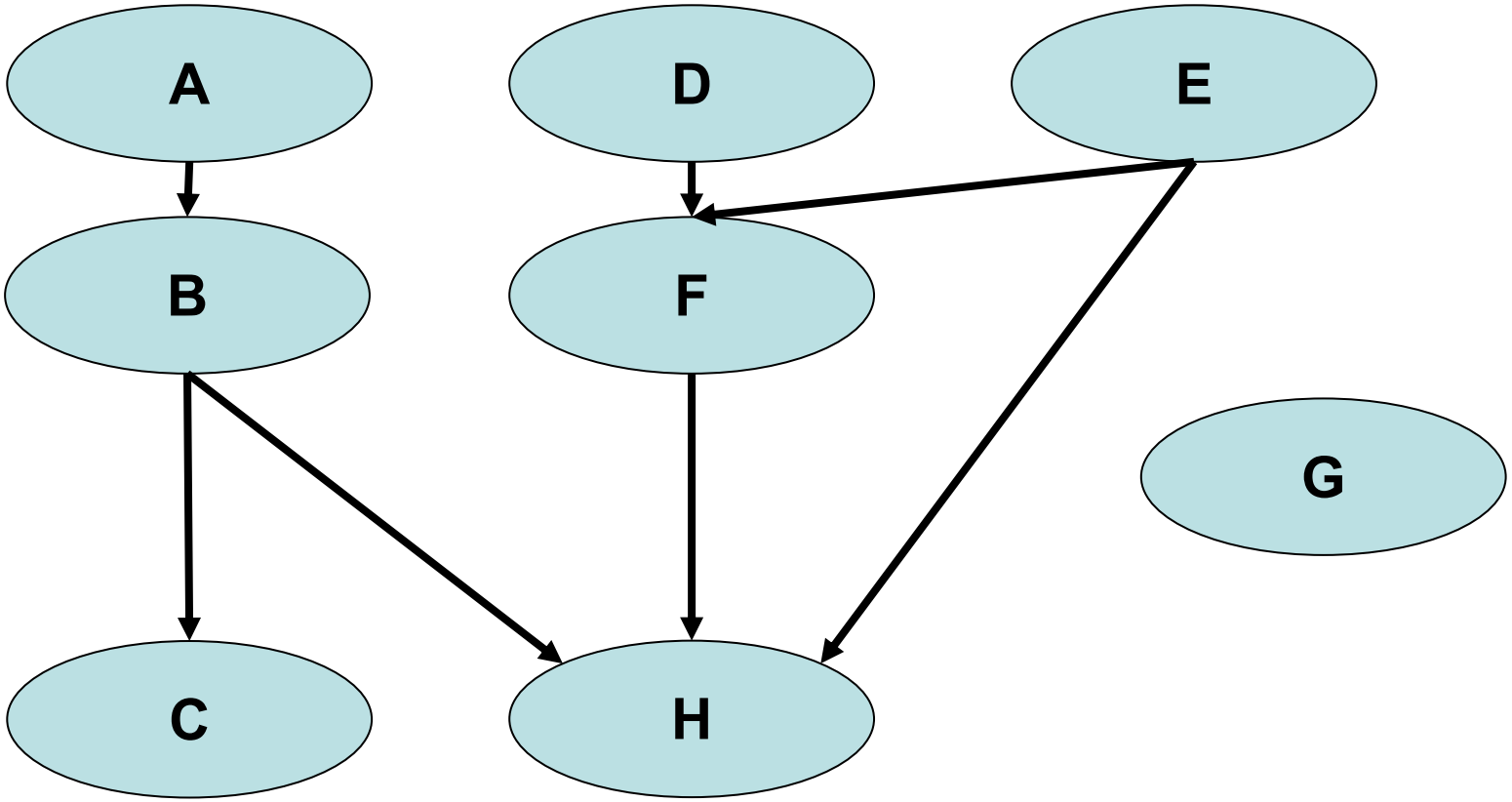



```
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
       - if none, no valid ordering possible
   Delete v and its outgoing edges from graph
   ordering += v
```

3 Key Ideas for Improvement

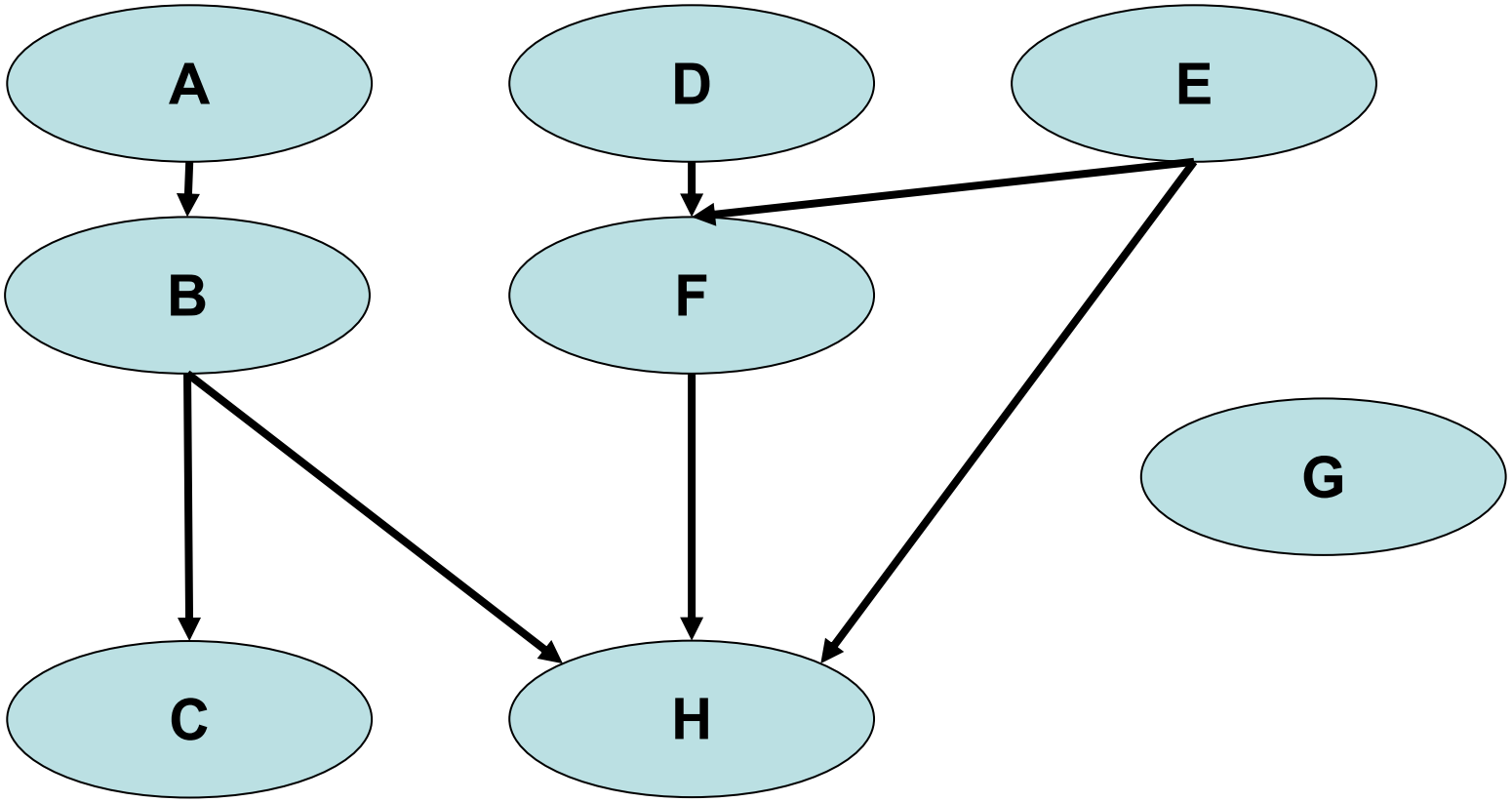
- Keep a *queue of nodes with in-degree 0* so we don't have to search for nodes multiple times.
- A node's in-degree only changes when one of its prerequisites is completed. Therefore, when completing a task, check if any of its neighbors now has in-degree 0.
- Keep a *map of nodes' in-degrees* so we don't need to modify the graph.

Topological Sort: Take 2



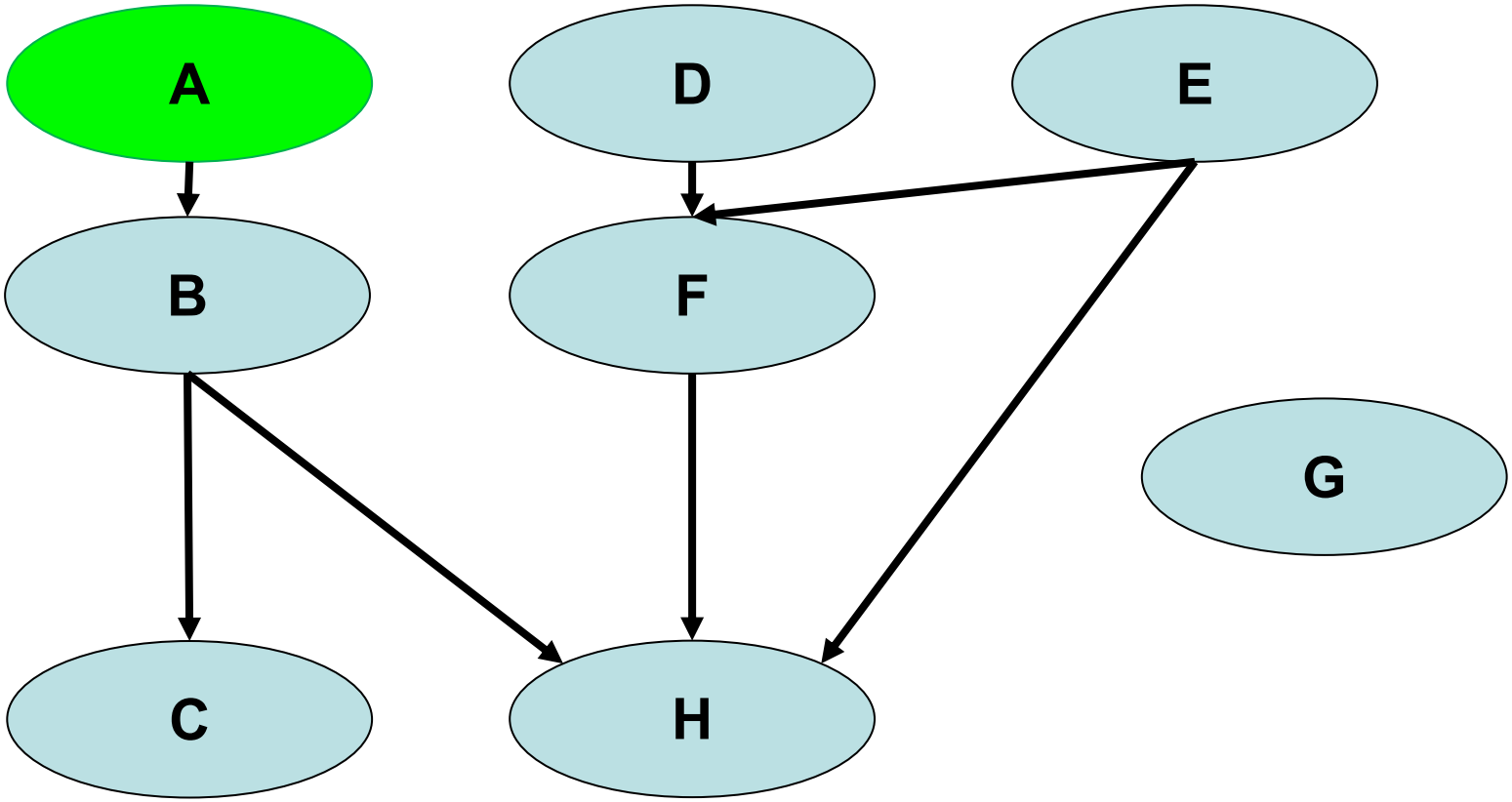
0-In-Degree Queue: { }
In-Degree Map: { }

Topological Sort: Take 2

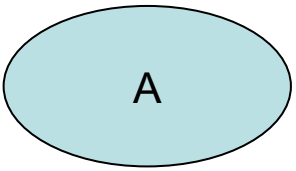


0-In-Degree Queue: { “A”, “D”, “E”, “G” }
In-Degree Map: { “A”:0, “B”:1, “C”:1, “D”:0, “E”: 0, “F”: 2, “G”:0, “H”:3 }

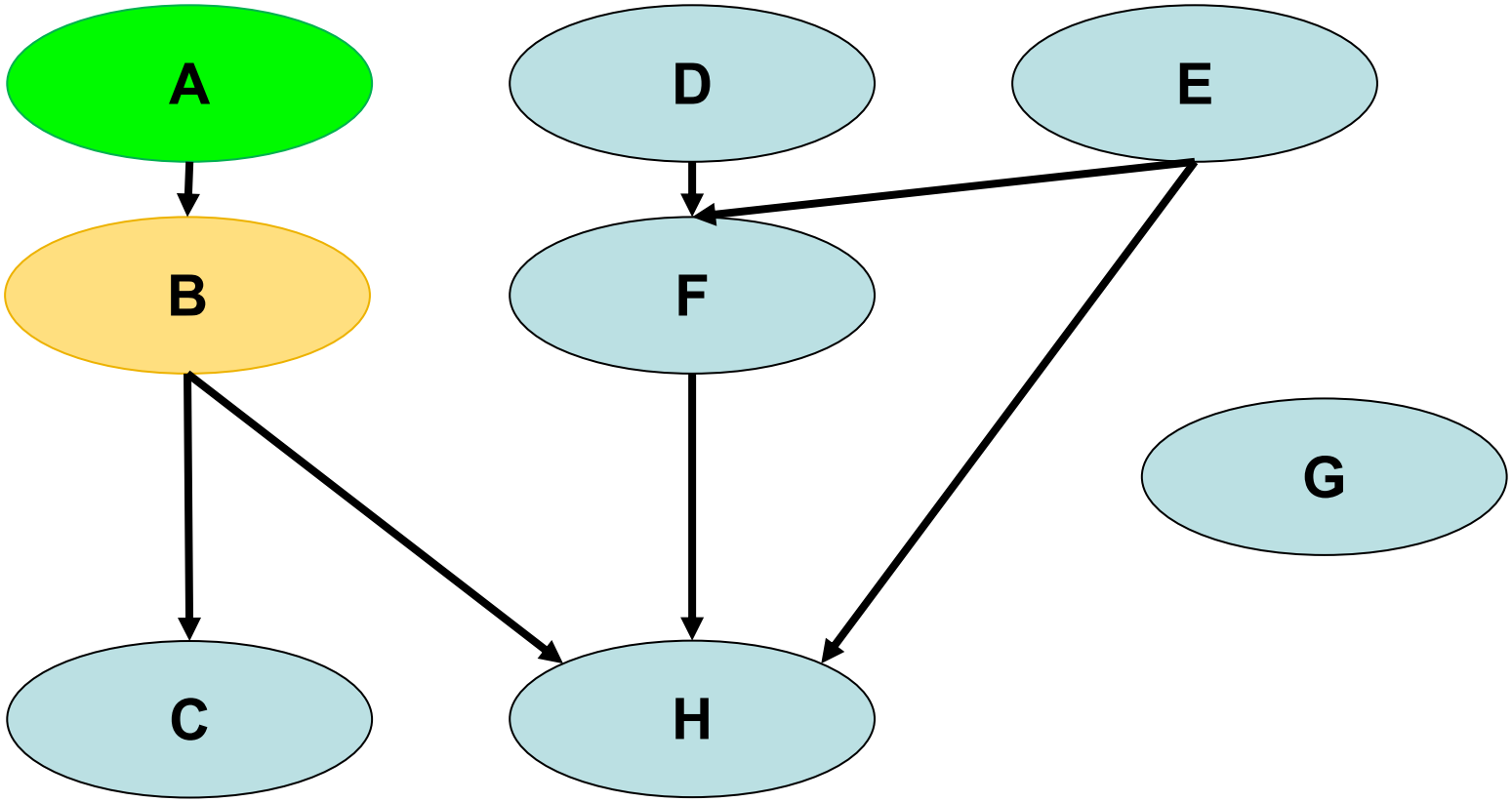
Topological Sort: Take 2



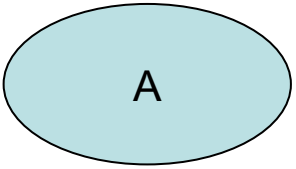
0-In-Degree Queue: { ~~"A"~~, "D", "E", "G" }
In-Degree Map: { "A":0, "B":1, "C":1, "D":0, "E": 0, "F": 2, "G":0, "H":3 }



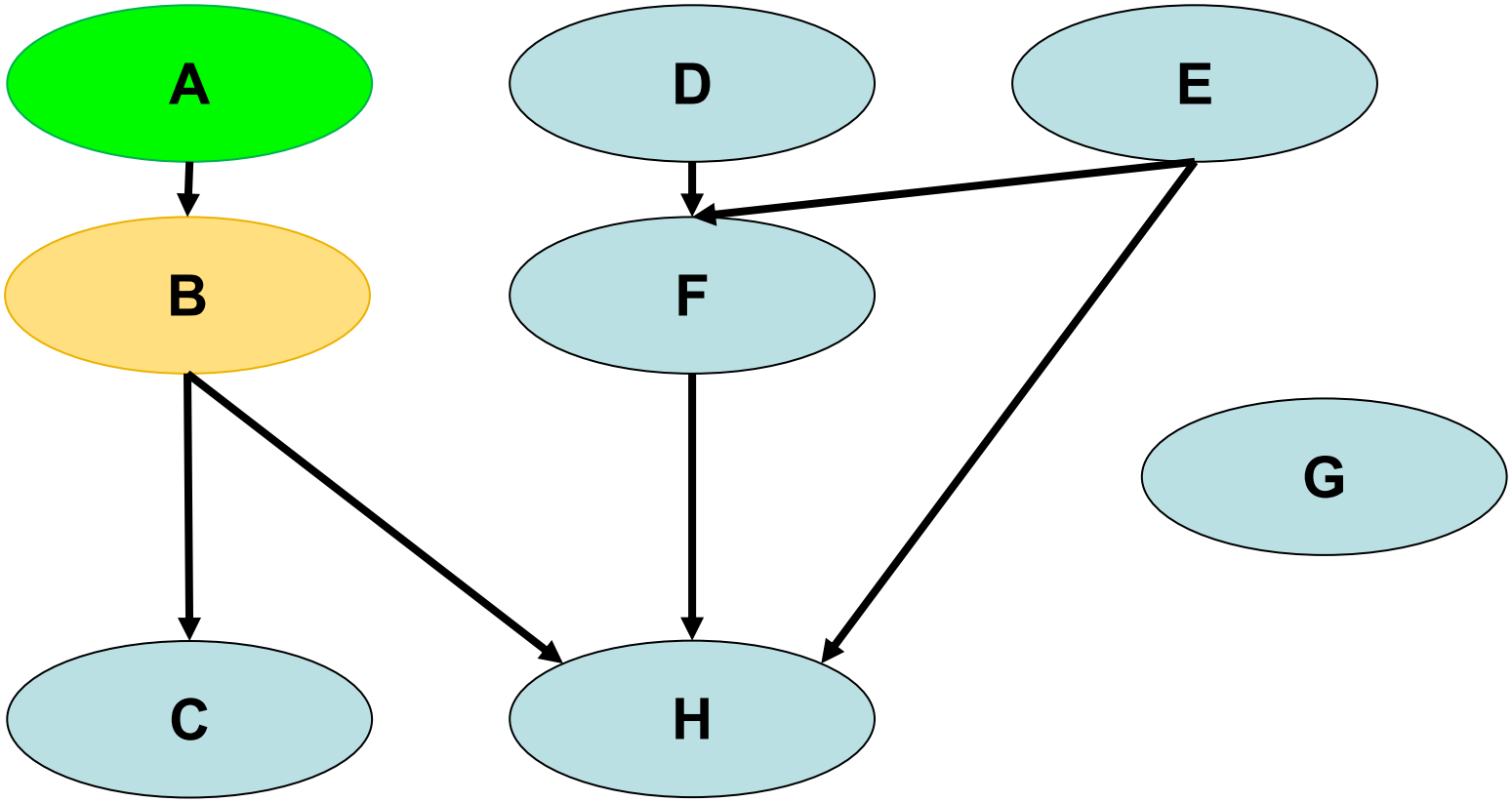
Topological Sort: Take 2



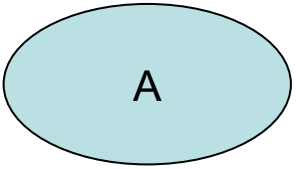
0-In-Degree Queue: { “D”, “E”, “G” }
In-Degree Map: { “A”:0, “B”:1, “C”:1, “D”:0, “E”: 0, “F”: 2, “G”:0, “H”:3 }



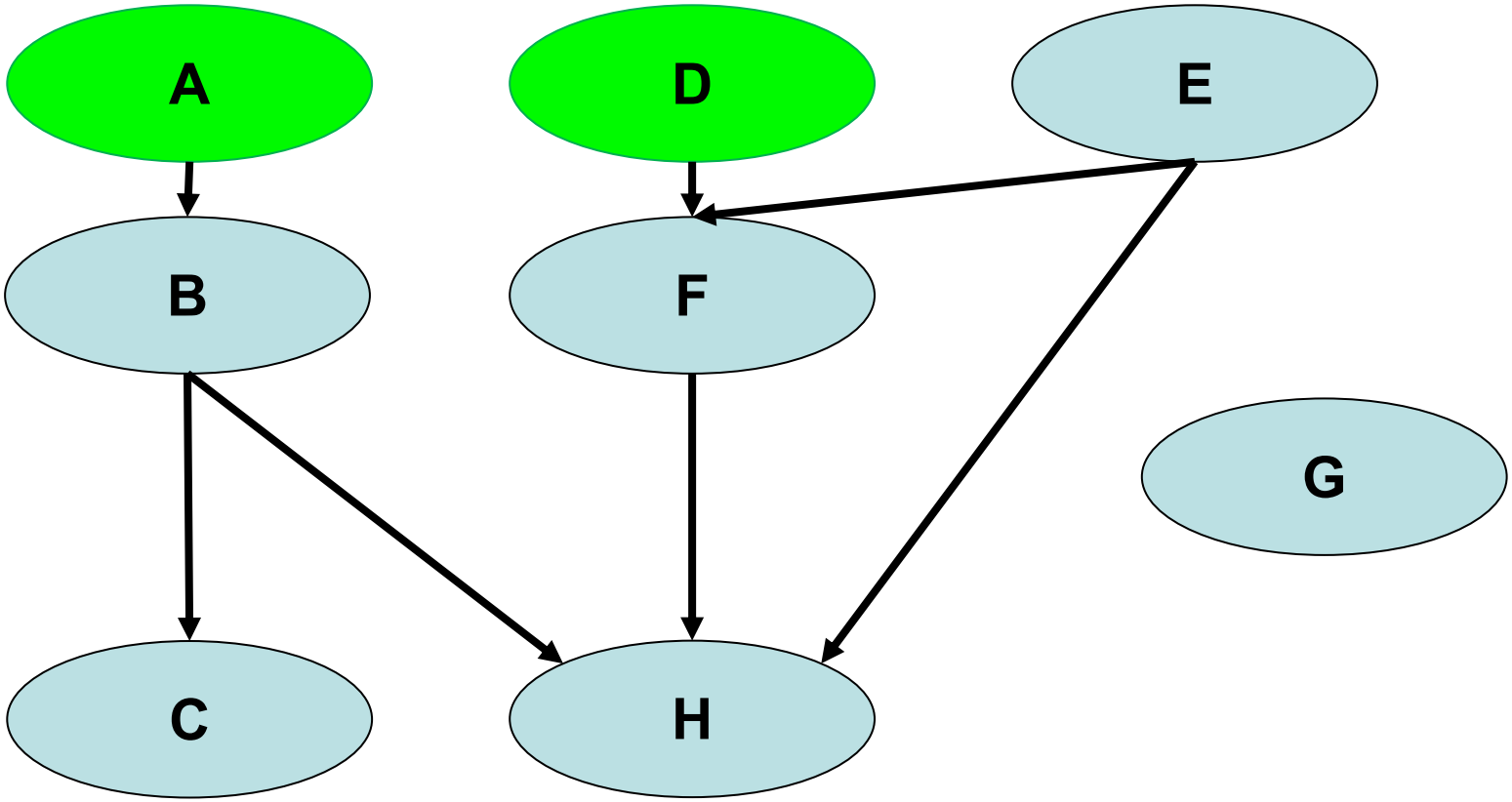
Topological Sort: Take 2



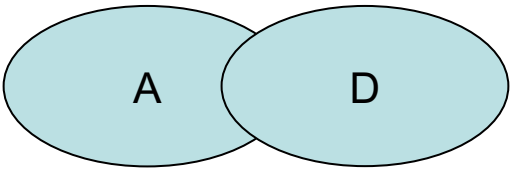
0-In-Degree Queue: { “D”, “E”, “G”, “B” }
In-Degree Map: { “A”:0, “B”:0, “C”:1, “D”:0, “E”: 0, “F”: 2, “G”:0, “H”:3 }



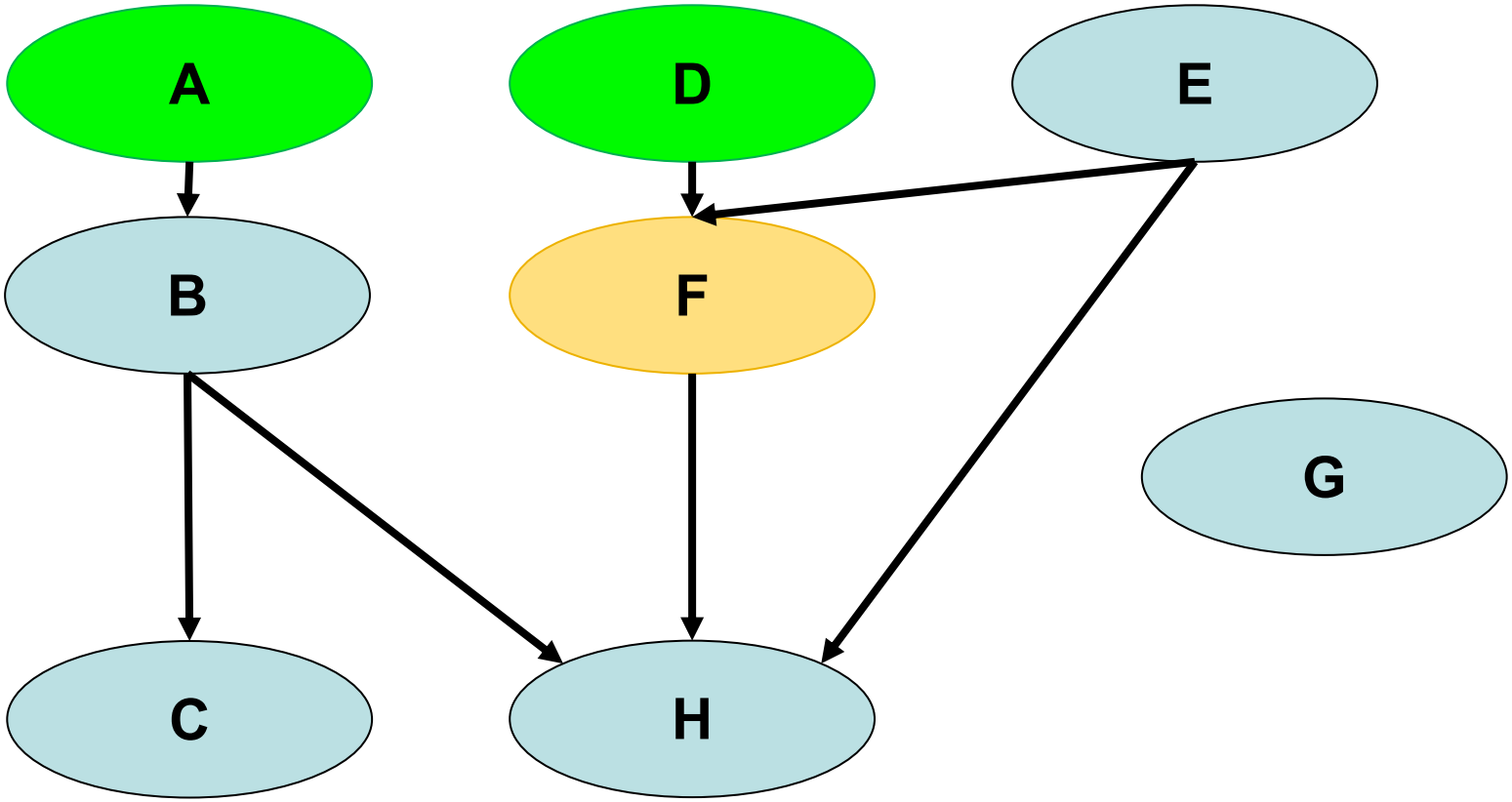
Topological Sort: Take 2



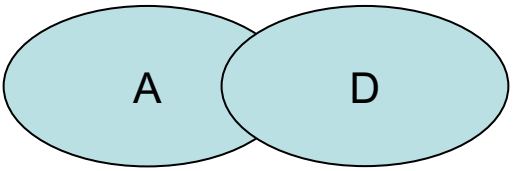
0-In-Degree Queue: { ~~"D"~~, "E", "G", "B" }
In-Degree Map: { "A":0, "B":0, "C":1, "D":0, "E": 0, "F": 2, "G":0, "H":3 }



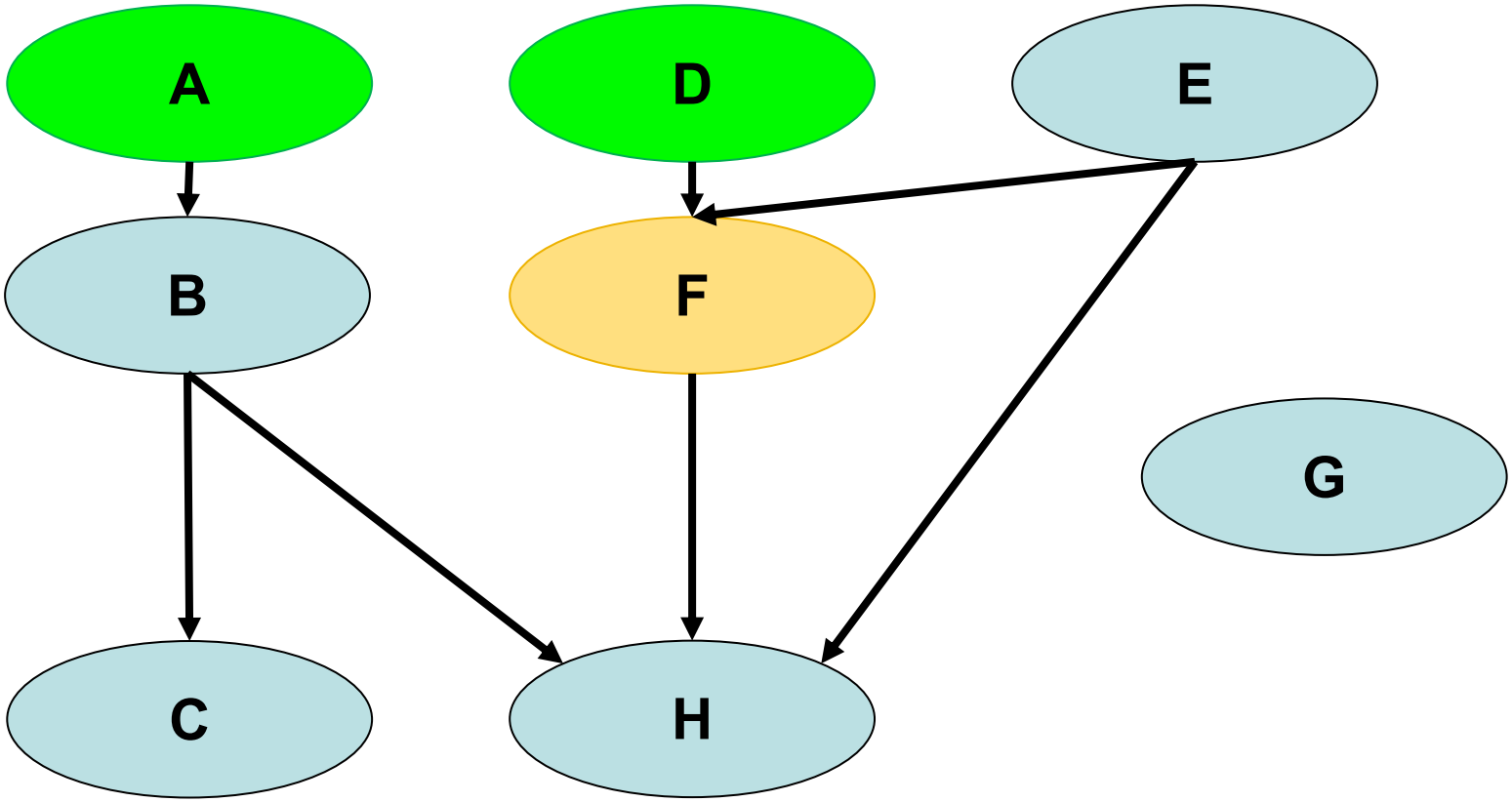
Topological Sort: Take 2



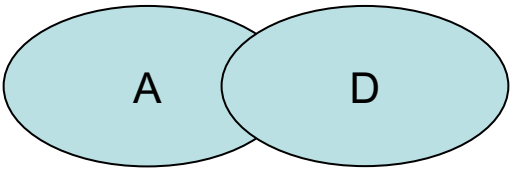
0-In-Degree Queue: { “E”, “G”, “B” }
In-Degree Map: { “A”:0, “B”:0, “C”:1, “D”:0, “E”: 0, “F”: 2, “G”:0, “H”:3 }



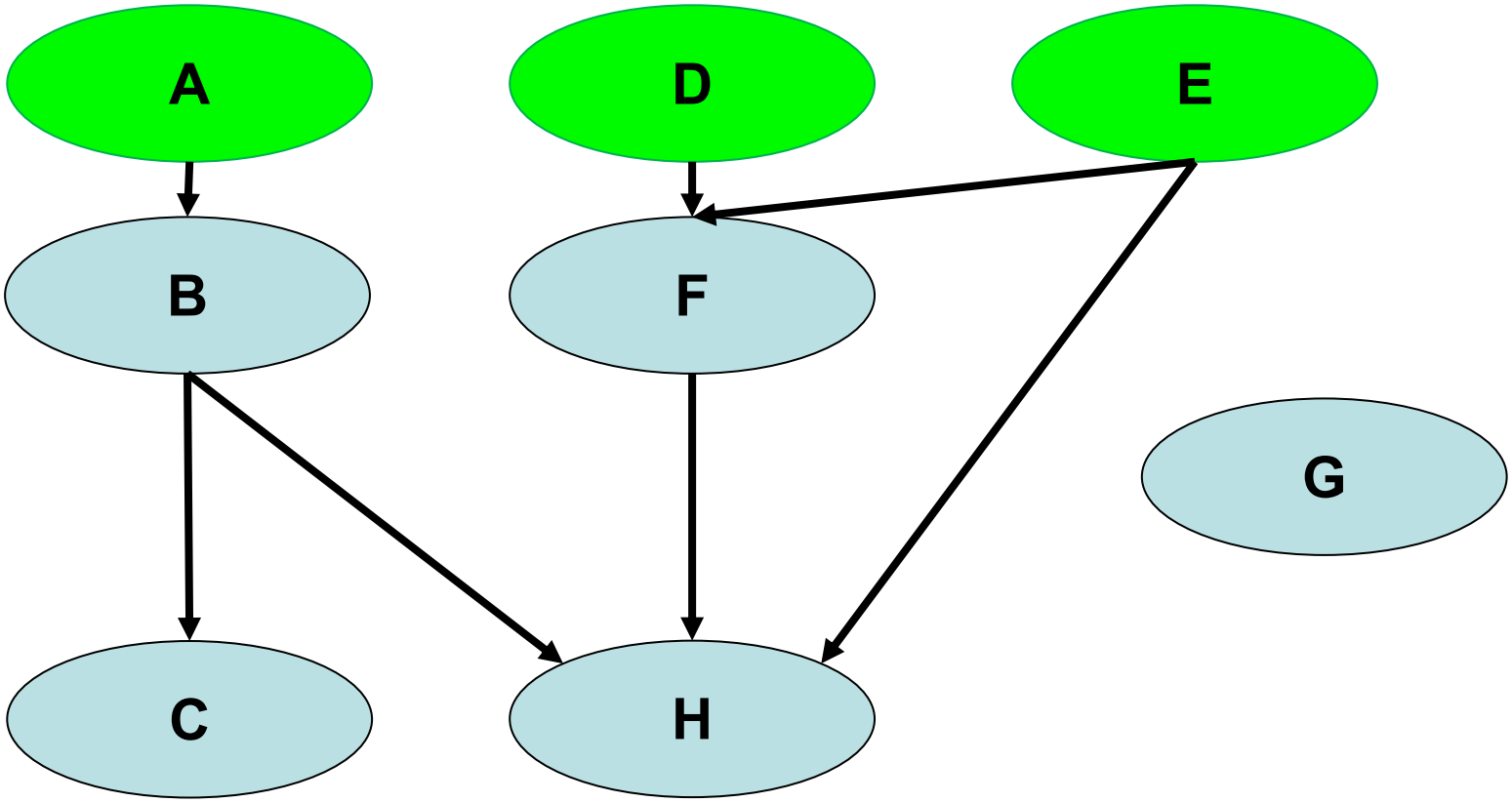
Topological Sort: Take 2



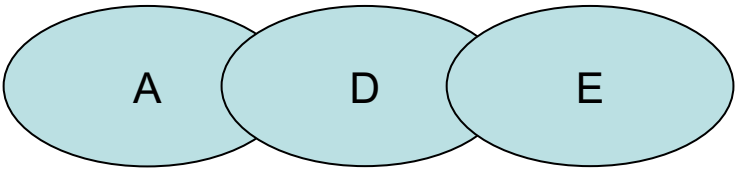
0-In-Degree Queue: { “E”, “G”, “B” }
In-Degree Map: { “A”:0, “B”:0, “C”:1, “D”:0, “E”: 0, “F”: 1, “G”:0, “H”:3 }



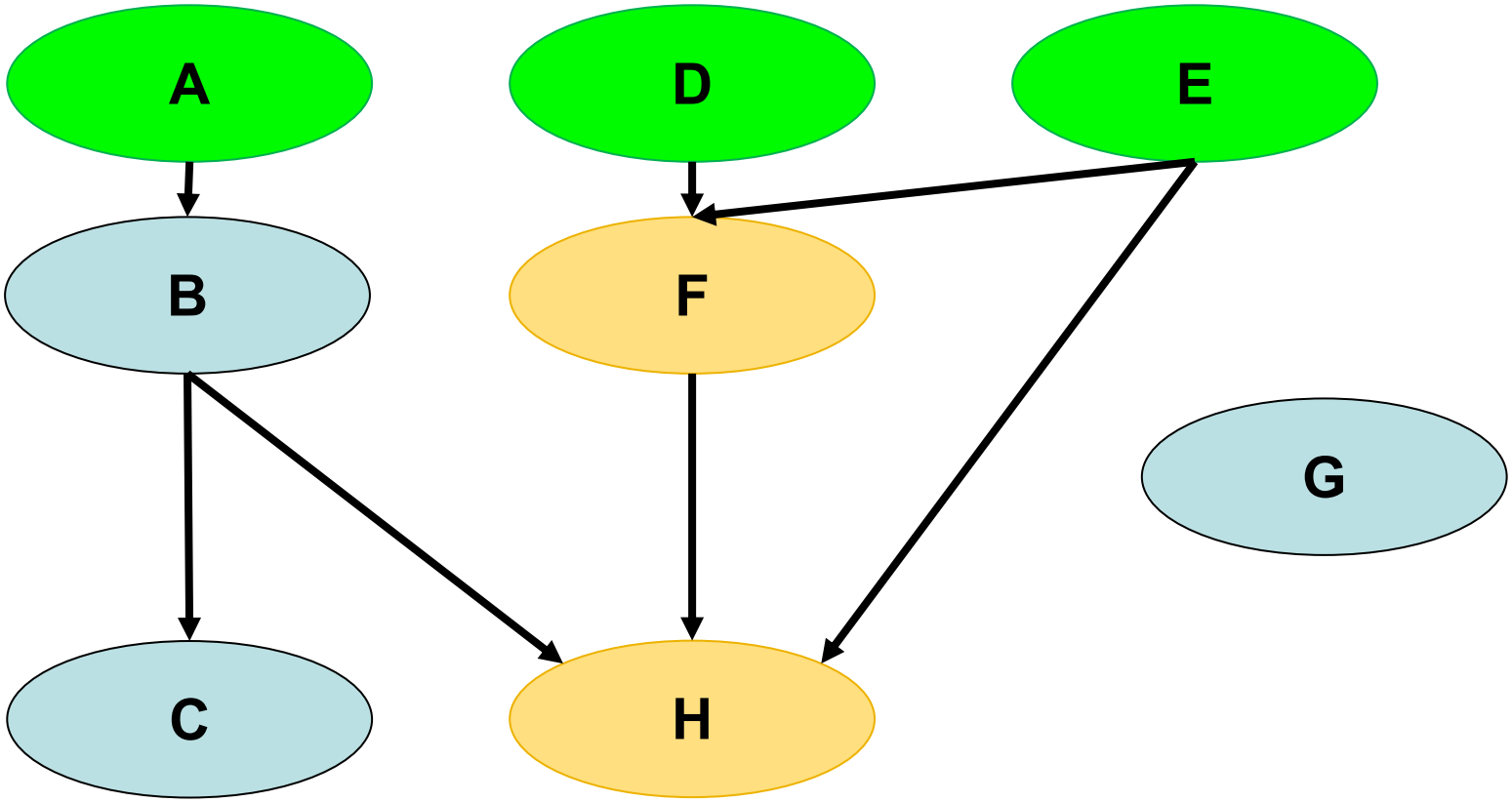
Topological Sort: Take 2



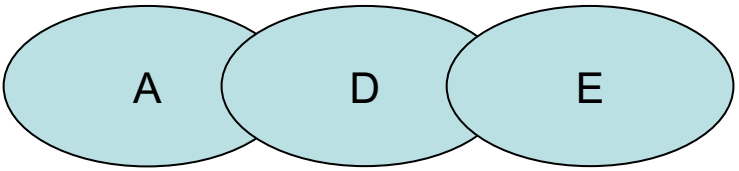
0-In-Degree Queue: { ~~"E"~~, "G", "B" }
In-Degree Map: { "A":0, "B":0, "C":1, "D":0, "E": 0, "F": 1, "G":0, "H":3 }



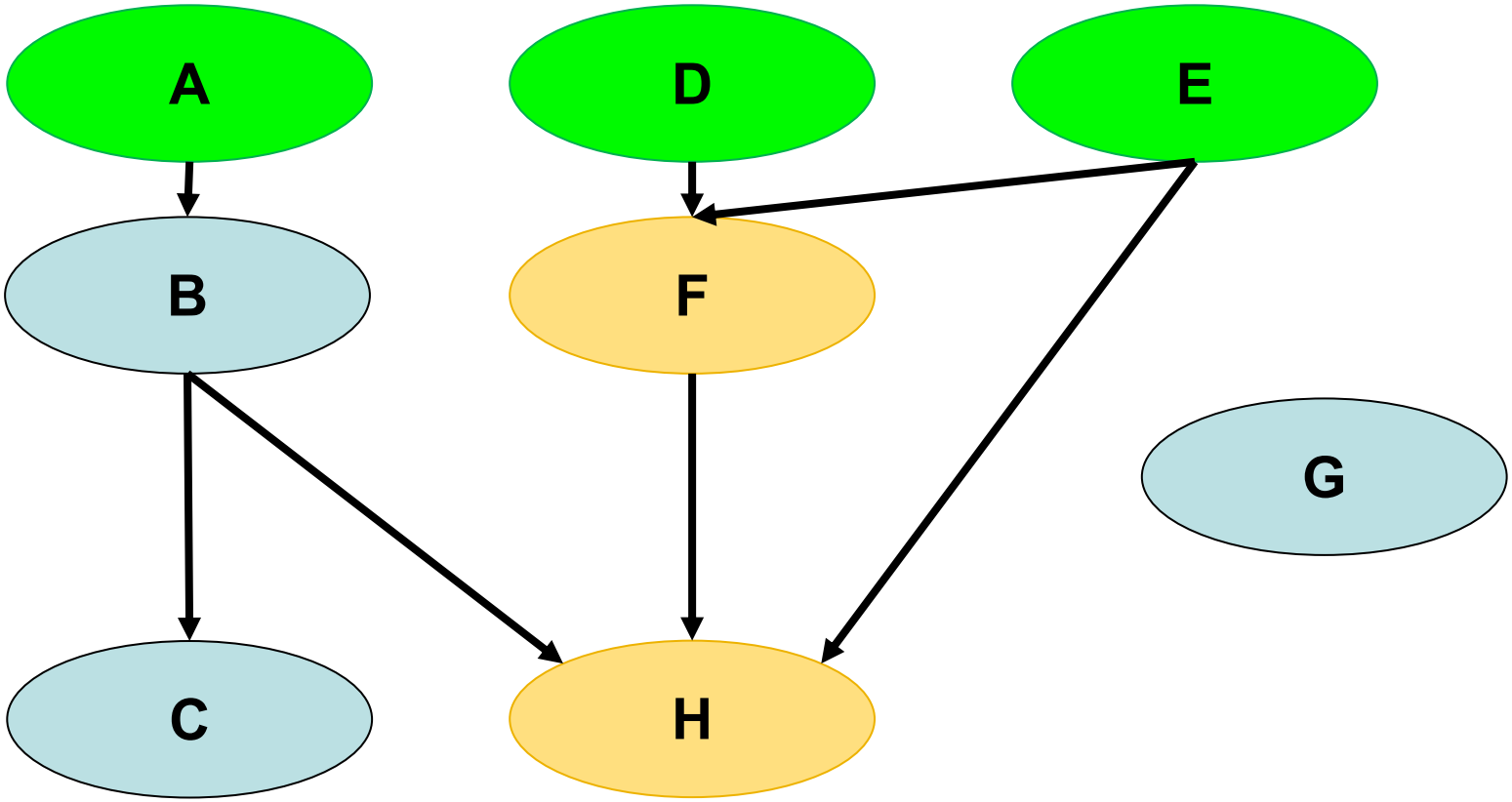
Topological Sort: Take 2



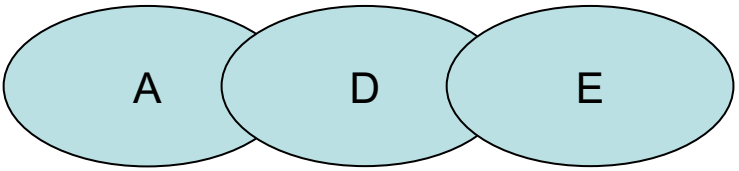
0-In-Degree Queue: { “G”, “B” }
In-Degree Map: { “A”:0, “B”:0, “C”:1, “D”:0, “E”: 0, “F”: 1, “G”:0, “H”:3 }



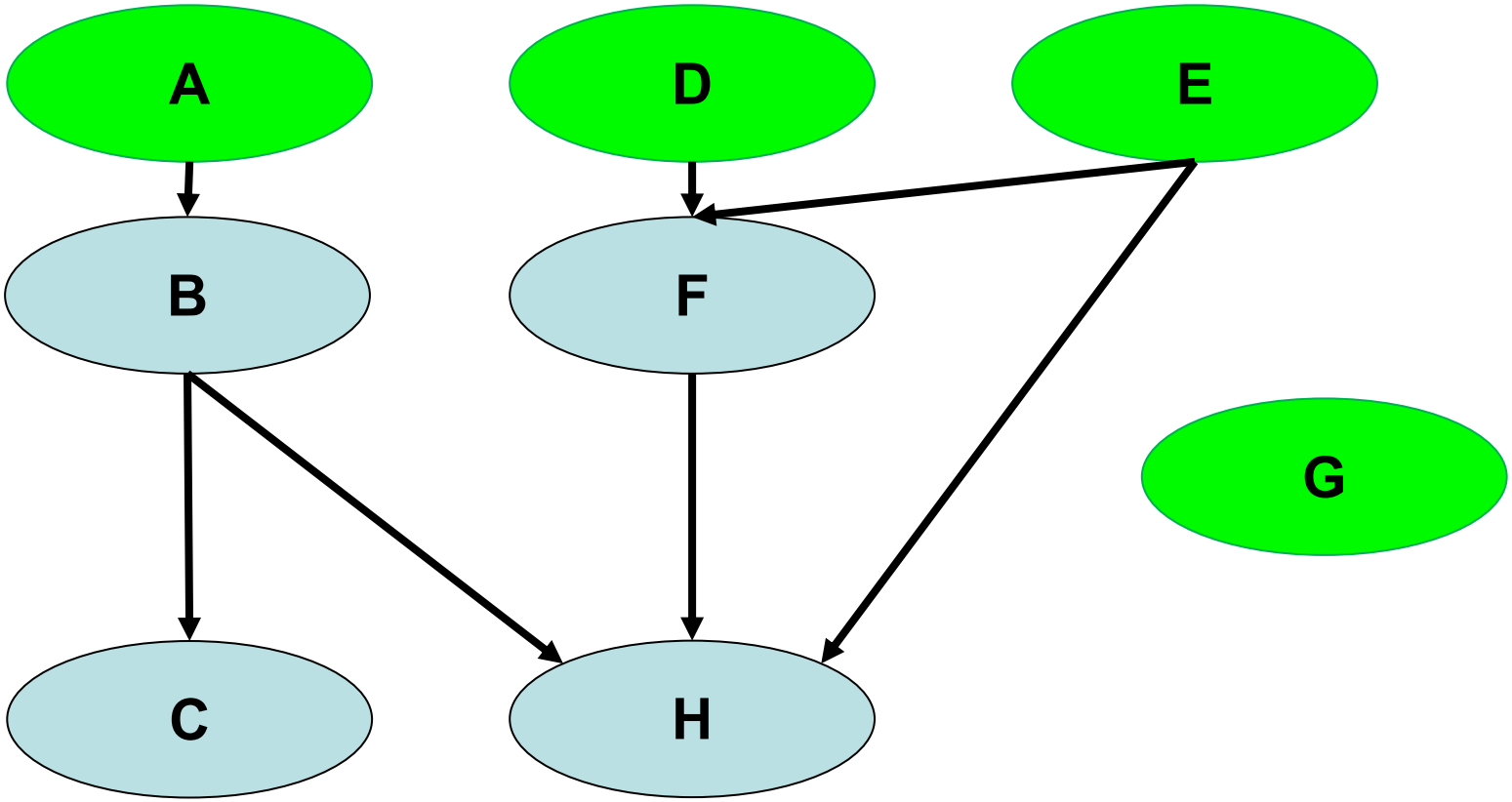
Topological Sort: Take 2



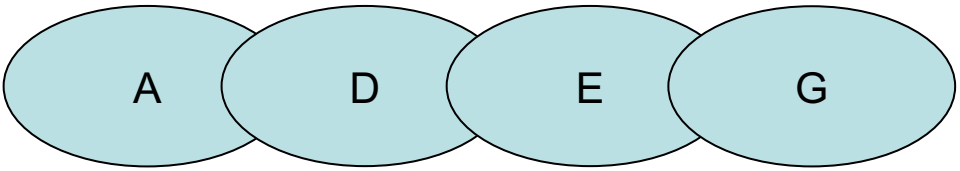
0-In-Degree Queue: { "G", "B", "F" }
In-Degree Map: { "A":0, "B":0, "C":1, "D":0, "E": 0, "F": 0, "G":0, "H":2 }



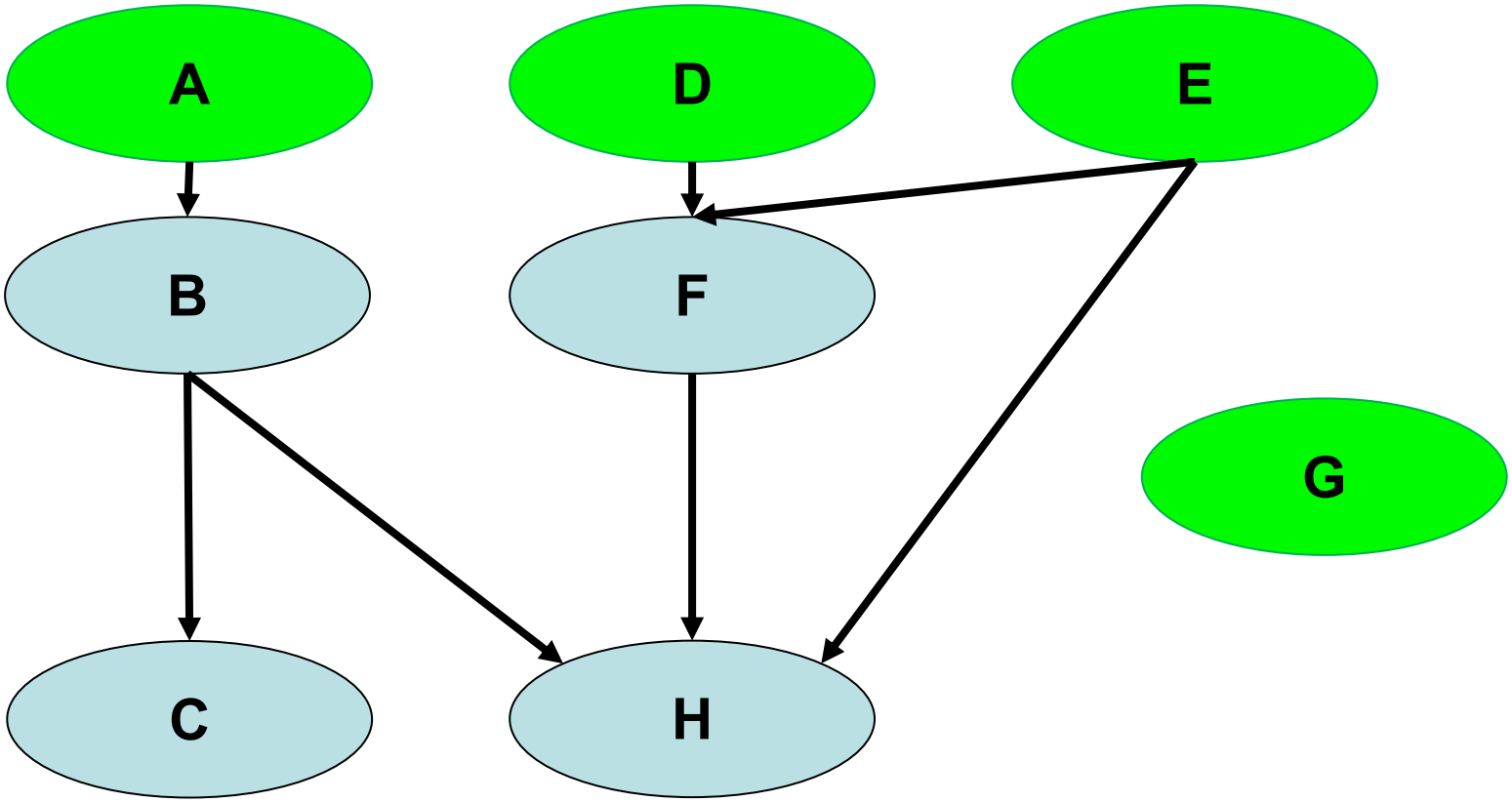
Topological Sort: Take 2



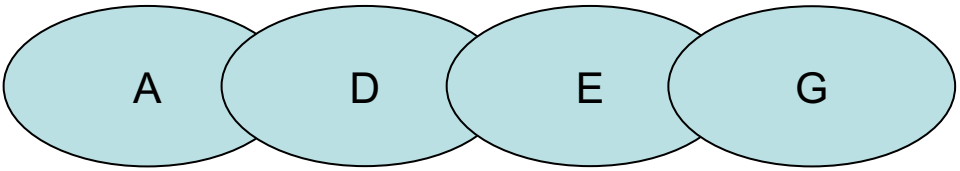
0-In-Degree Queue: { ~~"G",~~ "B", "F" }
In-Degree Map: { "A":0, "B":0, "C":1, "D":0, "E": 0, "F": 0, "G":0, "H":2 }



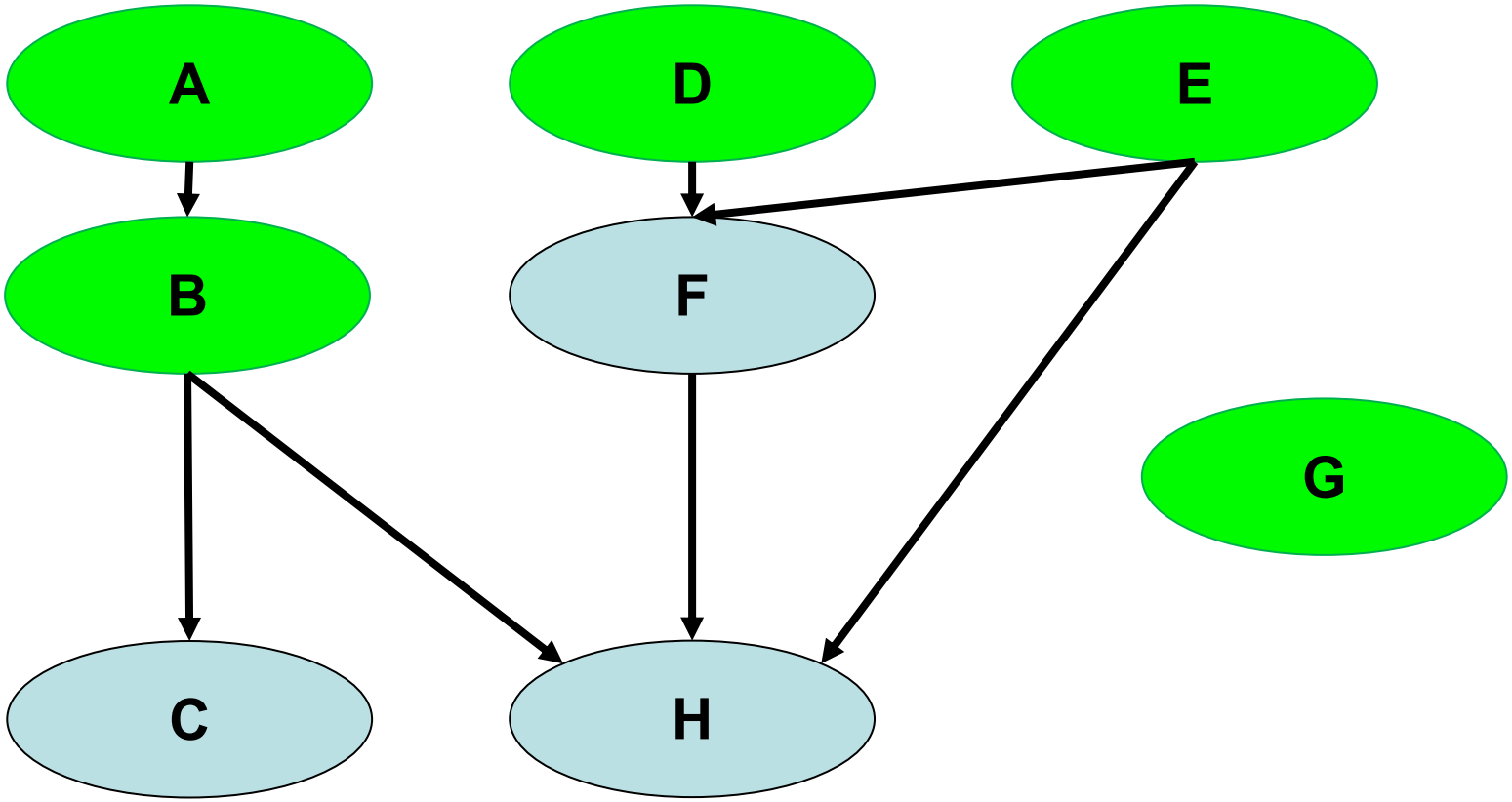
Topological Sort: Take 2



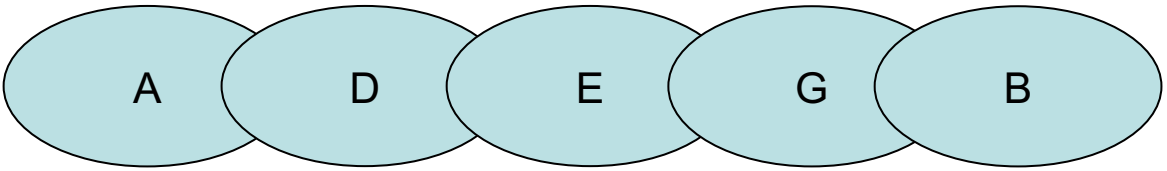
0-In-Degree Queue: { “B”, “F” }
In-Degree Map: { “A”:0, “B”:0, “C”:1, “D”:0, “E”: 0, “F”: 0, “G”:0, “H”:2 }



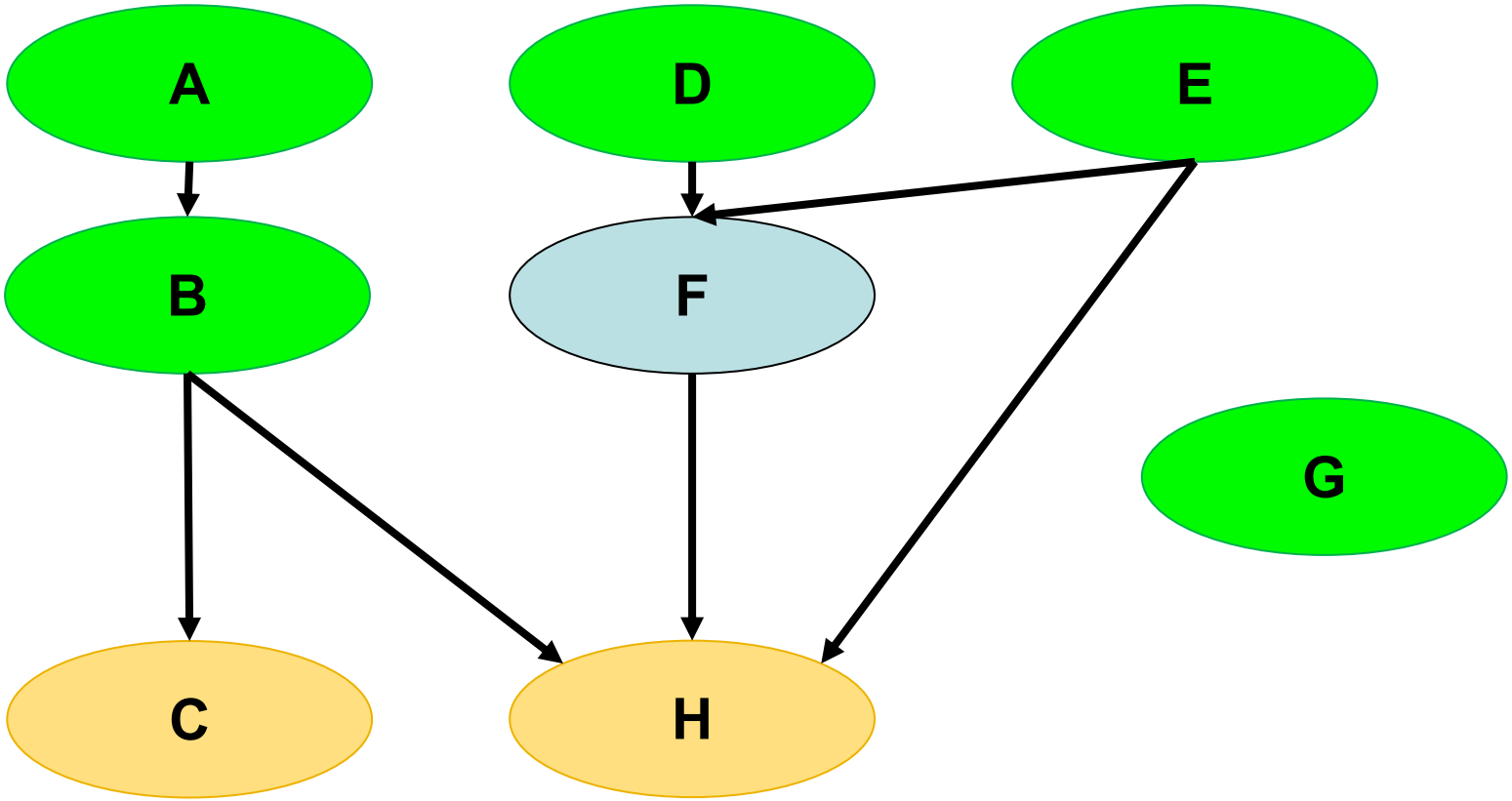
Topological Sort: Take 2



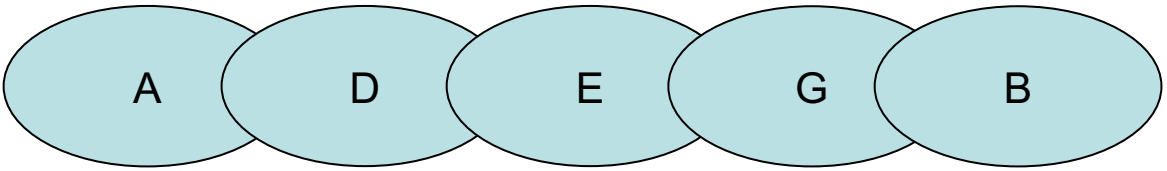
0-In-Degree Queue: { ~~"B"~~, "F" }
In-Degree Map: { "A":0, "B":0, "C":1, "D":0, "E": 0, "F": 0, "G":0, "H":2 }



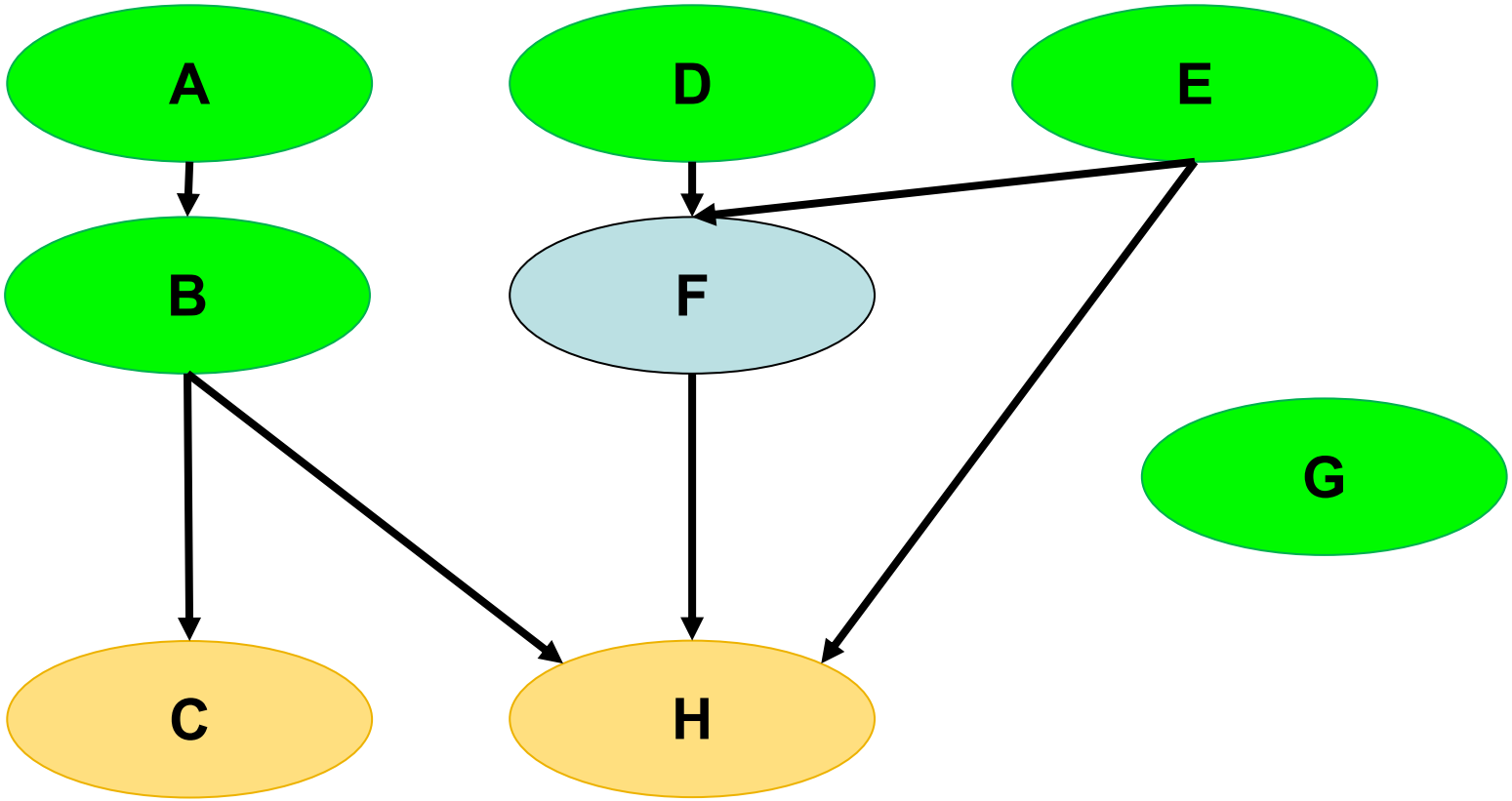
Topological Sort: Take 2



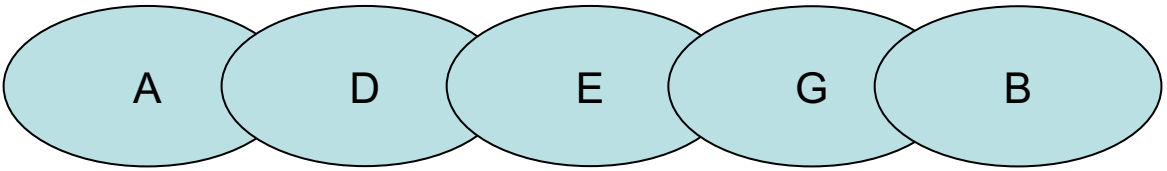
0-In-Degree Queue: { “F” }
In-Degree Map: { “A”:0, “B”:0, “C”:1, “D”:0, “E”: 0, “F”: 0, “G”:0, “H”:2 }



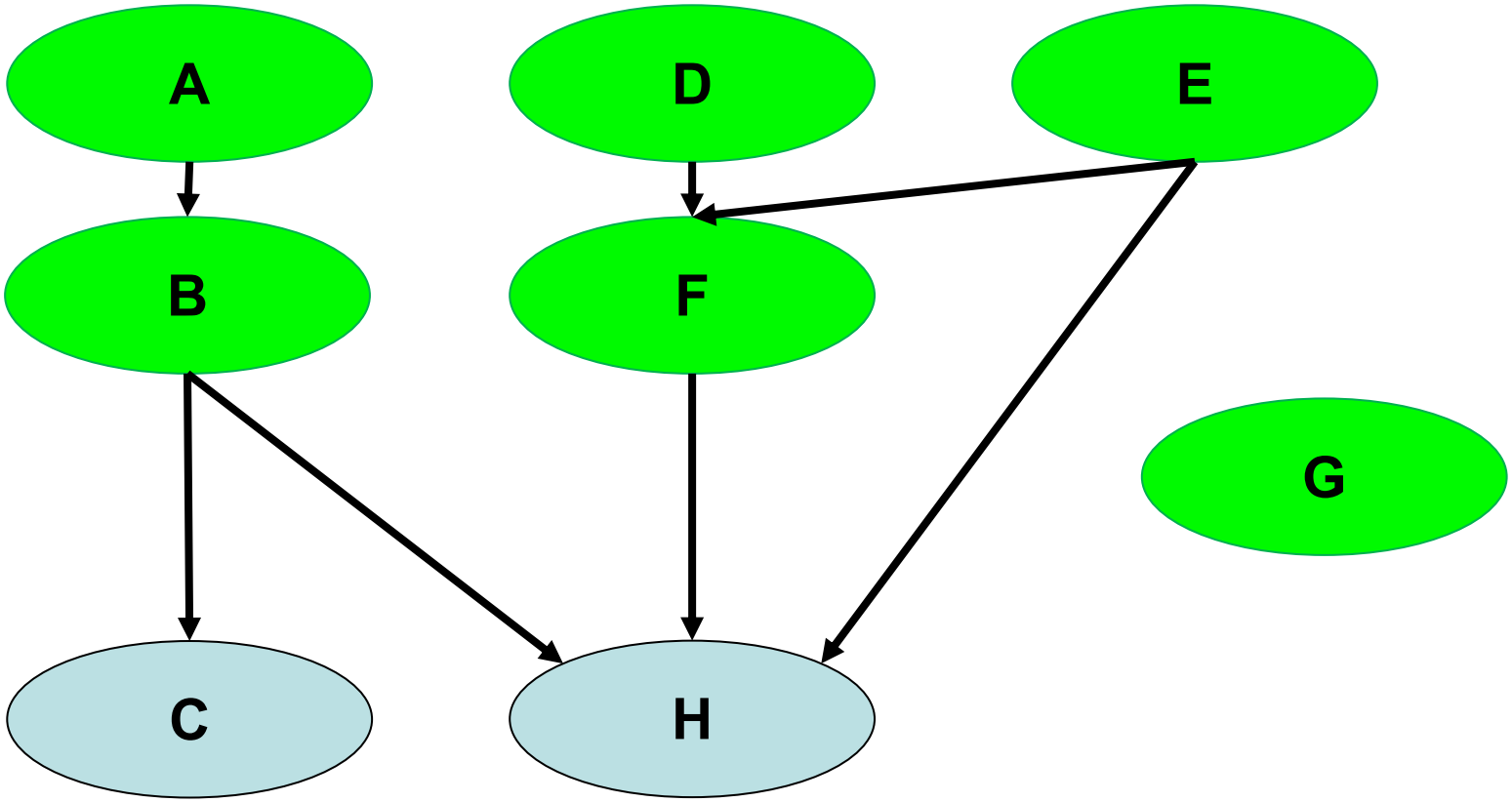
Topological Sort: Take 2



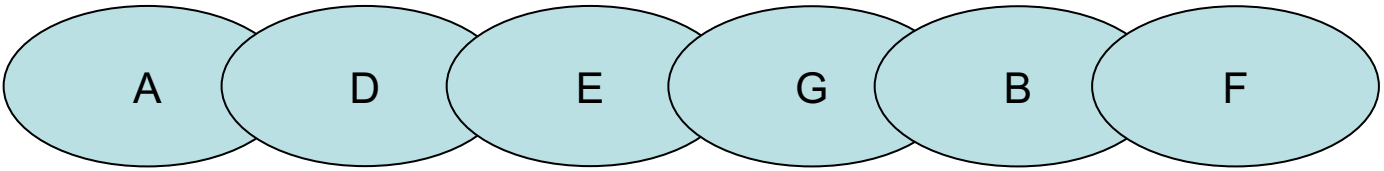
0-In-Degree Queue: { "F", "C" }
In-Degree Map: { "A":0, "B":0, "C":0, "D":0, "E": 0, "F": 0, "G":0, "H":1 }



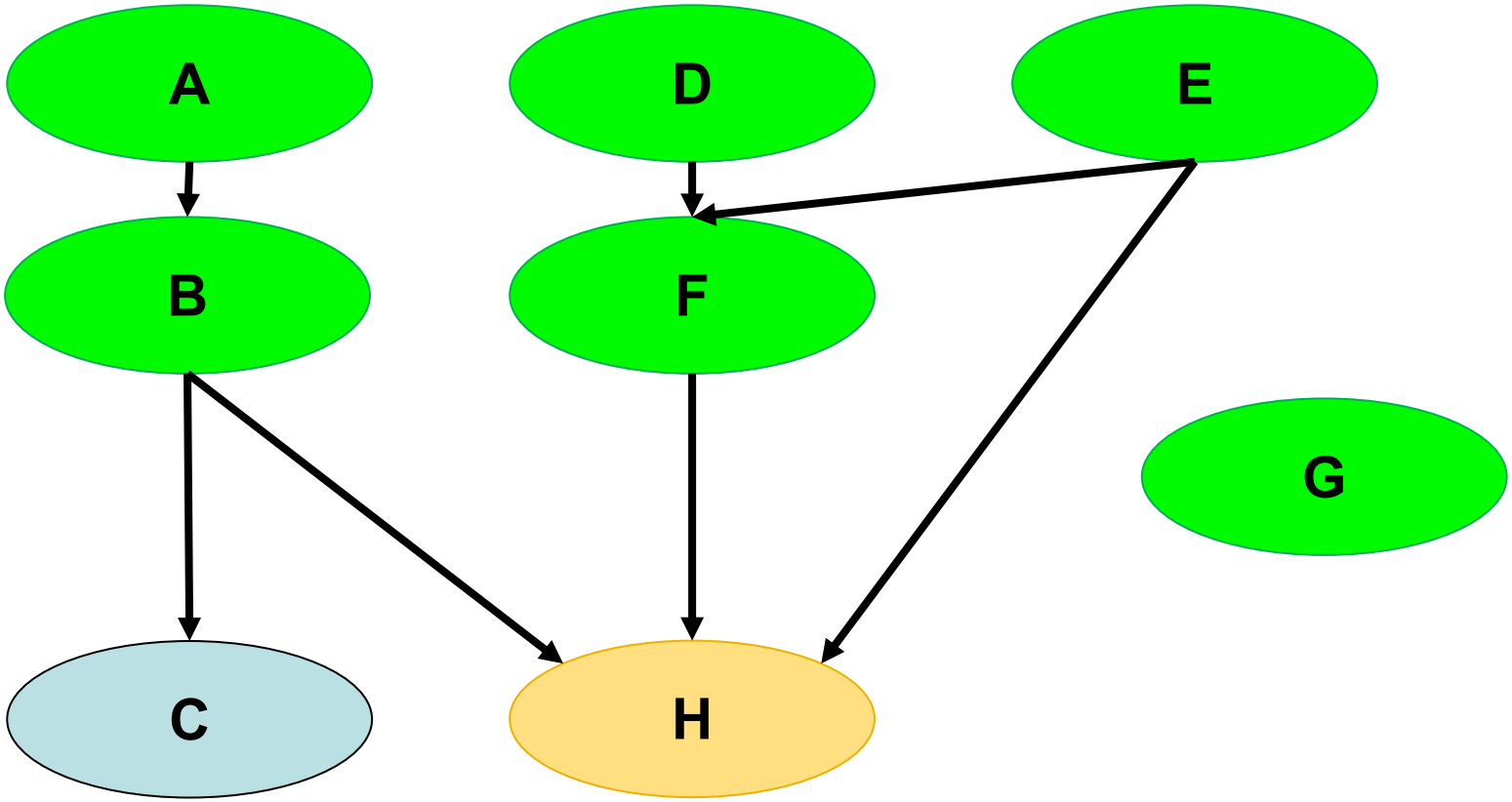
Topological Sort: Take 2



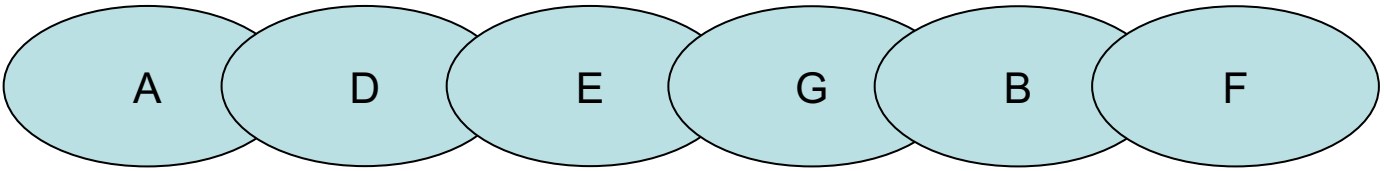
0-In-Degree Queue: { ~~"F"~~, "C" }
In-Degree Map: { "A":0, "B":0, "C":0, "D":0, "E": 0, "F": 0, "G":0, "H":1 }



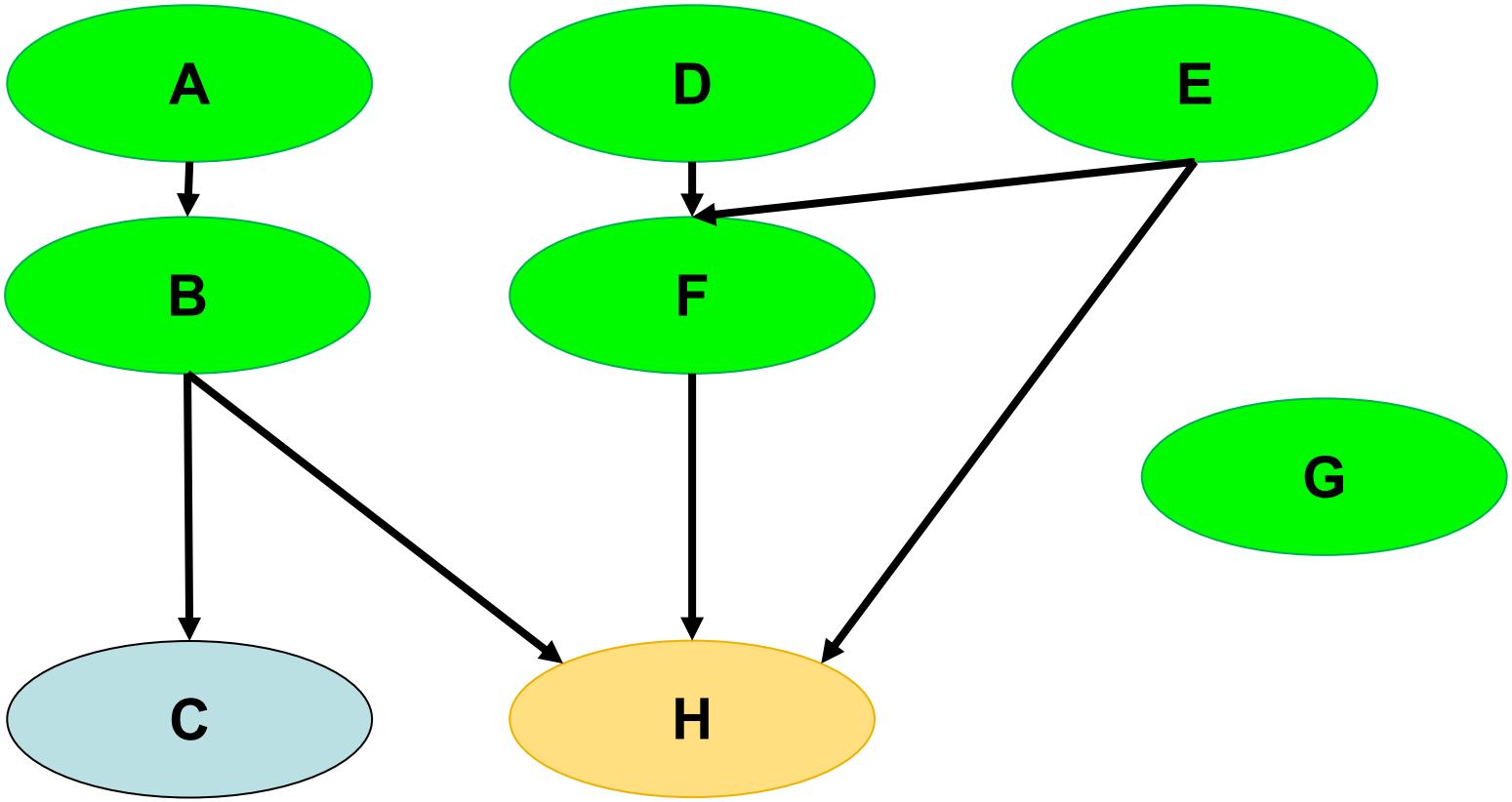
Topological Sort: Take 2



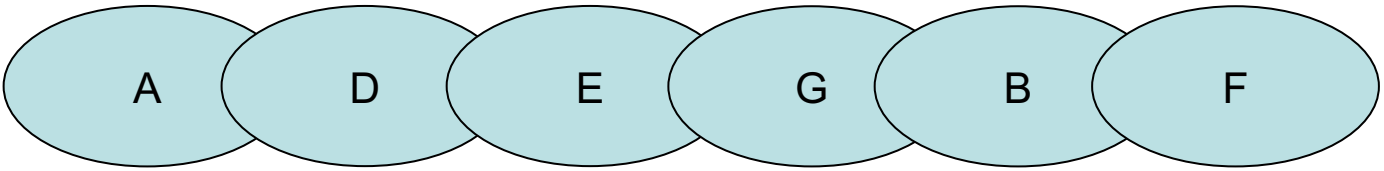
0-In-Degree Queue: { “C” }
In-Degree Map: { “A”:0, “B”:0, “C”:0, “D”:0, “E”: 0, “F”: 0, “G”:0, “H”:1 }



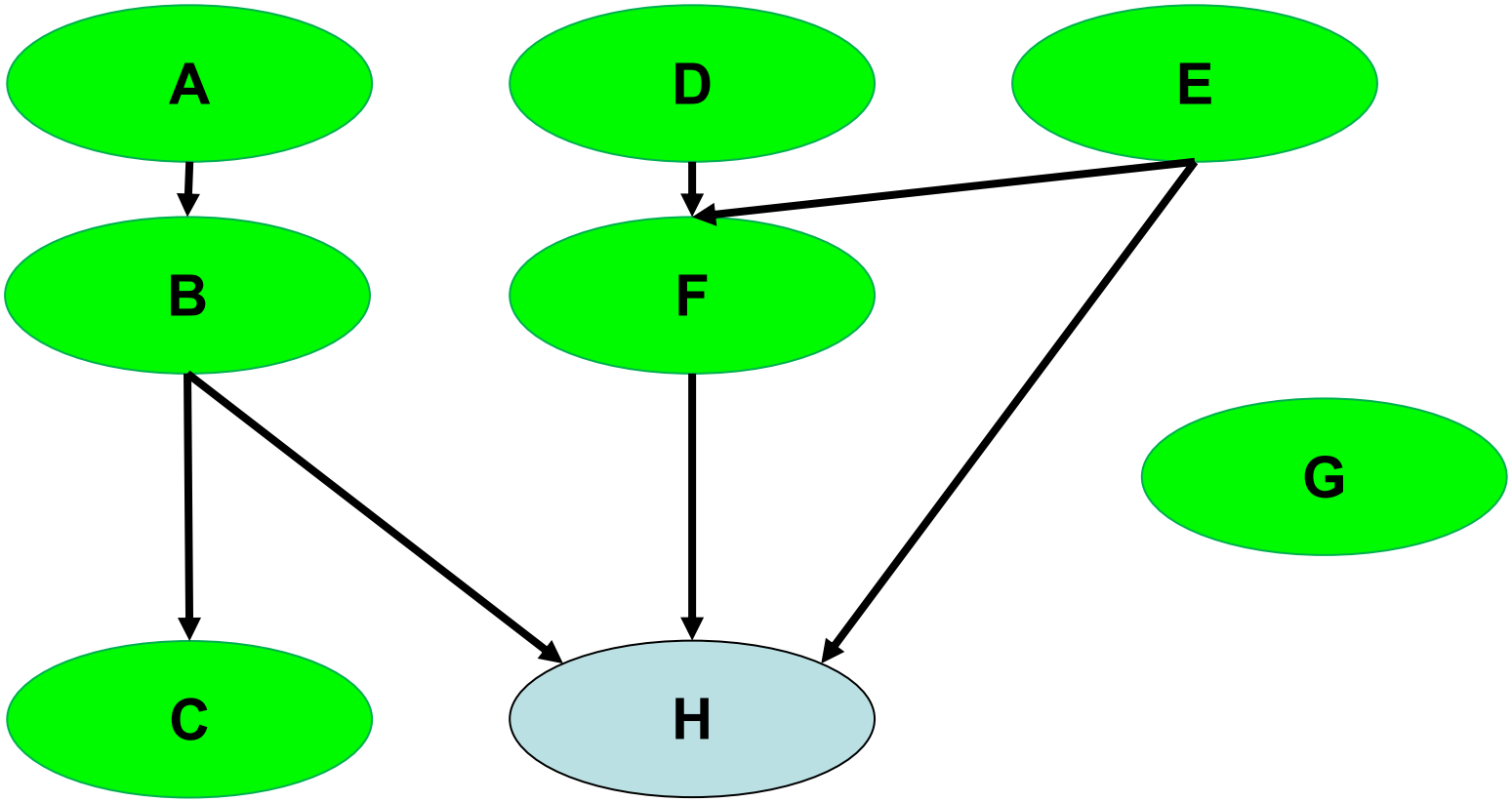
Topological Sort: Take 2



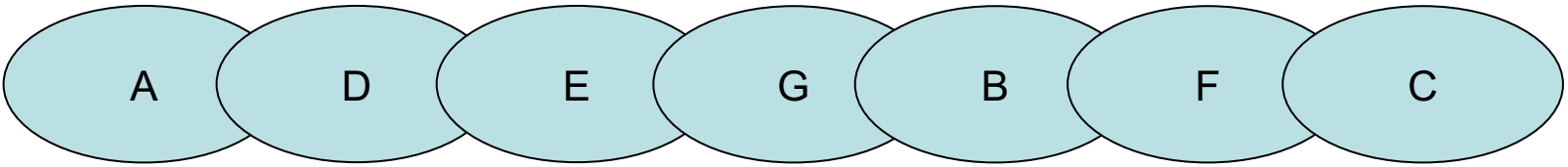
0-In-Degree Queue: { "C", "H" }
In-Degree Map: { "A":0, "B":0, "C":0, "D":0, "E": 0, "F": 0, "G":0, "H":0 }



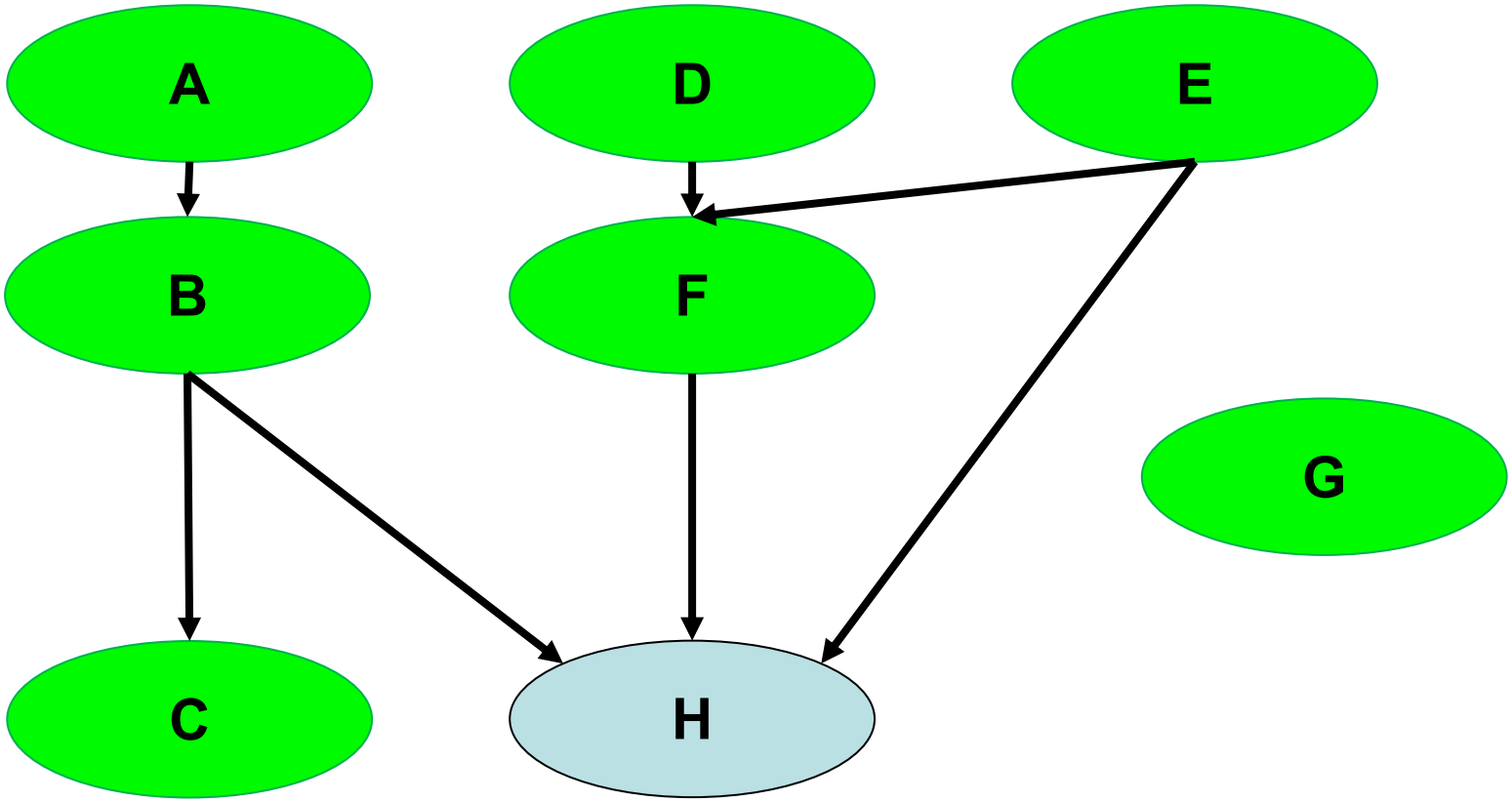
Topological Sort: Take 2



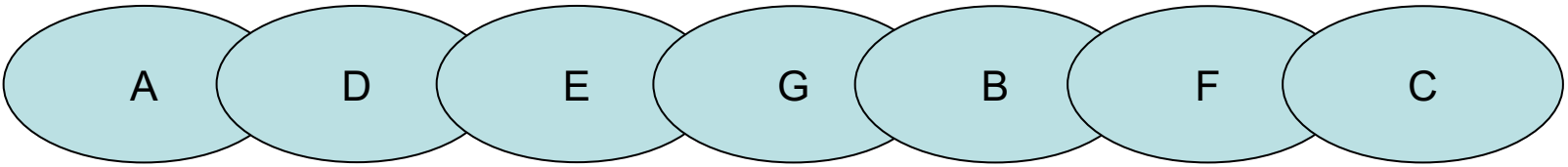
0-In-Degree Queue: { ~~"C"~~, "H" }
In-Degree Map: { "A":0, "B":0, "C":0, "D":0, "E": 0, "F": 0, "G":0, "H":0 }



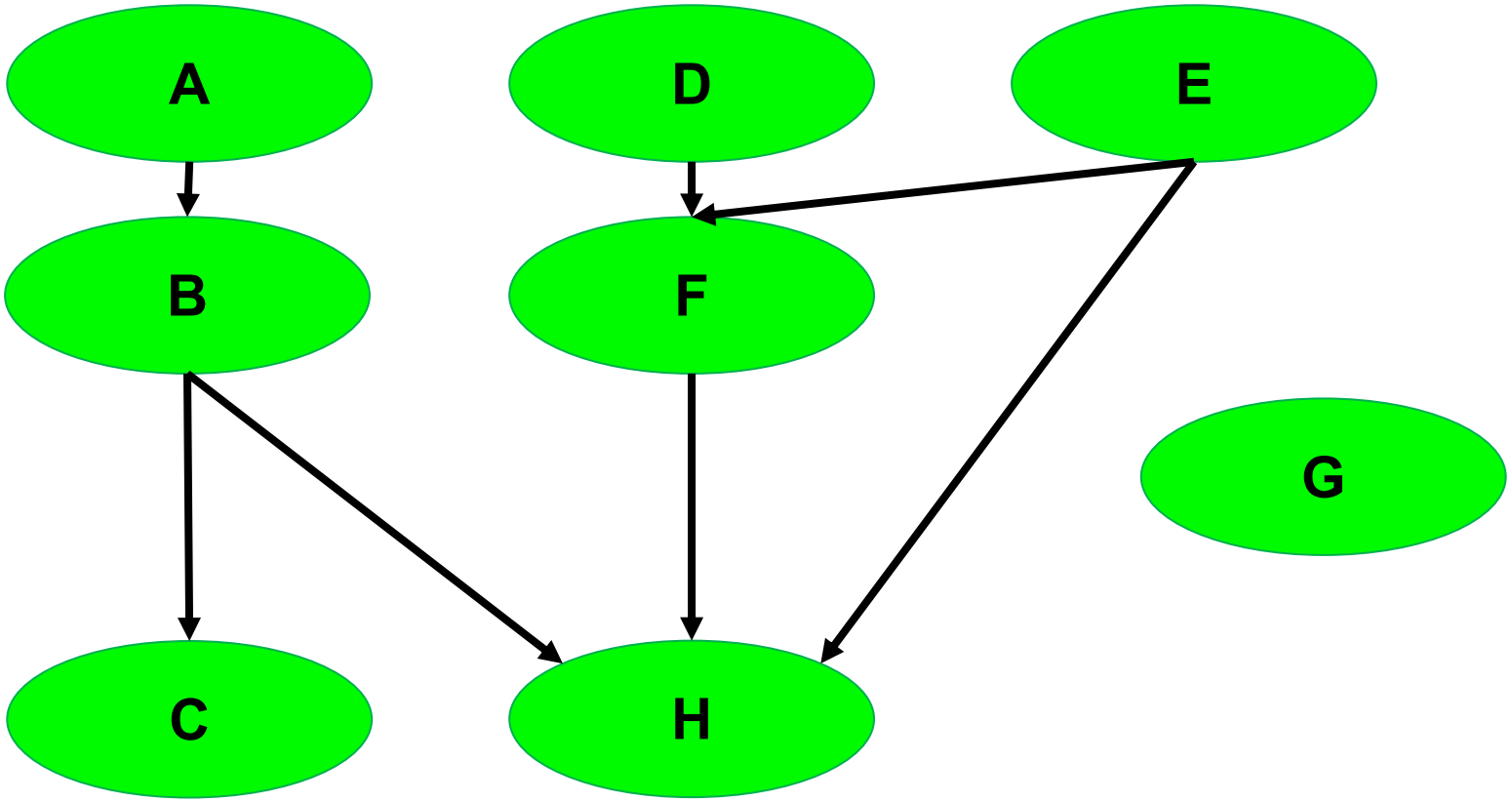
Topological Sort: Take 2



0-In-Degree Queue: { “H” }
In-Degree Map: { “A”:0, “B”:0, “C”:0, “D”:0, “E”: 0, “F”: 0, “G”:0, “H”:0 }

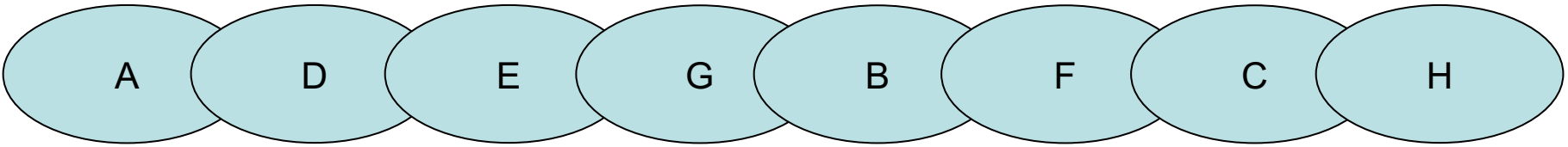


Topological Sort: Take 2

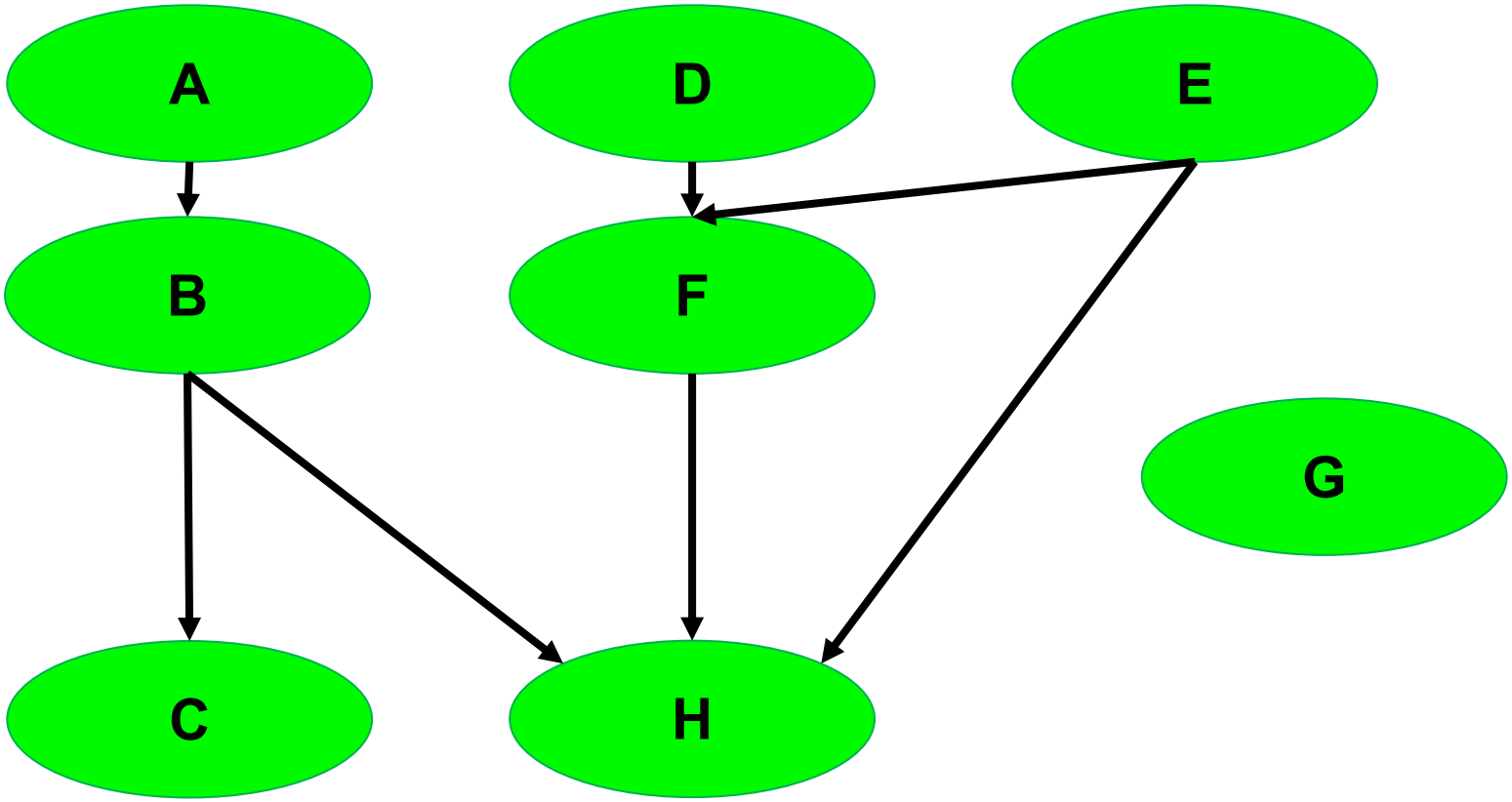


0-In-Degree Queue: { ~~"H"~~ }

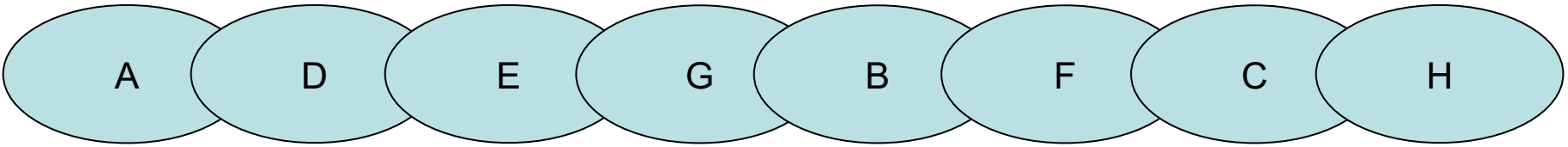
In-Degree Map: { "A":0, "B":0, "C":0, "D":0, "E": 0, "F": 0, "G":0, "H":0 }



Topological Sort: Take 2



0-In-Degree Queue: { }
In-Degree Map: { “A”:0, “B”:0, “C”:0, “D”:0, “E”: 0, “F”: 0, “G”:0, “H”:0 }



Plan for Today

- **Topological Sort**
  - Kahn's Algorithm
  - Recursive DFS
  - Spreadsheets
- Announcements

Kahn's Algorithm

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

- **This algorithm doesn't modify the passed-in graph! ☺**
- **Have we handled all edge cases?**
- **What is the runtime of this algorithm?**

Kahn's Algorithm

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

- **This algorithm doesn't modify the passed-in graph! ☺**
- **Have we handled all edge cases?**
- **What is the runtime of this algorithm?**

Kahn's Algorithm: Cycles

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

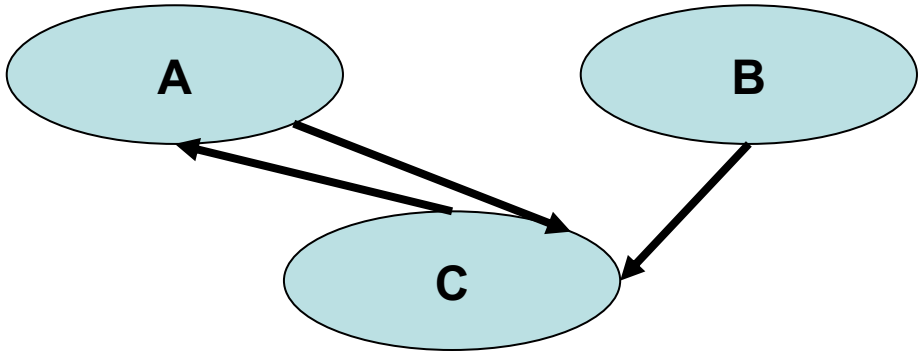
Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.



Kahn's Algorithm: Cycles

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

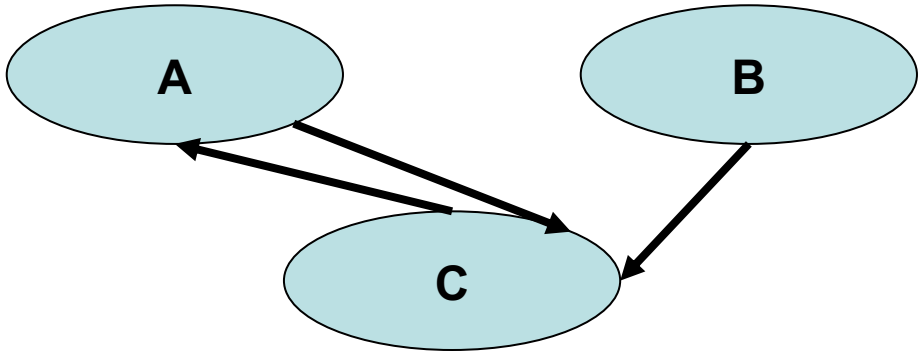
ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

0-In-Degree Queue: { "B" }
In-Degree Map: { A":1, "B":0, "C": 2 }

Ordering: { }



Kahn's Algorithm: Cycles

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

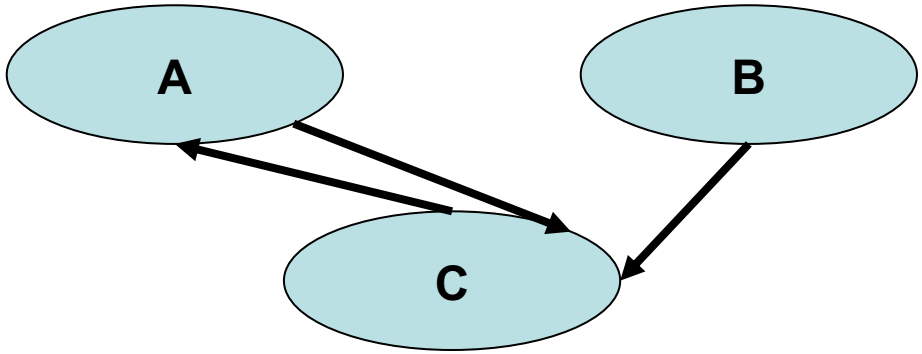
ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

0-In-Degree Queue: { "B" }
In-Degree Map: { A":1, "B":0, "C": 2 }

Ordering: { }



Kahn's Algorithm: Cycles

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

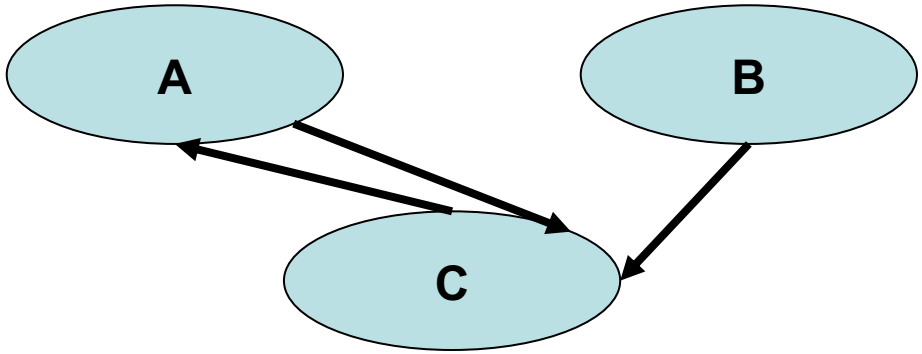
ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

0-In-Degree Queue: { ~~"B"~~ }
In-Degree Map: { A":1, "B":0, "C": 2 }

Ordering: { "B" }



Kahn's Algorithm: Cycles

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

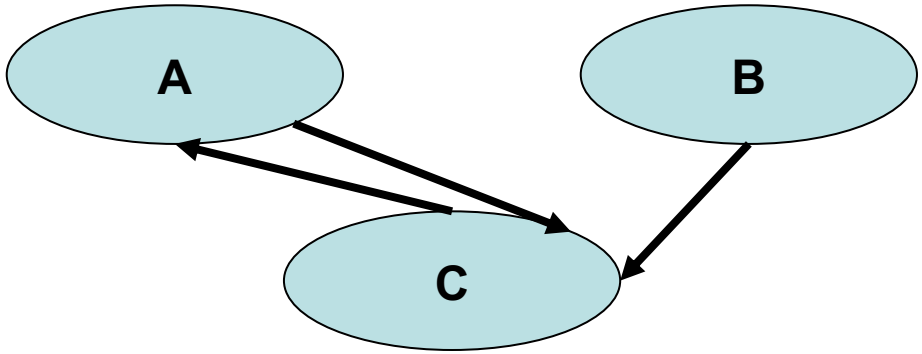
ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

0-In-Degree Queue: { }
In-Degree Map: { A":1, "B":0, "C": 1 }

Ordering: { "B" }



Kahn's Algorithm: Cycles

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

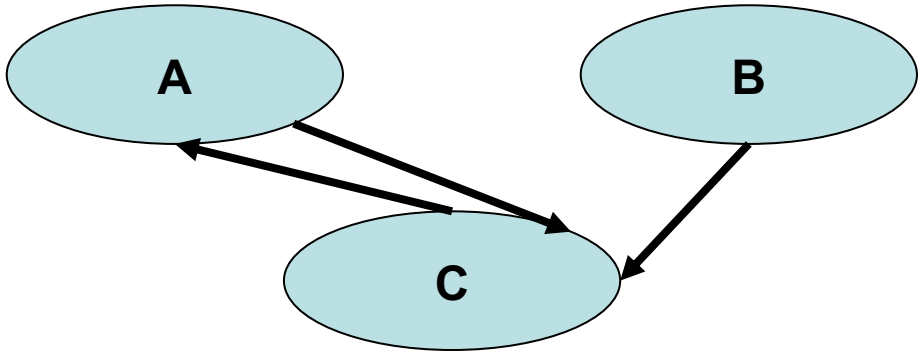
ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

0-In-Degree Queue: { }
In-Degree Map: { A":1, "B":0, "C": 1 }

Ordering: { "B" }



Kahn's Algorithm: Cycles

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

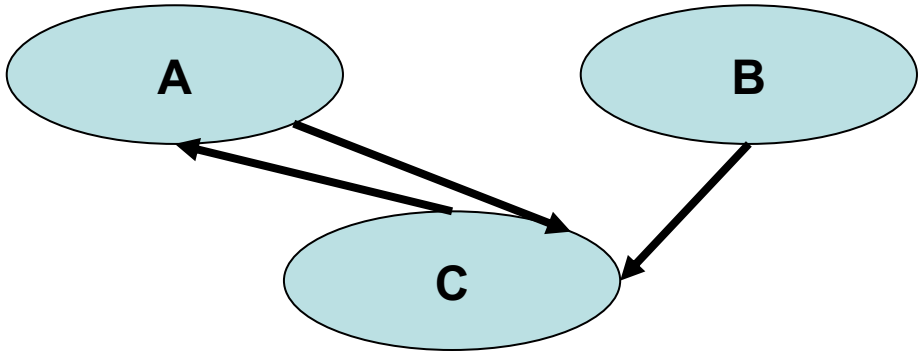
ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

0-In-Degree Queue: { }
In-Degree Map: { A":1, "B":0, "C": 1 }

Ordering: { "B" }



Kahn's Algorithm: Cycles

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

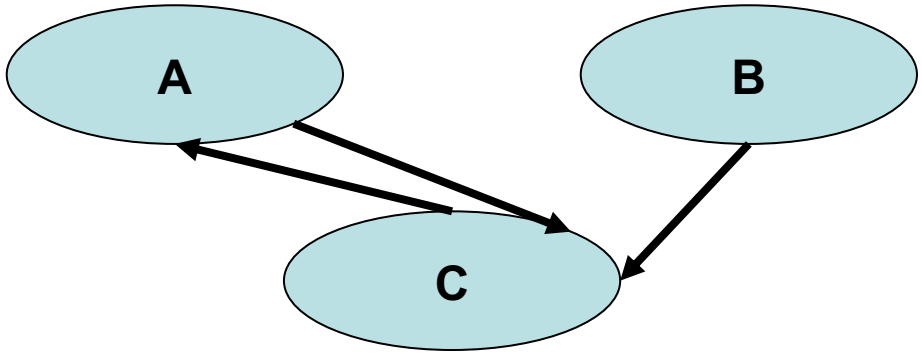
ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

0-In-Degree Queue: { }
In-Degree Map: { A":1, "B":0, "C": 1 }

Ordering: { "B" }



Kahn's Algorithm: Cycles

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

ordering += *v*.

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

If all vertices processed, success! Otherwise, there is a cycle.

- **This algorithm doesn't modify the passed-in graph! ☺**
- **Have we handled all edge cases?**
- **What is the runtime of this algorithm?**

Kahn's Algorithm: Runtime

For graph with V vertexes and E edges

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

- Dequeue the first vertex *v* from the queue.
- *ordering* += *v*.
- Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.
- *queue* += {any neighbors whose in-degree is now 0}.

Kahn's Algorithm: Runtime

For graph with V vertexes and E edges

O(V)

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

O(V)

Repeat until queue is empty:

O(1)

Dequeue the first vertex *v* from the queue.

ordering += *v*.

O(E)

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

queue += {any neighbors whose in-degree is now 0}.

O(VE)

Kahn's Algorithm: Runtime

For graph with V vertexes and E edges

O(V)

map := {each vertex → its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

O(V)

Repeat until queue is empty:

O(1)

Dequeue the first vertex *v* from the queue.

ordering += *v*.

O(E) *total*

Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

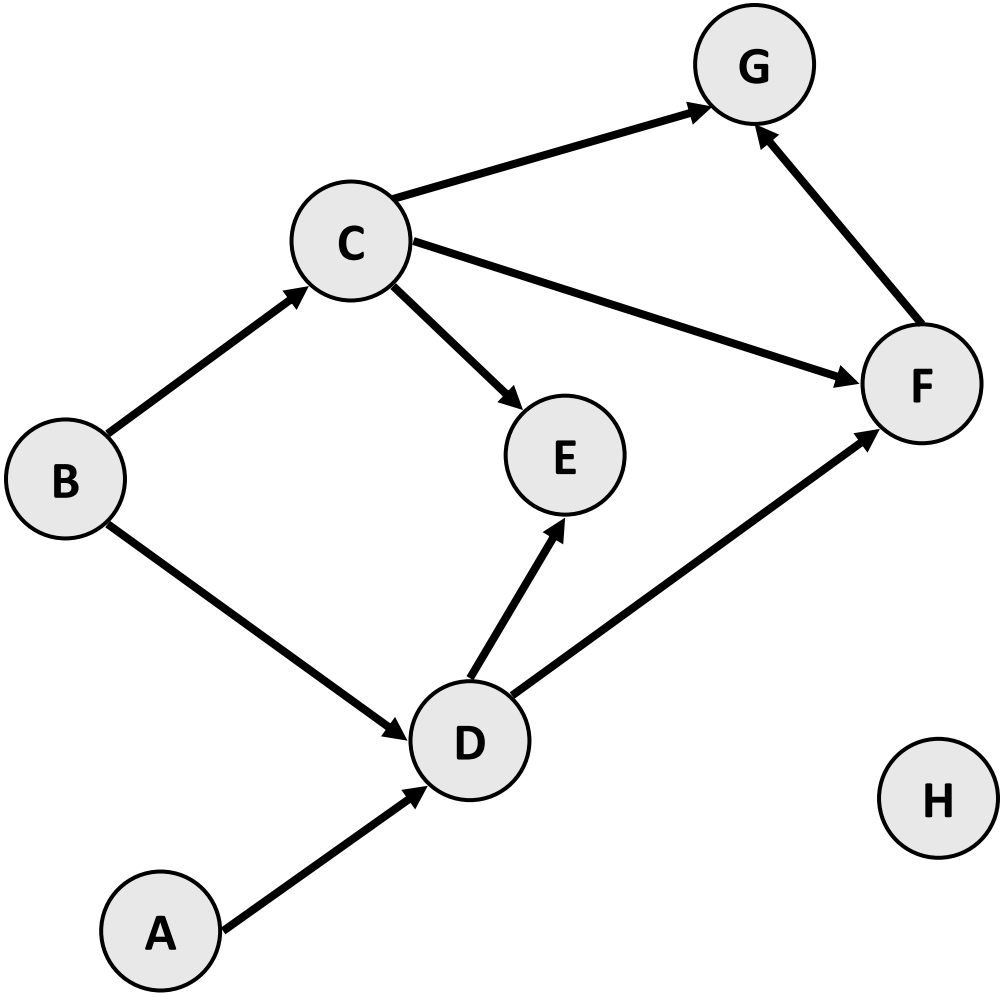
queue += {any neighbors whose in-degree is now 0}.

O(V + E)

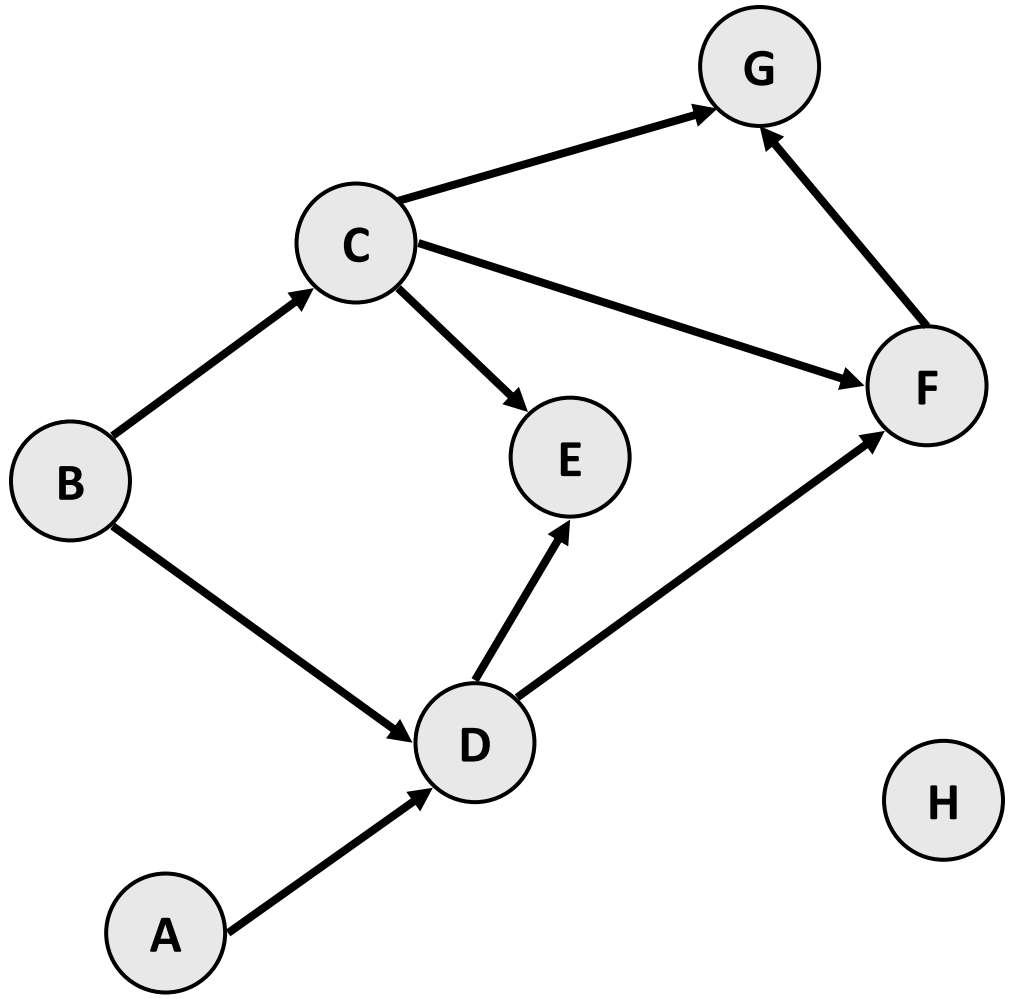
Plan for Today

- **Topological Sort**
  - Kahn's Algorithm
  - Recursive DFS
  - Spreadsheets
- Announcements

Revisiting Recursive DFS

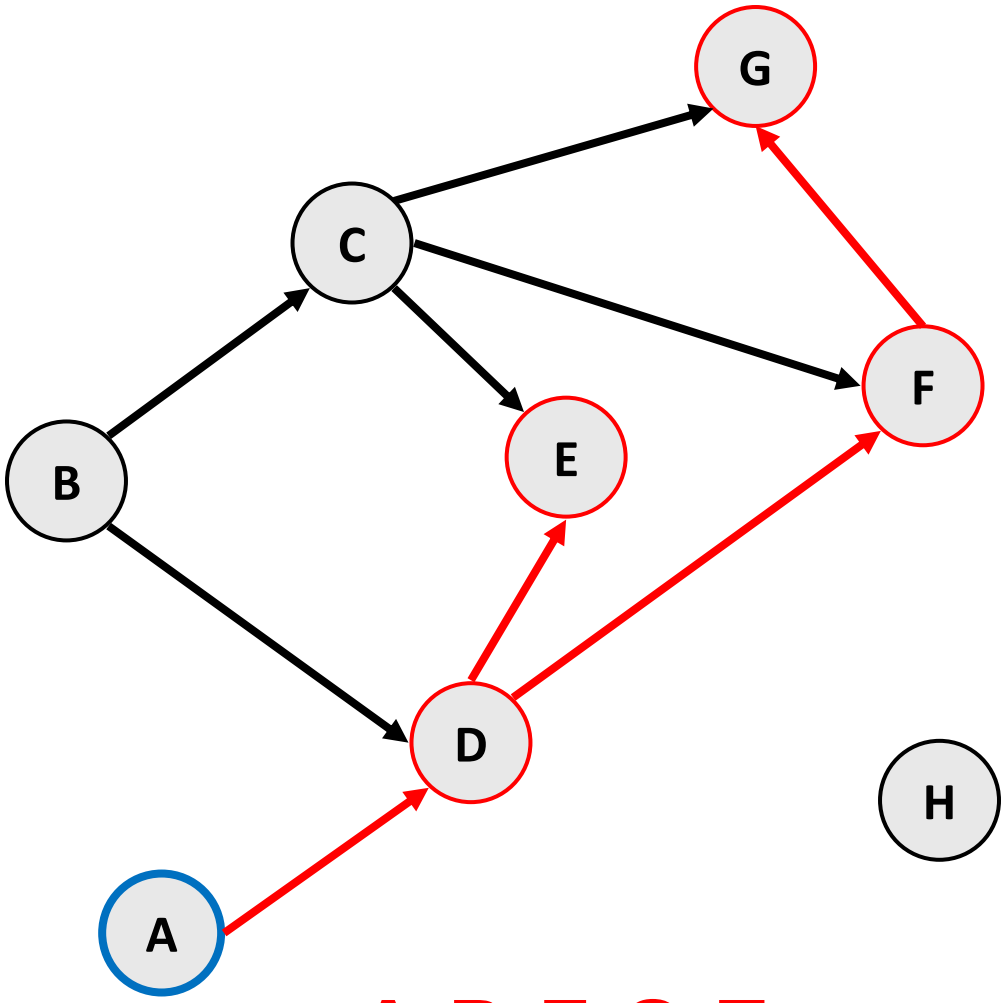


Revisiting Recursive DFS



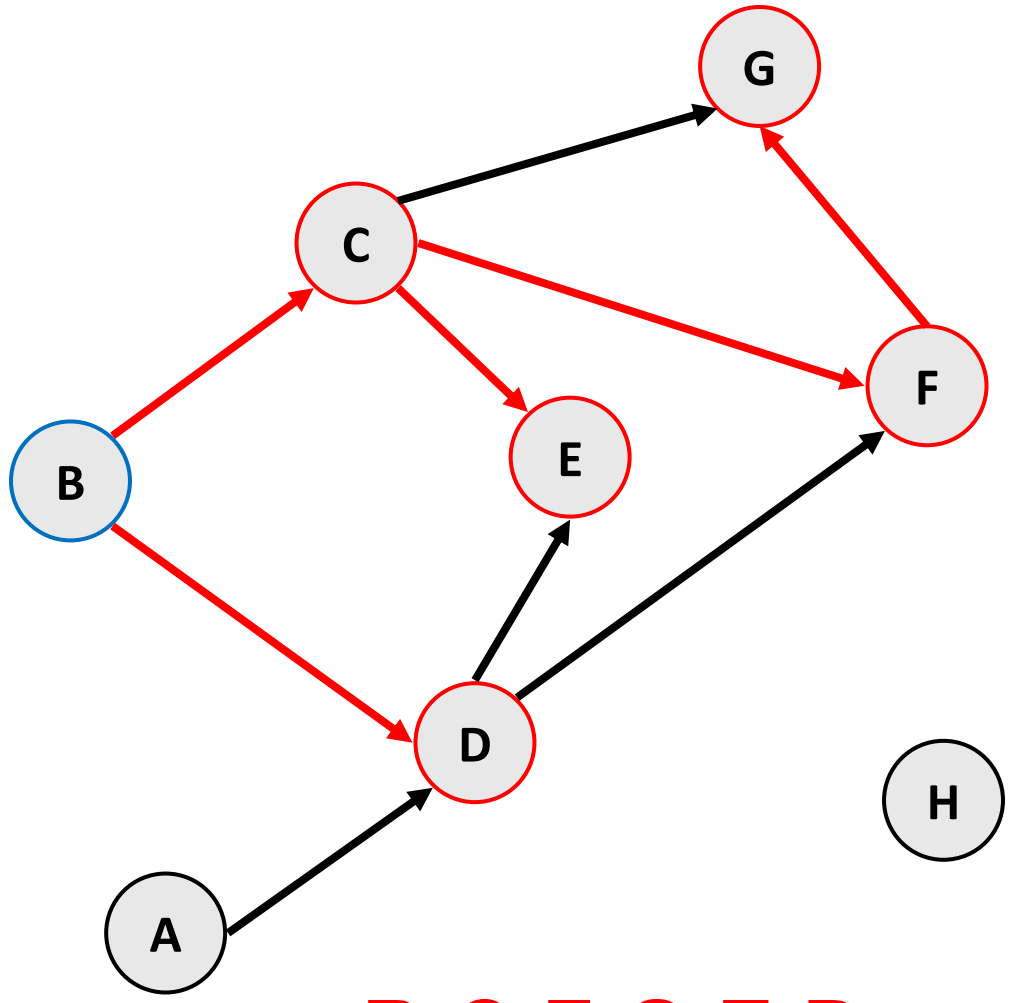
Revisiting Recursive DFS

- **Key idea:** visiting node *a* means we must have visited all nodes that have *a* as a prerequisite.



Revisiting Recursive DFS

- **Key idea:** visiting node *a* means we must have visited all nodes that have *a* as a prerequisite.
- But different starting points may yield different subsets. Is there a way we can combine them into one complete topological ordering?



B-C-F-G-E-D

Revisiting Recursive DFS

A D F G E

B C F G E D

C F G E

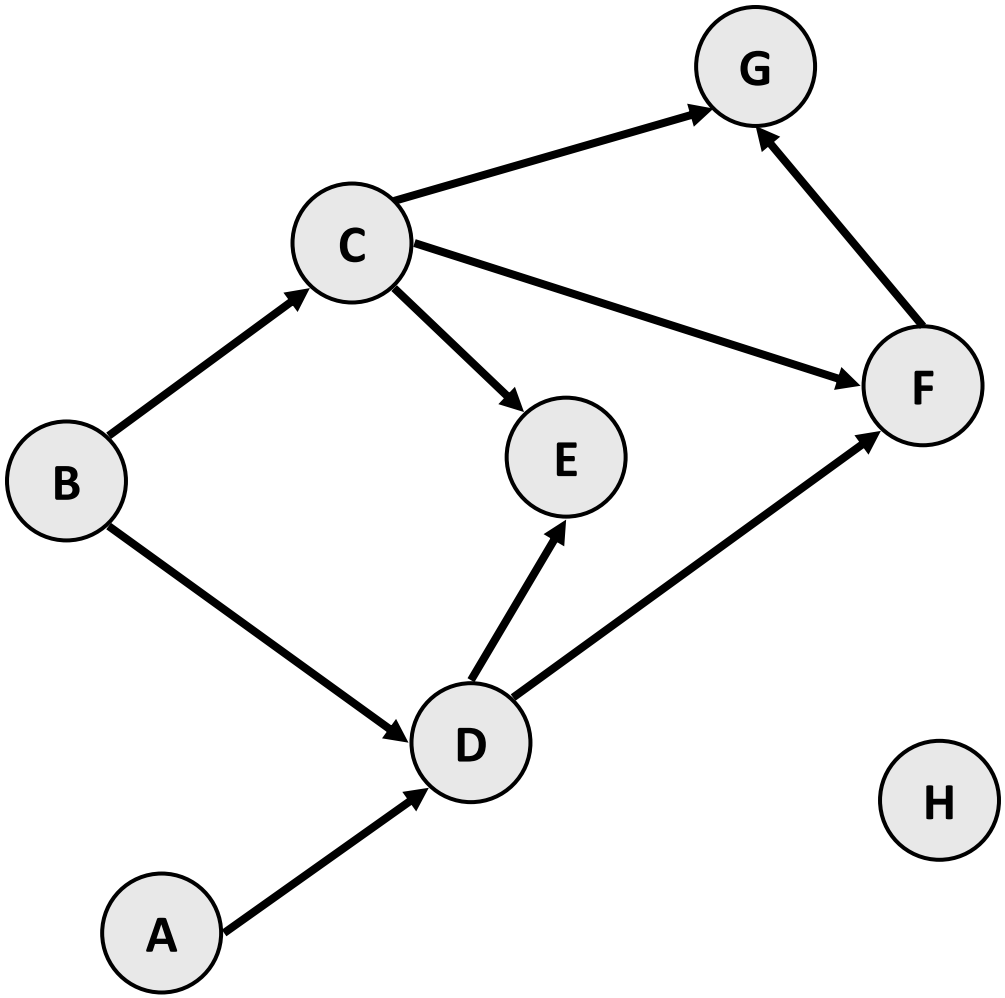
D F G E

E

F G

G

H

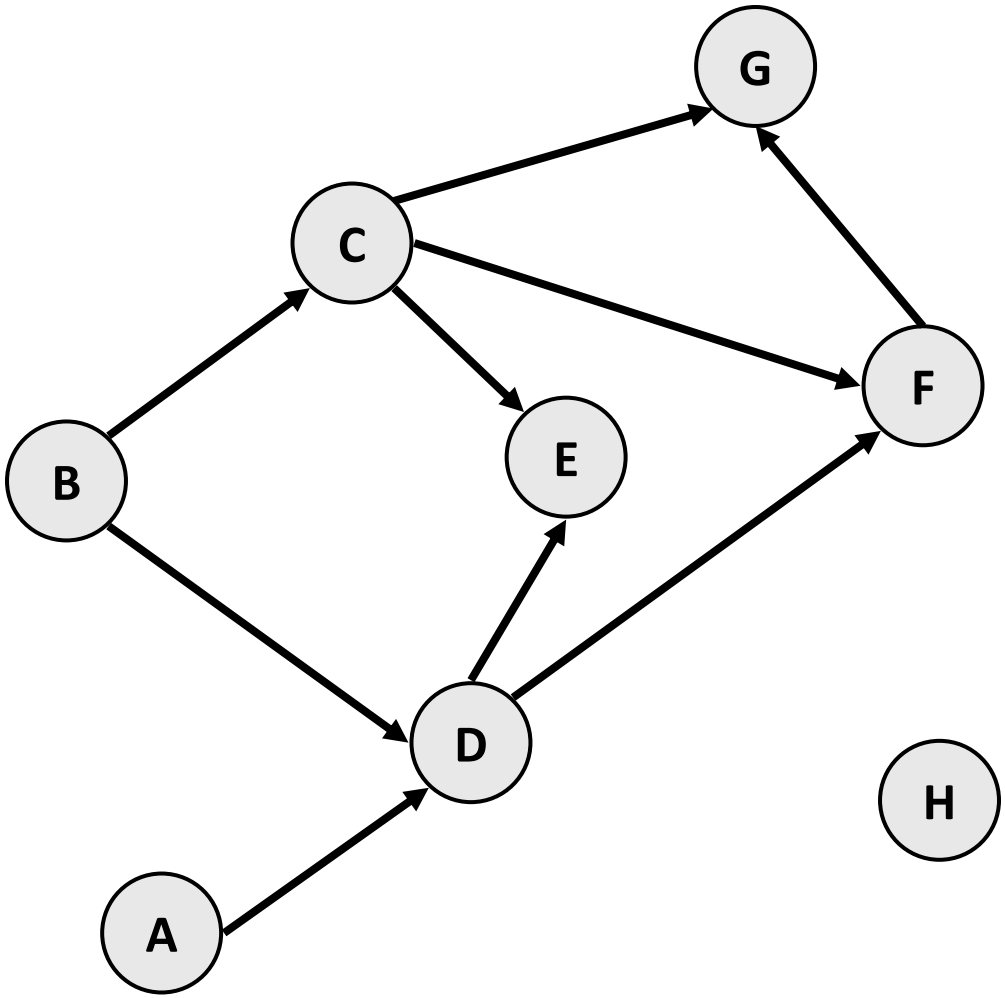


Revisiting Recursive DFS

A D F G E

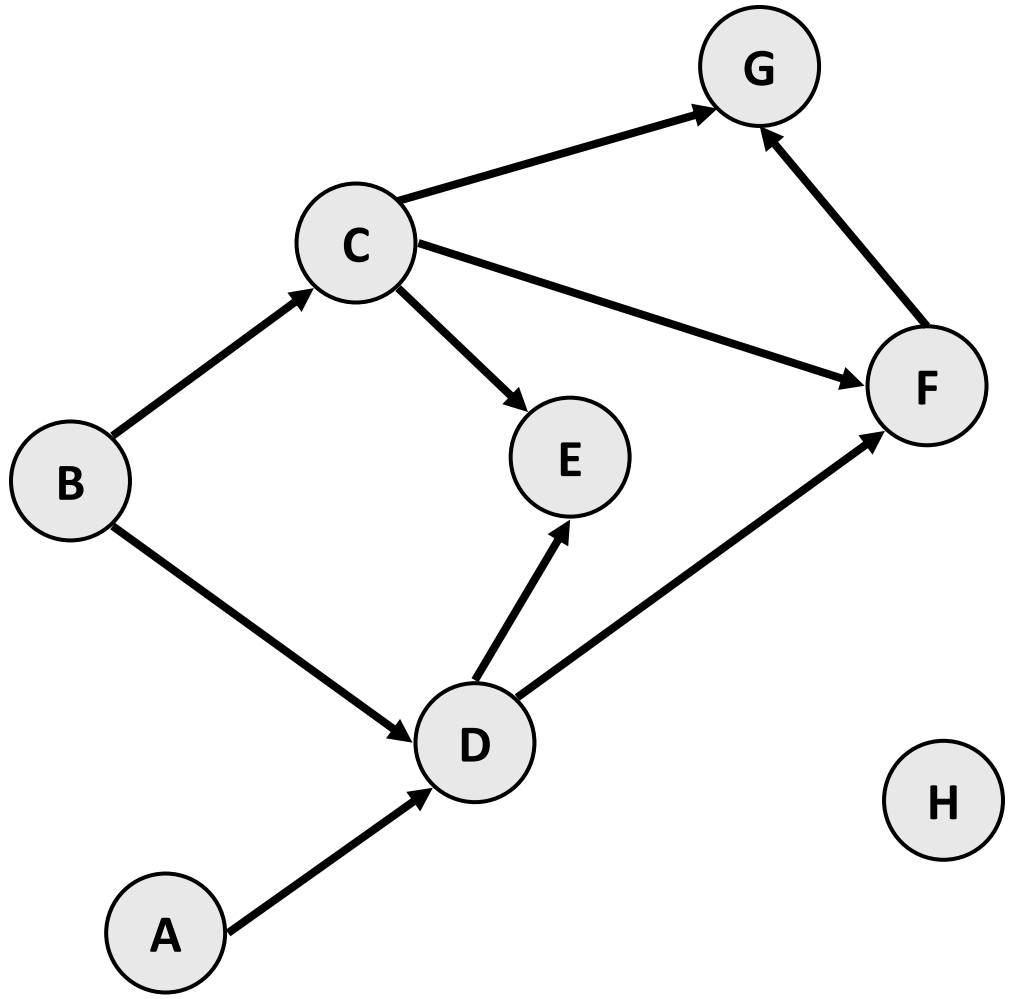
B C

H



Revisiting Recursive DFS

H B C A D F G E



Revisiting Recursive DFS

H

G

F G

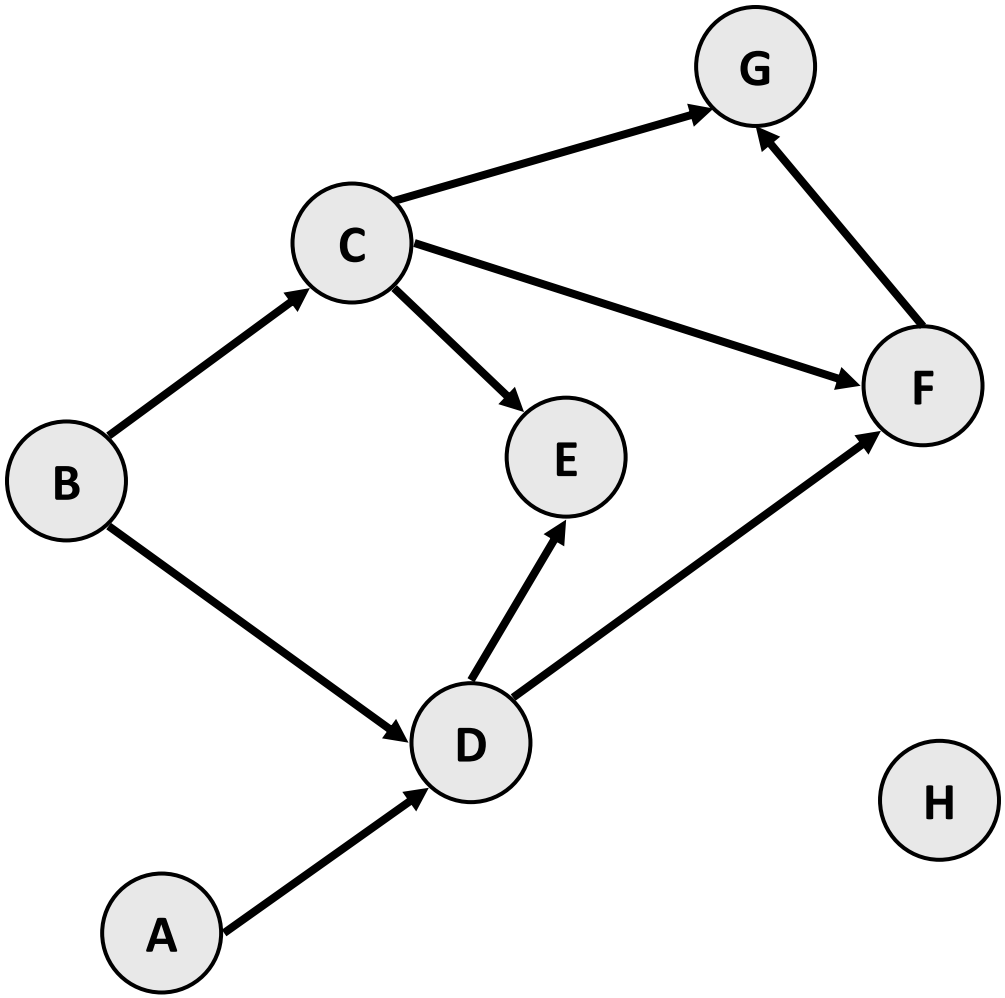
E

D F G E

C F G E

B C F G E D

A D F G E



Revisiting Recursive DFS

H

G

F

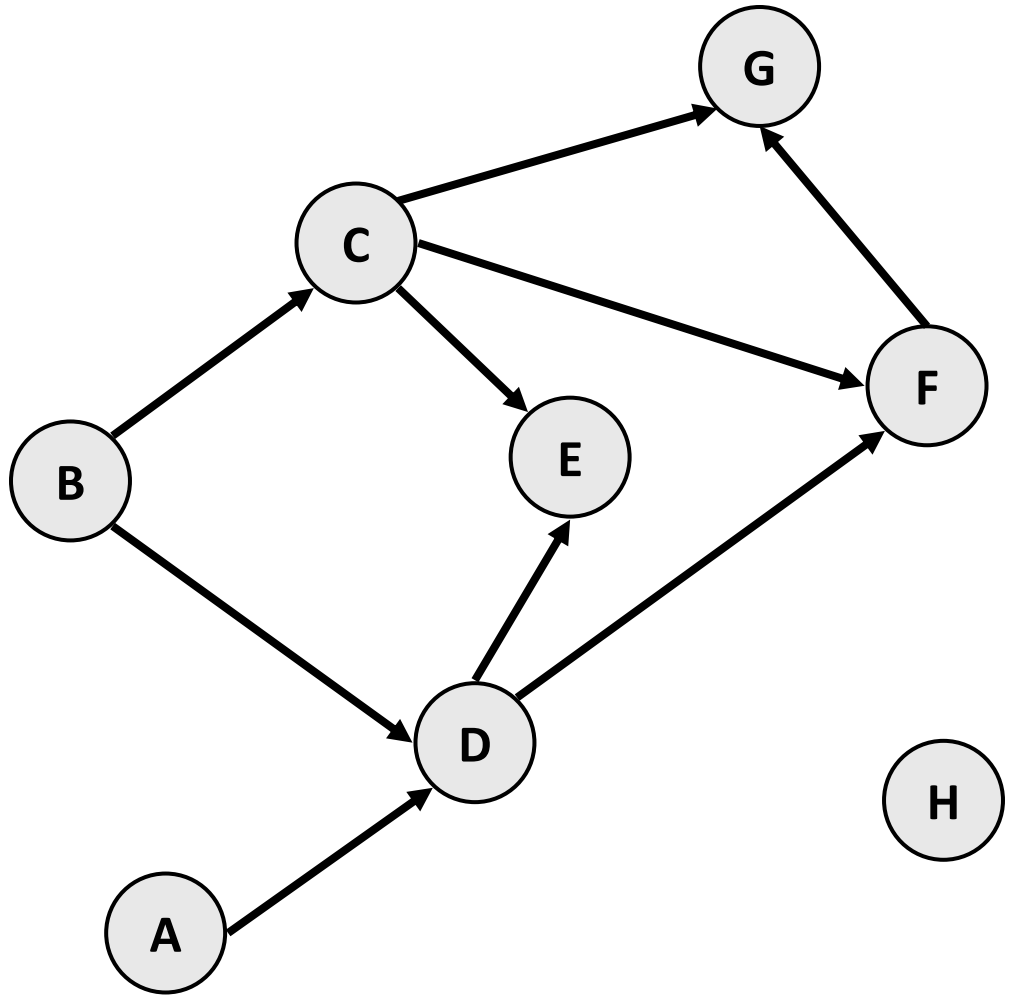
E

D

C

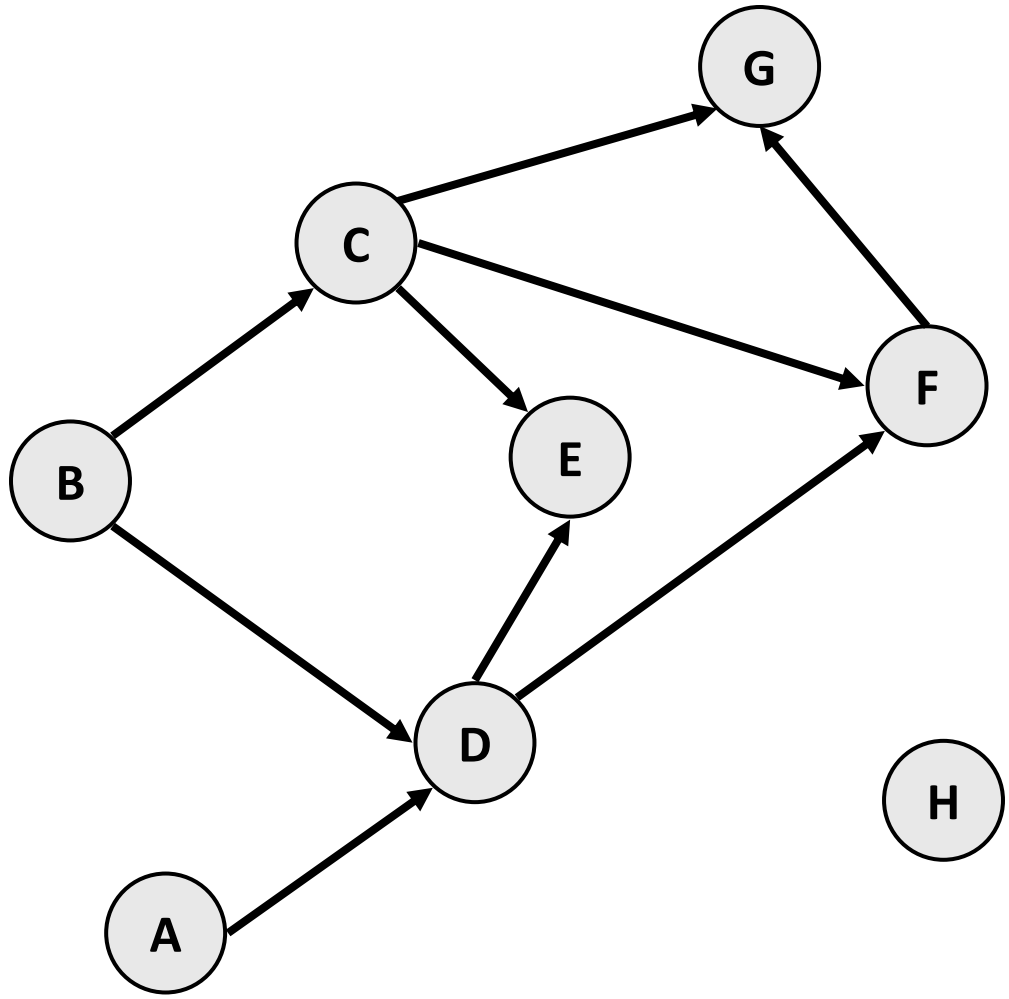
B

A



Revisiting Recursive DFS

A B C D E F G H



Topological Sort: DFS

- function **topologicalSortDFS**():

  L = { }. // to store the sorted vertexes

  while graph contains any unmarked vertexes:

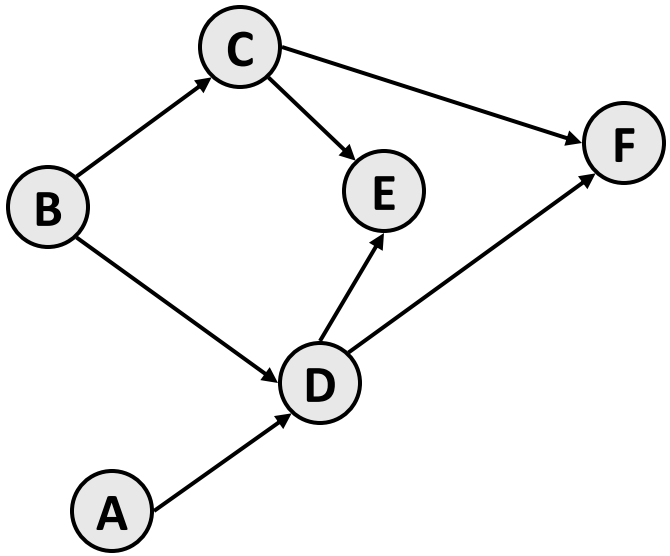
  - select any unmarked vertex *v*.
  - **visit**(*L*, *v*).

- function **visit**(*L*, v):

  if *v* is unmarked:

  - for each neighbor *n* of *v:*

    visit(*L*, *n*).
  - mark *v*.
  - add *v* to front of L.



Topological Sort: DFS

- function **topologicalSortDFS**():

  L = { }. // to store the sorted vertexes

  while graph contains any unmarked vertexes:

  - select any unmarked vertex *v*.
  - **visit**(*L*, *v*).

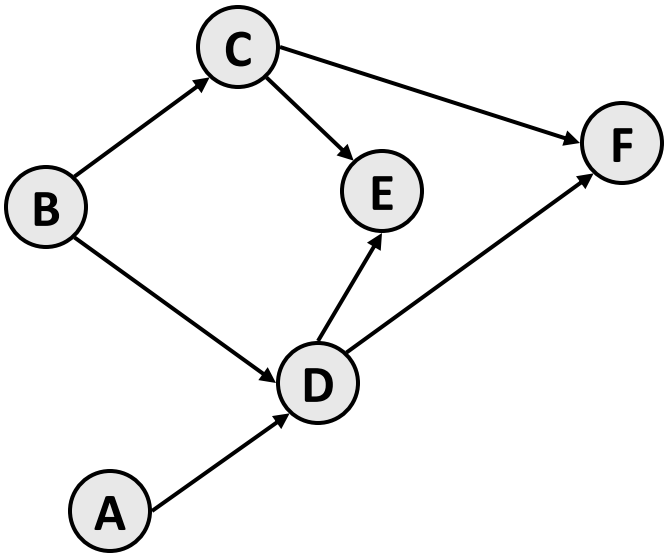
- function **visit**(*L*, v):

  if *v* has a temp.mark, error.

  if *v* is unmarked:

  - temp.mark *v.*
  - for each neighbor *n* of *v:*

    visit(*L*, *n*).
  - mark *v*.
  - add *v* to front of L.



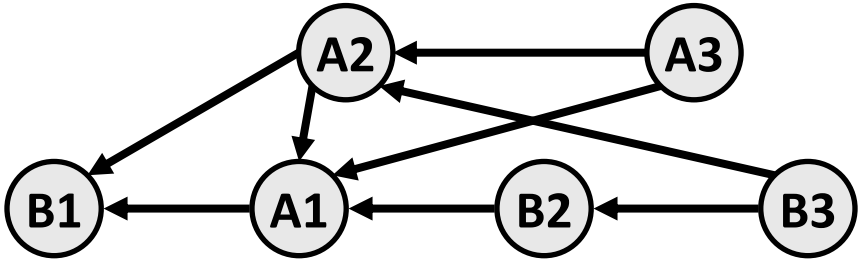
Plan for Today

- **Topological Sort**
  - Kahn's Algorithm
  - Recursive DFS
  - Spreadsheets
- Announcements

Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

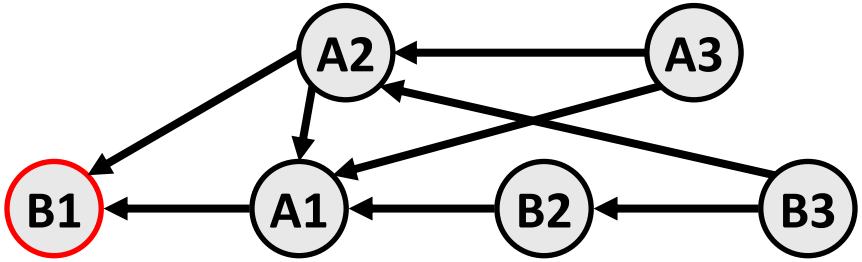
	A	B
1	**20** =B1*2	**10**
2	**30** =A1+B1	**25** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

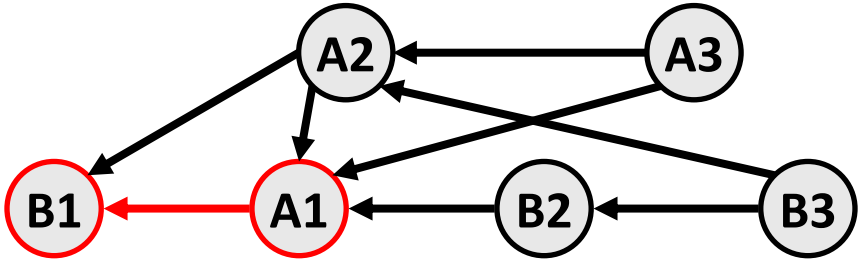
	A	B
1	**20** =B1*2	**15**
2	**30** =A1+B1	**25** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

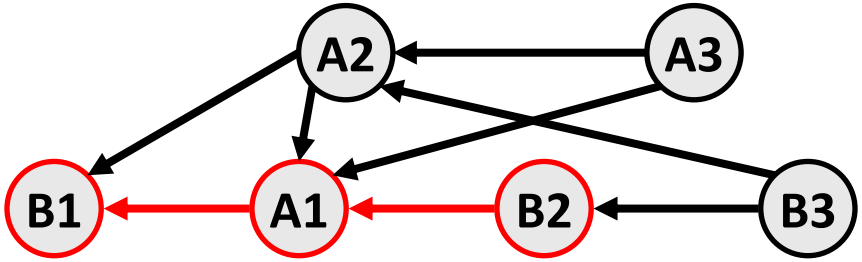
	A	B
1	**30** =B1*2	**15**
2	**30** =A1+B1	**25** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

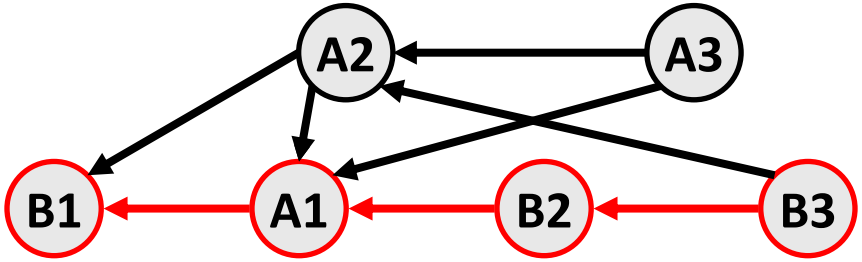
	A	B
1	**30** =B1*2	**15**
2	**30** =A1+B1	**35** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

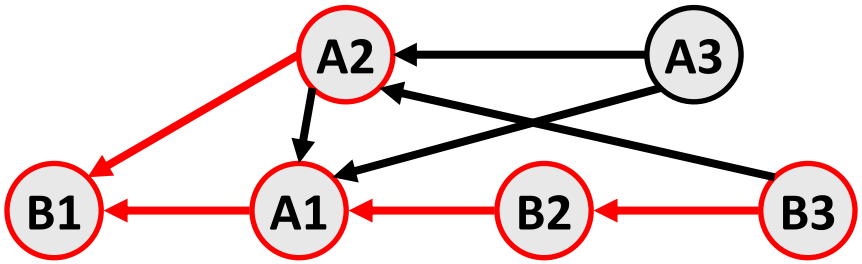
	A	B
1	**30** =B1*2	**15**
2	**30** =A1+B1	**35** =A1+5
3	**50** =SUM(A1:A2)	**1.166** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

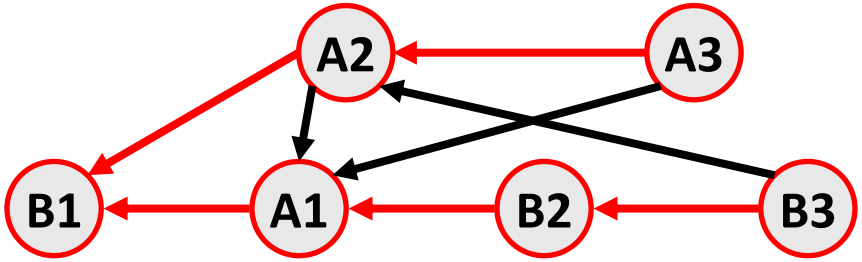
	A	B
1	**30** =B1*2	**15**
2	**45** =A1+B1	**35** =A1+5
3	**50** =SUM(A1:A2)	**1.166** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

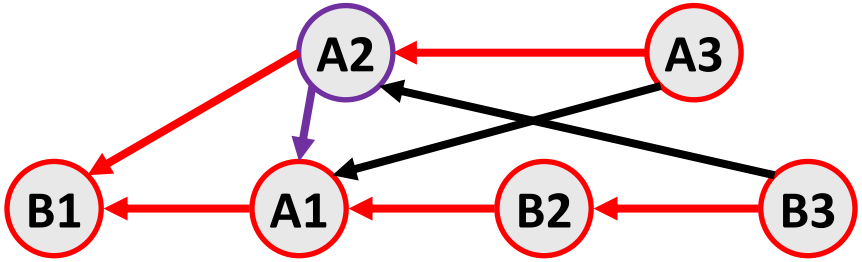
	A	B
1	**30** =B1*2	**15**
2	**45** =A1+B1	**35** =A1+5
3	**75** =SUM(A1:A2)	**1.166** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

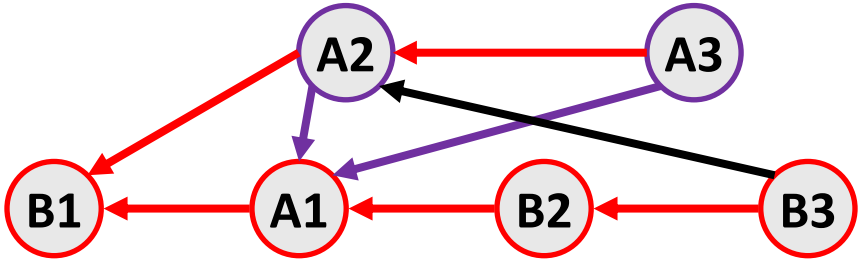
	A	B
1	**30** =B1*2	**15**
2	**45** =A1+B1	**35** =A1+5
3	**75** =SUM(A1:A2)	**1.166** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

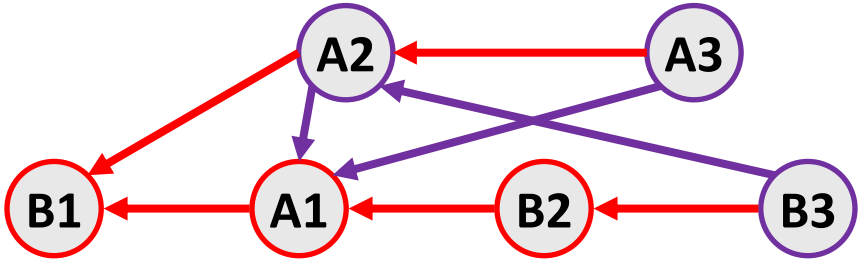
	A	B
1	**30** =B1*2	**15**
2	**45** =A1+B1	**35** =A1+5
3	**75** =SUM(A1:A2)	**1.166** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

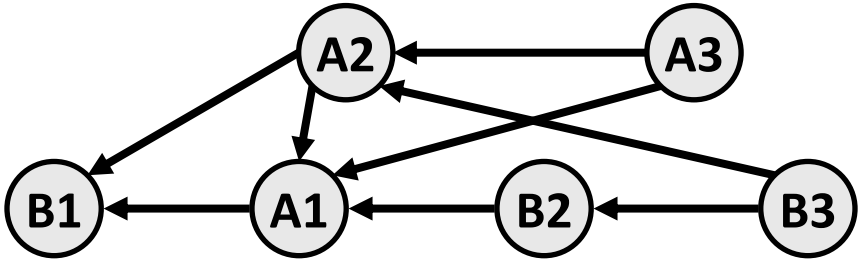
	A	B
1	**30** =B1*2	**15**
2	**45** =A1+B1	**35** =A1+5
3	**75** =SUM(A1:A2)	**0.777** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

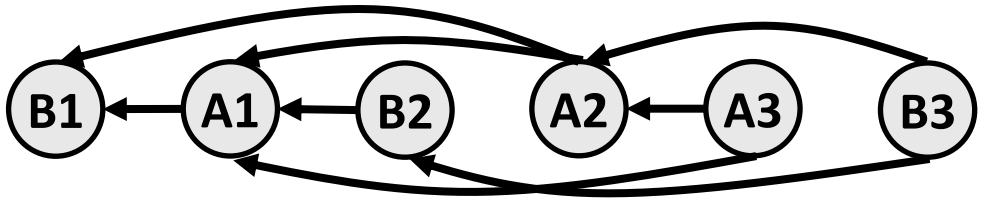
	A	B
1	**20** =B1*2	**10**
2	**30** =A1+B1	**25** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

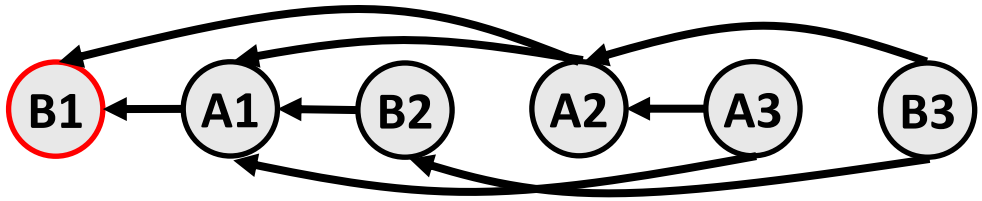
	A	B
1	**20** =B1*2	**10**
2	**30** =A1+B1	**25** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

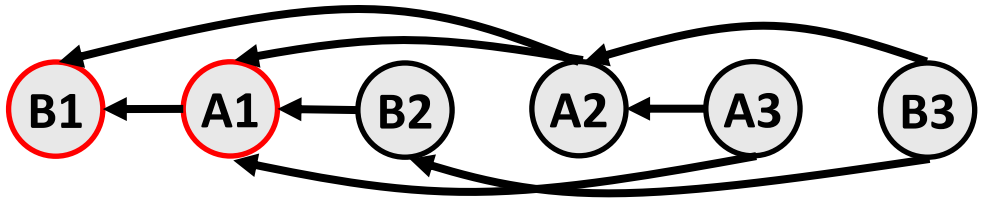
	A	B
1	**20** =B1*2	**15**
2	**30** =A1+B1	**25** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

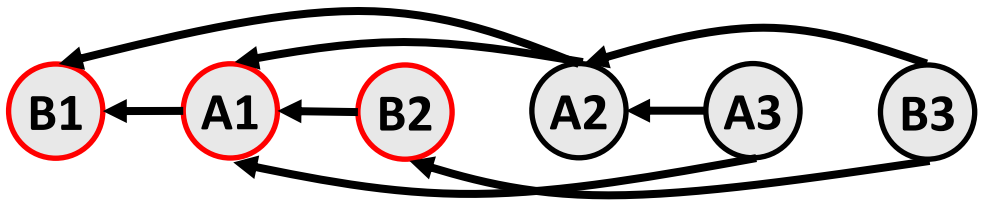
	A	B
1	**30** =B1*2	**15**
2	**30** =A1+B1	**25** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

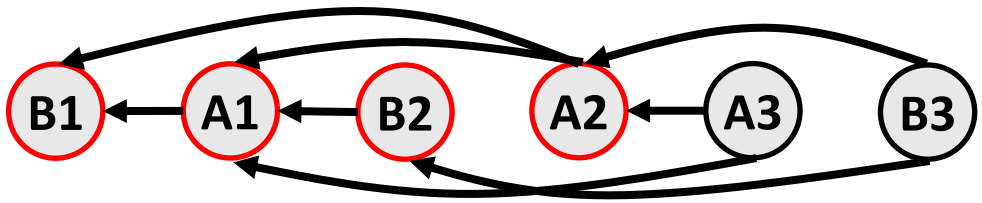
	A	B
1	**30** =B1*2	**15**
2	**30** =A1+B1	**35** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

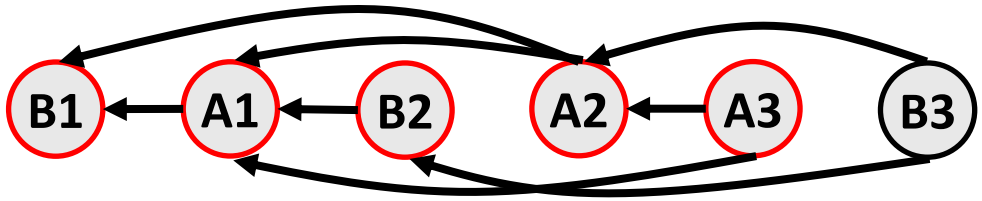
	A	B
1	**30** =B1*2	**15**
2	**45** =A1+B1	**35** =A1+5
3	**50** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

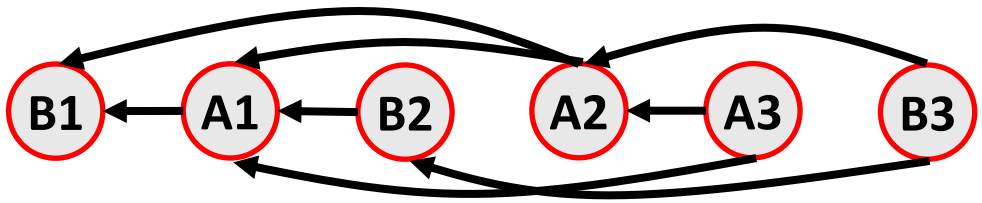
	A	B
1	**30** =B1*2	**15**
2	**45** =A1+B1	**35** =A1+5
3	**75** =SUM(A1:A2)	**0.833** =B2/A2



Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

	A	B
1	**30** =B1*2	**15**
2	**45** =A1+B1	**35** =A1+5
3	**75** =SUM(A1:A2)	**0.777** =B2/A2



Plan for Today

- Topological Sort
  - Kahn's Algorithm
  - Recursive DFS
  - Spreadsheets
- **Announcements**

Announcements

- Homework 8 (Excel) goes out on Wednesday 11/28, is due Friday 12/7. **No late submissions will be accepted**.
- Guest lecture on Hashing next Monday: **Zach!**

Recap

- Topological Sort
  - Kahn's Algorithm
  - Recursive DFS
  - Spreadsheets
- Announcements

Next time: Classes – Inheritance and Polymorphism