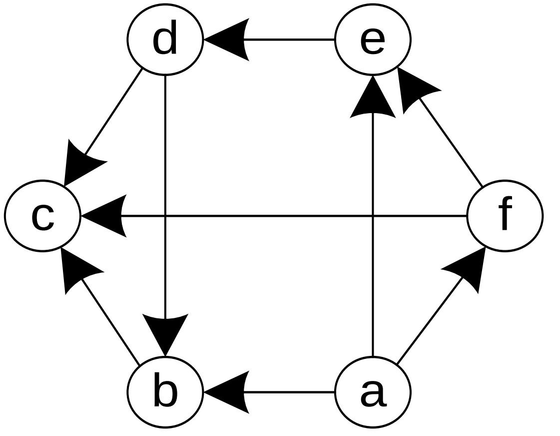
Graph Algorithms: Topological sorting

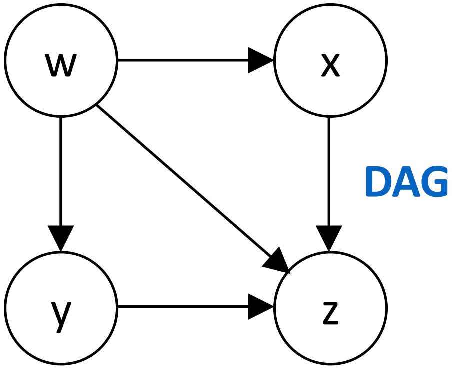
Textbook: Chapter 4.2

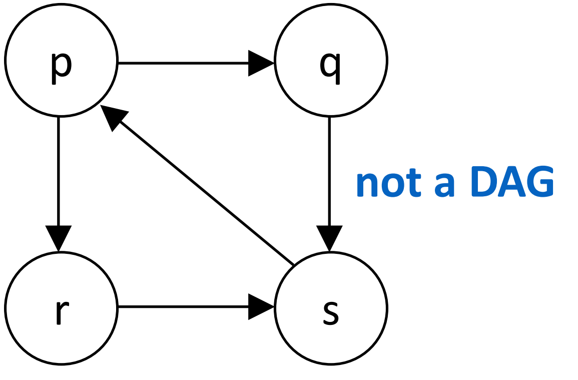
Directed acyclic graphs (DAGs)

- A **directed graph** is a graph whose edges are directional or one-way



- A **directed acyclic graph** is a directed graph that contains no cycles

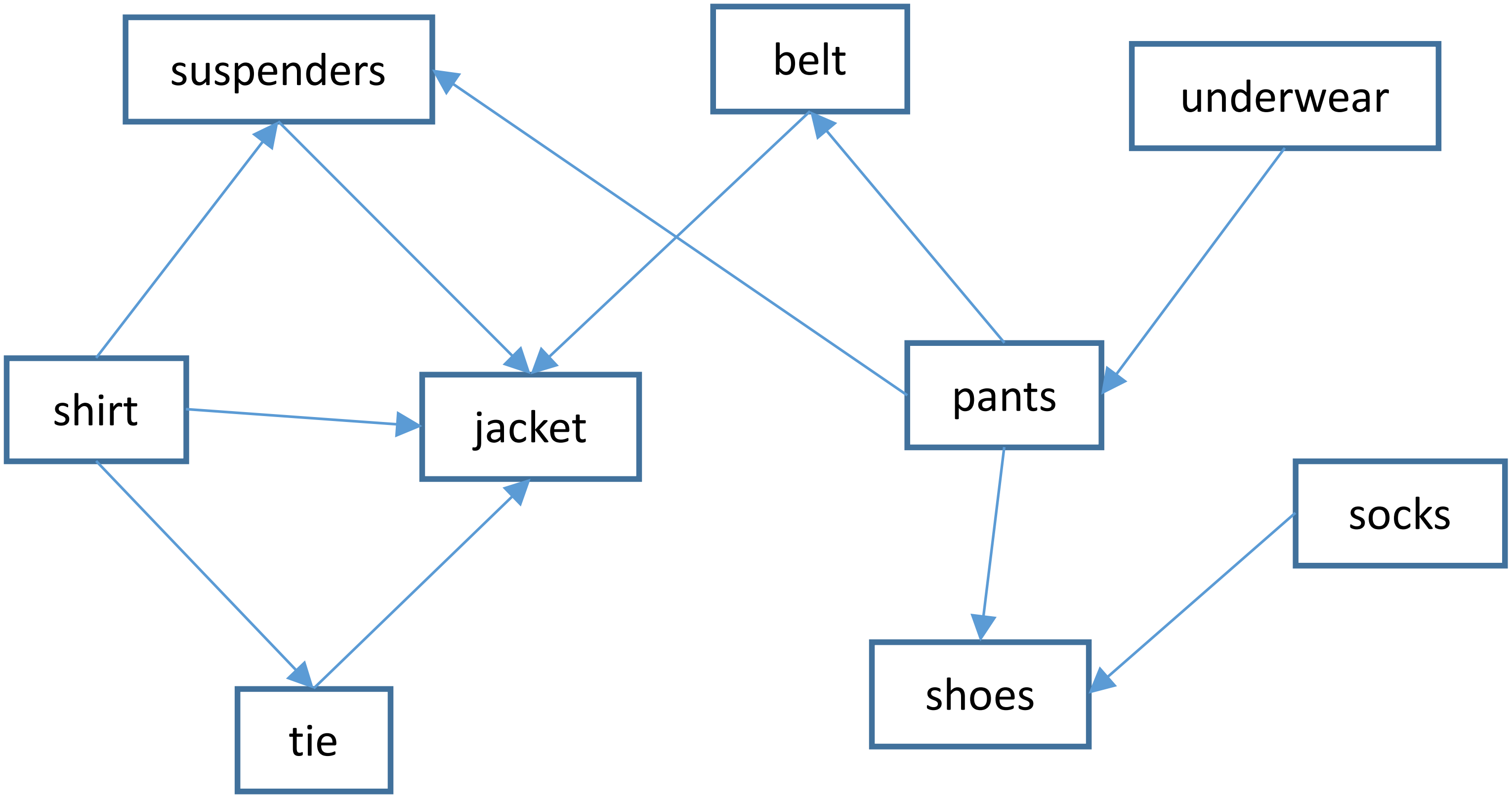




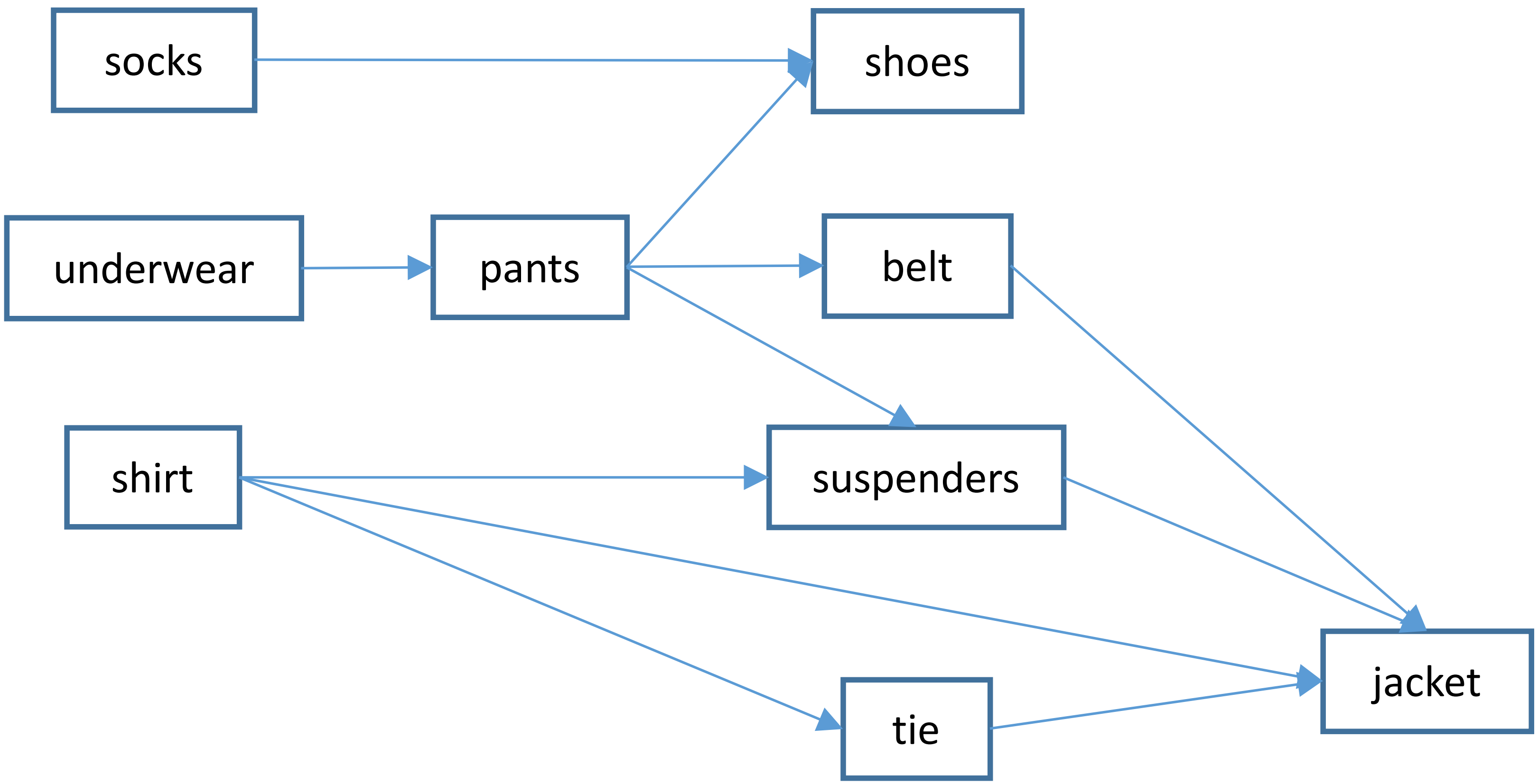
Topological sort problem

- Given a set of tasks with dependencies (precedence constraints), *e.g.*, “task A must be completed before task B”
- Find a linear ordering of the tasks that satisfies all dependencies

Real-life? example



Finding a solution



socks … underwear … shirt … pants … shoes … belt … suspenders … tie … jacket

TopoSort Algorithm 1: Use Depth First Search

1. Represent the problem as a directed graph G:
   a) V = vertices are the items (tasks)
   b) E = edges are the dependencies (constraints) between tasks
      - an edge ($v \rightarrow w$) means that w is dependent on v; i.e. that v must be done before w
2. Apply DFS to G
3. The order in which vertices become dead ends is the *reverse* of the topological sort order
   - Why?

Topo Sort via DFS

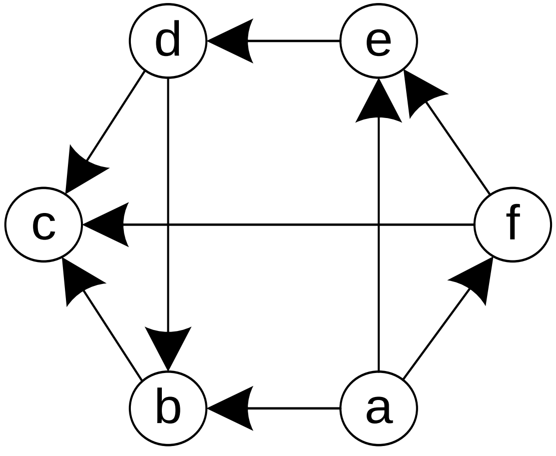
Recall:

- the DFS implementation is recursive
- each time a recursive call is made is equivalent to "pushing a vertex on a stack"
- the “order in which vertices become dead ends” is given by the “order in which vertices are popped off the stack”

Example 1

- Assume you have a set of 6 tasks (a, b, c, d, e, f) with the following dependencies:
  - a must be done before b, e, f
  - b must be done before c
  - d must be done before b and c
  - e must be done before d
  - f must be done before c and e

- <u>Step 1</u>: Construct a directed graph to represent the problem

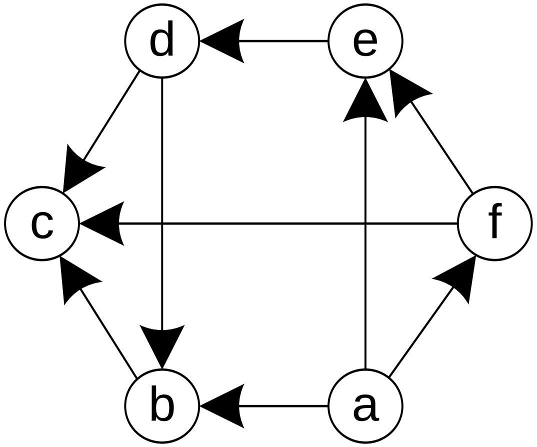


Example 1 (cont)

- Step 2: Apply DFS

  Order vertices become dead ends:

  c b d e f a



- Step 3: Reverse this order for the solution:

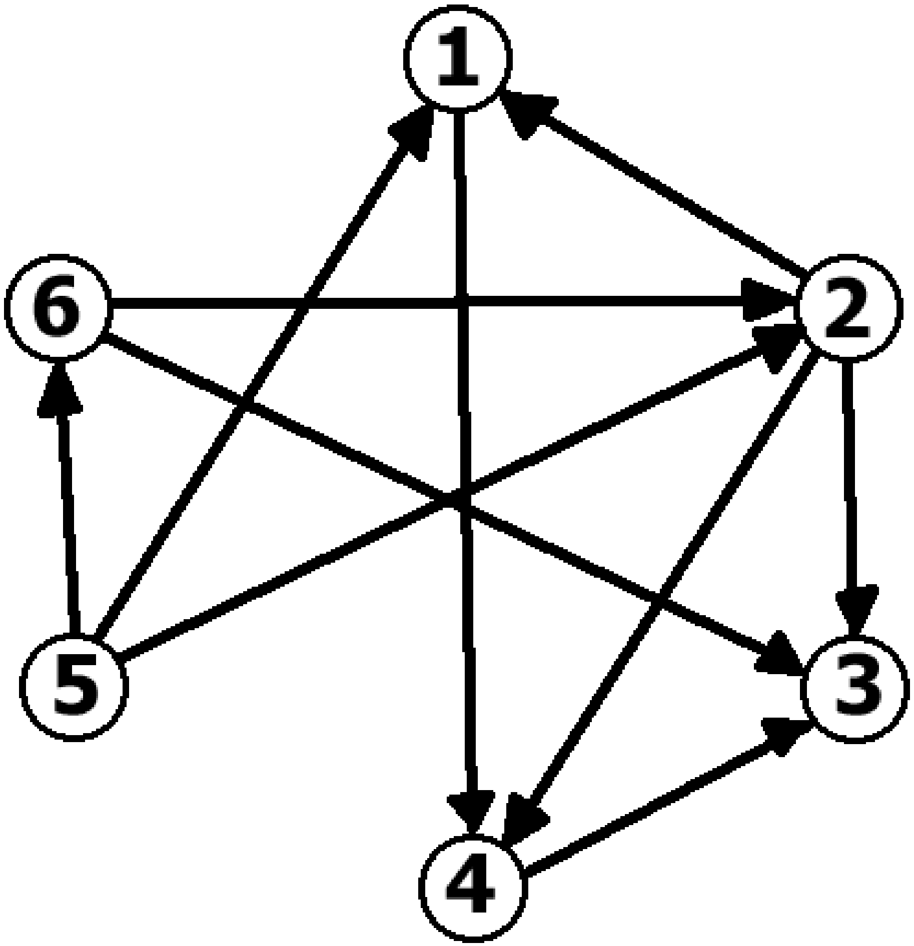
  a f e d b c

Example 2

2 1 (2 before 1)	4 3 (4 before 3)
1 4 (1 before 4)	5 2 (5 before 2)
2 3 (2 before 3)	5 1 (5 before 1)
5 6 (5 before 6)	6 3 (6 before 3)
2 4 (2 before 4)	6 2 (6 before 2)

- Step 1: draw the graph (and verify it is a DAG)
- Step 2: apply DFS
- Step 3: find the order vertices were removed from stack, and reverse this order to get topological sort order

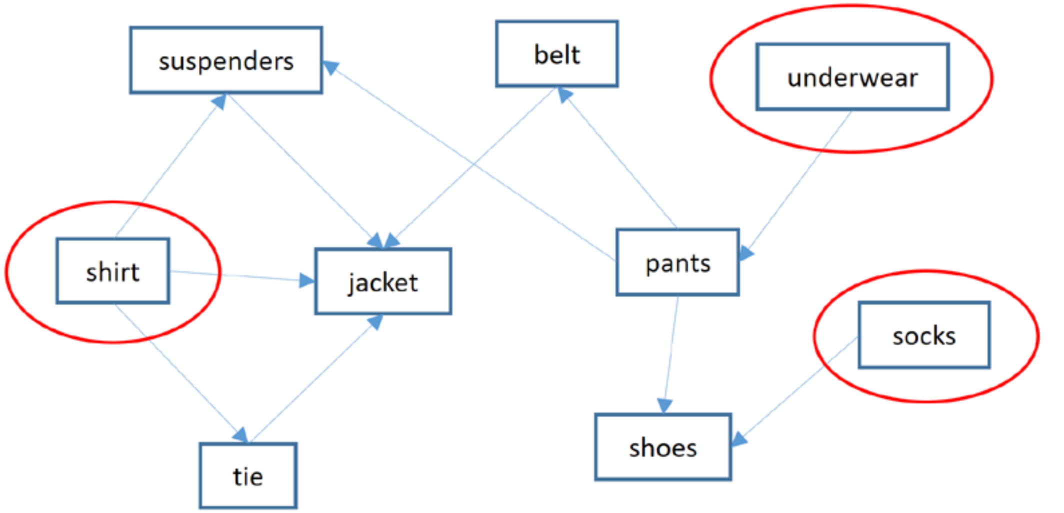
Example 2 (cont)



TopoSort Algorithm 2: Decrease (by 1) and conquer

- Key observation:
  - If a vertex v in the dependency graph G has no incoming arrows (*i.e.* in-degree(v) == 0), then v does not have any dependencies
  - It follows that any v that does not have dependencies is a candidate to be visited next in topological order

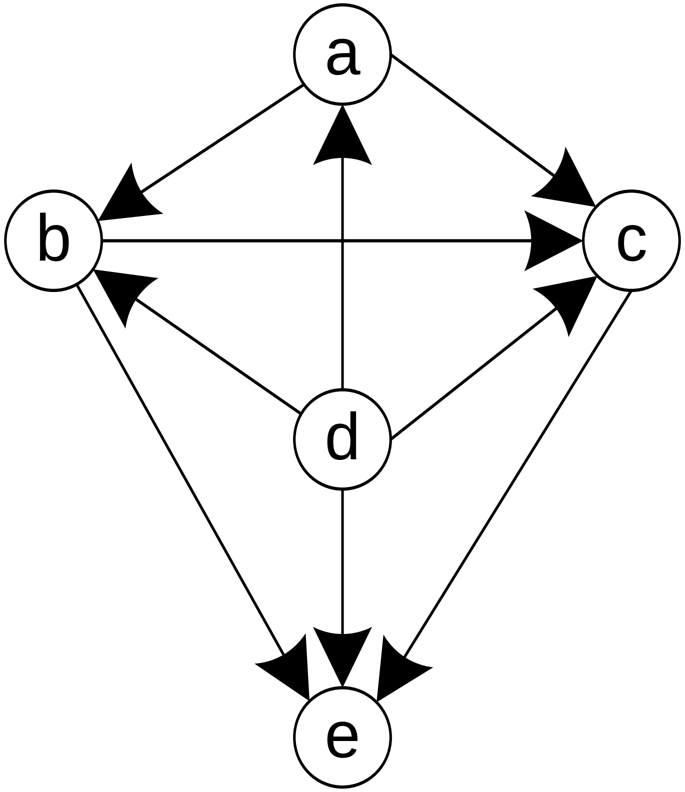
  - *i.e.* any of these can go first →



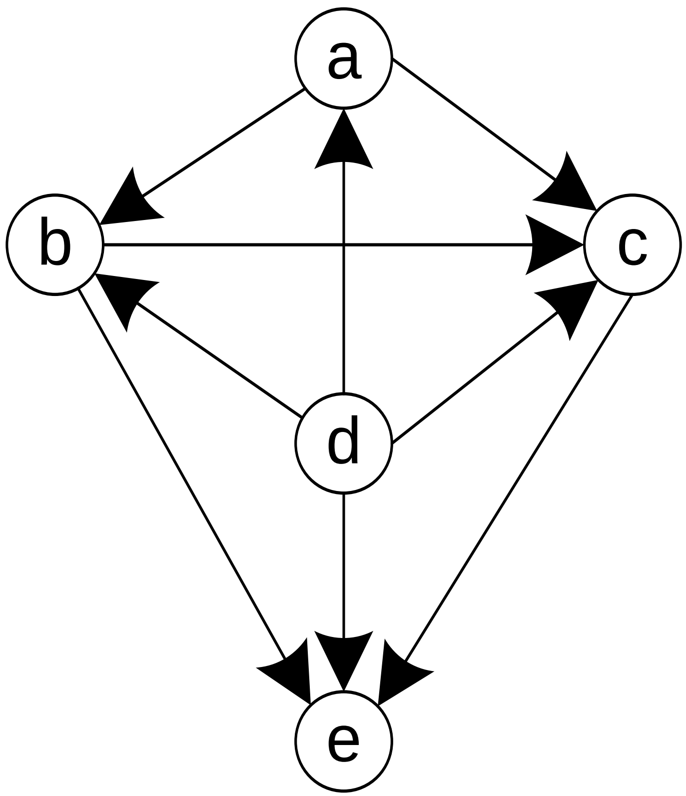
Idea of the algorithm

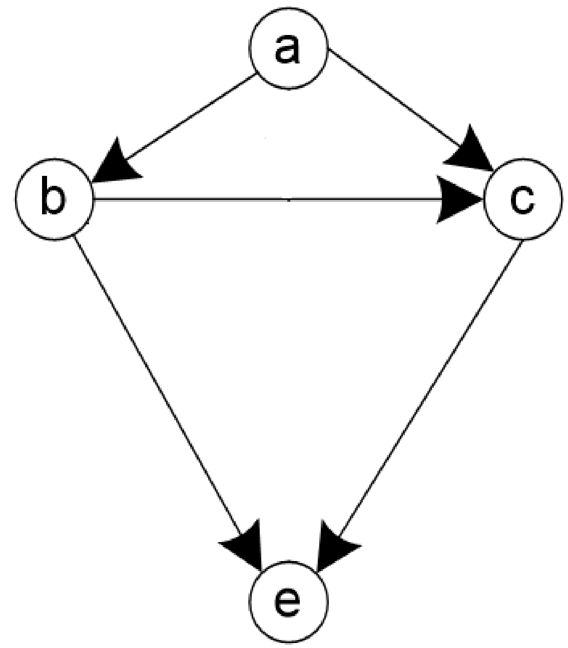
- Identify a $v \in V$ that has in-degree=0
- Delete v and all edges coming out of it
- Repeat until done
- The topological order is the order the vertices are deleted
- If there are $v \in V$, but no v has in-degree=0, the graph G is not a DAG (no feasible solution exists)

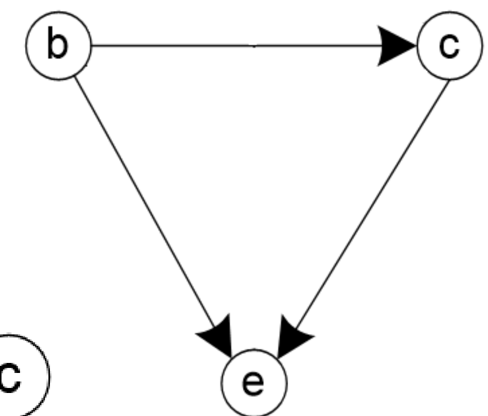
Example

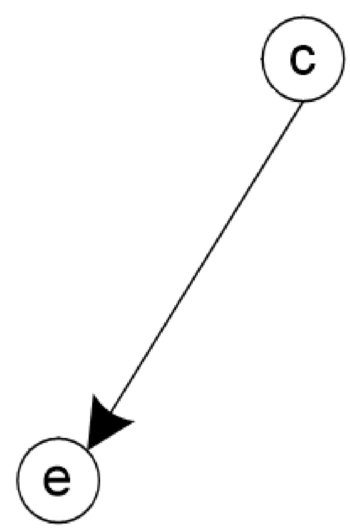


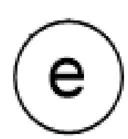
Example











Algorithm details

- Use a set to store the candidate vertices
  - *I.e.* the vertices with in-degree = 0
  - Any ordered set will do, e.g. TreeSet.
- Use an ordered list to store the delete order
  - Any ordered list will do, e.g. ArrayList

TopoSort "Decrease by one" pseudocode

```
Algorithm TopoSort(G)
   create an empty ArrayList A
   create an empty TreeSet Candidates
   add all v with inDegree=0 to Candidates
   while Candidates is not empty
      v = Candidates.first()
      add v to A
      for each vertex w adjacent to v
         remove edge (v,w) from G
         if w has inDegree=0
            add w to Candidates
      remove vertex v from G
   if there are no vertices remaining in G
      solution is in A
   else
      no solution exists
```

Practice problems

- Chapter 4.2, page 142, question 1