Directed Acyclic Graphs
&&
Topological Sorting

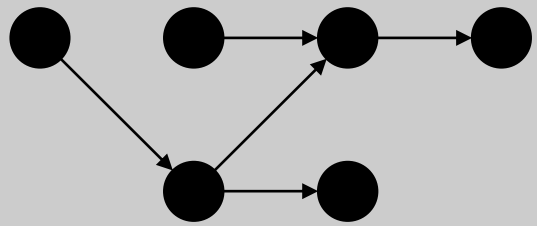
by Tian Cilliers

IOI Training Camp 2 (3-4 February 2018)

Definitions

Directed Acyclic Graph (DAG)

A graph such that all of its edges are directed and there exist no cycles

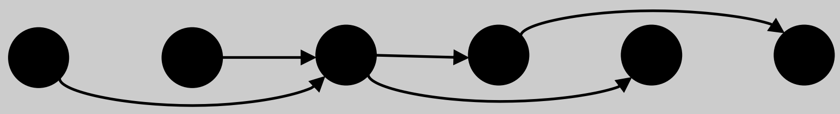


- A DAG can be used to represent any transitive operation (that is, any operation $\circ$ where if $a \circ b$ and $b \circ c$, then $a \circ c$)
- Used in scheduling, version control, compilation dependencies, cryptocurrencies etc.

Definitions

Topological Sort

A sorting of the vertices of a DAG such that for directed edge `uv` from vertex `u` to vertex `v`, `u` comes before `v` in the ordering

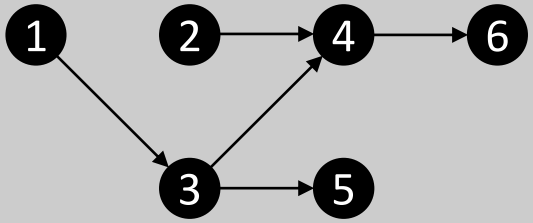


- A graph is a DAG if and only if it is directed and has a topological sort (no cycles)
- There may be multiple existing topological orderings for any DAG
- A topological sorting can be easily reversed by reversing each edge

Algorithms

Iterative Solution (Kahn's Algorithm)

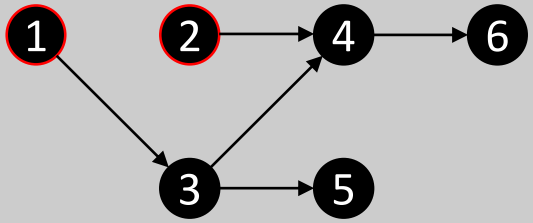
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

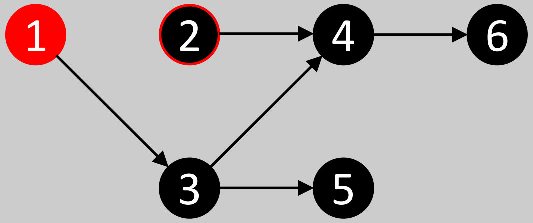
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

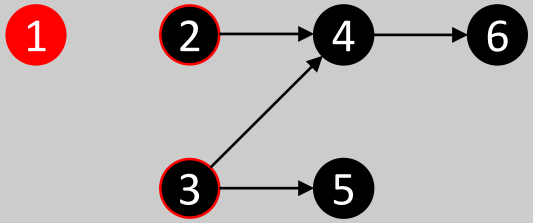
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

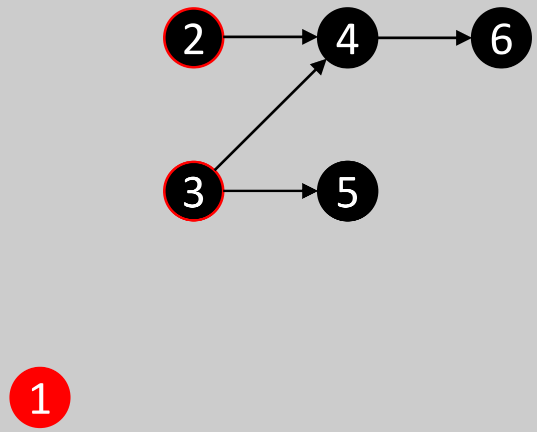
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

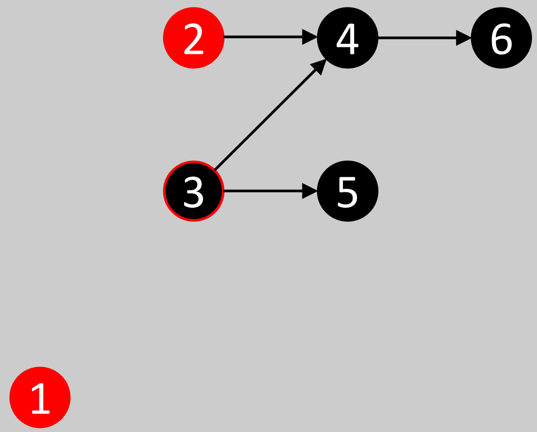
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

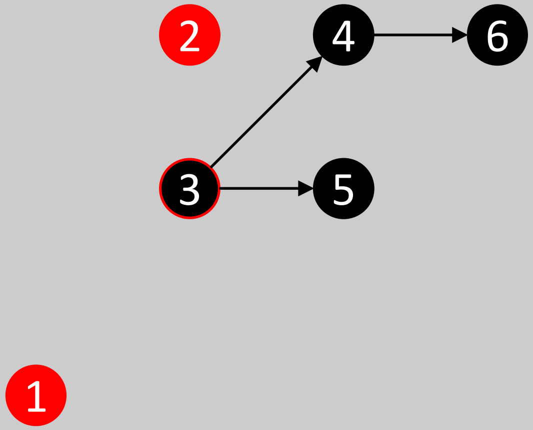
Demonstration:



Algorithms

Iterative Solution (Kahn’s Algorithm)

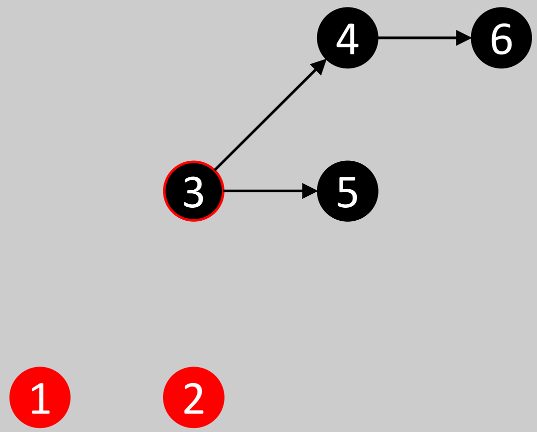
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

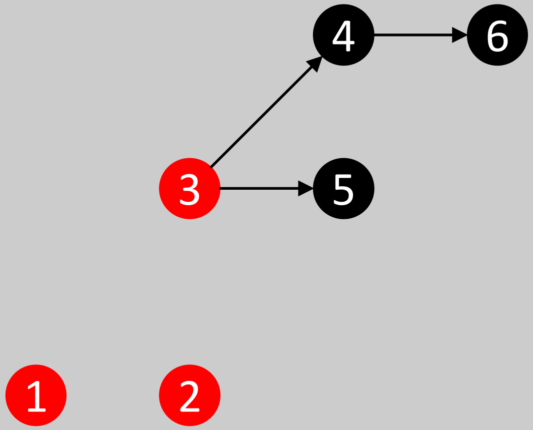
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

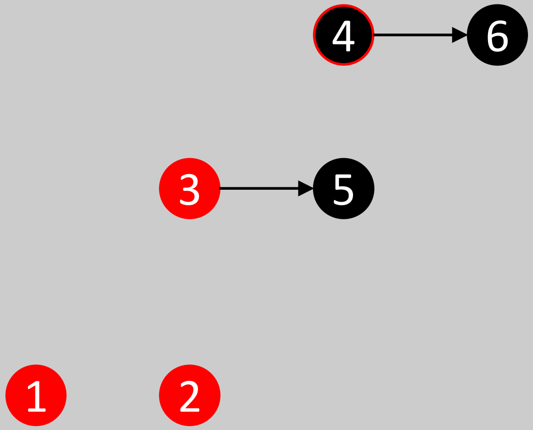
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

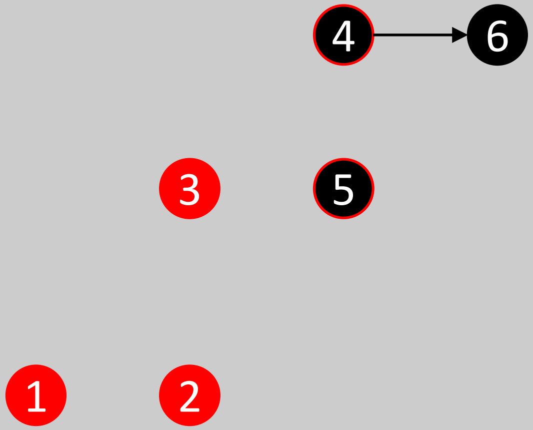
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

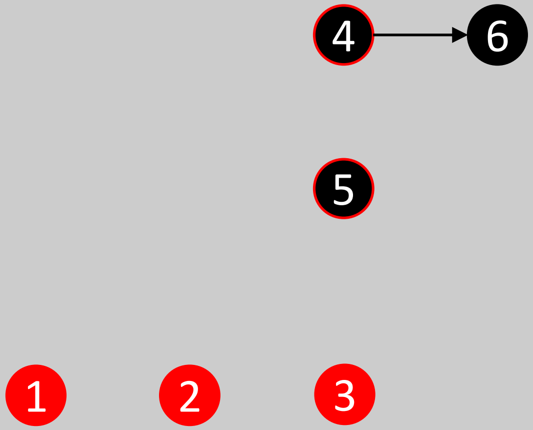
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

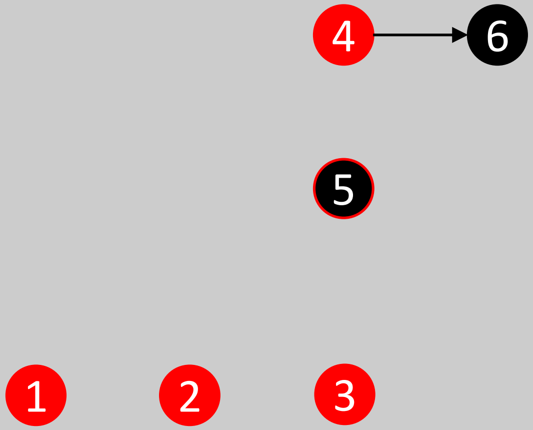
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

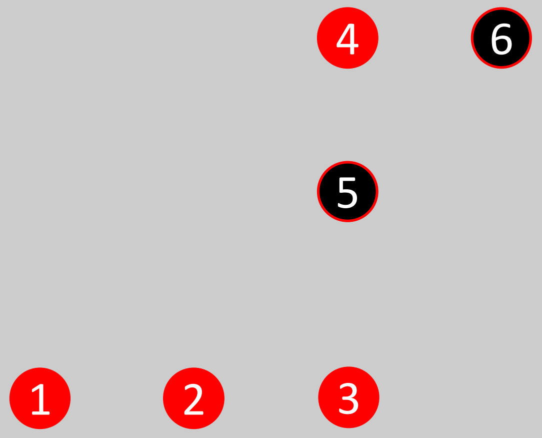
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

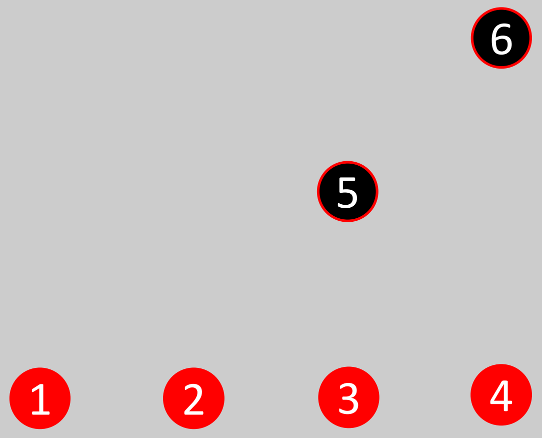
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

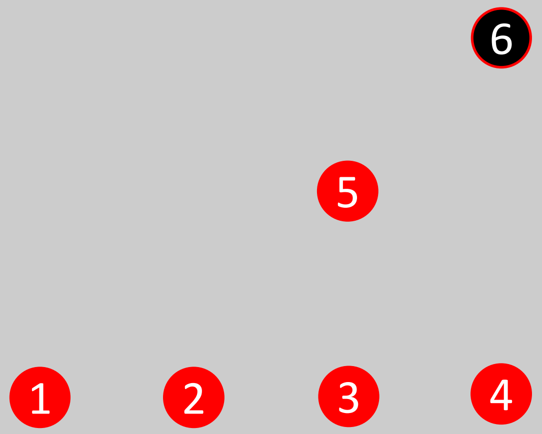
Demonstration:



Algorithms

Iterative Solution (Kahn’s Algorithm)

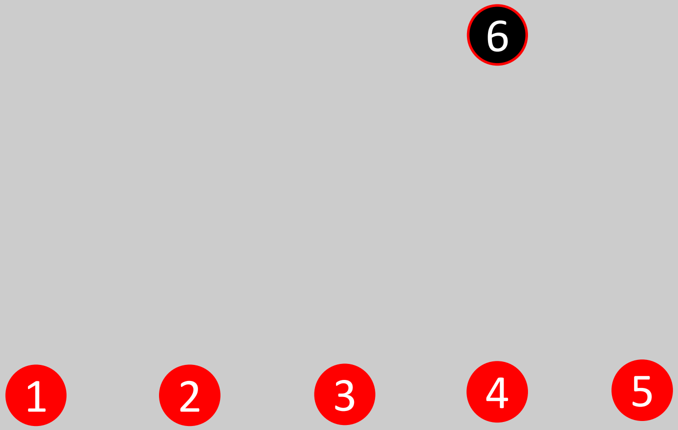
Demonstration:



Algorithms

Iterative Solution (Kahn’s Algorithm)

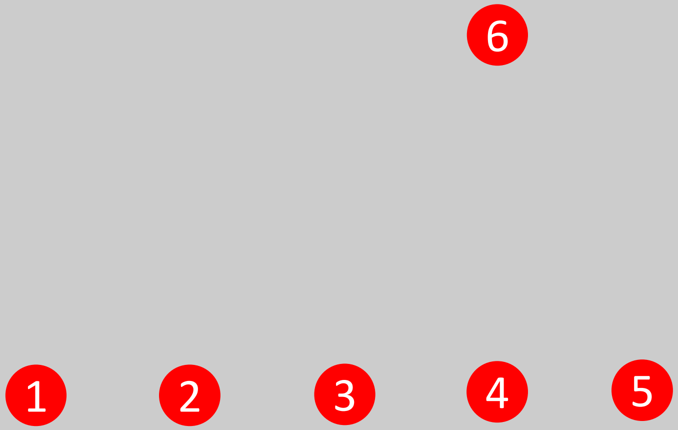
Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

Demonstration:



Algorithms

Iterative Solution (Kahn's Algorithm)

Demonstration:

Algorithms

Iterative Solution (Kahn's Algorithm)

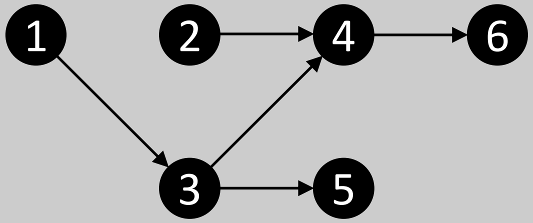
~~Python~~ Pseudocode:

```
def topsort_kahn(G, n):                                # G is a 2d array containing children of each node, n is the amount of nodes
    S  = []                                            # will contain topological sort
    indeg = [0 for i in range(n)]                      # will contain in-degree of each node
    for node in range(n):                              # update in-degree of each node
        for descendant in G[node]:
            indeg[descendant] += 1                     # by increasing in-degree of each descendant of each connection
    q     = [i for i in range(n) if indeg[i]==0]       # list of nodes with no incoming connections

    while len(q)>0:                                    # while there exist unprocessed nodes
        v = q.pop(0)                                   # pop a node that has no unprocessed parents
        for descendant in G[v]:                        # loop through all descendants of node
            indeg[descendant] -= 1                     # decrement in-degree (effectively removing connection)
            if indeg[descendant]==0: q.append(descendant) # add to q if all incoming connections has been processed
        S.append(v)                                    # add processed node to topological sort

    return S                                           # you really want me to explain this?
```

Algorithms

DFS Solution

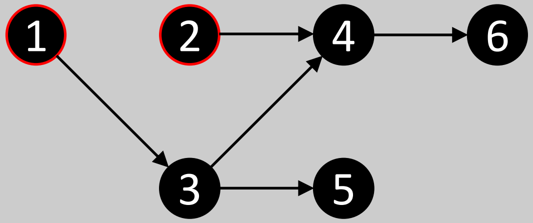
Demonstration:



Algorithms

DFS Solution

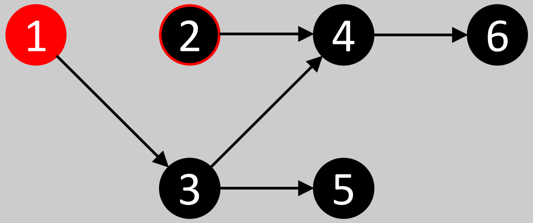
Demonstration:



Algorithms

DFS Solution

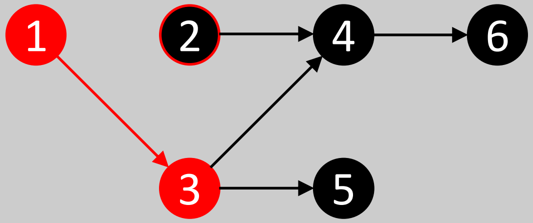
Demonstration:



Algorithms

DFS Solution

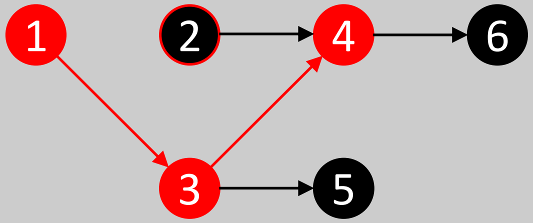
Demonstration:



Algorithms

DFS Solution

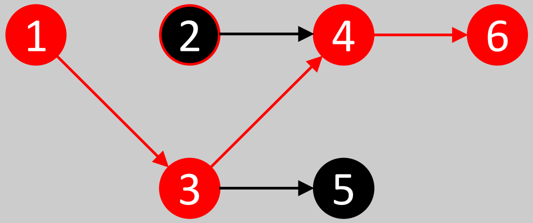
Demonstration:



Algorithms

DFS Solution

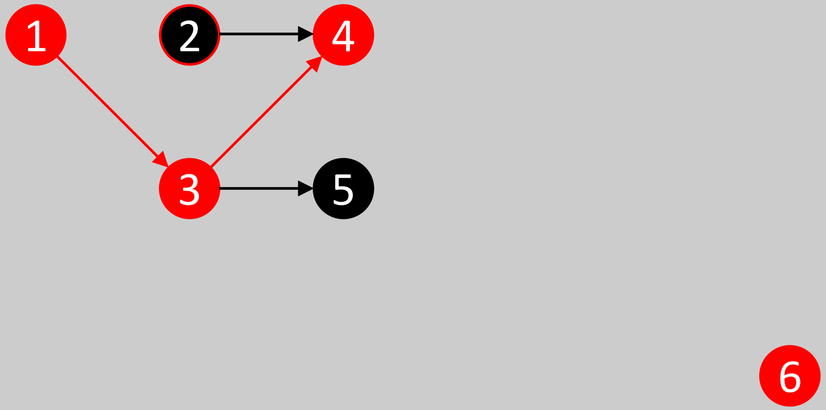
Demonstration:



Algorithms

DFS Solution

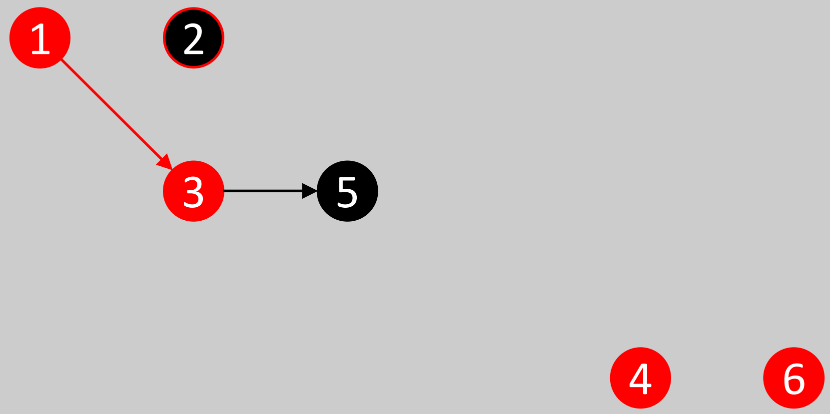
Demonstration:



Algorithms

DFS Solution

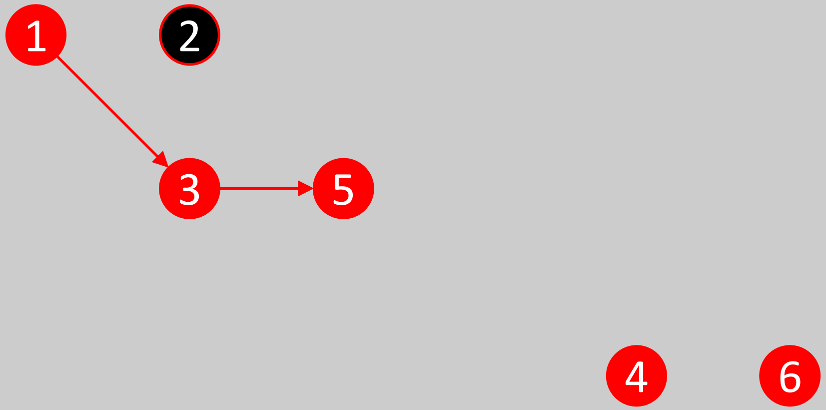
Demonstration:



Algorithms

DFS Solution

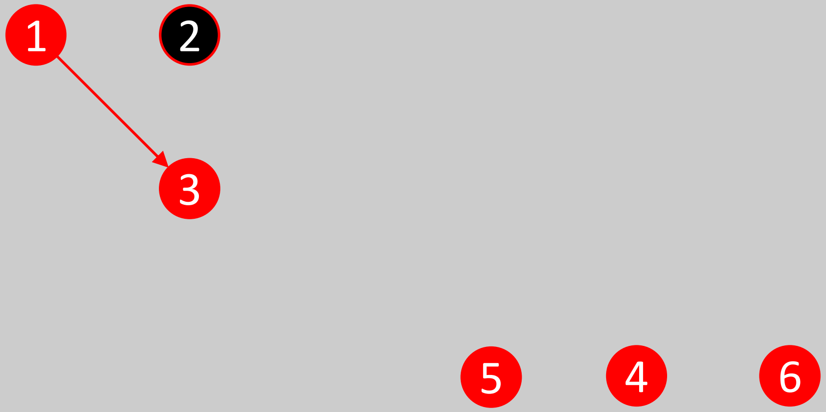
Demonstration:



Algorithms

DFS Solution

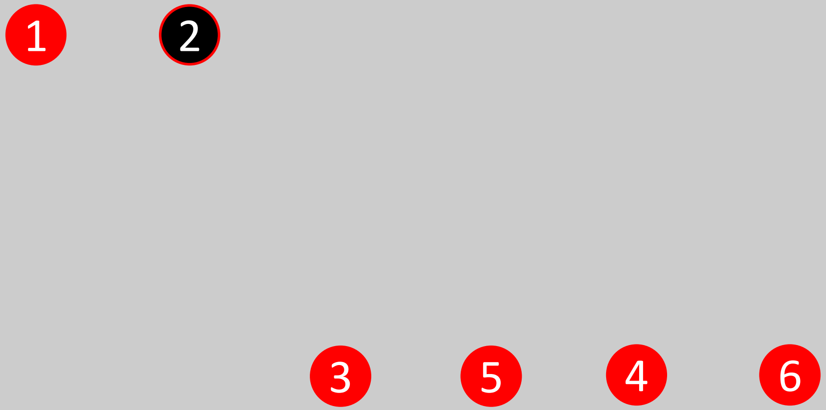
Demonstration:



Algorithms

DFS Solution

Demonstration:



Algorithms

DFS Solution

Demonstration:

Algorithms

DFS Solution

Demonstration:

Algorithms

DFS Solution

Demonstration:

Algorithms

DFS Solution

Pseudocode:

```
def topsort_dfs(G, n):                          # G is a 2d array containing children of each node, n is the amount of nodes
    S  = []                                     # will contain topological sort
    indeg = [0 for i in range(n)]               # will contain in-degree of each node
    for node in range(n):                       # update in-degree of each node
        for descendant in G[node]:
            indeg[descendant] += 1              # by increasing in-degree of each descendant of each connection
    q     = [i for i in range(n) if indeg[i]==0]  # list of nodes with no incoming connections
    V     = [False for i in range(n)]           # contains whether node has been visited by dfs

    for node in q:                              # run dfs on all nodes with no incoming connections
        dfs(G, V, S, node)

    return S                                    # ...

def dfs(G, V, S, current):                      # G, V, S is defined above, node is the current node to be processed
    for descendant in G[current]:               # loop through all descendants of node
        if not V[descendant]:                   # if node has not been visited run dfs on node
            dfs(G, V, S, descendant)
                                                # all children nodes have been visited and to sort
    V[current] = True                           # set node to visited
    S.insert(0, current)                        # insert node at the top of topological sort
```

Algorithms

Performance

Both algorithms run in O(V+E) time, looping over every edge and every node exactly once

Both algorithms require O(V+E) space, only differing by a constant

DFS Solution might be more straightforward to implement, but requires a array keeping track of visited nodes as well as nodes in the current path, while Iterative Solution only needs to update in-degree of each node

Example

Modified Alphabet (Codeforces Round 290 Div.1 Problem A)

A list of names are written in lexicographical order, but not in a normal sense. Some modification to the order of the letters in the alphabet is needed so that the order of the names becomes lexicographical. Given a list of names, does there exist an order of letters in the Latin alphabet so that the names are following in lexicographical order? If so, you should find any such order.

Sample IO:

Input	**Output**

`3`
`rivest`
`shamir`
`adleman`
	`bcdefghijklmnopqrsatuvwxyz`

Example

<u>Modified Alphabet (Codeforces Round 290 Div.1 Problem A)</u>

<u>Solution:</u>

Between every consecutive pair of words, draw an edge between the first two different letters indicating that for the first word to be before the second one, the selected letter in the first word should come before the selected letter in the second word. A topological sorting of this graph gives the answer.

Example

Substring (Codeforces Round 460 Div.2 Problem D)

You are given a graph with n nodes and m directed edges. One lowercase letter is assigned to each node. We define a path's value as the number of the most frequently occurring letter. For example, if letters on a path are "abaca", then the value of that path is 3. Your task is find a path whose value is the largest.

Sample IO:

Input	Output
5 4 abaca 1 2 1 3 3 4 4 5	3

Example

Substring (Codeforces Round 460 Div.2 Problem D)

Solution:

Use a DP approach, storing the amount of letters `j` you can get up to some node `i` in `f[i][j]`. Run a topological sorting algorithm (processing dependencies first) but for every node `i`, instead of adding any descendant `k` to a list containing the sort, update `f[k][*]` using `f[i][*]`

Questions?