Graph

- Graph
  - Introduction
  - Adjacency list
  - DFS, BFS
  - Challenges
- **Digraph – Directed Graphs**
  - digraph – DFS, BFS
  - **Applications – crawl web, topological sort**
- Minimum Spanning Tree(MST)

Major references:

1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

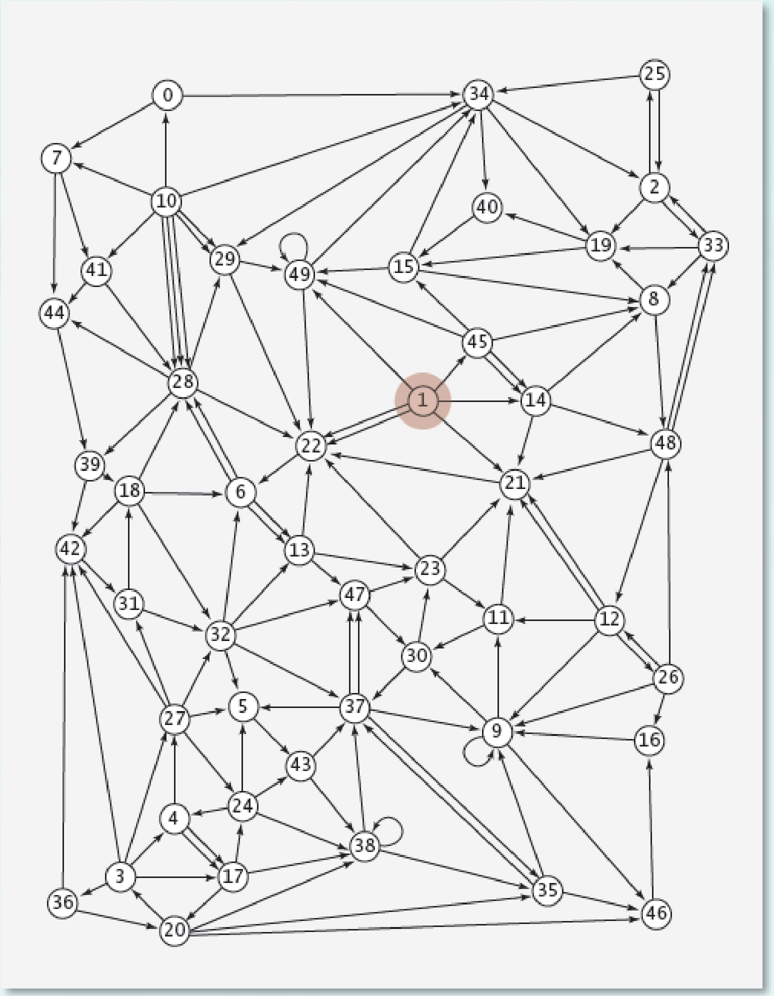
Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

Breadth-first search in digraphs application

Goal: Crawl web, starting from some root web page, or www.handong.edu

Solution: [BFS with implicit digraph]

- Choose root web page as source s.
- Maintain a queue of websites to explore.
- Maintain a SET of discovered websites.
- Dequeue the next website and enqueue websites to which it links
  (provided you haven't done so before).



Q: Why not use DFS?

Bare-bone web crawler: Java implementation

```
Queue<String> queue = new Queue<String>();
SET<String> marked = new SET<String>();

String root = "http://www.princeton.edu";
queue.enqueue(root);
marked.add(root);

while (!queue.isEmpty())
{
   String v = queue.dequeue();
   StdOut.println(v);
   In in = new In(v);
   String input = in.readAll();

   String regexp = "http://(\\w+\\.)*(\\w+)";
   Pattern pattern = Pattern.compile(regexp);
   Matcher matcher = pattern.matcher(input);
   while (matcher.find())
   {
      String w = matcher.group();
      if (!marked.contains(w))
      {
         marked.add(w);
         queue.enqueue(w);
      }
   }
}
```

queue of websites to crawl
set of marked websites

start crawling from root website

read in raw html from text
web site in queue

use regular expression to find all
URLs in website of form
http://xxx.yyy.zzz
[crude pattern misses relative]

if unmarked, mark it and put
on the queue.

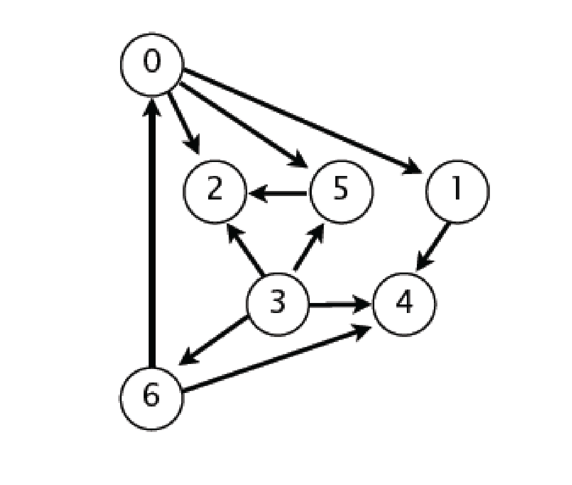
Precedence scheduling

Goal: Given **a set of tasks** to be completed with **precedence constraints**, in which order should we schedule the tasks?

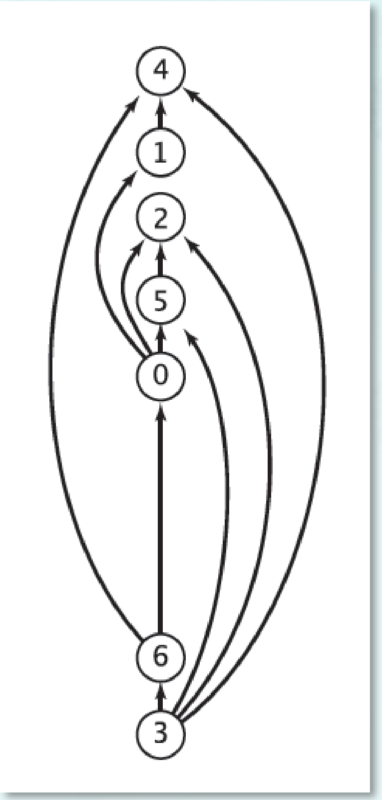
Digraph model: vertex = task; edge = precedence constraint.

0. Algorithms
1. Complexity Theory
2. Artificial Intelligence
3. Intro to CS
4. Cryptography
5. Scientific Computing
6. Advanced Programming

tasks



precedence constraint graph



feasible schedule

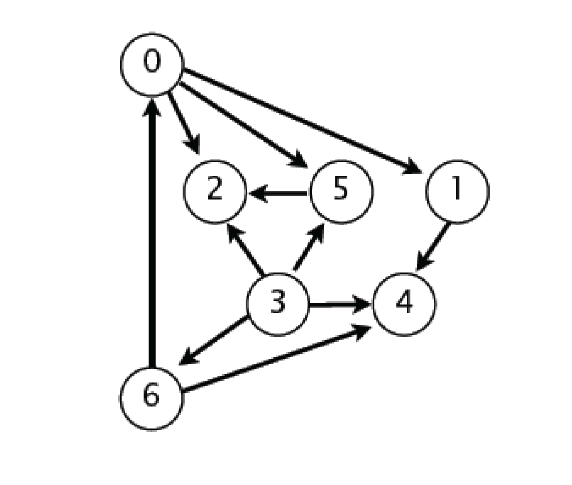
Precedence scheduling

DAG: Directed **acyclic** graph

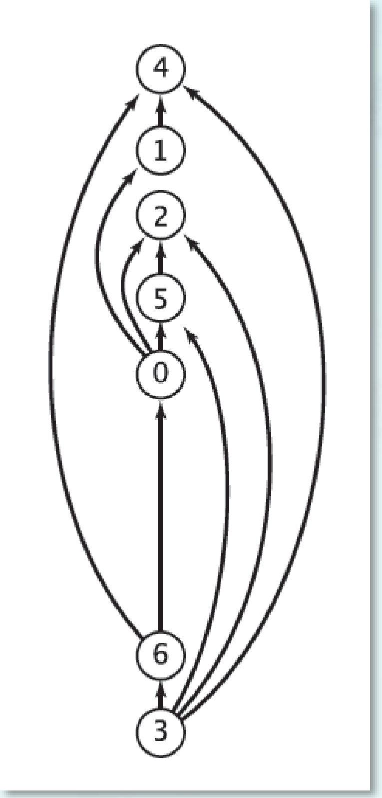
Topological sort: Redraw DAG so all edges point upwards.

0→5	0→2
0→1	3→6
3→5	3→4
5→2	6→4
6→0	3→2
1→4	

directed edges



DAG



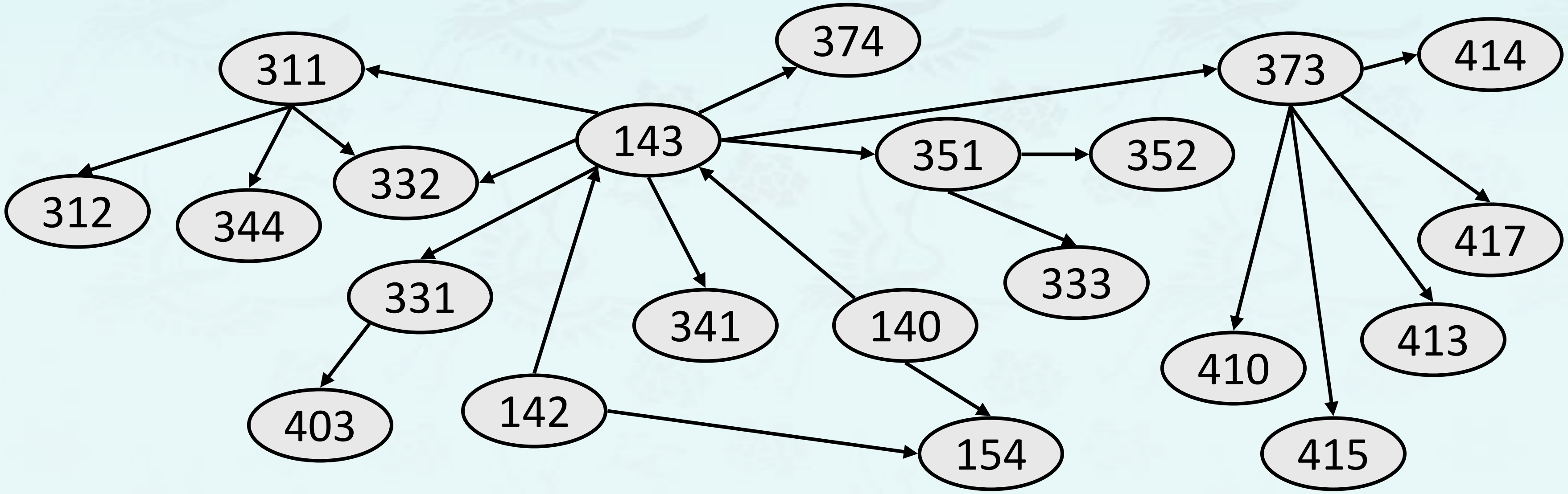
topological order

Solution I : DFS. What else?

Precedence scheduling

Example: Suppose we have a directed acyclic graph (DAG) of courses, and we want to find an order in which the courses can be taken.

- Must take all prereqs before you can take a given course.
- Example: [142, 143, 140, 154, 341, 374, 331, 403, 311, 332, 344, 312, 351, 333, 352, 373, 414, 410, 417, 413, 415]
  There might be more than one allowable ordering.
- How can we find a valid ordering of the vertices?



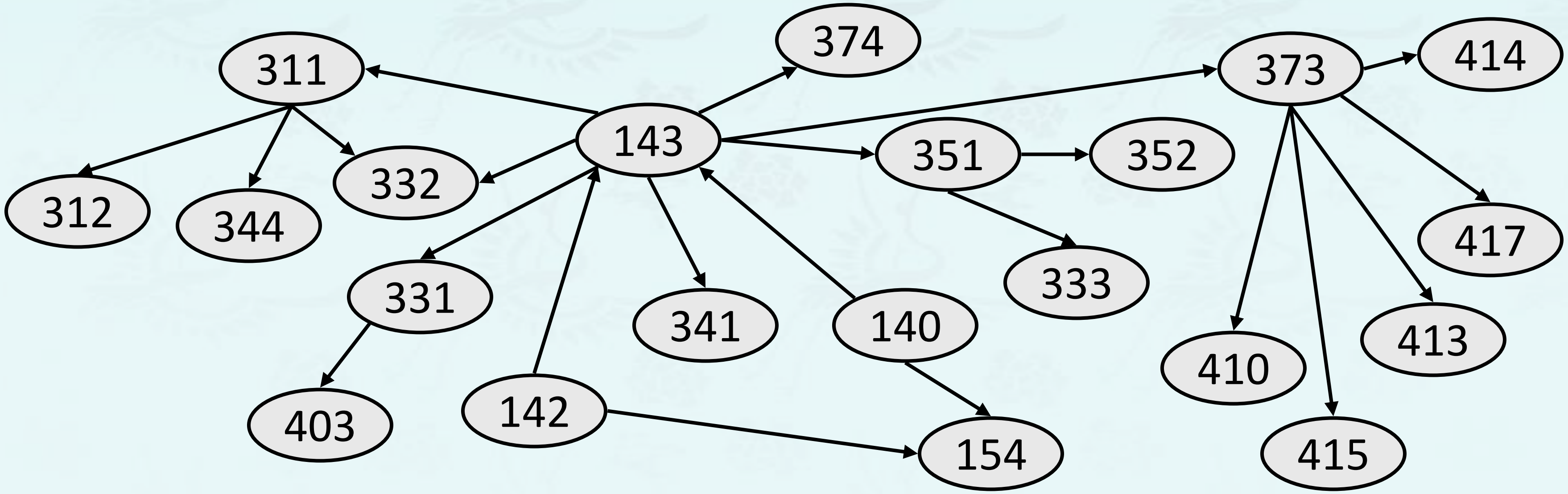
Precedence scheduling – Topological Sort

Topological Sort: Given a digraph *G* = (*V*, *E*), a total ordering of *G*'s vertices such that for every edge (*v*, *w*) in *E*, **vertex *v* precedes *w*** in the ordering.

Examples:

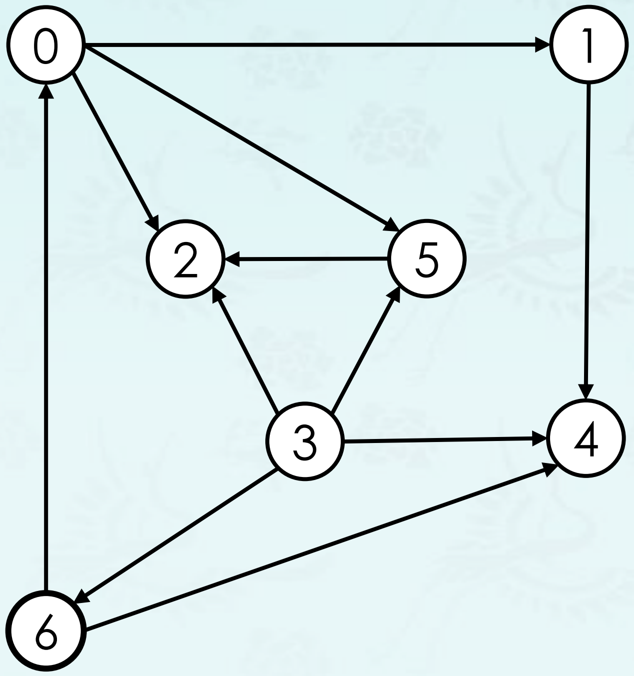
- determining the order to recalculate updated cells in a spreadsheet
- finding an order to recompile files that have dependencies

  (any problem of finding an order to perform tasks with dependencies)



Topological sort demo

- Run depth-first search.
- Return vertices in reverse **postorder**.



a directed acyclic graph

0->5
0->2
0->1
3->6
3->5
3->4
5->2
6->4
6->0
3->2
1->4

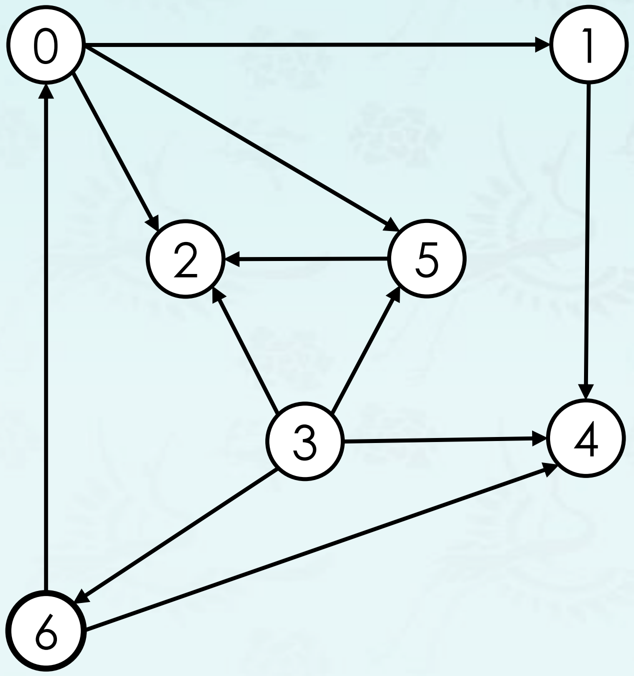
Topological sort demo

- Run depth-first search.
- Return vertices in reverse **postorder**.

List the vertices in the order in which they are **last visited** by DFS traversal.

Preorder

List the vertices in the order in which they are **first visited** by DFS traversal.

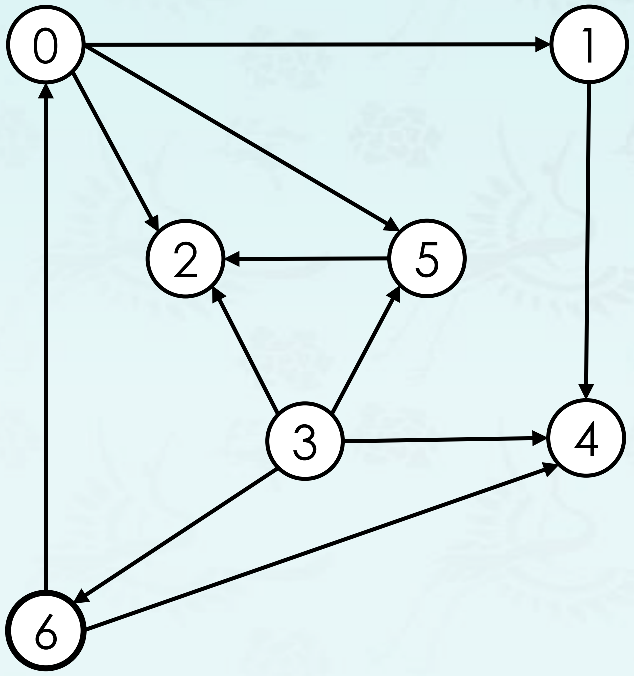


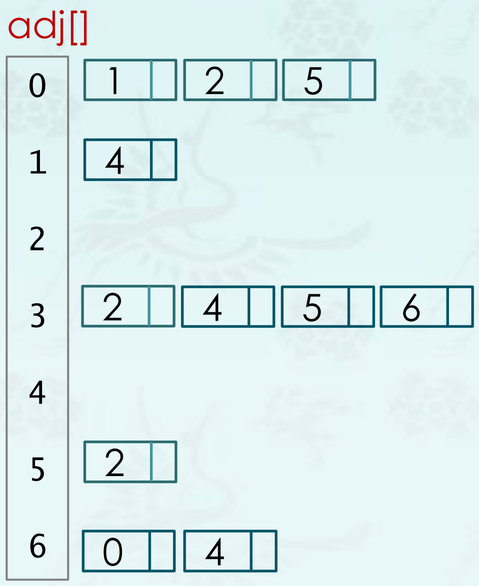
0->5
0->2
0->1
3->6
3->5
3->4
5->2
6->4
6->0
3->2
1->4

a directed acyclic graph

Topological sort demo

- **Run depth-first search.**
- **Return vertices in reverse postorder.**



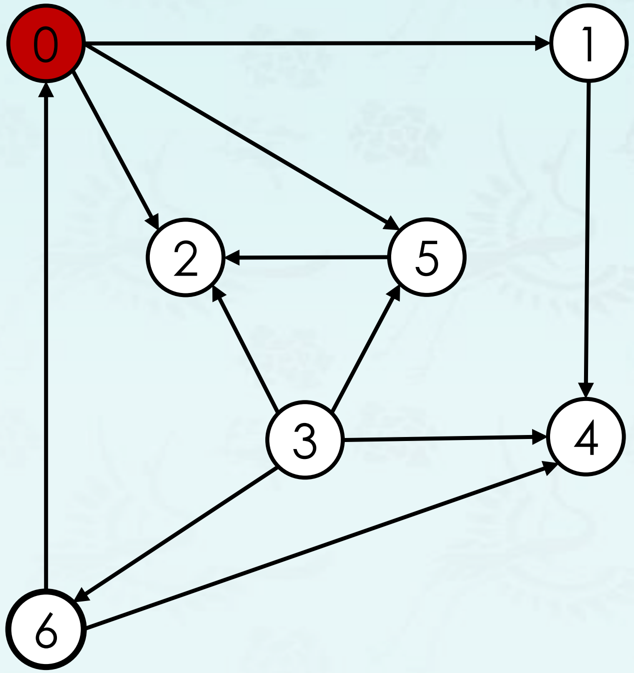


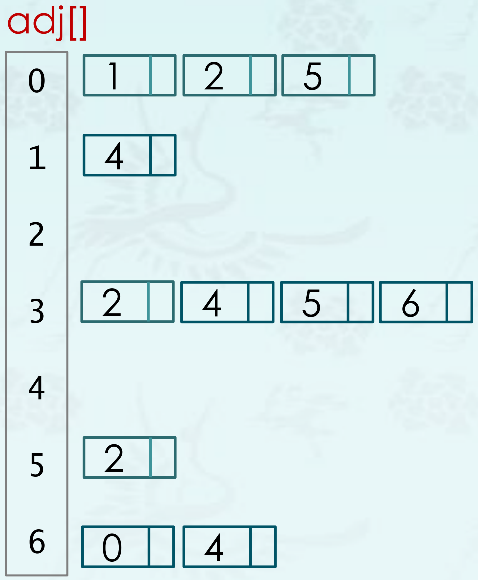
0->5
0->2
0->1
3->6
3->5
3->4
5->2
6->4
6->0
3->2
1->4

a directed acyclic graph

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



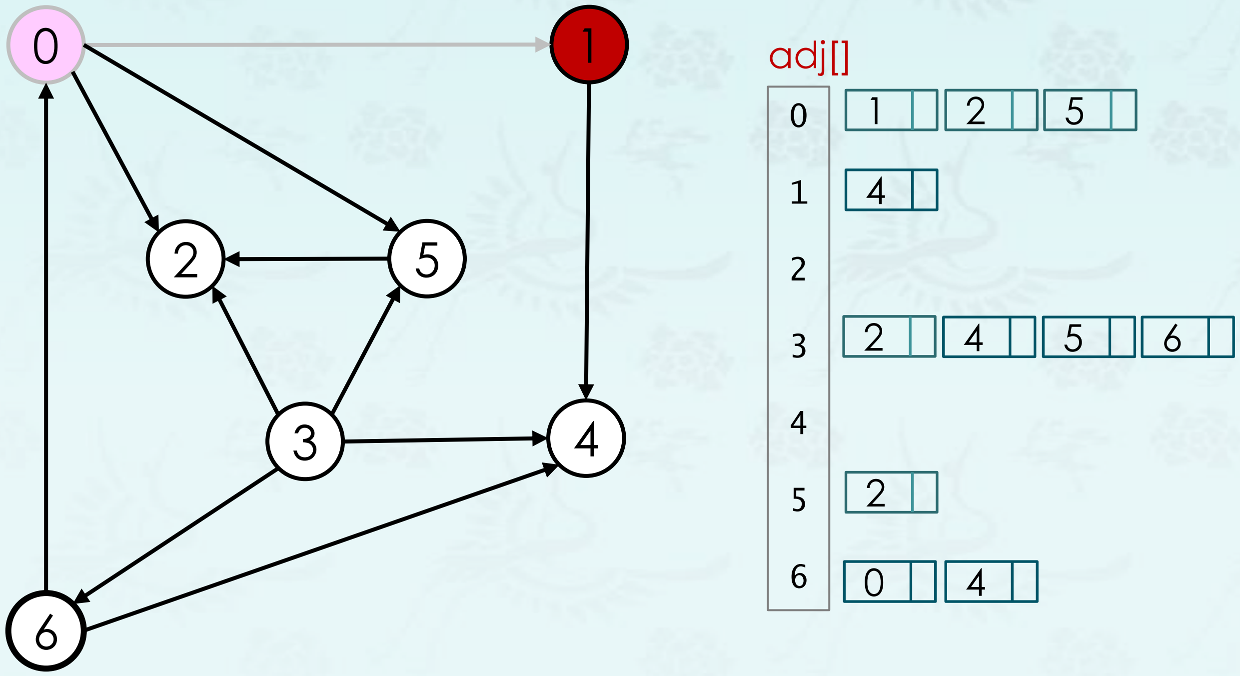


postorder

visit 0: check 1, check 2, and check 5

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

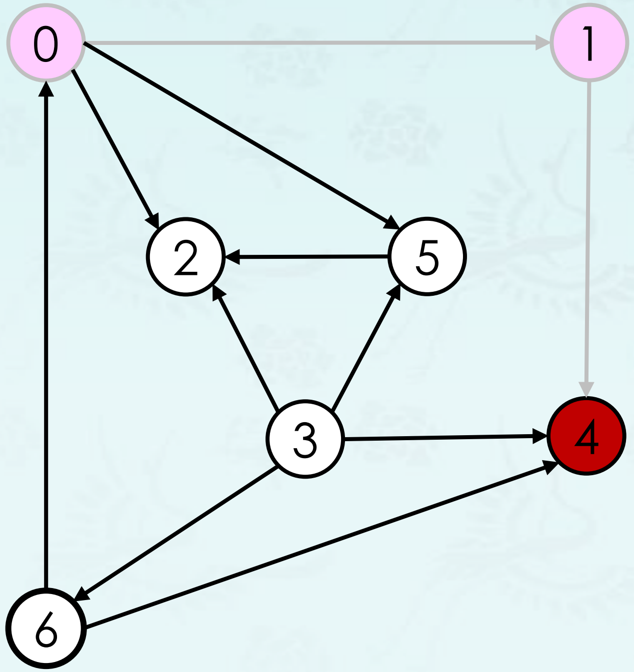


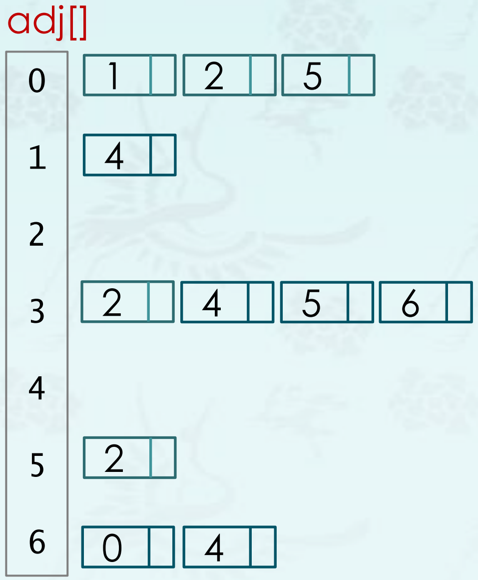
postorder

visit 1: check 4

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



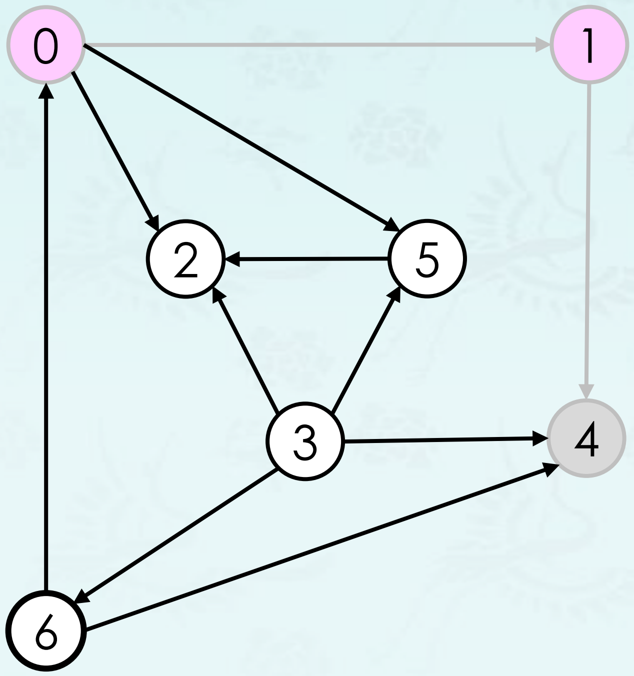


postorder

visit 4:

Topological sort demo

- **Run depth-first search.**
- **Return vertices in reverse postorder.**



adj[]

0	1	2	5	
1	4			
2				
3	2	4	5	6
4				
5	2			
6	0	4		

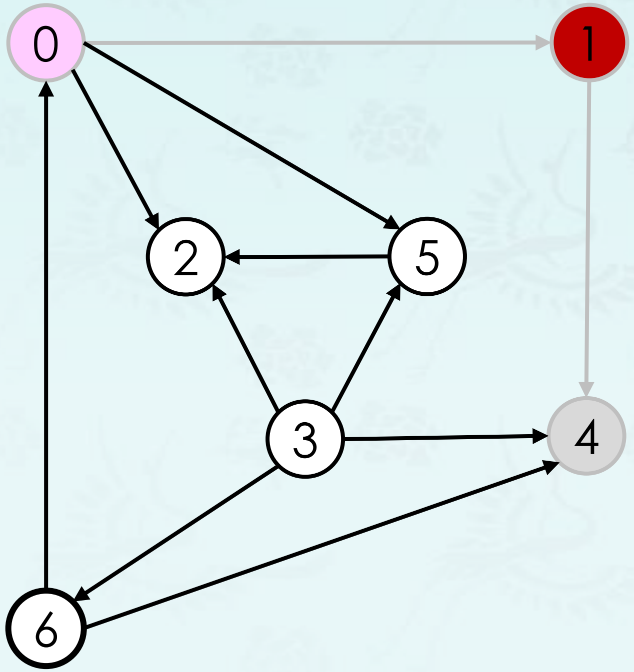
Once done, output to the stack

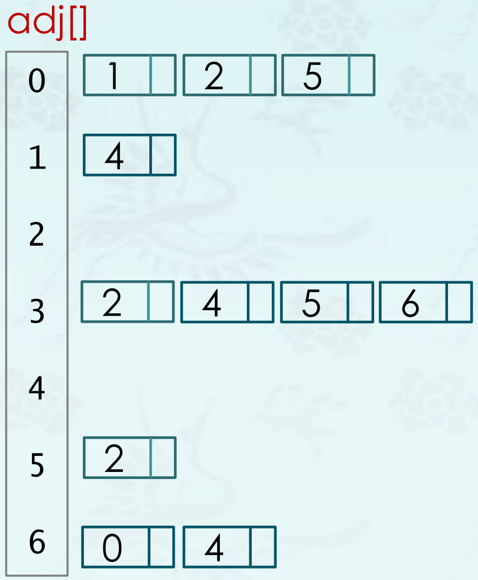
postorder
4

4 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



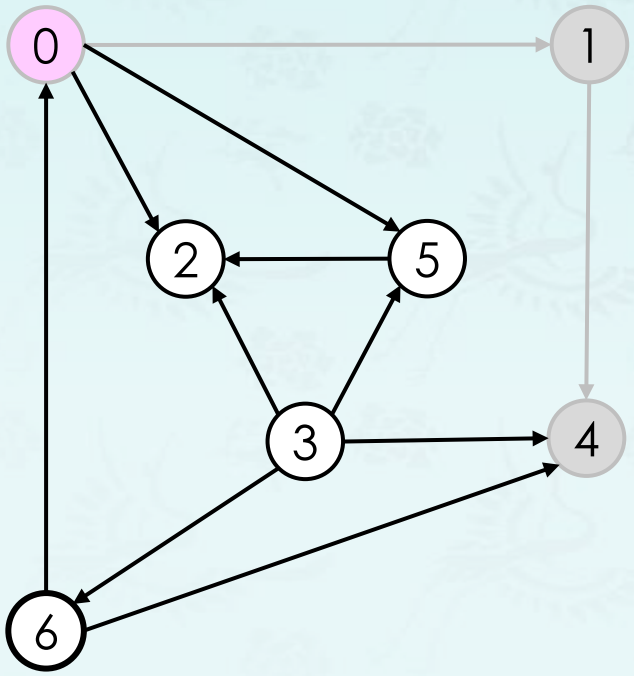


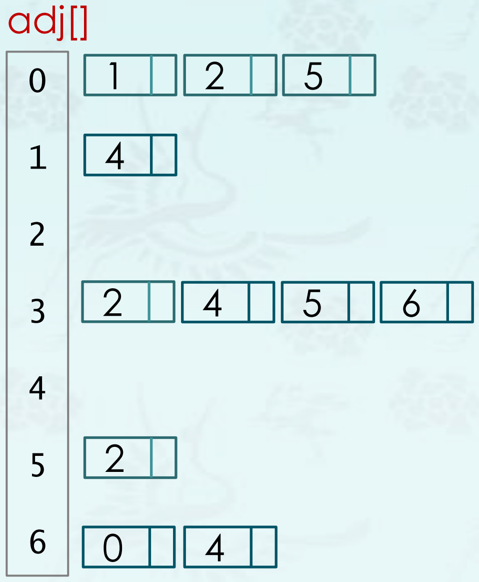
postorder
4

1 done

Topological sort demo

- **Run depth-first search.**
- **Return vertices in reverse postorder.**



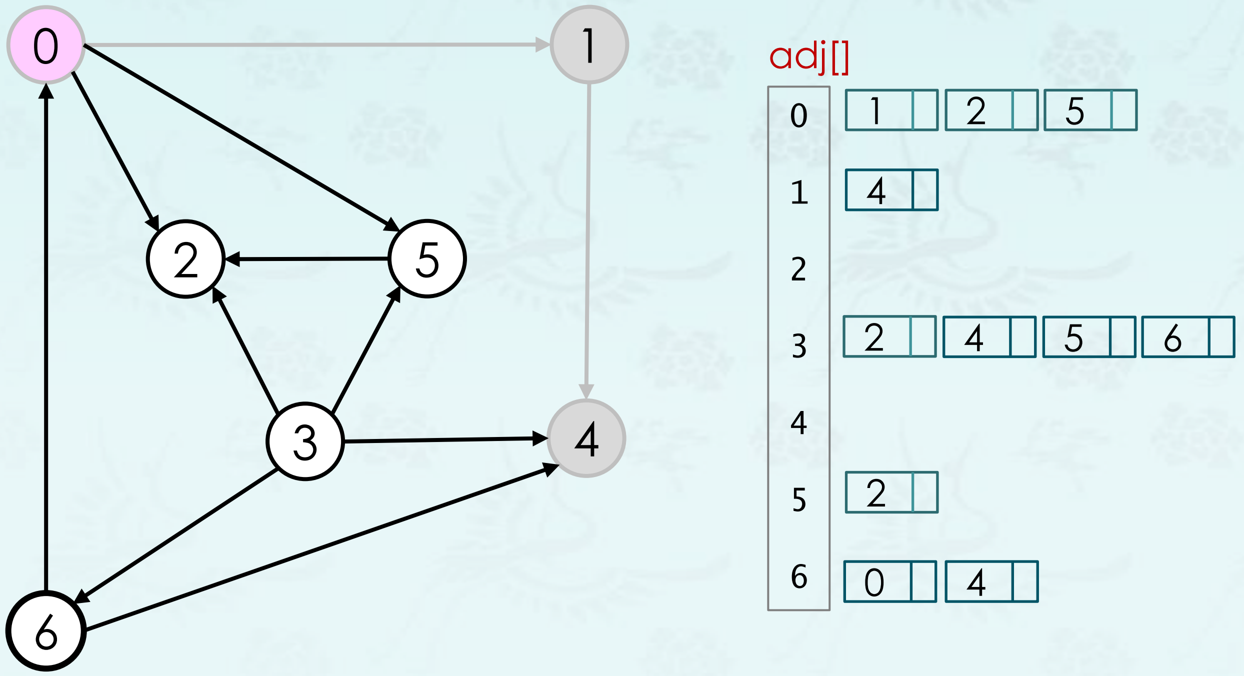


postorder
4 1

1 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



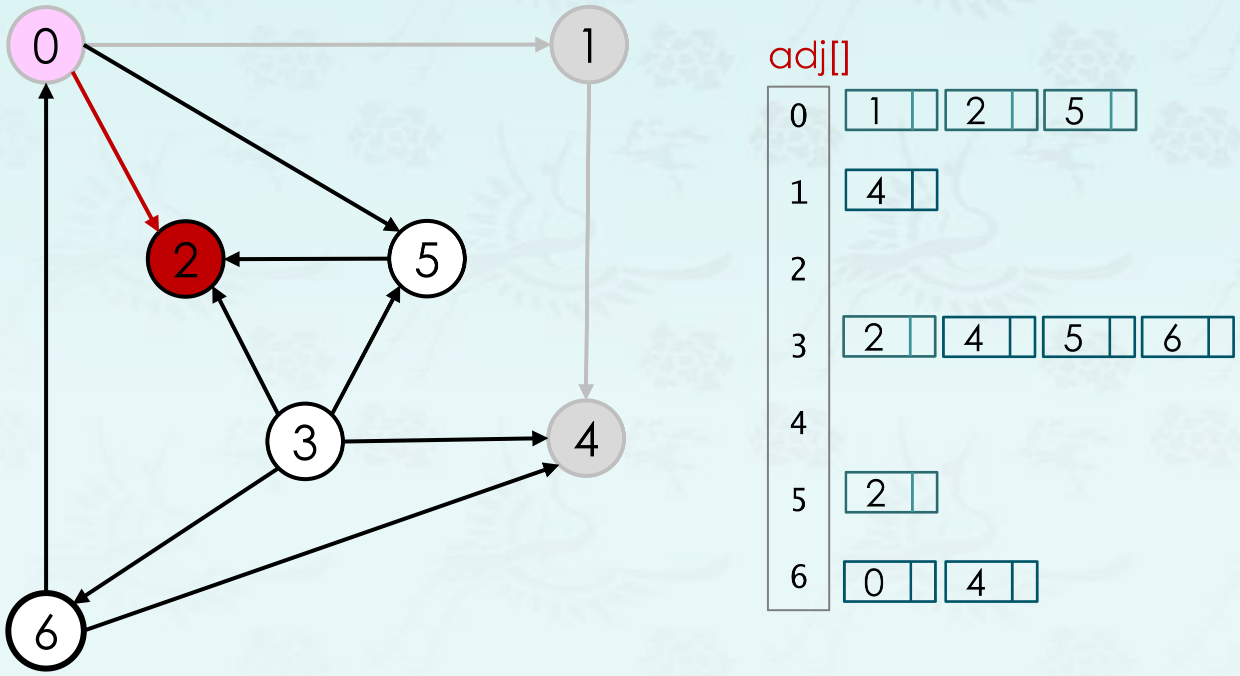
postorder

4 1

visit 0: check 1, **check 2**, and check 5

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



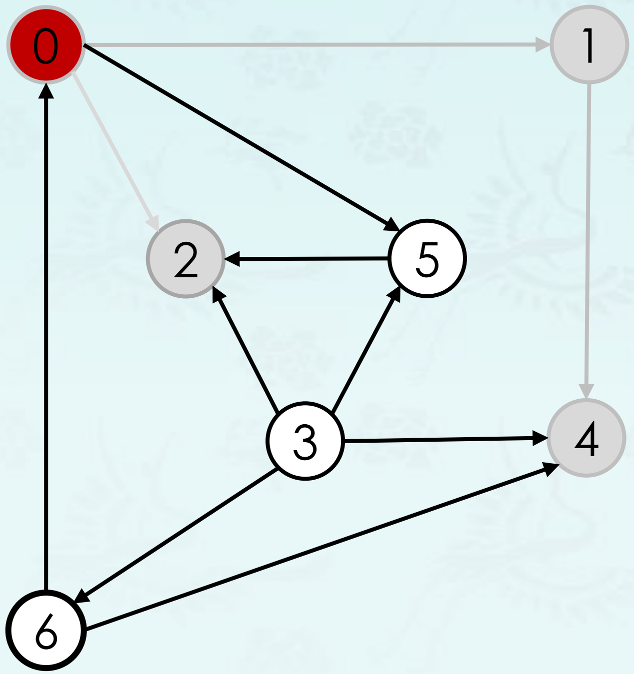
postorder

4 1

visit 2

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



adj[]

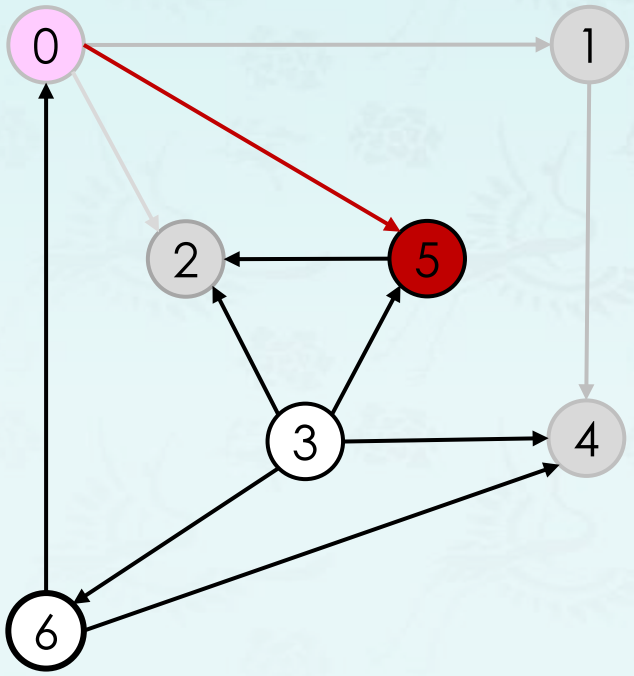
0: 1 2 5
1: 4
2:
3: 2 4 5 6
4:
5: 2
6: 0 4

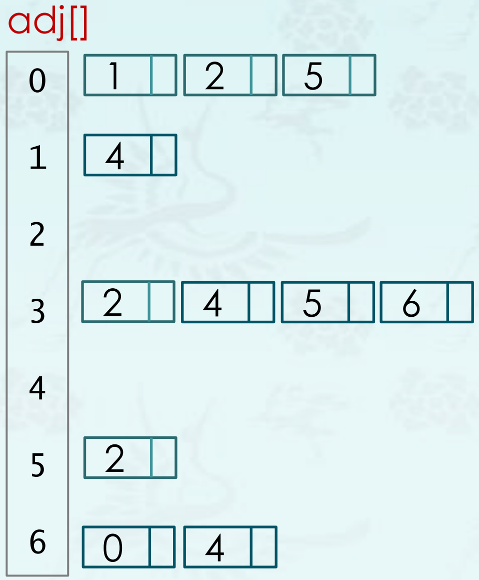
postorder
4 1 2

2 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.





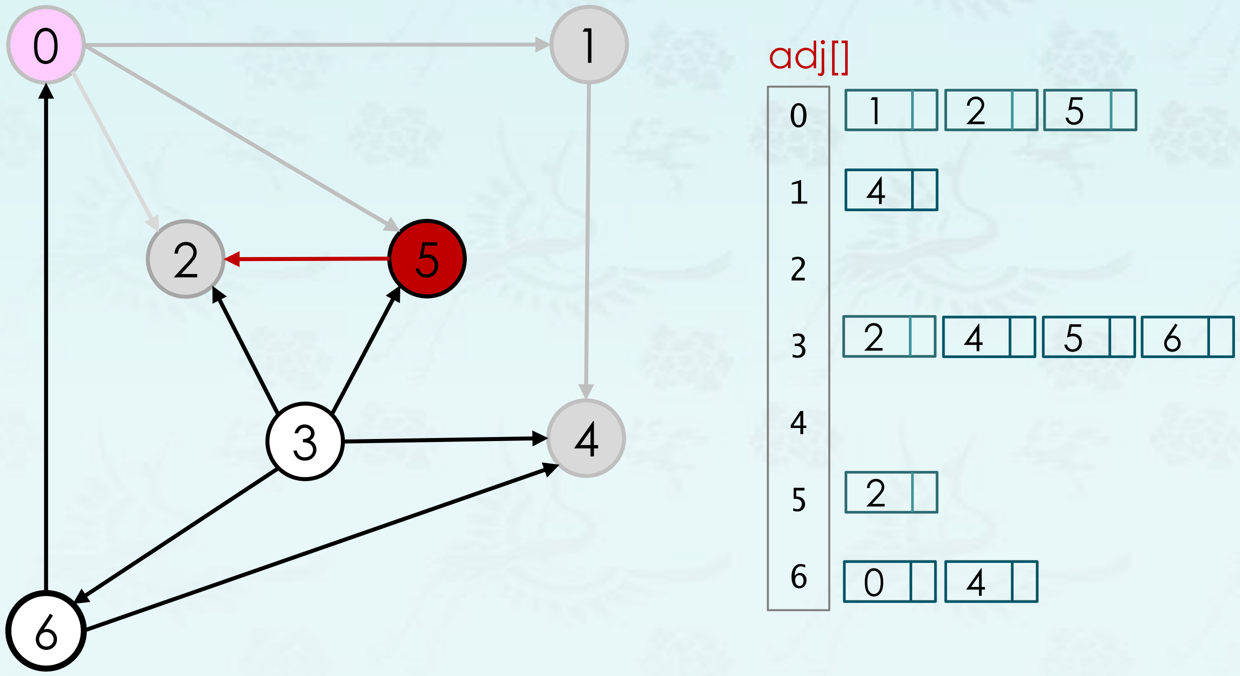
postorder

4 1 2

visit 5: check 2

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

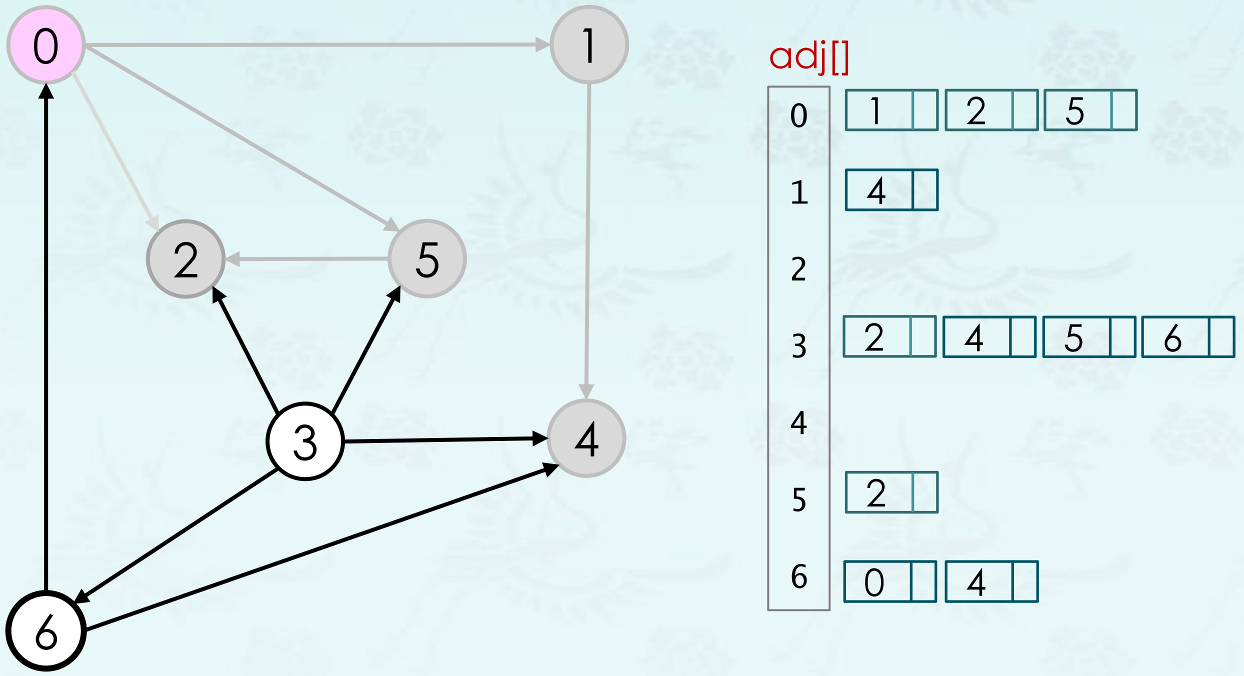


postorder
4 1 2

visit 5: check 2

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

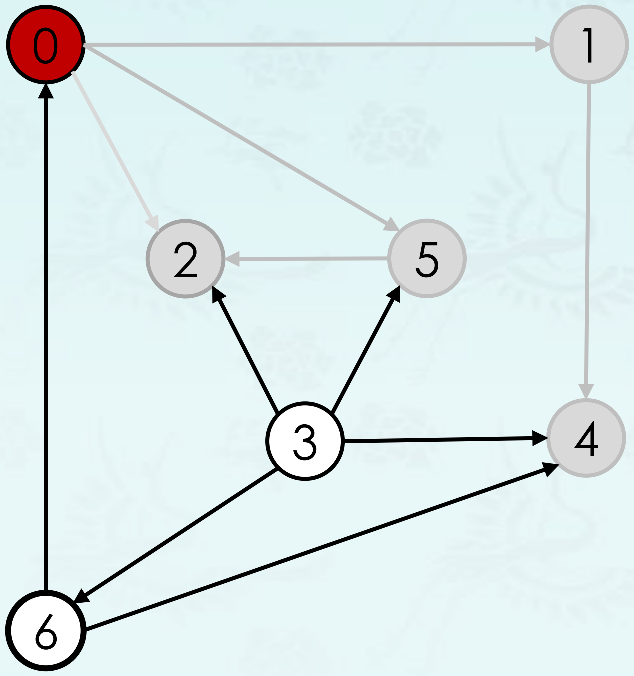


postorder
4 1 2 5

5 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



adj[]

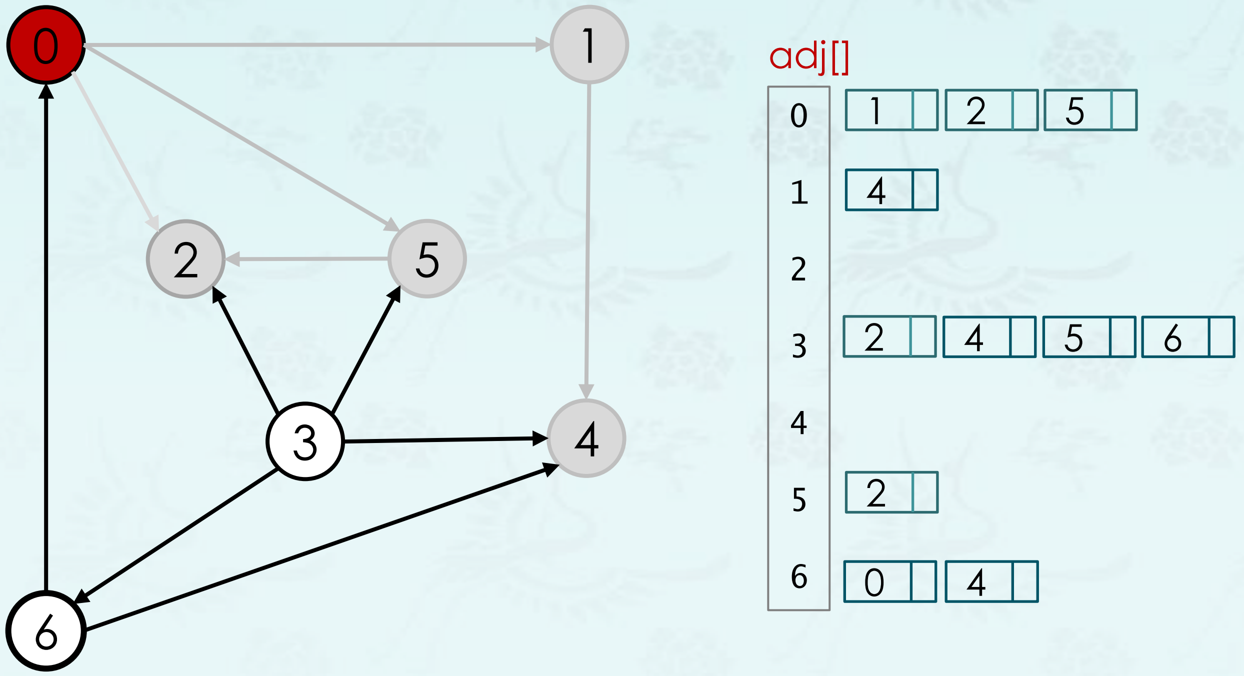
0	1 → 2 → 5
1	4
2	
3	2 → 4 → 5 → 6
4	
5	2
6	0 → 4

postorder
4 1 2 5

5 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

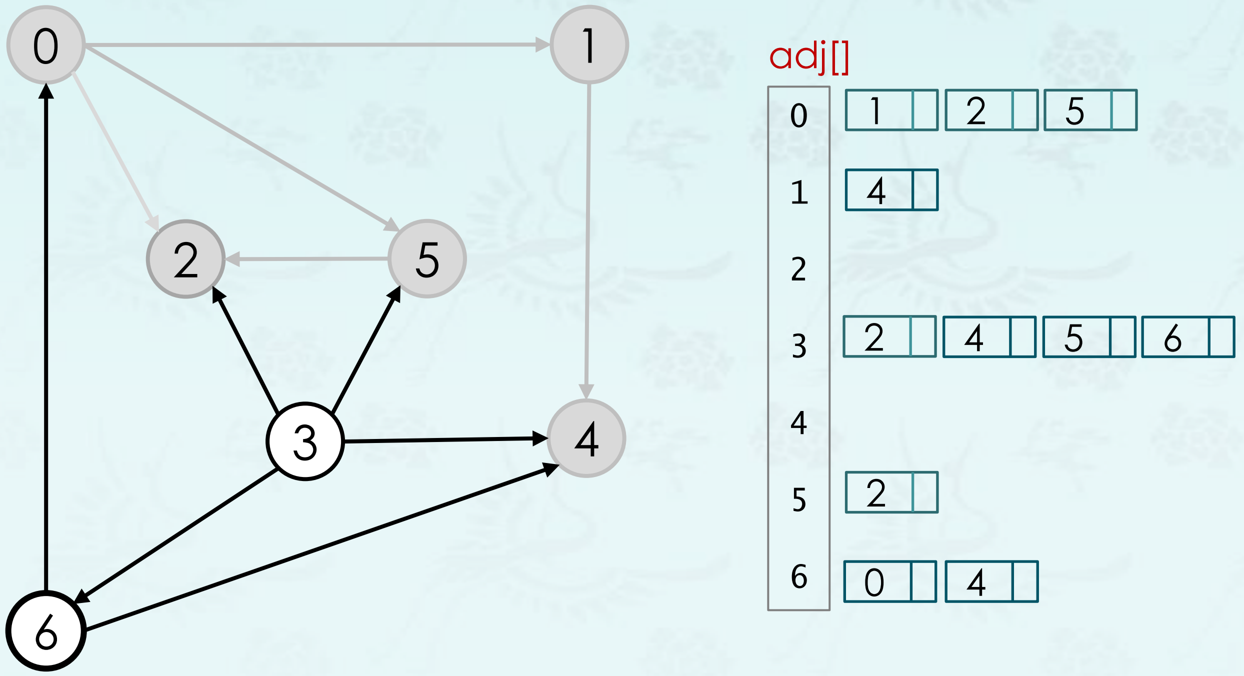


postorder
4 1 2 5 0

0 done

Topological sort demo

- **Run depth-first search.**
- **Return vertices in reverse postorder.**



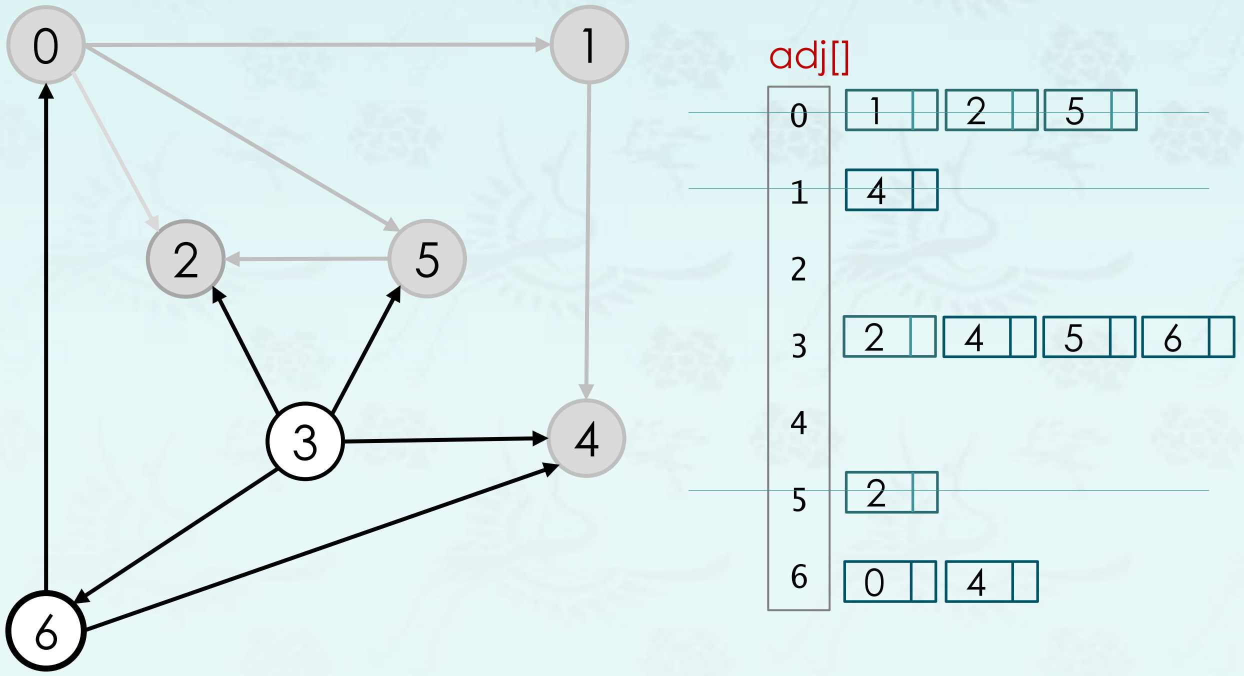
postorder

4 1 2 5 0

0 done since 1, 2, 5 are done, then 0 is done.

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

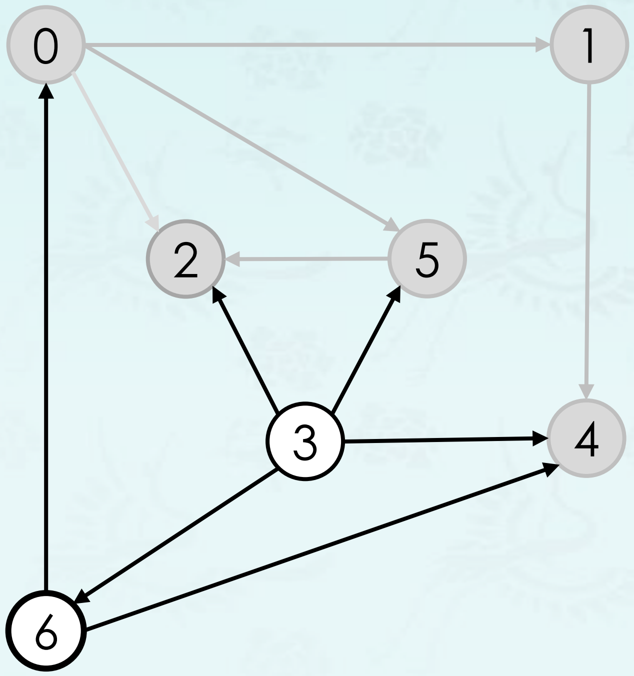


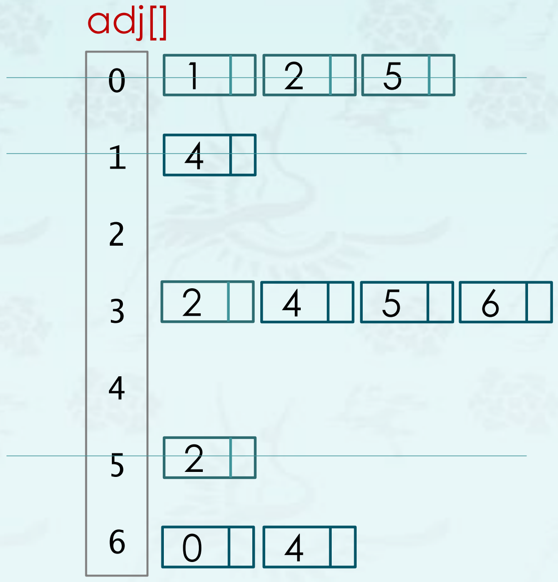
postorder
4 1 2 5 0

0 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.





postorder

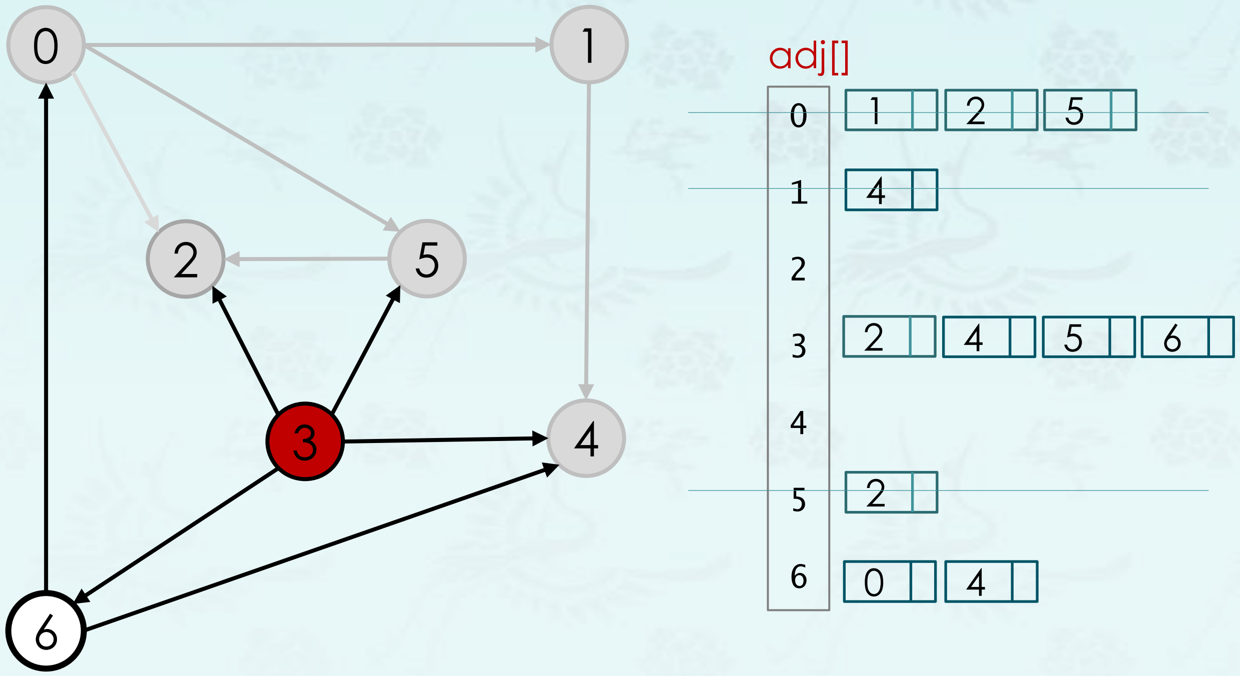
4 1 2 5 0

```
dfs(0)
   dfs(1)
      dfs(4)
      4 done
   1 done
   dfs(2)
   2 done
   dfs(5)
      check 2
   5 done
0 done
```

0 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

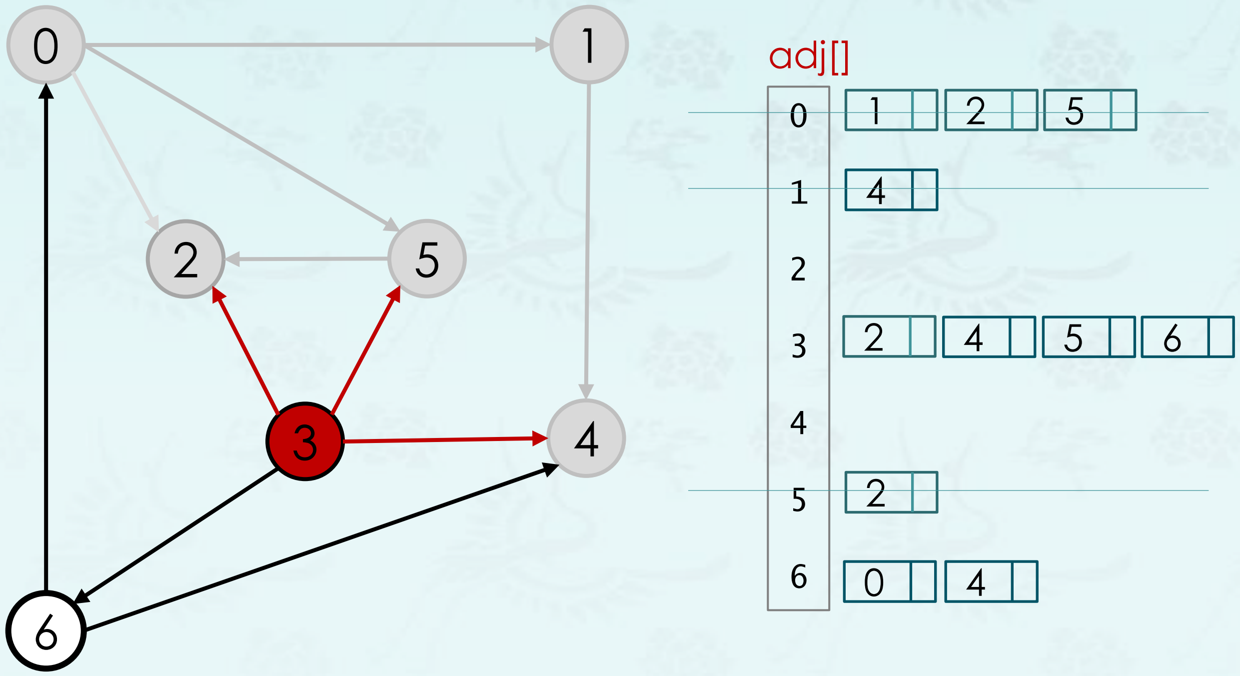


postorder
4 1 2 5 0

visit 3: check 2, check 4, check 5, and check 6

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



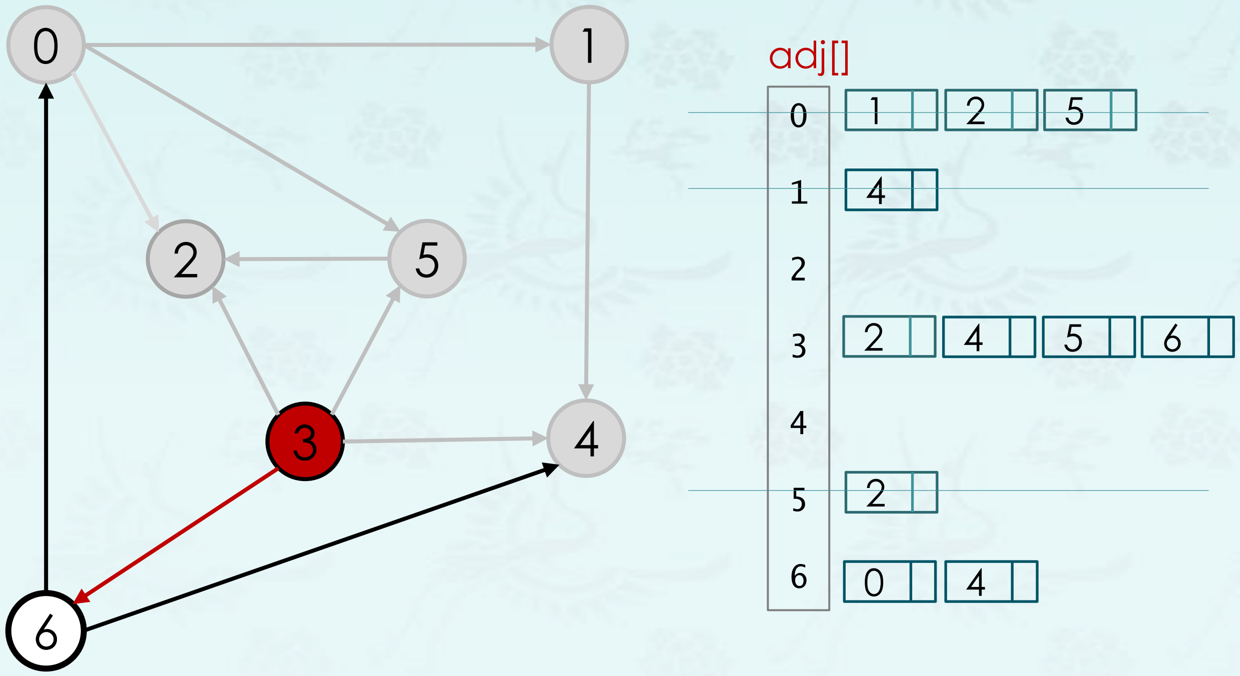
postorder

4 1 2 5 0

visit 3: check 2, check 4, check 5, and check 6

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

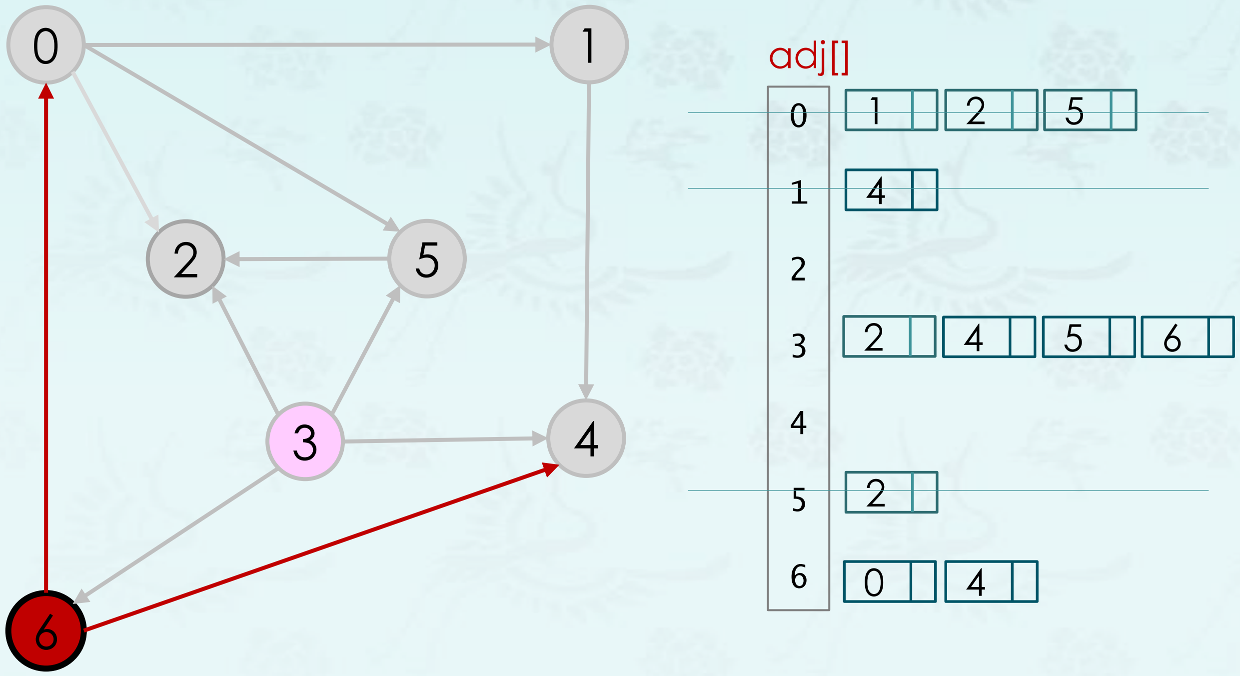


postorder
4 1 2 5 0

visit 3: check 2, check 4, check 5, and **check 6**

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

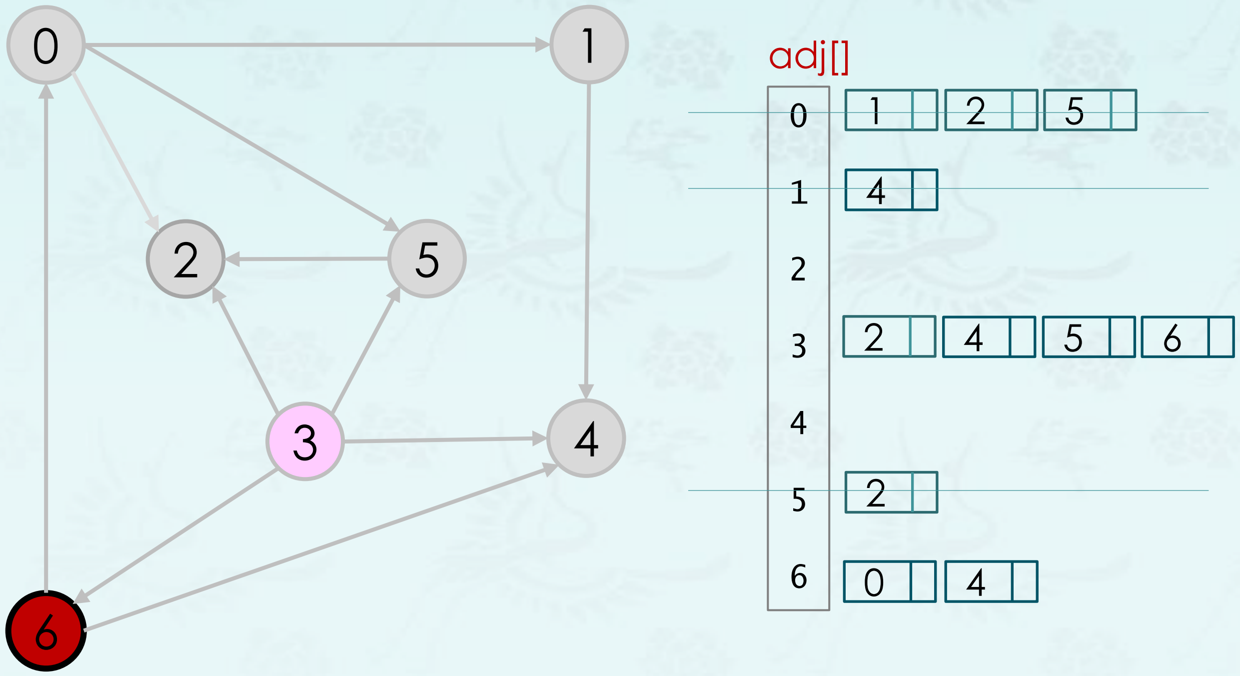


postorder
4 1 2 5 0

visit 6: check 0 and check 4

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

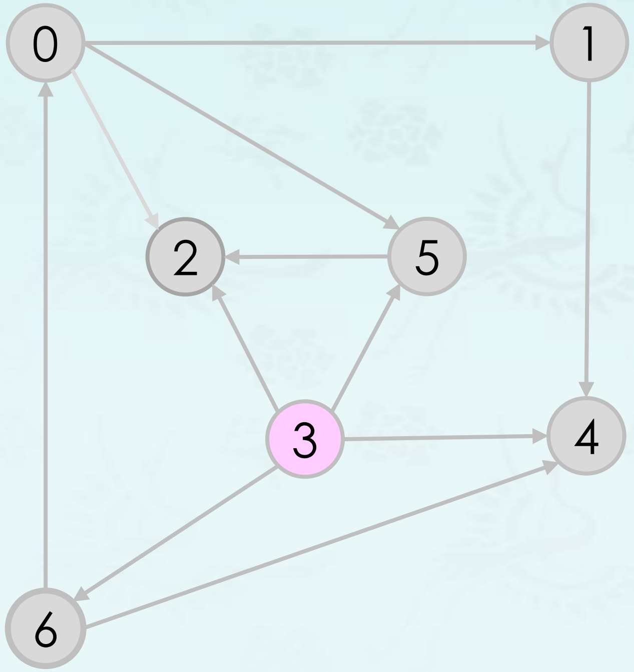


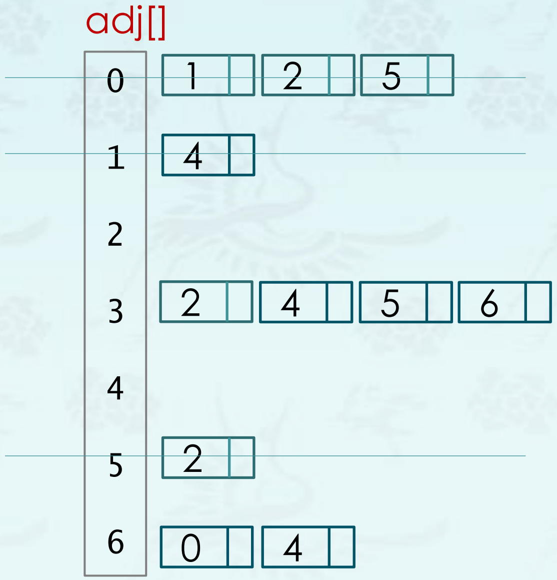
postorder
4 1 2 5 0

6 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.





postorder

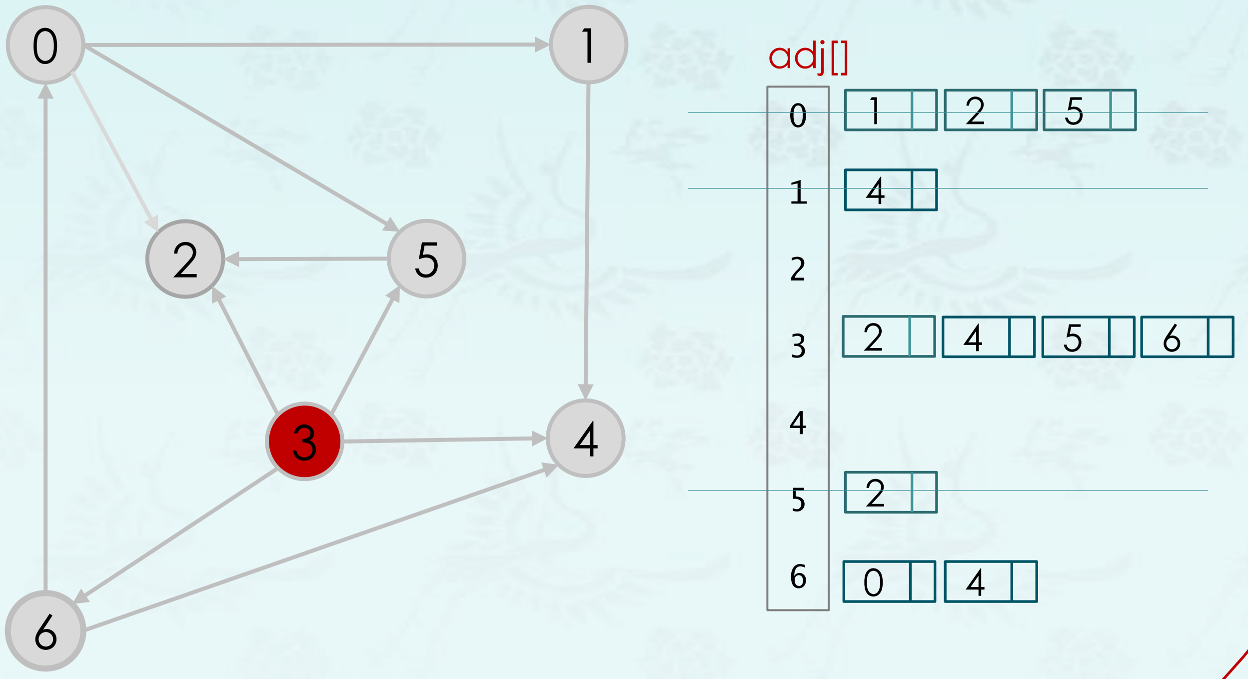
4 1 2 5 0

6

6 done

Topological sort demo

- **Run depth-first search.**
- **Return vertices in reverse postorder.**

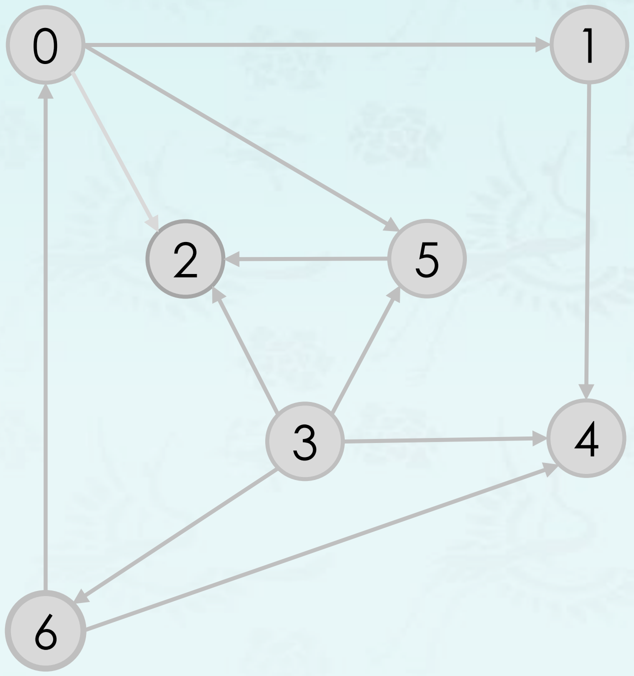


postorder
4 1 2 5 0 6 3

3 done since 0, and 4 are done, then, 6 is done,
since 2, 4, 5, and 6 are done, then 3 is done.

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



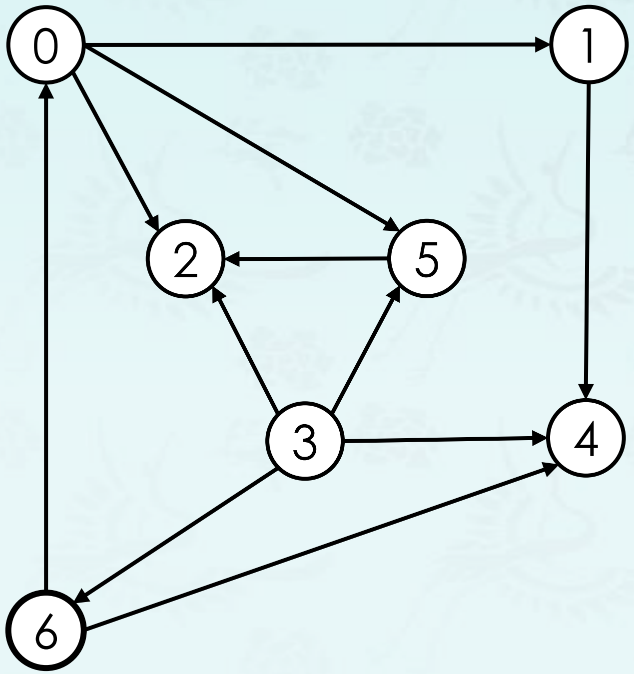
postorder
4 1 2 5 0 6 3

done

```
dfs(0)
  dfs(1)
    dfs(4)
    4 done
  1 done
  dfs(2)
  2 done
  dfs(5)
    check 2
  5 done
0 done
check 1
check 2
dfs(3)
  check 2
  check 4
  check 5
  dfs(6)
    check 0
    check 4
  6 done
3 done
check 4
check 5
check 6
done
```

Topological sort demo

- Run depth-first search.
- Return vertices in **reverse postorder.**



postorder

4 1 2 5 0 6 3

```
dfs(0)
  dfs(1)
    dfs(4)
    4 done
  1 done
  dfs(2)
  2 done
  dfs(5)
    check 2
  5 done
0 done
check 1
check 2
dfs(3)
  check 2
  check 4
  check 5
  dfs(6)
    check 0
    check 4
  6 done
3 done
check 4
check 5
check 6
done
```

Topological sort (reverse postorder) **: 3 6 0 5 2 1 4**

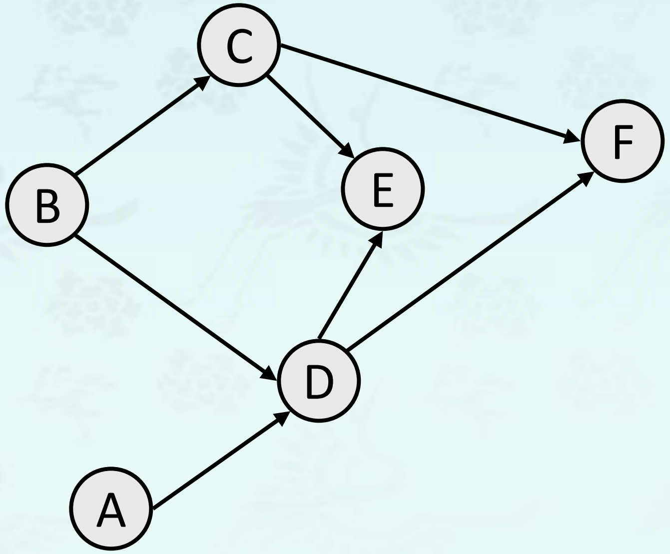
Depth-first search order – topological sort

```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Stack<Integer> reversePost;

   public DepthFirstOrder(Digraph G)
   {
      reversePost = new Stack<Integer>();
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) dfs(G, v);
   }

   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
      reversePost.push(v);
   }

   public Iterable<Integer> reversePost()
   {  return reversePost;  }
}
```

returns all vertices in "reverse DFS postorder"

Topological sort demo

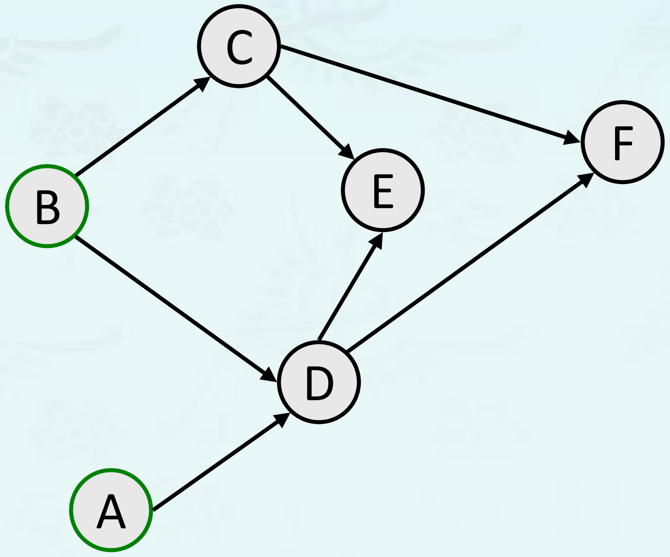
- How many valid topological sort orderings can you find for the vertices in the graph below?



- [A, B, C, D, E, F], [A, B, C, D, F, E],
- [A, B, D, C, E, F], [A, B, D, C, F, E],
- [B, A, C, D, E, F], [B, A, C, D, F, E],
- [B, A, D, C, E, F], [B, A, D, C, F, E],
- [B, C, A, D, E, F], [B, C, A, D, F, E],
- …

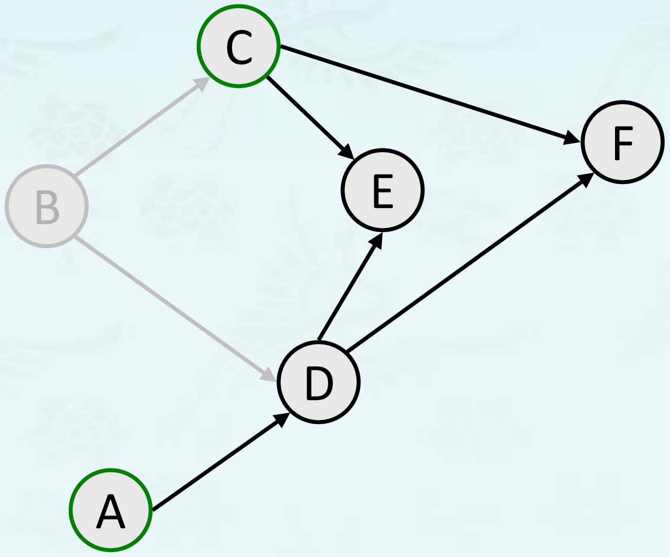
Topological sort – Algorithm I

- function topologicalSort():
  - *ordering* := { }.
  - Repeat until graph is empty:
    - Find a vertex v with **in-degree of 0** (no incoming edges).
      - (If there is no such vertex, the graph cannot be sorted; stop.)
    - Delete v and all of its outgoing edges from the graph.
    - *ordering* += *v* .



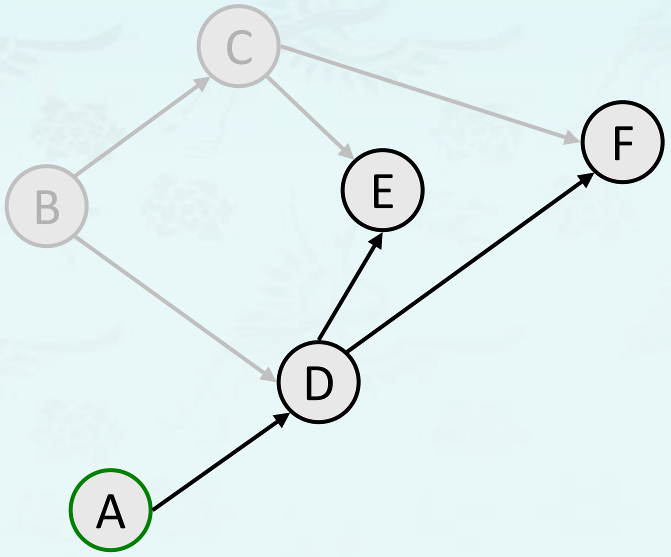
Topological sort – Example

- function topologicalSort():
  - *ordering* := { }.
  - Repeat until graph is empty:
    - Find a vertex v with in-degree of 0 (no incoming edges).
      - (If there is no such vertex, the graph cannot be sorted; stop.)
    - Delete v and all of its outgoing edges from the graph.
    - *ordering* += v .
  - *ordering* = { B }



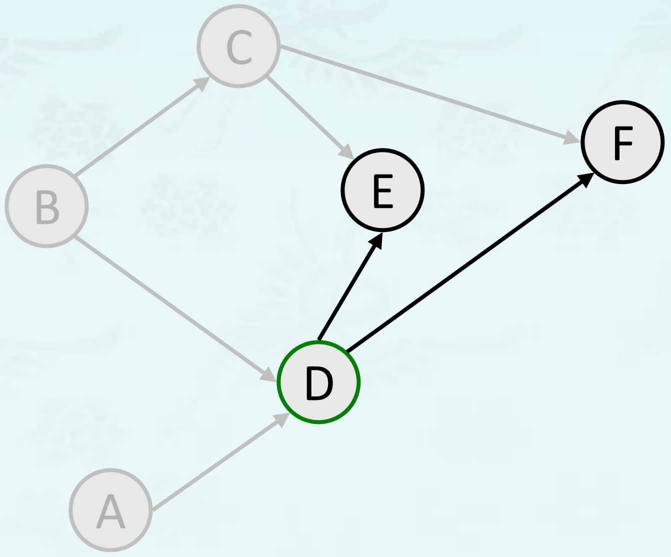
Topological sort – Example

- function topologicalSort():
  - *ordering* := { }.
  - Repeat until graph is empty:
    - Find a vertex v with in-degree of 0 (no incoming edges).
      - (If there is no such vertex, the graph cannot be sorted; stop.)
    - Delete v and all of its outgoing edges from the graph.
    - *ordering* += v .
  - *ordering* = { B, C }



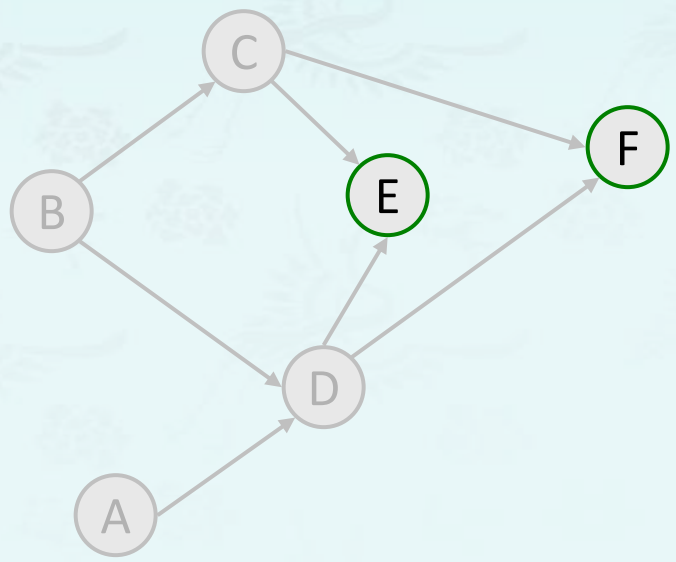
Topological sort – Example

- function topologicalSort():
  - *ordering* := { }.
  - Repeat until graph is empty:
    - Find a vertex v with in-degree of 0 (no incoming edges).
      - (If there is no such vertex, the graph cannot be sorted; stop.)
    - Delete v and all of its outgoing edges from the graph.
    - *ordering* += v .
  - *ordering* = { B, C, A }



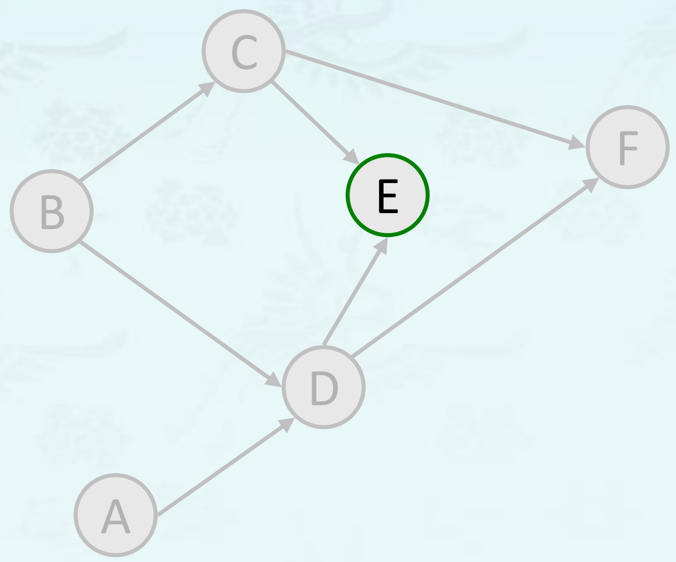
Topological sort – Example

- function topologicalSort():
  - *ordering* := { }.
  - Repeat until graph is empty:
    - Find a vertex v with in-degree of 0 (no incoming edges).
      - (If there is no such vertex, the graph cannot be sorted; stop.)
    - Delete v and all of its outgoing edges from the graph.
    - *ordering* += v .
  - *ordering* = { B, C, A, D }



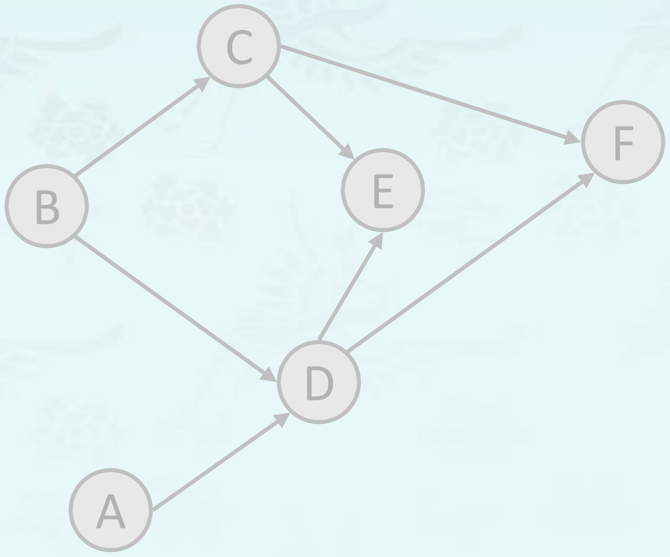
Topological sort – Example

- function topologicalSort():
  - *ordering* := { }.
  - Repeat until graph is empty:
    - Find a vertex v with in-degree of 0 (no incoming edges).
      - (If there is no such vertex, the graph cannot be sorted; stop.)
    - Delete v and all of its outgoing edges from the graph.
    - *ordering* += v .
  - *ordering* = { B, C, A, D, F }



Topological sort – Example

- function topologicalSort():
  - *ordering* := { }.
  - Repeat until graph is empty:
    - Find a vertex v with in-degree of 0 (no incoming edges).
      - (If there is no such vertex, the graph cannot be sorted; stop.)
    - Delete v and all of its outgoing edges from the graph.
    - *ordering* += v .
  - *ordering* = { B, C, A, D, F, E }



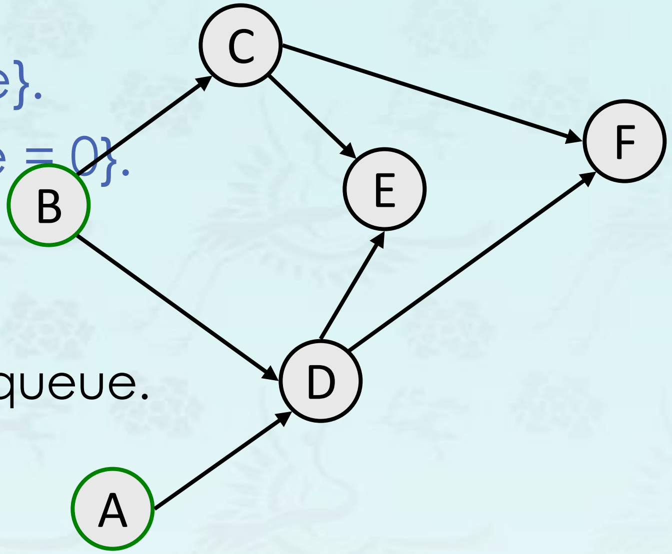
Topological sort – Revised Algorithm

We don't want to literally delete vertices and edges from the graph while trying to topological sort it; so let's revise the 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}.
- If all vertices are processed, success.
  Otherwise, there is a cycle.

Topological sort – Example 2 with revised algorithm

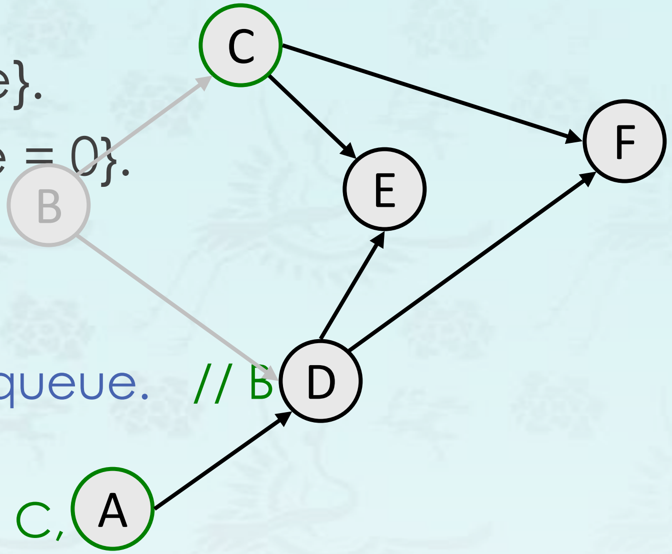
- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex v from the queue.
    - *ordering* += v.
    - Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.
    - *queue* += {any neighbors whose in-degree is now 0}.



- map := { A=0, B=0, C=1, D=2, E=2, F=2 }
- queue := { B, A }
- ordering := { }

Topological sort – Example 2 with revised algorithm

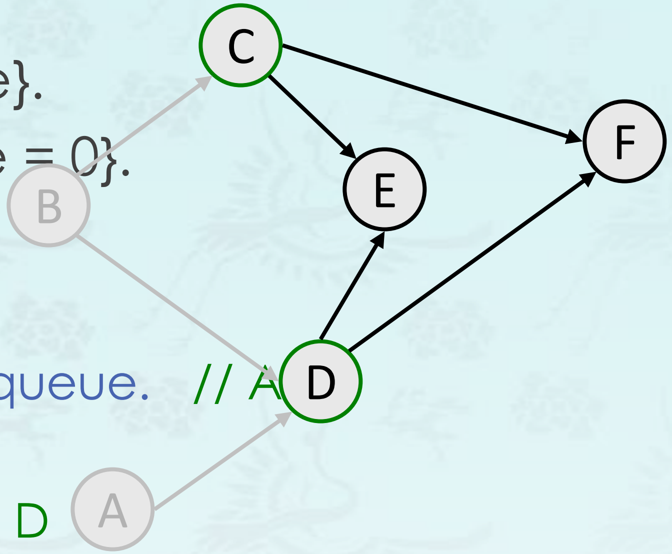
- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex v from the queue. // B
    - *ordering* += v.
    - Decrease the in-degree of all v's neighbors by 1 in the *map*. // C,
    - *queue* += {any neighbors whose in-degree is now 0}.



- map := { A=0, B=0, **C=0**, **D=1**, E=2, F=2 }
- queue := { B, A, **C** }
- ordering := { **B** }

Topological sort – Example 2 with revised algorithm

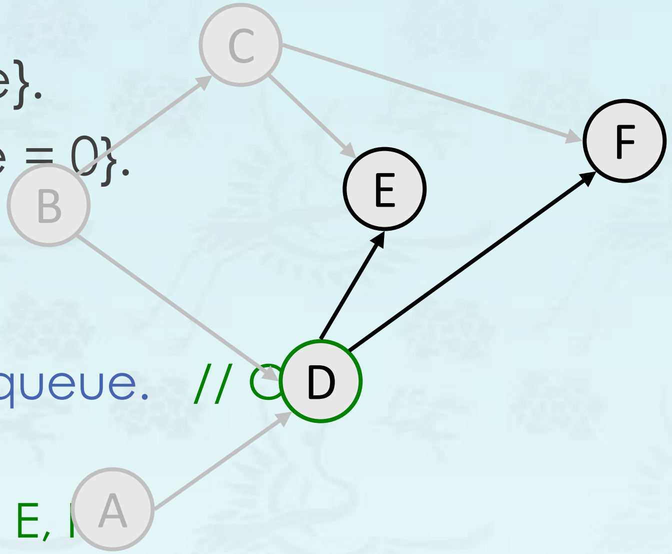
- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex v from the queue. // A
    - *ordering* += v.
    - Decrease the in-degree of all v's neighbors by 1 in the *map*. // D
    - *queue* += {any neighbors whose in-degree is now 0}.



- map := { A=0, B=0, C=0, **D=0**, E=2, F=2 }
- queue := { B, A, C, **D** }
- ordering := { B, **A** }

Topological sort – Example 2 with revised algorithm

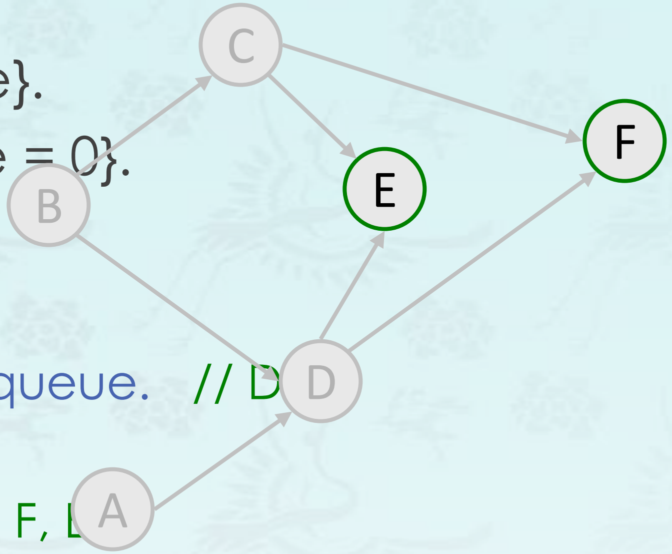
- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex v from the queue. // C
    - *ordering* += v.
    - Decrease the in-degree of all v's neighbors by 1 in the *map*. // E,
    - *queue* += {any neighbors whose in-degree is now 0}.



- map := { A=0, B=0, C=0, D=0, E=**1**, F=**1** }
- queue := { B, A, C, D }
- ordering := { B, A, **C** }

Topological sort – Example 2 with revised algorithm

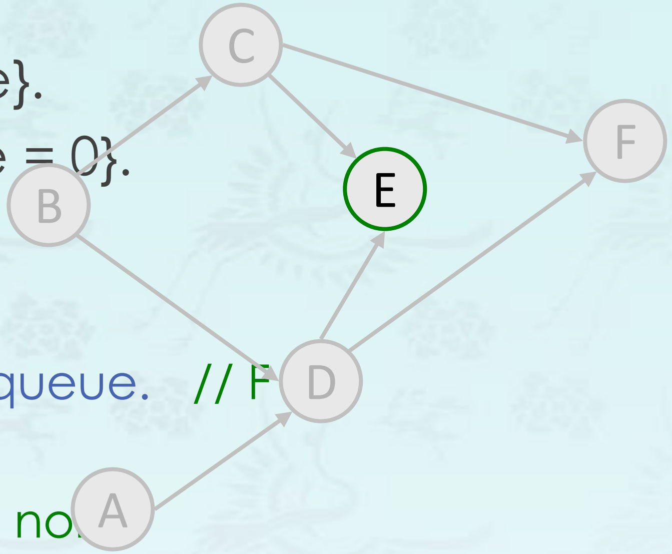
- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex v from the queue. // D
    - *ordering* += v.
    - Decrease the in-degree of all v's neighbors by 1 in the *map*. // F,
    - *queue* += {any neighbors whose in-degree is now 0}.



- map := { A=0, B=0, C=0, D=0, E=**0**, F=**0** }
- queue := { B, A, C, D, **F**, **E** }
- ordering := { B, A, C, **D** }

Topological sort – Example 2 with revised algorithm

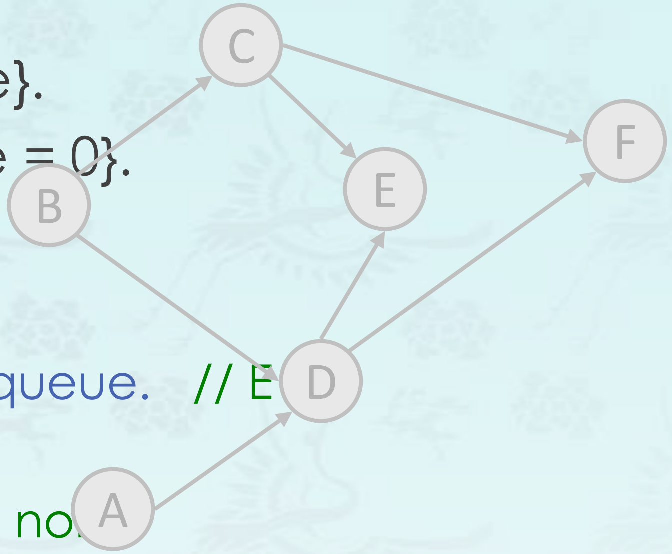
- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex v from the queue. // F
    - *ordering* += v.
    - Decrease the in-degree of all v's neighbors by 1 in the *map*. // no
    - *queue* += {any neighbors whose in-degree is now 0}.



- map := { A=0, B=0, C=0, D=0, E=0, F=0 }
- queue := { B, A, C, D, F, E }
- ordering := { B, A, C, D, **F** }

Topological sort – Example 2 with revised algorithm

- function topologicalSort():
  - *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. // E
    - *ordering* += v.
    - Decrease the in-degree of all v's neighbors by 1 in the *map*. // no
    - *queue* += {any neighbors whose in-degree is now 0}.



- map := { A=0, B=0, C=0, D=0, E=0, F=0 }
- queue := { B, A, C, D, F, E}
- ordering := { B, A, C, D, F, **E** }

Topological sorting is a linear arrangement of vertices such that for every directed edge **uv** from vertex u to vertex v, **u** comes before **v** in the ordering.

Topological sort – Time Complexity

What is the time complexity of our topological sort algorithm?

- (with an "adjacency map" graph internal representation)

- function topologicalSort():
  - *map* := {each vertex → its in-degree}. // O(V)
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty: // O(V)
    - Dequeue the first vertex v from the queue. // O(1)
    - *ordering* += v. // O(1)
    - Decrease the in-degree of all v's neighbors by 1 in the *map*. // O(*E*) for all passes
    - *queue* += {any neighbors whose in-degree is now 0}.

- Overall: **O(*V* + *E*)** ; essentially O(V) time on a **sparse** graph (fast!)

Graph

- Graph
  - Introduction
  - Adjacency list
  - DFS, BFS
  - Challenges
- **Digraph – Directed Graphs**
  - digraph – DFS, BFS
  - **Applications – crawl web, topological sort**
- Minimum Spanning Tree(MST)

Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet