Faculty of Information Technology,
Monash University

FIT2004: Algorithms and Data Structures

Week 12: Topological Sort and Design Principles

These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Announcements/Things to note

- Complete SETU before it closes (21 June)
- The post-semester consultation schedule is the same as the in semester consultation schedule
- Any changes to the schedule will be announced on Moodle

Overview

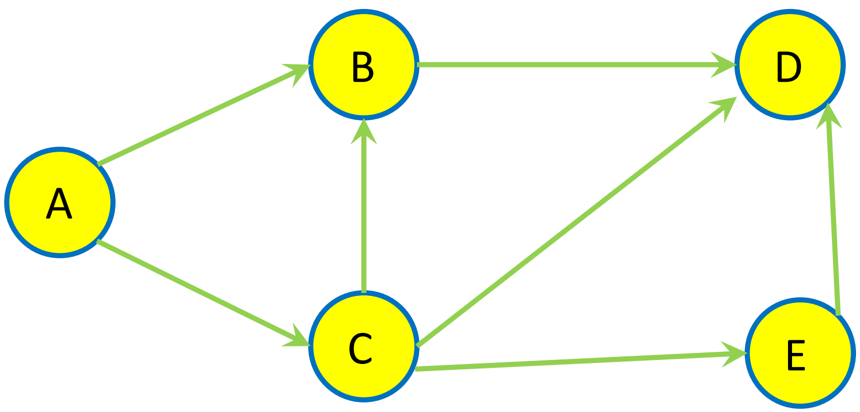
- Topological Sort
  - Kahn’s Algorithm
  - Depth First Search
- Design Principles (FIT2004 Summary)
- Final Exam etc.
- Review of all lecture material

Directed Acyclic Graph (DAG)

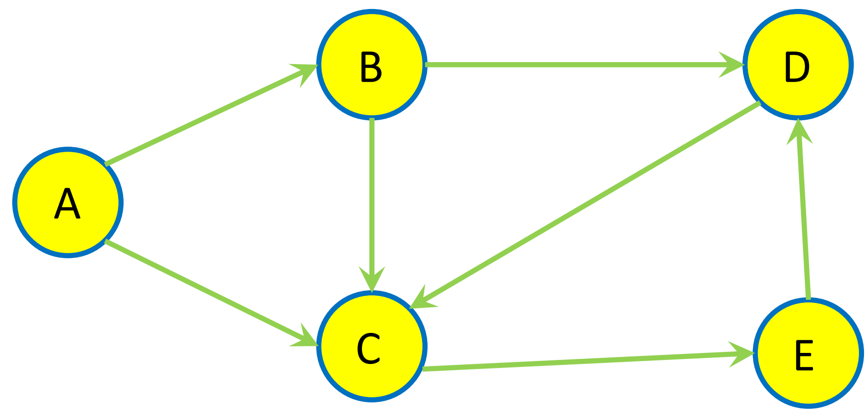
A Directed Acyclic Graph (DAG) is

- Directed
- Acylcic – has no cycles
- Graph

Which of the two graphs is a DAG?



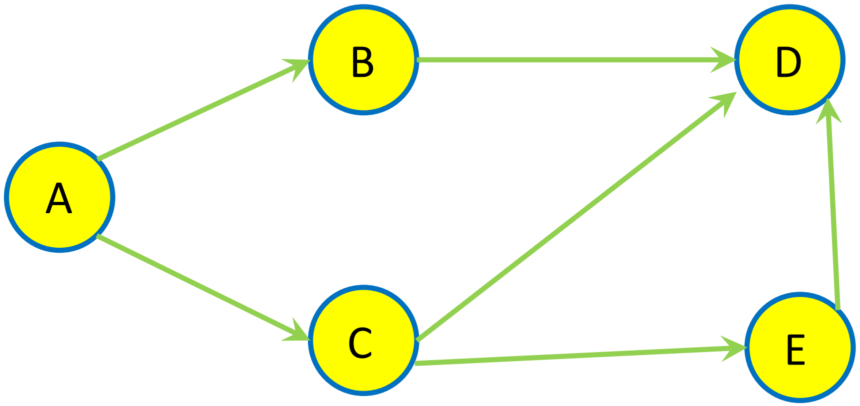
Graph 1

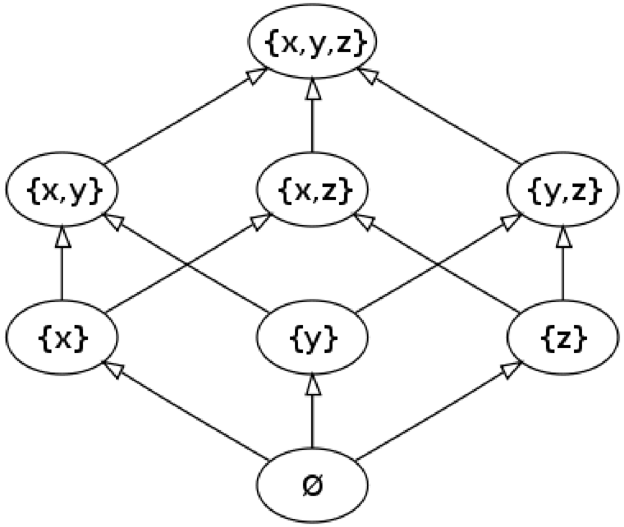


Graph 2

DAG: Examples

- sub-tasks of a project and which "must finish before"
  - A → B means task A must finish before task B
  - so, DAGs useful in project management
- relationships between subjects for your degree -- "is prerequisite for"
  - A→B means subject A must be completed before enrolling in subject B
- people genealogy – "is an ancestor of"
  - A → B means A is an ancestor of B
- power sets and "is a subset of"
  - A → B means A is a subset of B





Source: wikipedia

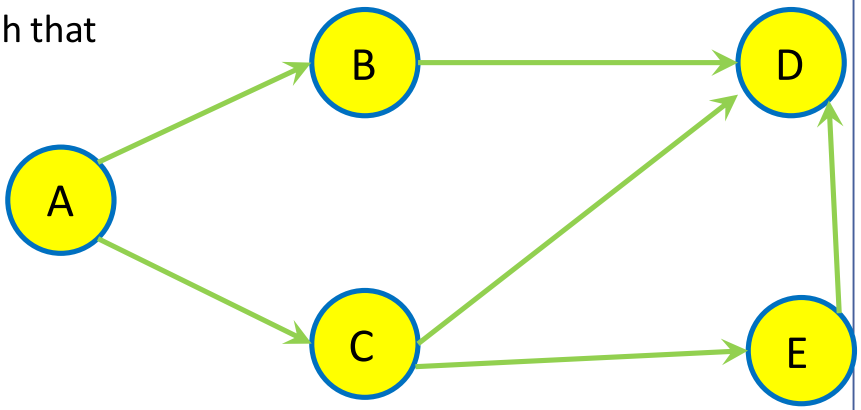
Topological Sort of a DAG

Order of vertices in a DAG

- A < B if A→B.
  - Note that if A → B and B→D, we have A < B and B < D which implies that A < D (i.e., transitivity).
- Some vertices may be incomparable (e.g., B and C are incomparable), i.e. A< B and A < C but we do not know whether C < B or B < C.

A topological order

- is a permutation of the vertices in the original DAG such that
- for **every** directed edge u→v of the DAG
  - u appears before v in the permutation



Example: A, B, C, E, D

- Topological sort of a DAG of "is prerequisite of" example gives an ordering of the subjects for studying your degree, one at a time, while obeying prerequisite rules.

Topological Sort of a DAG

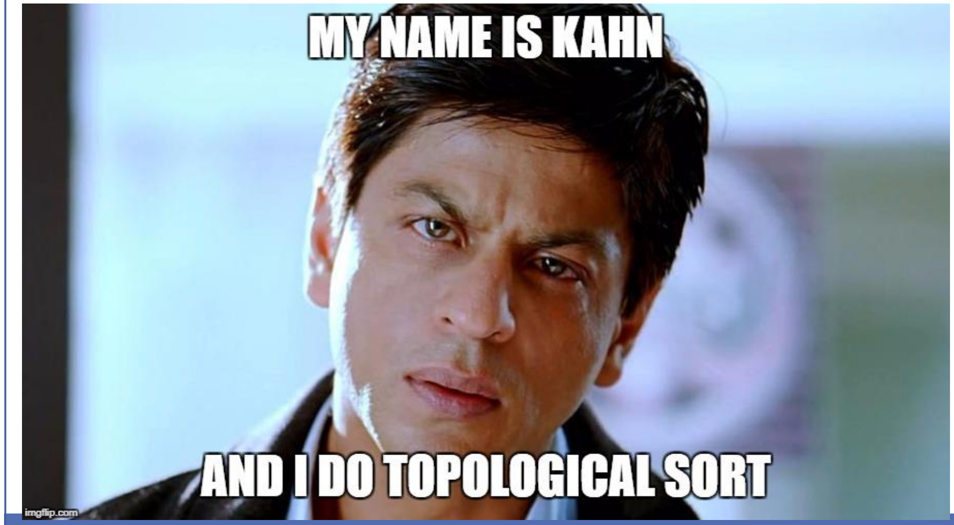
- A DAG can have many valid topological sorts, e.g., let u and v be two incomparable vertices, u may appear before or after v.

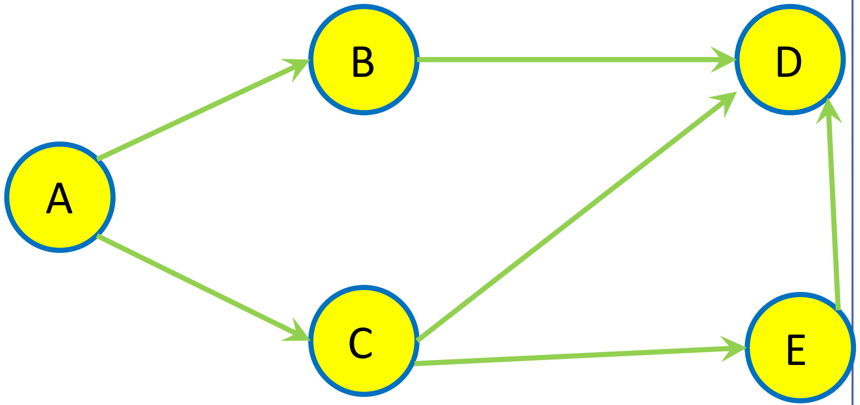
Which of these is NOT a valid topological ordering of the DAG

1. A, B, C, E, D
2. A, C, B, E, D
3. A, C, E, B, D
4. A, B, E, C, D

How to do topological sort?

- Kahn's Algorithm



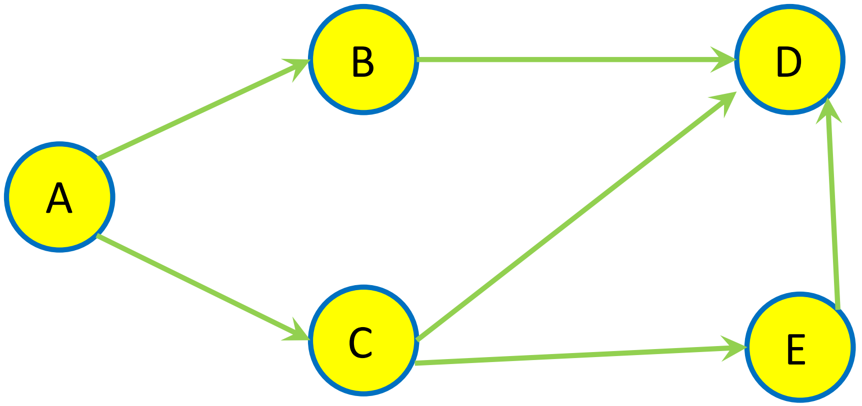


Overview

- Topological Sort
  - Kahn’s Algorithm
  - Depth First Search
- Design Principles (FIT2004 Summary)
- Final Exam etc.
- Review of all lecture material

Kahn’s Algorithm: High level idea

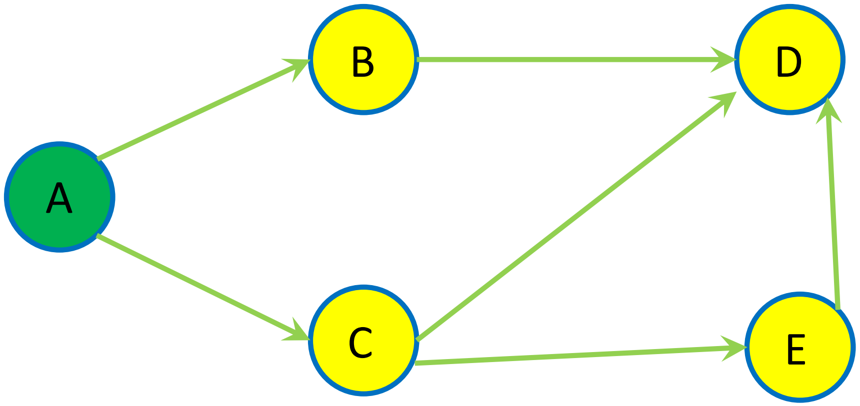
```
For each vertex v that does not have ANY incoming edge
    Add v to sorted
    Remove the outgoing edges of v
```



Sorted:

Kahn's Algorithm: High level idea

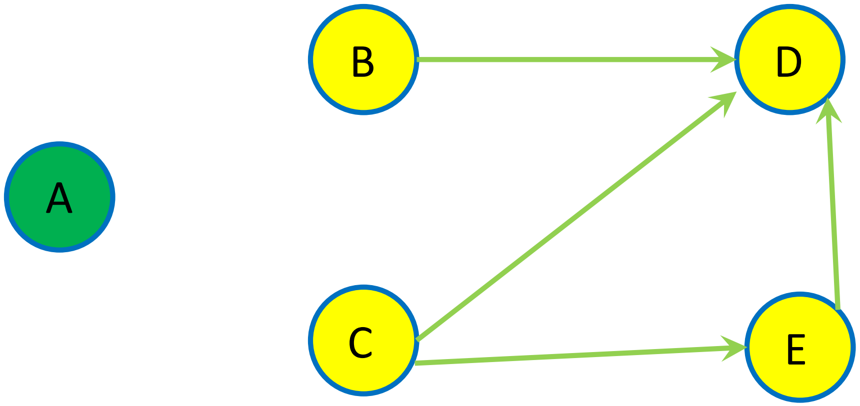
```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```



Sorted: **A**

Kahn's Algorithm: High level idea

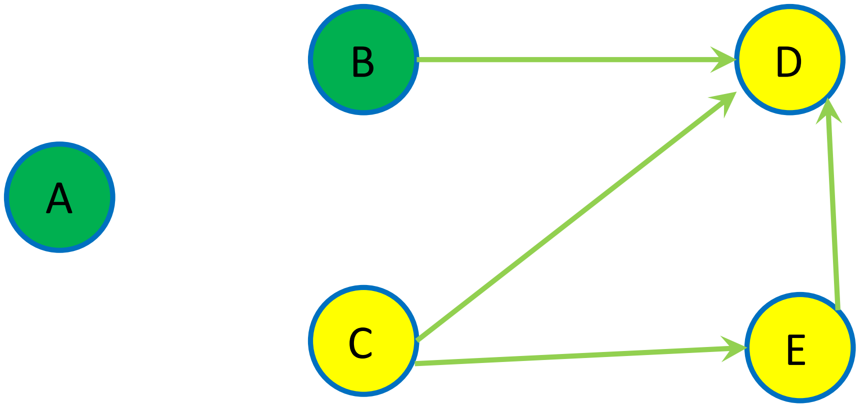
```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```



Sorted: **A**

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

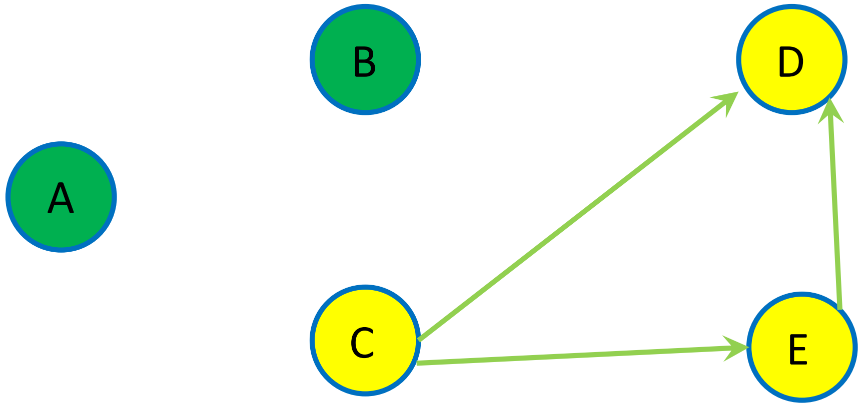


Sorted:

A	**B**

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

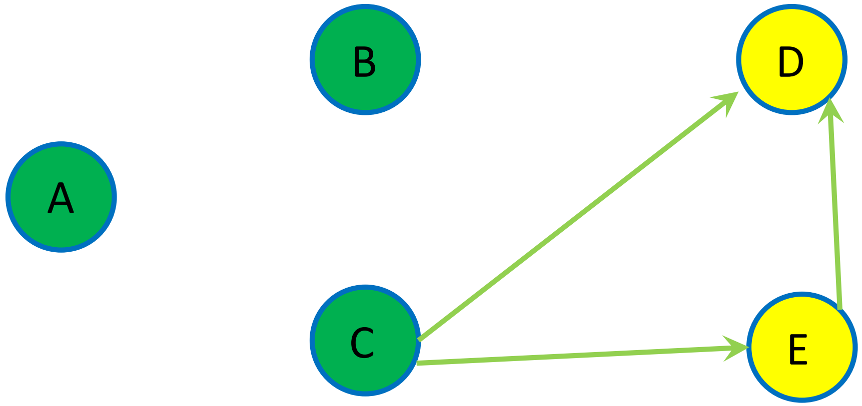


Sorted:

A	**B**

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

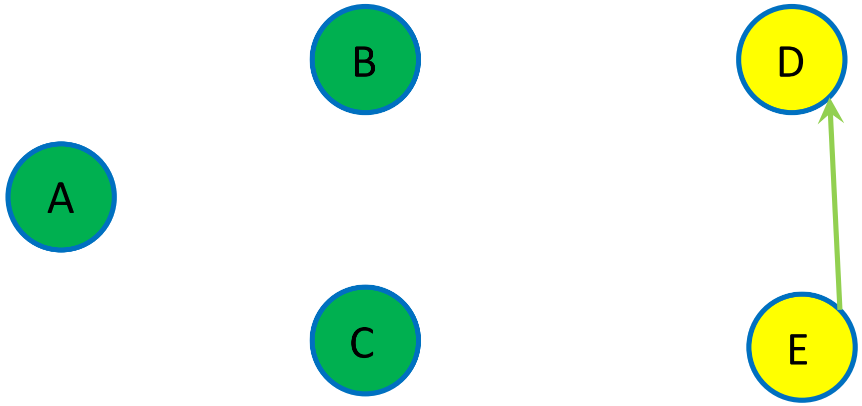


Sorted:

A	**B**	**C**

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
    Add v to sorted
    Remove the outgoing edges of v
```

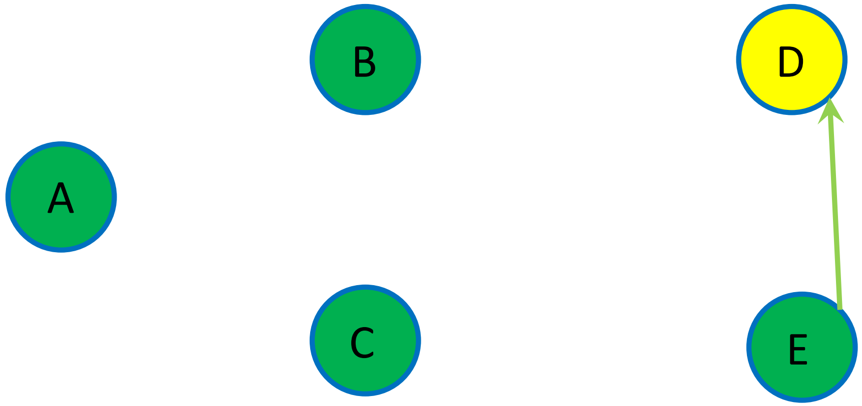


Sorted:

A	**B**	**C**

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
    Add v to sorted
    Remove the outgoing edges of v
```

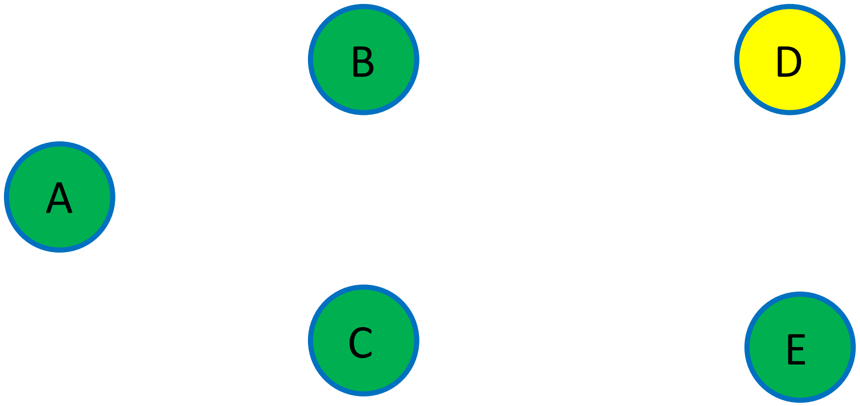


Sorted:

A	**B**	**C**	**E**

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

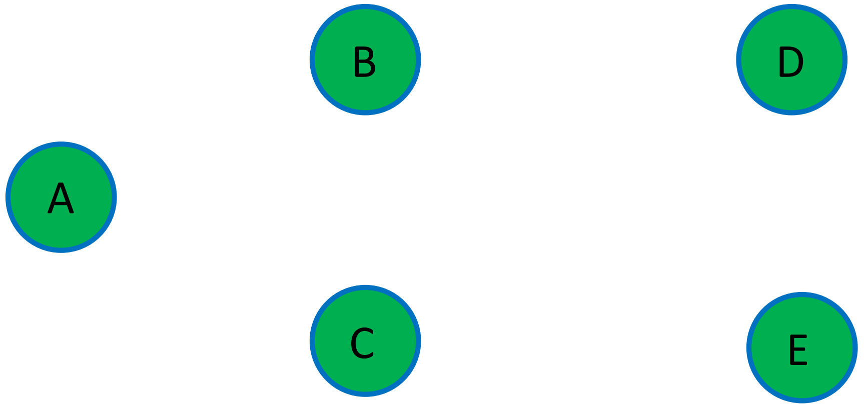


Sorted:

A	**B**	**C**	**E**

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```



Sorted:

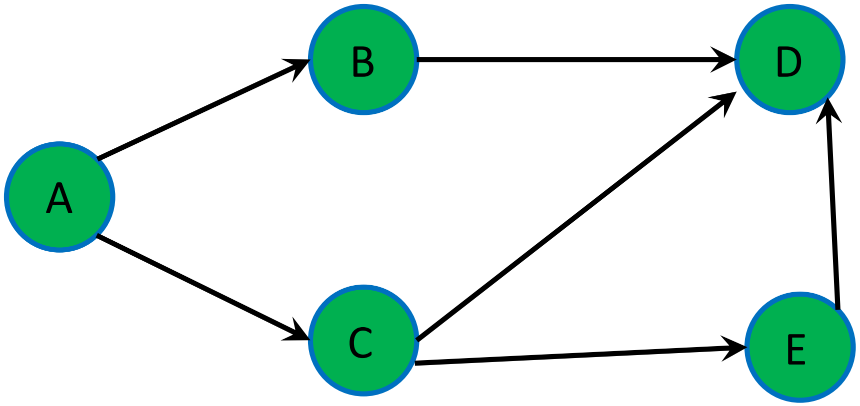
A	**B**	**C**	**E**	**D**

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

How can we efficiently track the number of incoming edges?

Quiz time!
https://flux.qa - RFIBMB



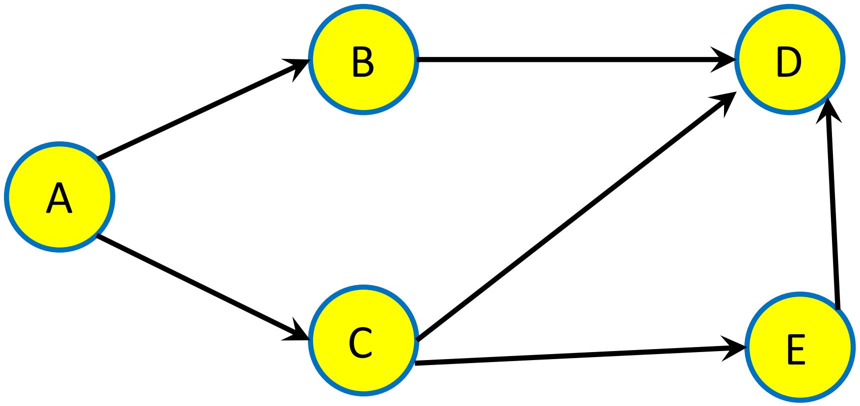
Sorted:

A	**B**	**C**	**E**	**D**

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

How can we efficiently track the number of incoming edges?



Order:

A	B	C	D	E
0	1	1	3	1

Sorted:

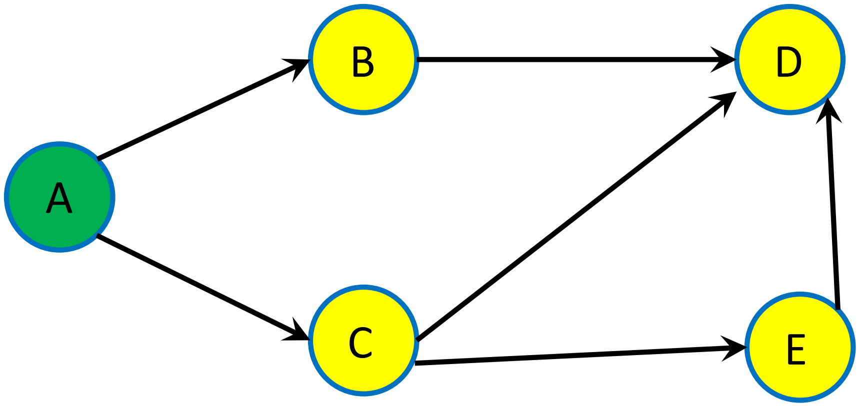
A	B	C	E	D

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

How can we efficiently track the number of incoming edges?

When we remove A, update it's children by -1



Order:

A	B	C	D	E
0	1	1	3	1

Sorted:

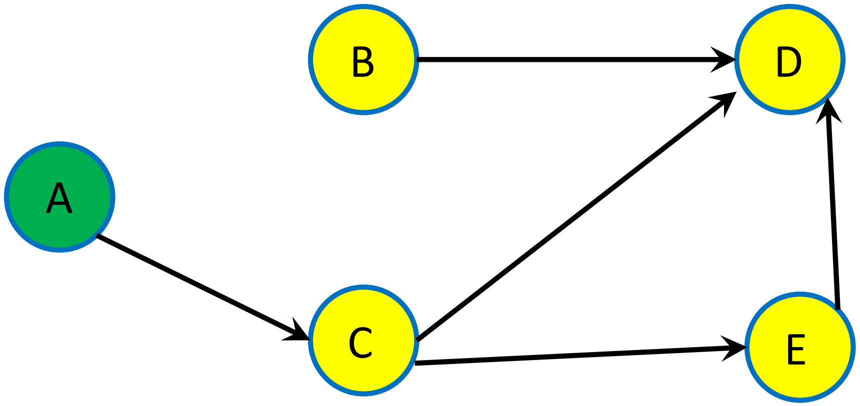
A	B	C	E	D

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

How can we efficiently track the number of incoming edges?

When we remove A, update it's children by -1



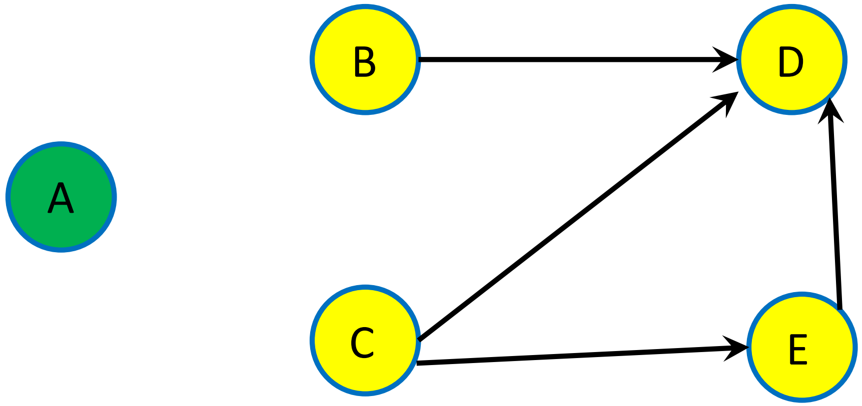
	A	B	C	D	E
Order:	0	0	1	3	1
Sorted:	A	B	C	E	D

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

How can we efficiently track the number of incoming edges?

When we remove A, update it's children by -1



Order:

A	B	C	D	E
0	0	0	3	1

Sorted:

A	B	C	E	D

Kahn's Algorithm: High level idea

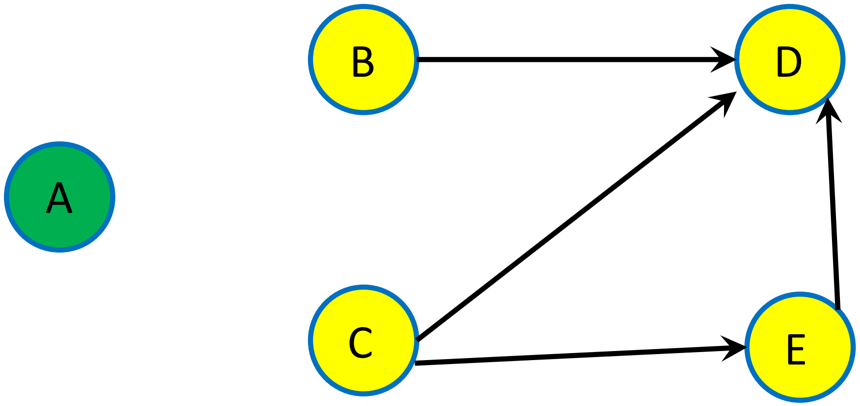
```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

How can we efficiently track the number of incoming edges?

When we remove A, update it's children by -1

Complexity of such an approach?

Quiz time!
https://flux.qa - RFIBMB



Order:

A	B	C	D	E
0	0	0	3	1

Sorted:

A	B	C	E	D

Kahn's Algorithm: High level idea

```
For each vertex v that does not have ANY incoming edge
    Add v to sorted
    Remove the outgoing edges of v
```

- Loop occurs V times
  - Finding a vertex with 0 in "order" takes O(V)
  - Adding to sorted is O(1)
  - Removing an outgoing edge costs O(1) (using order)
  - This happens E times over the life of the algorithm
- So this algorithm would be **$O(V*V + E) = O(V^2)$**
- **Can we do better?**

Quiz time!
https://flux.qa - RFIBMB

Sorted:

A	**B**	**C**	**E**	**D**

Order:

A	**B**	**C**	**D**	**E**
0	0	0	3	1

Kahn's Algorithm: Detailed pseudocode

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    in_degree = array of size V, initialised with the number of incoming edges to each vertex
    ready = queue of all vertices with no incoming edges
    while ready is not empty do
        u = ready.pop()
        order.append(u)
        for each edge (u, v) adjacent to u do
            in_degree[v] -= 1
            if in_degree[v] = 0 then
                ready.push(v)
    return order
```

- Loop occurs V times
  - Pop is O(1), append is O(1)
  - Inner loop runs E times in total
  - Removing an edge is O(1)
  - If is O(1), push is O(1)
- So this algorithm would be **O(V + E)**

Overview

- Topological Sort
  - Kahn’s Algorithm
  - Depth First Search
- Design Principles (FIT2004 Summary)
- Final Exam etc.
- Review of all lecture material

Depth First Search (DFS)

Below is the DFS algorithm we saw in week 8

- function DFS(v):
  - Mark v as Visited
  - For each adjacent edge (v,u)
    - If u is not visited
      - DFS(u)

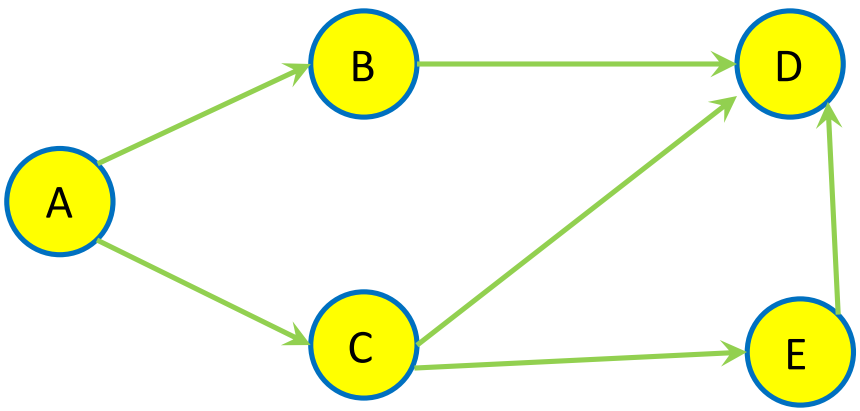
Assume we call DFS(A), which of the following is NOT a possible order in which vertices are marked visited.

A, B, D, C, E

A, C, E, D, B

A, C, D, E, B

A, C, E, B, D



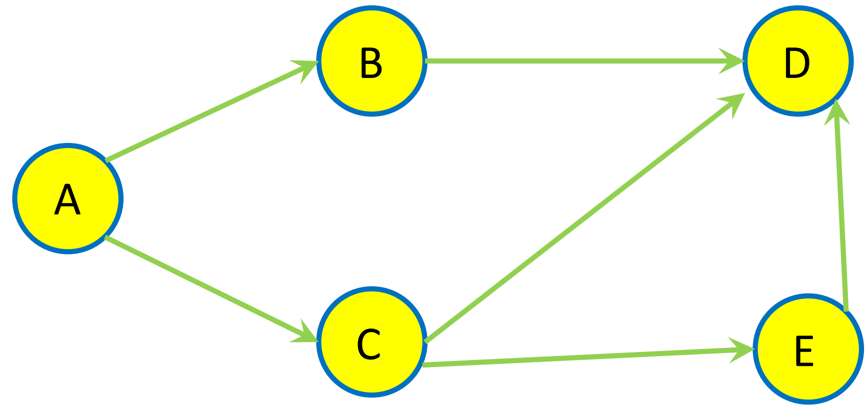
DFS for Topological Sort

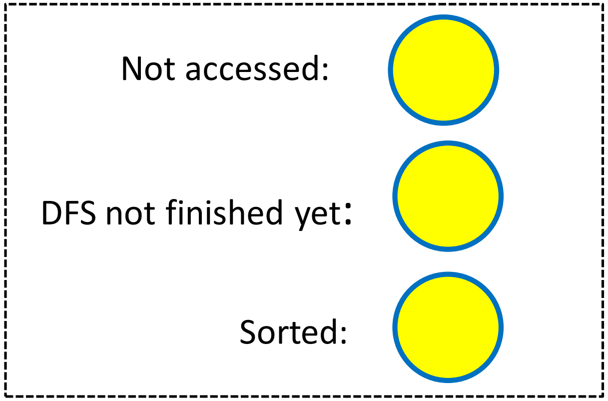
Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```

*// Add to order **after** visiting descendants*





Sorted:

A	C	E	B	D

Overview

- Topological Sort
  - Kahn’s Algorithm
  - Depth First Search
- Design Principles (FIT2004 Summary)
- Final Exam etc.
- Review of all lecture material

Design Principles (Summing up FIT2004)

Here are some broad strategies to (try to) solve algorithmic problems:

- Look out for good invariants to exploit
- Attempt to balance your work as much as possible
- Do not repeat work (so, store and re-use!)
- Use appropriate data structures
- Try well-known problem solving strategies
- Sometimes greed is good!
- These are general guidelines. As always, there are many exceptions

Look out for good invariants to exploit

- Here are some algorithms we considered in the unit that do precisely this!
- Binary Search (Refer Week 2 lecture)
- Sorting (Refer Lectures from Weeks 2 and 3)
- Shortest Paths and Connectivity
  - Dijkstra's algorithm (Refer Week 8 Lectures)
  - Floyd-Warshall algorithm (Refer Week 9 lectures)
- Minimum Spanning Tree Algorithms (Refer Week 10 lectures)

Balance your work as much as possible

- For problems that allow division of labour (eg. Divide and Conquer)
- Try to divide work equally as much as possible
- Merge sort achieves this
  - O(N log N)-time always!
- Quick sort does not necessarily achieve this – depends on the choice of the pivot (Refer week 3)
  - Good pivots give O(N log N)-time
  - Bad pivots give $O(N^2)$-time

Choose Data Structures with care

- Certain data representations are more efficient than others for a given problem
- Priority Queue in Dijkstra's algorithm (Refer Week 8)
- Union-Find data structure in Kruskal's algorithm (refer Week 9)
- Efficient Search and retrieval data structures of various kinds (Refer Weeks 5,6,7 lectures)

Don't repeat work

- Do not compute anything more than once (if there is room to store it for reuse)
- Underpins Dynamic Programming strategy
  - Edit Distance (Refer Week 4 Lecture)
  - Knapsack Problem (Refer Week 4 Lecture)

Try well known problem solving strategies

- Divide and Conquer (Refer Weeks 3, 4 lectures)
- Dynamic Programming (Refer Weeks 4, 8, 9 lectures)

Sometimes greed is good

- A greedy strategy is to make a “local” choice based on current information
- Sometimes gives optimal solution, e.g.
  - Dijkstra’s single source shortest paths algorithm (Refer Week 8 Lectures)
  - Minimim Spanning Tree Algorithms – Prim’s and Kruskal’s (Refer Week 10 lectures) minimum spanning tree algorithm.
- Greedy is sometimes a good heuristic!
  - Sometimes gives a “good” solution to a (combinatorial) problem even if not guaranteed optimal

Overview

- Topological Sort
  - Kahn's Algorithm
  - Depth First Search
- Design Principles (FIT2004 Summary)
- Final Exam etc.
- Review of all lecture material

Final Exam

- Time allowed: 2 hours + 10 minutes reading time
- Total Marks: 60
- Exam is open book
  - If a question asks you to **describe an algorithm**, you can write your idea in plain English
  - If a question asks you to **write pseudocode**, you **must** write your idea in a more structured way (like the ones in lecture slides or even Python code)
  - If a question asks for complexity, it means **big-O**, the **tightest bound** and the **worst case** unless otherwise specified
- Hurdles:
  - At least 16 out of 40 marks in in-semester assessments (assignment + mid-semester test + lecture/tutorial participation)
  - At least 24 out of 60 marks in the final exam
  - At least 50 marks overall
- Do not miss final exam even if you fail in-semester hurdle.
  - It affects your WAM

Non-Examinable Content

- Additional material in lecture notes is NOT examinable
  - In other words, anything NOT covered in lectures, tutorials, labs is NOT EXAMINABLE!
- Advanced questions in tutorials are NOT examinable
- Anything marked “not examinable” is not examinable

Consultations for Final Exam

- Please come to the consultations prepared
  - Do not ask questions like "Can you please explain Dynamic Programming from scratch?".

- Don't try getting hints about the questions on final exam!
  - E.g., Is Kruskal's algorithm going to be on the exam?

- Don't ask how hard the exam is
  - It is, **on average**, easier than questions in the tutorial
  - It is designed to test your knowledge of the unit
  - It is designed to allow everyone a chance to get some marks, but not allow everyone to get full marks (i.e. a spread of difficulty)

Suggestions for preparation

- **Understand** how each algorithm works
- Practice writing pseudocode for each algorithm
- **Understand** its complexity analysis
- Don’t confuse algorithms: Bellman-Ford vs Floyd-Warshall vs Ford-Fulkerson
  - Despite warning, every semester, students mix up algorithms losing all marks for the question
- Go over the early material and the week 11/12 material
- Go over the material which was not covered by assignments

Overview

- Topological Sort
  - Kahn's Algorithm
  - Depth First Search
- Design Principles (FIT2004 Summary)
- Final Exam etc.
- Review of all lecture material

Lecture 1

- correctness proof
- complexity recap
- recurrence relations
- proof by induction

Lecture 2

- intro to space complexity
- comparison costs
- stability
- selection sort analysis
- insertion sort analysis
- proof of lower bound for comparison based sorrts
- count sort
- stable count sort
- radix sort
- recursive complexity (space and time)
- output sensitive time complexity

Lecture 3

- quicksort review
- partition out of place/in place/stable
- complexity analysis of quicksort (best/worst/average)
- kth order stats
- quickselect
- quickselect complexity
- median of medians (not examinable)

Lecture 4

- intro to DP
- Fibonacci
- coin change
- unbounded knapsack
- 0/1 knapsack
- edit distance
- constructing optimal solutions (finding coins)
- backtracking vs decision array

Lecture 5

- Hash tables
- direct addressing
- hashing/collision
- Birthday paradox
- collisions always occur
- ideal properties of a hash
- open hashing (chaining)
- closed hashing
- linear probing with deletion (lazy)
- primary clustering
- quadratic probing
- secondary clustering
- double hashing
- cuckoo hashing
- BST
- search
- insert
- delete
- worst case shape
- avl tree
- balance factor
- rebalancing
- complexity analysis of AVL

Lecture 6

- trie
- construction
- edge-node labels
- search
- nodes being arrays
- pros and cons
- properties
- suffix trie
- substring search
- lookup
- counting occurrences of substring using tree
- longest repeated susbstring using tree
- suffix tree
- suffix array
- querying SA
- O(n) space SA
- longest repeated susbstring
- Construction of SA
- prefix doubling

Lecture 7

- BWT
- Last-First property
- justification of symbol clustering
- k-mers BWT inversion
- LF-mapping
- efficient inversion
- practice
- substring search + complexity

Lecture 8

- graph recap
- graph definition
- represntation (adj matrix, adj list)
- BFS
- DFS
- some applications of BFS and DFS
- BFS for distances in unit weight
- dijkstras algorithm
- updating the heap vs double inserting
- proof of correctness
- stoppping early with single target
- recovering path

Lecture 9

- dijkstras with negative weights does not work
- negative cycles make shortest path meaningless
- bellman ford
- correctness
- unreachable cycles
- all pairs shortest paths
- floyd warshall
- correctness
- transitive closure

Lecture 10

- spanning tree
- minimum spanning tree
- general strategy of adding safe edges
- prims algorithm
- correctness
- kruskals algorithm
- correctness
- union find

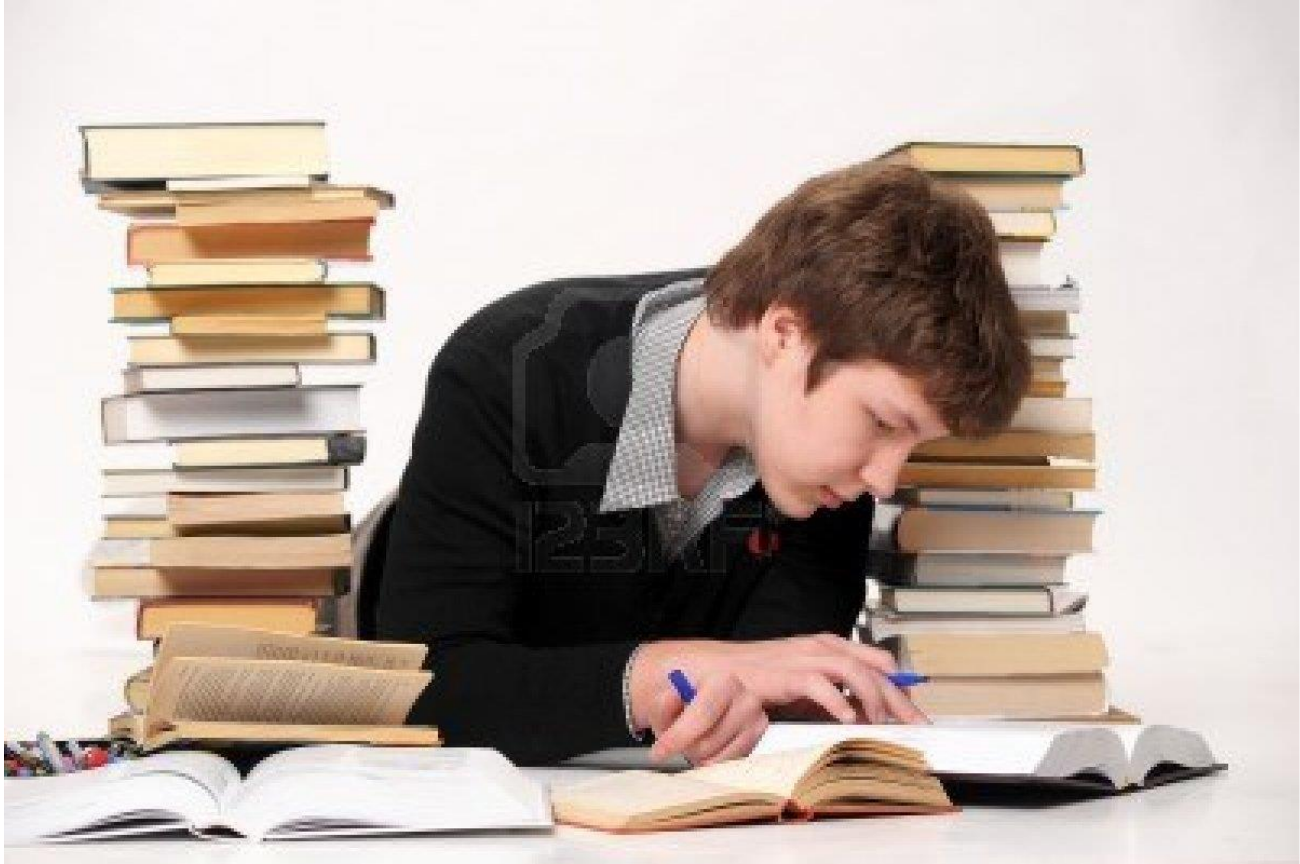
Lecture 11

- Flow problem
- Flow network properties
- For Fulkerson algorithm
- Augmenting paths
- Complexity analysis
- Cuts
- Max-Flow / Min-Cut
- Proof of correctness

Lecture 12

- Kahn's algorithm
- Complexity
- DFS for topological sort
- Summary of unit
- Overview of exam
- Review of all lecture material

Coming Up Next



SWOT VAC