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.

Overview

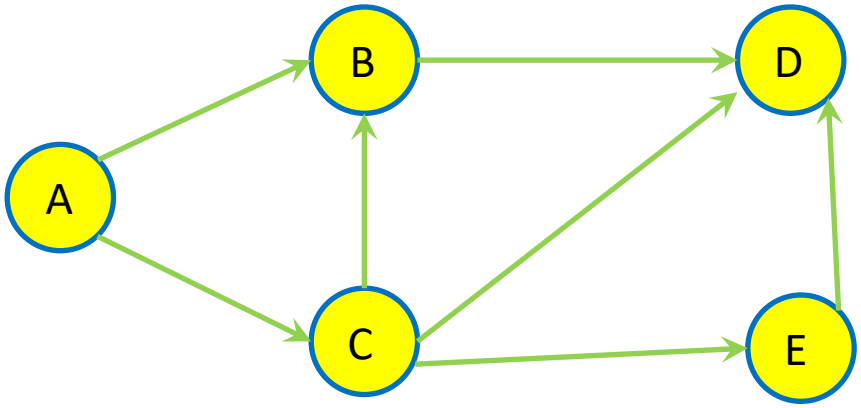
- Topological Sort
  - Kahn's Algorithm
  - Depth First Search
- Design Principles (FIT2004 Summary)
- Final Exam etc.
- Review of all lecture material

Directed Acyclic Graph (DAG)

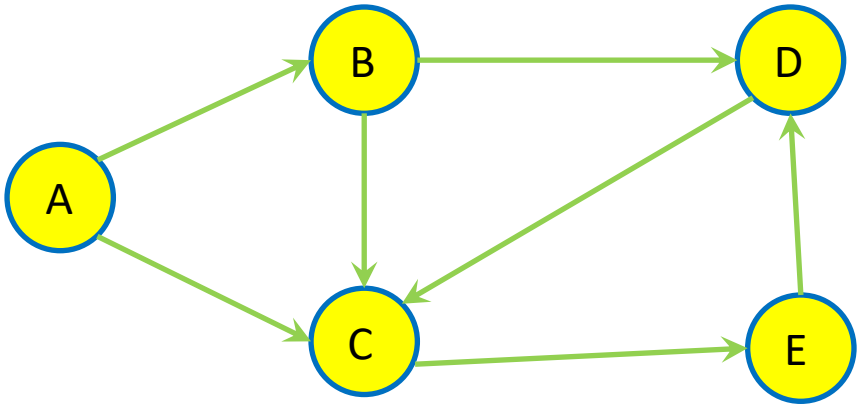
A Directed Acyclic Graph (DAG) is

- Directed
- Acyclic – has no cycles
- Graph

Which of the two graphs is a DAG?



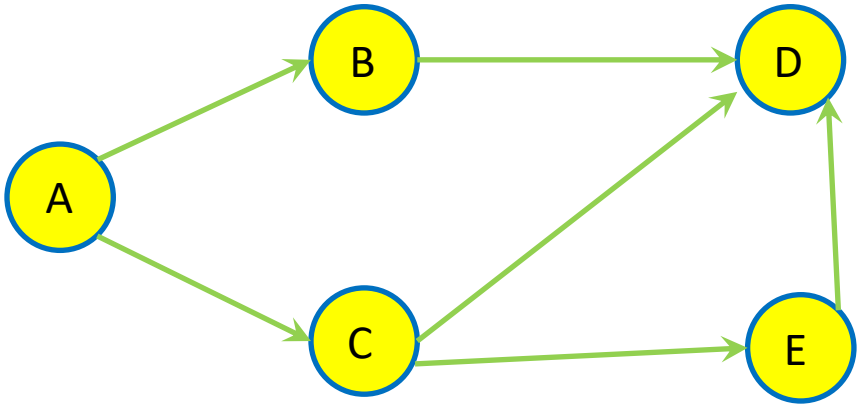
Graph 1

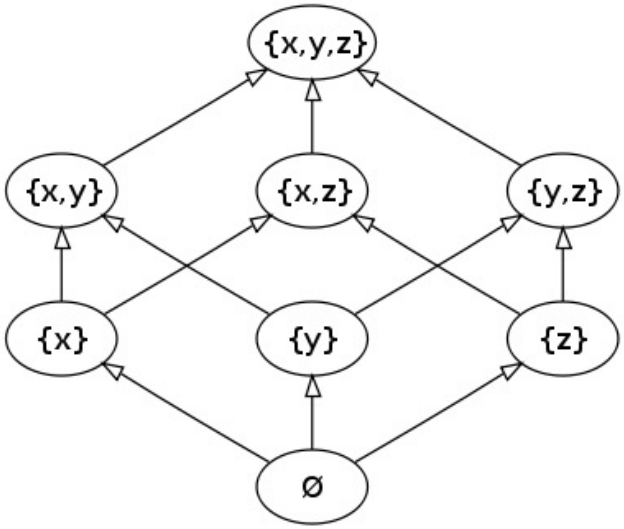


Graph 2

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

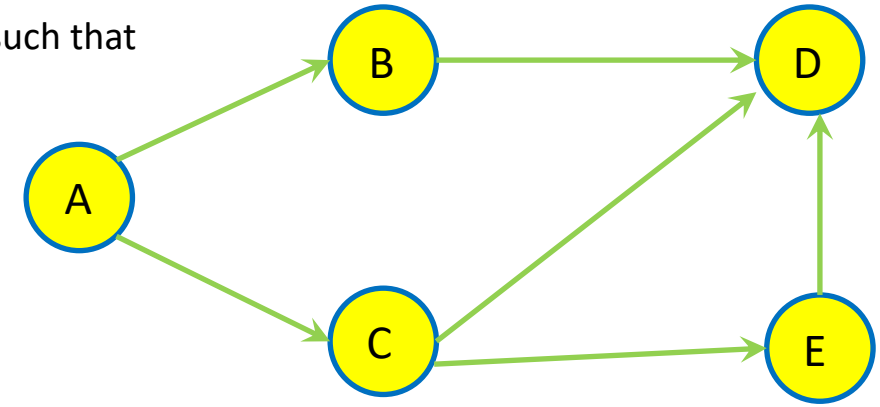
Topological Sort of a DAG

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.

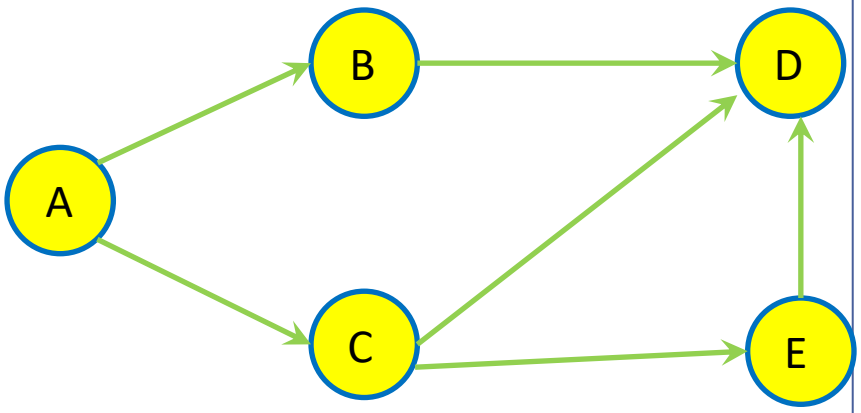
Which of these is NOT a valid topological ordering of the DAG

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

Quiz time!
https://flux.qa/YTJMAZ

- How to do topological sort?

- For the example graph, identify the nodes that you can safely give a number.
  Give the number and the reason.



Topological Sort of a DAG

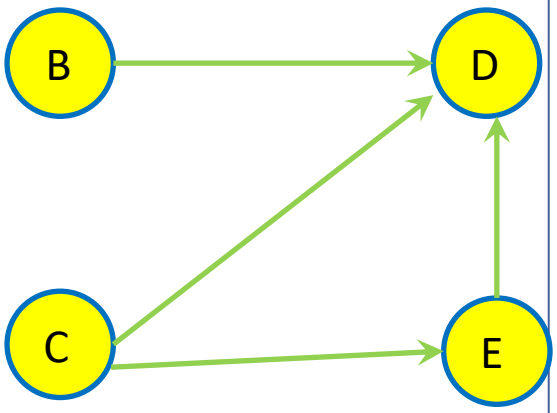
- A DAG can have many valid topological sorts, e.g., let u and v be two incomparable vertices, u may appear before or after v.

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

We have modified the graph by removing A

- Do we have to take B or C first?



Topological Sort of a DAG

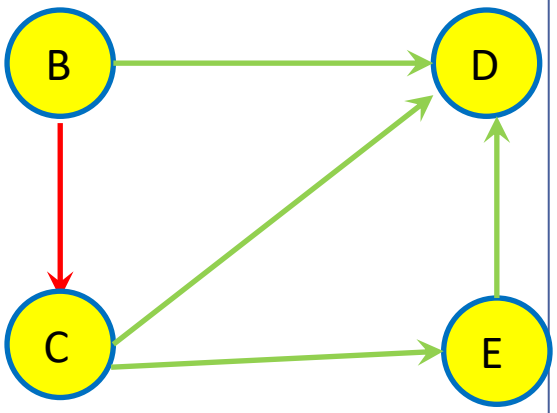
The only thing that could force us to take B before C would be an edge from B to C (or a longer path leading from B to C) – here added in red

In both cases, C would have to have an incoming edge!

It is safe to take nodes with no incoming edges.

And if we have multiple nodes with *no incoming edges,* we can break ties arbitrarily

This is **Khan's Algorithm (1962).**



Topological Sort of a DAG

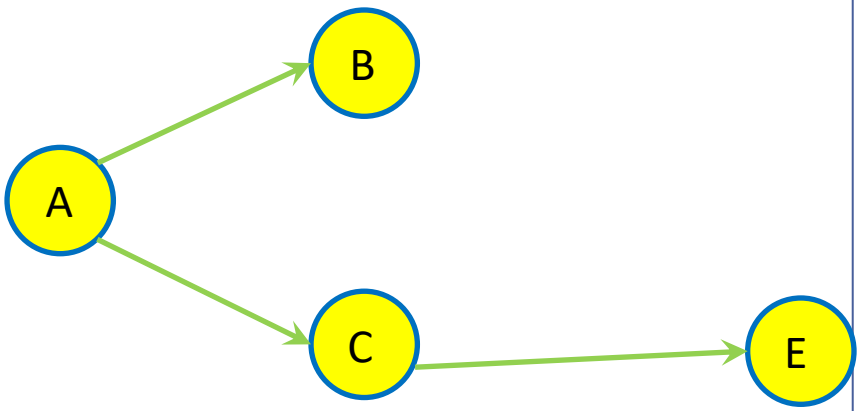
Note that this can also be done in the inverse order.

We have now modified the graph by removing D instead.

Taking D was safe. As it has **no outgoing edges**, this can't have any consequences for any nodes further down the track. There is no path onward from D.

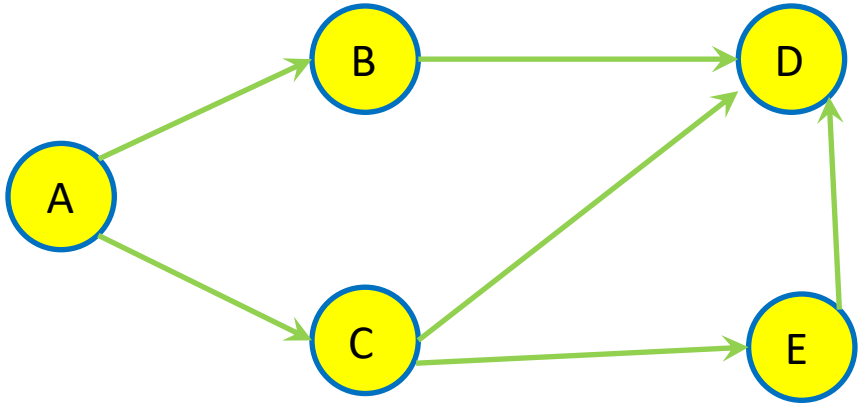
Number edges in decreasing order by removing a node with no outgoing edges.

Break ties arbitrarily.



Kahn's Algorithm: High level idea

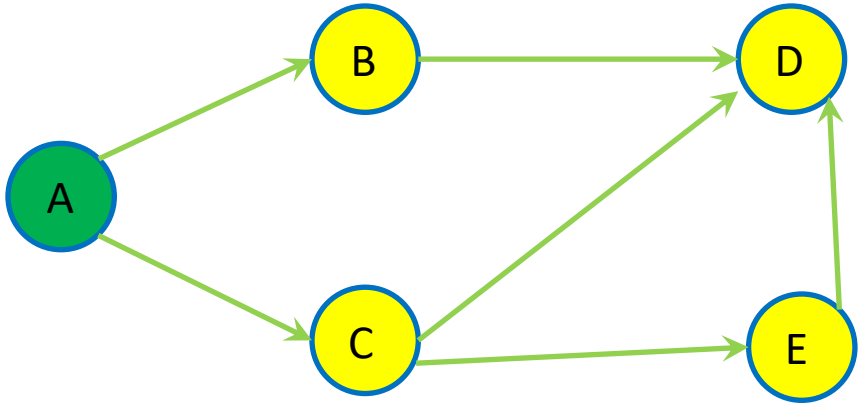
```
For each vertex v that does not have ANY incoming edge
    Add v to sorted
    Remove the outgoing edges of v
```



Sorted:

Kahn's Algorithm: High level idea

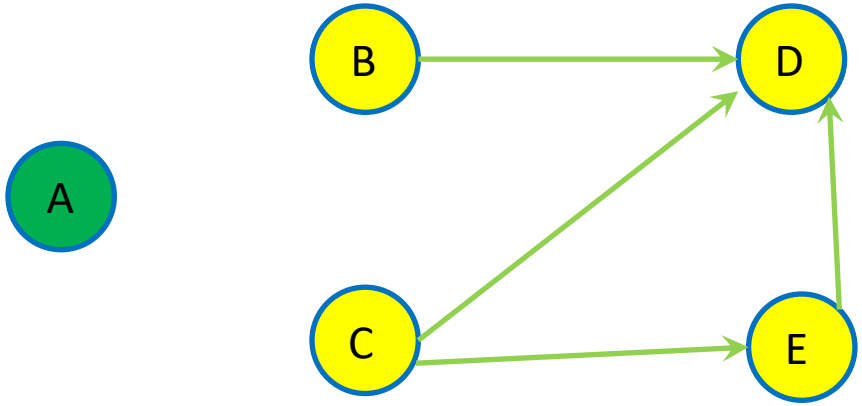
```
For each vertex v that does not have ANY incoming edge
    Add v to sorted
    Remove the outgoing edges of v
```



Sorted: **A**

Kahn's Algorithm: High level idea

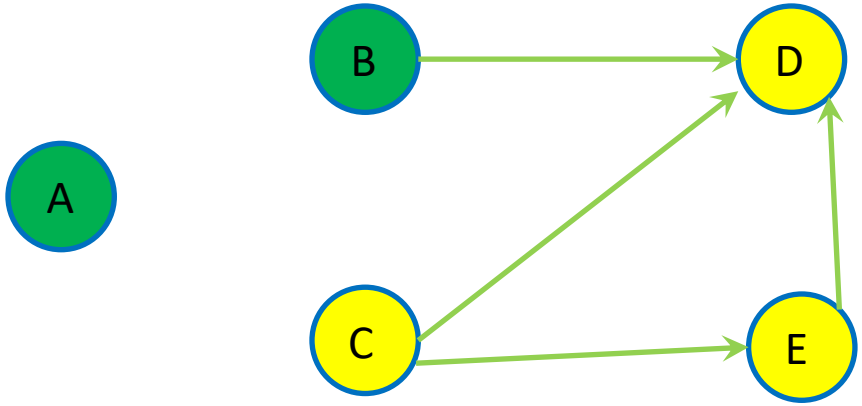
```
For each vertex v that does not have ANY incoming edge
    Add v to sorted
    Remove the outgoing edges of v
```



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
```

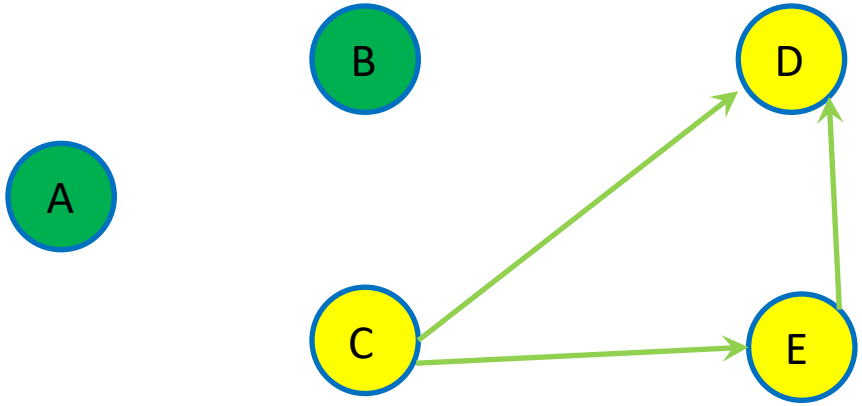


Sorted:

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
```

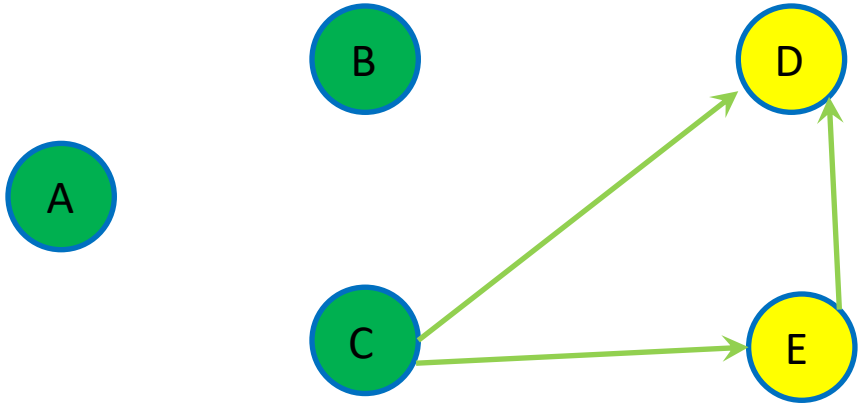


Sorted:

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
```

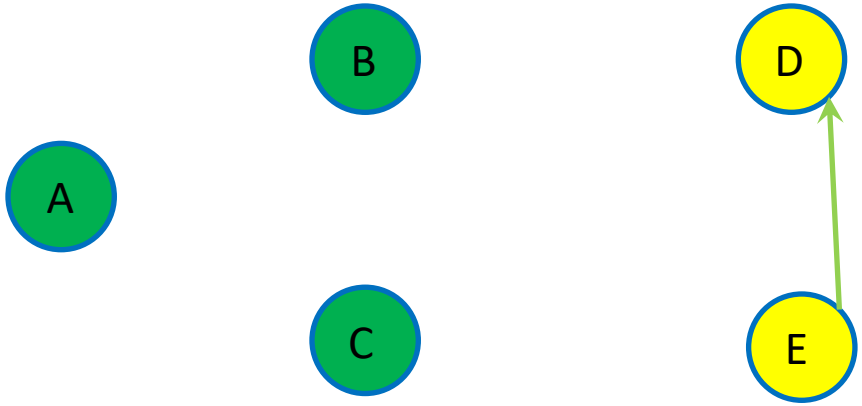


Sorted:

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
```

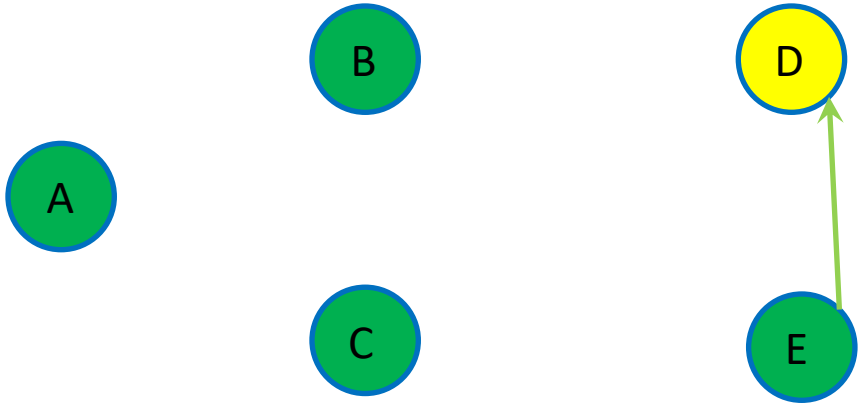


Sorted:

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
```

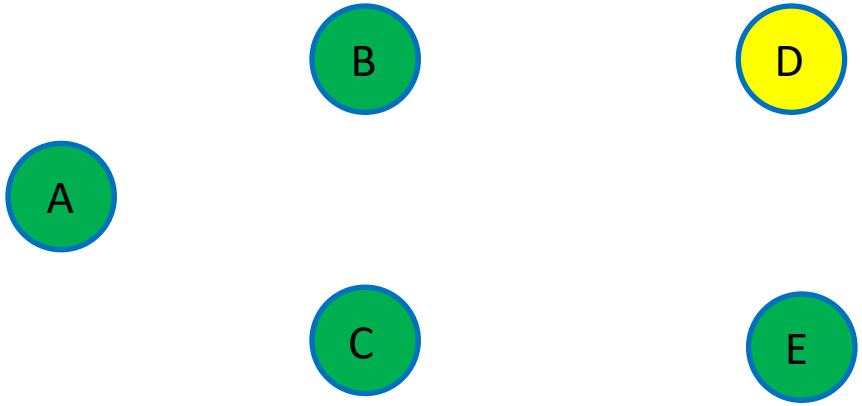


Sorted:

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
```

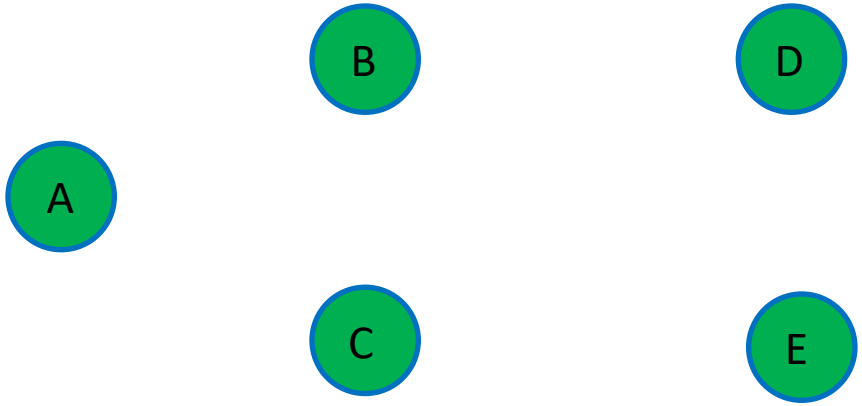


Sorted:

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:

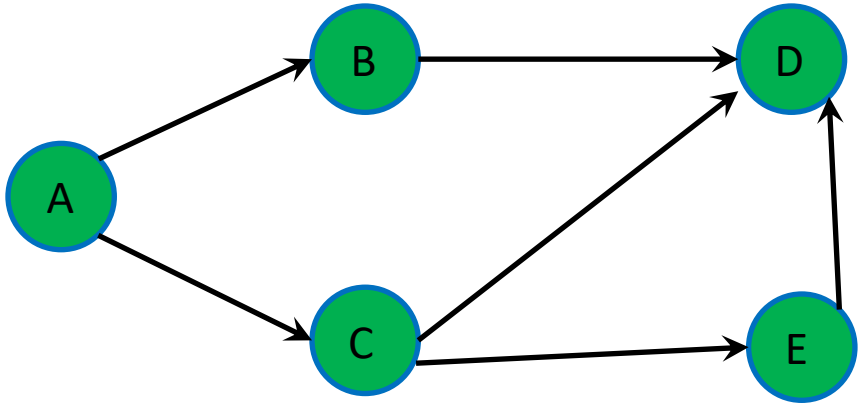
A	B	C	E	D

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/YTJMAZ



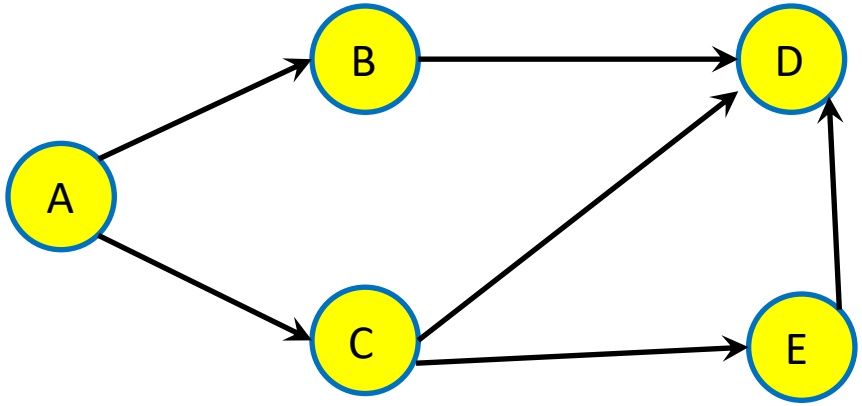
Sorted:

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:

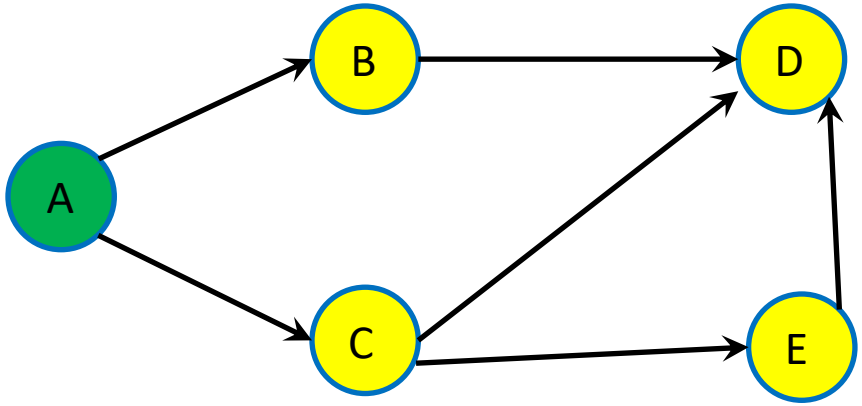
A	B	C	E	D

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:

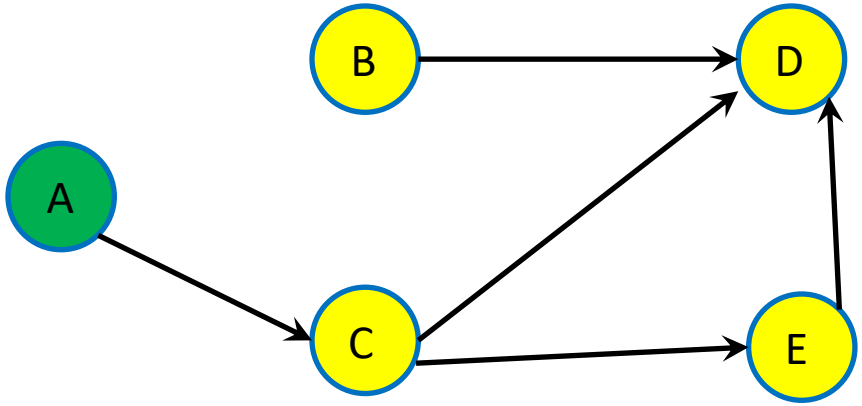
A	B	C	E	D

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



Order:

A	B	C	D	E
0	0	1	3	1

Sorted:

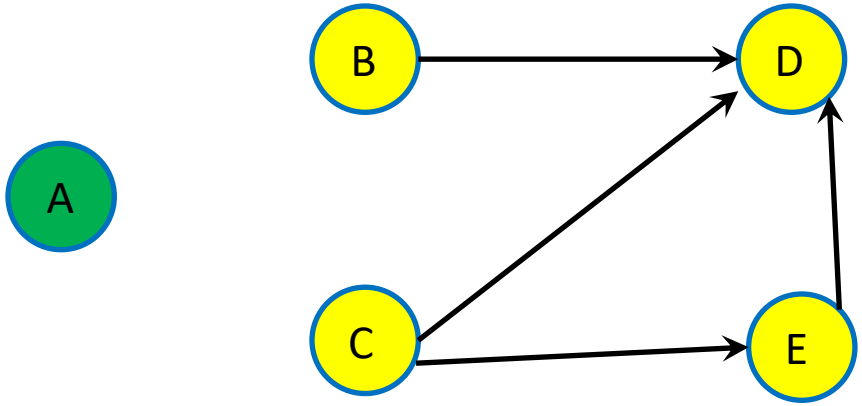
A	B	C	E	D

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



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

Sorted:	A	B	C	E	D

Kahn's Algorithm: High level idea

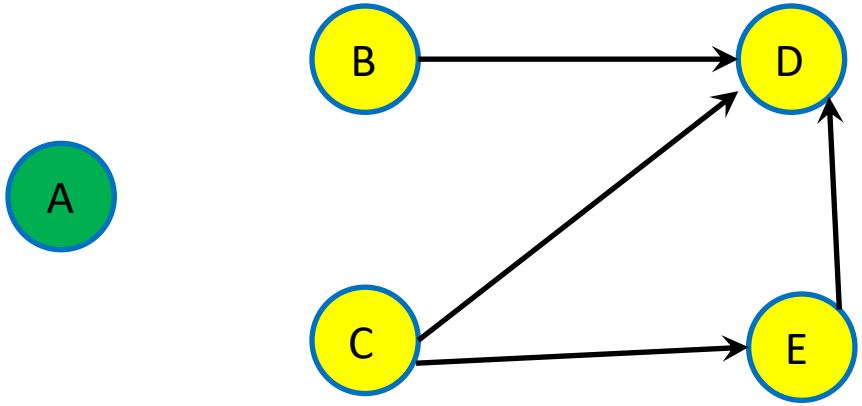
```
For each vertex v that does not have ANY incoming edge
   Add v to sorted
   Remove the outgoing edges of v
```

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/YTJMAZ



	A	B	C	D	E
Order:	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); this happens V times -> O(V*V)
  - Adding to sorted is O(1); this happens V times -> O(V)
  - Removing outgoing edges costs O(deg V) (using order); this happens for each V: $O(\Sigma_v: \deg v)=O(E)$
  - This happens V times over the life of the algorithm – for every vertex V
- So this algorithm would be **$O(V*V + E) = O(V^2)$**
- **Can we do better?**

Quiz time!
https://flux.qa/YTJMAZ

Sorted:

A	B	C	E	D

Order:

A	B	C	D	E
0	0	0	3	1

Kahn's Algorithm: Detailed pseudocode

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

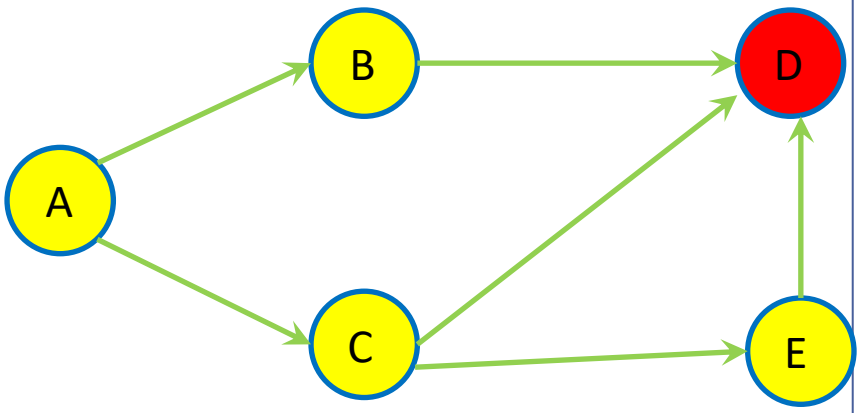
- Initialising in_degree is O(V+E)
- Loop occurs V times
  - Pop is O(1), append is O(1)
  - Inner loop runs E times in total (since no node *u* is visited twice)
  - Removing an edge is O(1)
  - Push is O(1)
- So this algorithm has time complexity **O(V + E)**

Reverse Topological Sort of a DAG

Recall: we can also process the nodes in reverse order.

by starting from nodes with no outgoing edges.

But how do we efficiently find nodes with no outgoing edges?



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)

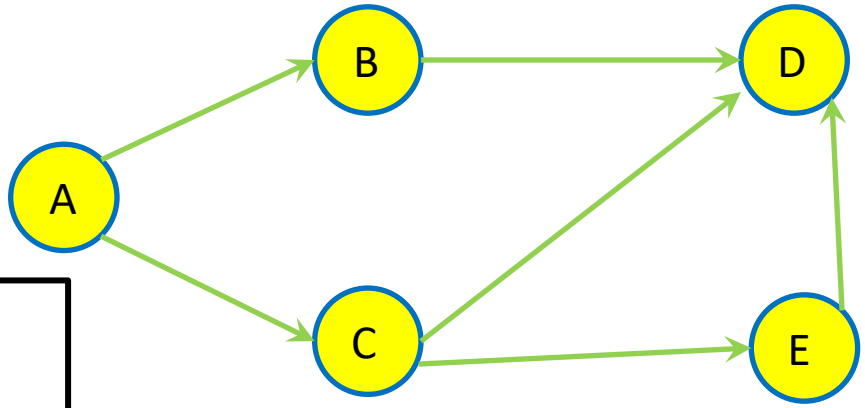
Assume we call DFS(A), which of the following is NOT a possible order in which vertices are marked visited.

A, B, D, C, E

A, C, E, D, B

A, C, D, E, B

A, C, E, B, D



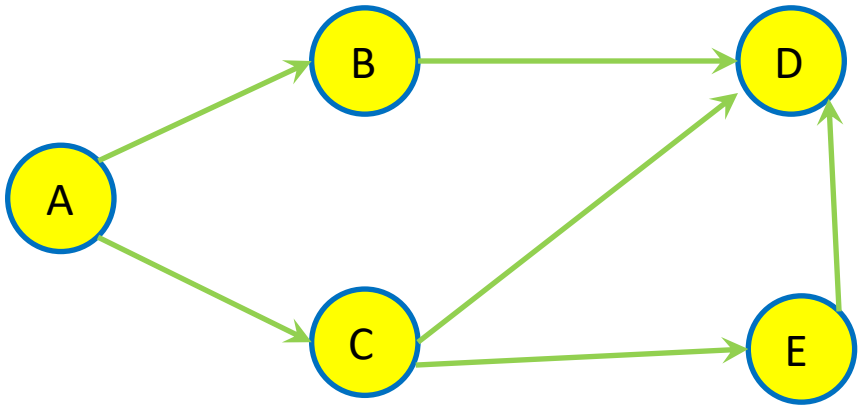
Quiz time!
https://flux.qa/YTJMAZ

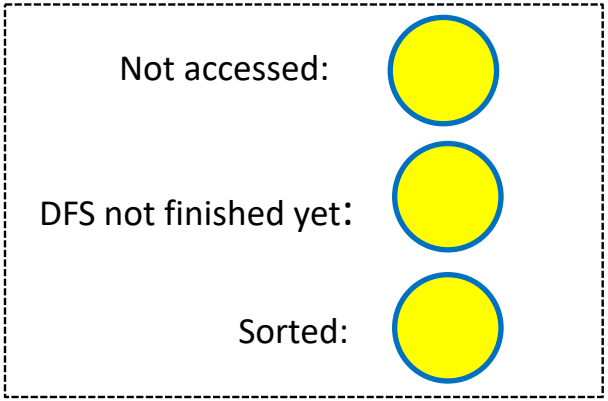
DFS for Topological Sort

Algorithm 72 Topological sorting using DFS

```
1:  function TOPOLOGICAL_SORT(G = (V, E))
2:      order = empty array
3:      visited[1..n] = false
4:      for each vertex v = 1 to n do
5:          if not visited[v] then
6:              DFS(v)
7:      return reverse(order)
8:
9:  function DFS(u)
10:     visited[u] = true
11:     for each vertex v adjacent to u do
12:         if not visited[v] then
13:             DFS(v)
14:     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:

- Try well-known problem-solving strategies
- When it works greedy is usually good/fast! (it rarely does)
- Use appropriate data structures
- Do not repeat work (factor it out, preprocess, store intermediate results)
- Attempt to balance the work as much as possible
- Look for good invariants to exploit

Design Principles (Summing up FIT2004)

- These are just general guidelines. Use your intuition but then prove that it works!!
- Generally, what often helps is to start with a naive solution, then trace carefully and analyse where you waste work
- i.e. incremental design

Try well-known problem solving strategies

- Greedy (but be suspicious)
- Divide and Conquer (Refer Weeks 3, 4 lectures)
- Dynamic Programming (Refer Weeks 4, 8, 9 lectures)
- Reduce the problem to a graph algorithm (MST, Dijkstra's, max flow, BFS, DFS)

When it works, greedy is usually good

- A greedy strategy is to make a "local" best choice based on current information and never look back
- Sometimes gives optimal solution, e.g.
  - Dijkstra's single source shortest paths algorithm (Refer Week 8 Lectures)
  - Minimum Spanning Tree Algorithms – Prim's and Kruskal's (Refer Week 10 lectures) minimum spanning tree algorithm.
- When it works, it is usually the key to *fast* algorithms
- Greedy is sometimes a good heuristic!
  - Sometimes gives a "good" solution to a (combinatorial) problem even if not guaranteed optimal

Use appropriate data structures

- 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)
- In graph algorithms, the base graph structure often determines the complexity (eg. Adjacency lists versus Matrix)

Don’t repeat work

- Do not compute anything more than once (if there is room to store it for reuse)

- We used this principle just today in Khan’s algorithm
- An extreme example are Tries (this is their whole point)

- Underpins Dynamic Programming strategy
  - Edit Distance (Refer Week 4 Lecture)
  - Knapsack Problem (Refer Week 4 Lecture)

Balance your work as much as possible

- For problems that can be split into independent subproblems (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

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)

Overview

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

Final Exam

- Time allowed: 2 hours
- Total Marks: 50
- Exam is open book (restricted materials allowed, please see edstem page linked below)
  - 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 45% (22.5 marks) in in-semester assessments (assignment + mid-semester test + lecture/tutorial participation)
  - At least 45% (22.5 marks) in the final exam
  - At least 50 marks overall
  - https://edstem.org/au/courses/6067/discussion/622984
- 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 make it hard to get full marks (i.e. a spread of difficulty); don’t be shoked if you can’t solve everything!

Suggestions for preparation

- **Understand**
  - how each algorithm operates
  - why it works!
  - its complexity analysis
- **Know**
  - the algorithms' common names
    eg Bellman-Ford vs Floyd-Warshall vs Ford-Fulkerson
- Practice writing pseudocode for each algorithm
- 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.
- Summary of Contents

FIT 2004 – Redux

- Note that the following is KNOWLEDGE ONLY!
- The important stuff is the WHY ie UNDERSTANDING
- And the underlying methods
  - determining complexity, master theorem, telescoping
  - Selecting appropriate data structures
  - applying invariants
  - etc…
- The other important thing is the mindset
  - not so much for the exam but for your future career!

FIT 2004 – Redux

Attitudes

- don't guess - show that it is correct
- think about complexity (both time and space!)
- think about appropriate data structures & algorithm patterns
- build on existing knowledge (ie do your homework!)
  - don't reinvent the wheel
  - most common problems have been solved (or at least investigated)
  - Learn to map your problems to known problems
- Think before you code (ie design the algorithms first)

FIT 2004 – Redux - Complexity

```
reverse(l)
    begin
        if (length(l)<2) return l
        else begin
            l1 := reverse(first_half(l));
            l2 := reverse(second_half(l));
            return append( l2, l1 );
        end
    end.
```

FIT 2004 – Redux - Complexity

Recall: the **Runtime Equation for Divide-and-Conquer** (provided we split into equal parts) is

$$T(n) = aT(n/b) + cn^k$$

This **Master Theorem** tell us this has three different solutions

$$T(n) = \begin{cases} O(n^{\log_b a}) & if \quad a > b^k \\ O(n^k \log n) & if \quad a = b^k \\ O(n^k) & if \quad a < b^k \end{cases}$$

It can be solved by backward substitution.

FIT 2004 – Redux - Complexity

1. Fix your assumptions:
    1. *length is O(1), append is O(1), first/second_half is O(n)*
2. Read off the recurrence relation
    1. T(1) = 1
    2. T(n) = 2*T(n/2) + n
3. Match parameters for Master Theorem
    1. a=2; b=2; c=1; k=1
4. Select case of Master Theorem and derive solution
    1. Second case ($a=b^k$)
    2. T(n) = O(n log n)

5. Alternative assumption:
    1. *first/second_half, length, append, are all O(1)*
6. Parameters:
    1. a=b=2; c=1; k=0
7. Solution
    1. First case ($a>b^k$)
    2. T(n) = O(n)

FIT 2004 – REDUX (Data Types)

Data Type	Data Structures		Core Issues
Dictionary, Set, Symbol Table			
Strings			
Queue			
Stack			
Priority Queue			
Graph			
Partition (set of disjoint sets)			

FIT 2004 – REDUX (Data Types)

Data Type	Data Structures		Core Issues
Dictionary, Set, Symbol Table	List		inefficient
	Binary Tree		keeping balanced
	AVL Tree		
Strings	Tries		
	Suffix Tree, Array		
Queue	List		
Stack	List		
Priority Queue	Heap		
Graph	Adjacency List		must be chosen with care, see complexity table in graph lectures
	Adjacency Matrix		
Partition (set of disjoint sets)	Union-Find		Path compression

FIT 2004 – REDUX (Paradigms)

Paradigm	Idea	Example
Greedy	optimal choice at every step, never look back	
Brute-force	enumeration	
Divide-and-Conquer	split into independent subproblems	
Dynamic Programming	split into overlaping subproblems, remove redundant computations by unwinding recursion bottom-up	
Memoisation	simpler alternative to DP	

FIT 2004 – REDUX (Paradigms)

Paradigm	Idea	Example
Greedy	optimal choice at every step, never look back	Dijkstra, Prim, Kruskal
Brute-force	enumeration	Generate-and-Test (e.g. Knapsack). Expensive. Almost NEVER a good idea
Divide-and-Conquer	split into independent subproblems	Merge, Quicksort, [Closest Point]
Dynamic Programming	split into overlaping subproblems, remove redundant computations by unwinding recursion bottom-up	Coin Change, Knapsack, Edit Distance, Bellman Ford
Memoisation	simpler alternative to DP	any DP example

FIT 2004 – REDUX (Paradigms)

Problem	Algorithm	Data Structure	RunTime	Paradigm	Core Idea	Note
Sorting						
Graph Traverals						
Test Connectedness						
Connected Components						
Path Finding						
Cycle Finding						
Spanning Forrest						
Shortest Paths (topological)						
TopSort (DAG only)						
Shortest Path (single source)						
Shortest Path (All pairs)						
Min Spanning Tree (MCST)						

FIT 2004 – REDUX (Paradigms)

Problem	Algorithm	Data Structure	RunTime	Paradigm	Core Idea	Note
Sorting	Quicksort	Array	O(n log n)	D & C		
	MergeSort	Array	O(n log n)	D & C		
	HeapSort	Array	O(n log n)			
	Count Sort	Array	O(n+|u|)			
	Radix Sort	Array	O(n)			
Graph Traverals	DFS, BFS	Adj List	O(V+E)			
Test Connectedness	DFS, BFS					
Connected Components	DFS, BFS					
Path Finding	DFS, BFS					
Cycle Finding	DFS, BFS					
Spanning Forrest	DFS, BFS		O(V+E)			
Shortest Paths (topological)	BFS					
TopSort (DAG only)	DFS	Adj List	O(V+E)			
	Khan	Queue	O(V+E)			
Shortest Path (single source)	Dijkstra	Adj List + Prio Queue	O((V+E) log V) = O(E log V)	Greedy		no negative edges
	Bellman Ford	either	O(VE)		relax each edge n times	no negative cycles
Shortest Path (All pairs)	Floyd	Adj Matrix	O(V^3)	DP		
	V*Dijkstra	Adj List	O(VE log V)	Greedy		
Min Spanning Tree (MCST)	Kruskal	Union-Find Partition+Prio Queue	O((V+E) log V) = O(E log V)	Greedy		
	Prim Jarmik	Adj List + Prio Queue	O((V+E) log V) = O(E log V)	Greedy	Dijkstra but relax from all nodes	

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

- 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
- longest repeated substring
- suffix tree
- suffix array
- querying SA
- O(n) space SA
- longest repeated substring
- 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
- representation (adj matrix, adj list)
- BFS
- DFS
- some applications of BFS and DFS
- BFS for distances in unit weight
- Dijkstra's algorithm
- updating the heap vs double inserting
- proof of correctness
- stopping early with single target
- recovering path

Lecture 9

- Dijkstra's 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
- Prim's algorithm
- correctness
- Kruskal's algorithm
- correctness
- union find

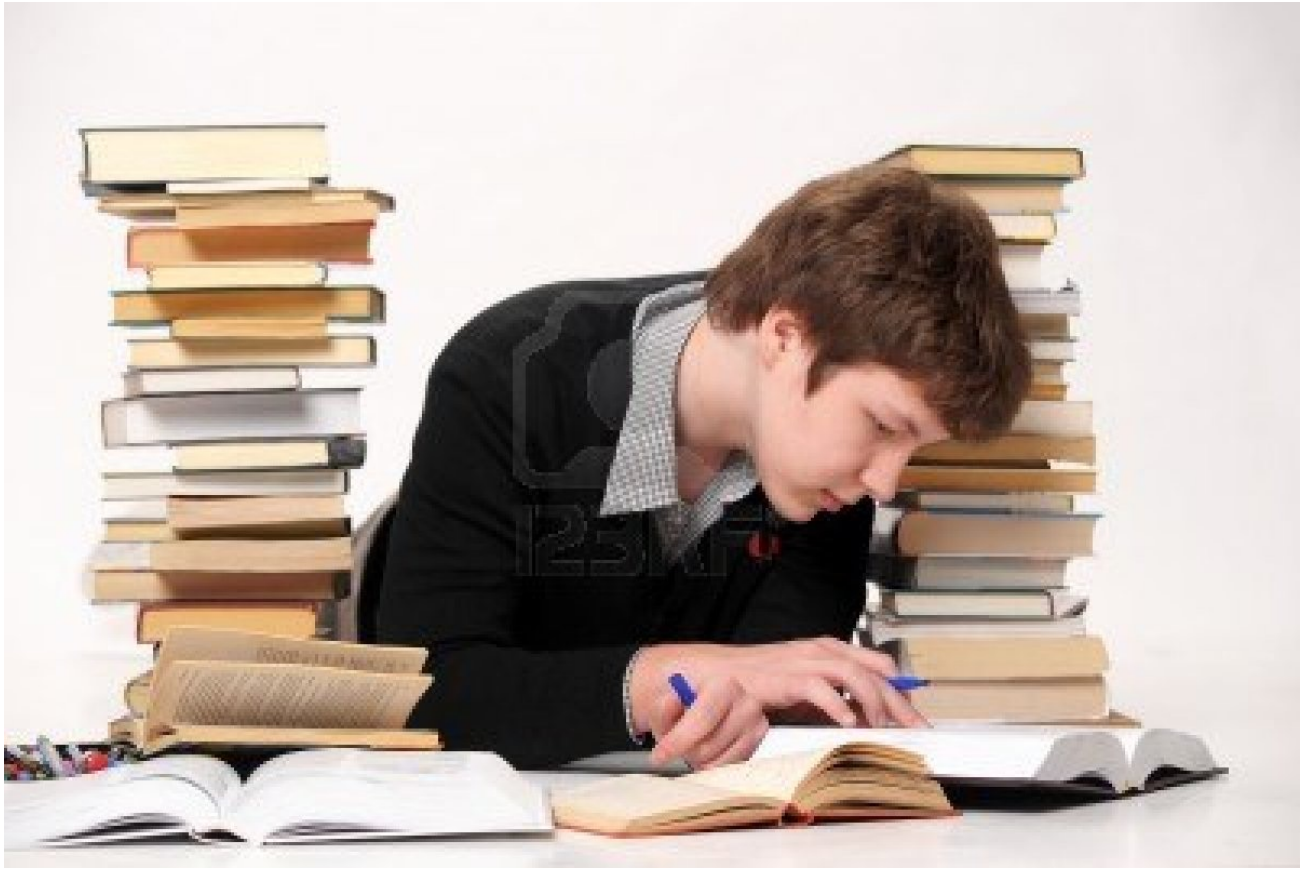
Lecture 11

- Flow problem
- Flow network properties
- Ford-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