Data Structure: Graph

Graph Searching and Topological Sort

Kenneth Lee

Outlines

- Terms & Definitions
- Representations of graphs
- Searching graphs
  - BFS – Breath-first search
  - DFS - Depth-first search
- Applications
- Exercise

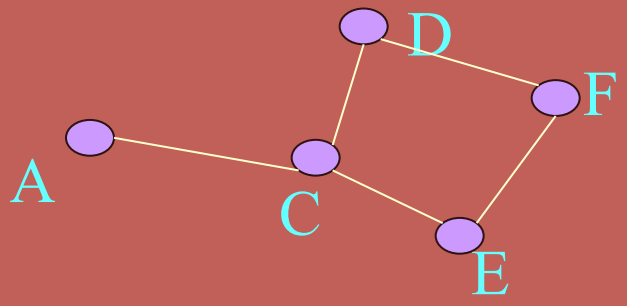
Terms and definitions

- A graph *G* consists of:
  - A non-empty set of vertices: *V*
  - A set of edges: *E*
  - *E* & *V* are related in a way that the vertices on both ends of an edge are members of *V*
  - Usually written as *G* = (*V*, *E*)
- Usually Vertices are used to represent a position or state meanwhile Edges are used to represent a transaction or relationship

Terms and definitions

- There are a large number of variety on graph structure, in here, we assume no loop & parallel/multiple edges *(loop: an edge which joins a vertex to itself)*

- We may be working with graph with cycle, though… *(cycle: a path (sequence of vertices, length >= 3) which go back to the starting vertex with each edge used at most once)*

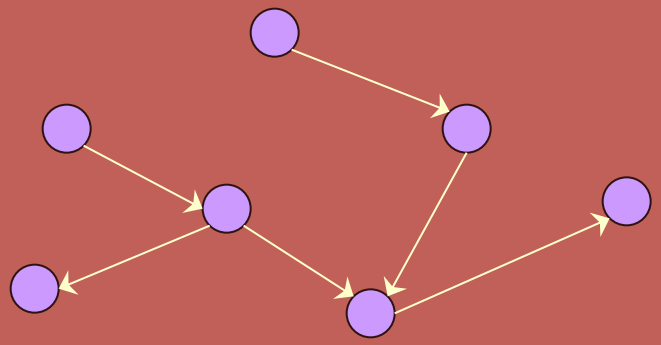


D-C-E-F-D is a cycle

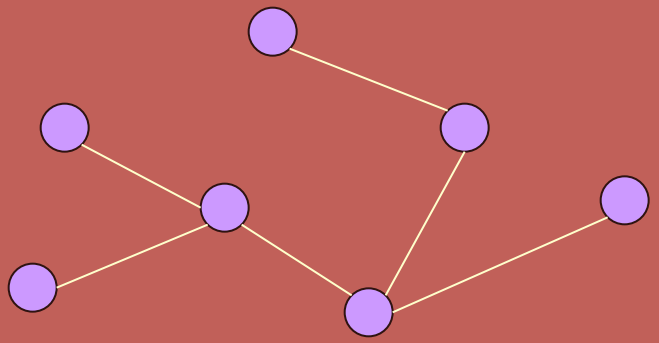
Terms and definitions

- Graph can be directed or undirected.
- In directed graph, there may be a pair of edges connecting the same pair of vertices
  (going in opposite direction…)

Directed graph

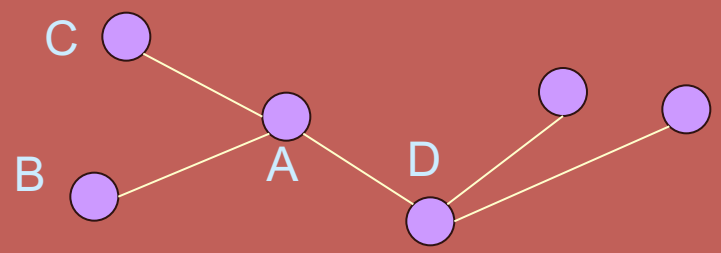


Undirected graph



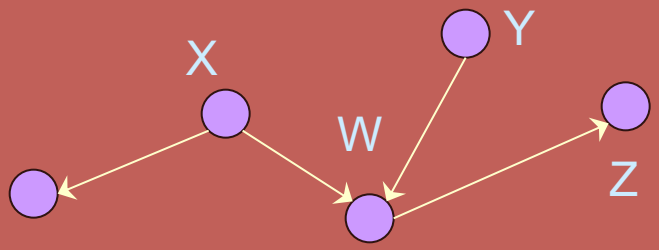
Terms and definitions

- Degree of a vertex is the number of edges connecting to it



Node A is having a degree of **3** (connects B, C, and D)

- For directed graph, degree is further classified as in-degree *(to this vertex)* & out-degree *(from this vertex)*

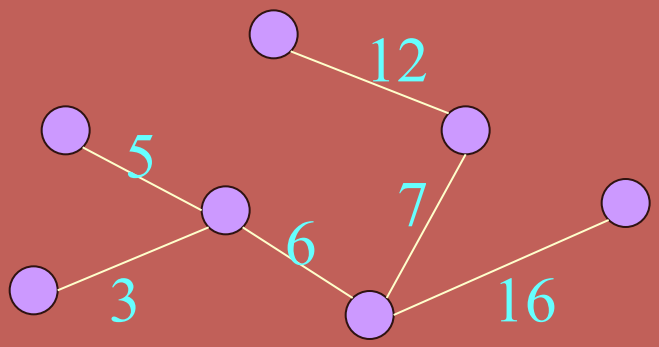


Node W is having an in-degree of **2** (X, Y) and an out-degree of **1** (Z)

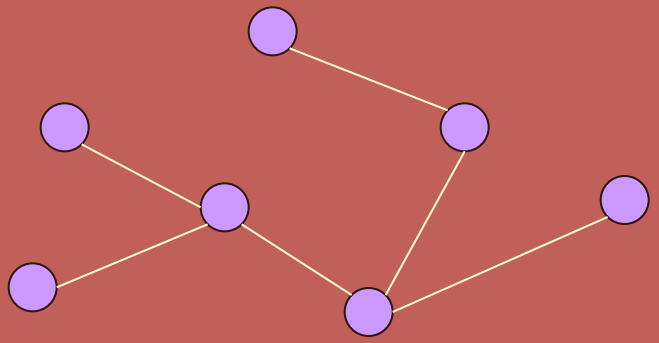
Terms and definitions

- Graph can be un-weighted or weighted, in which a value is associated with each edge.
- In directed graph, the weights of edges going in opposite directions can be different.
- For example:
  - *Whether a bus can go from Shatin to CityU:* Unweighted (=1...)
  - *The bus fee it takes from Shatin to CityU:* Weighted

Weighted graph

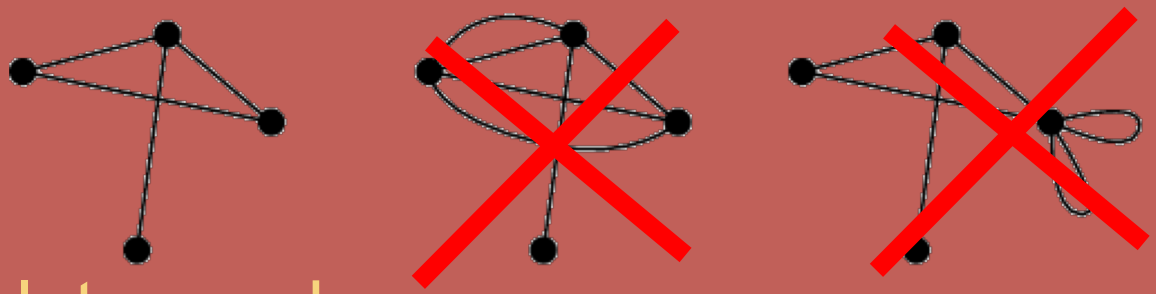


Unweighted graph

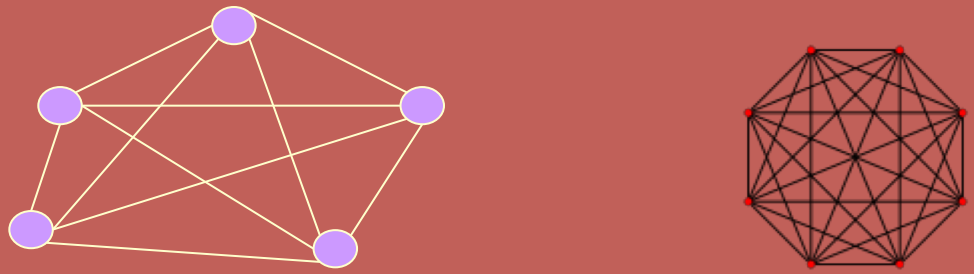


Terms and definitions

- Simple graph:
  - an un-weighted, undirected graph containing no graph loops or multiple edges

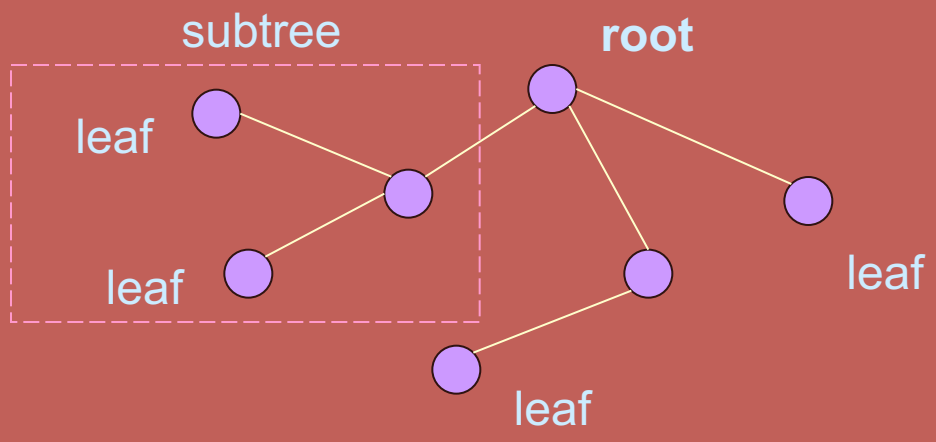


- Complete graph:
  - A simple graph in which every pair of vertices are connected directly.
  - If number of vertices = V, number of edges = ?



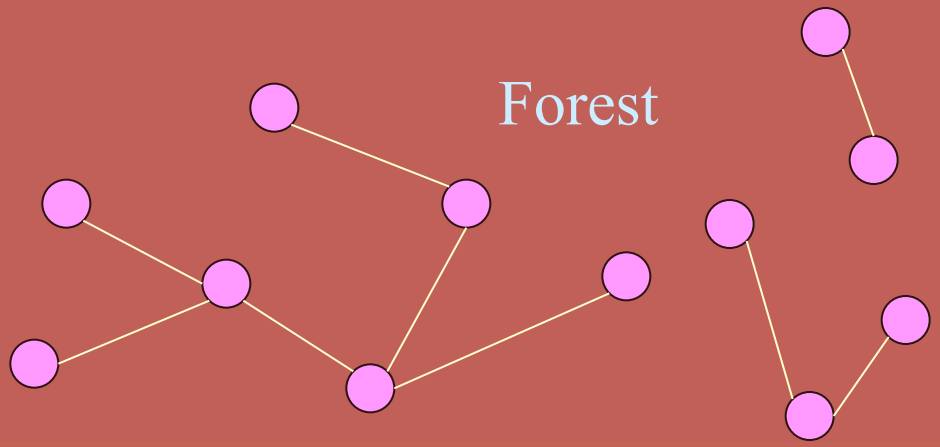
Terms and definitions

- Tree:
  - A special type of connected acyclic graph
  - Vertices connected by an edge inherit a parent-child relationship
  - Tree is recursively defined
    - A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself



Terms and definitions

- Tree:
  - Implementation and usage of tree is a bit different from typical graphs *(more on this in your yr2 course)*
  - What is the relationship between *the number of vertices* v.s. *the number of edges*?
- Forest:
  - Multiple connected components or a set of rooted trees



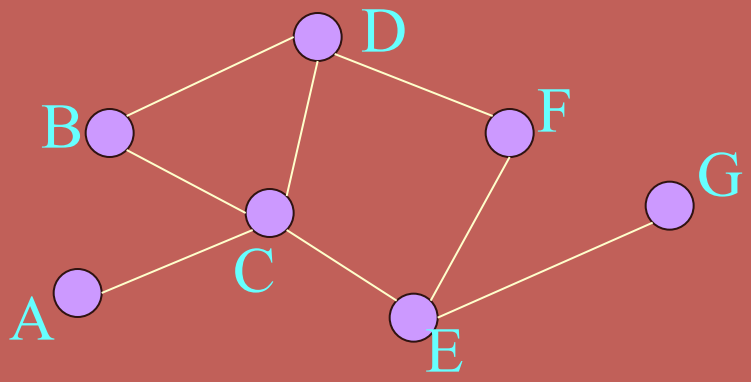
Representations of graphs

- When working with graph, we always perform one of the following operation:
  - Get the list of vertices connecting a given vertex.
  - Is vertices A & B connected?
  - What is the weight of edge from A to B?
  - What is the in/out degree of a vertex?
- 2 standard representations
  - Adjacency Matrix
  - Adjacency List

Representations of graph

- Adjacency Matrix
  - Use N*N 2D array to represent the weight (or T/F) of one vertex to another

A undirected graph



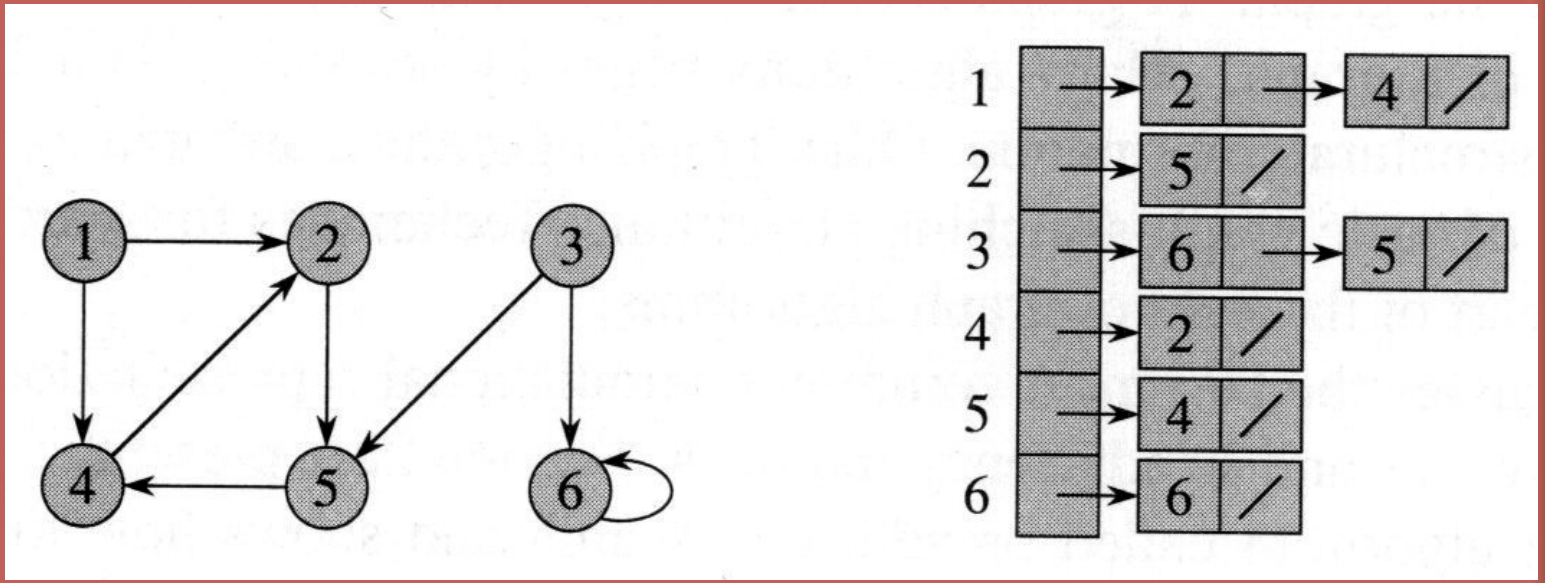
	A	B	C	D	E	F	G
A	0	0	1	0	0	0	0
B	0	0	1	1	0	0	0
C	1	1	0	1	1	0	0
D	0	1	1	0	0	1	0
E	0	0	1	0	0	1	1
F	0	0	0	1	1	0	0
G	0	0	0	0	1	0	0

Representations of graph

- Adjacency Matrix
  - For undirected graph, only half of the array is used
  - Fast query on edge weight & connection
  - Total memory used: N^2 *(what if num of vertex = 10K?)*
  - Waste memory if the graph is sparse
    i.e. #Edge is much smaller than half of (#Vertex)^2
    a large proportion of the array will be zero
  - Slow when querying the list of neighboring vertices if the graph is sparse

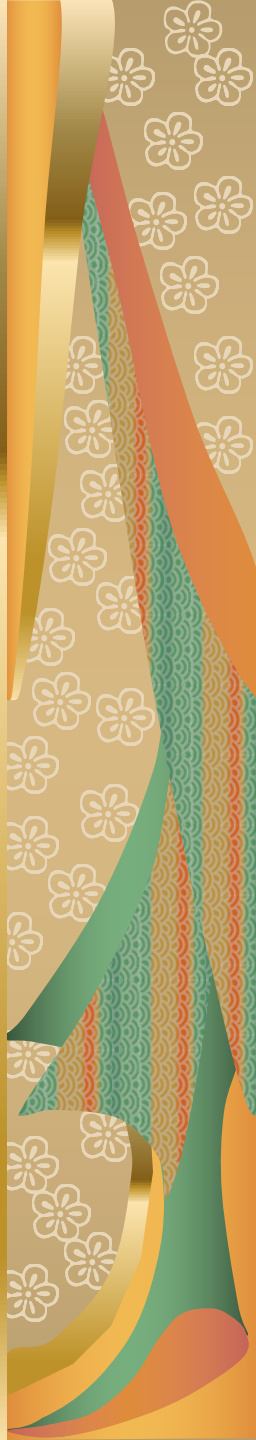
Representations of graph

- Adjacency List



Representations of graph

- Adjacency List
  - Use link list *(or equivalent)* to store the list of neighboring vertex. *(anyone use link-list in contest?)*
  - Save memory if the graph is sparse
  - Query on edge weight / connection can be slow
  - Graph update is slow *(especially if one have to maintain order of neighbors)*
  - Enumeration of all neighbors is fast



Representations of graph

- If memory is sufficient and graph update is in-frequent, can represent the graph in both methods at the same time…
  - *If you need fast enumeration of neighbors together with fast query of weight/connection*
  - *e.g. List out all the neighbors of vertex A which do not connect to vertex B or vertex C…*
- Link-list can be replaced by 1D array with count *(enumeration of neighbor and query on degree will be fast)*
- For un-weighted adjacency matrix, you may consider other possibilities other than 0 & 1….

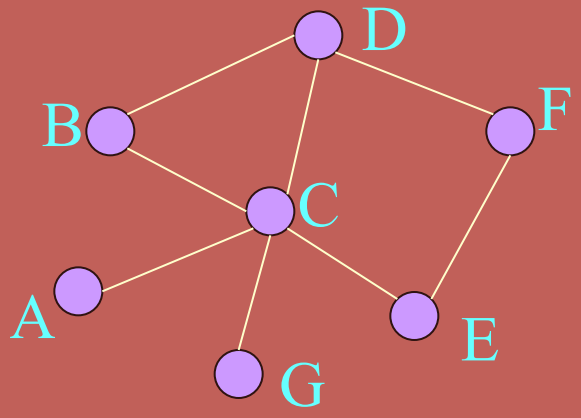
Graph Searching

- To determine whether two vertices are connected (indirectly via some intermediate)
  - A is a relative of B, B is a relative of C, are A & C relative?
- To list out all members of a *connected-component*
  - List out all the direct/indirect family members of A…
- To find the shortest path (of un-weighted graph) from one vertex to another
  - Travel from Shatin to Central with minimum number of change of transportation…
- TWO algorithms:
  - DFS (Depth First Search)
  - BFS (Breadth First Search)

Depth first search (DFS)

- Starts with vertex *v*:

```
DFS (v) {
    visited[v] = true;
    for each vertex w adjacent to v {
        if (! visited[w])
            DFS (w); //Recursion
    }
}
```

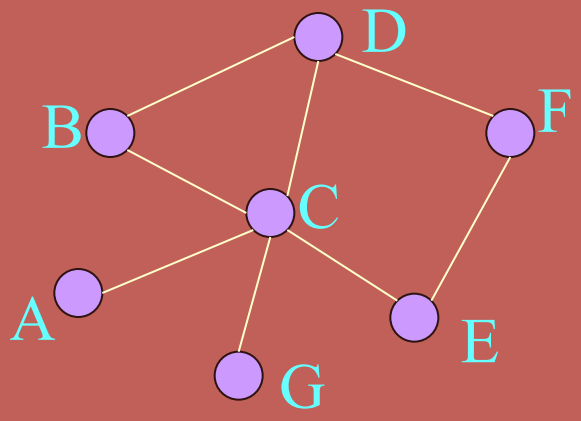


DFS(A) = A, C, B, D, F, E, G

Breadth first search (BFS)

- Non-recursive
- Make use of Queue
- Queue is a container with 1 entry & 1 exit, having FIFO (First-In-First-Out) property
- Enqueue/Dequeue: Add/Remove item from queue
- BFS can be used to find the shortest distance from a vertex to vertex within an unweighted acyclic graph

BFS (v)




```
BFS(v) {
  visited[v] = true;
  Enqueue(v);
  While queue not empty {
    x = Dequeue();
     for each vertex w adjacent to x {
          if (! visited[w]) {
                Enqueue (w);
                visited[w] = true;
          }
     }
  }
}
```

BFS(A) = A, C, B, D, E, G, F

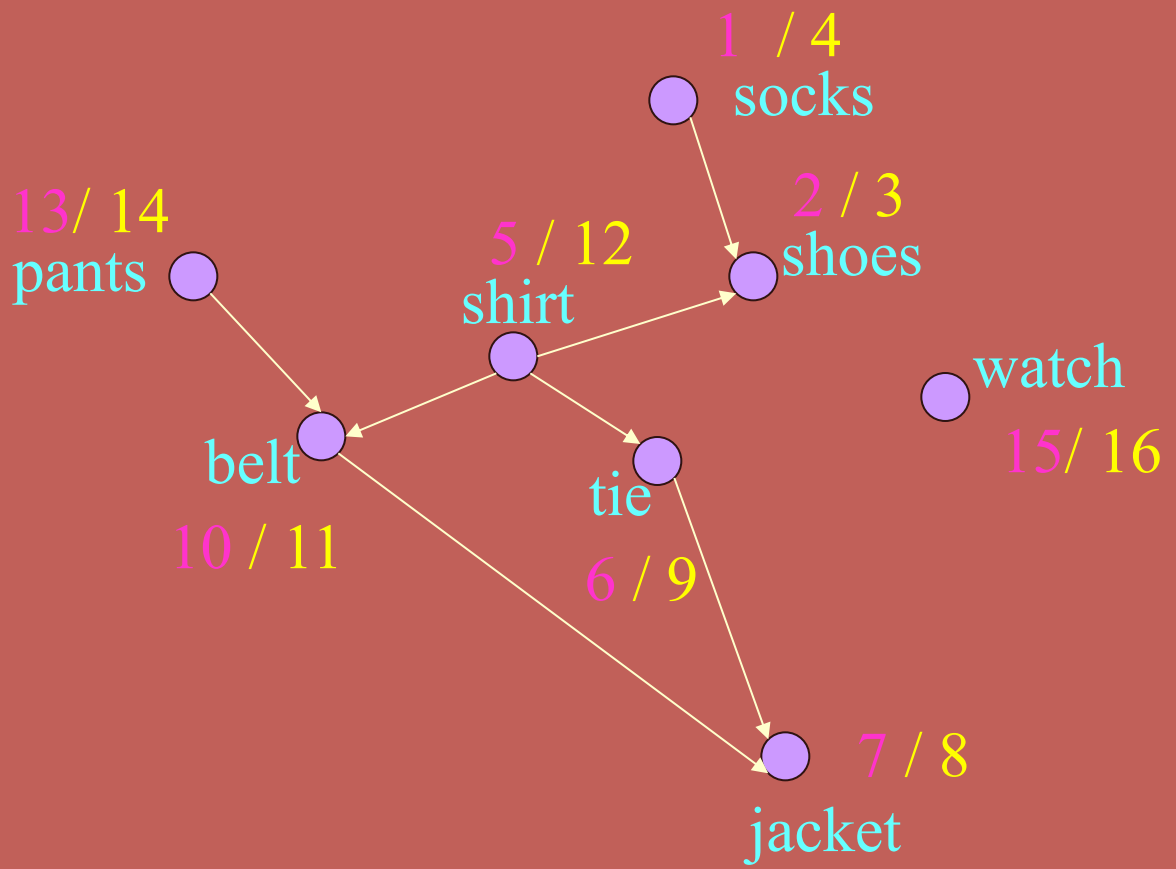
Topological sort

- One application of DFS is topological sort
  (Topological sort can be solved by some other methods...)
- A linear ordering of vertices:

  If the directed graph contains an edge (u, v), then u appears before v.

1. Use DFS to calculate the finishing time for each vertex
2. As each vertex is finished, insert it onto the front of a list
3. Return the list of vertices

An example *(Ref: Introduction to Algorithms 2nd ED)*



Topological order:

watch, pants, shirt, belt, tie, jacket, socks, shoes

Exercises

1. Connectivity

   Given an adjacency matrix of an undirected graph, you need to check if the graph is connected.

   Input format:

   1st line -- number of test cases

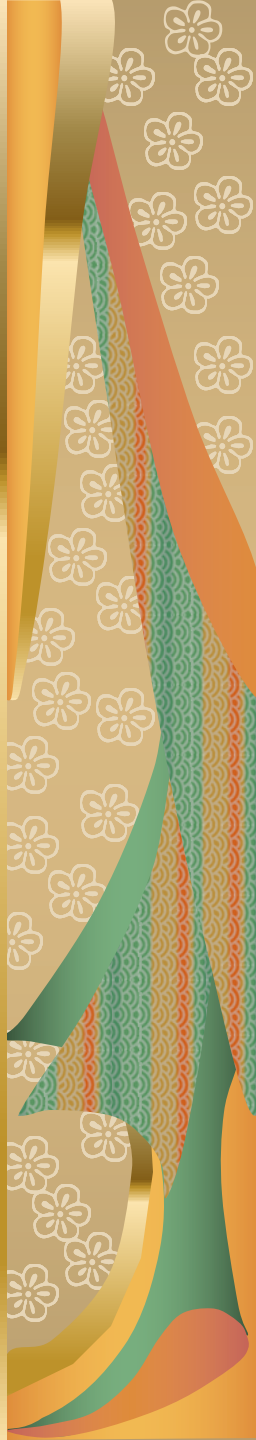
   2nd line -- blank

   3rd line -- (1st test case) n: width of the matrix ($1<n\leq 20$)

   4th to (4+n)th line -- adjacency matrix

   (5+n)th line -- blank

   (6+n)th line -- (2nd test case) ….

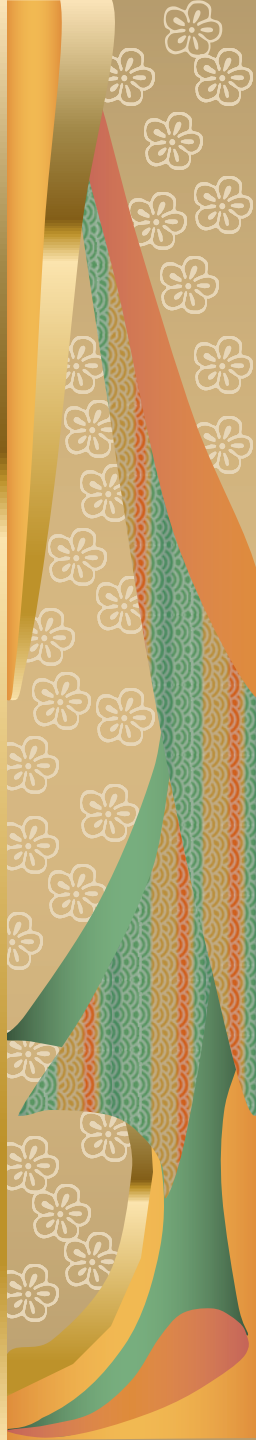


Sample input:

```
4

5
01111
10111
11011
11101
11110

6
001011
000100
100000
010000
100000
100000

12
001011010100
000100000000
100000000000
010000000001
100000000000
100000000000
000000001010
100000000000
000000100010
100000000000
000000101000
000100000000

20
01001100000000000000
10100000000100010000
01011000001100000000
00101010100000000001
10110000000000000100
10000000000000000100
00010000000000000010
00000000111001000000
00010001000000000001
00000001000000000000
00100001000001000000
01100000000010110000
00000000000100101000
00000001001000000000
00000000000110000000
01000000000100000000
00000000000010000000
00001100000000000000
00000010000000000000
00010000100000000000
```

Sample output:

```
Yes
No
No
Yes
```

2. Measuring water

You are given 2 empty containers having capacities of x, y liters respectively ($1 \le x \le y \le 100$), and you need to use them to measure exactly z liter ($0 \le z < 100$) of water. The supply of water is unlimited and can be pour into / out of the containers. However, the pouring action is restricted:

- When filling water in a container, the container must be fully filled.
- When pouring water out of a container, all the water must be poured out unless the destination container is full.

Assume all containers are initially empty, write a program to count the minimum number of steps (fill/pour water) to get z Liters of water in any container.

Print "No Solution" if it is impossible to measure z Liters exactly.

Input format:

1st line -- (1st test case) x y z

2nd line -- (2nd test case) ….

(continue input until end of file)

Sample input:

```
5 3 4
6 3 4
```

Sample output:

```
6
No Solution
```


Exercises

- Valladolid Programming Contest
  - #459 - Graph Connectivity
    http://acm.uva.es/p/v4/459.html
  - #567 - RISK
    http://acm.uva.es/p/v5/567.html
  - #10009 - All Roads Lead Where?
    http://acm.uva.es/p/v100/10009.html
  - 10113 - Exchange Rates *(requires GCD)*
    http://acm.uva.es/p/v101/10113.html
  - 10150 - Doublets
    http://acm.uva.es/p/v101/10150.html