CS1 Lecture 33 Apr. 10, 2017

- HW 8 available noon today, due next Wed.
  - 6 point question available today, 4 point question will be added when we cover relevant material Friday
- Discussion sections this week: A graph problem requiring slight modification of today's BFS algorithm.
- Exam 2: April 20, 6:30-8:00pm

Last time

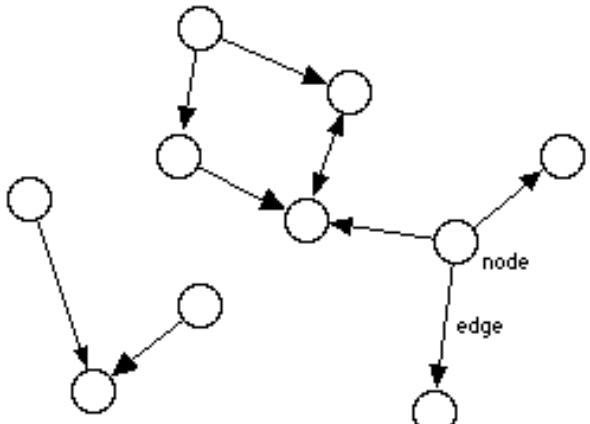
- Sorting summary
- A short discussion of optimization problems and greedy algorithms
- Introduction to graphs and graph algorithms

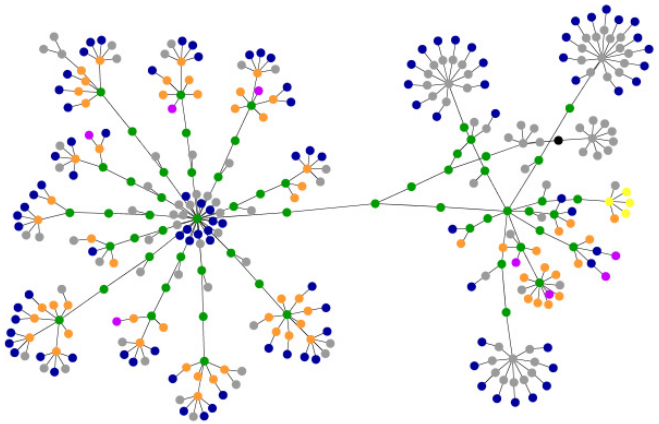
Today

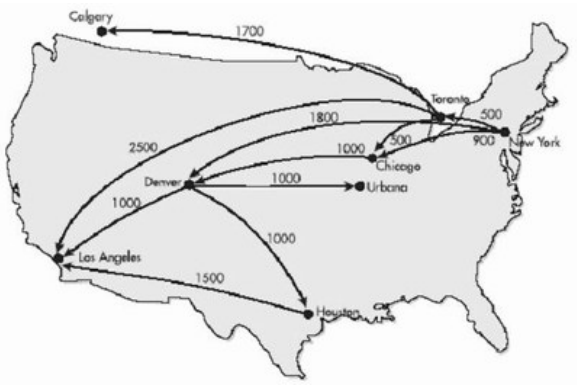
- Graph representation
- A graph traversal algorithm
- Introduction to HW8 graph problem

Graphs and optimization problems based on graphs

- *Many* important real-world problems can be modeled as optimization problems on **graphs**
- A graph is:
  - A set of nodes (vertices)
  - A set of edges (arcs) representing connections between pairs of nodes
- There are several types of graphs:
  - **Directed**. Edges are "one way" from source to destination)
  - **Undirected**. Edges have no particular direction – can travel either way, "see" each node from other, etc.
  - **Weighted**. Edges have associated numbers called weights that can be used to represent cost, time, flow capacity, etc.
- See Ch. 2 of follow-up book to our text – Think Complexity - http://greenteapress.com/complexity/html/thinkcomplexity003.html



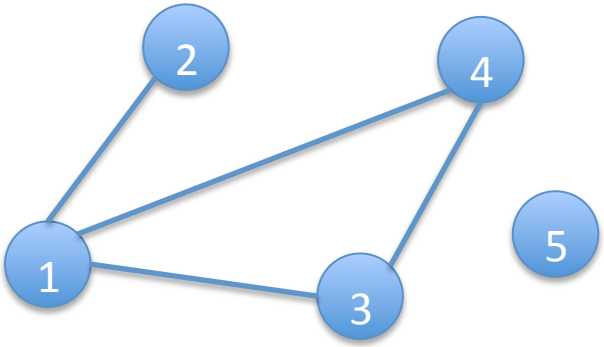




Representing graphs

- How can we represent general graph in Python?
  - Need to keep track of nodes
  - Need to keep track of edges
- Several ways to represent graphs have been developed
    - List of nodes and list of edges
    - Adjacency matrix
    - Adjacency lists
    - Dictionary of dictionaries
  - Efficiency of algorithms that solve graph problems can vary greatly depending on how graph are representated
  - a strong influence on choice is the fact that one of the most common things needed in graph algorithms is access to immediate neighbors of a node (nodes that are destinations of edges for which “current” node is source)

Last time - Adjacency matrix



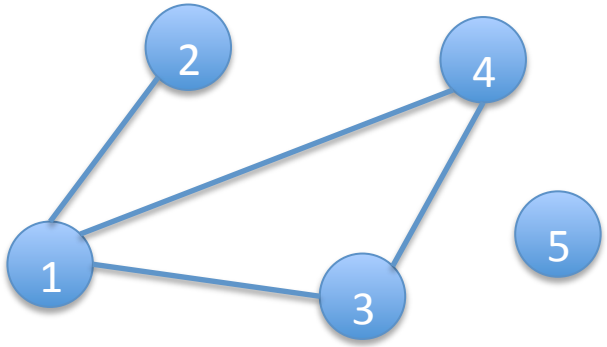
	1	2	3	4	5
1	False	True	True	True	False
2	True	False	False	False	False
3	True	False	False	True	False
4	True	False	True	False	False
5	False	False	False	False	False

- Appealingly simple to understand and implement
- Use, e.g. a list of lists containing True/False, 0/1, or similar
- NOT the most common graph representation for most problems. Can you think of a reason why?
  - Consider representing Facebook friends graph where each node is a FB user and an edge exists between two nodes whenever the two are FB friends.
  - One billion nodes. Adjacency matrix 1B x 1B in size! Your computer doesn't have that much storage. But FB graph ***can*** be represented in computer! How?
  - The 1B x 1B would be mostly False/0 – most people don't have huge number of friends. Should be representable in closer to 1B * median number of friends. Other representations enable this huge memory savings.

Adjacency list

Use a dictionary with

- Nodes as keys
- Values are lists of neighbor nodes



KEY	VALUE
1	[2, 4, 3]
2	[1]
3	[1, 4]
4	[3, 1]
5	[]

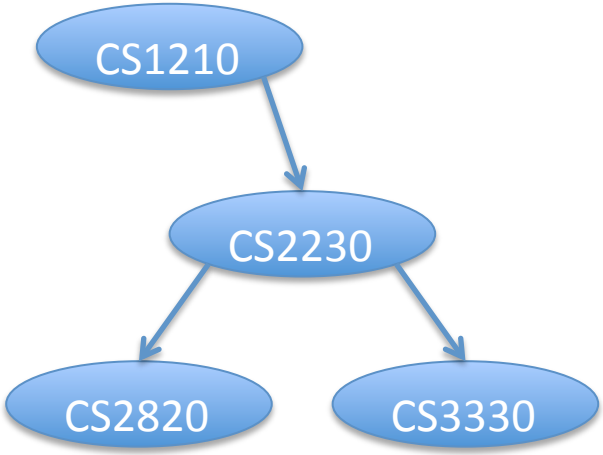
Compared to adjacency matrix:

+ Much less space (when, as is common, most nodes have only a small relatively small number of neighbors). Facebook graph. People have hundreds of friends, not many milliions

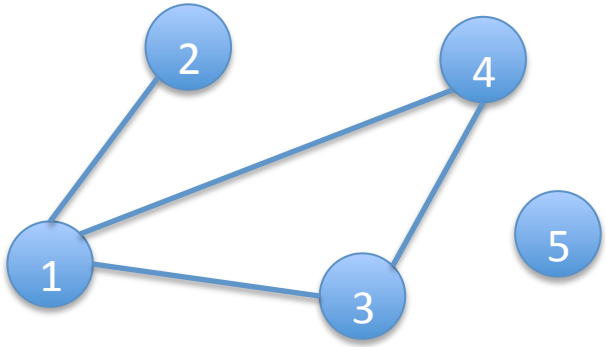
- Query of “does edge (i,j) exist?” not O(1). Must search list associated with node i to see if j is there. Turns out this is not crucial in many graph algorithms. (could address this using dictionary of dictionaries but often not necessary)

Adjacency list graph representation

Suitable for both undirected and directed graphs (and can be use for weighted graphs as well)



KEY	VALUE
CS2230	[CS2820, CS3330]
CS2820	[]
CS1210	[CS2230]
CS3330	[]



KEY	VALUE
1	[2, 4, 3]
2	[1]
3	[1, 4]
4	[3, 1]
5	[]

An **adjacency list** representation for undirected graphs in Python

Two classes: Node and Graph

basicgraph.py

Node

- properties:
  - name : string
  - status: string (we'll use this to "mark" nodes during traversals)
- methods
  - getName
  - __repr__ : we'll print nodes as <*name*>

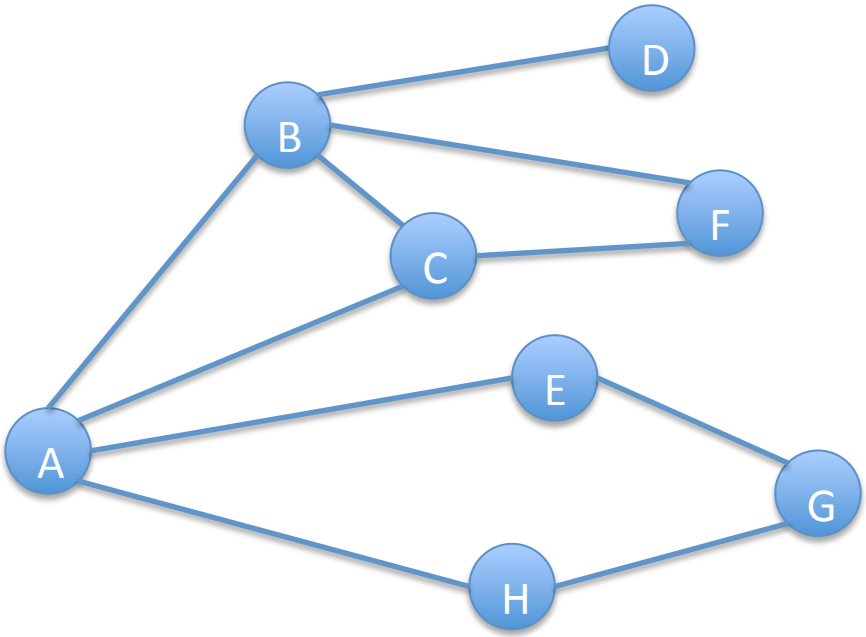
Note: in your HW8 you'll add one or more additional properties that help with traversing/walking through graphs to solve specific problems

Adjacency list representation for undirected graphs

Graph

- properties
  - nodes: a list of Node objects
  - edges: a list of 2-tuples of nodes
  - adjacencyLists: a dictionary with all nodes as keys. The value associated with a key n1 (where n1 is a node) is a list of all the nodes, n2, for which (n1,n2) is an edge.
- methods
  - addNode(node) : nodes must be added to graphs before edges
  - addEdge(node1, node2) : presumes both nodes in graph already
  - neighborsOf(node) : returns list of neighboring nodes
  - getNode(name)
  - hasNode(node)
  - hasEdge(node1, node2) : return T if edge node1-node2 in graph
  - __repr__

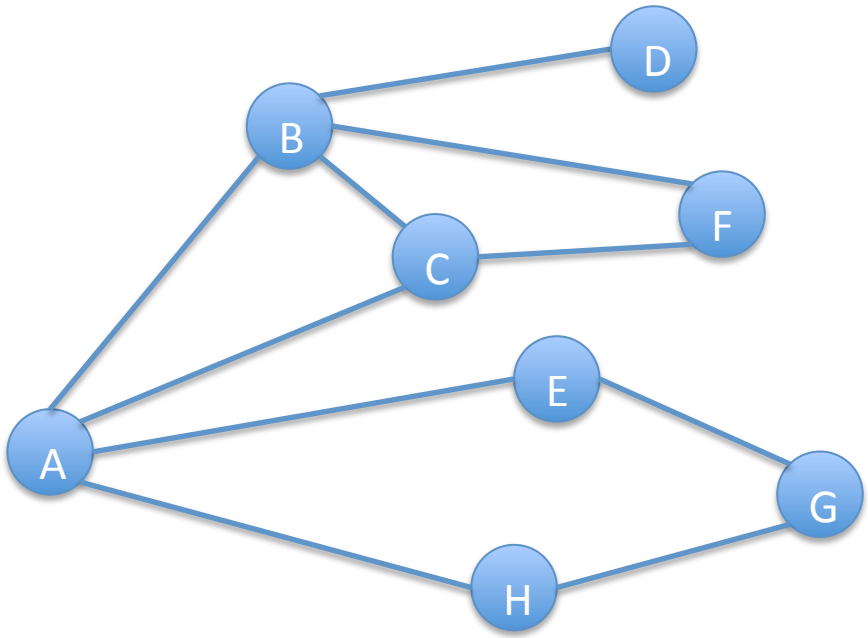
basicgraph.py



KEY	VALUE
A	[B,C,E,H]
B	[A,C,D,F]
C	[A,B,F]
D	[B]
E	[A,G]
F	[B,C]
G	[E,H]
H	[A,G]

This graph is generated by genDemoGraph() in basicGraph.py

Note: for exams, you need to be able to 1) draw graph given adjacency list dictionary, and/or 2) show adjacency list dictionary given graph drawing



KEY	VALUE
A	[B,C,E,H]
B	[A,C,D,F]
C	[A,B,F]
D	[B]
E	[A,G]
F	[B,C]
G	[E,H]
H	[A,G]

As I've said, many real-world problems can be represented as problems involving graphs. The algorithms to solve those problems often involve **graph traversals**, organized exploration or "walkthroughs" of the graph.

Two famous ones are: depth-first search and breadth-first search. I will present breadth-first search.

You will not be responsible for knowing the details of breadth-first search (for exam purposes) but you need to understand it well enough to *use* it in HW8.

Classic breadth-first search (bfs)

The goal of classic bfs is simply to travel to/explore, in an efficient and organized fashion, all nodes reachable from some given startNode. (The general goal of the other classic traversal, depth-first-search, is the same,. It just explores in a different order.)

First, we'll add a property, **status**, to nodes. Legal values will be 'unseen', 'seen', and 'processed'. The traversal process will use the values as it encounters nodes to keep from revisiting/re-processing already-processed nodes.

The algorithm in words:

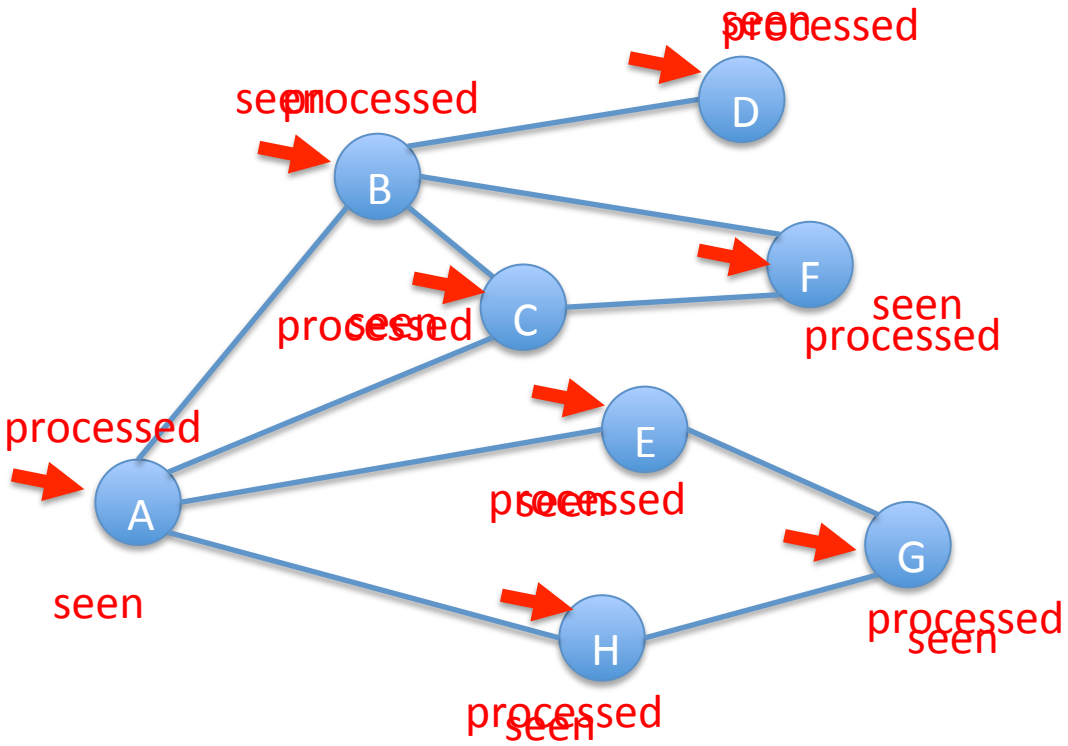
1. Mark all nodes as 'unseen'
2. Initialize an empty queue (you implemented a queue class in DS6. "First-in first-out")
3. Mark the starting node as 'seen' and place it in the queue.

4. Remove the front node of the queue and call it the current node
5. Consider each neighbor of the current node. If its status is 'unseen', mark it as seen and put it on the queue.
6. Mark the current node 'processed' (*note: this step can be left out*).
7. If queue not empty go back to step 4.

This will explore all node reachable from the startNode in a breadth-first manner.

- Suppose startNode has neighbors n1 and n2. "Breadth first manner" means it travels start to n1 and then and then from start to n2 before exploring "beyond n1" – the process moves "broadly" out from the start a single step at a time.
- This enables a very simple modification of bfs to compute shortest (unweighted) distances from start to all other nodes in the graph

BFS starting at node A



Mark all nodes ‘unseen’
Mark A ‘seen’ and put A on queue Q
Until queue empty do:

- Remove the front node of the queue and call it the current node
- Consider each neighbor of the current node. If its status is ‘unseen’, mark as ‘seen’ and put it on the queue.
- Mark the current node ‘processed’

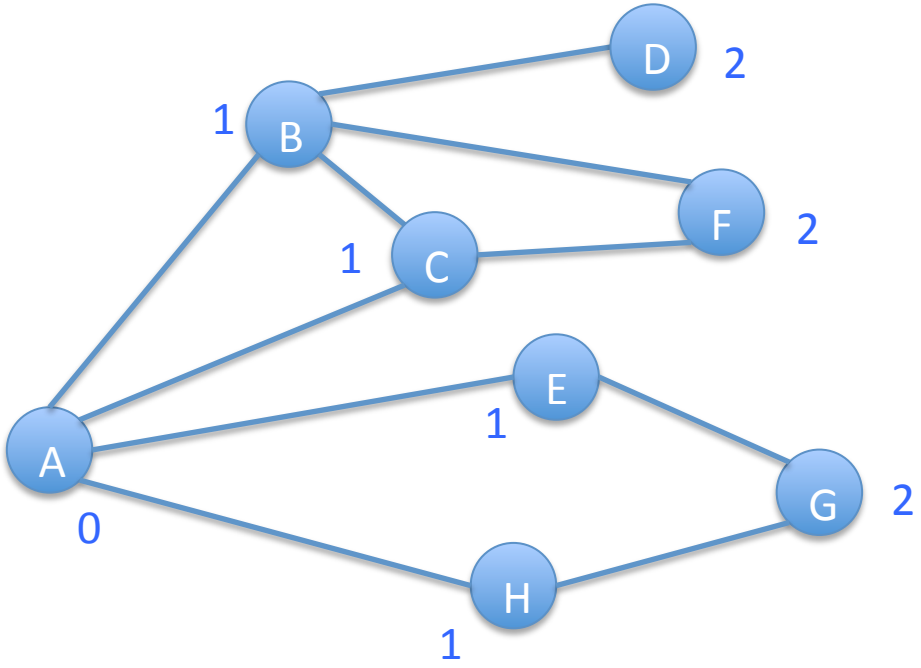
	Q: A
curr: A	Q:
	Q: B, C, E, H
curr: B	Q: C, E, H
	Q: C, E, H, D, F
curr: C	Q: E, H, D, F
	Q: E, H, D, F
curr: E	Q: H, D, F
	Q: H, D, F, G
curr: H	Q: D, F, G
	Q: D, F, G
curr: D	Q: F, G
	Q: F, G
curr: F	Q: G
	Q: G
curr: G	Q:
	Q: DONE

BFS

If we don't consider nodes fully explored until their status value becomes "processed", then BFS explores the graph in "levels"

- first level 0 – items distance 0 from start
- then level 1 – items distance 1 from start
- then level 2
- Etc.
- This is very useful.

We can make a very slight change to bfs to record distance of each node from start!



Mark all nodes ‘unseen’

Mark A ‘seen’ and put A on queue Q

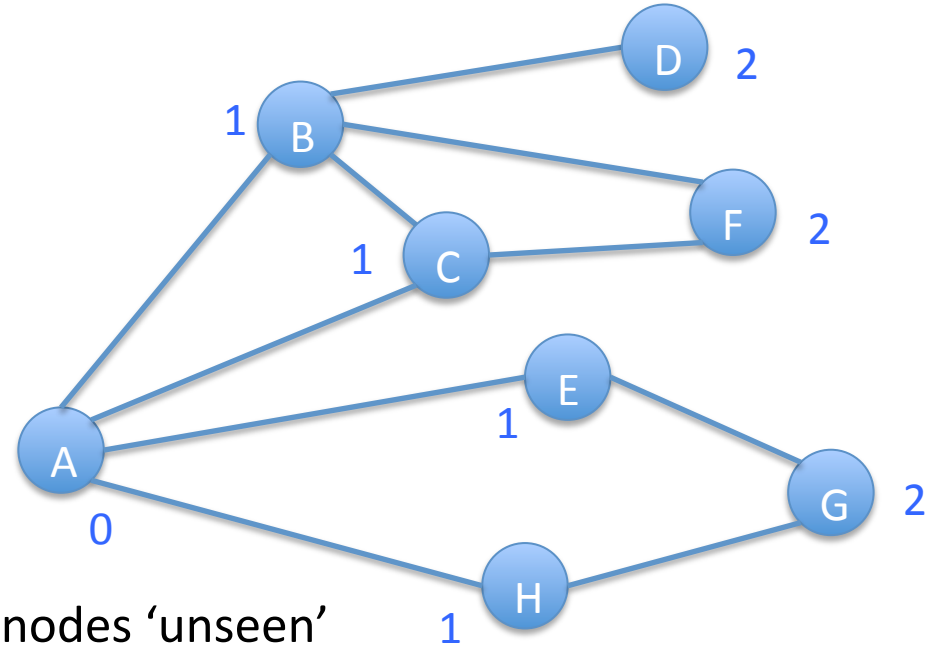
Until queue empty do:

- Remove the front node of the queue and call it the current node
- Consider each neighbor of the current node. If its status is ‘unseen’, mark as ‘seen’ and put it on the queue.
- Mark the current node ‘processed’

Add a distance property to Node

Update node distance when first seen

BFS starting at node A



Mark all nodes ‘unseen’

Mark A ‘seen’, set A’s distance to 0, and put A on queue Q

Until queue empty do:

- Remove the front node of the queue and call it the current node
- Consider each neighbor of the current node. If its status is ‘unseen’, mark as ‘seen’, set its distance to one more than current node’s distance and put it on the queue.
- Mark the current node ‘processed’

Bread first search

- For unweighted graphs, bfs efficiently finds shortest path from start node to every other node!
  - "Three and a half degrees of separation" https://research.fb.com/three-and-a-half-degrees-of-separation/
  - Wikipedia game: given two topics (with Wikipedia pages), race to get from one to the other clicking only on Wikipedia page links. https://en.wikipedia.org/wiki/Wikipedia:Six_degrees_of_Wikipedia https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia
  - HW8 word ladders!

- Links
  - http://en.wikipedia.org/wiki/Breadth-first_search
  - http://interactivepython.org/courselib/static/pythonds/Graphs/ImplementingBreadthFirstSearch.html
  - animations:
    - https://www.cs.usfca.edu/~galles/visualization/BFS.html
    - http://visualgo.net/dfsbfs.html
    - http://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

Word ladder puzzles

CAT	CAT	CAT
???	COT	COT
???	???	DOT
DOG	DOG	DOG

Find 3-letter English words for ??? Positions. Each must differ from previous and next word in only one location

This problem is easily representable and solvable using graphs! HW8 Q1

HW8 Q1 Word Ladder problem

Each part is pretty simple:

1. Read word list and create graph
   - One node per word
   - Edge between each pair of words differing by one letter
2. Execute breadth-first search starting from start word's node.
3. After breadth first search, can "extract" ladder from graph by following a parent path (you will add 'parent' property to Node class and set during bfs) from end word's node back to start node
4. print ladder

Start from wordladderStart.py (see HW8 assignment) with stubs for all the functions you need.

Next time and Friday

- BFS, DFS, and the HW8 word ladder problem
- Friday: Introduction to randomization and simulation/Monte Carlo techniques