Graphs

15-110 – Friday 10/10

Announcements

- Final exam is scheduled: Friday 12/12 1-4pm
  - Do not schedule travel before this date!
  - Location TBD

- Next week: **Fall Break!** Have fun!
  - No classes, no office hours, no guaranteed responses on Piazza from **Saturday 10/11 – Sunday 10/19**

Midsemester Grades

- Midsemester grades will include:
  - Exercises from weeks 1-4
  - Check1, Hw1, Check2, Hw2
  - Quizlets 1-4 (with your lowest score dropped)
  - Exam1
- Canvas shows your current grade. That grade is calculated as follows, to reflect the fact that the final exam correlates better with exam averages than homework averages:
  - Exercise average x 4%
  - Check average x 9%
  - Homework average x 15%
  - Quizlet average x 7%
  - Exam1 x 65%
- If you did poorly on Exam1 and it's dragging down your grade – **don't panic!** There's still time to turn things around and improve on Exam2 and the final.
  - Reach out to the professors or your TAs if you'd like to discuss strategies for improving your learning process

Midsemester Survey

- Please let us know what's working and what can be improved!

- **Course Survey:** https://bit.ly/110-f25-mid-course
- **TA Survey:** https://bit.ly/110-f25-mid-tas

- To thank you for your time, you get **3 bonus points on Hw4** for completing both surveys.
  - The survey itself is anonymous; follow the link provided when you submit the first form and that will lead you to a second non-anonymous form you can fill out for points.
  - Complete the surveys by the **Hw4 deadline** (Monday 10/27 noon) for bonus points.

Learning Goals

- Identify core parts of **graphs**, including **nodes, edges, neighbors, weights,** and **directions**.

- Use **graphs** implemented as dictionaries when reading and writing simple algorithms in code

Graphs

Graphs are Like More-Connected Trees

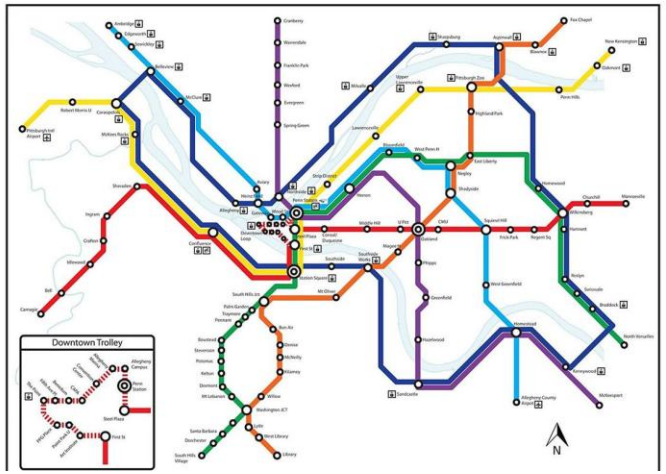
Last time we discussed trees, which let us store data by connecting nodes to each other to create a hierarchical structure.

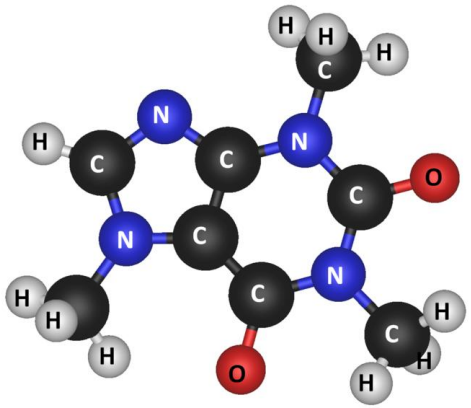
Graphs are like trees – they are composed of nodes and connect those nodes together. However, they have fewer restrictions on how nodes can be connected. **Any node can be connected to any other node in the graph**.

Graphs in the Real World

Graphs show up all the time in real-world data. We can use them to represent **maps** (with locations connected by roads) and **molecules** (with atoms connected by bonds).

We also commonly use graphs in algorithms, to represent data like **social networks** (with people connected by friendships), or **recommendation engines** (with items connected if they were purchased together).



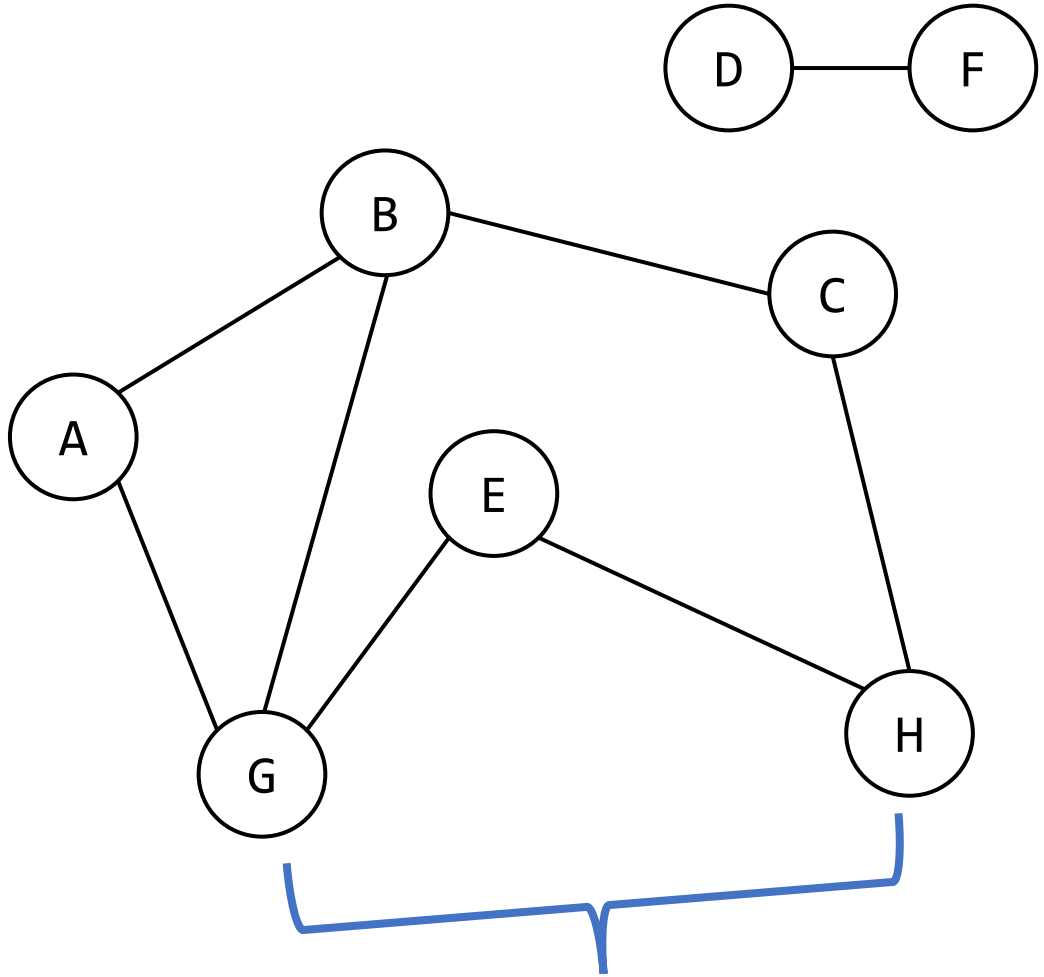


Graphs are Made of Nodes and Edges

The **nodes** in a graph are the same as the nodes in a tree – they hold the values stored in the structure.

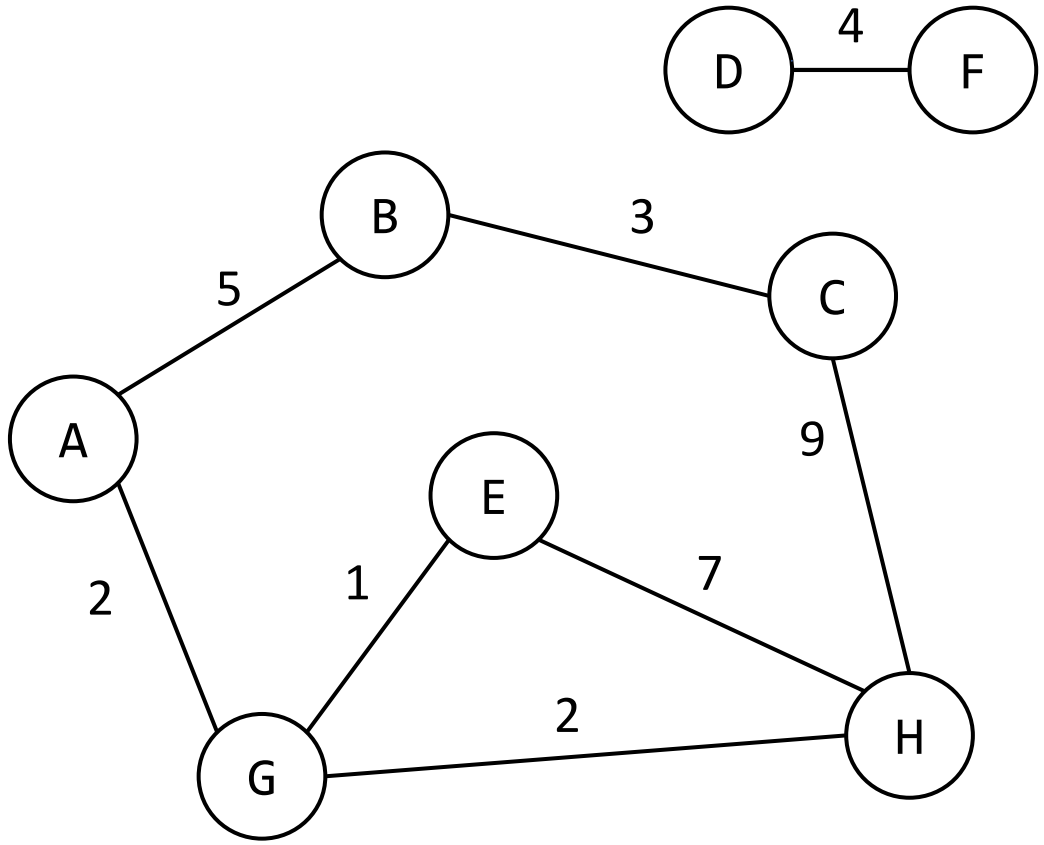
The **edges** of a graph are the connections between nodes.

We say that for a node X, any nodes that X connects to with an edge are X's **neighbors**.



Edges Can Have Weights

Sometimes edges can have **weights**, such as the length of a road or the cost of a flight. Our example graph here has weights- the numbers next to lines.



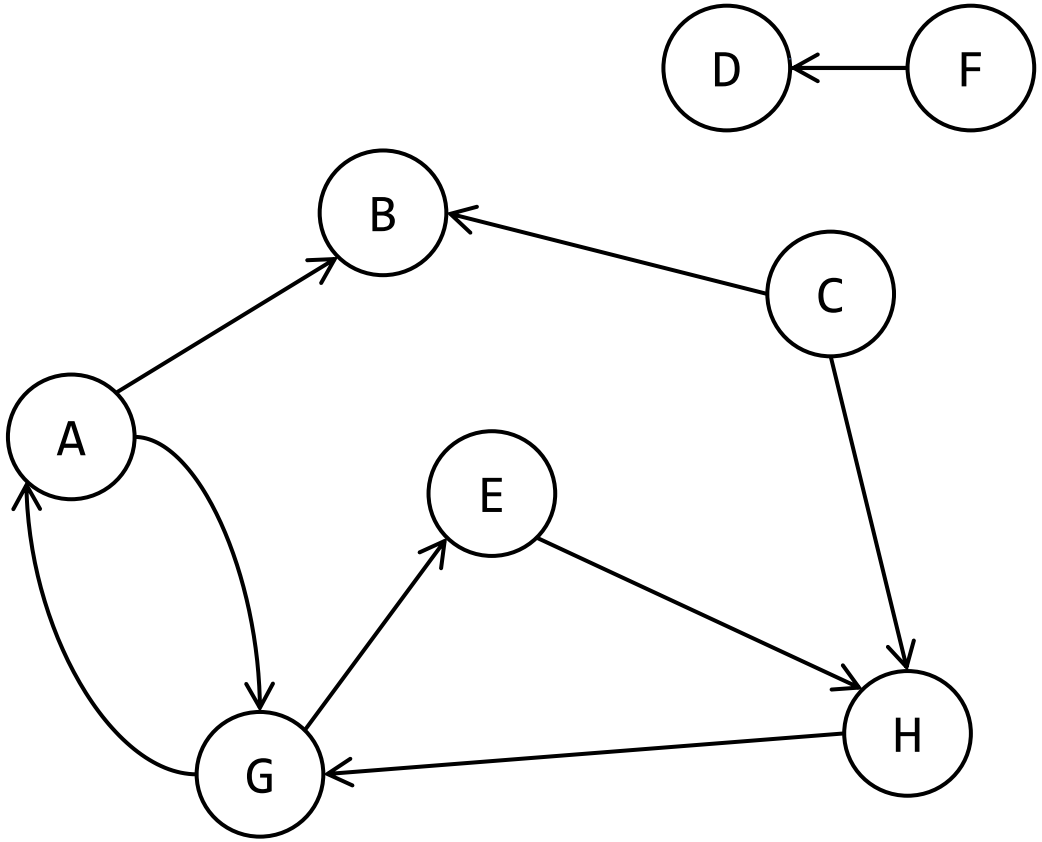
A graph with no weights is an **unweighted graph**; a graph with weights is a **weighted graph**.

Edges Can Have Directions

Technically D is F's neighbor, but F is not D's neighbor, because you can't go from D to F.

Edges can also be **directed** (from A to B but not from B to A unless there is another directed edge from B to A), or **undirected** (go in either direction on an edge between nodes).

The graph to the right is directed; for example, you can only go from G to E, not from E to G. The previous graphs we saw were undirected.



Activity: Recognize the Parts

Consider the graph to the right.

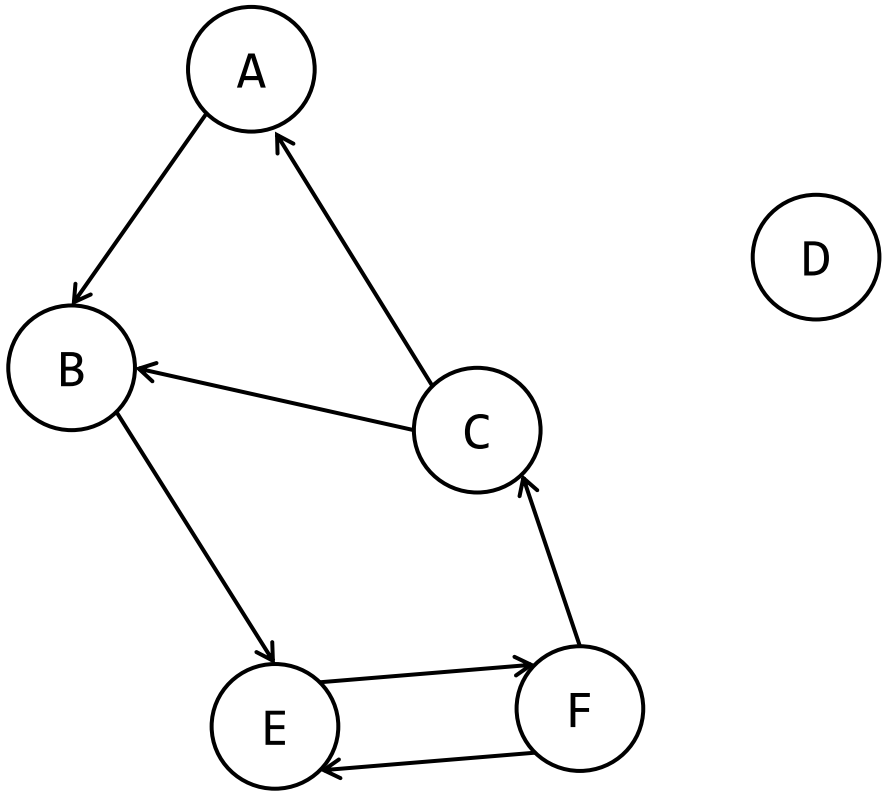
How many **nodes** does the graph have?

How many **edges?**

Do the edges have **weights?**

Are the edges **directed?**

What are the **neighbors** of node F?



Coding with Graphs

Represent Graphs in Python with Dictionaries

Like trees, graphs are not implemented directly by Python. We need to design an implementation using existing syntax.

Our implementation for this class will use a **dictionary** that maps node values to lists. This is commonly called an **adjacency list**.

Unlike the tree representation, graphs will not be recursively nested dictionaries; we'll be able to access all the node values directly. That's because graphs aren't inherently recursive.

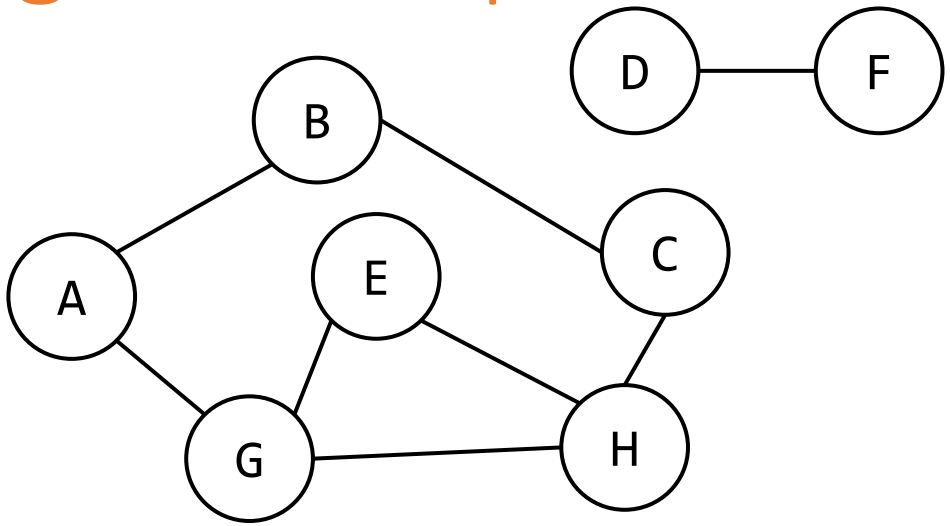
We'll need to slightly alter this representation based on whether or not the edges of the graph have weights.

Graphs in Python – Unweighted Graphs

First, how do we represent an **unweighted graph**?

The keys of the dictionary will be the **values of the nodes**. Each node maps to a **list of its adjacent nodes (neighbors)**, the nodes it has a direct connection to.

On the right, we show our example graph in its dictionary implementation.



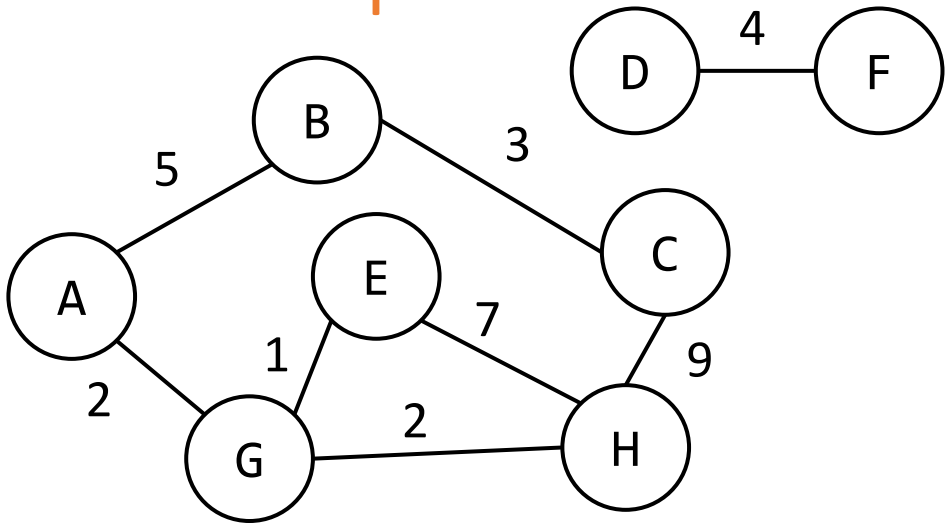

```
unweightedGraph = {
      "A" : [ "B", "G" ],
      "B" : [ "A", "C" ],
      "C" : [ "B", "H" ],
      "D" : [ "F" ],
      "E" : [ "G", "H" ],
      "F" : [ "D" ],
      "G" : [ "A", "E", "H" ],
      "H" : [ "C", "E", "G" ]
    }
```

Graphs in Python – Weighted Graphs

Weighted graphs have values associated with the edges. We need to store these values in the dictionary also.

We'll do this by changing the list of adjacent nodes to be a 2D list. Each of the inner lists represents a node/edge pair, so it has two values – the adjacent node's value and the weight of the edge.

On the right, we show our updated example graph in this format.




```
weightedGraph = {
      "A" : [ ["B", 5], ["G", 2] ],
      "B" : [ ["A", 5], ["C", 3] ],
      "C" : [ ["B", 3], ["H", 9] ],
      "D" : [ ["F", 4] ],
      "E" : [ ["G", 1], ["H", 7] ],
      "F" : [ ["D", 4] ],
      "G" : [ ["A", 2], ["E", 1], ["H", 2] ],
      "H" : [ ["C", 9], ["E", 7], ["G", 2] ]
    }
```

Finding a Graph's Nodes

Let's look at some basic examples of programming with graphs.

To print all the nodes in a graph, just look at every key in the dictionary.

```
def printNodes(g):
    for node in g:
        print(node)
```

Finding a Node's Neighbors

If we want to get the neighbors of a particular node, index into that node in the dictionary.

```
def getNeighbors(g, node):
    return g[node]
```

If the graph is weighted, we'll need to reconstruct the neighbor list:

```
def getNeighbors(g, node):
    neighbors = [ ]
    for pair in g[node]:
        neighbors.append(pair[0])
    return neighbors
```

Finding a Graph's Edges

To print all the edges, you'll need to loop over each value in the neighbor list.

```
def printEdges(g):
    for node in g:
        for neighbor in g[node]:
            print(node + "-" + neighbor)
```

Note that this example is for an unweighted graph. To get neighbor values in a weighted graph, index into `neighbor[0]` .

Finding an Edge's Weight

Finally, to find an edge's weight, index and loop to find the appropriate pair.

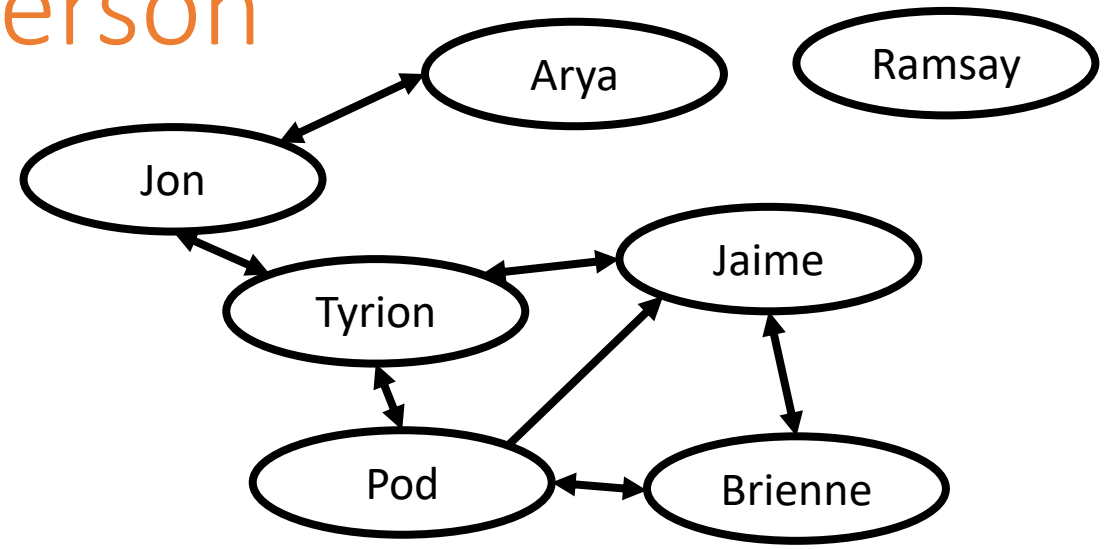
```
def getEdgeWeight(g, node1, node2):
    for pair in g[node1]:
        if pair[0] == node2:
            return pair[1]
```

Example: Most Popular Person

Now that we have the basics, we can start problem solving.

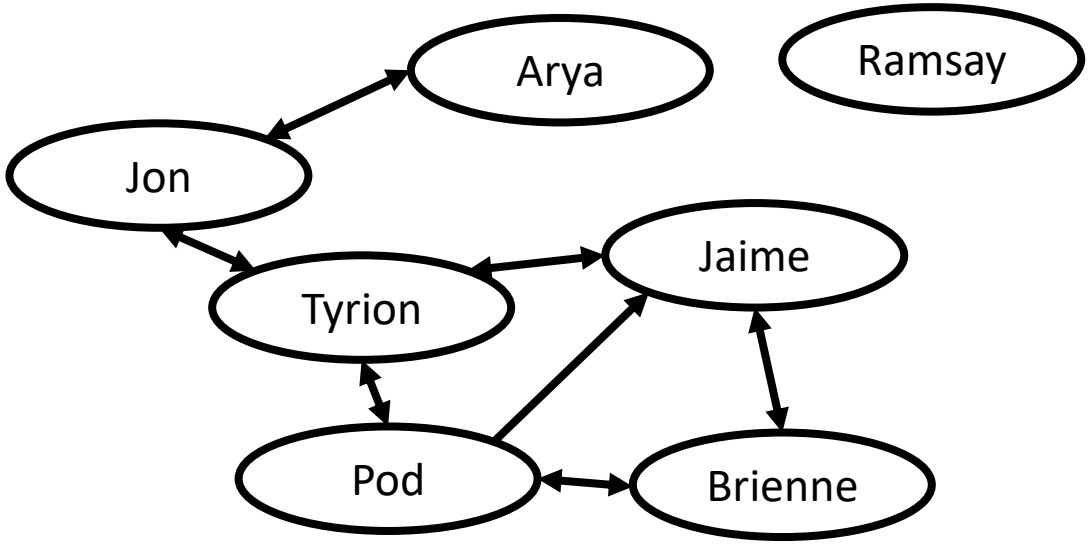
Let's write a function that takes a social network as a graph and returns the person in the network who has the most friends.

This is just our typical find-largest-property algorithm applied to a graph.




```
def findMostPopular(g):
    biggestCount = 0
    mostPopular = None
    for person in g:
        if len(g[person]) > biggestCount:
            biggestCount = len(g[person])
            mostPopular = person
    return mostPopular
```

Example: Make Invite List



Now let's say a person wants to make more friends, so they're holding a party. They want to invite their own friends, but also anyone who is a friend of one of their friends.

Now we have to loop over each of the person's friends, to access that node's own list of friends.

```
def makeInviteList(g, person):
    # start with immediate friends
    invite = g[person] + [ ] # break alias
    for friend in g[person]:
        # find friends-of-friends
        for theirFriend in g[friend]:
            if theirFriend not in invite and \
               theirFriend != person:
                invite.append(theirFriend)
    return invite
```

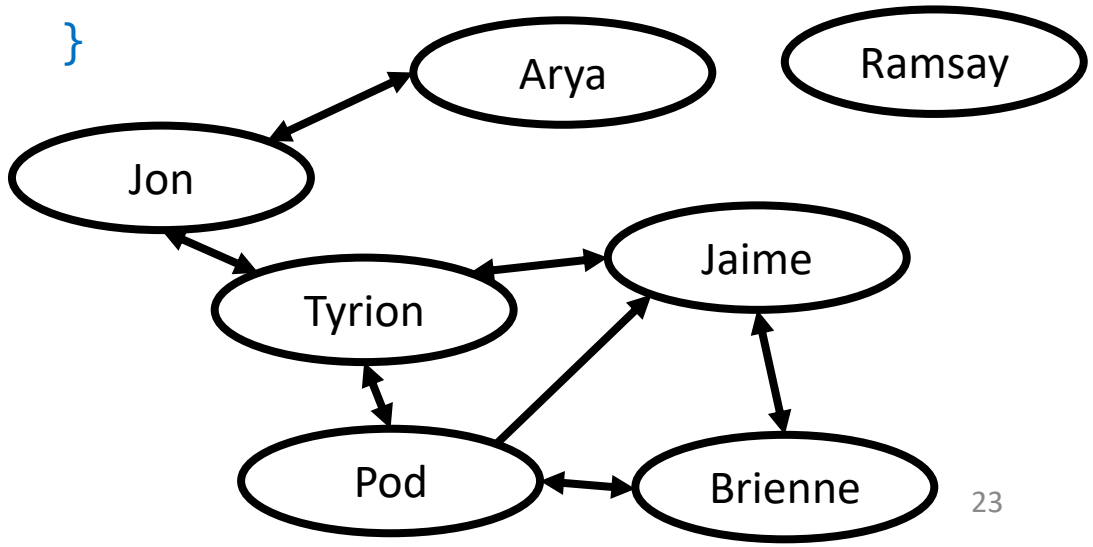
Activity: `friendsInCommon(g, p1, p2)`

You do: Given an unweighted graph of a social network (like in the previous two examples) and two nodes (people) in the graph, return a list of the friends that those two people have in common.

For example, in the graph shown to the right, calling `friendsInCommon` on `"Jon"` and `"Jaime"` would return the list `[ "Tyrion" ]`.

Hint: start by looping over all the friends of the first person. Check whether any of them are also friends of the second person and add them to a result list if they are.

```
g = { "Jon" : [ "Arya", "Tyrion" ],
      "Tyrion" : [ "Jaime", "Pod", "Jon" ],
      "Arya" : [ "Jon" ],
      "Jaime" : [ "Tyrion", "Brienne" ],
      "Brienne" : [ "Jaime", "Pod" ],
      "Pod" : [ "Tyrion", "Brienne", "Jaime" ],
      "Ramsay" : [ ]
    }
```



Learning Goals

- Identify core parts of **graphs**, including **nodes, edges, neighbors, weights,** and **directions**.

- Use **graphs** implemented as dictionaries when reading and writing simple algorithms in code