15-110 HW4 - Partial

Name:

AndrewID:

Complete the following problems in the hw4.py starter file

When you are finished, upload your hw4.py file to **Hw4 - Partial** on Gradescope. Make sure to check the autograder feedback after you submit!

Don't forget that you can get three bonus points on Hw4 by filling out the midsemester surveys! Check Piazza for links to those surveys and further instructions.

Programming Problems

For each of these problems (unless otherwise specified), write the needed code directly in the Python file, in the corresponding function definition.

All programming problems may also be checked by running 'Run File As Script' on the starter file, which calls the function `testAll()` to run test cases on all programs.

#1 - `makeIMDB(actorList, movieList)` - 5pts

Can attempt after Dictionaries lecture

Write the function `makeIMDB(actorList, movieList)` that takes two lists, a list of names of actors and a list of movie names (both strings), and returns a dictionary mapping names of actors to movies. You may assume that the two lists match up, i.e., each actor is at the same index as their movie.

You should use a **loop** to construct the dictionary. You'll need to loop over both the `actorList` and the `movieList` *at the same time* to access the key and value together. To do this, use the same loop control variable on both lists in each iteration.

If a person occurs in `actorList` multiple times (in other words, if they have multiple movies), you should map their name to the **first** movie they were paired with. For example, given the list of names:

```
    ["Ni Ni", "Sofia Vergara", "Ni Ni"]
```

and the list of movies

```
    ["Suddenly Seventeen", "Hot Pursuit", "Love Will Tear Us Apart"],
```

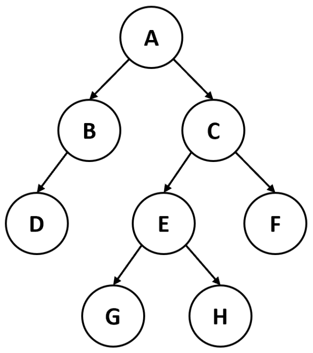
the function would return the dictionary

```
    {"Ni Ni" : "Suddenly Seventeen", "Sofia Vergara" : "Hot Pursuit" }.
```

#2 - `getLeftmost(t)` - 5pts

Can attempt after Trees lecture

Write the function `getLeftmost(t)` that takes a binary tree in our dictionary format and returns the contents of the **leftmost** child of that tree. This is the child we reach if we keep moving down and left from the root node until we cannot go left any further. For example, in the tree:



Which is represented as the dictionary:

```
t = { "contents" : "A",
      "left" : { "contents" : "B",
          "left" : { "contents" : "D", "left" : None, "right" : None},
          "right" : None },
      "right" : { "contents" : "C",
          "left" : { "contents" : "E",
              "left" : { "contents" : "G", "left" : None, "right" : None },
              "right" : { "contents" : "H", "left" : None, "right" : None } },
          "right" : { "contents" : "F", "left" : None, "right" : None } } }
```

We go from A to B, then from B to D, then we can't go left any further. `"D"` is the content of the leftmost node and is returned when we call the function on `t`.

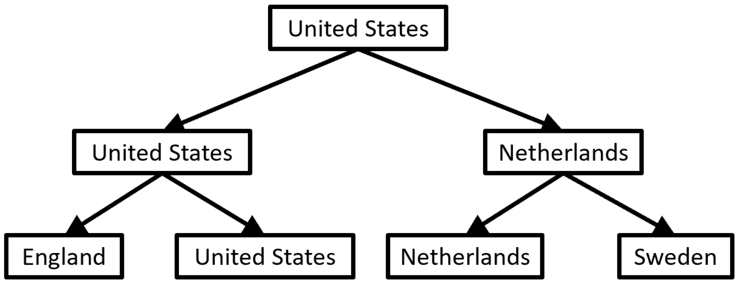
Hint: you can solve this using recursion, or you can just use a while loop.

#3 - `getInitialTeams(bracket)` - 5pts

Can attempt after Trees lecture

We can represent a tournament bracket from a sports competition as a binary tree. To do this, store the winning team as the root node. Its children are the winning team again, as well as the second-place team. In general, every node represents the winner of a match, and its two children are the two teams that competed in that match.

For example, the following bracket represents the last two rounds of the Women's World Cup in 2019.



In our binary tree dictionary format, this would look like:

```
t1 = { "contents" : "United States",
       "left" : { "contents" : "United States",
           "left"  : { "contents" : "England", "left" : None, "right" : None },
           "right" : { "contents" : "United States", "left" : None, "right" : None}},
       "right" : { "contents" : "Netherlands",
           "left"  : { "contents" : "Netherlands", "left" : None, "right" : None },
           "right" : { "contents" : "Sweden", "left" : None, "right" : None } }
     }
```

Write the function `getInitialTeams(bracket)` which takes a tournament bracket and returns a list of all the teams that participated in that tournament. For example, if the function is called on the tree above it might return `[ "England", "United States", "Netherlands", "Sweden" ]`. You will need to implement this function **recursively** to access all the nodes. We recommend that you start by looking at the `sumNodes` and `listValues` examples from the slides.

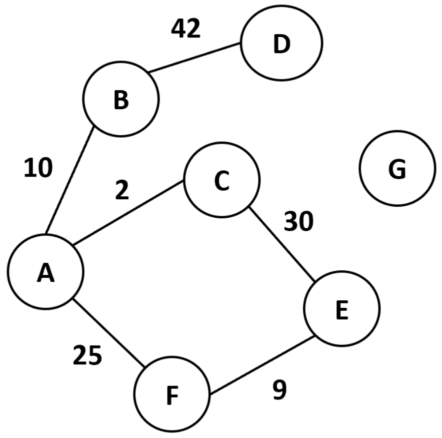
Hint 1: how can we get all of the teams to show up in the list exactly once? Every team occurs at the very beginning of the tournament, in the first set of matches. In the tree, this is represented by the **leaves,** so you should not include values on non-leaf nodes.

Hint 2: make sure the **type** you return is the same in both base and recursive cases!

#4 - `largestEdge(g)` - 10pts

Can attempt after Graphs lecture

We often want to find the **largest edge weight** in a graph. This can help us identify useful information, like the most congested street in a city or the two gas stops that are farthest apart on a highway. Write the function `largestEdge(g)` that takes a weighted graph in our dictionary format and returns a list holding two elements - the two endpoints of the edge with the largest weight in the graph. For example, in the graph:



Which is represented as the dictionary:

```
g = { "A" : [ [ "B", 10 ], [ "C",  2 ], [ "F", 25 ] ],
      "B" : [ [ "A", 10 ], [ "D", 42 ] ],
      "C" : [ [ "A",  2 ], [ "E", 30 ] ],
      "D" : [ [ "B", 42 ] ],
      "E" : [ [ "C", 30 ], [ "F",  9 ] ],
      "F" : [ [ "A", 25 ], [ "E",  9 ] ],
      "G" : [ ] }
```

The largest edge has the weight `42`. That edge is between the nodes B and D, so if we call the function on that graph, it will return `[ "B", "D" ]` (or `[ "D", "B" ]` - the order doesn't matter).

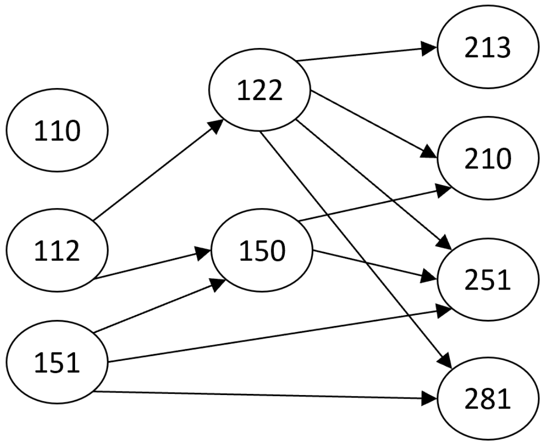
To find the largest edge, modify the find-most-common/find-largest-item pattern we've discussed several times in class. Iterate over each of the nodes in the graph, then for each node iterate over each of that node's neighbors to visit each edge.

Note: to make this easier, you are guaranteed that all edge weights will be **positive** and there will be at least one edge in the graph.

#5 - `getPrereqs(g, course)` - 5pts

Can attempt after Graphs lecture

College course prerequisites are notoriously complicated. However, we can make them a little easier to understand by representing the course dependency system as a **directed graph**, where the nodes are courses and an edge leads from course A to course B if A is a prerequisite of B. For example, the core Computer Science courses (almost) produce the following prereq graph:



Which would be represented in code as:

```
g = { "110" : [],
      "112" : ["122", "150"],
      "122" : ["213", "210", "251", "281"],
      "151" : ["150", "251", "281"],
      "150" : ["210", "251"],
      "213" : [],
      "210" : [],
      "251" : [],
      "281" : [] }
```

Write the function `getPrereqs(g, course)` that takes a directed graph (in our adjacency list dictionary format, without weights) and a string (a course name) and returns a list of all the immediate prerequisites of the given course. If we called `getPrereqs` on our graph above and `"210"`, for example, the function should return `["122", "150"]`.

Hint: you can't just return the neighbors of the course, because the edges are going in the opposite direction! Instead, iterate over all the nodes to find those that have the course as a neighbor. Construct a new list out of these nodes as the result.