# Programming Problems

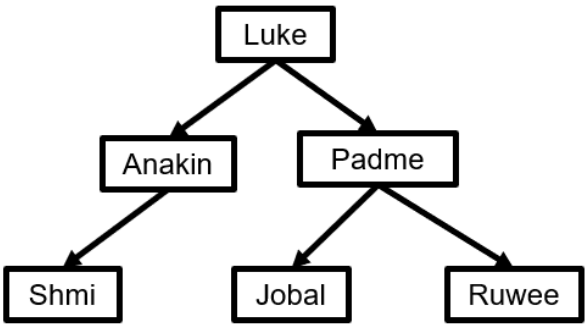
For each of these problems (unless otherwise specified), write the needed code directly in the Python file in the corresponding function definition.

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

## #1 - `getShortNames(t)` - 10pts

*Can attempt after Trees lecture*

Write the function `getShortNames(t)` that takes a binary tree in our dictionary format and returns a list of all the 'short' names that occur in the tree. We define a name as short if it is at most four characters in length. For example, given the following family tree:




Which is represented as the dictionary:

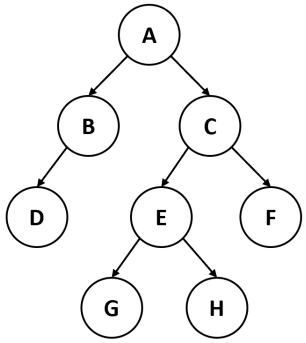
```
t = { "contents" : "Luke",
      "left"  : { "contents" : "Anakin",
                  "left"  : { "contents" : "Shmi", "left" : None, "right" : None },
                  "right" : None },
      "right" : { "contents" : "Padme",
                  "left"  : { "contents" : "Jobal", "left" : None, "right" : None },
                  "right" : { "contents" : "Ruwee", "left" : None, "right" : None }}}
```

The function would return `[ "Luke", "Shmi" ]`, as only these names are four characters or less in length. You can return the appropriate names in any order - the autograder will sort them before checking if the values are correct.

## #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`.

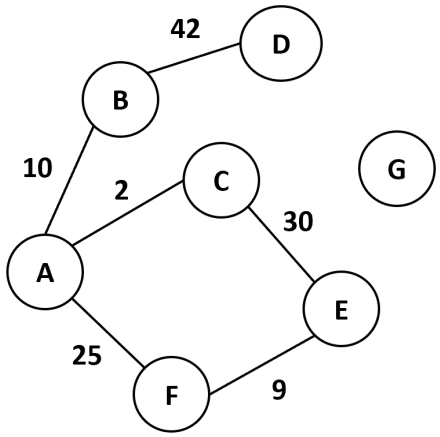
You are guaranteed that the input tree will have at least one node.

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

## #3 - `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).

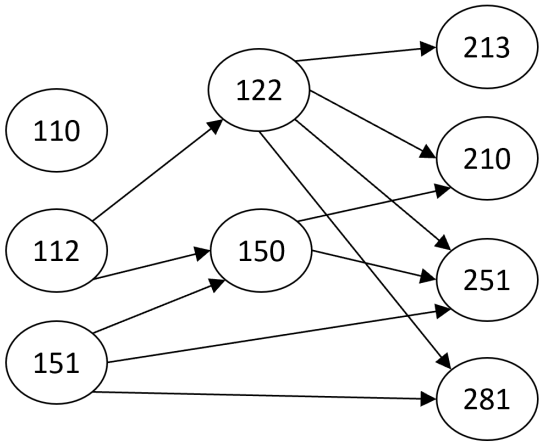
To find the largest edge, modify the find-most-common/find-largest-item pattern we've discussed several times in class. Iterate over each of the nodes in the graph, then for each node iterate over each of that node's neighbors to visit each edge.

**Note:** to make this easier, you are guaranteed that all edge weights will be **positive**, there will be at least one edge in the graph, and there won't be a tie for largest edge.

## #4 - `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.