6.009 First Practice Quiz 2 -- Spring 2017

This quiz is **open-book and closed-internet**. Feel free to use any physical materials you have brought, but you may not access resources online (except web.mit.edu/6.009). Proctors will be available to answer administrative questions and clarify specifications of coding problems, but they should not be relied on for coding help.

Solutions that hardcode answers will receieve no credit. Here, hardcoding refers to building in special code for the particular inputs and outputs from our test suite. Many normal and perfectly acceptable programming patterns lead to functions having special cases for particular inputs. For instance, recursive functions usually include base cases that check for particular inputs. In contrast, we consider it unacceptable hardcoding for one of the problems if one of your classmates who solved that problem successfully would read your code and say "huh, that's a rather long and specialized value to test the input against verbatim!".

You *must* submit your modified `quiz.py` via web.mit.edu/6.009 before the deadline in order to receive credit. This quiz assumes you have Python 3.5 (or a later version) installed on your machine.

The `resources` directory contains the **Python library documentation** in PDF form. Please **do not import any Python modules** to use as part of your solutions -- quizzes with `import` statements (or their equivalents!) will be given grades of 0.

As in the labs, `test.py` can be used to test your code in `quiz.py`. Remember that you can run specific tests by listing the test numbers, like so

```
    python test.py 1 3 5      # run tests #1, #3, #5
```

Your score will be computed from the number of tests you pass. If you pass all the tests for a problem, you'll receive full credit. If you pass 2 of the 5 tests, you'll receive 40% of the credit for the problem. Note that to pass some of the tests in the time allotted, your solution may have to be reasonably efficient.

Problem 1: `check_valid_paren` (tests 1-5)

Suppose we have a string expression that consists solely of left parenthesis `"("` and right parenthesis `")"` characters. We say that such an expression is *valid* if the following conditions are satisfied:

1. Each left parenthesis is closed by exactly one right parenthesis later in the string
2. Each right parenthesis closes exactly one left parenthesis earlier in the string

For Question 1 you will implement the `check_valid_paren` function to the specification below.

INPUTS:

1. `s` : a string expression that consists solely of `"("` and `")"` characters.

OUTPUT:

A Boolean (`True` / `False`) indicating whether `s` is *valid*.

EXAMPLES:

1. `check_valid_paren("()")` should return `True` .
2. `check_valid_paren(")")` should return `False` .
3. `check_valid_paren("())(")` should return `False` .

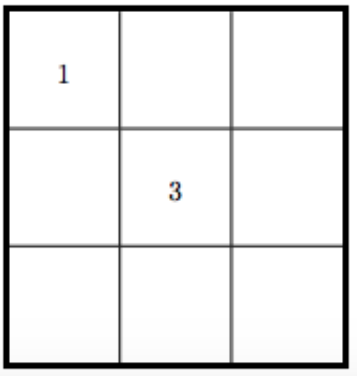
Problem 2: `solve_latin_square` (tests 6-10)

A Latin square is an `n x n` grid of numbers in which each row and each column contains the numbers `1, 2, ..., n` exactly once. Below is an example of a `3 x 3` Latin square.

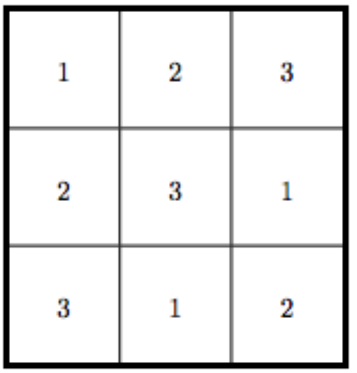
1	2	3
2	3	1
3	1	2

You are given a partially filled square with empty cells. Your task is to produce a Latin square by filling the empty cells, or determine that the task is impossible. We recommend that you perform a search over all possible values to place in the empty cells. This problem should remind you of Tent Packing lab.

Suppose you were given the following partially filled `3 x 3` square.



One way to fill in the empty squares with values of `1`, `2`, or `3` to produce a Latin square is the following.



For Question 2 you will implement the `solve_latin_square` method to the specification below.

INPUTS:

- `grid`: a list-of-lists representation of a partially filled square. `grid[r][c]` gives the value of the cell in row `r` and column `c` (grid[0][0] gives the value in the top left corner).**A value of `-1` indicates that the value is missing**-- it will be your job to fill in the values of these cells.

OUTPUT:

If a solution exists, your program should return a list-of-lists `finished_grid`, corresponding to `grid` with the empty cells filled in properly. Multiple solutions may exist -- you only need to return one.

If no solution exists, your program should return False.

EXAMPLES:

- Using the example above, one valid solution to `solve_latin_square([[1, -1, -1], [-1, 3, -1], [-1, -1, -1]])` is `[[1, 2, 3], [2, 3, 1], [3, 1, 2]]`.
- One valid solution to `solve_latin_square([[-1, -1], [-1, -1]])` is `[[1, 2], [2, 1]]`.

Problem 3: `is_proper` (tests 11-16)

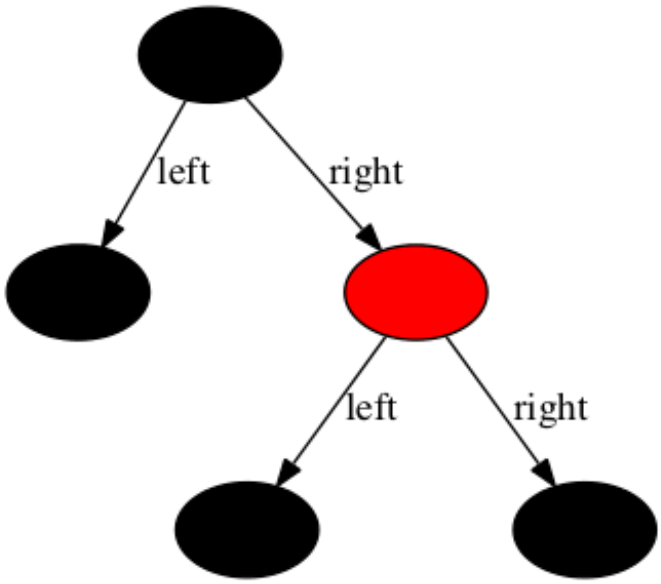
A black-red tree can store information across successive levels of nodes, starting with the root. **Each node is colored either black or red (hence the name) and points to up to 2 children: a left child and/or a right child.** The child(ren) of any node can be of either color.

We say that a black-red tree is *proper* if it satisfies the following property: **Any path from the root to a leaf contains the same number of black nodes.** Recall that a leaf is a node with no children.

We implement a black-red tree as a nested dictionary. Each node in the black-red tree is a dictionary containing:

- `"color"`: the color of the node as a string, either "black" or "red"
- `"left"`: the left child, as another dictionary, if it exists. If the node does not have a left child, then the value is `-1`.
- `"right"`: the right child, as another dictionary, if it exists. If the node does not have a right child, then the value is `-1`.

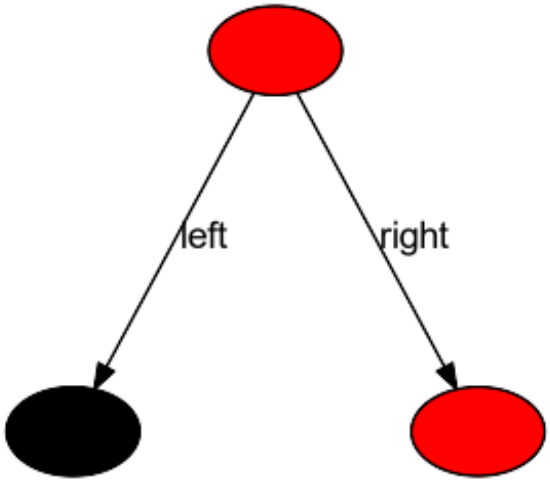
Consider the following example of a *proper* black-red tree in which any path from the root to a leaf contains exactly 2 black nodes.



`root1`, the root of this *proper* black-red tree, in our implementation, is

```
{
  "color": "black",
  "left": {
            "color":  "black",
            "left":   -1,
            "right":  -1
          },
  "right":{
            "color":  "red",
            "left":   {
                        "color":    "black",
                        "left":     -1,
                        "right":    -1
                      },
            "right":  {
                        "color":    "black",
                        "left":     -1,
                        "right":    -1
                      }
          }
}
```

In contrast, the following black-red tree is NOT *proper*. The path from the root to the left leaf contain 1 black node, while the path from the root to the right leaf contains 0 black nodes.



`root2`, the root of this second black-red tree, in our implementation, is:

```
{
  "color":  "red",
  "left":   {
              "color":  "black",
              "left":   -1,
              "right":  -1
            },
  "right":  {
              "color":  "red",
              "left":   -1,
              "right":  -1
            }
}
```

For Question 3 you will implement the `is_proper` method to the specification below.

INPUTS:

- `root` : the root of a black-red tree, as a dictionary.

OUTPUT:

A Boolean (True/False) indicating whether the black-red tree is*proper*.

EXAMPLES:

- `is_proper(root1)` with `root1` as defined above should return `True` .
- `is_proper(root2)` with `root2` as defined above should return `False` .

HINT: Many approaches can work here. One approach (might not be the simplest to implement) is to recursively find all paths from the root to a leaf. Then, you can check the number of black nodes in each path.