Quiz 2 Practice

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 funprog). Proctors will be available to answer administrative questions and clarify specifications of coding problems, but they should not be relied on for coding help.

Each problem is worth 33 1/3 points, for a total of 100 points.

You *must* submit your quiz via funprog before the deadline in order to receive credit. This quiz assumes you have Python 2.7 installed on your machine.

The `resources` directory contains **Python documentation** for commonly-used data structures.

Question 1: `check_valid_paren`

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` .

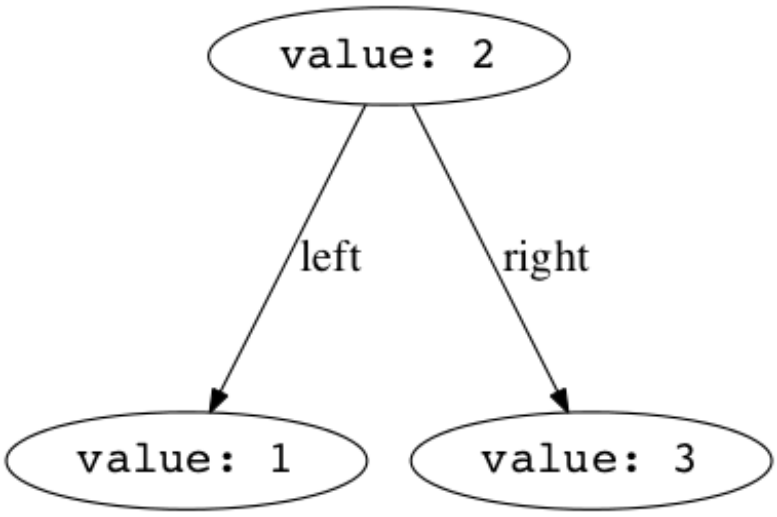
Question 2: `get_all_elements`

A binary tree can store a set of numbers across successive levels of nodes, starting with the root. Each node stores a value and points to up to 2 children: a left child and/or a right child.

Here we implement a binary tree as a nested dictionary. Each node in the binary tree is a dictionary containing:

1. `"value"` : the value stored in the node
2. `"left"` : the left child, as another dictionary, if it exists. If the node does not have a left child, then the value is `None` .
3. `"right"` : the right child, as another dictionary, if it exists. If the node does not have a right child, then the value is `None` .

Consider the following example of a simple binary tree containing three elements `[2, 1, 3]` .



The `root` of this binary tree, in our implementation, is then:

```
{
  "value":    2
  "left":     {
                "value": 1
                "left": None
                "right": None
              }
  "right":    {
                "value": 3
                "left": None
                "right": None
              }
}
```

For Question 2 you will implement the `get_all_elements` function to the specification below.

INPUTS:

1. `root` : the root of a binary tree, as a dictionary.

OUTPUT:

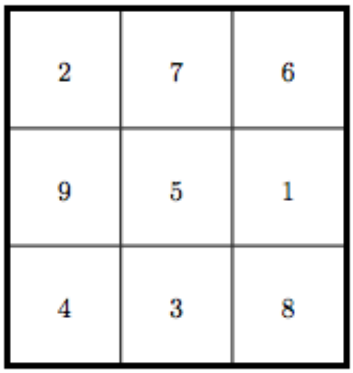
A list `L` of all numbers stored in the binary tree rooted at `root`, in any order.

EXAMPLES:

1. One valid solution to `get_all_elements(root)` is `[1, 2, 3]`, with `root` as defined in the example above.

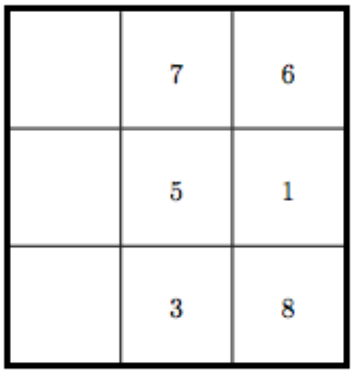
Question 3: `solve_magicsquare_recursive`

A magic square is an `n x n` grid of numbers in which each row, each column, and both diagonals add to the same "magic sum". For example, shown below is a `3 x 3` grid with a magic sum of 15.



2	7	6
9	5	1
4	3	8

Suppose we were only given the partially filled `3 x 3` square without `2`, `9`, or `4` filled in:

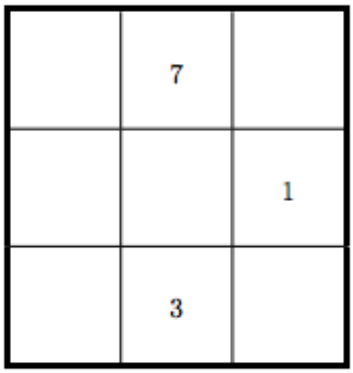


	7	6
	5	1
	3	8

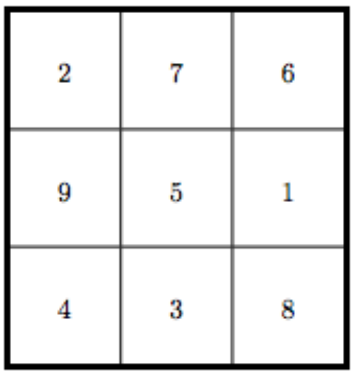
If we were told that the magic sum is 15, we would be able to fill in the proper values to place in the empty cells, through a series of "deductions". A "deduction" involves examining a row/column/diagonal and inferring a proper cell value by utilizing `magic_sum`. For example, we deduce that `2` belongs in the top-left cell by calculating 15 - 6 - 7 = 2.

In addition to the grid, you will now be given `choices`, a list of numbers that you can use to fill the empty cells. **The same number can be used to fill multiple empty cells.**We recommend that you perform a search over the possible values to put in the empty cells. This should remind you of the recursive solutions you implemented in your labs.

For example, consider the following partially filled square.



Given that `magic_sum = 15` and `choices = [1, 2, 3, 4, 5, 6, 7, 8, 9]`, then one possible solution is the following, where we have filled in the empty cells with values in `choices`.



Implement the `solve_magicsquare_recursive` function to the specification below. You are guaranteed that a valid solution exists.

INPUTS:

1. `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`. **A value of `-1` indicates that the value is missing** – it will be your job to fill in the values of these cells.
2. `magic_sum`: the magic sum of the completed magic square.
3. `choices`: a list of numbers that can be used to fill the missing values. Each number in `choices` can be used to fill multiple empty cells.

OUTPUT:

A list-of-lists `finished_grid`, corresponding to `grid` with the empty cells filled in properly.

An empty cell can only be filled in with a number in `choices`. Multiple solutions may exist – you only need to return one.

EXAMPLES:

1. Using the example above, one valid solution to `solve_magicsquare_recursive([[-1, 7, -1], [-1, -1, 1], [-1, 3, -1]], 15, [1, 2, 3, 4, 5, 6, 7, 8, 9])` is `[[2, 7, 6], [9, 5, 1], [4, 3, 8]]`.
2. One valid solution to `solve_magicsquare_recursive([[-1, -1], [-1, -1]], 4, [1, 2, 3, 4])` is `[[2, 2], [2, 2]]`.