6.009 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) or access files on your computer outside of the quiz directory. 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 test case is worth one point for a total of 20 points.

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 Language Reference** and the **Python Library Reference** in PDF form. Please **do not import any Python modules** to use as part of your solutions -- quizzes with `import` statements (or their equivalent!) will be given a grade of 0.

As in the labs, `test.py` can be used to test your code in `quiz.py`. The problems have different numbers of tests as specified. Remember that you can run specific tests by listing the test numbers, like so

```
python3 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.

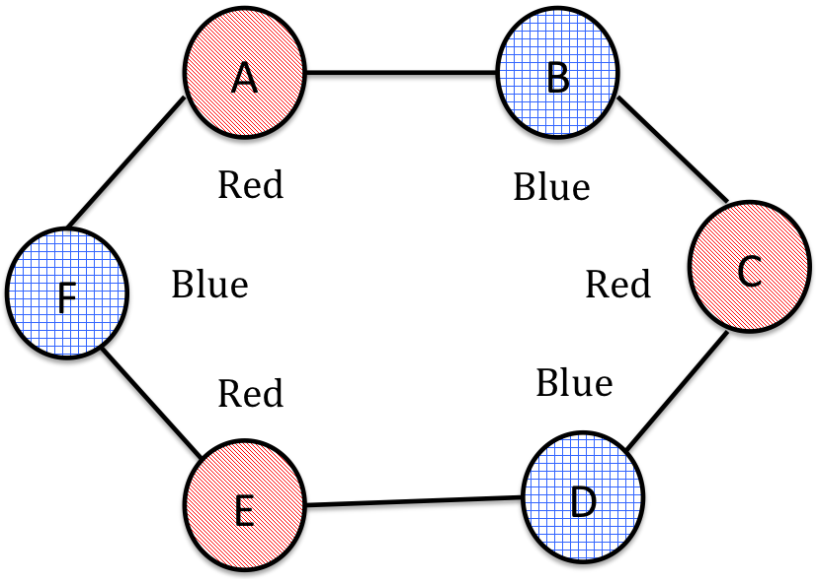
Problem 1: `alternating_colors( graph, start )` (tests 1-6)

`graph` is a graph represented as a dictionary. Your task is to check whether the graph can be colored using two colors with the constraint that no two vertices that have an edge between them can have the same color. `start` is the vertex from which you should start coloring. You can assume that all vertices in the graph are reachable from `start`.

If the graph is not 2-colorable, your function should return an empty dictionary `{}`. If the graph is 2-colorable, your function should return a dictionary of key:value pairs that represent a valid 2-coloring respecting the coloring constraint. Each key corresponds to a vertex in the graph and each value is one of `'Red'` or `'Blue'`. All vertices reachable from vertex `start` should

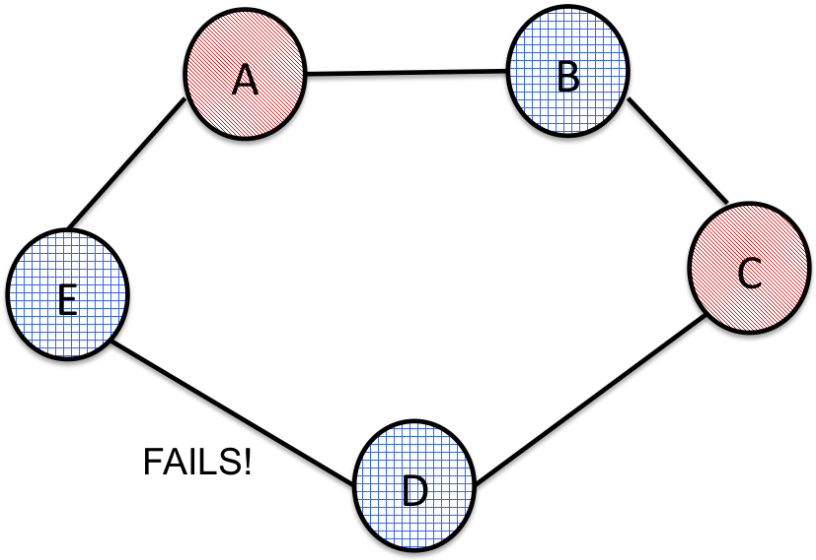
be included as keys.

Examples of function invocation and return are given below the pictures of graphs.



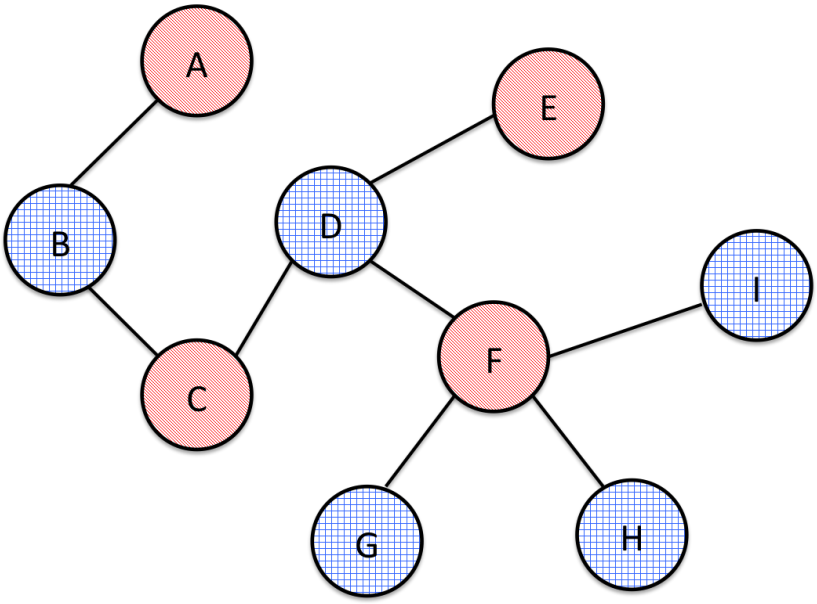

```
alternating_colors({'A': ['B', 'F'],
                    'B': ['C', 'A'],
                    'C': ['D', 'B'],
                    'D': ['C', 'E'],
                    'E': ['D', 'F'],
                    'F': ['A', 'E']}, 'A')
```

could return `{'A': 'Red', 'B': 'Blue', 'C': 'Red', 'D': 'Blue', 'E': 'Red', 'F': 'Blue'}` or a different valid coloring with `'Red'` and `'Blue'` interchanged.



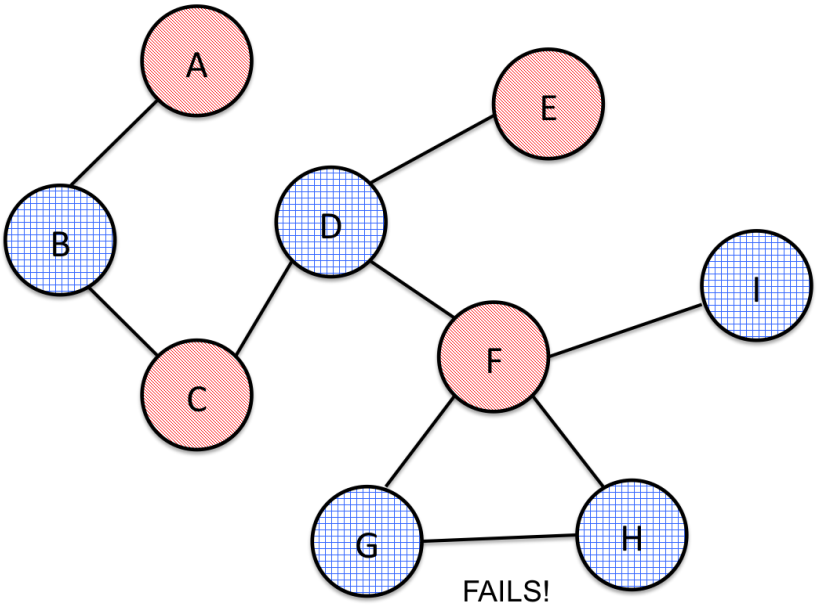
```
alternating_colors({'A': ['B', 'E'],
                    'B': ['C', 'A'],
                    'C': ['D', 'B'],
                    'D': ['C', 'E'],
                    'E': ['A', 'D']}, 'A')
```

should return `{}` since there is no valid coloring.




```
  alternating_colors({'A': ['B'],
                      'B': ['A', 'C'],
                      'C': ['B', 'D'],
                      'D': ['C', 'E', 'F'],
                      'E': ['D'],
                      'F': ['D', 'G', 'H', 'I'],
                      'G': ['F'],
                      'H': ['F'],
                      'I': ['F']}, 'A')
```

could return `{'A': 'Red', 'B': 'Blue', 'C': 'Red', 'D': 'Blue', 'E': 'Red', 'F': 'Red', 'G': 'Blue', 'H': 'Blue', 'I': 'Blue'}` or a different valid coloring with `'Red'` and `'Blue'` interchanged.




```
alternating_colors({'A': ['B'],
                    'B': ['A', 'C'],
                    'C': ['B', 'D'],
                    'D': ['C', 'E', 'F'],
                    'E': ['D'],
                    'F': ['D', 'G', 'H', 'I'],
                    'G': ['F', 'H'],
                    'H': ['F', 'G'],
                    'I': ['F']}, 'A')
```

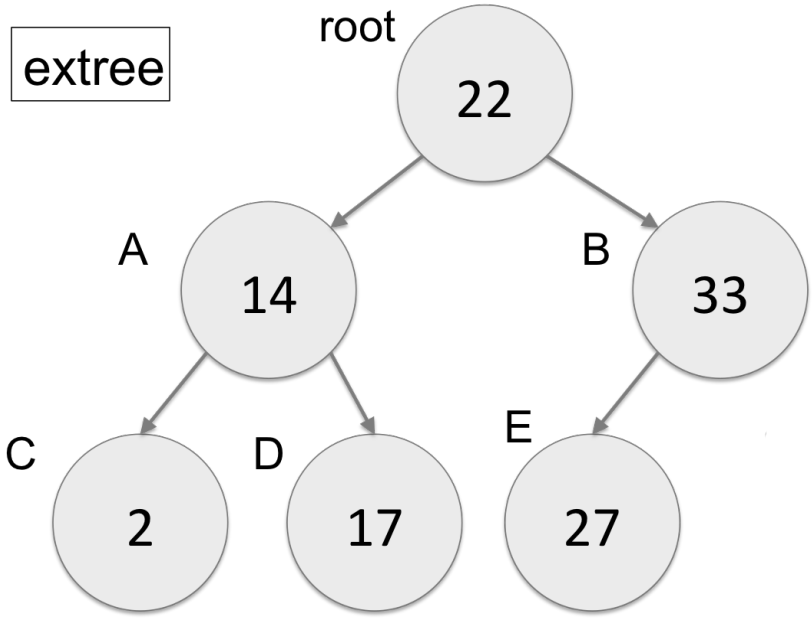
should return `{}` since there is no valid coloring.

Problem 2: `check_BST( btree, start )` (tests 7-12)

Assume that you are given a binary tree. Your task is to write a function to check whether it is a Binary Search Tree (BST). Recall that a BST has the following property: Each vertex has a value and the value of any vertex in the left subtree has to be less than the value of the vertex and the value of any vertex in the right subtree has to be greater than the value of the vertex. You can assume all values in any given tree are unique.

We will use the same dictionary representation for a binary tree or BST that we used in Lecture 5.

```
extree = {'root': [22, 'A', 'B'],
            'A': [14, 'C', 'D'],
            'B': [33, 'E', ''],
            'C': [2, '', ''],
            'D': [17, '', ''],
            'E': [27, '', '']}
```



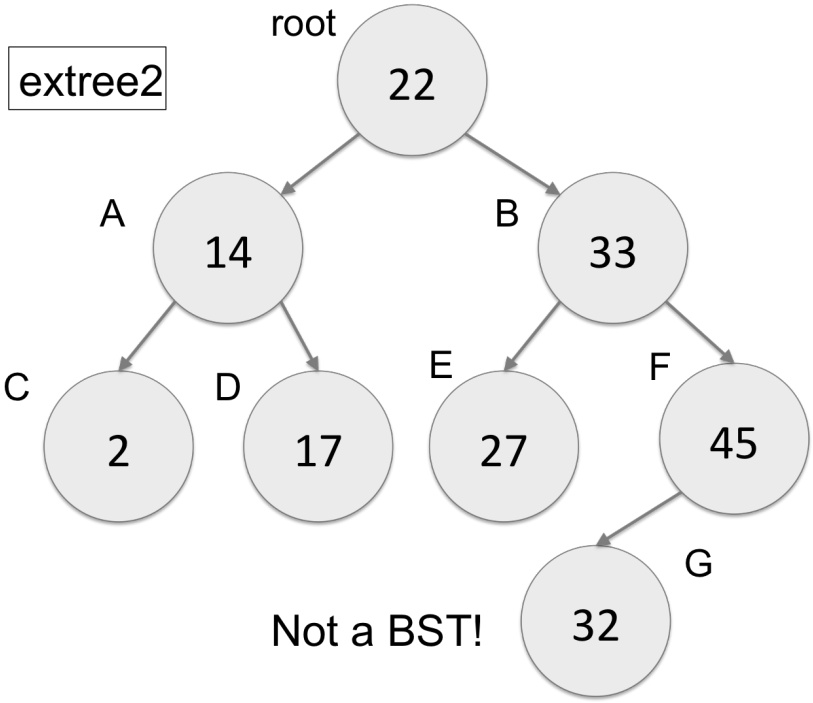
Examples:

`check_BST({'root': [22, '', '']}, 'root')` should return `True`.

`check_BST(extree, 'root')` should return `True`. `extree` corresponds to the example in the picture above.

```
check_BST({'root': [22, 'A', 'B'],
              'A': [14, 'C', 'D'],
              'B': [33, 'E', 'F'],
              'C': [2, '', ''],
              'D': [17, '', ''],
              'E': [27, '', ''],
              'F': [45, 'G', ''],
              'G': [32, '', '']},
          'root')
```

should return `False`, since vertex `G` with value 32 is in the right subtree of the vertex `B`, which has the value 33. The BST property needs to hold recursively throughout the tree, and it does not as shown below.



Note that in order to receive full credit, your code should work on binary trees with arbitrary depth, not for just the test cases.

Problem 3: `pipe_cutting` (tests 13-20)

You are tasked with buying plumbing pipes for a construction project. Your foreman gives you a list of the varying lengths of pipe needed. The local hardware store sells pipes in one fixed length but has a saw for you to cut up the pipes in any way you choose. Your job is to figure out the minimum number of stock pipes required to satisfy the list of requests, in order to save money and minimize waste.

Please implement `pipe_cutting(requests, stock_length)` where `requests` is a list of pipe lengths needed for the project and `stock_length` is the length of the pipes you can purchase at Home Depot. `pipe_cutting` should return an integer giving the minimum number of purchased pipes needed to satisfy the list of requests.

You can assume that all elements in `requests` are positive numbers that are no longer than the stock pipe length. So, in the worst case, it will take a number of stock pipes equal to the size of the request list. You do not need to report the cutting/division in the optimal solution, just report the minimum number of stock pipes needed.

There are different ways of solving this problem. We recommend, but do not require, a recursive approach for which the hint below will be useful.

Hint: For any particular request, the request might be satisfied by choosing a left-over pipe length or cutting from any left-over pipe lengths, or by choosing to buy a new pipe and cutting it to length.

EXAMPLES:

- `pipe_cutting([],8) = 0`
- `pipe_cutting([7],7) = 1`
- `pipe_cutting([7,6,4],10) = 2` // eg, cuts are `[7],[6,4]`
- `pipe_cutting([4,3,4,1,7,8],10) = 3` // eg, cuts are `[4,4,1],[3,7],[8]`