BMI 2005 Homework #3
Due March 6, 2019

Homework must be submitted to Blackboard using a Jupyter notebook!

1) Suppose we have numbers between `1` and `1000` in a BST and we search for the number `363`. Which (at least one) of the following sequences could not be the sequence of nodes examined?
   ```
   a) 2, 252, 401, 398, 330, 344, 397, 363
   b) 924, 220, 911, 244, 898, 258, 362, 363
   c) 925, 202, 911, 240, 912, 245, 363
   d) 2, 399, 387, 219, 266, 382, 381, 278, 363
   e) 935, 278, 347, 621, 299, 392, 358, 363
   ```

2) Define a hash table with an associated hash function `h(k)` mapping keys `k` to to their associated hash value.
   a) Suppose we wish to search a linked list of length `n`, where each element contains a key `k` along with a hash value `h(k)`. Each key is a long character string. In words, how might we take advantage of the hash values when searching the list for an element with a given key?
   b) Consider a hash table of size `m = 1000` and a corresponding hash function `h(k) = floor(m (kA mod 1))` for `A = (sqrt(5) - 1)/2)`. This type of divisive hashing has been shown to yield effective randomly distributed hashes. Note that `mod 1` means take the fractional component of `kA` i.e. `kA - floor(kA)`.
   Compute the hashes for `keys = {61, 62, 63, 64, 65}`.
   c) In simple uniform hashing, each key is assumed to have equal probability to map to any of the hashes in a given table of size `m`. Given an open-address table of size `100` and `2` random keys, what is the probability that they hash to the same value? What is the probability that they hash to different values?

3) Binary Search Trees are defined by each parent having at most 2 child nodes and maintain the invariant that all descendents to the left of a node are smaller than the node, while all descendants to the right are greater for all subtrees.
   a) What are all the possible valid BSTs drawn from the array [1, 2, 3]? How many are there?
   b) In lab we explored the number of possible binary tree topologies given a height of n and n nodes (spoiler: 2^{n-1}). It turns out that the number of binary tree topologies (given all possible heights) with `n` nodes is the **Catalan Number** C_n `= 1/(n+1)*(2n choose n)`. Recursively, this can also be written as $C_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$ with $c_0 = 1$ and $n >= 1$.

i) Write a function `cn_recursive(n)` that computes the n-th catalan number recursively.
ii) Write a function `cn_fast(n)` that computes the n-th catalan number using the closed form formulation.
iii) Conduct doubling experiments (just n = 1, 2, 4, and 8) to compare their runtimes. Plot the result on a log-log plot. What are the first 8 Catalan Numbers?

4) As discussed in lab and lecture, binary search trees derive a lot of their power from being appropriately balanced. Here we will show that the search time for random keys in a BST containing uniformly randomly distributed values is `c lg n` where `n` is the amount of nodes in the tree and `c = 1.39` - note that this is lg base 2. In greater detail:
   a) Create a dict mapping keys `(N = 100, 200, 400, ... 51200)` to arrays with linear entries derived from random sampling from a uniform distribution [0, 100000].
   b) Store entries in a binary search tree - the class implementation from lab is provided below, with a small modification to count the number of compares performed in each recursion for `find()`. You may find it helpful to similarly construct an analogous dict in (a) with the keys mapping to BST instances.
   c) For each array, compute the average time of 1000 keys drawn at random (from the linear array corresponding to each tree).
   d) What is your estimated constant `c` for each tree size?

```
class Node:
    def __init__(self, val):
        self.val = val
        self.leftChild = None
        self.rightChild = None

    def get(self):
        return self.val

    def set(self, val):
        self.val = val

    def getChildren(self):
        children = []
        if(self.leftChild != None):
            children.append(self.leftChild)
        if(self.rightChild != None):
            children.append(self.rightChild)
        return children
```

```
class BST:
    def __init__(self):
        self.root = None

    def setRoot(self, val):
        self.root = Node(val)

    def insert(self, val):
        if(self.root is None):
            self.setRoot(val)
        else:
            self.insertNode(self.root, val)

    def insertNode(self, currentNode, val):
        if(val <= currentNode.val):
            if(currentNode.leftChild):
                self.insertNode(currentNode.leftChild, val)
            else:
                currentNode.leftChild = Node(val)
        elif(val > currentNode.val):
            if(currentNode.rightChild):
                self.insertNode(currentNode.rightChild, val)
            else:
                currentNode.rightChild = Node(val)

    def find(self, val, count = 0):
        return self.findNode(self.root, val, count)

    def findNode(self, currentNode, val, count):
        if(currentNode is None):
            return False, count + 1
        elif(val == currentNode.val):
            return True, count + 1
        elif(val < currentNode.val):
            return self.findNode(currentNode.leftChild, val, count+1)
        else:
            return self.findNode(currentNode.rightChild, val,
count+1)

    def traverse(self):
        if self.root is not None:
            self.inorder_traverse(self.root.leftChild)
```

```
        print(self.root.val)
        self.inorder_traverse(self.root.rightChild)

def inorder_traverse(self, Node):
    if Node.leftChild is not None:
        self.inorder_traverse(Node.leftChild)
    print(Node.val)
    if Node.rightChild is not None:
        self.inorder_traverse(Node.rightChild)
```