UDACITY

PROJECT

Technical Interview Practice (Python)

PROJECT REVIEW

CODE REVIEW 6

NOTES

interview.py 6

```
1  def getCharCount(s):
2      charCount_s = {}
3      for char in s:
```

SUGGESTION

This actually takes O(t), and you are doing this on each iteration. There's a clever way to calculate this... Ins
count each time, simply remove the last characters and add the new one...

```
4          if char not in charCount_s:
5              charCount_s[char] = 1
6          else:
7              charCount_s[char] += 1
8      return charCount_s
9
10 def anagramTest(s, t):
11     if type(s) != str or type(t) != str or len(s) == 0 or len(s) != len(t):
12         #print('False: input error')
13         return False
14     else:
15         charCount_s = getCharCount(s)
16         charCount_t = getCharCount(t)
17         if charCount_s == charCount_t:
18             print('True: '+s+" is an anagram of "+t)
19             return True
20         return False
```

```
# QUESTION1: Given two strings s and t, determine whether some anagram of t is a subst
# For example: if s = "udacity" and t = "ad", then the function returns True.
# Your function definition should look like: question1(s, t) and return a boolean True
# Given 2 strings s and t, is some anagram of t a substring of s?
def question1(s, t):
    #check that s and t are strings, and s is longer than t
    if type(s) != str or type(t) != str or len(s) < len(t):
        print('False: Improperly formatted input')
        return False
    else:
        #check if t contains letters not found in s
        for char in t:
            if char not in s:
                print('False: t contains letters not in s')
                return False
        #for each len(t) long substring of s, run anagram test
        for i in range(0, len(s)):
            if anagramTest(t, s[i:i+len(t)]):
                print("True: "+s[i:i+len(t)]+" is an anagram of "+t)
                return True
        print("False, no anagrams found")
        #if all anagram tests fail, return false
        return False

def testQ1():
    print("Q1 Test1: expected outcome True")
    question1("udacity", "ad")
    print("Q1 Test2: expected outcome False")
    question1("udacity", "aj")
    print("Q1 Test3: expected outcome False")
    question1("udacity", 2)
#s = s[ beginning : beginning + LENGTH]
testQ1()

def isPalindrome(s):
    return s[::-1] == s

# QUESTION2: Given a string a, find the longest palindromic substring contained in a.
# Your function definition should look like question2(a), and return a string.
def question2(s):
    lps = ""
    #check if s is a string
    if type(s) != str:
        print("Error: non-string input")
        return "Error: non-string input"
    elif len(s)<2:
        print("True: input length is 1 or 0")
        return s
    else:
        length = len(s)
        substrings = []
        for x in range(0, length):
            for y in range(x, length):
                if isPalindrome(s[x:y + 1]) and len(s[x:y + 1]) > len(lps):
                    lps = s[x:y + 1]
        print("lps is "+lps)
        return lps
```

AWESOME

Looks great!

```
79
80 def testQ2():
81     print("Q2 test 1, should print 'lps is racecar'")
82     question2("driver racecarsdsadasgfdhgfsdsadsfgfdgjhgkguliuseweadsdadfghf")
83     print("Q2 test 2, should print 'lps is dad'")
84     question2("dad173123")
85     print("Q2 test 3, should print 'Error: non-string input'")
86     question2(1)
87
88 testQ2()
89 # QUESTION 3: Given an undirected graph G, find the minimum spanning tree within G.
90 # A minimum spanning tree connects all vertices in a graph with the smallest possible
91 # Find minimum spanning tree of a graph. Vertices are represented as unique strings.
92 #  The function definition should be question3(G)
93
94 ### isGraph function takes in a dictionary and determines whether it fits the format (
95 def isGraph(G):
96     if type(G) != dict:
97         #print("input is not dict")
98         return False
99     else:
100        for key in G:
101            if type(key) is not int:
102                return False
103            if isinstance(type(G[key]), list):
104                return False
105            else:
106                for i in range(0, len(G[key])):
107                    if type(G[key][i]) is not tuple:
108                        return False
109    return True
110
111 ##a nice, reasonably complex graph to test with. Source http://www.geeksforgeeks.org/g
112 graph1 = {
113     0:[(1,4),(7,8)],
114     1:[(2,8),(0,4),(7,11)],
115     2:[(1,8),(3,7),(5,4),(8,2)],
116     3:[(2,7),(5,14),(4,9)],
117     4:[(3,9),(5,10)],
118     5:[(4,10),(3,14),(2,4),(6,2)],
119     6:[(8,6),(5,2),(7,1)],
120     7:[(6,1),(8,7),(1,11),(0,8)],
121     8:[(7,7),(6,6),(2,2)]
122 }
123
124 graph1MST = {
125     0:[(1,4),(7,8)],
126     1:[(0,4)],
127     2:[(3,7),(5,4),(8,2)],
128     3:[(2,7),(4,9)],
129     4:[(3,9)],
130     5:[(2,4),(6,2)],
131     6:[(5,2),(7,1)],
132     7:[(6,1),(0,8)],
133     8:[(2,2)]
134 }
135
136 graph2 = {1: [(2, 2)],
```

```
 1: [(1, 2), (3, 5)],
 3: [(2, 5)]}

graph2MST = {1: [(2, 2)],
 1: [(1, 2), (3, 5)],
 3: [(2, 5)]}

def question3(G):
    #check that the input is a properly formatted graph adjacency tree
    if not isGraph(G):
        print("The input graph is not properly formatted")
        return False
    #get node set
    nodes = G.keys()
    #get edge set
    edges = set()
    for x in nodes:
        for y in G[x]:
            if x > y[0]:
                edges.add((y[1], y[0], x))
            elif x < y[0]:
                edges.add((y[1], x, y[0]))
    # sort edges
    edges = sorted(list(edges))
    # loop through edges and store only those which do not create cycles with disjoin
    mst_edges = []
    x = 0
    nodes = list(nodes)
    for node in nodes:
        nodes[x] = set([node])
        x += 1
    for x in edges:
        # get indices of both nodes
        for y in range(0, len(nodes)):
            if x[1] in nodes[y]:
                x1 = y
            if x[2] in nodes[y]:
                x2 = y
        # Store union in the smaller index and pop the larger. Append edge to mst_edg
        if x1 < x2:
            nodes[x1] = set.union(nodes[x1], nodes[x2])
            nodes.pop(x2)
            mst_edges.append(x)
        if x1 > x2:
            nodes[x2] = set.union(nodes[x1], nodes[x2])
            nodes.pop(x1)
            mst_edges.append(x)
        # break loop when all nodes are in one graph
        if len(nodes) == 1:
            break
    #  put mst in proper format
    mst = {}
    for x in mst_edges:
        if x[1] in mst:
            mst[x[1]].append((x[2], x[0]))
        else:
            mst[x[1]] = [(x[2], x[0])]
        if x[2] in mst:
            mst[x[2]].append((x[1], x[0]))
        else:
            mst[x[2]] = [(x[1], x[0])]
```

```
198     return mst
```

AWESOME

Awesome!

```
199
200 def testQ3():
201     ##Test case 1, input graph with cycles
202     for key in list(graph1.keys()):
203         for edge in question3(graph1)[key]:
204             if edge not in graph1MST[key]:
205                 print("Q3 Test1 (Graph with cycles): fail")
206         else:
207             print("Q3 Test 1 (Graph with cycles): pass")
208     ##Test case 2, input graph with no cycles
209     for key in list(graph2.keys()):
210         for edge in question3(graph2)[key]:
211             if edge not in graph2MST[key]:
212                 print("Q3 Test2 (Graph without cycles): fail")
213         else:
214             print("Q3 Test2 (Graph without cycles): pass")
215     ##Test case 3, non graph input
216     if not question3(0):
217         print("Q3 Test3 (non-graph input): Pass")
218     else:
219         print("Q3 Test3 (non-graph input): Fail")
220 testQ3()
221 # Question 4:Find least common ancestor (LCA) of two nodes in a binary search tree.
222 # The least common ancestor is the farthest node from the root that is an ancestor of
223 def findChildren(n):
224     children = []
225     x = 0
226     for each in n:
227         if each == 1:
228             children.append(x)
229         x +=1
230     return children
231 print("findChildren: "+str(findChildren([0,0,1,1])))
232
233 def findRight(n):
234     children = findChildren(n)
235     return children[-1]
236
237 def findLeft(n):
238     children = findChildren(n)
239     return children[0]
240
241 print("Find Right: "+ str(findRight([0,0,1,1])))
242 print("Find Left: "+ str(findLeft([0,0,1,1])))
243
244 def question4(m, r, n1, n2):
```

SUGGESTION

When determining space complexity, only consider the space that your algorithm consumes, ignore the sp

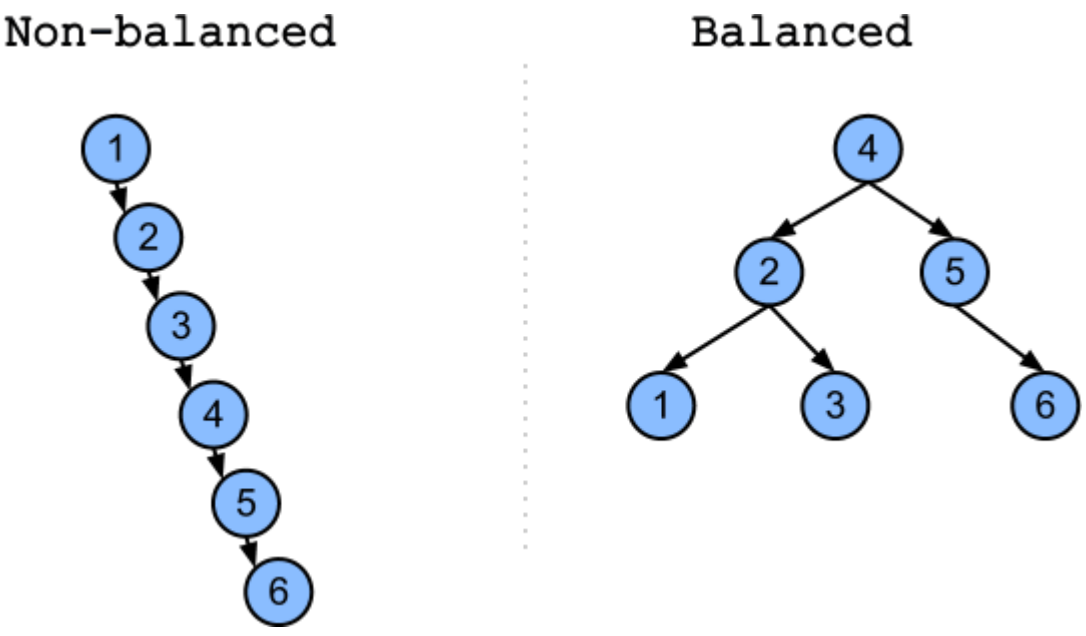
```
    nodeIndex = r
    root = m[nodeIndex]
    # make sure n1 and n2 are integers
    if type(n1) != int:
        return "n1 not int"
    if type(n2) != int:
        return "n2 not int"
    #Traverse tree starting at root
    current_node = root
    print("Node: "+str(current_node))
    while findLeft(current_node) != None or findRight(current_node) != None:
        try:
            # if the current node is greater than both n1 and n2, go left
            if nodeIndex > n1 and nodeIndex > n2:
                nodeIndex = findLeft(current_node)
                current_node = m[nodeIndex]
            # if the current node is less than both n1 and n2, go left
            elif nodeIndex < n1 and nodeIndex < n2:
                nodeIndex = findRight(current_node)
                current_node = m[nodeIndex]
            # If the current node is between n1 and n2, the current node is the lca
            else:
                return nodeIndex
        except:
            break
    return nodeIndex
```

SUGGESTION

RUNTIME

The runtime of your algorithm will be partly based on the height of your tree. When determining the runti of the tree is in terms of `n`. There are two possible scenarios I'd like you to consider:

1. A Balanced Tree O(log(n))
2. An Unbalanced Tree O(n)



```
####Chain together node objects to construct a tree for test purposes
def test4():
    print("Q4 Test 1: LCA is root (should return 3)"+"\n"+str(question4([[0, 1, 0, 0,
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [1, 0, 0, 0, 1],
        [0, 0, 0, 0, 0]],
        3,
        1,
        4)))

    print("Q4 Test 2: LCA is left of root (should return 2) "+"\n"+str(question4([
        [0,0,0,0,0,0],
        [1,0,0,0,0,0],
        [0,1,0,1,0,0],
        [0,0,0,0,0,0],
        [0,0,1,0,0,1],
        [0,0,0,0,0,0]],
        4,
        1,
        3)))

    print("Q4 Test 3: LCA is right of root (should return 4): "+"\n"+str(question4([
        [0,0,0,0,0,0],
        [1,0,0,0,0,0],
        [0,0,0,0,0,0],
        [0,1,0,0,1,0],
        [0,0,1,0,0,1],
        [0,0,0,0,0,0]
        ],
        3,
        4,
        5)))

test4()

#Question 5: Find the element in a singly linked list that's m elements from the end.
# if a linked list has 5 elements, the 3rd element from the end is the 3rd element.
#  The function definition should look like question5(ll, m), where ll is the first no
#  and m is the "mth number from the end"
class Node(object):
    def __init__(self, value):
        self.value = value
        self.next = None

#### String together a linked list: ["one", "two", "three", "four", "five", "six", "se

n1 = Node("one")
n2 = Node("two")
n3 = Node("three")
n4 = Node("four")
n5 = Node("five")
n6 = Node("six")
n7 = Node("seven")
n8 = Node("eight")
n9 = Node("nine")
n10 = Node("ten")

n1.next = n2
n2.next = n3
n3.next= n4
```

```
n4.next = n5
n5.next = n6
n6.next = n7
n7.next = n8
n8.next = n9
n9.next = n10

def findLength(n):
    x = 1
    currentNode = n
    while currentNode.next != None:
        currentNode = currentNode.next
        x += 1
    return x

print(str(findLength(n1)))

def question5(n, m):
    if type(n) != Node:
        return "n is not a node object"
    if type(m) != int:
        return "m is not int"
    lengthList = findLength(n)
    currentNode = n
    x = 0
    while x < lengthList - m - 1:
        currentNode = currentNode.next
        x += 1
    return currentNode.value

```

AWESOME

Great!

```
def testQ5():
    print("Q5 test1: m=6: expected outcome: 'four'")
    print(str(question5(n1, 6)))
    print("Q5 test2: m=0: outcome: 'ten'")
    print(str(question5(n1, 0)))
    print("Q5 test 3, n is not node. expected outcome: 'n is not a node object'")
    print(str(question5(1, 1)))
    print("Q5 test 4, m is not int. expected outcome: 'm is not int'")
    print(str(question5(n1, "1")))

testQ5()

```

RETURN TO PATH

Rate this review

Student FAQ