Project Prep

Question 1
Given two strings s and t, determine whether some anagram of t is a substring of s. 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 or False.

Efficiency
==========
1.) The code scans all characters present in both the string s and t and creates dictionary. This operations is linear in the size of s and t.
2.) During parsing of the string, the code checks to see if a particular character is already in the dictionary. This operation is in constant time as dictionaries are implemented as hash-maps in python.
3.) Next, when bothe the dictionaries have been created, the code parses through characters in t and figures out if that character is present in the dictionary of s. This is again in linear time with the size of t and lookups from dictionary happen in constant time as dictionaries are implemented as hash-maps in python.
4.) The dictionaries take the space proportional to the size of the string

The overall performance of the algorithm in time is: O(s+t)
The overall performance of the algorithm in space is: O(s+t)

Question 2
Given a string a, find the longest palindromic substring contained in a. Your function definition should look like question2(a), and return a string.

Efficiency
==========
1.) The code calculates the length of the string. This operation is linear in size of string a.
2.) Next, for a fixed length substr_len, the algorithm calculates all possible number of substrings. This is linear in size of string a
3.) The algorithm then calculates if the substring is a palindrome, which is linear in size of string(substr_len)
4.) Steps 2 and 3 when combined have overall complexity that is quadratic to the size of string(substr_len)
5.) Steps 2 and 3 are performed for all substring lengths from len(str) to 2. Hence, the overall complexity of the algorithm is cubic to the size of string

The overall performance of the algorithm in time is: O(a^3)
The overall performance of the algorithm in space is: O(a)

Question 3
Given an undirected graph G, find the minimum spanning tree within G. A minimum spanning tree connects all vertices in a graph with the smallest possible total weight of edges. Your function should take in and return an adjacency list structured like this:

{'A': [('B', 2)],
 'B': [('A', 2), ('C', 5)],
 'C': [('B', 5)]}
Vertices are represented as unique strings. The function definition should be question3(G)

Algorithm:
==========
Prim's algorithm has been implemented in the above code to find minimum spanning tree
The algorithm is as follows:
a.) Select a randon node n0, put it in visited list
b.) Get list of edges, nodes that n0 is connected to
c.) Choose smallest edge that connects to an unvisited node(new)
d.) Add node(new) to visited list
e.) Now look at nodes reachable from nodes in visited
f.) Pick smallest edge that connects to unvisited node
*Do not pick an edge between nodes already present in visited list
g.) The algorithm stops when all nodes have been visited

Implementation:
===============
First node from the graph is chosen and put in visited array.
An edge_list array is populated that lists all edges going out from the nodes present in the visited array
The shortest_edge from the above list is selected and added to the mst
The get_shortest_edge function goes through all edges in the list and picks one which has the smallest edge weight and does not connect nodes already in the visited list
The function add_to_mst adds the shortest edge to the minimum spanning tree variable mst

Efficiency
==========
1.) Calculating edges_list: Each vertex can connect to maximum of |V|-1 edges, making the complexity O(|V|^2)
2.) Find shortest edge: This can be performed in linear time
3.) Add to mst: Constant time

The overall performance of the algorithm in time is: O(|V|^2)
The overall performance of the algorithm in space is: O(|V|^2)

Question 4
Find the least common ancestor between two nodes on a binary search tree. The least common ancestor is the farthest node from the root that is an ancestor of both nodes. For example, the root is a common ancestor of all nodes on the tree, but if both nodes are descendents of the root's left child, then that left child might be the lowest common ancestor. You can assume that both nodes are in the tree, and the tree itself adheres to all BST properties. The function definition should look like question4(T, r, n1, n2), where T is the tree represented as a matrix, where the index of the list is equal to the integer stored in that node and a 1 represents a child node, r is a non-negative integer representing the root, and n1 and n2 are non-negative integers representing the two nodes in no particular order. For example, one test case might be

```
question4([[0, 1, 0, 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)
```

and the answer would be 3.

Algorithm:
==========
1.) A tree is constructed from the matrix representing a tree
2.) The tree is parsed from root to calculate the least common ancestor as follows:
   a.)if node.value equals one of the two input nodes, then node is lca
   b.)if node value lies between the values in the two input nodes, then node is lca
   c.)if both values are greater than node.value, parse the right subtree
   d.)if both values are less than node.value, parse the left subtree

Implementation:
===============
Given a matrix representing a tree, first a tree is constructed
add_nodes_to_tree function starts fron root node, figures out the child nodes, adds them to tree
Then the above function performs a depth first technique to populate the tree
The function find_lca calculates the least common ancestor by performing binary search

Efficiency:
===========
1.) add_nodes_to_tree function: complexity is O(nodes)
2.) find_lca: complexity: O(log(nodes))

The overall performance of the algorithm in time is: O(nodes)
The overall performance of the algorithm in space is: O(log(nodes))

Question 5
Find the element in a singly linked list that's m elements from the end. For example, 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 node of a linked list and m is the "mth number from the end". You should copy/paste the Node class below to use as a representation of a node in the linked list. Return the value of the node at that position.

```
class Node(object):
  def __init__(self, data):
    self.data = data
    self.next = None
```

Algorithm:
==========
1.) find length n of linked list
2.) mth node from end is the n-(m-1)=n-m+1 node from beginning

Efficiency:
==========

The overall performance of the algorithm in time is: O(n)
The overall performance of the algorithm in space is: O(1)