```python
#Submission for Technical Interview Questions
#5/12/13


# Question 1 ---------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#                    for loop.  However, since it is a "set" and not a string this
#                    function will run in O(n) time.
#
#Space Efficiency:  The will use up only the space needed for s and t.  O(n).
#
# Code Design:  Code was designed to run efficiently and is easy to understand.
#               The code was designed to try to cover all possible inputs and
#               still function.


def question1(s,t):
    #These must be converted to strings because the input could be a number.
    s = str(s)
    t = str(t)

    if t == "":
        return True

    slist = list(s)
    tlist = list(t)
    tn = len(tlist)

    for i in range(0,tn):
        if tlist[i] in slist:
            slist.remove(tlist[i])
        else:
            return False

    return True

#Testing the function
question1("udacity","udazx")    #Expected Output: False
question1("udacity","ad")       #Expected Output: True
question1(None,"ad")            #Expected Output: False
question1("udacity",None)       #Expected Output: False
question1("udacity","")         #Expected Output: True
question1("udacity",5)          #Expected Output: False
question1("uda5city",55)        #Expected Output: False
question1("uda55city",55)       #Expected Output: True
question1("hello","hallo")      #Expected Output: False
question1("aabbcc","abc")       #Expected Output: True


# Question 2 ---------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#               two for loops.  The first loop will not add a significant amount
#               of time to the function because it will only loop a few times.
#               This function will run in O(n^2) time.
#
#Space Efficiency:  The will use up only the space needed for a and the matrix created
#                   with the combinations function. O(n^2)
#
# Code Design:  Using Dynamic programming to find the solution.

def question2(a):
    #Modified from:
    http://www.geeksforgeeks.org/dynamic-programming-set-12-longest-palindromic-subsequence/
```

```
    if a == "":
        return "(Empty String)"
    elif a is None:
        return None

    #variable a must be converted to strings because the input could be a number.
    a = str(a)

    a_length = len(a)
    pal_table = [[0 for x in range(a_length)] for x in range(a_length)]

    for i in range(a_length):
        pal_table[i][i] = a[i]

    for substr_len in range(2, a_length+1):
        for i in range(a_length-substr_len+1):
            j = i+substr_len-1
            if a[i] == a[j] and substr_len == 2:
                pal_table[i][j] = a[i] + a[j]
            elif a[i] == a[j]:
                pal_table[i][j] = a[i] + pal_table[i+1][j-1] + a[j]
            else:
                if len(pal_table[i][j-1]) > len(pal_table[i+1][j]):
                    pal_table[i][j] = pal_table[i][j-1]
                else:
                    pal_table[i][j] = pal_table[i+1][j]

    return pal_table[0][a_length-1]



#Testing the function
question2("character")  #Expected Output: carac
question2("a")          #Expected Output: a
question2("abc")        #Expected Output: c
question2(None)         #Expected Output: (None)
question2(5)            #Expected Output: 5
question2("")           #Expected Output: (Empty String)

# Question 3 ----------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#                 two for loops.  This function will run in O(n*m) time.  Where n
#                 is the number of nodes and m is the number of connections.
#
#Space Efficiency:  The will use up only the space needed for G.  O(1)
#
# Code Design:  Functions and classes were used to make the code easier
#                 to read.
#

def question3(G):
    from copy import deepcopy
    Gnew = Graph()

    for key , value in G.iteritems():
        Gnew.add_node(key)
        for val in value:
            Gnew.add_edge(key,val[0], key = val[1])

    #edges = _spanning_edges(Gnew)

    subtrees = UnionFind()
    edges = Gnew.edges()
```

```
    getweight = lambda t: t[-1].get('weight', 1)
    edges = sorted(edges, key=getweight)

    edges = Gen(edges, subtrees)

    T = Graph(edges)

    for n in T:
        T.node[n] = deepcopy(Gnew.node[n])
    T.graph = deepcopy(Gnew.graph)

    return T


def Gen(edges, subtrees):
    for u, v, d in edges:
        if subtrees[u] != subtrees[v]:
            yield (u, v, d)
            subtrees.union(u, v)

#Modified from: https://github.com/networkx/networkx
#Not using networkx classes directly.

class Graph(object):

    node_dict_factory = dict
    adjlist_dict_factory = dict
    edge_attr_dict_factory = dict

    def __init__(self, data=None, **attr):

        self.node_dict_factory = ndf = self.node_dict_factory
        self.adjlist_dict_factory = self.adjlist_dict_factory
        self.edge_attr_dict_factory = self.edge_attr_dict_factory
        self.graph = {}
        self.node = ndf()
        self.adj = ndf()
        if data is not None:
            if self is None:
                self = Graph()
            else:
                self.clear()
                self.add_edges_from(data)
        self.graph.update(attr)
        self.edge = self.adj

    def __iter__(self):
        return iter(self.node)

    def __getitem__(self, n):
        return self.adj[n]

    def add_node(self, n, **attr):
        if n not in self.node:
            self.adj[n] = self.adjlist_dict_factory()
            self.node[n] = attr
        else:
            self.node[n].update(attr)

    def nodes(self):
        for n in self.node:
            yield n

    def number_of_nodes(self):
        return len(self.node)
```

```
    def add_edge(self, u, v, **attr):
        if u not in self.node:
            self.adj[u] = self.adjlist_dict_factory()
            self.node[u] = {}
        if v not in self.node:
            self.adj[v] = self.adjlist_dict_factory()
            self.node[v] = {}

        datadict = self.adj[u].get(v, self.edge_attr_dict_factory())
        datadict.update(attr)
        self.adj[u][v] = datadict
        self.adj[v][u] = datadict

    def add_edges_from(self, ebunch, **attr):
        for e in ebunch:
            ne = len(e)
            if ne == 3:
                u, v, dd = e
            elif ne == 2:
                u, v = e
                dd = {}

            if u not in self.node:
                self.adj[u] = self.adjlist_dict_factory()
                self.node[u] = {}
            if v not in self.node:
                self.adj[v] = self.adjlist_dict_factory()
                self.node[v] = {}
            datadict = self.adj[u].get(v, self.edge_attr_dict_factory())
            datadict.update(attr)
            datadict.update(dd)
            self.adj[u][v] = datadict
            self.adj[v][u] = datadict

    def neighbors(self, n):
        return iter(self.adj[n])

    def edges(self):
        seen = {}
        nodes_nbrs = self.adj.items()

        for n, nbrs in nodes_nbrs:
            for nbr, ddict in nbrs.items():
                if nbr not in seen:
                    yield (n, nbr, ddict)
            seen[n] = 1
        del seen

    def clear(self):
        self.name = ''
        self.adj.clear()
        self.node.clear()
        self.graph.clear()

class UnionFind:

    def __init__(self, elements=None):

        if elements is None:
            elements = ()
        self.parents = {}
        self.weights = {}

    def __getitem__(self, object):
```

```
        if object not in self.parents:
            self.parents[object] = object
            self.weights[object] = 1
            return object

        path = [object]
        root = self.parents[object]
        while root != path[-1]:
            path.append(root)
            root = self.parents[root]

        for ancestor in path:
            self.parents[ancestor] = root
        return root

    def union(self, *objects):
        roots = [self[x] for x in objects]

        heaviest = max(roots, key=lambda r: self.weights[r])
        for r in roots:
            if r != heaviest:
                self.weights[heaviest] += self.weights[r]
                self.parents[r] = heaviest

#Testing the function
val = question3({'A':[('B',2)],'B':[('A',2),('C',5)],'C':[('B',5)]})
print(sorted(val.edges()))  #Expected Output: [('A', 'B', {'key': 2}), ('C', 'B', {'key': 5})]

val = question3({'A':[('B',3)],'B':[('A',3),('C',10)],'C':[('B',10)]})
print(sorted(val.edges())) #Expected Output: [('A', 'B', {'key': 3}), ('C', 'B', {'key': 10})]

val = question3({'A':[('B',2)],'B':[('A',2)]})
print(sorted(val.edges())) #Expected Output: [('A', 'B', {'key': 2})]


# Question 4 ---------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#               5 for loops.  This function will run in O(n*m) time.  Where n
#               is the number of nodes and m is the number of connections.  Each
#               will have to run twice.  Once to generate the tree and the second
#               time to fill in the information about where in the tree that node
#               is.
#
#Space Efficiency:  The will use up only the space needed for T,r, n1, and n2.
#                   G, n, and m will be assigned during the call.  This will
#                   result in O(2+n).  The n is for G and the 2 is for n and m.
#
# Code Design:  Classes were used to make the code easier to read.

def question4(T, r, n1, n2):

    G = Graph()
    G.add_node(r)

    n = len(T)
    m = len(T[0])

    for i in range(0,n):
        for j in range(0,m):
            if T[i][j] == 1:
                G.add_edge(i,j)

    for a in G.nodes():
```

```
        G.node[a]['Level']= None
    G.node[r]['Level']= 0

    m = G.number_of_nodes()
    for L in range (0,m):
        for n in G.nodes():
            if G.node[n]['Level'] == L:
                L1 = G[n]
                for a in L1.iteritems():
                    if G.node[a[0]]['Level'] == None:
                        G.node[a[0]]['Level'] = L+1

    Gsucc=G.neighbors
    pred={n1:None}
    succ={n2:None}
    forward_fringe=[n1]
    reverse_fringe=[n2]

    while forward_fringe and reverse_fringe:
        this_level=forward_fringe
        forward_fringe=[]
        for v in this_level:
            for w in Gsucc(v):
                if w not in pred:
                    forward_fringe.append(w)
                    pred[w]=v
                if w in succ:
                    results = pred,succ,w

    pred,succ,w = results
    path=[]

    while w is not None:
        path.append(w)
        w=pred[w]
    path.reverse()
    w=succ[path[-1]]
    while w is not None:
        path.append(w)
        w=succ[w]

    path.pop(0)
    path.pop(len(path)-1)

    LCALevel = G.node[n1]['Level']
    for n in path:
        val = G.node[n]['Level']
        if val < LCALevel:
            LCALevel = G.node[n]['Level']
            LCA = n

    return LCA

#Testing the function
val = 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)
print val        #Expected Output: 3


val = question4([[0,0,0,0,0],[0,0,0,0,0],[0,1,0,1,1],[0,0,0,0,0],[1,0,0,0,0]],4,3,1)
print val        #Expected Output: 2

val = question4([[0,0,0,0,1],[0,0,1,0,0],[0,1,0,1,1],[0,0,1,0,0],[1,0,1,0,0]],4,3,1)
print val        #Expected Output: 2
```

```python
# Question 5 ------------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#                 while loop and the for loop.  This function will run in O(n) time.
#
#Space Efficiency:  The will use up only the space needed for ll and m.  O(1)
#
# Code Design:  The linked list is cycled through to find the end.  It is then
#                 cycled through to the m position away from the end.

def question5(ll, m):

    next_item = ll.get_next()
    count = 1

    while next_item.get_next() is not None:
        next_item = next_item.get_next()
        count += 1

    selected = ll
    for i in range(0, count-m):
        selected = selected.get_next()

    if m < 0:
        raise ValueError("Verify Inputs")
    m = count-m
    if m < 0:
        raise ValueError("Verify Inputs")

    val = selected.get_data()

    return val

#Source for Node and LinkedList Classes:
https://www.codefellows.org/blog/implementing-a-singly-linked-list-in-python
class Node(object):

    def __init__(self, data=None, next_node=None):
        self.data = data
        self.next_node = next_node

    def get_data(self):
        return self.data

    def get_next(self):
        return self.next_node

    def set_next(self, new_next):
        self.next_node = new_next

class LinkedList(object):

    def __init__(self, head=None):
        self.head = head

    def front_insert(self, data):
        new_node = Node(data)
        new_node.set_next(self.head)
        self.head = new_node


#Create a Linked List to input into question 5
LL = LinkedList()
LL.front_insert("E")
LL.front_insert("D")
```

```
LL.front_insert("C")
LL.front_insert("B")
LL.front_insert("A")

#Testing the function
question5(LL.head, 0)        #Expected Output: E
question5(LL.head, 1)        #Expected Output: D
question5(LL.head, 2)        #Expected Output: C
question5(LL.head, 3)        #Expected Output: B
question5(LL.head, 4)        #Expected Output: A
question5(LL.head, 5)        #Expected Output: Error - Verify Inputs
question5(LL.head, -1)       #Expected Output: Error - Verify Inputs
```