Solutions for Assignment 2

Q1

Solution one. Keep extra information in the node of the stack:
Assume Stack is a normal Stack class, defined somewhere. We push a pair to the stack on every push: the value, and the current min,

```
Class stack:

       def __init__(self):
              self.stack = Stack()

      def push(value):
           new_min = min(value, min());
           pair= (value, new_min)
           self.stack.push(pair)

      def min():
           If is_empy(s):
                Return None
           return self.stack.peek()[1]
      def peek():
           return self.stack.peek()[0]
      def pop():
           return self.stack.pop()[0]
```

This solution wastes a lot of space, a better solution is to use 2 stacks. This will be discussed in the tutorial.

Q2.

There are many solutions for this problem. However, I want to introduce one which uses a nice trick. You have to trace the code several times, on different linked lists, to see why it works! This technique, known as fast and slow runner, can be used in other similar problems:

It uses two pointers, one **fast**, and one **slow**. It works in two steps:
Step1. Let fast iterates the nodes one by one, and fast two nodes a time. If they meet, there is a loop
When they meet after k steps, they both are s - k steps into the linked list where s is the length of the loop and k is the length of the non-loop part of the list.
Step2. Move one of them, slow for example, and let both run again, this time with the same pace.
Where they meet is the starting point of the loop. The “corrupted” node is the node before it, in the loop, which we need to do another traversal to find (we could keep a pointer while we are iterating over the list for the first time, but if the loop starts right from the head, that would fail)

```
Node slow = linked_list.head;
Node fast = linked_list.head;
# Step 1.
while (fast != Null and fast.next != null):
     slow = slow.next;
     fast = fast.next.next;
     If slow == fast then: // Collision
          break;
if fast == null or fast.next == null:
     return null;

# Step 2.
slow = head;
while slow != fast :
     slow = slow.next;
     fast = fast.next;
# Step 3. Now fast and slow are pointing to the start of the loop,
and we want the node before it
corrupted = fast.next
while corrupted.next != fast:
     corrupted = corrupted.next
return corrupted
```

Q3.

There are many solutions for this problem, I will write the simplest one, and you think about the hardest case!

Case 1: We have parent attribute, or at least are allowed to set parents

Setting parents can easily be done in TravereseFromVeretex, simply add a line after line 11:

    w.tree.parent = v.tree

Then, finding Least Common Ancestor (or in Wordnet, called Least Common Subsumer) can be done like this:

```
Def find_lcs(x,y):
    n = x
    While n is not root:
        n.visited = true
        N = n.parent
    n = y
    While n is not root:
        If n.visited = true:
            return n
        n = n.parent
```

The complexity of this algorithm (ignoring set_paretns) is O(h) where h is the height of the tree.

This solution is bad for many reasons:

1. If the nodes do not already have parent, it sets the parent for the whole tree, we could do it only for x and y
2. It goes all the way back to root from x, while it could be smarter, for example keep two pointers for x and y, and move up one by one

A better approach can be discussed in the class.

Q4. If we want to find a path from source, to destination, we modify the TravereseFromVeretex algorithm discussed in the class, by adding two lines after line 11:

1. Set the parent:

```
W.tree.parent = v.tree
```

2. Check if we met the destination:

```
If w == destination:
      Found = True
      break
```

3. We then call it from source, and at the end if found is True, we can traverse back from w.tree to the root using parent attribute, and print the path out in the reverse order:

This easily can be done by a recursive function, try to complete it, and if you had a problem, ask in the tutorial:

```
def recrusive_print_path_to_root(s):
        #... complete this method
```