Nathan Newbury Hw3 Write-up

Problem 1

The Euler tour algorithm is as follows

```
ef euler(node,layer):

        # add node to visited list (only added once ) and euler tour
        #visited.append(node.get_label())
        E.append(node.get_label())
        # count to keep track of when node was visited in euler tour
        count[0] = count[0] + 1
        # layer is incremented when we look at child nodes
        layer = layer + 1
        for child in node.child_nodes():
            # check for self referential
            #if child not in visited:
            if child.get_label() not in F:
                F[child.get_label()] = count[0]
                H[child.get_label()] = layer
            #add children recursively
            euler(child,layer)
            # in each euler call, also add parents after recursively
            E.append(node.get_label())
            # increment nodes visited after adding to euler tour
            count[0] = count[0]+1
```

The count array keeps track of the number of vertices visited. The layer keeps track of what layer we are on. Each time we append E, or add to our euler tour, we increment count. Before we move to the child nodes, we increment the layer. In each call of euler, we first append to the euler tour. Then we call euler recursively for each child. This adds each child recursively to our euler tour in a similar fashion to a depth first search. If we stopped here, this would hit all of the nodes once. Finally, after calling euler for each of the child nodes, we add the current node to the euler tour again. This is because the euler tour has to return backwards on the incident edge. For each node, we also use our layer and count variables to create our H and F matrices.

This algorithm complexity is done in linear time O(|E|), the number of edges. The algorithm time complexity analysis is similar to a DFS. The number of recursive calls is exactly equal to the number of tree edges (like in a DFS). By using dictionaries with constant lookup time, each vertex lookup sub problem is completed in constant time. This means the problem only scales with the number of edges.

```
nnewbury@DESKTOP-P86I0OM:~/hw3-s23-nnewbury092423$ python3 autocheck_euler.py
.........
----------------------------------------------------------------------
Ran 9 tests in 27.596s

OK
```

Problem 2:

The LCA algorithm with sparse table preloading is as follows:

```
def find_LCAs(T,Q):
# find LCA for each of the list of nodes in Q
# return the label of the LCA node of each query

    def __query__(q):

        # finding the two nodes farthest apart
        indices = []
        for elem in q:
            indices.append(F[elem])


        R = max(indices)
        L = min(indices)


        # creating sparse table indices
        i = int(log2(R - L + 1))

        # finding solution preloaded in the sparse table
        option2 = [H[st[L][i]], H[st[R - (2**i) + 1][i]]]
        min_index = min(enumerate(option2), key=lambda x: x[1]) [0]
        if min_index == 0:
```

```
            minimum = st[L][i]
        else:
            minimum = st[R - (2**i) + 1][i]
        # returning preloaded soltuion
        return minimum
    E,F,H = euler_tour(T)


    # allocate space for the sparse table
    MAX = len(E) + 1
    st = [[0 for i in range(MAX)]
                 for j in range(MAX)]

    # The first entries range of 1
    for i in range(0, len(E)):
        st[i][0] = E[i]


    # preload solutions into sparse table
    j = 1
    while 2**j <= len(E):
        i = 0
        while i + 2**j -1 < len(E):

            # find the minimum of our ranges dynamically, using previous
range minimum solutions
            option =[H[st[i][j-1]], H[st[i + 2**(j-1)][j-1]]]
            min_index = min(enumerate(option), key=lambda x: x[1]) [0]
            if min_index == 0:
                st[i][j] = st[i][j-1]
            else:
                st[i][j] = st[i + 2**(j-1)][j-1]
            i = i+ 1
        j = j+1


    # initialize LCA array
    LCAs = []
    # call our query function  after preloading our solutions
    for q in Q:
```

```
        lca = __query__(q)
        LCAs.append(lca)
    return LCAs
```

This algorithm turns the least common ancestor problem into just a range minimum query (RMQ) using the euler tour of our tree. The least common ancestor of two nodes ,u and v, is the node between u and v in the euler tour with the minimum height. With an efficient way to find the minimum within this range, we can make many least common ancestor queries in a short period of time. A sparse table can be used to preload the solutions to range minimum queries. In the code above, the sparse table is st and is indexed with i and j. Each entry of the sparse table (i, j) is a solution to a range minimum problem. Within our nested while loop, we dynamically fill out our range sparse table:

```
while 2**j <= len(E):
        i = 0
        while i + 2**j -1 < len(E):

  St[i][j] = min(st[i][j-1],st[i + 2**(j-1)][j-1])
```

It would be two computationally expensive to compute the minimum of every permutation of ranges, however the sparse table is clever. In a similar fashion to converting a number to base 2, the sparse table precomputes all answers for range queries with power of 2 length. For example, if I needed to compute the minimum range query of [8 25], I can split the range into a union of ranges of a power of two. In our example this would be [8,16]U [17,21]U[22,24] U[25,25]. If we already precomputed the solution to these range problems, we can just take the minimum of this union (4 values), to find our solution. In our query. After we load our sparse table, we call our query function to find the least common ancestor for multiple queries. In our query function, we just need to find the minimum of our range unions, like above. This is done with the code:

```
option2 = [H[st[L][i]], H[st[R - (2**i) + 1][i]]]
        min_index = min(enumerate(option2), key=lambda x: x[1]) [0]
        if min_index == 0:
            minimum = st[L][i]
        else:
            minimum = st[R - (2**i) + 1][i]
```

The minimum of our query is the node that is the least common ancestor and is returned and added to our list of LCAS.

The time complexity to create the sparse table is O(n*log(n)) where n is the length of our euler tour. This is because we fill n rows and log(n) columns. See:

```
hile 2**j <= len(E):
        i = 0
        while i + 2**j -1 < len(E):
```

The variable i ranges to len(E), where j ranges to log(len(E)). In my algorithm I preallocated space for n rows and n columns, so our table wastefully uses O(n^2) memory. Once the sparse table is loaded up,each RMQ has a time complexity of O(c) which makes it invaluable for a multiple query problem. The time complexity is constant, as we only need to take one minimum:

```
min(st[L][i], st[R - (2**i) + 1][i])
```

The autochecker tests for this algorithm are shown below:

```
OK
nnewbury@DESKTOP-P86I0OM:~/hw3-s23-nnewbury092423$ python3 autocheck_lcas.py
.......
```

All of the autocheck tests did not run in time. The discrepancy is likely due to the sparse table being a list with lookup time = O(n^2)

Problem 3:

The main recursive algorithm for this problem is shown below:

```
if node.is_leaf():
            distrec[node]  = 0
            paths[node.get_label()] = []
        else:
            # best is the longest distance path so far
            best =0
            # biurn is recording which child should be added to our path
            biurn  = {}
            for children in node.child_nodes():
                # adding the edge lengths incident on these children, find
the one longest distance path so far
                if best <= distrec[children] +  children.edge_length:
                    best = distrec[children] +  children.edge_length
                    biurn = children
            # the longest distance from our node is then recorded in our
distrec list
```

```
            distrec[node] = best
            # this path is also recorded
            paths[biurn.get_label()].append(biurn.get_label())
            paths[node.get_label()] = paths[biurn.get_label()]
```

If the node is a leaf, obviously the distance will be 0, and the path will be empty. The postorder traversal sees all of the leaves first, and these entries in our dictionaries are initialized. If the node is not a leaf, we find the distance of the edges incident on the children. This edge length added with the distance recorded to that child is the total distance to our node. We pick the longest distance and record it as in our dictionary for that node. We also have to record the path. To do this, we append the chosen child to its path array and store it in our paths dictionary at our node key. If the node is the root node, we end the for loop. Most of the time, the root node is part of our diameter path. In this case, we need to add the two longest distance children together for the total path. Additionally, we need to add the root node and its children, with the paths of the children for the total diameter path. The code can be seen below:

```
 if node.is_root():
            dists =[]
            biurn  = []
            # in case there are more than 2 children/paths coming into
root, we only want to add 2 paths
            for children in node.child_nodes():
                dists.append(distrec[children] +  children.edge_length)
                biurn.append(children)



            # build path from our paths array, add the two largest
distances together
            indices = sorted(range(len(dists)),key=lambda index:
dists[index])
            d = dists[indices[-1]] + dists[indices[-2]]
            front = paths[biurn[indices[-1]].get_label()]
            n1 = biurn[indices[-1]].get_label()
            r = node.get_label()
            n2 = biurn[indices[-2]].get_label()
            middle = [n1, r, n2]
            back =paths[biurn[indices[-2]].get_label()]

            # we have to reverse the second path.
```

```
            P = front + middle +back[::-1]
```

This algorithm currently does not work if the root node is not part of the diameter path.

Here is some proof of correctness if the root node is part of the diameter path:

```
test_1_100_diameter.txt - Notepad
File   Edit   Format   View   Help
A173 A171 A170 A169 A149 A148 A44 A0 A1 A2 A3
9.176361716481384
```

```
--> Newest frame
(Pdb) P
['A173', 'A171', 'A170', 'A169', 'A149', 'A148', 'A44', 'A0', 'A1', 'A2', 'A3']
(Pdb) print(d)
9.176361716481384
(Pdb)
```

The time complexity of this algorithm should be O(|E|) with constant lookup time.We loop through every node with the initial for loop. The nested for loop through the children nodes represent multiple edges coming out of one node. For each sub problem, we only sum and take maximums which can be done in constant time.