William Argus
A12802324
ECE 208 Homework 3

Part 1

Description and Proof of correctness:

This function operates with four sections. In the first section, a for loop iterates through all nodes in the tree and finds the ancestors for each one. It does this by traversing in preorder and adding each node that it comes by to the running list of ancestors, and for each node, popping of the most recent element until it reaches its parent of the root. This way is valid because of the order that things are traverses in preorder. It also saves the values of the branch lengths of all the ancestors so that the distance from any node to the root can be easily found. In the second section, all the nodes in the tree are iterated through again, and using the ancestor list to sum the values, from the values dictionary created in the previous loop, of all the branch lengths of a nodes ancestors, the node furthest from the root is found. This is the starting node for the diameter. In the third for loop, the diameter is found. This is done by looping through the nodes, and getting the list of ancestors for the starting node and the list of ancestors for the current node, then using sets to only duplicate nodes between the two sets, and then remove those duplicate nodes from each list of ancestors before combining the two lists to create a list of the nodes between the starting node the current node. Then the dictionary of values and ancestors is used to sum the branch lengths of all nodes between the two nodes, giving the distance between the two nodes. The node that has the longest distance from the start node once the loop has ended is the end of the diameter path, and its list of nodes between it and the starting node, as well as the distance are saved. The final for orders the list of nodes that go from start node to end node, including the start and end node to create the path that is to be returned. This loop starts at the starting node makes that the current node, and grabs the node in the path list that is either the parent or child of the current node, adding to the path. This way all the nodes are added to the path in order, which is then returned, along with the path length.

Time-complexity analysis:

This algorithm was required to run in linear time. It the runtime will be described to show that the program does indeed run in linear time. By showing that the function computer_diameter only uses single for loops (not nested ones) to traverse through the tree, it will be shown that there is linear run time. The first for loop traverses through all the nodes in preorder, keeping a running list of ancestors and popping off nodes until either the given nodes parent or the root node is reached, in which case the remaining list is assigned to that node as its ancestors in the ancestor dictionary, to be used later. By using the traverse in preorder function, this loop visits each node once and therefore runs in linear time.

The next loop traverses through each node once again to find the deepest node in the tree, which is that one that the diameter will start from. The next loop finds the diameter by traversing the tree and comparing the distance from each node to the starting node by merging the respective ancestor lists from each of those two node, removing the duplicate nodes, and then summing the branch lengths for that list, returning the distance between the start node and current node. By doing this in a loop that visits each node once, the node furthest from the starting node, the path, and the distance is found. Unfortunately, the path is out of order so that last for loop puts the path in order, traversing over the ever decreasing path list, which contains only the nodes in the path, and since the path list is always decreasing when a node is added in order to the new path, this for loop is not in linear time either. Therefore the entire function, relative to the size of its input tree, runs in $O(n)$ time.

Part 2

Algorithm Description:

This algorithm first traverses all the leaves to add them to the unaligned list. Then it traverses over all the leaves and for each leaf, finds the distance between that leaf each of the other leaves by way of a nested for loop. From this loop it pulls the two leaves that are closest to each other by branch length in the tree. It should be noted that

this algorithm relies on doing an alignment of these first two closest nodes first, then adds the next closest node to the alignment and so on. This unfortunately was not a good approach as it disregarded the information contained in the actual structure of tree; in an ideal world this student would go back and have the algorithm do sub-alignments which then built into larger alignments but the student ran out of time so this was not possible. This poor algorithm performance is reflected in the very low SP scores of this algorithm. Next, this code defines a dictionary with a key corresponding to each species to store the gaps generated for that species during alignment, to be inserted at the end of the function. Next, the distances matrix is created and the alignment is performed, by doing number of leaves minus 1 alignments since the initial alignment adds 2 leaves and each one after that adds 1 leaf to the alignment. In this for-loop, the closest leaf to any of the already aligned leafs is found, then those sequences are passed into a twoway_align, which returns the aligned sequences, as well as a list of the indices of the gaps inserted into sequences to do the alignment. Next, the leaf that is being added to the aligned list has the gap list of the other sequence, the one that it was just aligned with that is already in the aligned list, pushed to it, followed by its own gap list from the alignment that it just did. Then the gap list for the sequence that was already in the aligned list is pushed to all other sequences in the aligned list. Then the new leaf is removed from the unaligned list and added to the aligned list. Next the dictionary of gaps is copied and set to zeros, since it will be modified and the original gap indices are also needed for the insertion of the gaps. Now each species is looped through in a for loop, and for each species, its gaps are looped through in a nested for loop. The commas in the gap list separate the alignments in which the gaps are added, so they are skipped over. When an index of a gap is come to, the gap is inserted into the sequence there, then the list of all other gaps are looped through. The gaps that came in the same alignment with this current gap are left alone, curtousey of all the indicator variable, but all gaps that follow, if they are larger than the gap just inserted, are pushed back 1 index to account for the fact that a gap was just inserted. This is why the insertion index is the index of gap, plus the index of the gaps_add, which is added to account for the gaps inserted before it. Once this is done, the proper alignments are returned.

Run Time Analysis:

The runtime of this algorithm in terms of number of leaves and leaf length is analyzed here. Visually it can be seen from looking at the code the first nested for loop is squared time, and can be ignored because of the triple nested for loop in the center of the function, which loops through number of leaves, then inside the first for loop is the two-way-align function, which is of $O(n^2)$. Therefore, this loop is $O(KN^2)$. At the end there is also a nested for loop that loops through all sequences, and then for each sequence, the indices of gaps, which worst case are on the order of the length of the sequences; this is the same complexity as the center loop, re-affirming that the complexity is $O(KN^2)$.

Part 3

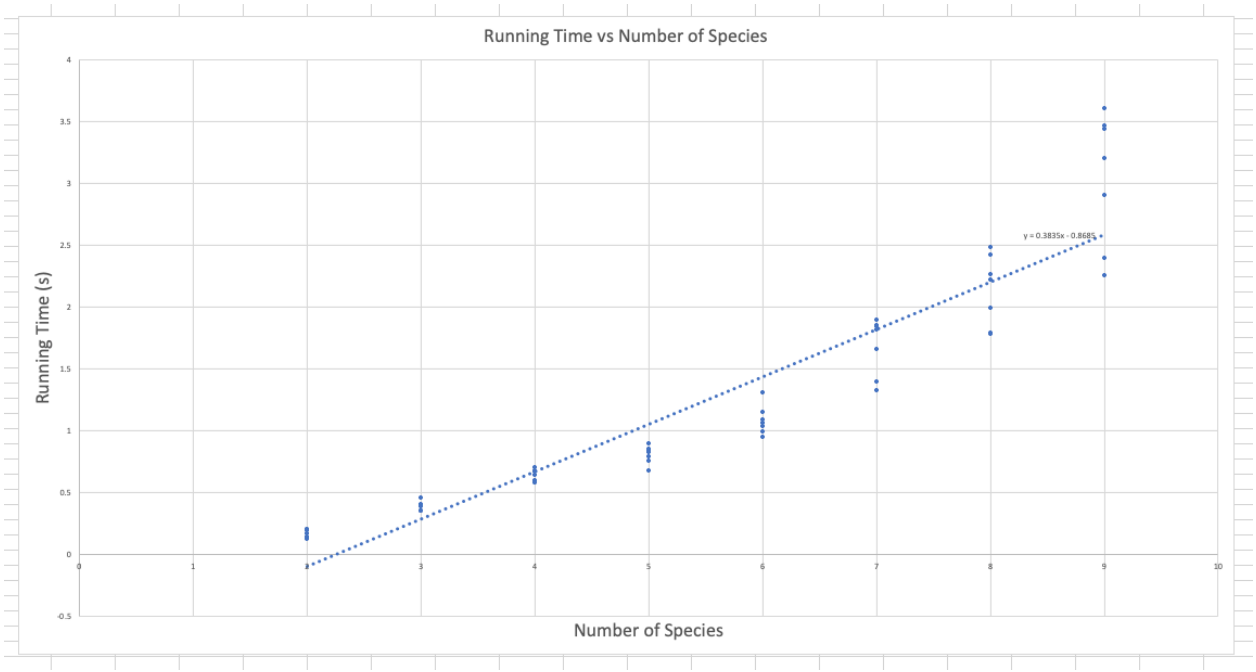

Figure 1: Running time vs number of species.

Here it can be observed and seen that the running time is roughly linear with the increasing number of species. The number of species are very small because this test is being run on the student's 8 year old laptop, so running, say 8 sample sizes with 10 different replicas caused the python ide to run out of memory for some reason, so the tests were run manually and with low number of species in order for the laptop to be able to complete them in a reasonable amount of time. It can be seen that the runtimes roughly match runtime for the code above, though with a larger number of species, it may become apparent that the increase is quadratic with the number of species, however, the student's hardware limitations prevent this, so it can be seen from the data that is available that it is *O(KN^2)*, (or at the very least linear in k, since n is not altered here), like the analysis above states.