Nathan Newbury Hw 4 Writeup:

Description of Algorithm:

We first traverse through the tree in post_order. Post_order always travels to children nodes before parents. The goal is to fill a dictionary for each node with the multiple sequence alignment (MSA) between all children nodes. In the base case, at every leaf, the alignment is just the original sequence. For all other nodes we call our alignment function with the children nodes to recursively align our way up the tree. At the root node, we should have a full multiple sequence alignment.

```
    for node in tree.traverse_postorder():
        #import pdb; pdb.set_trace()
        if node.is_leaf():
            #import pdb; pdb.set_trace()
            aln[node] = [sequences[node.get_label()]]
            #print(sequences[node.get_label()])
        else:
            #import pdb; pdb.set_trace()
            aln[node] = alignment(node.child_nodes())
```

The alignment function, which aligns nodes, works similar to the pairwise alignment algorithm. Instead of aligning single strings, we align two lists of previously aligned sequences. Gaps are added to the entirety of the list. Score is calculated as the “inner product” of the pairwise score of the list. For example, two lists with letters [l1 ,l2] and [l3,l4] have a score of S(l1,l3) + S(l1,l4) + S(l2,l3) + S(l2,l4). Similar to the pairwise algorithm, first we fill the axes of our score matrix. Instead of adding 1 gap we add the inner product of all letters from one list combined with gaps. This is just gap penalty*length(list)* length(list2)

```
        # fill in axes of score matrix
        S[0][0] = (0,None)
        for i in range(1, len(L1[0])+1):
            S[i][0] = (len(L2)*len(L1)*gap*i, 1)
        for j in range(1, len(L2[0])+1):
            S[0][j] = (len(L2)*len(L2)*gap*j, 2)
```

Next we fill in the face of our score matrix. In this case. I and j iterate through the letters in each string in list 1 and list 2 respectively. Each subproblem calculates the solution by taking the maximum score from three options. Either we add a gap to all strings in list 1, add a gap to all strings in list 2, or we add no gaps and align strings in each list at this letter. Calculating the total score for adding gaps is length(list)* length(list2) * gap penalty like above. Calculating the score of full alignment is just the inner product, as demonstrated above. This involves looping through each string as for each letter, and calculating the total Blosum score of that letter with each letter in the other list.

```
        # fill in full matrix
        # iterate through the score matrix i,j
        for i in range(1, len(L1[0])+1):
            for j in range(1, len(L2[0])+1):
                # Calculating total score of added letter requires
iterating through each string
                blosum = 0
                for s1 in L1:
                    for s2 in L2:
                        if s1[i-1]  == '-' and s2[j-1] == '-':
                            blosum = blosum + 0
                        elif  s1[i-1]  == '-' or  s2[j-1] == '-':
                            blosum = blosum + gap
                        else:
                            blosum = blosum + BLOSUM62[s1[i-1]][s2[j-1]]

                # once we have a score from iterating through each
combination of strings, do classic alignment
                option = [blosum + S[i-1][j-1][0], S[i-1][j][0] +
gap*len(L2)*len(L1), S[i][j-1][0] + gap*len(L2)*len(L1)]
                max_index = max(enumerate(option), key=lambda x: x[1])[0]
                arrow = max_index
                S[i][j] = (max(option), arrow)
```

We also record the arrow, as well as the score.

The arrow is recorded as 1,2 or 3 to indicate which subproblem yielded the maximum score for that step. The chosen subproblem indicates whether a gap is added in reconstruction.

```
        while arrow is not None:
```

```
            if arrow == 1:
                # add gap to L2 strings and letter to L1
                num = 0
                for s1 in L1:
                    alignL1[num]+=(s1[i-1])
                    num +=1
                num = 0
                for s2 in L2:
                    alignL2[num]+=('-')
                    num += 1
                i-=1
                # add gaps to L1 strings and letter to L2
            elif arrow == 2:
                num = 0
                for s1 in L1:
                    alignL1[num]+=('-')
                    num +=1
                num = 0
                for s2 in L2:
                    alignL2[num]+=(s2[j-1])
                    num += 1
                j-=1
                # add letters to both  L1 and L2
            elif arrow == 0:
                num = 0
                for s1 in L1:
                    alignL1[num]+=(s1[i-1])
                    num +=1
                num = 0
                for s2 in L2:
                    alignL2[num]+=(s2[j-1])
                    num += 1
                j-=1
                i-=1
            arrow = S[i][j][1]
```

Finally, we invert all of the strings. At the root node or our tree we have a list of all of the aligned strings (because the root node is connected to all leaves). We have to convert this list into a dictionary with the proper keys. To do this we iterate through the tree again seen here:

```
    alinlist = aln[tree.root]
    finalalign = {}
    num = 0
    for node in tree.traverse_postorder():
        if node.is_leaf():
            finalalign[node.get_label()] = alinlist[num]
            num = num +1

    return finalalign
```

Each leaf is a key to our final dictionary. These leaves are added to our list in the same order above so we extract them with the same order.

Proof of correctness:

This algorithm recursively calculates the score of every letter/gap combination. We use dynamic programming to use solutions to previous sub problems to solve new sub problems. Each subproblem in our matrix represents a pairwise aligned string with row number of letters from string 1 and column number letters from string 2. The score in this matrix entry is the score of the maximum alignment up until this point. After filling a nxn matrix (where n is the size of our sequences) the final row and column represents the score after placement of all potential. This will be the maximum score, and by backtracking with arrows, we can reconstruct the total alignment. This algorithm is extended for lists of sequences, but operates in the same way.

```
..........
----------------------------------------------------------------------
Ran 10 tests in 143.055s

OK
```

Complexity analysis:

Assuming that n is the number of sequences, and k is the length of the sequences. Each leaf of our tree represents a sequence, so the total number of nodes in our tree is (2n-1). We iterate through each node of our tree and call alignment. We also iterate through each node of our tree to convert our list back into a dictionary, but this takes much less time and is negligible. Each call of alignment takes two nodes. In one call of our alignment algorithm we loop through each letter in all strings in both lists. This represents k^2 computations. Additionally, we have to loop

through each string in both lists. For any node, there are on average log_2(n) sequences saved. This leaves a total run time of (2n-1) * O(k^2*log_2^2(n)) or a time complexity of O($k^2 n log_2^2(n)$) .