Problem Set #2 Comments

Problem 1

	LinkedList.py		ContinuousList.py	
Operation	Running Time	Memory	Running Time	Memory
`length(self)`	$\Theta(n)$	$\Theta(n)^{**}$ or $\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`__init__(self)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`access(self,index)`	$\Theta(n)$	$\Theta(n)$ or $\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`append(self,value)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`__str__(self)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

** Note: this answer is different from what was given in PS2 (see discussion below)

The `__init__` method for both `LinkedList.py` and `ContinuousList.py` is in $\Theta(1)$ for both running time and memory since everytime these classes are initialized, it allocates a constant amount of memory. There is no input size to scale here.

The running time of `access` for LinkedList is in $\Theta(n)$, but only $\Theta(1)$ for ContinuousList. This is because in the worst case, we need to call `access`, n times to get the last element. The memory usage comprises the memory that must be maintained on the state (for example, to store local variables and the return address), and any data structures created during the evaluation. The `length` and `access` methods use $\Theta(n)$ for the memory and running time. `length` in `LinkedList.py` calls itself n times. Each time it calls `length`, it needs to allocate a constant amount of memory. Hence, a straightforward implementation of `length` will use space in $\Theta(n)$. However, the interpreter may implement tail recursive calls more efficiently. When there is no computation to do at the callsite after the recursive call, the recursive call can reuse the stack space from the call site. Clever interpreters and compilers (such as Python), will not use stack space to keep track of tail recursive calls. If this is done, the amount of memory used in the `length` and `access` methods is constant ($\Theta(1)$).

Since both LinkedList and ContinuousList are immutable lists, the running time and memory use of `append` is in $\Theta(n)$. Both need to create a copy of the original list before adding the new element.

The `__str__` operations both construct a result that contains all the elements of the list, so its size grows with the length of the list. Hence, both its running time and space usage are in $\Theta(n)$.

Problem 2

Without changing data representation or semantics at all, it is not possible to improve the asymptotic running time of any of the methods. To do so would require keeping additional information or changing the structure of the data representation.

We can see this because for the LinkedList implementation, it requires following n next pointers to get to the end of the list. The `access`, `append`, `length`, and `__str__` methods all require reaching the last element in the list (in the worst case), so there is no way to scale better than $\Theta(n)$ with the basic linked representation. We can make the memory usage for all operations except `__str__` to be constant by using iteration instead of recursive calls. For `__str__`, we can't improve the space usage since the output produces scales as $\Theta(n)$.

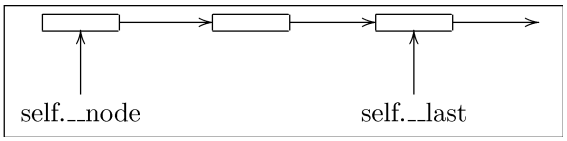
However, if we allow some changes to the data representation we can reduce the running time. For example, we can add a counter variable to the `LinkedList` representation we can do `length` in constant time. It would need to be updated in the `append` method, but that would not increase its asymptotic complexity. We could also improve the amortized running time and memory usage of `__str__` by maintaining a string variable that is updated everytime `append` modifies the list. (We'd better be careful to only do this when the list elements are immutable, though, otherwise other code may modify the value of list elements and our cached string will become inconsistent.)

Problem 3

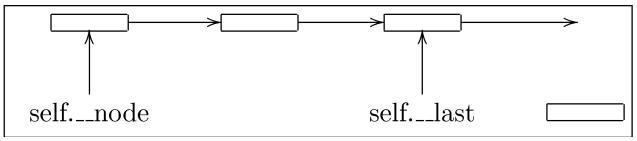
By changing the semantics of our list to be mutable, we no longer need to copy the list when we append. This provide the opportunity to do `append` in constant time, if we can find the end of the list in constant time. This can be done by adding a new private variable, `__last`. We modify `__init__` to initialize it to `None`:

```
def __init__(self):
    self.__node = None
    self.__last = None
```

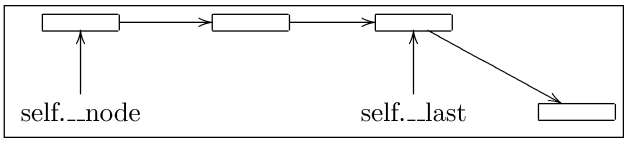
Now, you need to modify `append`. Assume you have a linked list like this:



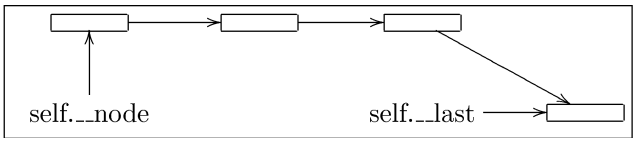
Thus the first thing append should do is to create a new node.



Then point `self.__last.__next` to the new node.



Then set the `self.__last` to the next of itself.



The case where the list is empty is trivial. The complete `append` definition in Python is

```
def append (self, e):
    if self.__node == None:
        self.__node = ListNode(e)
        self.__last = self.__node
    else:
        self.__last = self.__last.changeLink(ListNode(e))
    return self
```

And class `ListNode`, add

```
def changeLink(self, link):
    self.__next = link
    return self.__next
```

Problem 4

One way to do this is to create a table of what `bestAlignment("catg","atgg")` will call:

BestAlignment	catg	atg	tg	g
atgg	1	1	1	1
tgg	1	3	5	7
gg	1	5	13	25
g	1	7	25	63

Each cell is the number of time `bestAlignment` is called with the corresponding arguments. Note that each inner cell contains the sum of the values in the cells to its left, above, and diagonal above-left cells. This corresponds to the possible ways of reaching those arguments. From the table, there are 13 evaluations of `bestAlignment("tg","gg")`. To be able to call `bestAlignment("atg","tgg")`, it must be called from one of these functions:

```
bestAlignment("atgg","catg")
bestAlignment("tgg","catg")
bestAlignment("atgg","atg")
```

Hence, the number of evaluations of `bestAlignment("atg","tgg")` equal to the sum of the number of time it calls the list above.

Problem 5

The algorithm in the problem set is similar to Needleman-Wunsch in that it computes the best alignment of each substring pair once, and uses that to find the best possible alignment. The difference is Needleman-Wunsch finds the best alignment by building the matrix starting from the beginning (aligning or gapping the first element in the sequence), and our algorithm works just like the brute force algorithm from PS1, except instead of replicating effort recomputing the same best alignments, it saves the previously computed results.

However, the new implementation is slower than the Needleman-Wunsch algorithm because it needs to calculate `goodnessScore` everytime it need to compares. Since the number of times it needs to call `memoBestAlignment` and go through the case that it never found before is the same as the number of times we need to fill to a table a number in Needleman-Wunsch algorithm, the number of times it call `memoBestAlignment` is of order $\Theta(|U||V|)$. However, each time the function is called, it needs to spend time on copying the string, calculating the `goodnessScore`, and so on. These running times are in $\Theta(|U'| + |V'|) \subset O(|U| + |V|)$. Note that I use U', V' because most of the time, the strings that passes to `memoBestAlignment` are substring of U and V. Thus, this algorithm running time is of order $O((|U| + |V|)|U||V|)$. Note that we have the very strong assumption that look up in `dictionary` class has running time in $O(1)$.

The experimental results below support this hypothesis.

N	N^3	Est. Ratio	Est. Running Time	Running Time	Actual Time Ratio
100	1000000			1.7	
200	8000000	8	13.4	12.4	7.4
300	27000000	3.4	41.8	40.75	3.3
400	64000000	2.4	96.6	96.25	2.4
500	125000000	2.0	188.0	182.8	1.9
1000	1000000000	8	1680.0	(error)	(error)

Normal compiler/interpreter has a limit of how many depth you can call a function. Since `memoBestAlignment` is a recursive call, at $N = 1000$ the number of recursive calls Python allows is exceeded.

Problem 6

Here is our `__str__` definition:

```
    def __str__(self):
        def indentLines (str, level):
            if str == None: return None
            return str.replace('\n', "\n" + "".join([" " * level]))

        res = str(self.__value)

        for child in self.children ():
            res += indentLines ("\n" + str(child), 3)

        return res
```

Note that the running time of this algorithm is $\Theta(n)$ where n is the number of nodes in the tree.

Problem 7

```
def allPossiblePartitions (items):
    if len(items) == 1:
        yield [items[0]], []
        yield [], [items[0]]
    else:
        for left, right in allPossiblePartitions (items[1:]):
            lplus = left[:]
            lplus.insert (0, items[0])
            yield lplus, right
            rplus = right[:]
            rplus.insert (0, items[0])
            yield left, rplus
```

(a) The running time of

```
for p1,p2 in allPossiblePartitions(items):
    print p1,p2
```

Let $f(n)$ be the number of `yield` operations in `allPossiblePartitions(items)` where n is the number of element in `items`.

The base case is $f(1) = 2$, since if the length of `items` is one, it yields twice. For $n > 1$

$$f(n) = 2 * f(n-1)$$

since if the length of `items` is greater than 1, it calls `yeild` twice for each element it receives from the generator.

The close form of function $f(n)$ is 2^n. This makes sense: there are 2^n possible partitions of a set of n items. This is similar to the powerset, except instead of just collecting the subsets, we are collecting the subset and complement pairs.

This gives the number of iterations of the loop, but for the running time, we need to know how much work each iteration is. Suppose the list copy and insert operations have running time in $Theta(n)$. Then, the average length each list is $n/2$, so the work for each iteration is still $Theta(n)$. So, the total work is in $\Theta(n2^n)$. We leave it as an exercise for the reader to determine whether or not $\Theta(n2^n)$ is the same set as $\Theta(2^n)$.

(b) The memory space is easier to calculate. Each time `allPossiblePartitions` is called, it allocates new lists of size n and if $n > 1$, `allPossiblePartitions` will call itself with the size of list reduced by 1. Let $m(n)$ be the memory usage. We can describe $m(n)$ as

$$m(n) = \begin{cases} 1 & n = 1 \\ m(n-1) + n & n > 1 \end{cases}$$

which is the sum of series $m(n) = n + (n-1) + (n-2) + \ldots + 1$. Therefore, `allPossiblePartitions` uses $\Theta(n^2)$ for the memory space.

Problem 8

The brute force approach is to generate all possible trees, calculate the goodness score for each tree, and keep the trees with the maximal score.

We can generate all possible trees by picking each possible node as the root, and considering all possible trees we can generate for the left and right children of that node. The `allPossiblePartitions` procedure is useful for for this. We can consider the partitions as grouping the set of nodes in the left branch and right branch of the tree. So, we create all possible trees for each partition, and construct the tree from the root node and choices of left and right trees.

Recursively, we can think of the base case as being the set containing just one element. It has one possible tree with that element as the root. If the set has more than one element, we need to construct all possible trees by selecting each of the elements as the root, and constructing all possible trees with that node as the root, and the other elements partitioned between its right and left branches. We can have only one child, or two children. If we have one child, then we just need to construct all possible trees for the remaining nodes, and make each of those the child of the selected root node. If we have two children, we need to try all possible partitions of the remaining nodes, and all possible trees with each partition.

Problem 9

Assume that `generateAllTrees(alist)` will generate all possible binary trees that compose of element in `alist`.

In the base case, if the size of element in the `alist` is 1, just return the tree with that element as the root:

```
def generateAllTrees(alist):
   if len(alist) == 1:
       yield Tree(alist[0])
```

Otherwise, we need to construct the all possible trees from the `alist`. For each element in `alist`, it is the candidate to be a root. So we can say that

```
   else:
      for root in alist:
```

So the root should not be in alist because we want to generate tree from the remaining elements (note that we are careful to copy `alist` before removing the element, since we do not want to modify the original list):

```
         newlist = alist[:]
         newlist.remove(root)
```

Now, we have to generate left and right subtrees of the root we have. Fortunately, we have the function that can generate all possible binary trees, that is `generateAllTrees` itself. So we need to worry about what we should put in the list for `generateAllTrees` to generate left subtrees and right subtrees. We need to consider all ways of partitioning the nodes.

The left subtrees and the right subtrees come from the all possible partitions from the `newlist`:

```
for leftlist, rightlist in allPossiblePartitions(newlist):
```

Now we generate the tree with three cases : the root has child on both side, only left side and only right side.

```
for leftlist, rightlist in allPossiblePartitions(newlist):
   # have child on both side
   for leftTree in generateAllTrees(leftlist):
       for rightTree in generateAllTrees(rightlist):
           rootTree = Tree(root)
           rootTree.addLeft(leftTree)
           rootTree.addRight(rightTree)
           yield rootTree
       # have only left Tree
       if (len(rightlist)==0):
           rootTree = Tree(root)
           rootTree.addLeft(leftTree)
           yield rootTree
   # have child on right side
   if (len(leftlist) == 0):
       for rightTree in generateAllTrees(rightlist):
           rootTree = Tree(root)
           rootTree.addRight(rightTree)
           yield rootTree
```

Putting it all together, we have:

```
def generateAllTrees (sset):
    if len(sset)==1:
        yield Tree.Tree(sset[0])
    else:
        for species in sset:
            newset = sset[:]
            newset.remove(species)
            for p1,p2 in allPossiblePartitions(newset):
                for lefttree in generateAllTrees(p1):
                    if (len(p2)==0):
                        tree = Tree.Tree(species)
                        tree.addLeft(lefttree)
                        yield tree
                    else:
                        for righttree in generateAllTrees(p2):
                            tree = Tree.Tree(species)
                            tree.addLeft(lefttree)
                            tree.addRight(righttree)
                            yield tree
                if (len(p1)==0):
                    for righttree in generateAllTrees(p2):
```

```
                    tree = Tree.Tree(species)
                    tree.addRight(righttree)
                    yield tree
```

We can find the best tree by trying all possible trees to find the highest possible parsimony score, and then output all trees that match that score:

```
def findTree (sset):
    def parsimonyScore(tree, seqs, c, g):
        sum = 0
        for child in tree.children():
            sum += DynAlign.bestScore(seqs[tree.getValue()], seqs[child.getValue()], c, g) \
                   + parsimonyScore(child, seqs, c, g)
        return sum

    species = []
    for name in sset:
        species.append(name)
    maxcost = 0
    for tree in generateAllTrees(species):
        maxcost = max(maxcost, parsimonyScore(tree, sset, 10, 2))

    for tree in generateAllTrees(species):
        if (parsimonyScore(tree, sset, 10, 2)) == maxcost:
            yield tree
```

This code is simple, but inefficient. We can improve its efficiency by caching the goodness score values of each pair of sequences so that we need don't need to spend time on recalculating `DynAlign.bestScore`:

```
def findTree (sset):
    # constants we use for the match score and gap penalty
    c = 10
    g = 2

    def parsimonyScore (tree):
        # pre: goodness must be a completed goodness matrix for
        #      all the species in tree
        score = 0

        for child in tree.children():
            score += goodness[tree.getValue ()][child.getValue ()]
            score += parsimonyScore (child)

        return score

    # first, find the goodness scores between all pairs
    goodness = {}
    for key1 in sset.keys ():
        goodness[key1] = {}
        for key2 in sset.keys ():
            goodness [key1][key2] = DynAlign.bestScore (sset[key1], sset[key2], c, g)

    bestscore = 0
    besttrees = None
    # find the most parsimonious (highest scoring) tree
```

```
    for tree in allPossibleTrees (sset.keys ()):
        treescore = parsimonyScore (tree)
        if besttrees == None or treescore > bestscore:
            besttrees = [tree]
            bestscore = treescore
        elif treescore == bestscore:
            besttrees.append (tree)

    print "The best " + str(len(besttrees)) + " trees are (score: " + str(bestscore) + "):\n"
    for tree in besttrees:
        print tree
        print "=========="
```

Note that the code above generates all the trees including trees that are isomorphic (equivalent if the left and right branches are swapped). To make it only yield distinct trees, we need to add two rules:

- Every node has a right child must have a left child.
- Every node that has both left and right child, the value in the node in the left child must be less than the value in the node in the right child.

Here is a version of `generateAllTrees` that generates only the distinct trees:

```
def generateAllTrees (sset):
    if len(sset)==1:
        yield Tree.Tree(sset[0])
    else:
        for species in sset:
            newset = sset[:]
            newset.remove(species)
            for p1,p2 in allPossiblePartitions(newset):
                if (len(p1)!=0):
                    for lefttree in generateAllTrees(p1):
                        if (len(p2)==0):
                            tree = Tree.Tree(species)
                            tree.addLeft(lefttree)
                            yield tree
                        else:
                            for righttree in generateAllTrees(p2):
                                tree = Tree.Tree(species)
                                if (lefttree.getValue() < righttree.getValue()):
                                    tree.addLeft(lefttree)
                                    tree.addRight(righttree)
                                    yield tree
```

Problem 10

To solve this problem we need to analytically determine the rate of growth of our procedure, and experimentally measure its time on some inputs.

Let n be the number of elements, and k be the length of the sequences.

The `generateAllTrees` procedure in isomorphic case generates exactly $\frac{n!}{n+1}C(2n, n)$ trees. This grows enormously:

n	Number of Trees
1	1
2	4
3	30
4	336
5	5040
6	95040
7	2162160
8	57657600
9	1764322560
10	60949324800
11	2346549004800
12	99638080819200
13	4626053752320000
14	233153109116928000
15	12677700308232960000
16	739781100339240960000
17	46113021921146019840000
18	3058021453718104473600000
19	214978908196382744494080000
20	15969861751731289590988800000

However, the running time to generate all the tree is $O(n\frac{n!}{n+1}C(2n,n))$.

In this case, we go through all these trees, and each tree we calculate the goodnessScore from the tree.

To calculate goodnessScore of a tree, we look at each link, which consists of $n-1$ links and then find the best match of two sequences, which costs k^3.

Thus, the approximate running time is of order $\Theta(k^3 n\frac{n!}{n+1}C(2n,n))$.

Our implementation only works for $n <= r$:

n	k	Expected Value	Ratio	Expected Running Time (s)	Actual Running Time (s)
3	10	270000		None	0.5
3	20	2160000	8.0	None	2.7
3	40	17280000	8.0	21.7	20.4
3	60	58320000	3.4	68.9	65.2
3	70	92610000	1.6	103.5	110.9
3	80	138240000	1.5	165.5	166.2
4	30	145152000	1.1	174.6	189.4
4	20	43008000	0.3	56.1	57.2

With $n > 4$, Python runs out of resources before completing.

The question asked for an estimate of how long it would take to find a phylogeny with $n = 16$. The number of trees we would need to consider at $n = 16$ is 2201729465295360000 times the number of trees at $n = 4$. So, even if we optimized out all the other factors in the running time, if it takes 1 minute to solve the case where $n = 4$, it will take over 4 quadrillion years to solve the case where $n = 16$. Most biologists are not willing to wait so long. Hence, the greedy algorithm in PS3 and the reason biologists must settle for possibly non-optimal phylogenies.