University of Virginia Computer Science
CS216: Program and Data Representation, Spring 2006
http://www.cs.virginia.edu/cs216/ps/ps2/
30 January 2006

Problem Set 2
Phylogeny

Out: 30 January
Due: 8 February (11am)

What to Turn In

On Wednesday, 8 February, bring to class a stapled turn in containing your answers to questions 1-10. You and your partner should turn in a single assignment reflecting your best combined efforts with both of your names on it. Your answers should be printed in order and formatted clearly. Include all the code you wrote, but do not include large amounts of code we provided in your turn in.

Collaboration Policy - Read Carefully

For this assignment, you should do the first two parts (questions 1-5) on your own, and then meet with your assigned partner. Assigned partners will be emailed to the course list on Monday, January 30.

When you meet with your partner, you should first discuss your answers to the first two parts to arrive at a consensus best answer for each question. The consensus answer is the only answer you will turn in. Then, you should work as a team on the final part (questions 6-10). When you are working as a team, both partners should be actively involved all the time and you should take turns driving (who is typing at the keyboard).

You may consult any outside resources including books, papers, web sites and people, you wish for information on Python programming. Unlike Problem Set 1, you should feel free to conduct web searches or look at reference material on Sequence Alignment, Phylogeny, and related problems as you wish. You are also encouraged to discuss these problems with students in the class, including (but not limited to) your assigned partner.

You are **strongly encouraged** to take advantage of the staffed lab hours (which will be posted on the CS216 web site).

Purpose

- Get familiar with the Python programming language and tools
- Learn to experimentally analyze the asymptotic efficiency of a data structure
- Understand and analyze a brute-force genome alignment algorithm

Reading: Read Chapter 4 of the textbook.
Download: ps2.zip. This contains 5 Python files: `LinkedList.py` (a linked representation immutable list datatype), `ContinuousList.py` (a continuous representation immutable list datatype), `DynAlign.py` (a memoized implementation of sequence alignment), `Tree.py` (a simple binary tree datatype), and `Phylogeny.py` (some starting code for your phylogeny program).

List Representations

The provided files `LinkedList.py` and `ContinuousList.py` provide two simple immutable list datatype implementations.

1. For each of the operations listed below, describe the asymptotic running time and memory use. Use the variable *n* to represent the length of the self list. State clearly any assumptions you need to make about the underlying Python list implementation used in `ContinuousList.py`. (We have provided answers for the `length` operation.)

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

2. Is it possible to implement any of the operations with better asymptotic performance without changing the data representation or semantics? (Note that making the datatype mutable does change the semantics, so is not an option for this question.) If so, explain how you would improve the asymptotic running time or memory usage for one of the operations. If not, explain why it is not possible to improve the asymptotic running time or memory usage of any of the operations (without changing the representation or semantics).

Mutability

The list abstraction datatype described in the textbook is *immutable*. This means that once a list object is created, its value never changes. Operations that might appear to modify a list, such as `append`, are actually defined in a way that produces a new list and leaves the original list unchanged.

We can define a mutable abstract datatype with operations as follows (note that the first three operations are identical to those for the immutable list datatype from Lecture 3:

- Access (L, i) (corresponds to the `access(self, index)` method): returns $L[i]$
- Length (L) (corresponds to the `length(self)` method): returns $|L|$
- MakeEmptyList() (corresponds to the `__init__(self)` method): returns <>
- Append (L, e) (corresponds to the `append(self, value)` method): modifies the value of L. The value of $L_{post} = <l_0, l_1, ..., l_{|L|-1}, e>$ where $<l_0, l_1, ..., l_{|L|-1}>$ are the elements of L before the call.

We can modify the `ContinuousList.py` implementation to match the new semantics by modifying just the `append` operation:

```
    def append(self, value):
        self.__items.append(value)
        self.__len += 1
        return self
```

3. Modify the `LinkedList.py` implementation to support the mutable list semantics and to provide an `append` operation that whose running time and memory use are both in *O*(1). (Hint: you may need to add a field to the `LinkedList` class.)

Dynamic Programming

Dynamic programming is an algorithmic technique for avoiding duplicate computation by accumulating partial results. Typically, we turn a recursive definition that solves a problem by composing the solutions to smaller problems into an algorithm that accumulates and combines the partial results.

Consider our alignment code from PS1, excerpted below:

```
def bestAlignment (U, V, c, g):
    if len(U) == 0 or len(V) == 0:
        ...
    else:
        # try three possibilities:
        (U0, V0) = bestAlignment (U[1:], V[1:], c, g)
        ...
        (U1, V1) = bestAlignment (U, V[1:], c, g)
        ...
        (U2, V2) = bestAlignment (U[1:], V, c, g)
        ...
        # pick the best one
```

Although this is a clear way of finding the best alignment, as discussed in Lecture 4 it is very inefficient. So inefficient, that we cannot find alignments for non-trivial strings.

4. In computing `bestAlignment ("catg","atgg")`, how many times does the PS1 algorithm evaluate `bestAlignment ("tg","tt")`? (You may want to check your answer experimentally, but you should justify your answer analytically.)

An easy way to implement dynamic programming is to just store previously computed results in a table. Here, we modify the PS1 alignment code to do that. Note that Python's dictionary datatype makes this easy. We just need to find a unique key to use to identify results to different alignment problems. For this, we just concatenate the *U* and *V* inputs with a `%` separator between them (that may not appear in *U* or *V*.

The modified code is found in `DynAlign.py` and shown below. The key changes are bolded:

```
def bestAlignment (U, V, c, g):
    def memoBestAlignment (U, V, c, g):
        def makeKey (U, V):
            return U + "%" + V

        if memo.has_key(makeKey (U,V)):
            res = memo[makeKey (U,V)]
            return res[0], res[1]

        if len(U) == 0 or len(V) == 0:
            while len(U) < len(V): U = U + GAP
            while len(V) < len(U): V = V + GAP
            resU = U
            resV = V
        else:
            # try with no gap
            (U0, V0) = memoBestAlignment (U[1:], V[1:], c, g)
            scoreNoGap = goodnessScore (U0, V0, c, g)
            if U[0] == V[0]: scoreNoGap += c

            # try inserting a gap in U (no match for V[0])
            (U1, V1) = memoBestAlignment (U, V[1:], c, g)
            scoreGapU = goodnessScore (U1, V1, c, g) - g

            # try inserting a gap in V (no match for U[0])
            (U2, V2) = memoBestAlignment (U[1:], V, c, g)
            scoreGapV = goodnessScore (U2, V2, c, g) - g

            if scoreNoGap >= scoreGapU and scoreNoGap >= scoreGapV:
                resU = U[0] + U0
                resV = V[0] + V0
            elif scoreGapU >= scoreGapV:
                resU = GAP + U1
                resV = V[0] + V1
            else:
                resU = U[0] + U2
                resV = GAP + V2

        memo[makeKey(U,V)] = [resU, resV]
        return resU, resV

    memo = {}
    return memoBestAlignment (U, V, c, g)
```

5. How does the algorithm provided here compare to the Needleman-Wunsch algorithm (from Lecture 4)? A good answer will compare the asymptotic running time of the two algorithms as well as discuss other differences that may make one or the other a better choice. You may assume Python's dictionary type provides lookup and insert operations that have running times in *O*(1).

Phylogeny

As introduced in Lecture 1, a phylogeny organizes items according to their evolutionary relationships. A phylogeny of life shows how different species evolved from common ancestors. A phylogeny of language shows how different languages evolved from a common language.

The Tree of Life project is developing a phylogeny for organisms on Earth. If you are unsure of your place in the univere, try staring from Life on Earth and walking down the tree to find *Homo sapiens*.

The way biologists (or linguists) determine evolutionary relationships is to look for similarities and differences between species (or languages). This is done by identifying a set of features that describe properties of a species or language. For species, the features might be phenotypic properties (e.g., do organisms have wings or gills?) or genotypic properties (the DNA sequence). Genotypic properties are likely to produce more accurate results, since small changes in genomes can produce large phenotypic changes. Note that this is a *historical* study. It can rarely provide definitive proof of a particular relationship, but a preponderance of evidence can make one explanation appear to be the most likely.

If two species have similar genomes, it is likely they evolved from a relatively recent comon ancestor.

Biologists measure the similarity of genomes based on the number and likelihood of different kinds of mutations — base pairs may be inserted, deleted, duplicated, moved, or substituted. The number of mutations necessary to match two genomes gives an indication of the likelihood that the species evolved from a common ancestor. For this assignment we will assume a very simple model: the only mutation is a substitution or a single base pair and all substitutions are equally likely.

One measure of which tree is the most likely to represent the actual evolution of a set of species is *parsimony*. The parsimony principle is that if there are two possible explanations for an observed phenomenon, the simpler explanation is most likely to be correct. In producing phylogenetic trees, parsimony means we should look for the tree that requires the fewest possible total number of mutations. The goodness scores of the best possible alignments of two nucleotide sequences are one way of measuring how related they are. So, our goal is to construct a tree that maximizes the total goodness score of all connected pairs.

For example, consider the set of species described by the genomes below (of course, these are not their real genomes!):

Species	**Sequence**
Cat	catcat
Dog	gggggg
Feline	cccccc
Tiger	cccaat

The goodness scores of the possible pairs (using the *c*=10, *g*=2 goodness metric from PS1) :

Species	Cat	Dog	Feline	Tiger
Cat	-	0	20	36
Dog	0	-	0	0
Feline	20	0	-	30
Tiger	36	0	30	-

Note that our goodness score metric is symmetric (that is goodness(*a*,*b*) = goodness(*b*,*a*)).

Our goal is to find likely evolutionary relationships among the species by maximizing the sum of the goodness scores of all direct relationships. For example, consider the tree:

```
        Cat
        / \
       /   \
    Tiger   Dog
            /
           /
        Feline
```

The total goodness score is goodness(Cat, Tiger) + goodness (Cat, Dog) + goodness (Dog, Feline) = 36.

This is a less likely phylogeny than,

```
  Feline
   / \
  /   \
Dog  Tiger
      /
     /
   Cat
```

which has a total goodness score of 66. Other trees have the same score, but no tree has a higher score.

For the remaining questions on this assignment, and most of Problem Set 3, you will explore algorithms and data structures in the context of finding phylogenetic trees. Note that we have greatly simplified the actual problem of determining biological evolutionary relationships. In fact, many species evolved from common ancestors which are now extinct. So, a more realistic phlogeny program would need to insert additional nodes to find a likely tree.

Trees

The file `Tree.py` provides a simple binary tree implementation. It is missing the `__str__` method for converting a tree to a string representation (this is the method Python will call when a `Tree` object is printed using `print` or passed as a parameter to `str`). We need a `__str__` method that displays a Tree object in a way that reveals its structure. For example, the two example trees above could be printed:

```
Cat
   Tiger
   Dog
      Feline
```

and

```
Feline
   Dog
   Tiger
      Cat
```

Note that we do not need to distinquish between the left and right child when a tree has only one child.

6.a. Define the `__str__` method for `Tree.py`.
6.b. What is the asymptotic running time of your `__str__` method? Is it possible to do better?

Brute Force Phylogeny

A brute force algorithm for determining the best phylogenetic tree is to calculate the total goodness score for all possible trees and select the best one (or ones). As you will establish in the next question, this is not a scalable method; it guarantees that the best possible phylogenetic tree is found, but only works for very small input sizes.

Note: The generator mechanism described below was introduced in Phython version 2.2. The version of Phython running on the ITC lab machines (as of 30 January) is Python 2.1. ITC is working on upgrading the version of Phython on the lab machines, and we hope the new version will be available when you reach this question. If not, we will provide a version of this code that does not use generators.

You may find Python's generators an elegant mechanism for expressing your algorithm. A generator is similar to a procedure, except instead of returning once, it can yield a sequence of values. The caller uses it in a for loop instead of a procedure call. For example, `Tree.py` defines this generator:

```
    def children(self):
        if not self.__left == None:
            yield self.__left
        if not self.__right == None:
            yield self.__right
```

It will yield the left child (if there is one) the first iteration through the loop, and the right child (if there is one) the second iteration. When the generator exits, there are no more values to yield and the calling loop terminates. A client uses it like this,

```
childsum = 0
for child in tree.children():
    childsum += child.getValue ()
```

The generator defined below yields all possible two-part partitions of the input list:

```
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
```

7. Consider this code excerpt that prints out all possible partitions of the list *s*:

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

Use *n* to represent the number of elements in *s*. You may assume `print` is *O*(1).

a. What is its asymptotic running time?
b. What is its memory usage?

Justify your answer with a clear argument and explain carefully all assumptions you make.

8. Describe your plan for implementing a phylogeny algorithm. Your design should include an explanation of how you will break the problem into components. Once you have figured out a good design, you may divide the coding parts with your partner, but be sure to test them independently before you try to compose everything. (Hint: you may find the `allPossiblePartitions` generator helpful.)

9. Implement a brute force phylogeny program that takes as input a set of species and their genomes (represented using a Python dictionary), and produces as output a list of the best possible phylogenetic trees. For example:

```
findTree ({'feline':'cccccc', 'cat':'catcat', \
           'tiger':'cccaat', 'dog':'gggggg'})
```

should produce the all trees with maximal goodness score (66), including the tree above. (The number of trees is 18, if we count isomorphic trees where the trees would be identical if the left and right children are swapped. A better solution would remove these isomorphically equivalent trees, since there is no different meaning associated with the left and right children. It is acceptable for a "green star" level solution to this question to include trees that are isomorphically equivalent in your output.)

10. How much more computing power would you need for your program to produce a phylogeny for an input set consisting of 16 elements with nucleotide sequences of 1000 bases each within one day? Justify your answer using a combination of analytical reasoning and experimental results.

Credits: This problem set was created by David Evans and tested by Pitchaya Sitthi-Amorn and David Faulkner for CS216 Spring 2006.