University of Virginia Computer Science
CS216: Program and Data Representation, Spring 2006
http://www.cs.virginia.edu/cs216/ps/ps3/
10 February 2006

Problem Set 3
Phaster Phylogeny

Out: 9 February
Due: 15/17 February (see below)

Deadline

You may turn in thie assignment either on Wednesday, February 15 (at the beginning of class) or by dropping it off on paper in the folder outside Prof. Evans' office before 4:50pm on Friday, February 17. If you turn it in on Wednesday, it will be returned to you in section on February 20th or 21st, so you received feedback on it before Exam 1. If you turn it in after Wednesday's class, you may not receive feedback on it until after Exam 1.

Collaboration Policy - Read Carefully

For this assignment, you may work on your own or with any one other person of your choice. If you work with a partner, you should turn in one assignment with both of your names on it. Keep in mind that the main purpose of this assignment is to help you prepare for Exam 1. So, you should decide to work alone or with a partner based on which approach you believe will be most helpful to you in learning the material it covers.

You may consult any outside resources including books, papers, web sites and people you wish. You are also encouraged to discuss these problems with students in the class.

You are **strongly encouraged** to take advantage of the staffed lab hours posted on the CS216 web site.

Purpose

- Gain some experience with recursive definitions.
- Prepare for Exam 1 by reviewing concepts covered in CS216 so far including asymptotic algorithm analysis, data abstractions and their implementation (lists, trees), developing algorithms that manipulate data abstractions.
- Learn to develop and understand greedy algorithms.

Reading: Re-read the textbook sub-section on Greedy Algorithms (pages 60-61) and read Chapter 6 (List and Tree Implementations of Sets). (We are not covering Chapter 5 now, but will return to it later.)

Download: ps3.zip.

Recursive Definitions

These questions provide practice defining and analyzing methods that operate on trees.

1. Define a method, `equal` in the `Tree` class that takes a tree as its parameter, and evaluates to true if and only if the input tree is equal to self. Two trees are considered equal if they have the same branching structure and the values of every node are the same (== comparison) in both trees. Your definition should be recursive (it cannot use any looping control structure such as `for` or `while`).

2. For each of the subquestions, express your answer as a aymptotically tight (Θ) bound and briefly justify your answer. Use *N* to represent the number of nodes in the input tree. Assuming the Python interpreter implements procedure calls in a straightforward way (that is, it does not do any transformations to optimize tail recursive calls).

a. What is the worst case running time of your `equal` method?
b. What is the best case running time of your `equal` method?
c. What is the worst case space usage of your `equal` method?
d. What is the worst case space usage of your `equal` method if the input trees are both well-balanced?

3. Define a method `isomorphic` in the `Tree` class that takes a tree as its parameter, and evaluates to true if and only if the input tree is isomorphic to self. Two trees are considered isomorphic if their root nodes are equal, and each node in the tree either (1) has a left child that is isomorphic to the left child of the corresponding node in the self tree and has a right child that is isomorphic to the right childe of the corresponding node in the self tree; or (2) has a left child that is isomorphic to the right child of the corresponding node in the self tree and has a right child that is isomorphic to the left child of the corresponding node in the self tree. (The intuition behind our definition is the two trees would be equal if you could swap left and right children.)

4. Define a method, `iterEqual` in the `Tree` class that takes a tree as its parameter, and evaluates to true if and only if the input tree is equal to self (with the same behavior as the `equal` method in question 1). Your definition should **not** be recursive (it cannot use any recursive calls, but may use looping control structures such as `for` or `while`).

5. For each of the subquestions, express your answer as a aymptotically tight (Θ) bound and briefly justify your answer. Use *N* to represent the number of nodes in the input tree. Assuming the Python interpreter implements procedure calls in a straightforward way (that is, it does not do any transformations to optimize tail recursive calls).

a. What is the worst case running time of your `iterEqual` method?
b. What is the best case running time of your `iterEqual` method?
c. What is the worst case space usage of your `iterEqual` method?
d. What is the worst case space usage of your `iterEqual` method if the input trees are both well-balanced?

Dictionaries

`ContinuousTable.py` provides a continuous representation of a dictionary data type. (We considered the `lookup` method in Lecture 6.)

6. The provided `insert` method has expected running time in Θ(*N*)where *N* is the number of entries in the table. (We are optimistically assuming the Python slicing and access operations are in *O*(1).) Define an `insert` method that has expected running time in Θ(log *N*).

7. Ari Tern suggest replacing the implementation of `lookup` with this implementation (`tlookup` in `ContinuousTable.py`):

```
    def tlookup(self, key):
        def lookuprange(items):
            if len(items) == 0: return None
            if len(items) == 1:
                if items[0].key == key:
                    return items[0].value
                else:
                    return None
            split1 = len(items) / 3
            split2 = 2 * len(items) / 3

            if key < items[split1].key:
                return lookuprange (items[:split1])
            elif key < items[split2].key:
                return lookuprange (items[split1:split2])
            else:
                return lookuprange (items[split2:])

        return lookuprange(self.items)
```

Is this a good idea? (A good answer will consider the effect of Ari's change on both the asymptotic and absolute properties of the procedure.)

UPGMA

In Problem Set 2, we explored a brute force phylogeny algorithm. Although it was guaranteed to find the "best" (most parsimonious) phylogeny for a set of sequences, its brute force approach meant that the required computation time quickly exceeded the expected remaining time for the universe for even relatively small inputs.

The problem of finding the best phylogeny for a set of sequences is known to be NP-Complete (don't worry if you dont know what this means yet, we will cover it later). This means that it is unlikely that any solution asymptotically better than trying all possible trees exists. (If a faster approach is found, it would mean that lots of other believed to be hard problems could also be solved quickly.) So, to solve phylogeny construction problems of a non-trivial size, we need to make compromises. We tradeoff the guarantee of finding the best phylogeny, for the practicality of finding a phylogeny that is likely to be reasonably good quickly.

The approach we will use is an example of a *greedy algorithm*. A greedy algorithm makes the locally optimal solution first and at each successive step. This strategy is fast, since it only involves considering each immediate possibility, instead of considering all possibilities for the entire solution. However, it is not guaranteed to lead to a globally optimal solution (in this case, it might not find the best possible phylogeny).

The algorithm we will use is a simple version of the UPGMA (unweighted pair group method with arithmetic mean) algorithm (which is a bit simpler than the most popular current phylogeny construction algorithms).

The idea behind UPGMA is to greedily form groupings by forming subtrees by connecting the most similar sequences at every step. We start by computing a table of the goodness scores of all pairs of sequences. Then, we find the two elements with the highest goodness score, and connect them (one element will be the parent and the other its left child). Then, we add the other elements to the tree greedily — with each step we find the addition with the maximal parsimony score possible (without altering the existing tree). Each iteration considers all remaining elements in the set, and all possible positions in the tree where they could be added — as a new root (with the existing root as its left child) and as a child of any node that does not already have two children. We continue in this manner until all nodes are added to the tree.

For example, consider the example from PS2 with goodness matrix:

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

Our greedy algorithm will start by linking the two elements with the highest goodness score:

```
Tiger
   Cat
```

Of course it is symmetric, so we could also do,

```
Cat
   Tiger
```

Next, we will add another element to the tree. First, we consider adding Feline. There are 3 possibilities:

```
Feline
   Tiger
      Cat
               goodness = 30 + 36 = 66
Tiger
   Cat
      Feline
               goodness = 36 + 20 = 56
Tiger
   Cat
   Feline
               goodness = 36 + 30 = 66
```

We also consider adding Dog:

```
Dog
   Tiger
      Cat
               goodness = 0 + 36 = 36
Tiger
   Cat
      Dog
               goodness = 36 + 0 = 36
Tiger
   Cat
   Dog
               goodness = 36 + 0 = 36
```

Of the six trees we consider, the best are the 1st and 3rd (with equally good scores of 66). So, we greedily pick one of them (say the 1st) and continue.

Now, we have one element left to add. We consider all possibilities of adding dog to the tree:

```
Dog
   Feline
      Tiger
         Cat
```

```
Feline
   Tiger
      Cat
   Dog

Feline
   Tiger
      Cat
      Dog

Feline
   Tiger
      Cat
         Dog
```

All of them are equally good (since the goodness score of Dog with any other element is 0).

In this case we were lucky — the greedy algorithm found the best possible phylogeny with far less work than the brute force algorithm. However, the greedy algorithm is not guaranteed to always find the best phylogeny.

8. Construct a simple example where the greedy algorithm does not find the best phylogeny. Explain why the greedy algorithm does not find the best possible phylogeny for your example.

9. What is the asymptotic running time of the greedy phylogeny algorithm? Explain your reasoning clearly and any assumptions you make.

These two questions are intended for ambitious students. It is not necessary to answer this question to obtain "green star" level performance on this assignment, if you answer all the other questions well.

10. Implement the greedy phylogeny algorithm. (Feel free to reuse any code you want from your PS2 implementation or our provided solutions.)

11. Analyze experimentally the running time of your implementation and compare it to the analytical result from question 10. If your implementation's running time does not seem to be consistent with your answer to question 10, speculate on why it might be different.