“I wonder about Trees” – Robert Frost

Hey! How come no turtles in this tree?
Trees and the parsimony principle

Data

1 CCA
2 GC
3 C
4 GCTG
Trees and the parsimony principle

Data

1 CCAGT
2 GCACT
3 CCTGA
4 GCTGT

"Better", i.e. more parsimonious
Another general strategy for inferring phylogenies

- Generate all possible trees
- Pick the most parsimonious given some data
Generating all possible trees from the ground up

Num. Species (leaves) 1 2 3 4
Num. possible trees 1 1 3
Draw all possible trees that result from adding a species 4 to this tree.
Draw all possible trees that result from adding a species 4 to this tree.
A convention for naming internal/ancestral nodes

Tree = ( 'Anc', (1,(),()), (2,(),()) )

Tree diagram:

```
     'Anc'
    /   \
   /
  1   

     2
```
The `addLeaf` function

```
>>> leaf = ( 3, (), () )
>>> Tree = ( 'Anc', (1,(),()), (2,(),()) )
>>> addLeaf(leaf,Tree)
[
    ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ()))),
    ('Anc', ('Anc', (3, (), ()), (1, (), ())), (2, (), ())),
    ('Anc', (1, (), ()), ('Anc', (3, (), ()), (2, (), ()))),
]
```

Which illustration does the last tuple tree correspond to?
The `addLeaf` function

```
>>> leaf = ( 4, (), () )
>>> Tree = ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ()) )
>>> addLeaf(leaf,Tree)
[
  ('Anc', (4, (), ()), ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ()) ))),
  ('Anc', ('Anc', (4, (), ()), (3, (), ()) ), ('Anc', (1, (), ()), (2, (), ())))),
  ('Anc', (3, (), ()), ('Anc', (4, (), ()), 'Anc', (1, (), ()), (2, (), ())))),
  ('Anc', (3, (), ()), ('Anc', ('Anc', (4, (), ()), (1, (), ()) ), (2, (), ()) ))),
  ('Anc', (3, (), ()), ('Anc', (1, (), ()), ('Anc', (4, (), ()), (2, (), ())))
]```
def addLeaf(newleaf, Tree):
    """Returns a list of all possible trees that result from adding newleaf to Tree."""
    leftTree = Tree[1]
    rightTree = Tree[2]
def addLeaf(newleaf, Tree):
    """Returns a list of all possible trees that result from adding newleaf to Tree.""
    leftTree = Tree[1]
    rightTree = Tree[2]
    if leftTree == ():  # Tree is a leaf.
        newTree = ("Anc", newleaf, Tree)
        return [newTree]  # wrap it in a list!
General case: three steps at each node

```plaintext
1
2
3
addLeaf(newleaf, Tree)
```

outgroup  right  left
def addLeaf(newleaf, Tree):
    """Returns a list of all possible trees that result from adding newleaf to Tree."""

    leftTree = Tree[1]
    rightTree = Tree[2]

    if leftTree == ():  # Tree is a leaf.
        newTree = ("Anc", newleaf, Tree)
        return [newTree]  # wrap it in a list!

    else:
        outPutTreeL = []
        # put newleaf as outgroup
def addLeaf(newleaf, Tree):
    """Returns a list of all possible trees that result from adding newleaf to Tree."""

leftTree=Tree[1]
rightTree=Tree[2]

if leftTree==():  # Tree is a leaf.
    newTree = ("Anc", newleaf, Tree)
    return [newTree]  # wrap it in a list!
else:
    outputTreeL=[]
    # put newleaf as outgroup
    outputTreeL.append(("Anc", newleaf, Tree))
General case: three steps at each node

\[
\text{addLeaf(newleaf, Tree)}
\]

outgroup  right  left
The `addLeaf` chop shop: right tree

\[\text{addLeaf(newleaf, rightTree)}\]

http://bananajams.wordpress.com/
The `addLeaf` chop shop: left tree

```
1
/   \
2    3
```

```
1
/   \
2    4
```

```
1
/   \
2    3
```

```
1
/   \
2    4
```

http://bananajams.wordpress.com/
def addLeaf(newleaf, Tree):
    """Returns a list of all possible trees that result from adding newleaf to Tree."""
leftTree = Tree[1]
rightTree = Tree[2]
if leftTree == ():  # Tree is a leaf.
    newTree = ("Anc", newleaf, Tree)
    return [newTree]  # wrap it in a list!
else:
    outPutTreeL = []
    # put newleaf as outgroup
    outPutTreeL.append(("Anc", newleaf, Tree))
def addLeaf(newleaf, Tree):
    """Returns a list of all possible trees that result from adding newleaf to Tree."""
    leftTree = Tree[1]
    rightTree = Tree[2]

    if leftTree == ():  # Tree is a leaf.
        newTree = ("Anc", newleaf, Tree)
        return [newTree]  # wrap it in a list!
    else:
        outPutTreeL = []  # put newleaf as outgroup
        outPutTreeL.append(("Anc", newleaf, Tree))

        # recurse to add newleaf on branches of right subtree
        tempRightTreeL = addLeaf(newleaf, rightTree)
        for tempRightTree in tempRightTreeL:
            newTree = ("Anc", leftTree, tempRightTree)
            outPutTreeL.append(newTree)
```python
def addLeaf(newleaf, Tree):
    """Returns a list of all possible trees that result from
    adding newleaf to Tree."""
    leftTree = Tree[1]
    rightTree = Tree[2]
    if leftTree == ():  # Tree is a leaf.
        newTree = ("Anc", newleaf, Tree)
        return [newTree]  # wrap it in a list!
    else:
        outPutTreeL = []
        # put newleaf as outgroup
        outPutTreeL.append(("Anc", newleaf, Tree))

        # recurse to add newleaf on branches of right subtree
        tempRightTreeL = addLeaf(newleaf, rightTree)
        for tempRightTree in tempRightTreeL:
            newTree = ("Anc", leftTree, tempRightTree)
            outPutTreeL.append(newTree)

        # recurse to add newleaf on branches of left subtree
        tempLeftTreeL = addLeaf(newleaf, leftTree)
        for tempLeftTree in tempLeftTreeL:
            newTree = ("Anc", tempLeftTree, rightTree)
            outPutTreeL.append(newTree)

    return outPutTreeL
```

Demo!
Homework problem: allTrees

def allTrees(leafNameL):
    """Given a list of species, returns a list of all possible tree topologies."""

    
    >>> allTrees( [1,2,3] )
    [
        ('Anc', (1, (), ()), ('Anc', (2, ((), ())), (3, ((), ()))),
        ('Anc', ('Anc', (1, (), ())), (2, ((), ()))), (3, ((), ()))),
        ('Anc', (2, ((), ())), ('Anc', (1, (), ())), (3, ((), ()))))
    ]
allTrees([4, 3, 2, 1])
allTrees([1])

allTrees([2, 1])

allTrees([3, 2, 1]) <- allTrees([4, 3, 2, 1])
allTrees([1])

allTrees([1, 1])

allTrees([1, 2, 1])

allTrees([3, 2, 1])

allTrees([4, 3, 2, 1])
allTrees([1])

allTrees([2, 1])

allTrees([3, 2, 1])

allTrees([4, 3, 2, 1])
allTrees( [3, 2, 1])

allTrees( [2, 1])

allTrees( [1])

allTrees( [4, 3, 2, 1])
One general strategy for inferring phylogenies

- Generate all possible trees
- Pick the most parsimonious given some data
The number of trees grows quickly...

<table>
<thead>
<tr>
<th>Num. Species (leaves)</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. possible trees</td>
<td>1</td>
<td>105</td>
<td>34,459,425</td>
<td>2.13458 x 10^{14}</td>
<td>8.200795 x 10^{21}</td>
<td>1.192568 x 10^{30}</td>
</tr>
</tbody>
</table>
Programming motifs: all vs. all

protsA = ['PLLYK', 'QSTE', 'NITQIVG', 'INE', 'QVAEA', 'YMSA']
protsB = ['LAGADLEQ', 'LAL', 'EAMERY', 'ENLEL']

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Programming motifs: all vs. all

protsA = ['PLLYK', 'QSTE', 'NITQIVG', 'INE', 'QVAEA', 'YMSA']
protsB = ['LAGADLEQ', 'LAL', 'EAMERY', 'ENLEL']

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

scD = {}
for pA in protsA:
    for pB in protsB:
        scD[(pA,pB)] = memoAlignScore(protSeq1, protSeq2, -9, blosum62, {})
Programming motifs: running the gauntlet

```
rnaL = ['AUGACGCAGUAGUCA', 'UAGACAGUA', 'AGGUACAUUC' ...]
```

- If no RNA has a fold score above 7, return False
- Otherwise return True
Programming motifs: running the gauntlet

```python
rnaL = ['AUGACGCAGUAGUCA', 'UAGACAGUA', 'AGGUACAUC', ...]

• If no RNA has a fold score above 7, return False
• Otherwise return True

for rna in rnaL:
    if fold(rna) > 7:
        return True

return False
```
Programming motifs: finding extremz

dictionary = ['abdomen', 'abdominal', 'abduct', 'abduction', 'aberration', 'abet', 'abhor', 'abhorrence', 'abhorrent', 'abide', 'abiding', 'ability', 'abject', 'ablaze', ...
...

... }

... etc.

... ...

...]

def z(input):
    '''Count z's in a string'''
    counter = 0
    for symbol in input:
        if symbol == 'z':
            counter = counter + 1
    return(counter)
def extremz(wordList):
    '''Find and return the word with the most z's'''
    bestCount=0
    bestWord=""
    for word in wordList:
        count=z(word)
        if count>bestCount:
            bestCount=count
            bestWord=word
    return bestWord
Recursion on trees: \texttt{graft}

\begin{verbatim}
Groodies = ("Q",
            ("R", (), ()),
            ("S",
              ("T", (), ()),
              ("U", (), ()))
            )

UTree = ("U",
            ("V", (), ()),
            ("W", (), ())
            )

>>> graft(Groodies, UTree)
('Q', ('R', (), ()), ('S', ('T', (), ()), ('U', ('V', (), ()), ('W', (), ()))))

>>> graft(Groodies, ('X',(),()))
('Q', ('R', (), ()), ('S', ('T', (), ()), ('U', (), ())))
\end{verbatim}

- at most one leaf of \texttt{primaryTree} has the same name as the root of \texttt{graftTree}.
- If there is one such match, the function returns a new tree that is identical to \texttt{primaryTree} but with that leaf in \texttt{primaryTree} replaced by the entire \texttt{graftTree}.
- If there is no leaf in \texttt{primaryTree} that matches the name of the root of \texttt{graftTree}, the function simply returns \texttt{primaryTree}.
- no internal node of the \texttt{primaryTree} will have a name that matches the root of the \texttt{graftTree}. 
def graft(primaryTree, graftTree):

    """If primaryTree has a leaf whose name is the same as the root of graftTree then we return a new tree identical to primaryTree except with that leaf replaced by graftTree. Otherwise, we just return primaryTree."

    """
def graft(primaryTree, graftTree):
    """If primaryTree has a leaf whose name is the same as the root of
    graftTree then we return a new tree identical to primaryTree
    except with that leaf replaced by graftTree. Otherwise, we
    just return primaryTree.
    """

    if primaryTree[0] == graftTree[0]:
        return graftTree

    elif primaryTree[1] == ():
        return primaryTree

    else:
        leftgr = graft(primaryTree[1], graftTree[1])
        rightgr = graft(primaryTree[2], graftTree[2])
        return (primaryTree[0], leftgr, rightgr)