Entertainment: CS 5 Green book to be made into feature-length movie starring George Clooney as the happy turtle.

News in Brief

HMC CS 5 Green Professors discover new discovery

Claremont, CA: Researchers at Harvey Mudd College have made an extremely important new discovery said a spokesperson for the College. The discovery was evidently discovered while the researchers were trying to discover another discovery. “The professors discovered that their discovery had not been previously discovered, which is an important discovery in its own right,” said the excited spokesperson. A number of prominent scientists also expressed their tremendous enthusiasm and said that they looked forward to reading what was actually discovered.

New computer program predicts today’s weather: IndexError: list index out of range

Sports: CS 5 Green Prof runs to class in record time. Were steroids involved?
A request...

On your worksheet, could you please answer:

How many hours, outside of lecture, did you spend on this course this past week?

(If you went to lab, you can count that as 'outside' for the purposes of this question.)
Trees and the parsimony principle

Data

1 CCAGT
2 GCACT
3 CCTGA
4 GCTGT
Trees and the parsimony principle

“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,(),()) )
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

```python
>>> 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

1. addLeaf(newleaf, Tree)

2. Outgroup

3. Right

4. 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

```plaintext
addLeaf(newleaf, Tree)
```

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

```
1
  \  \\
  1   2
  \    \\
  3    4
```

```
1
  \  \\
  1   2
  \    \\
  3    4
```

```
1
  \  \\
  1   2
  \    \\
  3    4
```

```
1
  \  \\
  1   2
  \    \\
  3    4
```

```
1
  \  \\
  1   2
  \    \\
  3    4
```

```
1
  \  \\
  1   2
  \    \\
  3    4
```

```
1
  \  \\
  1   2
  \    \\
  3    4
```

```
1
  \  \\
  1   2
  \    \\
  3    4
```

```
1
  \  \\
  1   2
  \    \\
  3    4
```

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

```
1

2

3

4

→ addLeaf(newleaf, leftTree)

1

2

3

4

→

1

2

4

→

1

2

4

→

1

2

4

→

1

2

4

→

1

2

4

→

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

allTrees([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])
allTrees([1])

allTrees([2, 1])

allTrees([3, 2, 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

```python
protsA = ['PLLKY', '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

\[
\text{rnaL} = ['\text{AUGACGCA}G\text{UAGUCA}', '\text{UAGACAGUA}', '\text{AGGUACAU}C', \ldots]
\]

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

```python
for rna in rnaL:
    if fold(RNA) > 7:
        return True
return False
```

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

- If no RNA has a fold score above 7, return False
- Otherwise return True
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: graft

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', (), ())))

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

Try not to use any helper functions on this one!