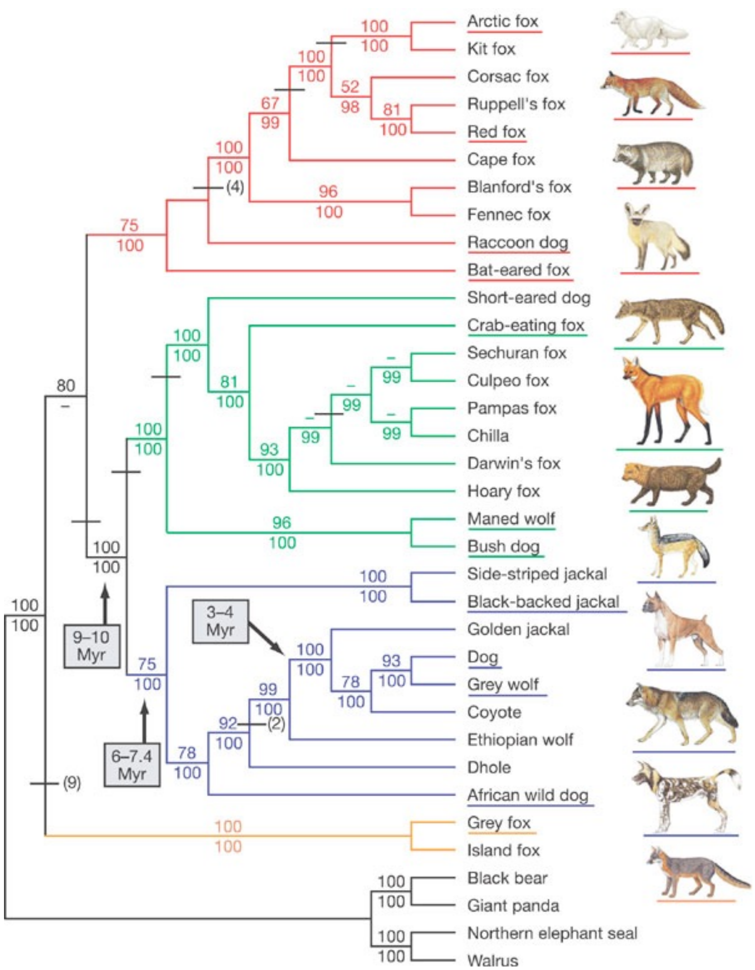
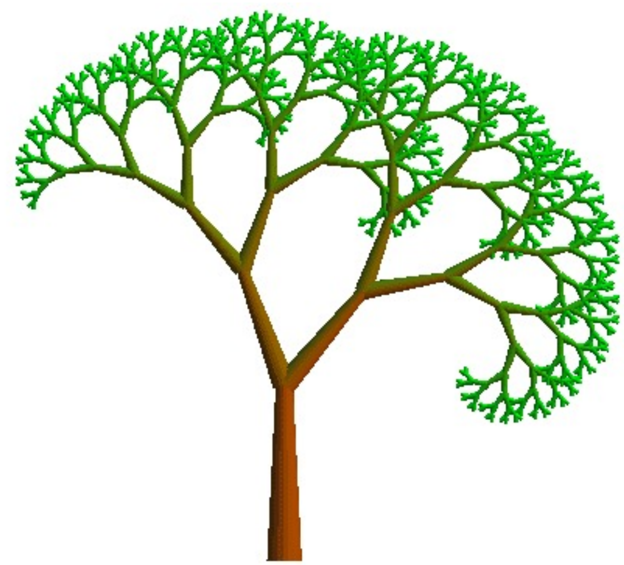
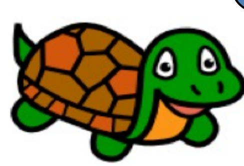
“I wonder about Trees” –Robert Frost



“We wonder about Robert Frost” - Trees



Hey! How come no turtles in this tree?

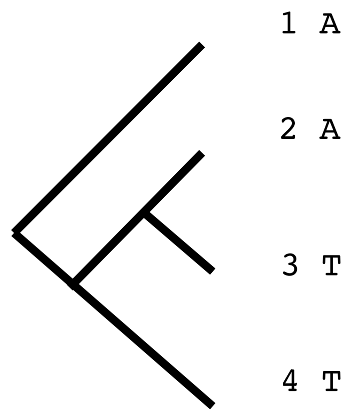


CS 5 Green

Learning Goals

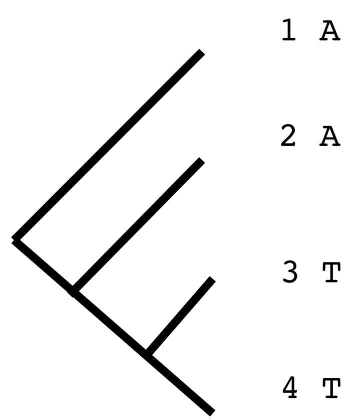
- Describe the parsimony principle
- Introduce method for enumerating all trees

Trees and the parsimony principle



Data

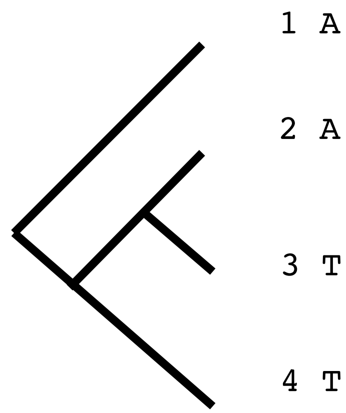
1 CC**A**GT
2 GC**A**CT
3 CC**T**GA
4 GC**T**GT

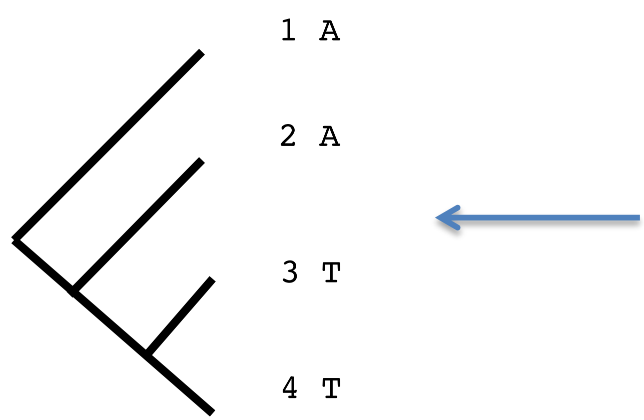


Trees and the parsimony principle

Data

1 CC**A**GT
2 GC**A**CT
3 CC**T**GA
4 GC**T**GT



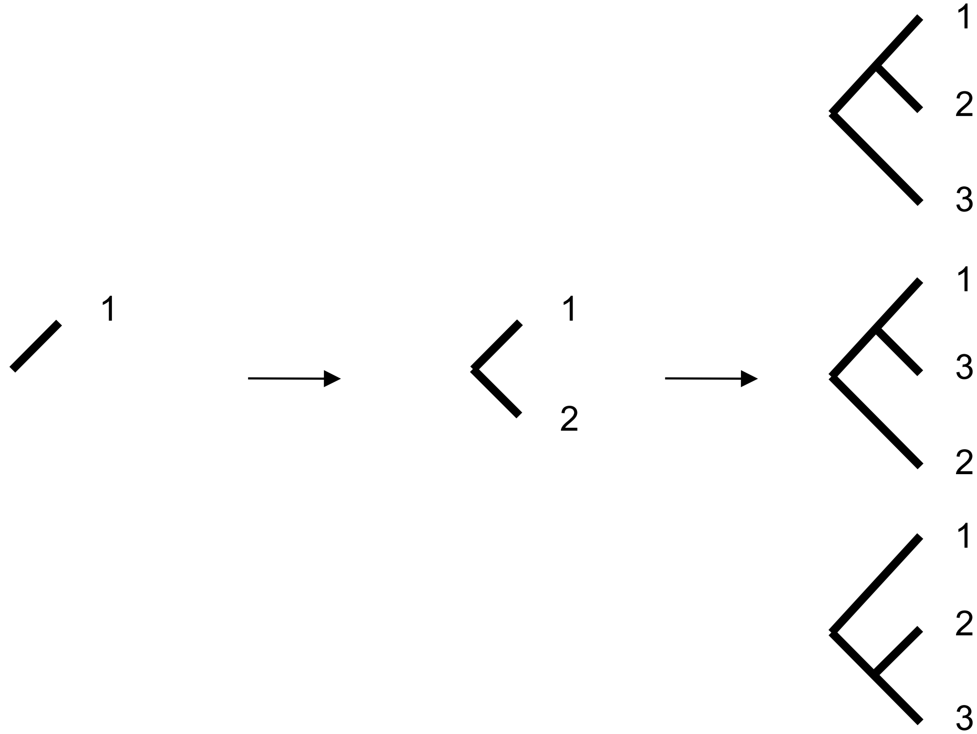


"Better", i.e. more parsimonious

Another general strategy for inferring phylogenies

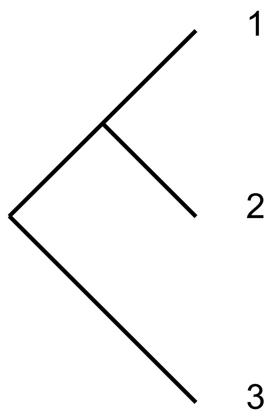
- Generate all possible trees
- Pick the most parsimonious given some data

Generating all possible trees from the ground up

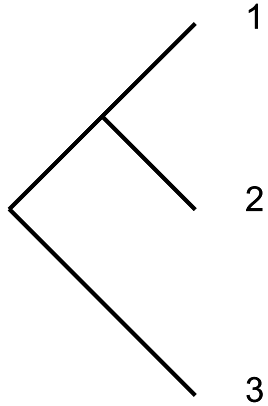


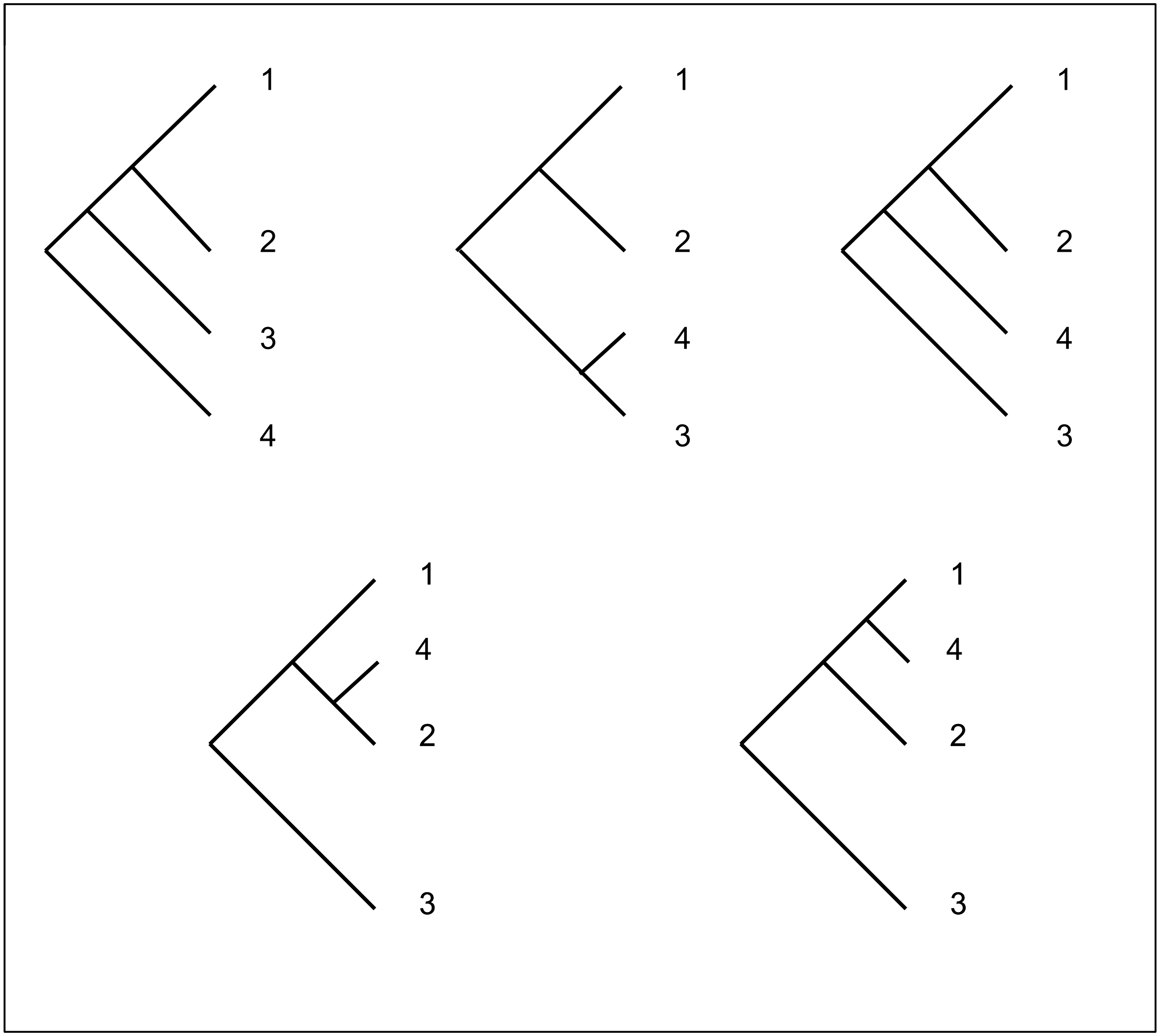
# species (leaves)	1	2	3	4
# possible trees	1	1	3	☐

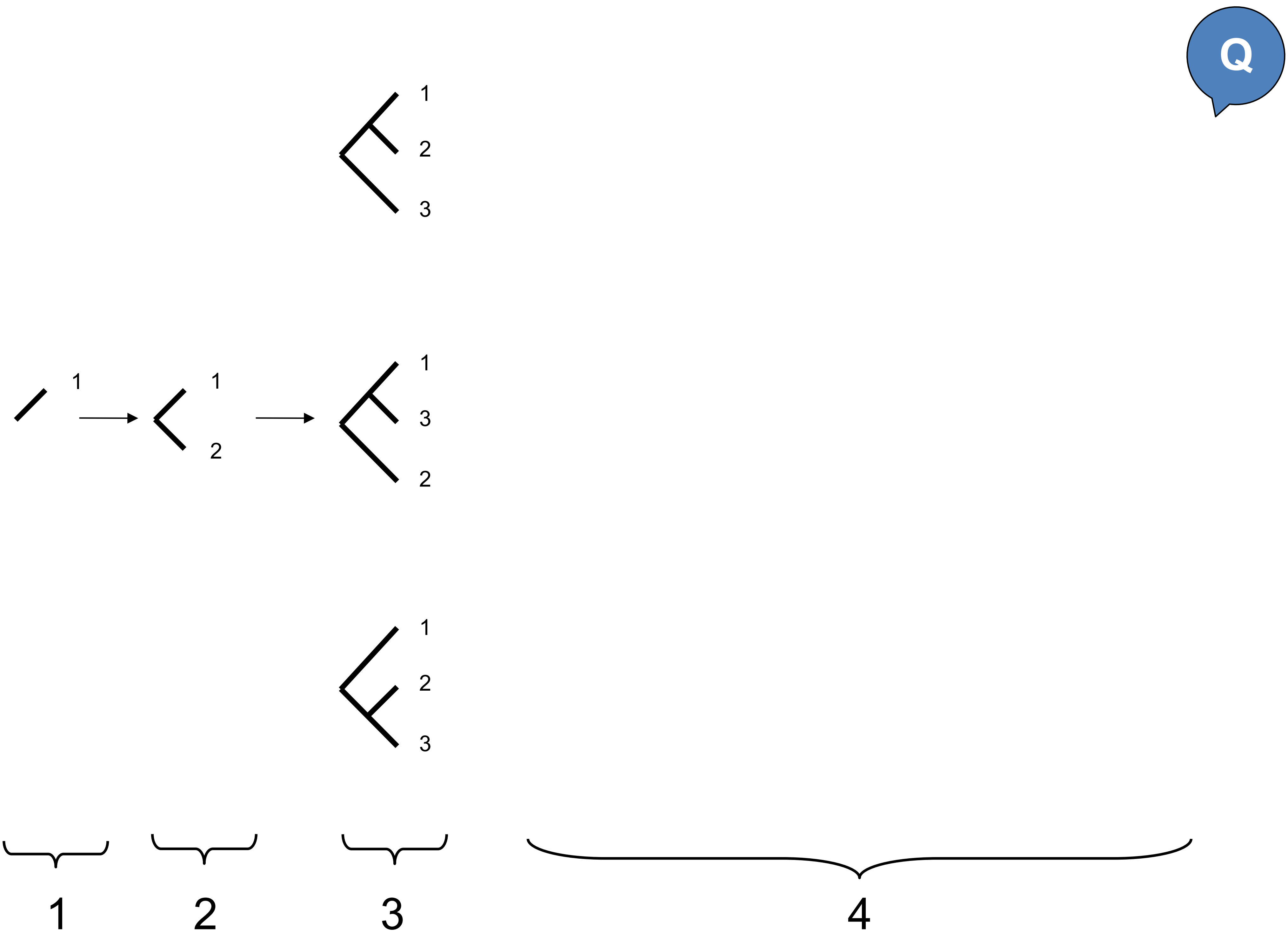
Draw all possible trees that result from adding a species 4 to this tree.



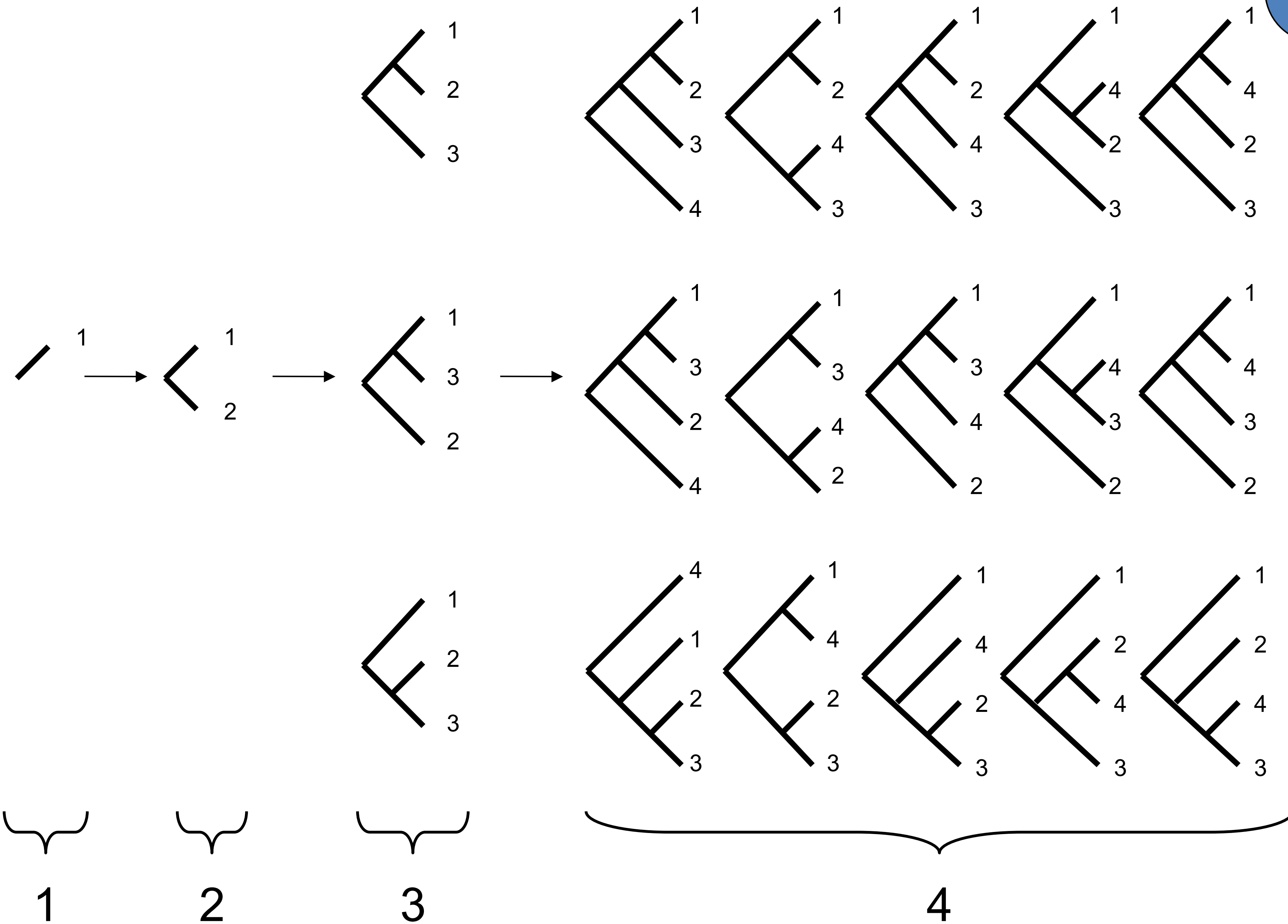
Draw all possible trees that result from adding a species 4 to this tree.





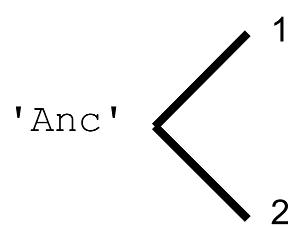


Q
1
2
3
1
1
2
1
3
2
1
2
3
1
2
3
4

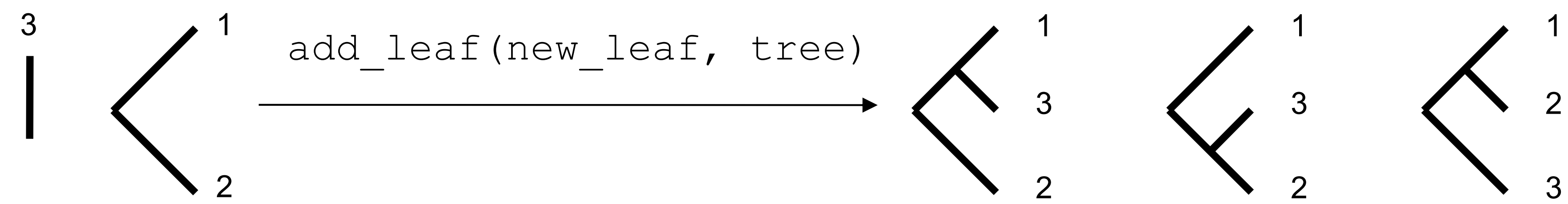

1
2
3
4

A convention for naming internal/ancestral nodes

```
tree = ( 'Anc', (1,(),()) , (2,(),()) )
```



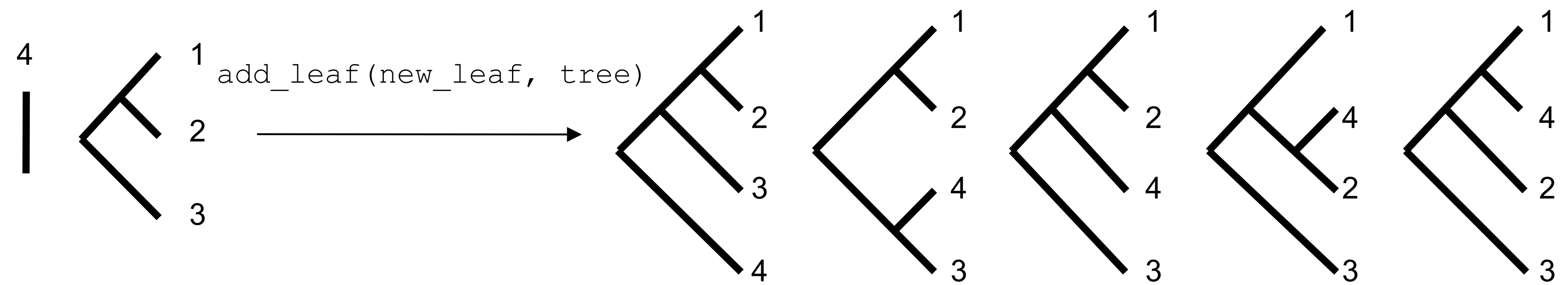
The `add_leaf` function




```
>>> leaf = ( 3, (), () )
>>> tree = ( 'Anc', (1,(),()) , (2,(),()) )
>>> add_leaf(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 `add_leaf` function

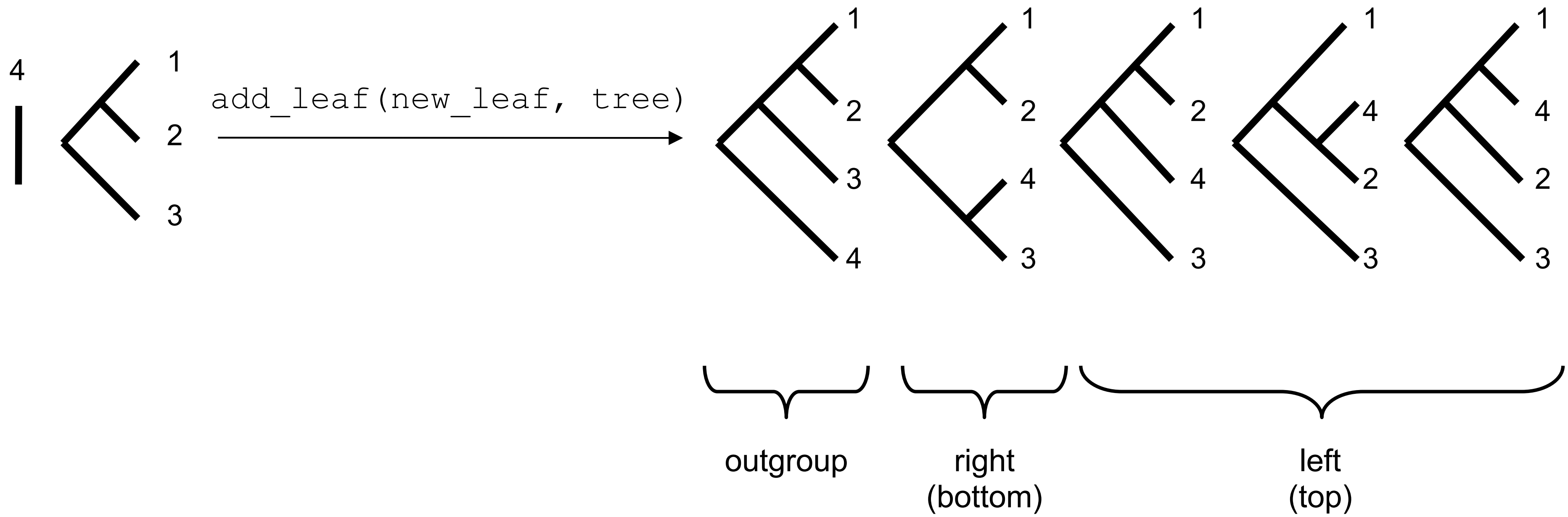



```
>>> leaf = ( 4, (), () )
>>> tree = ('Anc', (3, (), ()) , ('Anc', (1, (), ()), (2, (), ())) )
>>> add_leaf(leaf, tree)
[
 ('Anc', (4, (), ()), ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ())))),
 ('Anc', ('Anc', (4, (), ()), (3, (), ())), ('Anc', (1, (), ()), (2, (), ()))),
 ('Anc', (3, (), ()), ('Anc', (4, (), ()), ('Anc', (1, (), ()), (2, (), ())))),
 ('Anc', (3, (), ()), ('Anc', ('Anc', (4, (), ()), (1, (), ())), (2, (), ()))),
 ('Anc', (3, (), ()), ('Anc', (1, (), ()), ('Anc', (4, (), ()), (2, (), ()))))
]
```

```python
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from
    adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
```

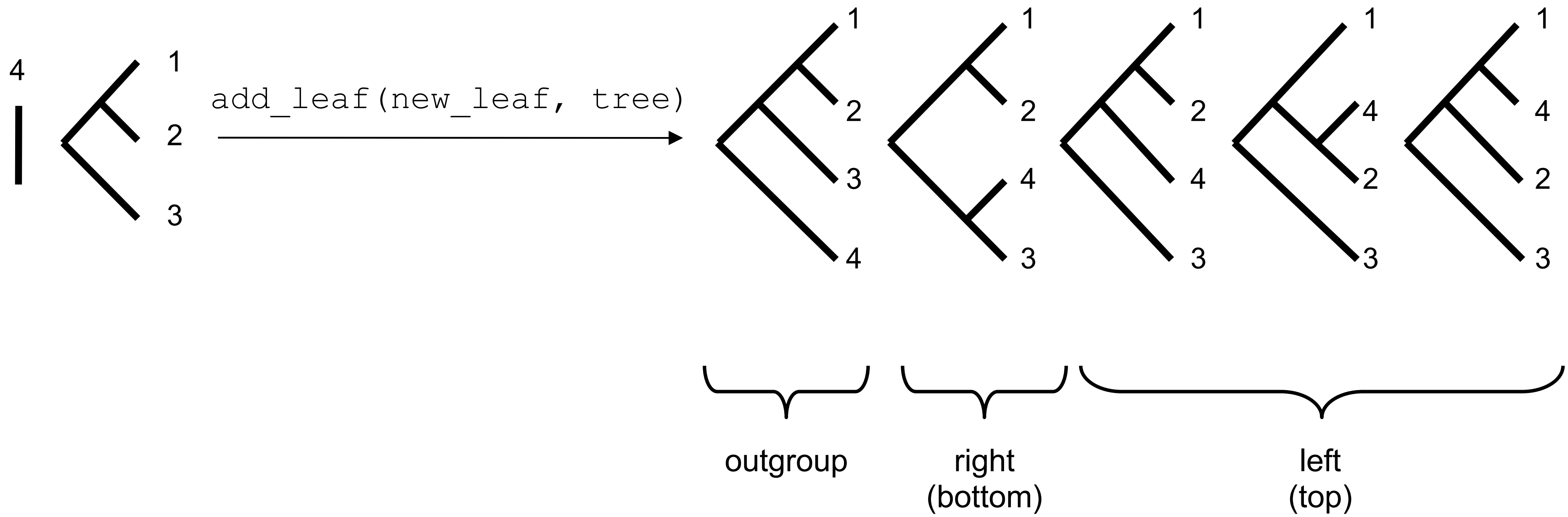
```python
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from
    adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == (): # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree] # wrap it in a list!
```

General case: three steps at each node

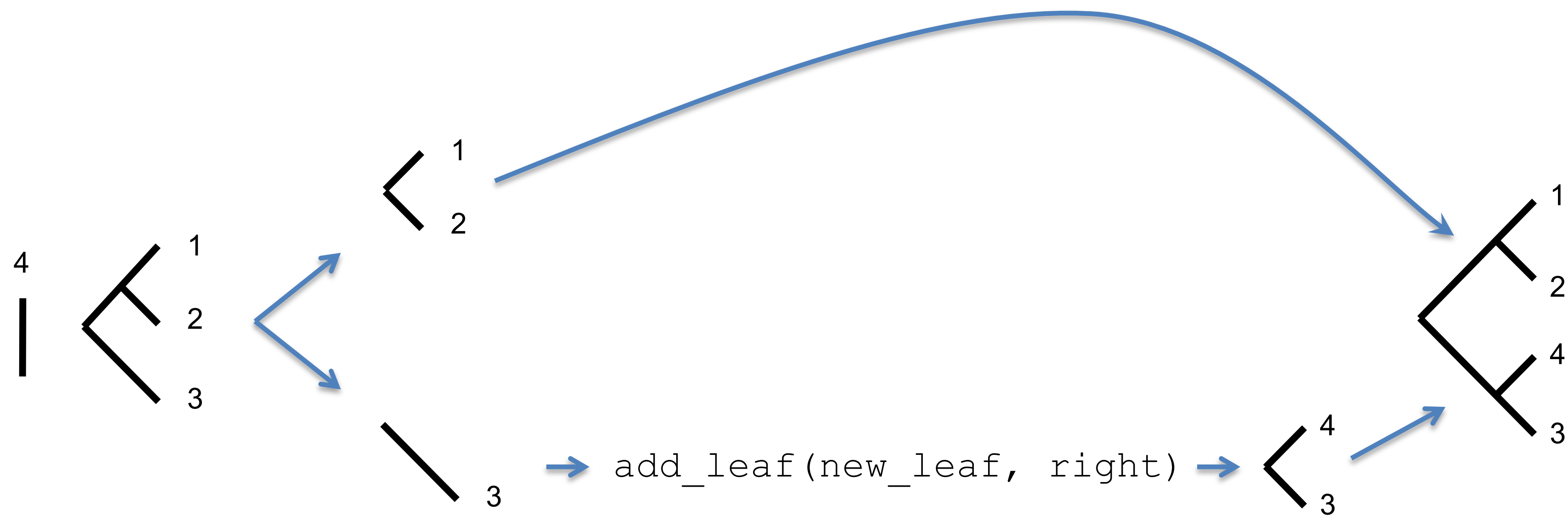


```
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from
    adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == (): # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree] # wrap it in a list!
    else:
        output_trees = []

        # put new_leaf as outgroup
        output_trees.append((anc, new_leaf, tree))
```

General case: three steps at each node

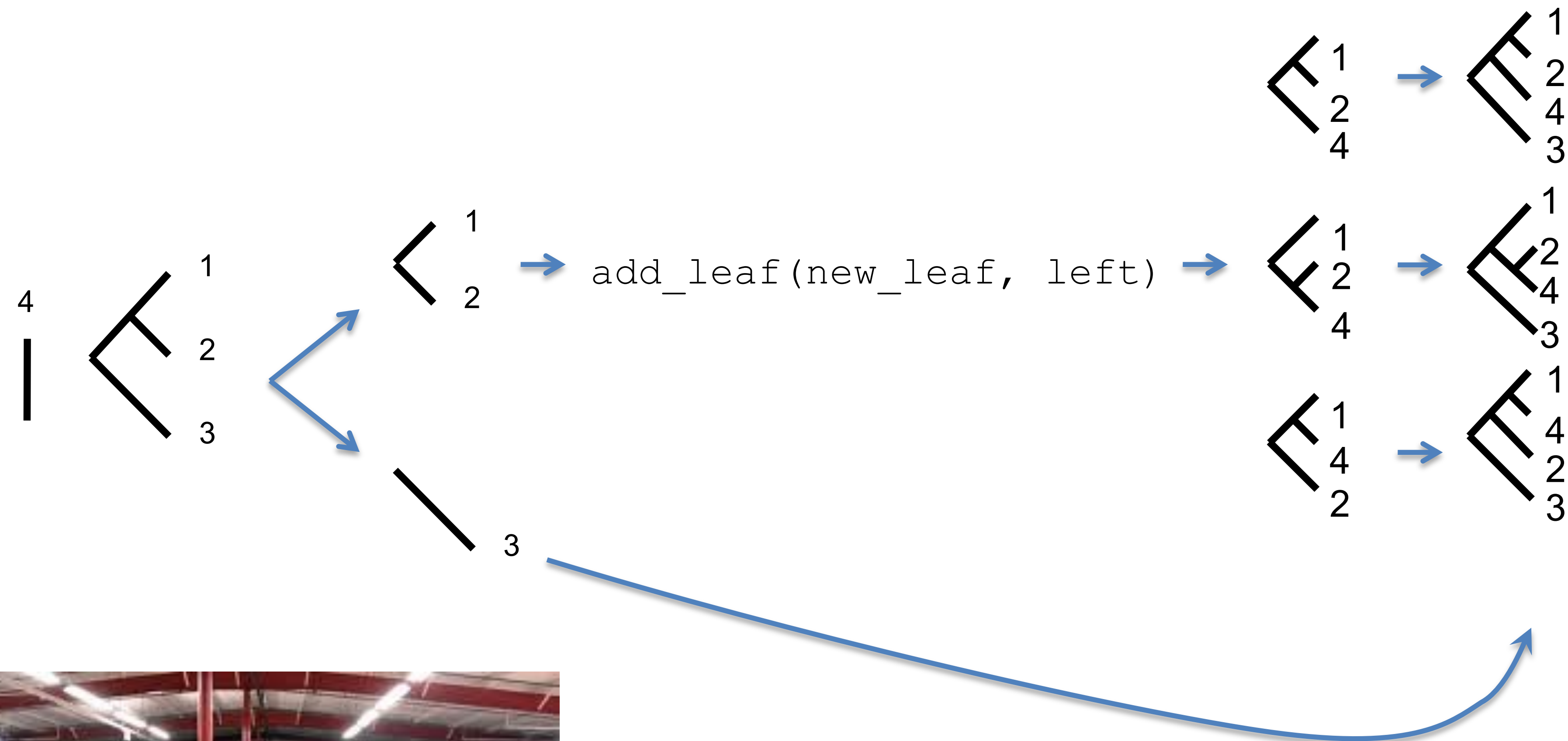


The `add_leaf` chop shop: right tree



http://bananajams.wordpress.com/

The `add_leaf` chop shop: left tree



http://bananajams.wordpress.com/

Worksheet

Q

```
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from
    adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == (): # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree] # wrap it in a list!
    else:
        output_trees = []

        # put new_leaf as outgroup
        output_trees.append((anc, new_leaf, tree))
```

Worksheet

```python
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from
    adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == (): # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree] # wrap it in a list!
    else:
        output_trees = []

        # put new_leaf as outgroup
        output_trees.append((anc, new_leaf, tree))

        # recur to add new_leaf on branches of right subtree
        temp_right_trees = add_leaf(new_leaf, right)
        for temp_right_tree in temp_right_trees:
            new_tree = (anc, left, temp_right_tree)
            output_trees.append(new_tree)
```

Worksheet

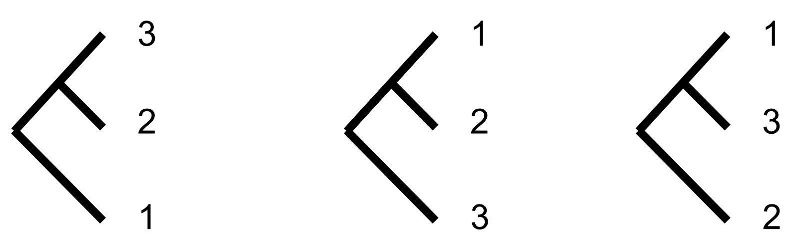
```python
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from
    adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == (): # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree] # wrap it in a list!
    else:
        output_trees = []

        # put new_leaf as outgroup
        output_trees.append((anc, new_leaf, tree))

        # recur to add new_leaf on branches of right subtree
        temp_right_trees = add_leaf(new_leaf, right)
        for temp_right_tree in temp_right_trees:
            new_tree = (anc, left, temp_right_tree)
            output_trees.append(new_tree)

        # recur to add new_leaf on branches of left subtree
        temp_left_trees = add_leaf(new_leaf, left)
        for temp_left_tree in temp_left_trees:
            new_tree = (anc, temp_left_tree, right)
            output_trees.append(new_tree)

        return output_trees
```

Demo!

(Bonus) homework problem: `all_trees`

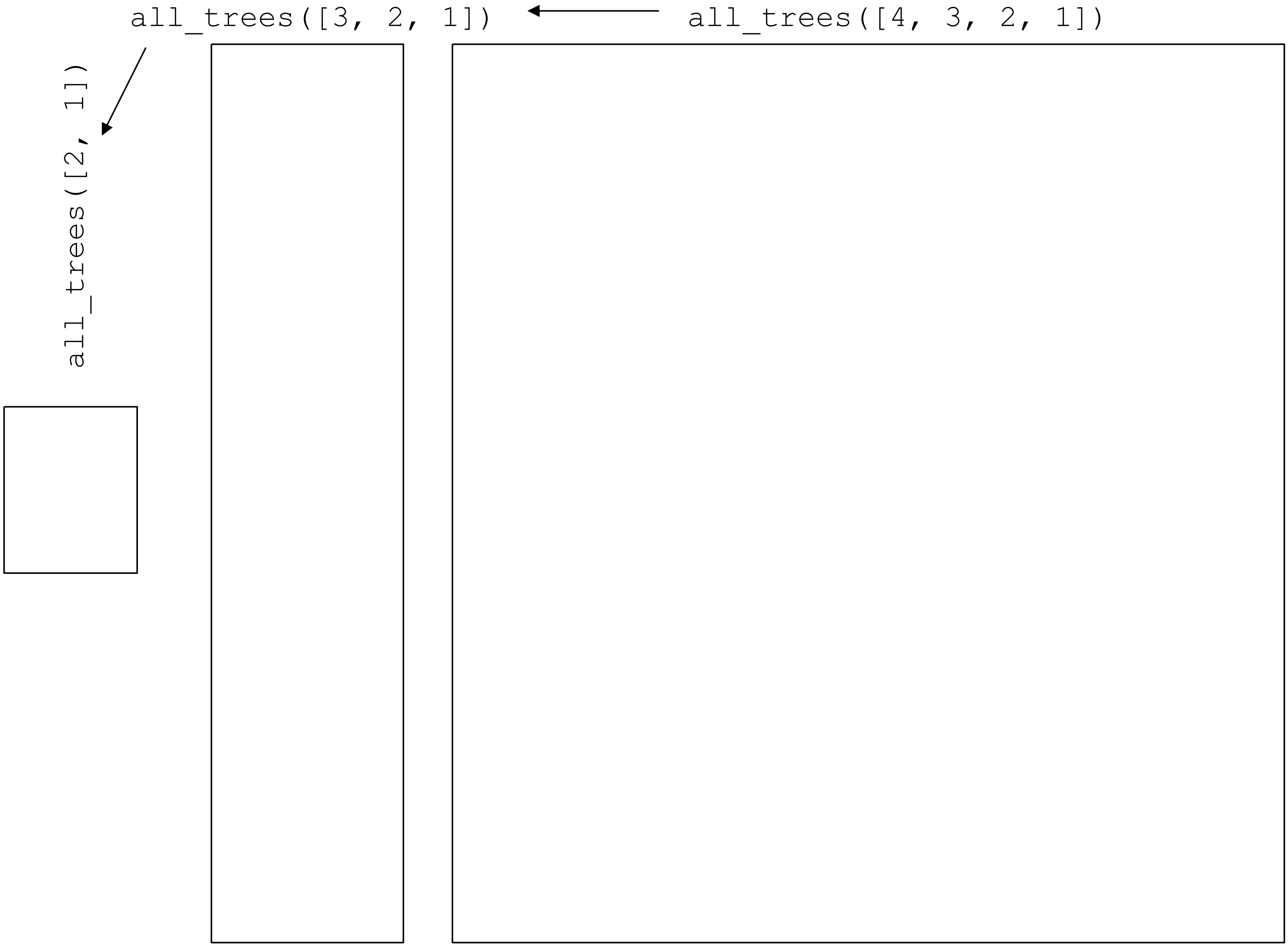
```
def all_trees(leaf_names):
    """Given a list of species, returns a list of all
    possible tree topologies."""
```

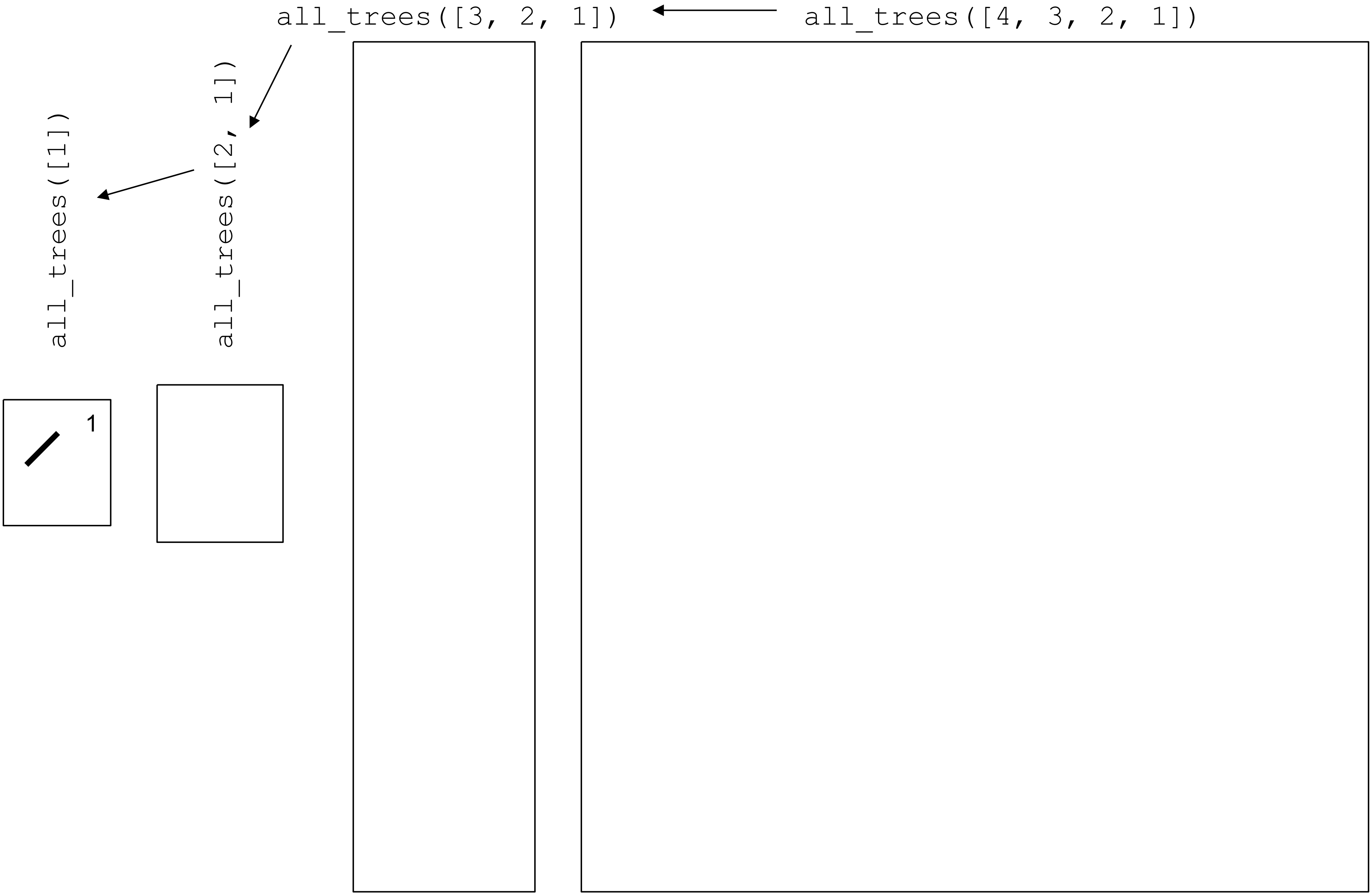
```
>>> all_trees( [1,2,3] )
[
 ('Anc', (1, (), () ) , ('Anc', (2, (), ()), (3, (), ()))),
 ('Anc', ('Anc', (1, (), ()), (2, (), ())), (3, (), ())),
 ('Anc', (2, (), ()), ('Anc', (1, (), ()), (3, (), ())))
]
```

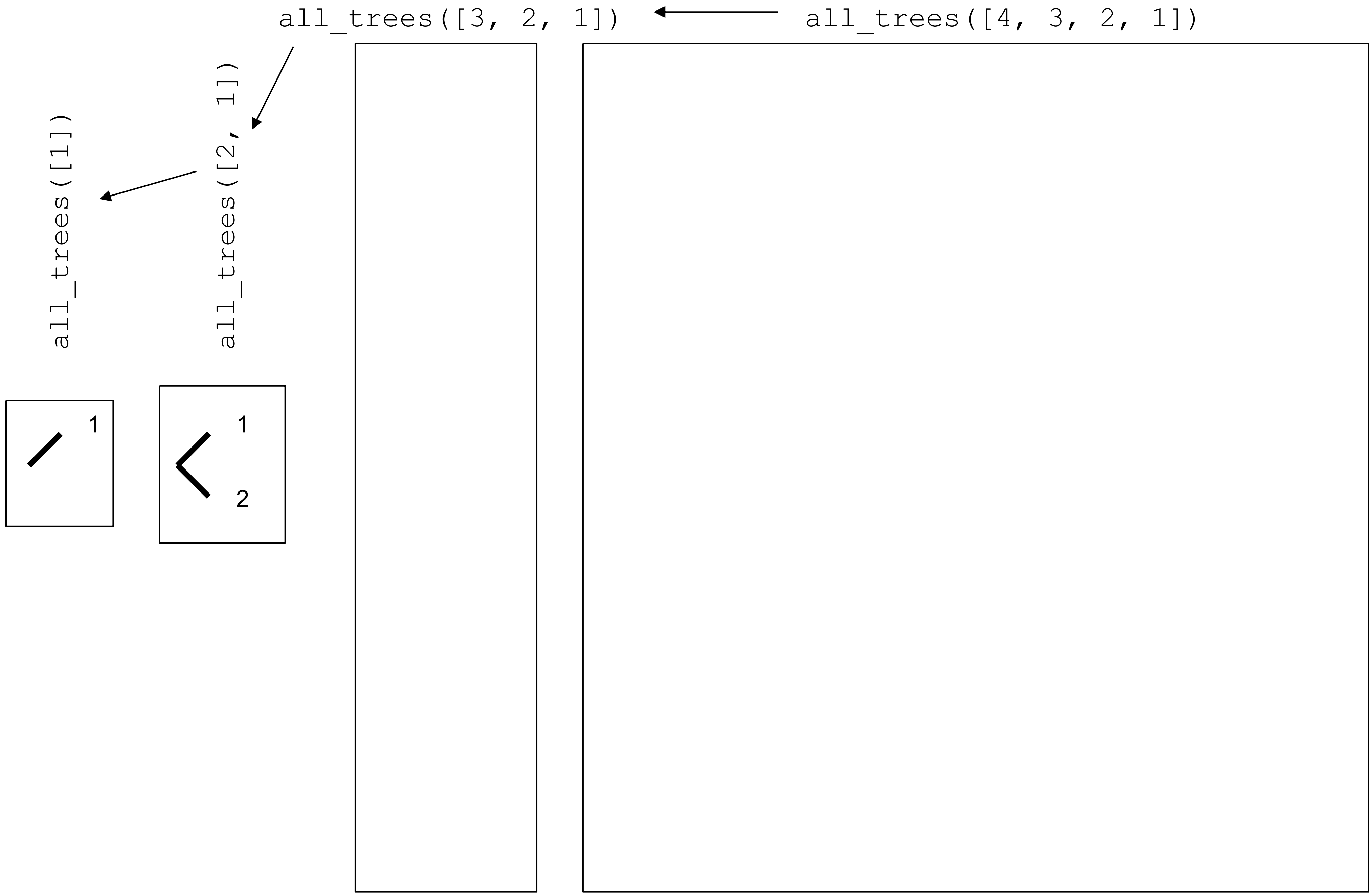


all_trees([4, 3, 2, 1])

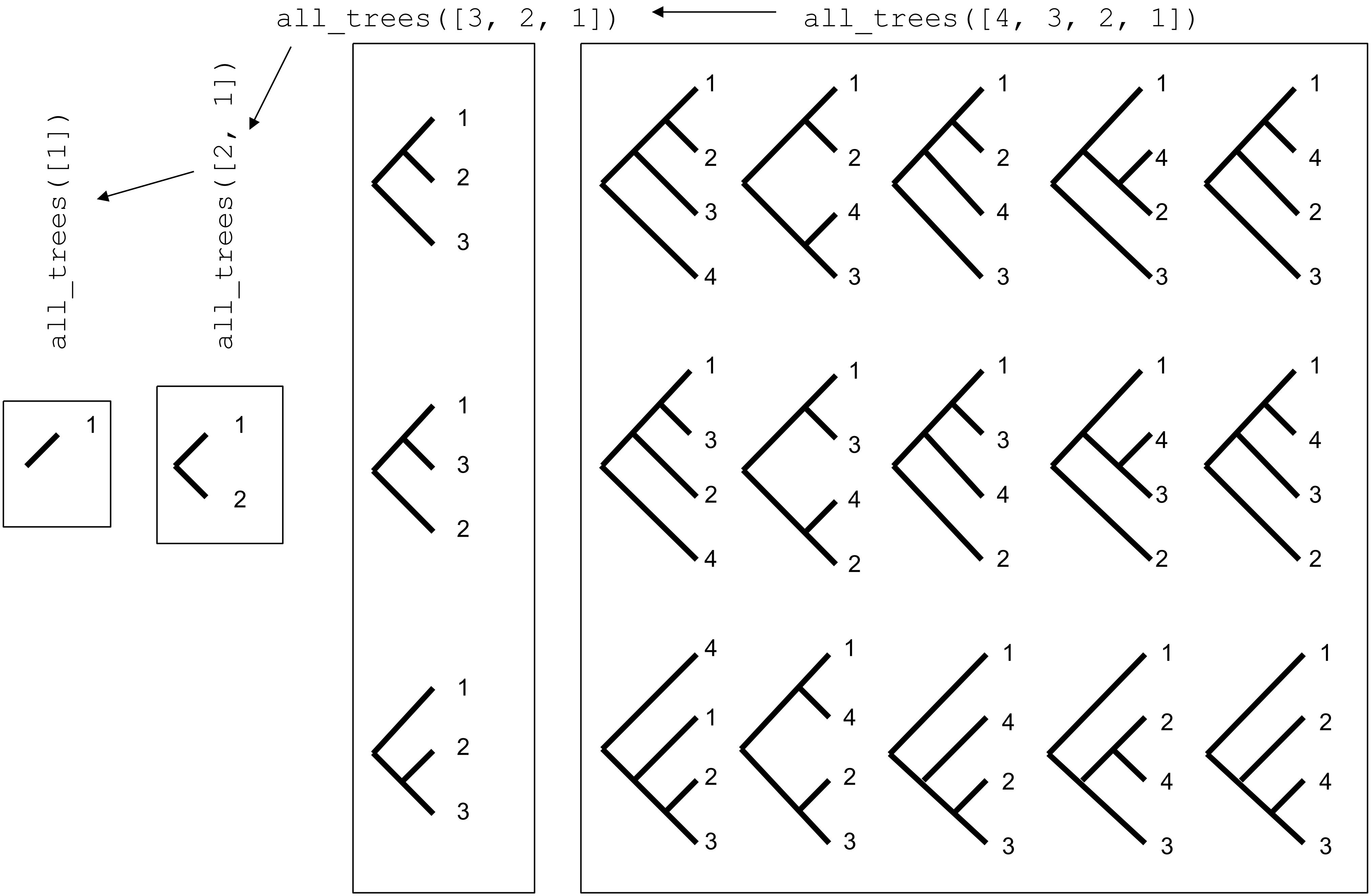
`all_trees([3, 2, 1])` ← `all_trees([4, 3, 2, 1])`


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


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


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


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



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

One general strategy for inferring phylogenies

- Generate all possible trees
- Pick the most parsimonious given some data

The number of trees grows quickly…

# species (leaves)	1	5	10	15	20	25
# possible trees	1	105	34,459,425	2.13458 x 10^14	8.200795 x 10^21	1.192568 x 10^30


Programming motifs: all vs. all

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

	B1	B2	B3	B4
A1				
A2				
A3				
A4				
A5				
A6				

```
d = {}
for pA in protsA:
    for pB in protsB:
        d[(pA,pB)] = memoAlignScore(pA, pB, -9, blosum62, {})
```

Programming motifs: running the gauntlet

```
rnas = ['AUGACGCAGUAGUCA', 'UAGACAGUA', 'AGGUACAUC'...]
```

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

```
for rna in rnas:
    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(words):
    '''Find and return the word with the most z's'''
    best_count = 0
    best_word = ""
    for word in words:
        count = z(word)
        if count > best_count:
            best_count = count
            best_word = word
    return best_word
```

Recursion on trees: `graft`

```
groodies = ("Q",
               ("R", (), ()),
               ("S",
                    ("T", (), ()),
                    ("U", (), ())
               )
             )
```

```
utree = ("U",
            ("V", (), ()),
            ("W", (), ())
         )
```

primary tree   graft tree

```
>>> graft(groodies, utree)
('Q', ('R', (), ()), ('S', ('T', (), ()), ('U', ('V', (), ()), ('W', (), ()))))

>>> graft(groodies, ('X',(),()))
('Q', ('R', (), ()), ('S', ('T', (), ()), ('U', (), ())))
```

- At most one *leaf* of `primary_tree` has the same name as the root of `graft_tree`.
- If there is one such match, the function returns a new tree that is identical to `primary_tree` but with that leaf in `primary_tree` replaced by the entire `graft_tree`.
- If there is no leaf in `primary_tree` that matches the name of the root of `graft_tree`, the function simply returns `primary_tree`.
- No internal node of the `primary_tree` will have a name that matches the root of the `graft_tree`.

Q

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

Try not to use any helper functions on this one!

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

    root, left, right = primary_tree
    if root == graft_tree[0]:
        return graft_tree

    elif left == ():
        return primary_tree

    else:
        left_graft = graft(left, graft_tree)
        right_graft = graft(right, graft_tree)
        return (root, left_graft, right_graft)
```

Try not to use any helper functions on this one!

Reminder:

- Lecture feedback form (https://forms.gle/aPmkpXDUTp4Xo4CV7)