Trees + Recursion Review Session

CS61A • Spring 2018 • Karina & Dennis

Recursion

Recursion...

... when you call a function from inside itself!

What could possibly go wrong?

```
>>> def recursion(recurse):
...     return recursion(recurse)
```

Recursion...

... when you call a function from inside itself!

What could possibly go wrong?

```
>>> def recursion(recurse):
...     return recursion(recurse)
```

Two parts of recursion

1. Base case
2. Recursive leap of faith

(d) **(4 pt)** Implement `count_sums` which counts the number of ways that a positive integer `n` can be partitioned into a subset of the positive values `m, f(m), f(f(m)), ...` for a shrinking function `f`. **No negative values or repeated values can be included in the sum.**

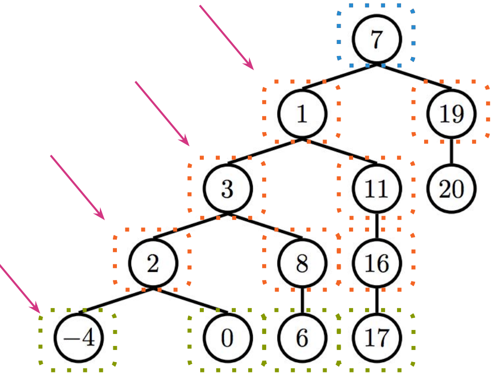
```
def count_sums(n, f, m):
    """Return the number of ways that n can be partitioned into unique positive
    values obtained by applying the shrinking function f repeatedly to m.

    >>> count_sums(6, lambda k: k-1, 4) # 4+2, 3+2+1
    2
    >>> count_sums(12, lambda k: k-2, 12) # 12, 10+2, 8+4, 6+4+2
    4
    >>> count_sums(11, lambda k: k//2, 8) # 8+2+1
    1
    """
    if n == 0:
        return 1
    elif m <= 0 or n < 0:
        return 0
    else:

        a = ________________________________________________________________


        b = _________________________________________________________________

        return a + b
```

(d) (4 pt) Implement count_sums which counts the number of ways that a positive integer n can be partitioned into a subset of the positive values m, f(m), f(f(m)), ... for a shrinking function f. **No negative values or repeated values can be included in the sum.**

```
def count_sums(n, f, m):
    """Return the number of ways that n can be partitioned into unique positive
    values obtained by applying the shrinking function f repeatedly to m.

    >>> count_sums(6, lambda k: k-1, 4) # 4+2, 3+2+1
    2
    >>> count_sums(12, lambda k: k-2, 12) # 12, 10+2, 8+4, 6+4+2
    4
    >>> count_sums(11, lambda k: k//2, 8) # 8+2+1
    1
    """
    if n == 0:
        return 1
    elif m <= 0 or n < 0:
        return 0
    else:

        a = count_sums (n -m , f , f ( m ))

        b = count_sums (n , f , f ( m ))

        return a + b
```

Trees

Terminology



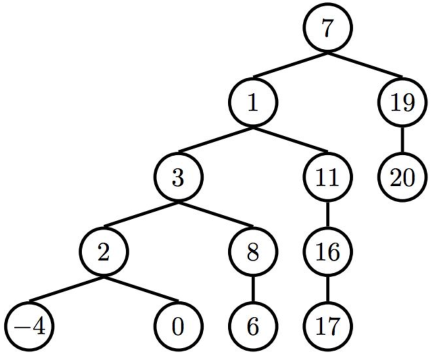
- branches down; root is at *top* and leaves are at bottom
- **parent node**: node w/branches
  - can have multiple branches
- **branch node:** node w/a parent
  - previously known as "child"
  - each "child"/branch node can only have one parent
- **root**: v top of ur tree
- **label:** value *inside* of the node
- **leaf**: node w/no branches

branches: the subtree that extends from the branch node

Slides thanks to Annie :`)

Terminology

- **depth**: # of edges from root of tree to the node
  - think: how many edges in the *path* from root -> node X?
- **height**: depth of the furthest/lowest leaf
  - aka, maximum depth in the tree!



Trees are an Abstract Data Type

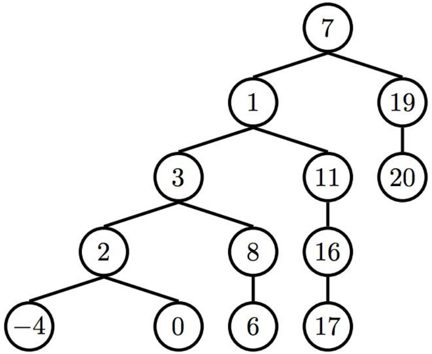
(a Concept™ that you choose the implementation for)

Implementation

ABSTRACTION!!

- **Constructors**
  - how would you *create* a tree, using only the knowledge/information of what a tree contains? (labels & branches)
  - think of the domain (input) and abstracted range (expected output)

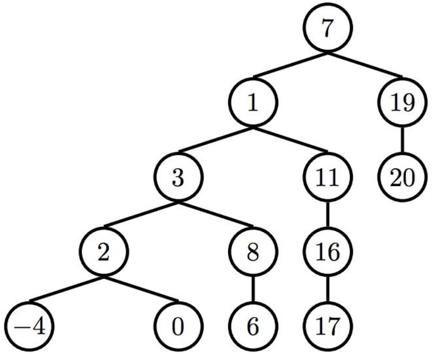
```
def tree(label, branches=[]):
    # returns a Tree of some form
```



Implementation

ABSTRACTION!!

- **Selectors**
  - how do you grab/extract information from this ~tree~ you've just created, without knowing the implementation details?



```
def label(tree):
    # returns the label of your root node in the Tree
def branches(tree):
    # returns the branches in your Tree
def is_leaf(tree):
    # tells you if a node is a leaf or not
```

Slides thanks to Annie :`)

Practice Time

(a) **(3 pt)** Implement `bigpath`, which takes a `Tree` instance `t` and an integer `n`. It returns the number of paths in `t` whose sum is *at least* `n`. Assume that all node values of `t` are integers.

```
def bigpath(t, n):
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> bigpath(t, 3)
    3
    >>> bigpath(t, 6)
    2
    >>> bigpath(t, 9)
    1
    """


    if t.is_leaf():


        return one(________________________________________________)


    return sum([____________________________________________________])
```

Definition. A path through a Tree is a list of adjacent node values that starts with the root value and ends with a leaf value.

```
def one(b):
    if b:
        return 1
    else:
        return 0
```

(a) **(3 pt)** Implement `bigpath`, which takes a `Tree` instance `t` and an integer `n`. It returns the number of paths in `t` whose sum is *at least* `n`. Assume that all node values of `t` are integers.

```
def bigpath(t, n):
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> bigpath(t, 3)
    3
    >>> bigpath(t, 6)
    2
    >>> bigpath(t, 9)
    1
    """

    if t.is_leaf():

        return    one ( t.label >= n )          ____________________________)

    return   sum ([bigpath(b , n - t.label) for b in t.branches ])
```

Definition. A path through a Tree is a list of adjacent node values that starts with the root value and ends with a leaf value.

```
def one(b):
    if b:
        return 1
    else:
        return 0
```

(c) **(3 pt)** Implement `allpath` which takes a `Tree` instance `t`, a one-argument predicate `f`, a two-argument reducing function `g`, and a starting value `s`. It returns the number of paths `p` in `t` for which `f(reduce(g, p, s))` returns a true value. The `reduce` function is on the final study guide. You do not need to call it, though.

```
def allpath(t, f, g, s):
    """Return the number of paths p in t for which f(reduce(g, p, s)) is true.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> even = lambda x: x % 2 == 0
    >>> allpath(t, even, max, 0)  # Path maxes are 2, 4, and 5; 2 & 4 are even
    2
    >>> allpath(t, even, pow, 2)  # E.g., pow(pow(2, 1), 2) is even
    3
    >>> allpath(t, even, pow, 1)  # Raising 1 to any power is odd
    0
    """


    if t.is_leaf():


        return one(__________________________________________)


    return sum([____________________________________________])
```

Definition. A path through a Tree is a list of adjacent node values that starts with the root value and ends with a leaf value.

```
def one(b):
    if b:
        return 1
    else:
        return 0
```

Definition. A *path* through a `Tree` is a list of adjacent node values that starts with the root value and ends with a leaf value. For example, the paths of `Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])` are

```
[1, 2]
[1, 3, 4]
[1, 3, 5]
```

(c) **(3 pt)** Implement `allpath` which takes a `Tree` instance `t`, a one-argument predicate `f`, a two-argument reducing function `g`, and a starting value `s`. It returns the number of paths `p` in `t` for which `f(reduce(g, p, s))` returns a true value. The `reduce` function is on the final study guide. You do not need to call it, though.

```
def allpath(t, f, g, s):
    """Return the number of paths p in t for which f(reduce(g, p, s)) is true.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> even = lambda x: x % 2 == 0
    >>> allpath(t, even, max, 0)  # Path maxes are 2, 4, and 5; 2 & 4 are even
    2
    >>> allpath(t, even, pow, 2)  # E.g., pow(pow(2, 1), 2) is even
    3
    >>> allpath(t, even, pow, 1)  # Raising 1 to any power is odd
    0
    """

    if t.is_leaf():

        return one ( f ( g (s , t.label )))                  ________________)

    return sum ([ allpath (b , f , g , g (s , t.label )) for b in t.branches ])
```

```
def one(b):
    if b:
        return 1
    else:
        return 0
```

Unfortunately, multiplication in Python is broken on your computer. Implement eval_with_add, which evaluates an expression without using multiplication. You may fill the blanks with names or call expressions, but the only way you are allowed to combine two numbers is using addition.

```
>>> plus = Tree('+', [Tree(2), Tree(3)])
>>> eval_with_add(plus)
5
>>> times = Tree('*', [Tree(2), Tree(3)])
>>> eval_with_add(times)
6
>>> deep = Tree('*', [Tree(2), plus, times])
>>> eval_with_add(deep)
60
>>> eval_with_add(Tree('*'))
1
```

```
def eval_with_add(t):
    if t.entry == '+':
        return sum(________________________________________________________)
    elif t.entry == '*':
        total = ________________________________________________________
        for b in t.branches:
            total, term = 0, ____________________________________________
            for ___________ in ____________________________________________:
                total = total + term
        return total
    else:
        return t.entry
```

Unfortunately, multiplication in Python is broken on your computer. Implement eval_with_add, which evaluates an expression without using multiplication. You may fill the blanks with names or call expressions, but the only way you are allowed to combine two numbers is using addition.

```
>>> plus = Tree('+', [Tree(2), Tree(3)])
>>> eval_with_add(plus)
5
>>> times = Tree('*', [Tree(2), Tree(3)])
>>> eval_with_add(times)
6
>>> deep = Tree('*', [Tree(2), plus, times])
>>> eval_with_add(deep)
60
>>> eval_with_add(Tree('*'))
1
```

```
def eval_with_add(t):
    if t.entry == '+':
        return  sum ([ eval_with_add(b) for b in t . branches ])
    elif t.entry == '*':
        total =  1
        for b in t.branches:
            total, term = 0, total
            for _          in range ( eval_with_add (b)):
                total = total + term
        return total
    else:
        return t.entry
```

(b) **(4 pt)** Write a function `overlap` that takes two strings `word1` and `word2` and returns the maximum overlap between the end of `word1` and the beginning of `word2`. *Assume both strings have the same length.*

```
>>> overlap('ball', 'ball')
'ball'
>>> overlap('pirate', 'teepee')
'te'
>>> overlap('fish', 'bowl')
''
```

```
def overlap(word1, word2):
```

(b) **(4 pt)** Write a function `overlap` that takes two strings `word1` and `word2` and returns the maximum overlap between the end of `word1` and the beginning of `word2`. *Assume both strings have the same length.*

```
>>> overlap('ball', 'ball')
'ball'
>>> overlap('pirate', 'teepee')
'te'
>>> overlap('fish', 'bowl')
''
```

```
def overlap(word1, word2):

    if word1 == word2:
        return word1
    return overlap(word1[1:], word2[:len(word2)-1])
```

7. Implement a function nth_largest, which takes a binary search tree and a number n(greater than or equal to 1), and returns the nth largest item in the tree. For example, nth_largest(b, 1) should return the largest item in b. If n is greater than the number of items in the tree, return None.
Note: For this problem, you can assume there is a size function that returns the number of elements in a given tree.

```
def nth_largest(b, n):
    """Returns the Nth largest item in T.
    >>> b1 = Tree(2,
    ...           Tree(1),
    ...           Tree(4, Tree(3)))
    >>> nth_largest(b1, 1)
    4
    >>> nth_largest(b1, 3)
    2
    >>> nth_largest(b1, 4)
    1
    """
```

7. Implement a function nth_largest, which takes a binary search tree and a number n(greater than or equal to 1), and returns the nth largest item in the tree. For example, nth_largest(b, 1) should return the largest item in b. If n is greater than the number of items in the tree, return None.
Note: For this problem, you can assume there is a size function that returns the number of elements in a given tree.

```
def nth_largest(b, n):
    """Returns the Nth largest item in T.
    >>> b1 = Tree(2,
    ...           Tree(1),
    ...           Tree(4, Tree(3)))
    >>> nth_largest(b1, 1)
    4
    >>> nth_largest(b1, 3)
    2
    >>> nth_largest(b1, 4)
    1
    if b is None:
        return None
    right = size(b.right)
    if right == n - 1:
        return b.root
    elif right > n - 1:
        return nth_largest(b.right, n)
    elif right < n - 1:
        return nth_largest(b.left, n - 1 - right)
```