CS 61A
Summer 2018

Recursion & Tree Recursion

Discussion 3: June 28, 2018 Solutions

1 Recursion

A *recursive* function is a function that calls itself. A good example is the `factorial` function. Although we haven't finished defining `factorial`, we are still able to call it since the function body is not evaluated until the function is called. Note that when `n` is `0` or `1`, we just return `1`. This is known as the *base case*, and it prevents the function from infinitely recursing. Now we can compute `factorial(2)` in terms of `factorial(1)`, and `factorial(3)` in terms of `factorial(2)`, and `factorial(4)` – well, you get the idea.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

There are **three** common steps in a recursive definition:

1. **Figure out your base case**: The base case is usually the simplest input possible to the function. For example, `factorial(0)` is 1 by definition. You can also think of a base case as a stopping condition for the recursion. If you can't figure this out right away, move on to the recursive case and try to figure out the point at which we can't reduce the problem any further.
2. **Make a recursive call with a simpler argument**: Simplify your problem, and assume that a recursive call for this new problem will simply work. This is called the "leap of faith". For `factorial`, we reduce the problem by calling `factorial(n-1)`.
3. **Use your recursive call to solve the full problem**: Remember that we are assuming the recursive call works. With the result of the recursive call, how can you solve the original problem you were asked? For `factorial`, we just multiply $(n-1)!$ by n.

Questions

1.1 Write a function that takes two numbers `m` and `n` and returns their product. Assume `m` and `n` are positive integers. **Use recursion**, not `mul` or `*`!

Hint: `5*3 = 5 + 5*2 = 5 + 5 + 5*1`.

For the base case, what is the simplest possible input for `multiply`?

If one of the inputs is one, you simply return the other input.

For the recursive case, what does calling `multiply(m - 1, n)` do? What does calling `multiply(m, n - 1)` do? Do we prefer one over the other?

The first call will calculate a value that is `m` less than the total, while the second will calculate a value that is `n` less.

Either recursive call will work, but only `multiply(m, n - 1)` is needed.

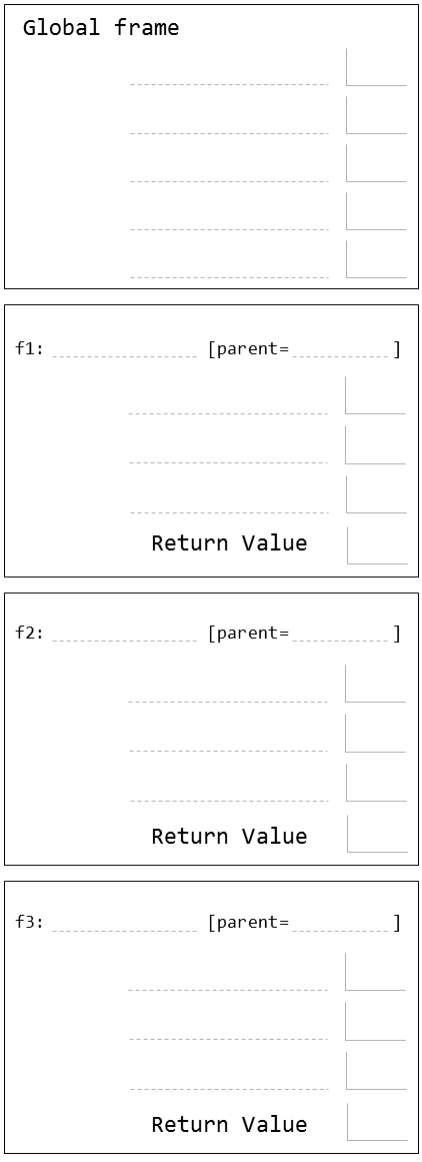
```
def multiply(m, n):
    """
    >>> multiply(5, 3)
    15
    """


    if n == 1:
        return m
    else:
        return m + multiply(m, n - 1)
```

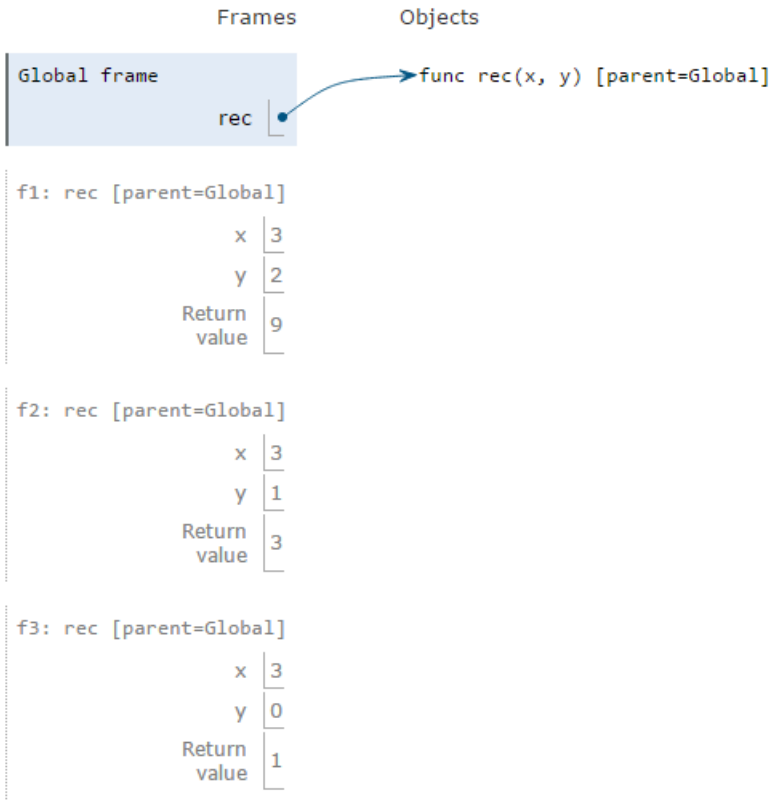
Video walkthrough

1.2 Draw an environment diagram for the following code:

```
def rec(x, y):
    if y > 0:
        return x * rec(x, y - 1)
    return 1
rec(3, 2)
```

Bonus question: what does this function do?





This function computes `x ** math.ceil(y)`.

Video Walkthrough

1.3 Write a recursive function that takes in an integer `n` and prints out a countdown from `n` to 1.

First, think about a base case for the `countdown` function. What is the simplest input the problem could be given?

When `n` equals `0`

After you've thought of a base case, think about a recursive call with a smaller argument that approches the base case. What happens if you call `countdown(n - 1)`?

A countdown starting from `n - 1` is printed.

Then, put the base case and the recursive call together, and think about where a print statement would be needed.

```
def countdown(n):
    """
    >>> countdown(3)
    3
    2
    1
    """


    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

Video walkthrough

1.4 How can we change `countdown` to count up instead without modifying a lot of the code?

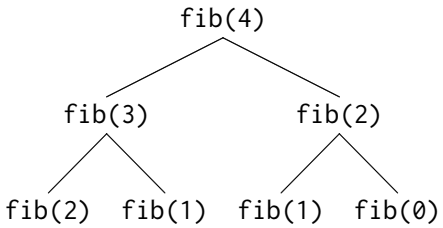
Move the `print` statement to after the recursive call.
Video walkthrough

2 Tree Recursion

Consider a function that requires more than one recursive call. A simple example is the recursive `fibonacci` function:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This type of recursion is called `tree recursion`, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:



We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

How to diagram Tree Recursion

Questions

2.1 I want to go up a flight of stairs that has `n` steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function `count_stair_ways` that solves this problem for me. Assume `n` is positive.

Before we start, what's the base case for this question? What is the simplest input?

When there is only 1 step, there is only one way to go up the stair. When there are two steps, we can go up in two ways: take a two-step, or take 2 one-steps.

What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

`count_stair_ways(n - 1)` represents the number of different ways to go up the first $n-1$ stairs. `count_stair_ways(n - 2)` represents the number of different ways to go up the first $n-2$ stairs. Our base cases will take care of the remaining 1 or 2 steps.

Use those two recursive calls to write the recursive case:

```
def count_stair_ways(n):


    if n == 1:
        return 1
    elif n == 2:
        return 2
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

Video walkthrough (Leap of Faith)
Video Walkthrough (Diagramming Trees)

2.2 Consider a special version of the `count_stairways` problem, where instead of taking 1 or 2 steps, we are able to take **up to and including** `k` steps at a time.

Write a function `count_k` that figures out the number of paths for this scenario. Assume `n` and `k` are positive.

```
def count_k(n, k):
    """
    >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
    4
    >>> count_k(4, 4)
    8
    >>> count_k(10, 3)
    274
    >>> count_k(300, 1) # Only one step at a time
    1
    """


    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        total = 0
```

```
    i = 1
    while i <= k:
        total += count_k(n - i, k)
        i += 1
    return total
```

Video Walkthrough