CS 61A
Fall 2020

Recursion

Discussion 3: September 16, 2020

1 Recursion

A *recursive* function is a function that is defined in terms of 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.

Another way to understand recursion is by separating out two things: "internal correctness" and not running forever (known as "halting").

A recursive function is internally correct if it is always does the right thing assuming that every recursive call does the right thing. For example, the factorial function reproduced to the right is internally correct, since $2! = 2$ and $n! = n * (n-1)!$ are both true statements.

```
def factorial(n): # WRONG!
    if n == 2:
        return n
    return n * factorial(n-1)
```

The bad factorial function to the right does not halt on all inputs, however, since `factorial(1)` results in a call to `factorial(0)`, and then to `factorial(-1)` and so on.

A recursive function is correct if and only if it is both internally correct and halts; but you can check each property separately. The "recursive leap of faith" is temporarily placing yourself in a mindset where you only check internal correctness.

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`?

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?

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

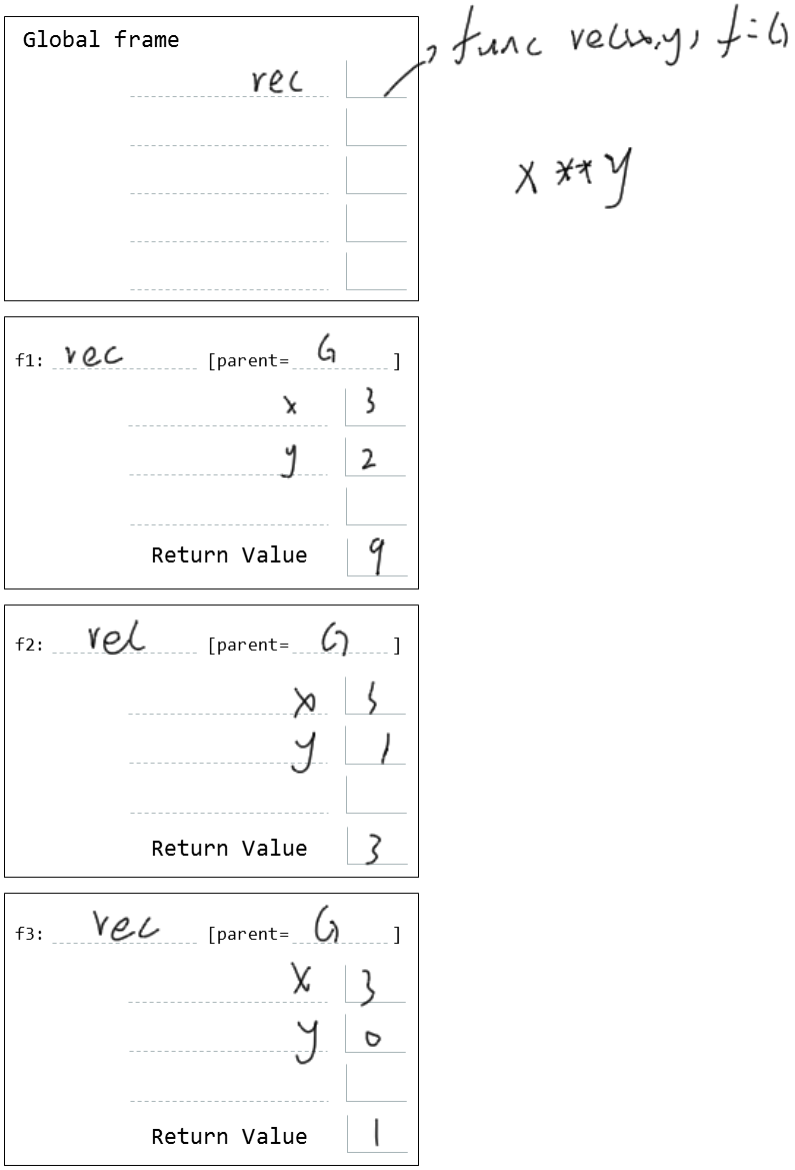
if n == 0:
    return 0
else:
    return m + multiply(m, n-1)

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?

Note: This problem is meant to help you understand what really goes on when we make the "recursive leap of faith". However, when approaching or debugging recursive functions, you should avoid visualizing them in this way.



1.3 **Tutorial:** Recall the `hailstone` function from Homework 1. First, pick a positive integer `n` as the start. If `n` is even, divide it by 2. If n is odd, multiply it by 3 and add 1. Repeat this process until n is 1. Write a recursive version of hailstone that prints out the values of the sequence and returns the number of steps.

Hint: When taking the recursive leap of faith, consider both the return value and side effect of this function.

```
def hailstone(n):
    """Print out the hailstone sequence starting at n, and return the
    number of elements in the sequence.

    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    """
```

```
print(n)
c=1
if(n==1):
    return 1
else:
    if n%2==0:
        c+=hailstone(n//2)
    else:
        c+=hailstone(n*3+1)
return c.
```

1.4 Write a procedure `merge(n1, n2)` which takes numbers with digits in decreasing order and returns a single number with all of the digits of the two, in decreasing order. Any number merged with 0 will be that number (treat 0 as having no digits). Use recursion.

Hint: If you can figure out which number has the smallest digit out of both, then we know that the resulting number will have that smallest digit, followed by the merge of the two numbers with the smallest digit removed.

```
def merge(n1, n2):
    """ Merges two numbers
    >>> merge(31, 42)
    4321
    >>> merge(21, 0)
    21
    >>> merge (21, 31)
    3211
    """
```

```
if n1 == 0 or n2 == 0:
        return n1 if n1 != 0 else n2

while n1 % 10 == 0
        n1 //= 10
while n2 % 10 == 0
        n2 //= 10

if n1 % 10 < n2 % 10:
        return n1 % 10 + 10 * merge(n1 // 10, n2)

else
        return n2 % 10 + 10 * merge(n1, n2 // 10)
```

1.5 **Tutorial: (Optional)**

Define a function make_fn_repeater which takes in a one-argument function f and an integer x. It should return another function which takes in one argument, another integer. This function returns the result of applying f to x this number of times.

Make sure to use recursion in your solution.

```
def make_func_repeater(f, x):
    """
    >>> incr_1 = make_func_repeater(lambda x: x + 1, 1)
    >>> incr_1(2) #same as f(f(x))
    3
    >>> incr_1(5)
    6
    """

    def repeat(____y____________):

        if ____y == 1____________:

            return ___f(x)__________

        else:

            return make_func_repeater(f, f(x))(y-1)

    return ___repeat______________
```

1.6 Below is the iterative version of `is_prime`, which returns `True` if positive integer `n` is a prime number and `False` otherwise:

```
def is_prime(n):
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

Implement the recursive `is_prime` function. Do not use a while loop, use recursion. As a reminder, an integer is considered prime if it has exactly two unique factors: 1 and itself.

```
def is_prime(n):
    """
    >>> is_prime(7)
    True
    >>> is_prime(10)
    False
    >>> is_prime(1)
    False
    """
def prime_helper(k):
    if k <= 1:
        return True
    elif n % k == 0:
        return False
    else:
        return prime_helper(k-1)
return prime_helper(n-1)
```