Recursion and Tree Recursion

Computer Science Mentors

September 23, 2019 to September 26, 2019

1 Recursion

There are three steps to writing a recursive function:

1. Create base case(s) * try to think of simple cases where you can immediately determine the output without needing to make recursive calls
2. Reduce your problem to a smaller subproblem and call your function recursively on the smaller subproblem * if the input doesn't get smaller (or closer to the base case), the recursion might never end!
3. Figure out how to get from the smaller subproblem back to the larger problem

Real World Analogy for Recursion

Imagine that you're in line for boba, but the line is really long, so you want to know what position you're in. You decide to ask the person in front of you how many people are in front of them. That way, you can take their response and add 1 to it. Now, the person in front of you is faced with the same problem that you were trying to solve, with one less person in front of them than you. They decide to take the same approach that you did, by asking the person in front of them. This continues until the very first person in line is asked. At this point, the person at the front knows that there are 0 people in front of them, so they can tell the person behind them that there are 0 people in front. Now, the second person can figure out that there is 1 person in front of them, and can relay that back to the person behind them, and so on, until the answer reaches you.

Looking at this example, we can see that the "recursive function" is trying to figure out how many people are in front of you. The base case is if you are in position 1 (since then there are 0 people in front of you, and you don't have anyone else). The recursive case is if you are in any position greater than 1, since then you can "recursively ask" the person in front of you (which is our smaller subproblem!) Finally, we can get from the smaller subproblem back to the big problem by adding 1 to the number from the person in front of you, since you have to include them in your count.

1. Write a function `is_sorted` that takes in an integer `n` and returns true if the digits of that number are nondecreasing from right to left.

```
def is_sorted(n):
    """
    >>> is_sorted(2)
    True
    >>> is_sorted(22222)
    True
    >>> is_sorted(9876543210)
    True
    >>> is_sorted(9087654321)
    False
    """
```

three considerations:

1. how do we "iterate" over the digits of a number? (hint: remember % and //)
2. when would we be able to confidently say that the digits are/aren't sorted, without needing more recursive calls?
   - some examples to consider: n = 35, n = 4
3. if we know our list is sorted all the way up till the element before last, how could we quickly/easily determine whether the entire list is sorted?

```
    if n < 10:
        return True        # A list with only one item is considered sorted
    last, rest = n % 10, n // 10
    if last > rest % 10:   # Comparing last digit with second-to-last digit
        return False       # No point in looking at the rest of the digits; we've already found
                           # an unsorted area in n
    else:
        return is_sorted(rest)
```

2. (Spring 2015 MT1 Q3C) Implement the `combine` function, which takes a non-negative integer `n`, a two-argument function `f`, and a number `result`. It applies `f` to the first digit of `n` and the result of combining the rest of the digits of `n` by repeatedly applying `f` (see the doctests). If `n` has no digits (because it is zero), `combine` returns `result`.

★ note: we don't get to choose what to return in the base case because it's already been written for us. So, we must make sure that as we gradually build up our output, we keep storing it into result so that it will eventually get returned from the base case.

```
def combine(n, f, result):
    """
    Combine the digits in non-negative integer n using f.

    >>> combine(3, mul, 2) # mul(3, 2)
    6
    >>> combine(43, mul, 2) # mul(4, mul(3, 2))
    24
    >>> combine(6502, add, 3) # add(6, add(5, add(0, add(2, 3)
       )))
    16
    >>> combine(239, pow, 0) # pow(2, pow(3, pow(9, 0))))
    8
    """
    if n == 0:
        return result
    else:
        return combine(n // 10 , f ,
                       f(n % 10, result) )
```

2 Tree Recursion

Tree Recursion vs Recursion

In most recursive problems we've seen so far, the solution function contains only one call to itself. However, some problems will require multiple recursive calls – we colloquially call this type of recursion tree recursion, since the propagation of function frames reminds us of the branches of a tree. "Tree recursive" or not, these problems are still solved the same way as those requiring a single function call: a base case, the recursive leap of faith on a subproblem, and solving the original problem with the solution to our subproblems. The difference? We simply may need to use multiple subproblems to solve our original problem.

Tree recursion will often be needed when solving counting (how many ways are there of doing something?) problems and optimization (what is the maximum or minimum number of ways of doing something?) problems, but remember there are all sorts of problems that may need multiple recursive calls! Always come back to the recursive leap of faith.

Two rules that are often useful in solving counting problems:

"and" → 1. If there are *a ways* of doing something and *b ways* of doing another thing, there are *ab ways* of doing **both** at the same time.

"or" → 2. If there are *a ways* of doing one thing and *b ways* of doing another, but we can't do both things at the same time, there are $a + b$ ways of doing either the first thing **or** the second thing.

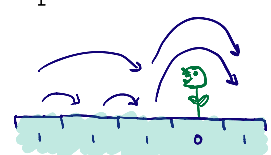
e.g. in count_partitions, we can either use a partition of size m, or not use a partition of size m. We cannot do both of these at the same time. So, the total # of ways to partition is [# of ways if we use m] + [# of ways if we don't use m]

CS61A Fall 2019: Catherine Cang and Danelle Nachum, with
Addison Chan, Aini Xu, Albert Xu, Bridget Cheng, Evan Corriere, Imaani Choudhuri, Jack Coyle, Jade Singh, Jason Chang, Jemmy Zhou, Jessica Yeh, Lucas Pan, Mary Xu, Olivia Shiah, Patricia Ouyang, Richard Roggenkemper, Risham Sidhu, Samy Cherfaoui

1. Mario needs to jump over a series of Piranha plants, represented as a string of 0's and 1's. Mario only moves forward and can either *step* (move forward one space) or *jump* (move forward two spaces) from each position. How many different ways can Mario traverse a level without stepping or jumping into a Piranha plant? Assume that every level begins with a 1 (where Mario starts) and ends with a 1 (where Mario must end up).

   Hint: Does it matter whether Mario goes from left to right or right to left? Which one is easier to check? If I find some path to cross the level, what happens if I follow the reverse of that path, starting from the end?

```
def mario_number(level):
    """
    Return the number of ways that Mario can traverse the
    level, where Mario can either hop by one digit or two
    digits each turn. A level is defined as being an integer
    with digits where a 1 is something Mario can step on and
    0 is something Mario cannot step on.
    >>> mario_number(10101)
    1
    >>> mario_number(11101)
    2
    >>> mario_number(100101)
    0
    """
```



2 valid paths

```
    if n % 10 == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return mario_number(n // 10) + mario_number(n // 100)
```

\# if the "current" position (the last digit of n) is 0, then there are no ways to get to the ending from here!

\# if n == 1, we have reached the end!

mario_number(n // 10): number of paths if we hop 1 square

mario_number(n // 100): number of paths if we hop 2 squares

CS61A Fall 2019: Catherine Cang and Danelle Nachum, with
Addison Chan, Aini Xu, Albert Xu, Bridget Cheng, Evan Corriere, Imaani Choudhuri, Jack Coyle, Jade Singh, Jason Chang, Jemmy Zhou, Jessica Yeh, Lucas Pan, Mary Xu, Olivia Shiah, Patricia Ouyang, Richard Roggenkemper, Risham Sidhu, Samy Cherfaoui

2. James wants to print this week's discussion handouts for all the students in CS 61A. However, both printers are broken! The first printer only prints multiples of `n` pages, and the second printer only prints multiples of `m` pages. Help James figure out whether or not it's possible to print exactly `total` number of handouts!

```
def has_sum(total, n, m):
    """
    >>> has_sum(1, 3, 5)
    False
    >>> has_sum(5, 3, 5) # 0 * 3 + 1 * 5 = 5
    True
    >>> has_sum(11, 3, 5) # 2 * 3 + 1 * 5 = 11
    True
    """
    if total < 0: # can't print negative pages
        return False
    elif total == 0:
        return True
    return has-sum(total-n, n, m) or has-sum(total-m, n, m)
```

Note: we don't need both of our recursive calls to return True in order for us to return True. Remember, we just want to know if there's *any* way for us to print total pages.

3. The next day, the printers break down even more! Each time they are used, the first printer prints a random `x` copies $50 \leq x \leq 60$, and the second printer prints a random `y` copies $130 \leq y \leq 140$. James also relaxes his expectations: he's satisfied as long as there's at least `lower` copies so there are enough for everyone, but no more than `upper` copies to prevent waste.

```
def sum_range(lower, upper):
    """
    >>> sum_range(45, 60) # Printer 1 prints within this range
    True
    >>> sum_range(40, 55) # Printer 1 can print a number 50-60
    False
    >>> sum_range(170, 201) # Printer 1 + 2 will print between
        180 and 200 copies total
    True
    """
    def copies(pmin, pmax):
        if pmin >= lower and pmax <= upper:

            return True

        elif pmin > upper: # if pmin >upper, we will definitely print too many.

            return False

        return sum_range(pmin + 50, pmax+60) or sum_range(pmin + 130, pmax + 140)
               # using first printer              # using second printer
    return copies(0, 0)
```

The starter code has already specified how the helper fn will get called. Think about how you can work with this. Why did they start pmin and pmax as 0?

For this problem, we just care that the # of pages printed will be within some range, no matter how many copies actually get printed each time we use a printer.