Recursion and Tree Recursion

CS 61A

July 2, 2021

1 Recursion

There are three steps to writing a recursive function:

1. Create base case(s)
2. Reduce your problem to a smaller subproblem and call your function recursively on the smaller subproblem
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 see that we have broken down the problem of "how many people are there in front of me?" to 1 + "how many people are there in front of the person in front of me"? This problem will terminate with the person at the front of the line (with 0 people in front of them). Putting this into more formal terms, we are breaking down the problem into a **recurrence relationship**, and the termination case is called the **base case**.

1. What is wrong with the following function? How can we fix it?

```
def factorial(n):
    return n * factorial(n)
```

2. 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
    """
```

3. Fill in `collapse`, which takes in a non-negative integer `n` and returns the number resulting from removing all digits that are equal to an adjacent digit, i.e. the number has no adjacent digits that are the same.

```
def collapse(n):
    """
    >>> collapse(12234441)
    12341
    >>> collapse(11200000013333)
    12013
    """
    rest, last = n // 10, n % 10

    if _____________________________________:

        _____________________________________

    elif _____________________________________:

        _____________________________________
    else:

        _____________________________________
```

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

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?

```
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
    """
    if _____________________________:

        _____________________________

    elif _____________________________:

        _____________________________

    else:

        ___________________________________________________________
```

2. In an alternate universe, 61A software is not that good (inconceivable!). Laryn is in charge of assigning students to discussion sections, but sections.cs61a.org only knows how to list sections with either `m` or `n` number of students (the two most popular sizes). Given a `total` number of students, can Laryn create sections and not have any leftover students? Return `true` if they can, `false` otherwise.

```
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
    >>> has_sum(61, 11, 15) # 61 is a prime number and can
       't be divided!
    False
    """
    if
       _____________________________________________________
       :

        return _____________________________________

    elif
       _____________________________________________________:

        return _____________________________________

    return
       _____________________________________________________
```

3. Realizing the need for improvement, Laryn has recruited you to help them make 61A sections more flexible! Laryn would like discussion sections to have $20 \leq \texttt{x} \leq 30$ students each, and tutoring sections to have $6 \leq \texttt{y} \leq 8$ students. Additionally, it's okay to have up to `upper` total slots, as long as we have at least `lower` amount to accomodate all students. Is it possible to assign each student a section? (Note: In this alternate universe, students can choose either a tutoring section or a discussion section, but not both.)

```
def sum_range(lower, upper):
    """
    >>> sum_range(25, 30) # Everyone can go into one
       discussion section
    True
    >>> sum_range(9, 10) # If we make a tutoring section,
       there will be 1-4 extra students
    False
    >>> sum_range(56, 64) # 2 discussion sections, 2
       discussions 1 tutoring section, etc. all work
    True
    """
    def discussions(pmin, pmax):
        if
           ____________________________________________________
           :

            return ____________________________________________

        elif
           ____________________________________________________:

            return ____________________________________________

        return
           ____________________________________________________

    return discussions(0, 0)
```