Recursion and Tree Recursion Solutions

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)
```

There is no base case and the recursive call is made on the same `n`.

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

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
    """
    right_digit = n % 10
    rest =  n // 10
    if rest == 0:
        return True
    elif right_digit > rest % 10:
        return False
    else:
        return is_sorted(rest)
```

First, let's look into the base case. At what point will you know a number is sorted/not sorted immediately?

1. If `n` only has 1 digit or is 0, we know it is definitely sorted with itself. This corresponds to the first if condition, `rest == 0`.
2. If the 2nd-to-last and last digits are not in sorted order, we know the number is not sorted. To do this, we need at least 2 digits in `n` to compare, which is why we check this in **`elif`** after ensuring `n` is not 0.

Next, let's go into the recursive step. We build off of the base cases: if the base cases fail, then we can now work off of the assumption that `n` has at least 2 digits and the last 2 digits of n are in sorted order. Next, notice that after chopping off the last digit, to check that the rest of `n` is sorted, we can use our function `is_sorted` on the number `rest`. So finally, we make the recursive call with `rest` as the argument.

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:

        ________________________________
```

```
def collapse(n):
    rest, last = n // 10, n % 10
    if rest == 0:
        return last
    elif last == rest % 10:
        return collapse(rest)
    else:
        return collapse(rest) * 10 + last
```

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:

        ______________________________________________________
```

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

You can think about this tree recursion problem as testing out all of the possible ways Mario can traverse the level, and adding 1 every time you find a possible traversal.

Here it doesn't matter whether Mario goes left to right or right to left; either way we'll end up with the same number of ways to traverse the `level`. In that case, we can simply choose for Mario to start from the right, then we can process the level like we process other numbers in digit-parsing related questions by using floor div (`//`) and modulo (`%`)

At every time step, Mario can either take one or two steps, and these would take form in two corresponding recursive calls. A single floor division (`//`) of `level` by 10 means taking one step at this point in the `level` (if we took a step, then the entire `level` would be left except for the last number), while two floor divisions by 10 (or equivalently one floor division by 100) corresponds to a jump at this point in the `level` (if we took a jump, then the entire `level` would be left except for the last two numbers).

To think of the base cases, you can consider the two ways that Mario ends his journey. The first, corresponding to `level == 1`, means that Mario has successfully reached the end of the level. You can `return 1` here, because this means you've found one additional path to the end. The second, corresponding to `level % 10 == 0`, means that Mario has landed on a Piranha plant. This returns 0 because it's a failed traversal of the `level`, so you don't want to add anything to your result.

In tree recursion, you need to find a way to combine separate recursive calls. In this case, because `mario_number` returns an integer and the base cases are integers and you're trying to count the total number of ways of traversal, it makes sense to add your recursive calls.

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
       ________________________________________________
```

```
def has_sum(total, n, m):
    if total == 0:
        return True
    elif total < 0: # you could also put total < min(m, n)
        return False
    return has_sum(total - n, n, m) or has_sum(total - m,
       n, m)
```

An alternate solution you could write that may be slightly faster in certain cases:

```
def has_sum(total, n, m):
    if total == 0 or total % n == 0 or total % m == 0:
        return True
    elif total < 0: # you could also put total < min(m, n)
        return False
    return has_sum(total - n, n, m) or has_sum(total - m,
       n, m)
```

(Solution continues on the next page)

When thinking about the recursive calls, we need to think about how each step of the problem works. Tree recursion allows us to explore the two options we have: either create a new `m`-person discussion at this step or create a new `n`-person discussion at this step and can combine the results after exploring both options. Inside the recursive call for `has_sum(total - n, n, m)`, which represents accommodating `n` students, we again consider adding either `n` or `m` students to the next section.

Once we have these recursive calls we need to think about how to put them together. We know the return should be a boolean so we want to use either **and** or **or** to combine the values for a final result. Given that we only need one of the calls to work, we can use **or** to reach our final answer.

In the base cases we also need to make sure we return the correct data type. Given that the final return should be a boolean we want to return booleans in the base cases.

Another alternate base case would be: `total == 0 or total % n == 0 or total % m == 0`. This solution would also work! You would just be stopping the recursion early, since the total can be a multiple of `n` or `m` in order to trigger the base case - it doesn't have to be 0 anymore. Just be sure to still include the `total == 0` check, just in case someone inputs 0 as the total into the function.

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 $20 \leq$ `x` ≤ 30 students each, and tutoring sections to have $6 \leq$ `y` ≤ 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)
```

```
def sum_range(lower, upper):
    def discussions(pmin, pmax):
        if lower <= pmin and pmax <= upper:
            return True
        elif upper < pmin:
            return False
        return discussions(pmin + 6, pmax + 8) or
           discussions(pmin + 20, pmax + 30)
    return discussions(0, 0)
```

(Solution continues on the next page)

This question is similar to the last one but now we have to deal with two parameters `pmin` and `pmax`. Let's start with our recursive calls. Each call represents using either a discussion or tutoring section and returns a boolean value that indicates whether or not we can include the number of students in our desired range (set by lower and upper). If we use a discussion section, then our current minimum is + 20 and the maximum is whatever our current maximum is + 30. We need to take the two recursive calls, which each represents a choice we have (either add a tutoring or discussion section) and combine them in some way. If one of these choices led to `True` then we should return `True` so we use an **`or`** to combine the recursive calls.

Our first base case represents successfully accomodating the number of students within our range in which case we can stop the recursion because we know it is possible to print that number of pages within the range. Our second base case handles the case where we know for sure that we cannot fit the amount of students within our range so we return `False` and stop doing the recursion.