

RECURSION AND TREE RECURSION

COMPUTER SCIENCE MENTORS

September 21, 2020 to September 24, 2020

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. 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  
    """
```

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`.

```
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(_____, _____,
                       _____)
```

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.

Two rules that are often useful in solving counting problems:

1. If there are x ways of doing something and y ways of doing another thing, there are xy ways of doing **both** at the same time.
2. If there are x ways of doing one thing and y ways of doing another, but we can't do both things at the same time, there are $x + y$ ways of doing either the first thing **or** the second thing.

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. 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 _____:  
        return _____  
    elif _____:  
        return _____  
    return _____
```

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 _____:

            return _____

        elif _____:

            return _____

        return _____

    return copies(0, 0)
```