MORE RECURSION AND TREE RECURSION

COMPUTER SCIENCE MENTORS CS 88

March 15th to 19th

1 Recursion

1. Write a function that takes in an integer n, and returns True if the digits of the number are strictly increasing from left to right, and False otherwise.

```
def is_increasing(n):
    """
    >>> is_increasing(2222)
    False
    >>> is_increasing(56789)
    True
    >>> is_increasing(56788)
    False
    """
```

Solution:

```
    right_digit = n % 10
    rest = n // 10
    if rest == 0: # We've gone through all the digits!
        return True
    elif right_digit <= rest % 10:
        return False
    return is_increasing(rest)
```

2. Implement a recursive `fizzbuzz`.

```
def fizzbuzz(n):
    """Prints the numbers from 1 to n. If the number is
       divisible by 3, it instead prints 'fizz'. If the number
        is divisible by 5, it instead prints 'buzz'. If the
       number is divisible by both, it prints 'fizzbuzz'.
    >>> fizzbuzz(15)
    1
    2
    fizz
    4
    buzz
    fizz
    7
    8
    fizz
    buzz
    11
    fizz
    13
    14
    fizzbuzz
    """
```

Solution:

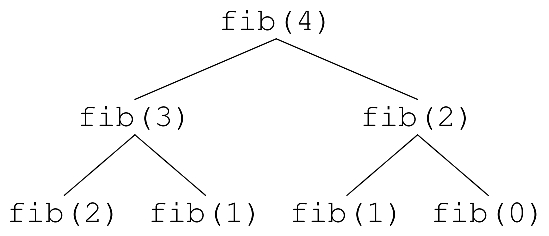
```
    if n == 1:
        print(n)
    else:
        fizzbuzz(n - 1)
        if n % 3 == 0 and n % 5 == 0:
            print('fizzbuzz')
        elif n % 3 == 0:
            print('fizz')
        elif n % 5 == 0:
            print('buzz')
        else:
            print(n)
```

2 Tree Recursion

Consider a function that requires more than one recursive call. A simple example is the recursive `fibonacci` function:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This type of recursion is called `tree recursion`, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:



We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. It is sometimes the case that a tree recursive problem also involves iteration: for example, you might use a while loop to add together multiple recursive calls.
As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

Solution: How to diagram Tree Recursion

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:

        ___________________________________________________
```

Solution:

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

2. Write a procedure `merge(s1, s2)` which takes two sorted (smallest value first) lists and returns a single list with all of the elements of the two lists, in ascending order. Use recursion.

   Hint: If you can figure out which list has the smallest element out of both, then we know that the resulting merged list will have that smallest element, followed by the merge of the two lists with the smallest item removed. Don't forget to handle the case where one list is empty!

```
def merge(s1, s2):
    """ Merges two sorted lists
    >>> merge([1, 3], [2, 4])
    [1, 2, 3, 4]
    >>> merge([1, 2], [])
    [1, 2]
    """
```

Solution:

```
    if len(s1) == 0:
        return s2
    elif len(s2) == 0:
        return s1
    elif s1[0] < s2[0]:
        return [s1[0]] + merge(s1[1:], s2)
    else:
        return [s2[0]] + merge(s1, s2[1:])
```

3. We will now write one of the faster sorting algorithms commonly used, known as *merge sort*. Merge sort works like this:

   1. If there is only one (or zero) item(s) in the sequence, it is already sorted!
   2. If there are more than one item, then we can split the sequence in half, sort each half recursively, then merge the results, using the `merge` procedure from earlier in the notes. The result will be a sorted sequence.

   Using the algorithm described, write a function `mergesort(seq)` that takes an unsorted sequence and sorts it. You can use `merge(s1, s2)` as a helper function.

```
def mergesort(seq):
```

Solution:

```
    if len(seq) <= 1:
        return seq
    else:
        midpt = len(seq)//2
        return merge(mergesort(seq[:midpt]), mergesort(seq[
           midpt:]))
```

3 Challenge Problems

1. Tony wants to print this week's discussion handouts for all the students in CS 88. 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 Tony 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 ________________________________________________
```

Solution:

```
def has_sum(total, n, m):
    if total == 0: # (total == n or total == m) works too
       except when total equals 0
        return True
    elif total < 0: # (total < min(n1, n2)) works given
       alternate base case
        return False
    return has_sum(total - n, n, m) or has_sum(total - m, n
       , m)
```

2. 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 $130 \leq y \leq 140$. Tony 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 56-60
    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)
```

Solution:

```
def sum_range(lower, upper):
    def copies(pmin, pmax):
        if lower <= pmin and pmax <= upper:
            return True
        elif upper < pmin:
            return False
        return copies(pmin + 50, pmax + 60) or copies(pmin
           + 130, pmax + 140)
    return copies(0, 0)
```