RECURSION, TREE RECURSION

COMPUTER SCIENCE MENTORS 61A

September 23 – September 27, 2024

1 Recursion

1. Implement a recursive version of `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'. You
       must do this recursively!

    >>> fizzbuzz(15)
    1
    2
    fizz
    4
    buzz
    fizz
    7
    8
    fizz
    buzz
    11
    fizz
    13
    14
    fizzbuzz
    """
```


Provided by CSM 61A contributors under CC BY 4.0/3-clause BSD.

2. Complete the definition for `sum_prime_digits`, which returns the sum of all the prime digits of `n`. Recall that 1 is not prime. Assume you have access to a function `is_prime`; `is_prime(n)` returns `True` if `n` is prime, and `False` otherwise.

```
def sum_prime_digits(n):
    """
    >>> sum_prime_digits(12345)
    10 # 2 + 3 + 5
    >>> sum_prime_digits(4681029)
    2 # 2 is the only prime number
    """
    if _____________________________________________:

        return _____________________________________

    if _____________________________________________:

        return _____________________________________

    return _____________________________________
```

3. Fill in `near`, which takes in a non-negative integer `n` and returns the largest, non-consecutively repeating, near increasing sequence of digits within n as an integer. The arguments `smallest` and `d` are part of the implementation; you must determine their purpose. You may **not** use any values except integers and booleans (`True` and `False`) in your solution (no lists, strings, etc.).

   A sequence is *near increasing* if each element but the last two is smaller than all elements following its subsequent element. That is, element i must be smaller than elements $i + 2$, $i + 3$, $i + 4$, etc. A *non-consecutively repeating* number is one that do not have two of the same digits next to each other. [Adapted from CS61A Fa18 Final Q3(c)]

```
def near(n, smallest=10, d=10):
    """
    >>> near(123)
    123
    >>> near(153)
    153
    >>> near(1523)
    153
    >>> near(15123)
    153
    >>> near(985357)
    537
    >>> near(11111111)
    1
    >>> near(14735476)
    143576
    >>> near(14735476)
    1234567
    """
    if n == 0:

        return ____________________________________________

    no = near(n//10, smallest, d)

    if (smallest > ____________________) and (____________________):

        yes = ____________________________________________

        return ____________________________________________(yes, no)

    return ____________________________________________
```

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.

Let's work through an example of tree recursion in action! The **counting partitions** problem is a common utilization of tree recursion. Say we want to define a function that will count all the ways we can fill up a space with a certain amount of blocks. For the purposes of working out this problem, we'll usea space of size 4. If we use brute force to solve this problem in terms of adding blocks of sizes 1, 2, 3, and 4, here's what we come up with:

- 4 = 4 // filling up the space with a block that perfectly fits it
- 4 = 3 + 1 // filling up the space with a block of size 3 and a block of size 1
- 4 = 2 + 2
- 4 = 2 + 1 + 1
- 4 = 1 + 1 + 1 + 1

This doesn't seem too bad, but with bigger spaces, this is exponentially harder to brute force! With this in mind, however, we can recognize a pattern with the number of possibilities present with different "partition" sizes.

Recall that within recursive questions, arguments are used as ways to keep track of certain possibilities. We want to specify our arguments within this problem to be what's changing on iteration. Therefore, within our recursive arguments, we want to keep track of the maximum block size on each iteration, and how much more space we have left after adding that partition.

Now we know what arguments we want, we can construct the body of our code!

```
def count_partitions(space_left, block_size):
```

We know that a recursive function will have base cases and recursive cases. While everybody's way of working through these is different, for the sake of explanation, we'll start with our base cases. When working with recursion questions, we generally want to start with the biggest problem and work our way down to the most atomic ways of solving our problem, similar to the way in which our doctests are structured.

We will consider a way of counting partitions valid if the partitions we add perfectly fit the space provided. When a space is perfectly fit, the `space_left` will be zero! Therefore, we will count a case when it's `space_left` is equal to 0.

A doctest demonstrating this would be the most atomic iteration of `count_partitions`, `count_partitions(0, 0)`. There is only one way to count to zero with the maximum number to add being zero!

Now we consider invalid ways of counting partitions – a way of counting would be invalid if we *overfill* or *underfill* the space provided. Therefore, if our `block_size` is greater than the space left, we would overfill the space! Similarly if the `block_size` we're adding is 0, that means we've run out of partitions to add, resulting in an *underfilled* space.

With this in mind, here is our current working of `count_partitions` with the base cases in hand.

```
def count_partitions(space_left, block_size):
    if (space_left == 0):
        return 1 // valid! we've fit the space perfectly
    elif (space_left < block_size):
        return 0 // not valid! overfilled
    elif (block_size == 0):
        return 0 // not valid! space is underfilled
    else:
        // recursive cases :D
```

As with any recursive case, we want to consider how we can break our problem into possibilities that allow us to get closer to our base cases.

On each frame of `count_partitions`, we want to have both the `space_left` and `block_size` go closer to zero such that they can reach the base case.

Consider the first case for the 4 example we used earlier. We want to see how many different potential ways we can fill up a space of 4 blocks with maximum block size of 4. Upon our first call to this function we would want to include the way of filling up the space with the partition of size 4 in our recursive call. With this in mind, to include this way of counting, we would have to subtract the `block_size` of size 4 from the `space_left` of size 4 such that we can get to our base case! This would result in a recursive call of `count_partitions(space_left - block_size, block_size)` (We keep m the same as we would want to see if we can add more blocks of that same size). However, having just that case would end our counting prematurely. Thus we need to include a separate branch where we *don't* use that block and instead use smaller block sizes.

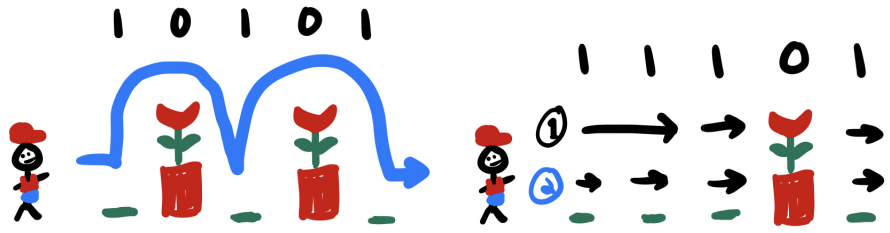
Similar to the doctest, we want to see every possibility using blocks of every possible size. The next block size down from 4 would be 3, thus resulting in a recursive call of `count_partitions(space_left, block_size - 1)`. From here, there are no other recursive cases we can call upon, thus completing our code. We add the recursive cases together as we want to sum all possibilities of counting such partitions, resulting in the following:

```
def count_partitions(space_left, block_size):
    if (space_left == 0):
        return 1
    elif (space_left < block_size):
        return 0
    elif (block_size == 0):
        return 0
    else:
        return count_partitions(space_left - block_size, block_size) +
            count_partitions(space_left, block_size - 1)
```

1. 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 _____
```

2. Mario needs to get from one end of a level to the other, but there are deadly Piranha plants in his way! Mario only moves forward and can either *step* (move forward one space) or *jump* (move forward two spaces) from each position. A level is represented as a series of ones and zeros, with zeros denoting the location of Piranha plants. Mario can step on ones but not on zeros. 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):
    """
    >>> mario_number(10101)
    1
    >>> mario_number(11101)
    2
    >>> mario_number(100101)
    0
    """
    if ____________________________:

        ____________________________

    elif ____________________________:

        ____________________________

    else:

        ____________________________________________________________
```

3. **Fast Modular Exponentiation:** In many computing applications, we need to quickly compute $n^x \mod z$ where $n > 0$, and x and z are arbitary whole numbers. Computing $n^x \mod z$ for large numbers can get extremely slow if we repeatedly multiply n for x times. We can implement the following recursive algorithm to help us speed up the exponentiation operation.

$$x^n \mod z = \begin{cases} x * (x^2)^{(n-1)/2} \,\%\, z & \text{if } n \text{ is odd} \\ (x^2)^{(n/2)} \,\%\, z & \text{if } n \text{ is even} \end{cases}$$

This is an example of a "divide & conquer" algorithm and follows the same train of thought as tree-recursion problems (you are dividing some complex problem into smaller parts and performing both options).

```
def modular_exponentiation(base, exponent, modulus):
    """
    >>> modular_exponentiation(2, 2, 2)
    0
    >>> modular_exponentiation(4, 2, 3)
    1
    """
    if ________________________:

        return ________________________________________

    if ________________________:

        half_power = ________________________________________

        return ________________________________________ % modulus

    else:

        return ________________________________________________ % modulus
```

Note: The algorithm you just implemented is a key part of modern day cryptography techniques such as RSA and Diffie-Hellman key exchange. In some cases, the exact operations you just implemented is used in modern day, state of the art, programs (if you are curious, Google "Right-to-left binary method"). You will learn more about RSA in CS70. If you want to learn more about computer security, consider taking CS161 after CS61C.