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
    """
```

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)]

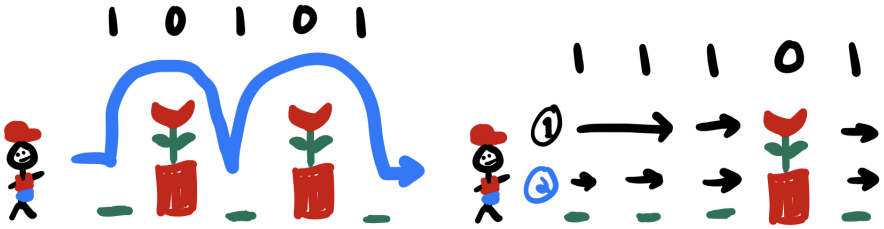
```
def near(n, smallest=10, d=10):
    """
    >>> near(123)
    123
    >>> near(153)
    153
    >>> near(1523)
    153
    >>> near(15123)
    153
    >>> near(985357)
    537
    >>> near(11111111)
    1
    >>> near(14735476)
    143576
    >>> near(14735476)
    1234567
    """
    if n == 0:

        return ____________________________________________

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

    if (smallest > ____________________) and (____________________):

        yes = ____________________________________________

        return ____________________________________________(yes, no)

    return ____________________________________________
```

2 Tree Recursion

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.