# Recursion, Tree Recursion Solutions

## Computer Science Mentors 61A

September 23 – September 27, 2024

Created by Jim Fang, Bryan Adolfo Ventura Benitez. Worksheet feedback: links.jimfang.dev/feedback

# 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
    """


    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. 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 _____________________________________


    if n == 0:
        return 0
    if is_prime(n % 10):
        return n % 10 + sum_prime_digits(n // 10)
    return sum_prime_digits(n // 10)
```

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 _____________________________________
```

```
def near(n, smallest=10, d=10):
    if n == 0:
        return 0

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

    if (smallest > n % 10) and (n % 10 != d):
        yes = 10 * near(n//10, min(smallest, d), n%10) + n%10
        # OR yes = 10 * near(n//10, d, min(d, n%10)) + n%10
        return max(yes, no)

    return no
```

`smallest` = smallest digit

`d` = previous digit

For a video walkthrough of the unadapted exam problem, see https://youtu.be/NnE6qFZsoGo. This should give you a good idea how to approach the adapted version.

## 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 _____
```

```
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)**

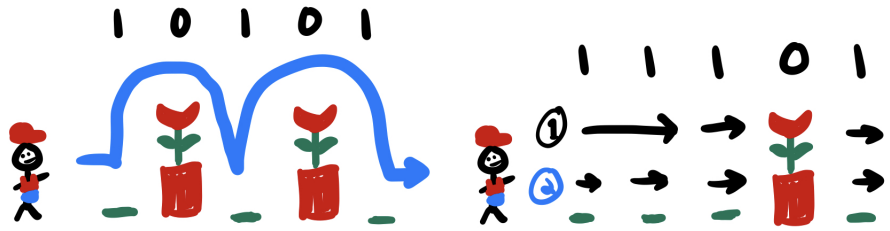
When thinking about the recursive calls, we need to think about how each step of the problem works. Tree recursion allows us to explore the two options we have while printing: either print `m` papers at this step or print `n` papers at this step and can combine the results after exploring both options. Inside the recursive call for `has_sum(total - n, n, m)`, which represents printing `n` papers, we again consider printing either `n` or `m` papers.

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.

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:

        ____________________________________________________________
```

```
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, and then we can process the level like we process other numbers in digit-parsing related questions by using floor division (`//`) and modulo (`%`)

At each point in time, Mario can either step or jump. We use a single floor division (`//`) of `level` by 10 to represent taking one step (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.

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.

```
def modular_exponentiation(base, exponent, modulus):
# Base case: exponent is 0
if exponent == 0:
    return 1

# Recursive case
if exponent % 2 == 0:
    half_power = modular_exponentiation(base, exponent // 2,modulus)
    return (half_power * half_power) % modulus
else:
    return (base * modular_exponentiation(base, exponent - 1,
       modulus)) % modulus
```