Question 0

a) Explain Recursion. If it doesn't take too long, write your explanation in haiku.

Function calls itself
to reduce a large problem
down to a base case

You find a problem
Solve or break down the problem
Go back to line one?

Source: https://piazza.com/class/hktwdlzfjnw680?cid=1127

b) What are three things you find in every recursive function?

1. One or more Base Cases

2. Way(s) to make the problem into a smaller problem of the same type (so that it can be solved recursively).

3. One or more Recursive Cases that solve the smaller problem and then uses the solution the smaller problem to solve the original (large) problem

c) When you write a Recursive function, you seem to call it before it has been fully defined. Why doesn't this break the Python interpreter? Explain in haiku if possible.
When you define a function, Python does not evaluate the body of the function.

Python does not care
about a function’s body
until it is called

d) What does the following code do? Describe it in English. Hint: diagram the function in terms of the base cases, the recursive cases, and reducing the problem.

```
def mystery(lst, toggle=False):
 if lst == []:
   return []
 elif toggle:
   return [lst[0]] + mystery(lst[1:], False)
 else:
   return mystery(lst[1:], True)
```

This is the function that takes in a list and returns a list of every other element. If toggle is initially True, it'll add the 0th element and skip the 1st, add the 2nd, etc. If toggle is initially False, it'll skip the 0th element and add the 1st, skip the 2nd, and add the 3rd, etc.

e) What's the base case? What are the recursive cases?
The base case is the first if clause, and the recursive cases are the other clauses.

f) What would mystery([1, 2, 3, 4])) output?
[2, 4]
g) How about mystery([2, 3, 4, 5], True)?
[2, 4]
h) What does toggle do?
See the answer for part e
i) Could this function work with strings? E.g. mystery("I am a string").
No, this function cannot work with strings because of the base case. And also the code would try to add a List to a String.

Question 1
Hint: Domain is the type of data function takes in as argument. The Range is the type of data that a function returns.
E.g. the *domain* of the function `square` is numbers. The *range* is numbers.

removed) What does the mathematical factorial function do?
n! = n·(n − 1)·(n − 2)· … ·(2)·(1)

Here is a Python function that computes factorial. What's it's domain and range? Identify the three things from **Q0b**:

```
def factorial(n): Domain is integers, Range is integers
  if n <= 1: # base case
    return 1
  else: # recursive case
    return n*factorial(n-1) # make a smaller problem
```

Write out the recursive calls made when we do `factorial(4)`.

```
factorial(4) => 4 * factorial(3)
factorial(3) => 3 * factorial(2)
factorial(2) => 2 * factorial(1)
factorial(1) => 1
factorial(2) => 2 * factorial(1) => 2 * 1 => 2
factorial(3) => 3 * factorial(2) => 3 * 2 => 6
factorial(4) => 4 * factorial(3) => 4 * 6 => 24
```

For example, here's the same idea for `factorial(2)`:

```
factorial(2) # n is not <= 1 so do the else
    ||
2 * factorial(1) # n is <= 1 so return 1
        ||
         1
2 * 1 # replace factorial(1) expression with its value
2 # replace factorial(2) expression with its value
```

a) What does the mathematical Fibonacci function do?

fib(n) = fib(n − 1) + fib(n − 2) ∀
 n > 1
fib(0) = 0
fib(1) = 1

Here is a Python function that computes the nth Fibonnaci Number. What's it's domain and range? Identify the three things from **Q0b**:

```
def fib(n): Domain is integers, Range is integers
  if n == 0: # base case
    return 0
  elif n == 1: another base case
    return 1
  else: # ONE recursive CASE with TWO recursive CALLS
    return fib(n-1) + fib(n-2) # reducing the problem
```

Write out the recursive calls made when we do `fib(4)` (this will look like an upsidedown tree).

```
               fib(4)
               /    \
           fib(3)   fib(2)
          /    |      |    \
     fib(2)  fib(1) fib(1)  fib(0)
     /    |
fib(1)   fib(0)
```

b) What does `cascade2` do? (maybe look at lecture?)
Takes in a number n and prints out n, n excluding the ones digit, n excluding the tens digit, n excluding the hundreds digit, etc, then back up to the full number

```
cascade2(2094)
2094
209
20
2
20
209
2094
```

Here is one of the python functions that computes cascade. Domain and range? Identify the three things from **Q0b**:

```
def cascade2(n): Domain is integers, Range is None
    """Print a cascade of prefixes of n."""
    print(n) # Base case is when n < 10
    if n >= 10: # recursive case
```

```
        cascade2(n//10) # reducing the problem
        print(n)
```

Question 2

a) What's wrong with the following code? Identify all the mistakes and fix them.

```
def multiply(x, y):
  """ Multiplies two numbers together without using the * operator or mul

  >>> multiply(3, 4)
  12
  """
  if x == 0:
    return 1 # Should be 0
  else:
    return y + multiply(x, y) # Should be x - 1
```

b) What does `sum_eo_list` do?

```
>>> sum_eo_list([1, 2, 3, 4, 5, 6, 7])
12
>>> sum_eo_list([7, 2, 7, 4, 7, 6, 7])
12
>>> sum_eo_list([1])
0
```

Takes a list and returns a sum of every other element.

What's wrong with the following code? Identify all the things and fix them.

```
def sum_eo_list(lst, sum=False):
  first, rest = lst[0], lst[1:]
  if sum:
    return first + sum_eo_list(rest, False)
  if lst == []:
    0
  else:
    return sum_eo_list(rest, True)
```

```
def sum_eo_list(lst, sum=False):
  if lst == []: # THE MOST SPECIFIC BASE CASE GOES FIRST
    return 0
  first, rest = lst[0], lst[1:]
  if sum:
    return first + sum_eo_list(rest, False)
  else:
    return sum_eo_list(rest, True)
```

Question 3

Write recursive `fizzbuzz`. Hint: first identify the three things from **Q0b**.
Hint: Remember that `%` (mod) is an useful operator.

```
>>> fizzbuzz(16)
1
2
fizz!
4
buzz!!
fizz!
7
8
fizz!
buzz!!
11
fizz!
13
14
fizzbuzz!!!
16
```

```
def fizzbuzz(n):
    """ Prints everything from 1 to n. If it's divisible by 3, instead it
    prints 'fizz!'. If it's divisible by 5, it prints 'buzz!!'. If it's
    divisible by both, it prints 'fizzbuzz!!!'
    """
    if n == 1:
        print(n)
    else:
        fizzbuzz(n-1)
        if n % 3 == 0 and n % 5 == 0: # Alternatively, n % 15 == 0
            print('fizzbuzz!!!')
        elif n % 3 == 0:
            print('fizz!')
        elif n % 5 == 0:
            print('buzz!!')
        else:
            print(n)
```

Question 4

Write recursive `foobar`: Hint: first identify the three things from **Q0b**.

```
>>> foobar(0)
"foo"
>>> foobar(1)
"foobar"
>>> foobar(2)
"foobarbar"
>>> foobar(3)
"foobarbarfoo"
>>> foobar(4)
"foobarbarfoobar"
>>> foobar(14)
"foobarbarfoobarbarfoobarbarfoobarbarfoobarbar"
```

```
def foobar(n):
  if n == 0:
    return "foo"
  elif n % 3 == 0:
    return foobar(n-1) + "foo"
  else:
    return foobar(n-1) + "bar"
```

Question 5
Write deep_lst_sum. Hint: first identify the three things from **Q0b**.

```
>>> deep_lst_sum((6, 2, (8, 4, 5), (9, (5,))))
39
```

It's dangerous to go alone, take this!

```
def is_list(e):
  """ tests if an element is a list.
  >>> is_list([1, 2])
  True
  >>> is_list(4)
  False
  """
  return type(e) == list

def is_int(e):
  """ tests if an element is an integer
  >>> is_int([1, 2])
  False
  >>> is_int(4)
  True
  """
  return type(e) == int

def deep_lst_sum(lst)
  if lst == []:
    return 0
  else:
    first, rest = lst[0], lst[1:]
    if is_list(first): # Yay function abstraction!
      return deep_lst_sum(first) + deep_lst_sum(rest)
    else:
      return first + deep_lst_sum(rest)
```

Question 6

a) Write `insert_every`. Hint: first identify the three things from **Q0b**.

```
>>> insert_every(1, [[2], [2, 3]])
[[1, 2], [1, 2, 3]]
```

```
def insert_every(elem, lst):
  """ Inserts elem into the front of every
      element in lst (a list of lists)

 >>> insert_every(1, [[2], [2, 3]])
[[1, 2], [1, 2, 3]]
  """
  if lst == []:
    return []
  else:
    return [[elem] + lst[0]] + insert_every(elem, lst[1:])
```

b) Write `powerset`. Hint: first identify the three things from **Q0b**. Hint2: use `insert_every`

```
>>> powerset([1, 2, 3])
[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
```

```
def powerset(lst):
  """ Given a list of distinct elements, produces
  a list of lists where each list is a subset
  of the original list.
  """
  if lst == []:
    return [[]]
  else:
    first, rest = lst[0], lst[1:]
    without_first = powerset(rest)
    return without_first + \
           insert_every(first, without_first)
```

Question 7

Write `longest_subseq`. Hint: first identify the three things from **Q0b**. Hint2: use a helper!

```
>>> longest_inc_subseq([1, 2, 3])
[1, 2, 3]
>>> longest_inc_subseq([2, 1, 2, 3])
[1, 2, 3]
>>> longest_inc_subseq([2, 1, 2, 3, 1])
[1, 2, 3]
>>> longest_inc_subseq([1, 8, 9, 2, 41, 10, 19, 2, 1])
[1, 8, 9, 10, 19]
>>> longest_inc_subseq([1, 1, 12, 3, 8, 1, 9])
[1, 3, 8, 9]
```

```
# I spent a bit of time staring at this and couldn't find a way to make it
easier to read. (Tired Andrew is tired.) The reader that edited this added
comments to try and clarify how the solution works

def longest_inc_subseq(lst):
    def helper(lst, biggest_elem_encountered):
        if lst == []:
            return [] # Longest increasing sequence in an empty list is []
        else:
            first, rest = lst[0], lst[1:] # Split the list

            # What would be the longest increasing subsequence without the
            # first element in this list?
            without_first = helper(rest, biggest_elem_encountered)

            # If no element has been encountered or the first element is
            # bigger than the biggest element encountered (aka, the sequence
            # is increasing)
            if biggest_elem_encountered == None or biggest_elem_encountered <
first:

                # What would be the longest increasing subsequence with the
                # first element in this list?
                with_first = [first] + helper(rest, first)

                # Is the sequence longer if you include the first element?
                if len(with_first) > len(without_first):
                    return with_first
            return without_first
return helper(lst, None)
```

Question 8

Write `middle` without using `len`. Hint: first identify the three things from **Q0b**. Hint2: write a helper function.

```
>>> middle([1, 2, 3, 4, 5])
3
>>> middle([1, 2, 3, 4])
2
>>> middle([1])
1
>>> middle([])
False
```

```
def middle(lst):
  def find_middle(p1, p2):
    if p1 == []:
      return False
    elif p2[2:] == []:
      return p1[0]
    else:
      return find_middle(p1[1:], p2[2:])
  return find_middle(lst, lst)
```

Challenge: did your solution involve counting the number of elements? There's another way. Hint: use `[1:]` and `[2:]`

See above solution. If you have a hard time understanding what the code is doing, try tracing through the code manually.

Question 9

a) Write `add2up`

```
>>> add2up(10, [1, 2, 3, 4, 5])
False
>>> add2up(8, [1, 2, 3, 4, 5]) # 3 + 5 = 8
True
>>> add2up(3, [1, 2, 3, 4, 5]) # 1 + 2 = 3
True
>>> add2up(2, [1, 2, 3, 4, 5]) # ? + ? = 2
False
```

```
def add2up(n, lst):
  if lst == []:
    return Fals``~e
  else:
    first, rest = lst[0], lst[1:]
    return (n - first) in rest or add2up(n, rest)
```

b) Write `addup`. Hint: Tree recursion!

```
>>> addup(10, [1, 2, 3, 4, 5])
True
>>> addup(8, [1, 2, 3, 4, 5])
True
>>> addup(3, [1, 2, 3, 4, 5])
True
>>> addup(2, [1, 2, 3, 4, 5])
True
>>> addup(-1, [1, 2, 3, 4, 5])
False
>>> addup(100, [1, 2, 3, 4, 5])
False
>>> addup(11, [2, 2, 2, 2, 2, 2])
False
```

```
def addup(n, lst):
  if n == 0:
    return True
  if lst == []:
    return False
  else:
     first, rest = lst[0], lst[1:]
     return addup(n-first, rest) or addup(n, rest)
```

Challenge Questions (AKA Welcome to CS 170)

```
def edit_distance(a, b):
    """
    Find the edit distance between two strings.
  Edit distance is defined as the minimum number of insertions
  deletions and character changes it takes to get from one string
  to another.

    >>> edit_distance("saturday", "sunday")
    3
    >>> edit_distance("kitten", "sitting")
    3
    >>> edit_distance("", "this is really weird")
    20
    >>> edit_distance("book", "back")
    2
    >>> edit_distance("hello world", "yo man")
    9
    """
    if not a and not b:
        return 0
    if not a:
        return len(b)
    if not b:
        return len(a)
    elif a[0] == b[0]:
        return edit_distance(a[1:], b[1:])
    insertion = edit_distance(a, b[1:])
    changes = edit_distance(a[1:], b[1:])
    deletion = edit_distance(a[1:], b)
    return 1 + min(insertion, changes, deletion)



def knapsack(weights, profits, weight_capacity):
    """
    You are a thief, and you have a bunch of items in front of you.
    Each item (represented by a certain index) has a corresponding
  weight and profit.
```

```
    You have a weight capacity than you can not exceed (so you can't
  take all the items), what is the maximum profit you can gain
  given the items and the capacity you can carry?

    >>> knapsack([23, 31, 29, 44, 53, 38, 63, 85, 89, 82], \
     [92, 57, 49, 68, 60, 43, 67, 84, 87, 72], 165)
    309
    >>> knapsack([1, 1, 1], [1, 2, 3], 3)
    6
    >>> knapsack([1, 2, 2], [1, 2, 3], 3)
    4
    >>> knapsack([1, 2, 2], [1, 3, 3], 2)
    3
    """
    if len(weights) != len(profits):
        return False
    if weights == [] or weight_capacity <= 0:
        return 0
    first_weight, first_profit = weights[0], profits[0]
    if first_weight > weight_capacity:
        return knapsack(weights[1:], profits[1:], weight_capacity)
    else:
        return max(profits[0] + knapsack(weights[1:], profits[1:],
weight_capacity - first_weight), knapsack(weights[1:], profits[1:],
weight_capacity))




def hanoi(n_rings):
    """
    Calculate the minimum number of moves to solve towers of
    hanoi. Do it recursively.

    >>> hanoi(3)
    7
    >>> hanoi(8)
    255
    >>> hanoi(1)
    1
    >>> hanoi(5)
    31
    """
    if n_rings < 0:
        return 0
    return 2 * hanoi(n - 1) + 1
     #prove that this is exactly pow(2, n) - 1
```

```
def is_maze_solvable(grid):
    """
    Determine if you can get from the top left corner (0, 0) of the
    grid to the bottom right corner (length-1, length-1)

    >>> grid = \
    [[0, 0, 0, 0, 0, 1], \
     [1, 1, 0, 0, 0, 1], \
     [0, 0, 0, 1, 0, 0], \
     [0, 1, 1, 0, 0, 1], \
     [0, 1, 0, 0, 1, 0], \
     [0, 1, 0, 0, 0, 2]]
    >>> is_maze_solvable(grid)
    True
    >>> grid = \
    [[0, 0, 0, 0, 0, 1], \
     [1, 1, 0, 0, 0, 1], \
     [0, 0, 0, 1, 0, 0], \
     [0, 1, 1, 0, 0, 1], \
     [0, 1, 0, 0, 1, 0], \
     [0, 1, 0, 0, 1, 2]]
    >>> is_maze_solvable(grid)
    False
    """
```

```
def min_to_palindrome(start_string):
    """
    Determine the minimum number of insertions and changes
    required to convert start_string into a palindrome.
    (A palindrome is a string that is the same as its reverse.
    E.g: racecar)
```

```
>>> min_to_palindrome("racecar")
0
>>> min_to_palindrome("abc")
1
>>> min_to_palindrome("happ")
2
>>> min_to_palindrome("madama")
1
>>> min_to_palindrome("abcdefghijk")
10
"""
```