Homework 2: Recursion

> Adapted from cs61a of UC Berkeley.

Instructions

Readings: You might find the following references useful:

- Section 1.7
- Section 2.3

Starter Files

Get your starter file by cloning the repository: https://github.com/JacyCui/sicp-hw02.git

```
git clone https://github.com/JacyCui/sicp-hw02.git
```

`hw02.zip` is the starter file you need, you might need to unzip the file to get the skeleton code.

```
unzip hw02.zip
```

`README.md` is the handout for this homework. `solution` is a probrab solution of the homework. However, I might not give my solution exactly when the homework is posted. You need to finish the task on your own first. If any problems occurs, please make use of the comment section.

Required questions

Q1: Num eights

Write a recursive function `num_eights` that takes a positive integer `x` and returns the number of times the digit 8 appears in `x`. *Use recursion - the tests will fail if you use any assignment statements.*

```
def num_eights(x):
    """Returns the number of times 8 appears as a digit of x.

    >>> num_eights(3)
    0
    >>> num_eights(8)
    1
    >>> num_eights(88888888)
```

```
    8
    >>> num_eights(2638)
    1
    >>> num_eights(86380)
    2
    >>> num_eights(12345)
    0
    >>> from construct_check import check
    >>> # ban all assignment statements
    >>> check(HW_SOURCE_FILE, 'num_eights',
    ...       ['Assign', 'AugAssign'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q num_eights --local
```

Q2: Ping-pong

The ping-pong sequence counts up starting from 1 and is always either counting up or counting down. At element `k`, the direction switches if `k` is a multiple of 8 or contains the digit 8. The first 30 elements of the ping-pong sequence are listed below, with direction swaps marked using brackets at the 8th, 16th, 18th, 24th, and 28th elements:

Index	1	2	3	4	5	6	7	[8]	9	10	11	12	13	14	15	[16]	17	[18]	19	20	21	22	23
PingPong Value	1	2	3	4	5	6	7	[8]	7	6	5	4	3	2	1	[0]	1	[2]	1	0	-1	-2	-3

Index	[24]	25	26	27	[28]	29	30
PingPong Value	[-4]	-3	-2	-1	0	-1	-2

Implement a function `pingpong` that returns the nth element of the ping-pong sequence *without using any assignment statements*.

You may use the function `num_eights`, which you defined in the previous question.

Use recursion - the tests will fail if you use any assignment statements.

> *Hint*: If you're stuck, first try implementing `pingpong` using assignment statements and a `while` statement. Then, to convert this into a recursive solution, write a helper function that has a parameter for each variable that changes values in the body of the while loop.

```
def pingpong(n):
    """Return the nth element of the ping-pong sequence.

```

```
    >>> pingpong(8)
    8
    >>> pingpong(10)
    6
    >>> pingpong(15)
    1
    >>> pingpong(21)
    -1
    >>> pingpong(22)
    -2
    >>> pingpong(30)
    -2
    >>> pingpong(68)
    0
    >>> pingpong(69)
    -1
    >>> pingpong(80)
    0
    >>> pingpong(81)
    1
    >>> pingpong(82)
    0
    >>> pingpong(100)
    -6
    >>> from construct_check import check
    >>> # ban assignment statements
    >>> check(HW_SOURCE_FILE, 'pingpong', ['Assign', 'AugAssign'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q pingpong --local
```

Q3: Missing Digits

Write the recursive function `missing_digits` that takes a number `n` that is sorted in increasing order (for example, `12289` is valid but `15362` and `98764` are not). It returns the number of missing digits in `n`. A missing digit is a number between the first and last digit of `n` of a that is not in `n`.

Use recursion - the tests will fail if you use while or for loops.

```
def missing_digits(n):
    """Given a number a that is in sorted, increasing order,
    return the number of missing digits in n. A missing digit is
    a number between the first and last digit of a that is not in n.
```

```
    >>> missing_digits(1248) # 3, 5, 6, 7
    4
    >>> missing_digits(1122) # No missing numbers
    0
    >>> missing_digits(123456) # No missing numbers
    0
    >>> missing_digits(3558) # 4, 6, 7
    3
    >>> missing_digits(35578) # 4, 6
    2
    >>> missing_digits(12456) # 3
    1
    >>> missing_digits(16789) # 2, 3, 4, 5
    4
    >>> missing_digits(19) # 2, 3, 4, 5, 6, 7, 8
    7
    >>> missing_digits(4) # No missing numbers between 4 and 4
    0
    >>> from construct_check import check
    >>> # ban while or for loops
    >>> check(HW_SOURCE_FILE, 'missing_digits', ['While', 'For'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q missing_digits --local
```

Q4: Count coins

Given a positive integer `total`, a set of coins makes change for `total` if the sum of the values of the coins is `total`. Here we will use standard US Coin values: 1, 5, 10, 25 For example, the following sets make change for `15`:

- 15 1-cent coins
- 10 1-cent, 1 5-cent coins
- 5 1-cent, 2 5-cent coins
- 5 1-cent, 1 10-cent coins
- 3 5-cent coins
- 1 5-cent, 1 10-cent coin

Thus, there are 6 ways to make change for `15`. Write a recursive function `count_coins` that takes a positive integer `total` and returns the number of ways to make change for `total` using coins. Use the `next_largest_coin` function given to you to calculate the next largest coin denomination given your current coin. I.e. `next_largest_coin(5)` = `10`.

> *Hint:* Refer the implementation of `count_partitions` for an example of how to count the ways to sum up to a total with smaller parts. If you need to keep track of more than one value across recursive calls, consider writing a helper function.

```
def next_largest_coin(coin):
    """Return the next coin.
    >>> next_largest_coin(1)
    5
    >>> next_largest_coin(5)
    10
    >>> next_largest_coin(10)
    25
    >>> next_largest_coin(2) # Other values return None
    """
    if coin == 1:
        return 5
    elif coin == 5:
        return 10
    elif coin == 10:
        return 25

def count_coins(total):
    """Return the number of ways to make change for total using coins of value of 1, 5, 10, 25.
    >>> count_coins(15)
    6
    >>> count_coins(10)
    4
    >>> count_coins(20)
    9
    >>> count_coins(100) # How many ways to make change for a dollar?
    242
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(HW_SOURCE_FILE, 'count_coins', ['While', 'For'])

    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q count_coins --local
```

Just for Fun Questions

> This question demonstrates that it's possible to write recursive functions without assigning them a name in the global frame.

Q5: Anonymous factorial

The recursive factorial function can be written as a single expression by using a conditional expression(covered in previous lectures).

```
>>> fact = lambda n: 1 if n == 1 else mul(n, fact(sub(n, 1)))
>>> fact(5)
120
```

However, this implementation relies on the fact (no pun intended) that `fact` has a name, to which we refer in the body of `fact`. To write a recursive function, we have always given it a name using a `def` or assignment statement so that we can refer to the function within its own body. In this question, your job is to define fact recursively without giving it a name!

Write an expression that computes `n` factorial using only call expressions, conditional expressions, and lambda expressions (no assignment or def statements). *Note in particular that you are not allowed to use `make_anonymous_factorial` in your return expression.* The `sub` and `mul` functions from the `operator` module are the only built-in functions required to solve this problem:

```
from operator import sub, mul

def make_anonymous_factorial():
    """Return the value of an expression that computes factorial.

    >>> make_anonymous_factorial()(5)
    120
    >>> from construct_check import check
    >>> # ban any assignments or recursion
    >>> check(HW_SOURCE_FILE, 'make_anonymous_factorial', ['Assign', 'AugAssign',
'FunctionDef', 'Recursion'])
    True
    """
    return 'YOUR_EXPRESSION_HERE'
```

Use Ok to test your code:

```
python3 ok -q make_anonymous_factorial --local
```

Finally, you can run doctest to check your answer again.

```
python3 -m doctest hw02.py
```