CS 61A: Homework 3

Due by 11:59pm on Thursday, 2/19

Instructions

Download hw03.zip. Inside the archive, you will find a file called hw03.py, along with a copy of the OK autograder.

Submission: When you are done, submit with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be scored.

Using OK

The `ok` program helps you test your code and track your progress. The first time you run the autograder, you will be asked to log in with your @berkeley.edu account using your web browser. Please do so. Each time you run ok, it will back up your work and progress on our servers. You can run all the doctests with the following command:

```
python3 ok
```

To test a specific question, use the `-q` option with the name of the function:

```
python3 ok -q <function>
```

By default, only tests that fail will appear. If you want to see how you did on all tests, you can use the `-v` option:

```
python3 ok -v
```

If you do not want to send your progress to our server or you have any problems logging in, add the `--local` flag to block all communication:

```
python3 ok --local
```

When you are ready to submit, run `ok` with the `--submit` option:

```
python3 ok --submit
```

Readings: You might find the following references useful:

- Section 1.7

Table of Contents

Question 1

A mathematical function G on positive integers is defined by two cases:

```
G(n) = n,                                  if n <= 3
G(n) = G(n - 1) + 2 * G(n - 2) + 3 * G(n - 3),  if n > 3
```

Write a recursive function g that computes G(n). Then, write an iterative function g_iter that also computes G(n):

```
def g(n):
  """Return the value of G(n), computed recursively.

  >>> g(1)
  1
  >>> g(2)
  2
  >>> g(3)
  3
  >>> g(4)
  10
  >>> g(5)
  22
  """
  "*** YOUR CODE HERE ***"

def g_iter(n):
  """Return the value of G(n), computed iteratively.

  >>> g_iter(1)
  1
  >>> g_iter(2)
  2
  >>> g_iter(3)
  3
  >>> g_iter(4)
  10
  >>> g_iter(5)
  22
  """
  "*** YOUR CODE HERE ***"
```

Question 2

Write a function has_seven that takes a positive integer n and returns whether n contains the digit 7. Do not use any assignment statements - use recursion instead:

```
def has_seven(k):
  """Returns True if at least one of the digits of k is a 7, False otherwise.

  >>> has_seven(3)
  False
  >>> has_seven(7)
  True
  >>> has_seven(2734)
  True
  >>> has_seven(2634)
  False
  >>> has_seven(734)
  True
  >>> has_seven(7777)
  True
  """
  "*** YOUR CODE HERE ***"
```

Question 3

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 7 or contains the digit 7. The first

30 elements of the ping-pong sequence are listed below, with direction swaps marked using brackets at the 7th, 14th, 17th, 21st, 27th, and 28th elements:

```
1 2 3 4 5 6 [7] 6 5 4 3 2 1 [0] 1 2 [3] 2 1 0 [-1] 0 1 2 3 4 [5] [4] 5 6
```

Implement a function pingpong that returns the nth element of the ping-pong sequence. Do not use any assignment statements; however, you may use def statements.

Hint: If you're stuck, try implementing pingpong first using assignment and a while statement, then try a recursive implementation without assignment:

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

  >>> pingpong(7)
  7
  >>> pingpong(8)
  6
  >>> pingpong(15)
  1
  >>> pingpong(21)
  -1
  >>> pingpong(22)
  0
  >>> pingpong(30)
  6
  >>> pingpong(68)
  2
  >>> pingpong(69)
  1
  >>> pingpong(70)
  0
  >>> pingpong(71)
  1
  >>> pingpong(72)
  0
  >>> pingpong(100)
  2
  """
  "*** YOUR CODE HERE ***"
```

Question 4

Once the machines take over, the denomination of every coin will be a power of two: 1-cent, 2-cent, 4-cent, 8-cent, 16-cent, etc. There will be no limit to how much a coin can be worth.

A set of coins makes change for n if the sum of the values of the coins is n. For example, the following sets make change for 7:

- 7 1-cent coins
- 5 1-cent, 1 2-cent coins
- 3 1-cent, 2 2-cent coins
- 3 1-cent, 1 4-cent coins
- 1 1-cent, 3 2-cent coins
- 1 1-cent, 1 2-cent, 1 4-cent coins

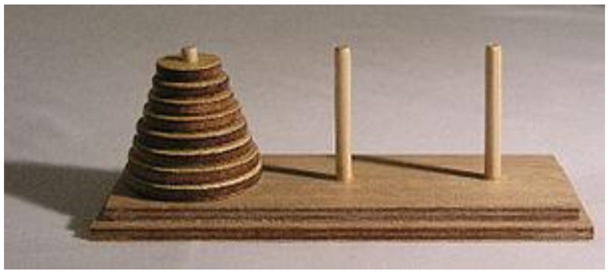
Thus, there are 6 ways to make change for 7. Write a function count_change that takes a positive integer n and returns the number of ways to make change for n using these coins of the future:

```
def count_change(amount):
  """Return the number of ways to make change for amount.

  >>> count_change(7)
  6
  >>> count_change(10)
  14
  >>> count_change(20)
  60
  >>> count_change(100)
  9828
  """
  "*** YOUR CODE HERE ***"
```

Question 5

A classic puzzle called the Towers of Hanoi is a game that consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with n disks in a neat stack in ascending order of size on a start rod, the smallest at the top, forming a conical shape.



The objective of the puzzle is to move the entire stack to an end rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the top (smallest) disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

Complete the definition of towers_of_hanoi which prints out the steps to solve this puzzle for any number of n disks starting from the start rod and moving them to the end rod:

```
def towers_of_hanoi(n, start, end):
  """Print the moves required to solve the towers of hanoi game, starting
  with n disks on the start pole and finishing on the end pole.

  The game is to assumed to have 3 poles.

  >>> towers_of_hanoi(1, 1, 3)
  Move the top disk from rod 1 to rod 3
  >>> towers_of_hanoi(2, 1, 3)
  Move the top disk from rod 1 to rod 2
  Move the top disk from rod 1 to rod 3
  Move the top disk from rod 2 to rod 3
  >>> towers_of_hanoi(3, 1, 3)
  Move the top disk from rod 1 to rod 3
  Move the top disk from rod 1 to rod 2
  Move the top disk from rod 3 to rod 2
  Move the top disk from rod 1 to rod 3
  Move the top disk from rod 2 to rod 1
  Move the top disk from rod 2 to rod 3
  Move the top disk from rod 1 to rod 3
  """
  assert 0 < start <= 3 and 0 < end <= 3 and start != end, "Bad start/end"
  "*** YOUR CODE HERE ***"
```

Question 6: Challenge Problem (optional)

The recursive factorial function can be written as a single expression by using a conditional expression.

```
>>> 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 function 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
  """
  return 'YOUR_EXPRESSION_HERE'
```