Homework 2: Higher Order Functions and Recursion hw02.zip (hw02.zip)

Due by 11:59pm on Wednesday, July 7

Instructions

Download hw02.zip (hw02.zip). Inside the archive, you will find a file called hw02.py (hw02.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. Check that you have successfully submitted your code on okpy.org (https://okpy.org/). See Lab 0 (/~cs61a/su21/lab/lab00#submitting-the-assignment) for more instructions on submitting assignments.

Using Ok: If you have any questions about using Ok, please refer to this guide. (/~cs61a/su21/articles/using-ok)

Readings: You might find the following references useful:

- Section 1.6 (https://composingprograms.com/pages/16-higher-order-functions.html)
- Section 1.7 (https://composingprograms.com/pages/17-recursive-functions.html)

Grading: Homework is graded based on correctness. Each incorrect problem will decrease the total score by one point. There is a homework recovery policy as stated in the syllabus. **This homework is out of 3 points.**

The `construct_check` module is used in this assignment, which defines the function `check`. For example, a call such as

```
check("foo.py", "func1", ["While", "For", "Recursion"])
```

checks that the function `func1` in file `foo.py` does *not* contain any `while` or `for` constructs, and is not an overtly recursive function (i.e., one in which a function contains a call to itself by name.) This restriction does not apply to all problems in this assignment. If this restriction applies, you will see a call to `check` somewhere in the problem's doctests.

Required questions

Important Functions

Several doctests refer to these functions:

```
from operator import add, mul, sub

square = lambda x: x * x

identity = lambda x: x

triple = lambda x: 3 * x

increment = lambda x: x + 1
```

Higher Order Functions

Q1: Make Repeater

Implement the function `make_repeater` so that `make_repeater(func, n)(x)` returns `func(func(...func(x)...))`, where `func` is applied `n` times. That is, `make_repeater(func, n)` returns another function that can then be applied to another argument. For example, `make_repeater(square, 3)(42)` evaluates to `square(square(square(42)))`.

```
def make_repeater(func, n):
    """Return the function that computes the nth application of func.

    >>> add_three = make_repeater(increment, 3)
    >>> add_three(5)
    8
    >>> make_repeater(triple, 5)(1) # 3 * 3 * 3 * 3 * 3 * 1
    243
    >>> make_repeater(square, 2)(5) # square(square(5))
    625
    >>> make_repeater(square, 4)(5) # square(square(square(square(5))))
    152587890625
    >>> make_repeater(square, 0)(5) # Yes, it makes sense to apply the function zero times!
    5
    """
    "*** YOUR CODE HERE ***"
```

> For an extra challenge, try defining `make_repeater` using `compose1` and your `accumulate` function in a single one-line return statement.

```
def compose1(func1, func2):
    """Return a function f, such that f(x) = func1(func2(x))."""
    def f(x):
        return func1(func2(x))
    return f
```

Watch the hints video below for somewhere to start:

Use Ok to test your code:

```
python3 ok -q make_repeater
```

Recursion

Q2: Num eights

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

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

    >>> 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 ***"
```

Watch the hints video below for somewhere to start:

Use Ok to test your code:

```
python3 ok -q num_eights
```

Q3: 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 (cont.)	**[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 ***"
```

Watch the hints video below for somewhere to start:

Use Ok to test your code:

```
python3 ok -q pingpong
```

Q4: 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(19) # 2, 3, 4, 5, 6, 7, 8
    7
    >>> 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(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 ***"
```

Watch the hints video below for somewhere to start:

Use Ok to test your code:

```
python3 ok -q missing_digits
```

Q5: Count coins

Given a positive integer `change`, a set of coins makes change for `change` if the sum of the values of the coins is `change`. 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 `change` and returns the number of ways to make change for `change` using coins. Use the `get_next_coin` function given to you to calculate the next largest coin denomination given your current coin. I.e. `get_next_coin(5) = 10`.

> *Hint:* Refer the implementation (http://composingprograms.com/pages/17-recursive-functions.html#example-partitions) of `count_partitions` for an example of how to count the ways to sum up to a final value with smaller parts. If you need to keep track of more than one value across recursive calls, consider writing a helper function.

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

def count_coins(change):
    """Return the number of ways to make change 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 ***"
```

Watch the hints video below for somewhere to start:

Use Ok to test your code:

```
python3 ok -q count_coins
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Just for fun Question

> *This question is out of scope for 61a. Do it if you want an extra challenge or some practice with HOF and abstraction!*

Q6: Church numerals

The logician Alonzo Church invented a system of representing non-negative integers entirely using functions. The purpose was to show that functions are sufficient to describe all of number theory: if we have functions, we do not need to assume that numbers exist, but instead we can invent them.

Your goal in this problem is to rediscover this representation known as *Church numerals*. Here are the definitions of `zero`, as well as a function that returns one more than its argument:

```
def zero(f):
    return lambda x: x

def successor(n):
    return lambda f: lambda x: f(n(f)(x))
```

First, define functions `one` and `two` such that they have the same behavior as `successor(zero)` and `successsor(successor(zero))` respectively, but *do not call* `successor` *in your implementation*.

Next, implement a function `church_to_int` that converts a church numeral argument to a regular Python integer.

Finally, implement functions `add_church`, `mul_church`, and `pow_church` that perform addition, multiplication, and exponentiation on church numerals.

```
def one(f):
    """Church numeral 1: same as successor(zero)"""
    "*** YOUR CODE HERE ***"

def two(f):
    """Church numeral 2: same as successor(successor(zero))"""
    "*** YOUR CODE HERE ***"

three = successor(two)

def church_to_int(n):
    """Convert the Church numeral n to a Python integer.

    >>> church_to_int(zero)
    0
    >>> church_to_int(one)
    1
    >>> church_to_int(two)
    2
    >>> church_to_int(three)
    3
    """
    "*** YOUR CODE HERE ***"

def add_church(m, n):
    """Return the Church numeral for m + n, for Church numerals m and n.

    >>> church_to_int(add_church(two, three))
    5
    """
    "*** YOUR CODE HERE ***"

def mul_church(m, n):
    """Return the Church numeral for m * n, for Church numerals m and n.

    >>> four = successor(three)
    >>> church_to_int(mul_church(two, three))
    6
    >>> church_to_int(mul_church(three, four))
    12
    """
    "*** YOUR CODE HERE ***"

def pow_church(m, n):
    """Return the Church numeral m ** n, for Church numerals m and n.

    >>> church_to_int(pow_church(two, three))
    8
    >>> church_to_int(pow_church(three, two))
    9
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q church_to_int
python3 ok -q add_church
python3 ok -q mul_church
python3 ok -q pow_church
```