Instructions

Please download homework materials `hw03.zip` from our QQ group if you don't have one.

In this homework, you are required to complete the problems described in section 2. The starter code for these problems is provided in `hw03.py`, which is distributed as part of the homework materials in the `code` directory.

We have also prepared two optional problems just for fun in section 3. You can find further descriptions there.

Submission: As instructed before, you need to submit your work with Ok by `python ok --submit`. You may submit more than once before the deadline, and your score of this assignment will be the highest one of all your submissions.

Readings: You might find the following references to the textbook useful:

- Section 1.7

The `construct_check` module in `construct_check.py` 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). Note that 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 Problems

In this section, you are required to complete the problems below and submit your code to OJ website.

Remember, you can use `ok` to test your code:

```
$ python ok  # test all functions
$ python ok -q <func>  # test single function
```

Problem 1: Number of Six (100pts)

Write a recursive function `number_of_six` that takes a positive integer `x` and returns the number of times the digit 6 appears in `x`.

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

```
def number_of_six(n):
    """Return the number of 6 in each digit of a positive integer n.

    >>> number_of_six(666)
    3
    >>> number_of_six(123456)
    1
    >>> from construct_check import check
    >>> # ban all assignment statements
    >>> check(HW_SOURCE_FILE, 'number_of_six',
    ...       ['Assign', 'AugAssign'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Problem 2: Ping-pong (200pts)

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 6 or contains the digit 6. The first 30 elements of the ping-pong sequence are listed below, with direction swaps marked using brackets at the 6th, 12th, 16th, 18st, 24th, 26th and 30th elements:

Index	**1**	**2**	**3**	**4**	**5**	**[6]**	**7**	**8**	**9**	**10**
PingPong Value	1	2	3	4	5	[6]	5	4	3	2
Index (cont.)	**11**	**[12]**	**13**	**14**	**15**	**[16]**	**17**	**[18]**	**19**	**20**
PingPong Value	1	[0]	1	2	3	[4]	3	[2]	3	4
Index (cont.)	**21**	**22**	**23**	**[24]**	**25**	**[26]**	**27**	**28**	**29**	**[30]**
PingPong Value	5	6	7	[8]	7	[6]	7	8	9	[10]

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

You may use the function `number_of_six`, 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.

注意：我们期望你的“递归”代码**和等价的“循环”代码有差不多的效率**。这道题的`n`可能很大，你应该试试你的代码计算`pingpong(500)`要花多少时间（一瞬间？几秒钟？还是几千年？）才能得到结果。

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

    >>> pingpong(7)
    5
    >>> pingpong(8)
    4
    >>> pingpong(15)
    3
    >>> pingpong(21)
    5
    >>> pingpong(22)
    6
    >>> pingpong(30)
    10
    >>> pingpong(68)
    0
    >>> pingpong(69)
    1
    >>> pingpong(70)
    0
    >>> pingpong(71)
    -1
    >>> pingpong(72)
    -2
    >>> pingpong(100)
    6
    >>> from construct_check import check
    >>> # ban assignment statements
    >>> check(HW_SOURCE_FILE, 'pingpong', ['Assign', 'AugAssign'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Problem 3: Missing Digits (100pts)

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` 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 n 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 n 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(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 ***"
```

Problem 4: Count change (200pts)

Denomination is a proper description of a currency amount, usually for coins or banknotes. For example, we have 1 Yuan coins and 100 Yuan bills in Chinese Yuan.

A money function is a function which takes a positive integer representing a denomination and returns the next larger denomination of the input or `None` if there is no larger denomination. Its behavior is undefined if the input is not a valid denomination.

Money function can be used to describe a currency system. For example, the following money function describes Chinese Yuan (1 Yuan, 5 Yuan, 10 Yuan, 20 Yuan, 50 Yuan and 100 Yuan). You can assume the smallest denomination of all currencies is always 1.

```
def chinese_yuan(money):
    if money == 1:
        return 5
    if money == 5:
        return 10
    if money == 10:
        return 20
    if money == 20:
        return 50
    if money == 50:
        return 100
```

Given an amount of money `total` and a currency system, a set of banknotes (coins) makes change for `total` if sum of their values is exactly `total`. For example, the following sets make change for 15 Chinese Yuan:

- 15 1-Yuan
- 10 1-Yuan, 1 5-Yuan
- 5 1-Yuan, 2 5-Yuan
- 5 1-Yuan, 1 10-Yuan
- 3 5-Yuan
- 1 5-Yuan, 1 10-Yuan

Thus, there are 6 different ways to make change for 15 Chinese Yuan.

Write a recursive function `count_change` that takes a positive integer `total` and a money function `next_money` and returns the number of ways to make change for `total` under the currency system described by `next_money`.

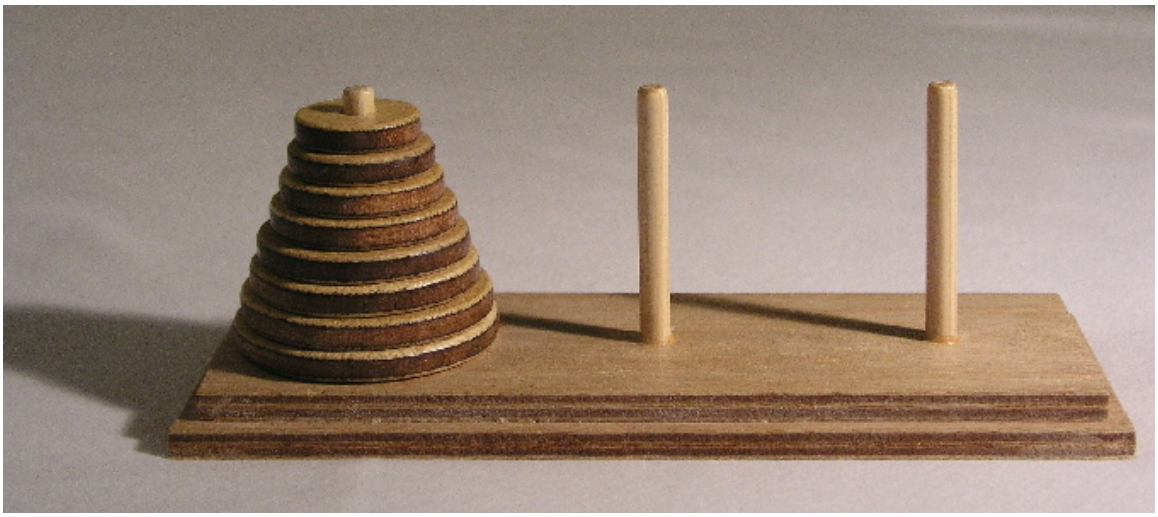
Hint: Refer the implementation of `count_partitions` for an example of how to count the ways to sum up to a total with smaller parts. If you need to keep track of more than one value across recursive calls, consider writing a helper function.

```
def count_change(total, next_money):
    """Return the number of ways to make change for total,
    under the currency system described by next_money.

    >>> def chinese_yuan(money):
    ...     if money == 1:
    ...         return 5
    ...     if money == 5:
    ...         return 10
    ...     if money == 10:
    ...         return 20
    ...     if money == 20:
    ...         return 50
    ...     if money == 50:
    ...         return 100
    >>> def us_cent(money):
    ...     if money == 1:
    ...         return 5
    ...     elif money == 5:
    ...         return 10
    ...     elif money == 10:
    ...         return 25
    >>> count_change(15, chinese_yuan)
    6
    >>> count_change(49, chinese_yuan)
    44
    >>> count_change(49, us_cent)
    39
    >>> count_change(49, lambda x: x * 2)
    692
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(HW_SOURCE_FILE, 'count_change', ['While', 'For'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Problem 5: Towers of Hanoi (100pts)

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 `move_stack`, which prints out the steps required to move `n` disks from the `start` rod to the `end` rod without violating the rules. The provided `print_move` function will print out the step to move a single disk from the given `origin` to the given `destination`.

Hint: Draw out a few games with various `n` on a piece of paper and try to find a pattern of disk movements that applies to any `n`. In your solution, take the recursive leap of faith whenever you need to move any amount of disks less than `n` from one rod to another. If you need more help, see the following hints.

Extra hints

- *Hint1*: See the animation of the Towers of Hanoi, found on Wikimedia by user Trixx
- *Hint2*: The strategy used in Towers of Hanoi is to move all but the bottom disc to the second peg, then moving the bottom disc to the third peg, then moving all but the second disc from the second to the third peg.
- *Hint3*: One thing you don't need to worry about is collecting all the steps. `print` effectively "collects" all the results in the terminal as long as you make sure that the moves are printed in order.

```
def print_move(origin, destination):
    """Print instructions to move a disk."""
    print("Move the top disk from rod", origin, "to rod", destination)

def move_stack(n, start, end):
    """Print the moves required to move n disks on the start pole to the end
    pole without violating the rules of Towers of Hanoi.

    n -- number of disks
    start -- a pole position, either 1, 2, or 3
    end -- a pole position, either 1, 2, or 3

    There are exactly three poles, and start and end must be different. Assume
    that the start pole has at least n disks of increasing size, and the end
    pole is either empty or has a top disk larger than the top n start disks.

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

Problem 6: Multiadder (100pts)

An order `1` numeric function is a function that takes a number and returns a number. An order `2` numeric function is a function that takes a number and returns an order `1` numeric function. Likewise, an order `n` numeric function is a function that takes a number and returns an order `n-1` numeric function. The argument sequence of a nested call expression is the sequence of all arguments in all subexpressions, in the order they appear. For example, the expression `f(3)(4)(5)(6)(7)` has the argument sequence 3, 4, 5, 6, 7. Note that all numeric functions are pure functions.

Implement `multiadder`, which takes a positive integer `n` and returns an order `n` numeric function that sums an argument sequence of length `n`.

```
def multiadder(n):
    """Return a function that takes N arguments, one at a time, and adds them.
    >>> f = multiadder(3)
    >>> f(5)(6)(7) # 5 + 6 + 7
    18
    >>> multiadder(1)(5)
    5
    >>> multiadder(2)(5)(6) # 5 + 6
    11
    >>> multiadder(4)(5)(6)(7)(8) # 5 + 6 + 7 + 8
    26
    >>> # Make sure multiadder is a pure function.
    >>> check(HW_SOURCE_FILE, 'number_of_six',
    ...       ['Nonlocal', 'Global'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Just for fun Problems

This section is out of scope for our course, so the problems below is optional. That is, the problems in this section **don't** count for your final score and **don't** have any deadline. Do it at any time if you want an extra challenge or some practice with higher order function and abstraction!

To check the correctness of your answer, you can submit your code to Contest 'Just for fun'.

Problem 7: Anonymous factorial (0pts)

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
```

> The ternary operator `<a> if <bool-exp> else <b>` evaluates to `<a>` if `<bool-exp>` is truthy and evaluates to `<b>` if `<bool-exp>` is false-y.

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'
```

Problem 8: All-Ys Has Been (0pts)

Given mystery function `Y`, complete `fib_maker` and `number_of_six_maker` so that the given doctests work correctly.

When `Y` is called on `fib_maker`, it should return a function which takes a positive integer `n` and returns the `n`th Fibonacci number.

Similarly, when `Y` is called on `number_of_six_maker` it should return a function that takes a positive integer `x` and returns the number of times the digit 6 appears in `x`.

> Hint: You may use the ternary operator `<a> if <bool-exp> else <b>`, which evaluates to `<a>` if `<bool-exp>` is truthy and evaluates to `<b>` if `<bool-exp>` is false-y.

```
Y = lambda f: (lambda x: x(x))(lambda x: f(lambda z: x(x)(z)))
fib_maker = lambda f: lambda r: 'YOUR_EXPRESSION_HERE'
number_of_six_maker = lambda f: lambda r: 'YOUR_EXPRESSION_HERE'

my_fib = Y(fib_maker)
my_number_of_six = Y(number_of_six_maker)

# This code sets up doctests for my_fib and my_number_of_six.

my_fib.__name__ = 'my_fib'
my_fib.__doc__="""Given n, returns the nth Fibonacci nuimber.

>>> my_fib(0)
0
>>> my_fib(1)
1
>>> my_fib(2)
1
>>> my_fib(3)
2
>>> my_fib(4)
3
>>> my_fib(5)
5
"""

my_number_of_six.__name__ = 'my_number_of_six'
my_number_of_six.__doc__="""Return the number of 6 in each digit of a positive integer
n.

>>> my_number_of_six(666)
3
>>> my_number_of_six(123456)
1
"""
```

Remember to use Ok to test your code:

```
$ python ok -q my_fib
$ python ok -q my_number_of_six
```