Homework 03: Recursion and Tree Recursion

1. 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: When you are done, submit your code to our OJ website as instructed in lab00. You may submit more than once before the deadline; only the final submission will be scored. See lab00 for more instructions on submitting assignments.

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

- Section 1.7

> The `construct_check` module in `code/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.

2. Required Problems

In this section, you are required to complete the problems below and submit your code to `Contest hw03` in our OJ website as instructed in lab00 to get your answer scored.

Problem 1: Num sevens (100pts)

Write a recursive function `num_sevens` that takes a positive integer `x` and returns the number of times the digit 7 appears in `x`.

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

```
def num_sevens(x):
    """Returns the number of times 7 appears as a digit of x.

    >>> num_sevens(3)
    0
    >>> num_sevens(7)
    1
    >>> num_sevens(7777777)
```

```
    7
    >>> num_sevens(2637)
    1
    >>> num_sevens(76370)
    2
    >>> num_sevens(12345)
    0
    >>> from construct_check import check
    >>> # ban all assignment statements
    >>> check(HW_SOURCE_FILE, 'num_sevens',
    ...       ['Assign', 'AugAssign'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Remember to use doctest to test your code:

```
$ python -m doctest hw03.py
```

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 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:

Index	1	2	3	4	5	6	[7]	8	9	10	11	12	13	[14]	15
PingPong Value	1	2	3	4	5	6	[7]	6	5	4	3	2	1	[0]	1

Index (cont.)	16	[17]	18	19	20	[21]	22	23	24	25	26	[27]	[28]	29	30
PingPong Value	2	3	4	[5]	[4]	5	6	2	[3]	2	1	0	[-1]	0	1

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_sevens`, 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(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
    >>> from construct_check import check
    >>> # ban assignment statements
    >>> check(HW_SOURCE_FILE, 'pingpong', ['Assign', 'AugAssign'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Problem 3: Count change (200pts)

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.

Given a positive integer `total`, a set of coins makes change for `total` if the sum of the values of the coins is `total`. 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 recursive function `count_change` that takes a positive integer `total` and returns the number of ways to make change for `total` using these coins of the future.

> *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):
    """Return the number of ways to make change for total.

    >>> count_change(7)
    6
    >>> count_change(10)
    14
```

```
    >>> count_change(20)
    60
    >>> count_change(100)
    9828
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(HW_SOURCE_FILE, 'count_change', ['While', 'For'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Problem 4: 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` 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(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 ***"
```

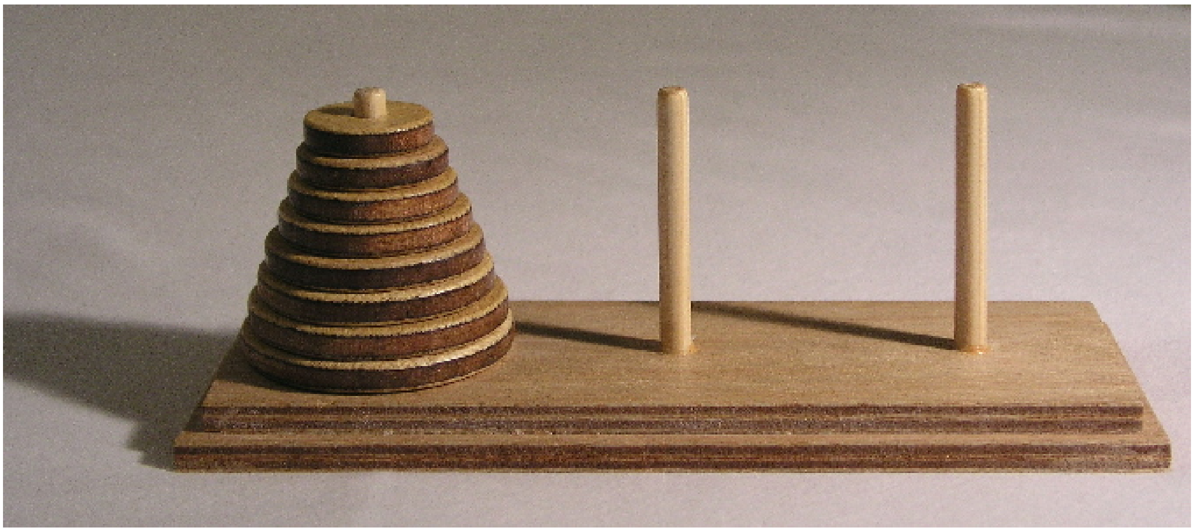
3. Just for fun Problems

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 high order function and abstraction!

To check the correctness of your answer, you can submit your code to `Contest 'Just for fun'` in our [OJ website]() as instructed in lab00.

Problem 5: Towers of Hanoi (0pts)

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: 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
```

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'
```