Homework 3 Solutions **hw03.zip (hw03.zip)**

Solution Files

You can find solutions for all questions in hw03.py (hw03.py).

Required questions

Q1: Has Seven

Write a recursive function `has_seven` that takes a positive integer `n` and returns whether `n` contains the digit `7`. *Use recursion - the tests will fail if you use any assignment statements.*

```
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(374)
    True
    >>> has_seven(140)
    False
    >>> from construct_check import check
    >>> check(HW_SOURCE_FILE, 'has_seven',
    ...       ['Assign', 'AugAssign'])
    True
    """
    if k % 10 == 7:
        return True
    elif k < 10:
        return False
    else:
        return has_seven(k // 10)
```

Use Ok to test your code:

```
python3 ok -q has_seven
```

The equivalent iterative version of this problem might look something like this:

```
while n > 0:
    if n % 10 == 7:
        return True
    n = n // 10
```

The main idea is that we check each digit for a seven. The recursive solution is similar, except that you depend on the recursive call to check the rest of the number for a seven. All that's left is to check the last digit in the current step.

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 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 *without using any assignment statements*.

You may use the function `has_seven` from the previous problem.

> *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
    >>> check(HW_SOURCE_FILE, 'pingpong', ['Assign', 'AugAssign'])
    True
    """
    def helper(result, i, step):
        if i == n:
            return result
        elif i % 7 == 0 or has_seven(i):
            return helper(result - step, i + 1, -step)
        else:
            return helper(result + step, i + 1, step)
    return helper(1, 1, 1)

# Alternate solution 1
def pingpong_next(x, i, step):
    if i == n:
        return x
    return pingpong_next(x + step, i + 1, next_dir(step, i+1))
```

```
def next_dir(step, i):
    if i % 7 == 0 or has_seven(i):
        return -step
    return step

return pingpong_next(1, 1, 1)

# Alternate solution 2
def pingpong(n):
    if n <= 7:
        return n
    return direction(n) + pingpong(n-1)

def direction(n):
    if n < 7:
        return 1
    if (n-1) % 7 == 0 or has_seven(n-1):
        return -1 * direction(n-1)
    return direction(n-1)
```

Use Ok to test your code:

```
python3 ok -q pingpong
```

This is a fairly involved recursion problem, which we will first solve through iteration and then convert to a recursive solution.

Note that at any given point in the sequence, we need to keep track of the current *value* of the sequence (this is the value that might be output) as well as the current *index* of the sequence (how many items we have seen so far, not actually output).

For example, 14th element has *value* 0, but it's the 14th *index* in the sequence. We will refer to the value as `x` and the index as `i`. An iterative solution may look something like this:

```
def pingpong(n):
    i = 1
    x = 1
    while i < n:
        x += 1
        i += 1
    return x
```

Hopefully, it is clear to you that this has a big problem. This doesn't account for changes in directions at all! It will work for the first seven values of the sequence, but then fail after that. To fix this, we can add in a check for direction, and then also keep track of the current direction to make our lives a bit easier (it's possible to compute the direction from scratch at each step, see the `direction` function in the alternate solution).

```
def pingpong(n):
    i = 1
    x = 1
    is_up = True
    while i < n:
        is_up = next_dir(...)
        if is_up:
            x += 1
        else:
            x -= 1
        i += 1
    return x
```

All that's left to do is to write the `next_dir` function, which will take in the *current direction* and *index* and then tell us what direction to go in next (which could be the same direction):

```
def next_dir(is_up, i):
    if i % 7 == 0 or has_seven(i):
        return not is_up
    return is_up
```

There's a tiny optimization we can make here. Instead of calculating an increment based on the value of `is_up`, we can make it directly store the direction of change into the variable (`next_dir` is also updated, see the solution for the new version):

```
def pingpong(n):
    i = 1
    x = 1
    step = 1
    while i < n:
        step = next_dir(step, i)
        x += step
        i += 1
    return x
```

This will work, but it uses assignment. To convert it to an equivalent recursive version without assignment, make each local variable into a parameter of a new helper function, and then add an appropriate base case. Lastly, we seed the helper function with appropriate starting values by calling it with the values we had in the iterative version.

You should be able to convince yourself that the version of `pingpong` in the solutions has the same logic as the iterative version of `pingpong` above.

Video walkthrough: https://youtu.be/74gwPjgrN_k (https://youtu.be/74gwPjgrN_k)

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
```

Q3: Taxicab Distance

An intersection in midtown Manhattan can be identified by an avenue and a street, which are both indexed by positive integers. The *Manhattan distance* or *taxicab distance* between two intersections is the number of blocks that must be traversed to reach one from the other, ignoring one-way street restrictions and construction. For example, Times Square (https://goo.gl/maps/LeXMb2vHuAB2) is on 46th Street and 7th Avenue. Ess-a-Bagel (https://goo.gl/maps/nM9ecFDD66D2) is on 51st Street and 3rd Avenue. The taxicab distance between them is 9 blocks (5 blocks from 46th to 51st street and 4 blocks from 7th avenue to 3rd avenue). Taxicabs cannot cut diagonally through buildings to reach their destination!

Implement `taxicab`, which computes the taxicab distance between two intersections using the following data abstraction. *Hint*: You don't need to know what a Cantor pairing function is; just use the abstraction.

```
def intersection(st, ave):
    """Represent an intersection using the Cantor pairing function."""
    return (st+ave)*(st+ave+1)//2 + ave

def street(inter):
    return w(inter) - avenue(inter)

def avenue(inter):
    return inter - (w(inter) ** 2 + w(inter)) // 2

w = lambda z: int(((8*z+1)**0.5-1)/2)

def taxicab(a, b):
    """Return the taxicab distance between two intersections.

    >>> times_square = intersection(46, 7)
    >>> ess_a_bagel = intersection(51, 3)
    >>> taxicab(times_square, ess_a_bagel)
    9
    >>> taxicab(ess_a_bagel, times_square)
    9
    """
    return abs(street(a)-street(b)) + abs(avenue(a)-avenue(b))
```

Use Ok to test your code:

```
python3 ok -q taxicab
```

The main focus of this problem is to get familiar with using data abstraction. With some previous problems involving abstract data types, it might have been possible to break the abstraction barrier and still solve the problem. This time around, the abstraction uses the Cantor pairing function to obfuscate the original data!

Through the power of abstraction however, you don't need to understand how the Cantor pairing function works. In truth, we could have also not told you anything about how the abstract data type was implemented. As long as you use the provided selectors, you should be able to solve the problem.

Speaking of which, the selectors give the `street` and `avenue` of an intersection. If we have the street and the avenue for each intersection, the taxicab distance is just the sum of the absolute difference of the two.

For more information, Wikipedia (https://en.wikipedia.org/wiki/Taxicab_geometry) has a useful visualization.

Video walkthrough: https://youtu.be/QueVasKQQBI (https://youtu.be/QueVasKQQBI)

Q4: Squares only (optional)

> **NOTE! This question is now optional, as it refers to material that will be covered in Wednesday's lecture.**

Implement the function `squares`, which takes in a list of positive integers. It returns a list that contains the square roots of the elements of the original list that are perfect squares. Try using a list comprehension.

> You may find the `round` function useful.
>
> ```
> >>> round(10.5)
> 10
> >>> round(10.51)
> 11
> ```

```
def squares(s):
    """Returns a new list containing square roots of the elements of the
    original list that are perfect squares.

    >>> seq = [8, 49, 8, 9, 2, 1, 100, 102]
    >>> squares(seq)
    [7, 3, 1, 10]
    >>> seq = [500, 30]
    >>> squares(seq)
    []
    """
    return [round(n ** 0.5) for n in s if n == round(n ** 0.5) ** 2]
```

Use Ok to test your code:

```
python3 ok -q squares
```

It might be helpful to construct a skeleton list comprehension to begin with:

```
[sqrt(x) for x in s if is_perfect_square(x)]
```

This is great, but it requires that we have an `is_perfect_square` function. How might we check if something is a perfect square?

- If the square root of a number is a whole number, then it is a perfect square. For example, `sqrt(61) = 7.81024...` (not a perfect square) and `sqrt(49) = 7` (perfect square).
- Once we obtain the square root of the number, we just need to check if something is a whole number. The `is_perfect_square` function might look like:

```
def is_perfect_square(x):
    return is_whole(sqrt(x))
```

- One last piece of the puzzle: to check if a number is whole, we just need to see if it has a decimal or not. The way we've chosen to do it in the solution is to compare the original number to the round version (thus removing all decimals), but a technique employing floor division (`//`) or something else entirely could work too.

We've written all these helper functions to solve this problem, but they are actually all very short. Therefore, we can just copy the body of each into the original list comprehension, arriving at the solution we finally present.

Video walkthrough: https://youtu.be/YwLFB9paET0 (https://youtu.be/YwLFB9paET0)

Q5: Count change

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 `amount`, a set of coins makes change for `amount` if the sum of the values of the coins is `amount`. 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 `amount` and returns the number of ways to make change for `amount` using these coins of the future.

> *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 an amount 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(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
    >>> from construct_check import check
    >>> check(HW_SOURCE_FILE, 'count_change', ['While', 'For'])
    True
    """
    def constrained_count(amount, smallest_coin):
        if amount == 0:
            return 1
        if smallest_coin > amount:
            return 0
        without_coin = constrained_count(amount, smallest_coin * 2)
        with_coin = constrained_count(amount - smallest_coin, smallest_coin)
        return without_coin + with_coin
    return constrained_count(amount, 1)
```

Use Ok to test your code:

```
python3 ok -q count_change
```

This is remarkably similar to the `count_partitions` problem, with a few minor differences:

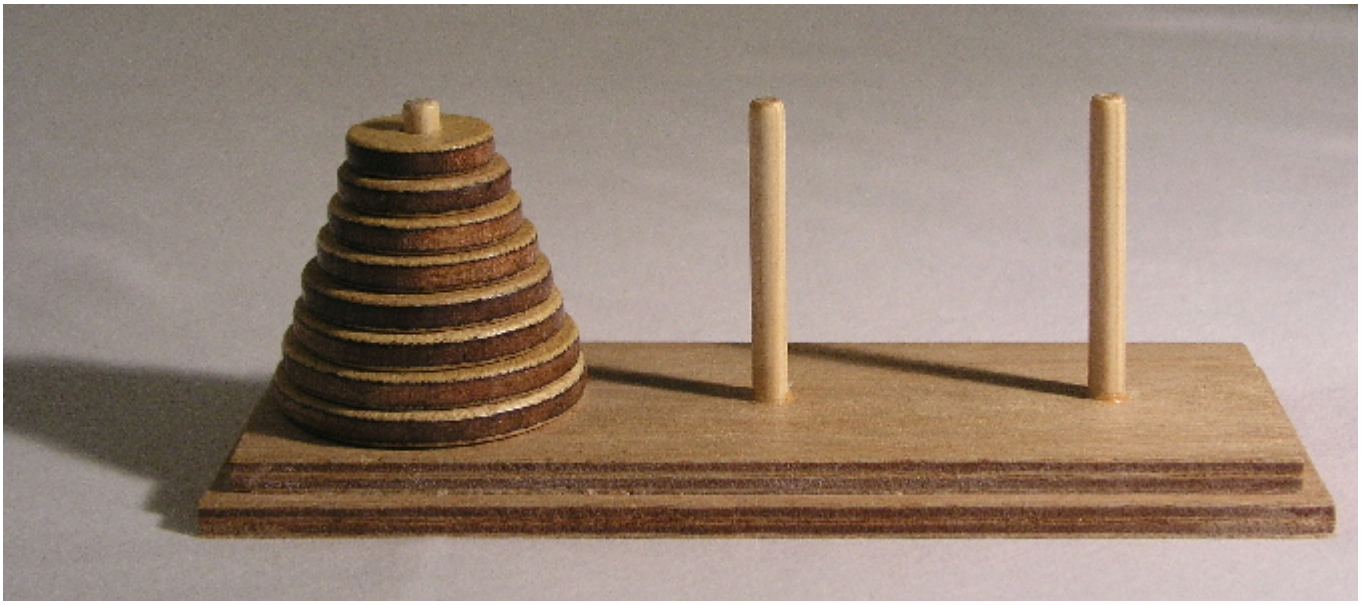
- A maximum partition size `m` is not given, so we need to create a helper function that takes in two arguments and also create another helper function to find the max coin.
- Partition size is not linear, but rather multiples of two. To get the next partition you need to divide by two instead of subtracting one.

One other implementation detail here is that we enforce a *maximum* partition size, rather than a *minimum* coin. Many students attempted to start at 1 and work there way up. That will also work, but is less similar to `count_partitions`. As long as there is some ordering on the coins being enforced, we ensure we cover all the combinations of coins without any duplicates.

See the walkthrough for a more thorough explanation and a visual of the recursive calls. Video walkthrough: https://youtu.be/EgZJPNFnoxM (https://youtu.be/EgZJPNFnoxM).

Q6: Towers of Hanoi

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.

```
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"
    if n == 1:
        print_move(start, end)
    else:
        other = 6 - start - end
        move_stack(n-1, start, other)
        print_move(start, end)
        move_stack(n-1, other, end)
```

Use Ok to test your code:

```
python3 ok -q move_stack
```

To solve the Towers of Hanoi problem for `n` disks, we need to do three steps:

1. Move everything but the last disk (`n-1` disks) to someplace in the middle (not the start nor the end rod).
2. Move the last disk (a single disk) to the end rod. This must occur after step 1 (we have to move everything above it away first)!
3. Move everything but the last disk (the disks from step 1) from the middle on top of the end rod.

We take advantage of the fact that the recursive function `move_stack` is guaranteed to move `n` disks from `start` to `end` while obeying the rules of Towers of Hanoi. The only thing that remains is to make sure that we have set up the playing board to make that possible.

Since we move a disk to end rod, we run the risk of `move_stack` doing an improper move (big disk on top of small disk). But since we're moving the biggest disk possible, nothing in the `n-1` disks above that is bigger. Therefore, even though we do not explicitly state the Towers of Hanoi constraints, we can still carry out the correct steps.

Video walkthrough: https://youtu.be/VwynGQiCTFM (https://youtu.be/VwynGQiCTFM)

Extra questions

> Extra questions are not worth extra credit and are entirely optional. They are designed to challenge you to think creatively!

Q7: Anonymous factorial

The recursive factorial function can be written as a single expression by using a conditional expression (http://docs.python.org/py3k/reference/expressions.html#conditional-expressions).

```
>>> 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
    >>> check(HW_SOURCE_FILE, 'make_anonymous_factorial', ['Assign', 'AugAssign', 'FunctionD
    True
    """
    return (lambda f: lambda k: f(f, k))(lambda f, k: k if k == 1 else mul(k, f(f, sub(k, 1)
    # Alternate solution:
    #   return (lambda f: f(f))(lambda f: lambda x: 1 if x == 0 else x * f(f)(x - 1))
```

Use Ok to test your code:

```
python3 ok -q make_anonymous_factorial
```

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)