Homework 3 Solutions hw03.zip (hw03.zip)

Solution Files

You can find the solutions in hw03.py (hw03.py).

Required Questions

Q1: 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`.

Important: Use recursion; the tests will fail if you use any assignment statements. (You can however use function definitions if you so wish.)

```
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', 'AnnAssign', 'AugAssign', 'NamedExpr'])
    True
    """
    if pos % 10 == 8:
        return 1 + num_eights(pos // 10)
    elif pos < 10:
        return 0
    else:
        return num_eights(pos // 10)
```

Use Ok to test your code:

```
python3 ok -q num_eights
```

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

```
total = 0
while pos > 0:
    if pos % 10 == 8:
        total = total + 1
    pos = pos // 10
return total
```

The main idea is that we check each digit for a eight. The recursive solution is similar, except that you depend on the recursive call to count the occurences of eight in the rest of the number. Then, you add that to the number of eights you see in the current digit.

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 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 are allowed to use function definitions.)

You may use the function `num_eights`, which you defined in the previous question.

Important: Use recursion; the tests will fail if you use any assignment statements. (You can however use function definitions if you so wish.)

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', 'AnnAssign', 'AugAssign', 'NamedExpr'])
    True
    """
    def helper(result, i, step):
        if i == n:
            return result
        elif i % 8 == 0 or num_eights(i) > 0:
            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 % 8 == 0 or num_eights(i) > 0:
        return -step
    return step

# Alternate solution 2
def pingpong_alt(n):
    if n <= 8:
        return n
    return direction(n) + pingpong_alt(n-1)

def direction(n):
    if n < 8:
        return 1
    if (n-1) % 8 == 0 or num_eights(n-1) > 0:
        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 eight 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 % 8 == 0 or num_eights(i) > 0:
        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:

YouTube link (https://youtu.be/74gwPjgrN_k)

Q3: Missing Digits

Write the recursive function `missing_digits` that takes a number `n` that is sorted in non-decreasing 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`.

Important: Use recursion; the tests will fail if you use any loops.

```
def missing_digits(n):
    """Given a number a that is in sorted, non-decreasing 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
    """
    if n < 10:
        return 0
    last, rest = n % 10, n // 10
    return max(last - rest % 10 - 1, 0) + missing_digits(rest)
# Alternate solution
def missing_digits_alt(n):
    def helper(n, digit):
        if n == 0:
            return 0
        last, rest = n % 10, n // 10
        if last == digit or last + 1 == digit:
            return helper(rest, last)
        return 1 + helper(n, digit - 1)
    return helper(n // 10, n % 10)
```

Use Ok to test your code:

```
python3 ok -q missing_digits
```

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

missing = 0 while n > 10: last, rest = n % 10, n // 10 missing += max(last - rest % 10 - 1, 0) n //= 10 return missing

A tricky case for this problem was handling adjacent numbers that are the same

- that's why we wrap the digit difference each recursive call with a max comparison call to 0.

Q4: 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.

You can use either of the functions given to you:

- `ascending_coin` will return the next larger coin denomination from the input, i.e. `ascending_coin(5)` is `10`.
- `descending_coin` will return the next smaller coin denomination from the input, i.e. `descending_coin(5)` is `1`.

There are two main ways in which you can approach this problem. One way uses `ascending_coin`, and another uses `descending_coin`.

Important: Use recursion; the tests will fail if you use loops.

> **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 ascending_coin(coin):
    """Returns the next ascending coin in order.
    >>> ascending_coin(1)
    5
    >>> ascending_coin(5)
    10
    >>> ascending_coin(10)
    25
    >>> ascending_coin(2) # Other values return None
    """
    if coin == 1:
        return 5
    elif coin == 5:
        return 10
    elif coin == 10:
        return 25

def descending_coin(coin):
    """Returns the next descending coin in order.
    >>> descending_coin(25)
    10
    >>> descending_coin(10)
    5
    >>> descending_coin(5)
    1
    >>> descending_coin(2) # Other values return None
    """
    if coin == 25:
        return 10
    elif coin == 10:
        return 5
    elif coin == 5:
        return 1

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
    >>> count_coins(200)
    1463
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(HW_SOURCE_FILE, 'count_coins', ['While', 'For'])
    True
    """
    def constrained_count(change, smallest_coin):
        if change == 0:
            return 1
        if change < 0:
            return 0
        if smallest_coin == None:
            return 0
        without_coin = constrained_count(change, ascending_coin(smallest_coin))
        with_coin = constrained_count(change - smallest_coin, smallest_coin)
        return without_coin + with_coin
    return constrained_count(change, 1)

    # Alternate solution: using descending_coin
    def constrained_count_desc(change, largest_coin):
        if change == 0:
            return 1
        if change < 0:
            return 0
        if largest_coin == None:
            return 0
        without_coin = constrained_count(change, descending_coin(largest_coin))
        with_coin = constrained_count(change - largest_coin, largest_coin)
        return without_coin + with_coin
    return constrained_count(change, 25)
```

Use Ok to test your code:

```
python3 ok -q count_coins
```

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

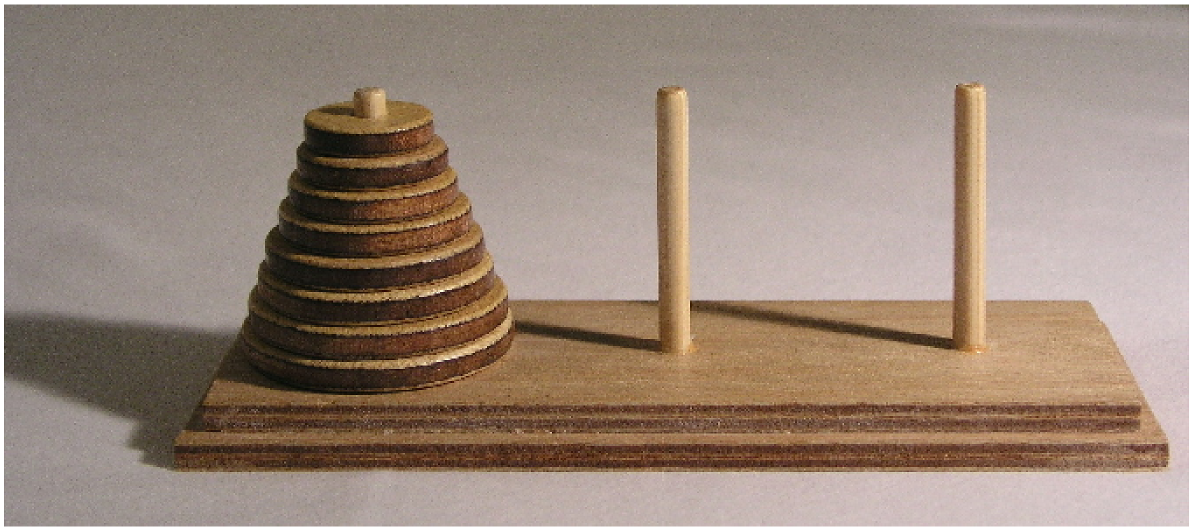
- A maximum partition size is not given, so we need to create a helper function that takes in two arguments and also create another helper function to find the max coin.
- Partition size is not linear. To get the next partition you need to call `ascending_coin` if you are counting up (i.e. from the smallest coin to the largest coin), or `descending_coin` if you are counting down.

Just for fun Questions

These questions are out of scope for 61a. You can try them if you want an extra challenge, but they're just puzzles that are not required or recommended at all. Almost all students will skip them, and that's fine.

Q5: 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:

YouTube link (https://youtu.be/VwynGQiCTFM)

Q6: Anonymous factorial

> This question demonstrates that it's possible to write recursive functions without assigning them a name in the global frame.

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:** 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', 'AnnAssign', 'AugAssign', 'NamedExpr', 'FunctionDef', 'Recursion'])
    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
```