Homework 3: Recursion, Tree Recursion

hw03.zip (hw03.zip)

Due by 11:59pm on Thursday, February 15

Instructions

Download hw03.zip (hw03.zip). Inside the archive, you will find a file called hw03.py (hw03.py), along with a copy of the `ok` autograder.

Submission: When you are done, submit the assignment by uploading all code files you've edited to Gradescope. 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 Gradescope. See Lab 0 (/~cs61a/sp24/lab/lab00#task-c-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/sp24/articles/using-ok)

Readings: You might find the following references useful:

- Section 1.7 (http://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. **This homework is out of 2 points.**

Required Questions

Getting Started Videos

Q1: Num Eights

Write a recursive function `num_eights` that takes a positive integer `n` and returns the number of times the digit 8 appears in `n`.

Important: Use recursion; the tests will fail if you use any assignment statements or loops. (You can, however, use function definitions if you'd like.)

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

    >>> num_eights(3)
    0
    >>> num_eights(8)
    1
    >>> num_eights(88888888)
    8
    >>> num_eights(2638)
    1
    >>> num_eights(86380)
    2
    >>> num_eights(12345)
    0
    >>> num_eights(8782089)
    3
    >>> from construct_check import check
    >>> # ban all assignment statements
    >>> check(HW_SOURCE_FILE, 'num_eights',
    ...       ['Assign', 'AnnAssign', 'AugAssign', 'NamedExpr', 'For', 'While'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q num_eights
```

Q2: Digit Distance

For a given integer, the *digit distance* is the sum of the absolute differences between consecutive digits. For example:

- The digit distance of `6` is `0`.
- The digit distance of `61` is `5`, as the absolute value of `6 - 1` is `5`.

- The digit distance of `71253` is `12` (`6 + 1 + 3 + 2`).

Write a function that determines the digit distance of a given positive integer. You must use recursion or the tests will fail.

> **Hint:** There are multiple valid ways of solving this problem! If you're stuck, try writing out an iterative solution first, and then convert your iterative solution into a recursive one.

```
def digit_distance(n):
    """Determines the digit distance of n.

    >>> digit_distance(3)
    0
    >>> digit_distance(777)
    0
    >>> digit_distance(314)
    5
    >>> digit_distance(31415926535)
    32
    >>> digit_distance(3464660003)
    16
    >>> from construct_check import check
    >>> # ban all loops
    >>> check(HW_SOURCE_FILE, 'digit_distance',
    ...       ['For', 'While'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q digit_distance
```

Q3: Interleaved Sum

Write a function `interleaved_sum`, which takes in a number `n` and two one-argument functions: `odd_func` and `even_func`. It applies `odd_func` to every odd number and `even_func` to every even number from 1 to `n` *including* `n` and returns the sum.

For example, executing `interleaved_sum(5, lambda x: x, lambda x: x * x)` returns `1 + 2*2 + 3 + 4*4 + 5 = 29`.

Implement this function without using any loops or directly testing if a number is odd or even -- no modulos (`%`) allowed! Instead of checking whether a number is even or odd, start with 1, which you know is an odd number.

> Hint: Introduce an inner helper function that takes an odd number `k` and computes an interleaved sum from `k` to `n` (including `n`).

```
def interleaved_sum(n, odd_func, even_func):
    """Compute the sum odd_func(1) + even_func(2) + odd_func(3) + ..., up
    to n.

    >>> identity = lambda x: x
    >>> square = lambda x: x * x
    >>> triple = lambda x: x * 3
    >>> interleaved_sum(5, identity, square) # 1   + 2*2 + 3   + 4*4 + 5
    29
    >>> interleaved_sum(5, square, identity) # 1*1 + 2   + 3*3 + 4   + 5*5
    41
    >>> interleaved_sum(4, triple, square)   # 1*3 + 2*2 + 3*3 + 4*4
    32
    >>> interleaved_sum(4, square, triple)   # 1*1 + 2*3 + 3*3 + 4*3
    28
    >>> from construct_check import check
    >>> check(HW_SOURCE_FILE, 'interleaved_sum', ['While', 'For', 'Mod']) # ban loops and
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q interleaved_sum
```

Q4: Count Coins

Given a positive integer `total`, a set of coins makes change for `total` if the sum of the values of the coins is `total`. 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 `total` and returns the number of ways to make change for `total` using coins.

You can use either of the functions given to you:

- `next_larger_coin` will return the next larger coin denomination from the input, i.e. `next_larger_coin(5)` is `10`.
- `next_smaller_coin` will return the next smaller coin denomination from the input, i.e. `next_smaller_coin(5)` is `1`.
- Either function will return `None` if the next coin value does not exist

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

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

> **Hint:** Refer the implementation (https://www.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 next_larger_coin(coin):
    """Returns the next larger coin in order.
    >>> next_larger_coin(1)
    5
    >>> next_larger_coin(5)
    10
    >>> next_larger_coin(10)
    25
    >>> next_larger_coin(2) # Other values return None
    """
    if coin == 1:
        return 5
    elif coin == 5:
        return 10
    elif coin == 10:
        return 25

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

def count_coins(total):
    """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
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q count_coins
```

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit

Submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 (https://cs61a.org/lab/lab00/#submit-with-gradescope) has detailed instructions.

In addition, all students who are **not** in the mega lab must complete this attendance form (https://go.cs61a.org/lab-att). Submit this form each week, whether you attend lab or missed it for a good reason. The attendance form is not required for mega section students.

Exam Practice

Homework assignments will also contain prior exam-level questions for you to take a look at. These questions have no submission component; feel free to attempt them if you'd like a challenge!

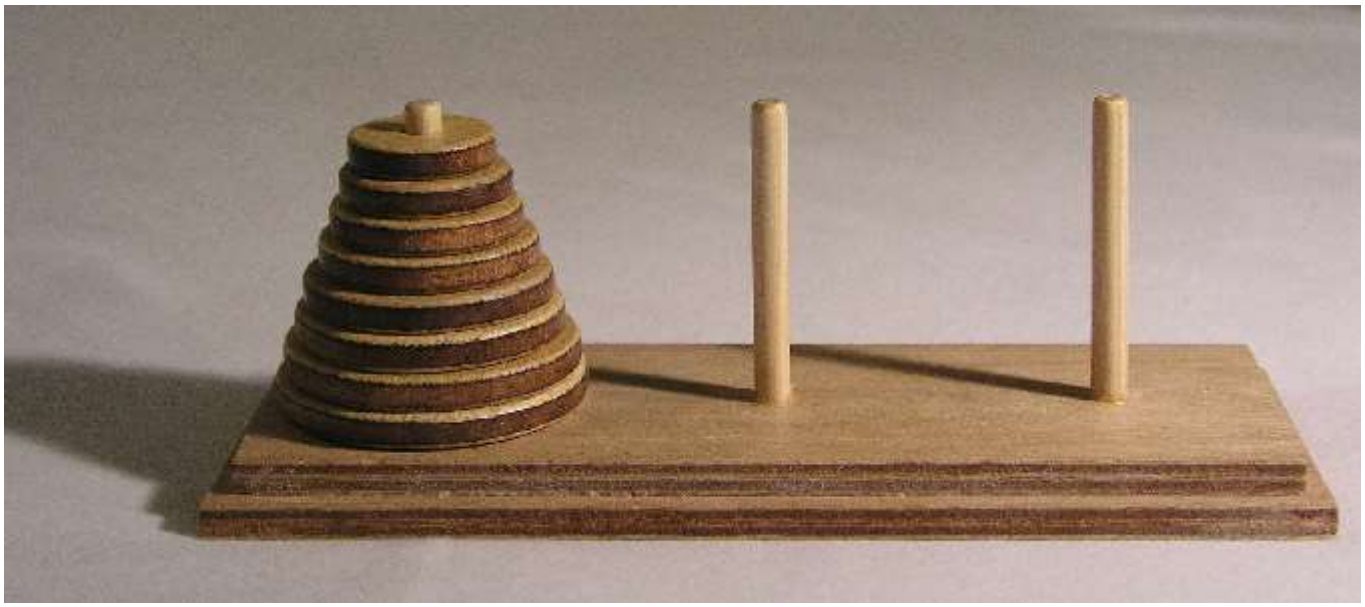
1. Fall 2017 MT1 Q4a: Digital (https://inst.eecs.berkeley.edu/~cs61a/fa21/exam/fa17/mt1/61a-fa17-mt1.pdf#page=5)
2. Summer 2018 MT1 Q5a: Won't You Be My Neighbor? (https://inst.eecs.berkeley.edu/~cs61a/su18/assets/pdfs/61a-su18-mt.pdf#page=5)
3. Fall 2019 Final Q6b: Palindromes (https://inst.eecs.berkeley.edu/~cs61a/sp21/exam/fa19/final/61a-fa19-final.pdf#page=6)

Just For Fun Questions

The questions below 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 for the course. Almost all students will skip them, and that's fine. We will **not** be prioritizing support for these questions on Ed or during Office Hours.

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.

Hint 1

Hint 2

Hint 3

```
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 ***"
```

Use Ok to test your code:

```
python3 ok -q move_stack
```

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 'YOUR_EXPRESSION_HERE'
```

Use Ok to test your code:

```
python3 ok -q make_anonymous_factorial
```