CSC148 Summer 2018: More Recursion Practice

This document contains additional recursion practice problems. Try to solve all of these problems recursively.

(The last few are more challenging/detailed, but are good preparation and practice.)

get_all_elements(x)

`x` is anything: either an object, or a list containing other lists or non-list objects. `get_all_elements` should return all of the non-list object in `x`.

For example:

- `get_all_elements(5)` should return `[5]`
- `get_all_elements([5, 15])` should return `[5, 15]`
- `get_all_elements([5, 15, [[25, 30], 45]])` should return `[5, 15, 25, 30, 45]`
- `get_all_elements([[5], 3, [1, [2], [3]]])` should return `[5, 3, 1, 2, 3]`

Below are some test cases:

```
assert get_all_elements(5) == [5]
assert get_all_elements([5, 15]) == [5, 15]
assert get_all_elements([5, 15, [[25, 30], 45]]) == [5, 15, 25, 30, 45]
assert get_all_elements([[5], 3, [1, [2], [3]]]) == [5, 3, 1, 2, 3]
assert get_all_elements([[1, 2], [3, [4], [[5]]], [6, [7, [8, 9]], 10]]) == [1,
2, 3, 4, 5, 6, 7, 8, 9, 10]
```

passes_condition(f, x)

`x` is either an int or a list containing either lists or ints, and `f` is a function that takes in a single int and returns a boolean. We want to return all elements in `x` that return True when passed into `f`. For example, if we define `f` as:

```
def f(x):
    return x > 15
```

With x being the list `[100, [[25, 3], 150, 10], [[30, [15, 20]], 5]]`, then we would want `[100, 25, 150, 30, 20]` to be returned. If x is an int that doesn't pass `f()`, then `passes_condition()` should return an empty list. Below are some test cases:

```
def f(x):
    return x > 15

assert passes_condition(f, 5) == []
assert passes_condition(f, 25) == [25]
assert passes_condition(f, [15, 25, 30]) == [25, 30]
assert passes_condition(f, [[100, 3], [[15, 25, 30]], 45]) == [100, 25, 30, 45]
assert passes_condition(f, [100, [[25, 3], 150, 10], [[30, [15, 20]], 5]]) ==
[100, 25, 150, 30, 20]
```

count_number_of_lists(x)

`x` is either an int or a list containing either lists or ints. `count_number_of_lists` should return the total number of lists (and nested sublists) in `x`.

For example:

- `count_number_of_lists(5)` should return 0
- `count_number_of_lists([5])` should return 1
- `count_number_of_lists([[5], 3, [1, [2], [3]]])` should return 5

Below are some test cases:

```
assert count_number_of_lists(5) == 0
assert count_number_of_lists([5]) == 1
assert count_number_of_lists([[5], [3], [1]]) == 4
assert count_number_of_lists([[5], 3, [1, [2], [3]]]) == 5
```

all_sections_are_words(word, word_list)

In this function, we want to make sure `word` is in `word_list`, as well as all halves (and halves of those halves and so on), hereby referred to as "sections' of the word.

For example, suppose we have the word `'something'`. The first half of this is `'some'`, and the other half is `'thing'` (i.e. `word[:len(word)//2]` and `word[len(word)//2:]`). For the sake of this problem, the smallest length a section can be is 2 letters (i.e. we'll recurse until the length is <= 3). Then all of the sections (and sections of those sections) would be:

- `something`
  - `some`
    - `so`
    - `me`
  - `thing`
    - `th`
    - `ing`

We return `True` if all of these sections appear in `word_list`. i.e. if `word_list` was `['me', 'so', 'something', 'thing']` then we'd return `False` since `'th'` and `'ing'` aren't in `word_list`. Similarly, if `word_list` was `['me', 'ng', 'so', 'something', 'thi', 'thing']`, then we'd return `True`. Below are some test cases:

```
assert all_sections_are_words("today", ['to', 'today']) == False
assert all_sections_are_words("today", ['to', 'day', 'today']) == True
assert all_sections_are_words("sometome", ['so', 'me', 'to']) == False
assert all_sections_are_words("sometome", ['so', 'me', 'to', 'some']) == False
assert all_sections_are_words("sometome", ['so', 'me', 'to', 'some', 'tome'])
== False
```

```
assert all_sections_are_words("sometome", ['so', 'me', 'to', 'some', 'tome',
'sometome']) == True
assert all_sections_are_words("sometome", ['so', 'me', 'some', 'tome',
'sometome']) == False
assert all_sections_are_words("sometome", ['so', 'me', 'to', 'tome',
'sometome']) == False
```

get_nth_fibonacci(n)

The Fibonacci numbers are an integer sequence that follow the fulfill:

$$F_1 = 1$$

$$F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

The first 8 Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, and 21. Write a function, `get_nth_fibonacci`, that returns the nth Fibonacci number (where n > 0). Below are some test cases:

```
assert get_nth_fibonacci(1) == 1
assert get_nth_fibonacci(2) == 1
assert get_nth_fibonacci(3) == 2
assert get_nth_fibonacci(10) == 55
assert get_nth_fibonacci(15) == 610
```

x_in_list(x, lst)

`x` is an int and `lst` is a list of ints. **Without using any loops or 'in':** return whether x is in lst or not.

Hint: if `lst` is `[5]` and `x` is `5`, then `lst[0] == x` is `True`. This would be your base case.

You'll want to recurse on smaller and smaller lists. Below are some test cases:

```
assert x_in_list(2, [-1, 0, 1]) == False
assert x_in_list(1, [-5, 3, 1]) == True
assert x_in_list(10, [1, -2, 10, -1, 15, 20]) == True
assert x_in_list(15, [-1, 0, -4, 15, 3, 1]) == True
assert x_in_list(20, [-1, 0, -4, 15, 3, 1]) == False
```

x_in_sorted_list(x, lst)

`x` is an int and `lst` is a sorted list of ints. **Without using any loops or 'in':** return whether x is in lst or not.

Try to make as few recursive calls as possible. Take advantage of the fact that lst is sorted.

Hint: Split `lst` in half with each recursive call you make. You'll only have to recurse on one half. Below are some test cases:

```
assert x_in_sorted_list(2, [-1, 0, 1]) == False
assert x_in_sorted_list(1, [-5, 0, 1]) == True
assert x_in_sorted_list(10, [1, 10, 11, 15, 20]) == True
assert x_in_sorted_list(15, [1, 2, 4, 10, 15, 18]) == True
assert x_in_sorted_list(20, [1, 5, 7, 8, 10, 20]) == True
assert x_in_sorted_list(25, [1, 5, 7, 8, 10, 20]) == False
```

binary_search(x, lst)

In Binary Search, we look for the index of an item in a sorted list (returning -1 if it's not in the list). To do this, we look in halves of lst, so we'd only have to look at a very small number of elements each time (mathematically, we look at only $\log_2(n)$ elements where `n` is the size of the list).

For example, suppose we have the following list:

[1, 4, 5, 7, 8, **10**, 13, 15, 19, 22]

And we're trying to find the index of the number 7. Then we know the middle of the list is the number 10. That means we can ignore everything from 10 and onwards, since 7 has to be to the left of 10.

[1, 4, **5**, 7, 8, 10, 13, 15, 19, 22]

From the remaining 'unsearched' part of the list, we know 5 is in the middle. Since 7 is larger than 5, we should be searching to the right of 5.

[1, 4, 5, 7, **8**, 10, 13, 15, 19, 22]

Afterwards, our midpoint is 8 (assuming we do len(lst) // 2, since the length of the unsearched part is 2, then 2 // 2 == 1; so lst[1] == 8). 7 is less than 8, so we go to the left of 8 and search for 7 there.

[1, 4, 5, **7**, 8, 10, 13, 15, 19, 22]

7 is the value at this point, so we would want to return that index.

- If lst == [7], then that means we return 0.
- If lst == [7, 8], then we'd still want to return 0.
- If lst == [1, 4, 5, 7, 8], then we'd want to return 3.
- And then for our overall list, [1, 4, 5, 7, 8, 10, 13, 15, 19, 22], we'd want to return 3.

If your x_in_sorted_list split in halves each time, then implementing this should just be a matter of finding out how to add the index together from each of your recursive calls. Below are some test cases:

```
assert binary_search(2, [-1, 0, 1]) == -1
assert binary_search(1, [-5, 0, 1]) == 2
assert binary_search(10, [1, 10, 11, 15, 20]) == 1
assert binary_search(15, [1, 2, 4, 10, 15, 18]) == 4
assert binary_search(20, [1, 5, 7, 8, 10, 20]) == 5
assert binary_search(25, [1, 5, 7, 8, 10, 20]) == -1
```

flatten_dictionary(d) Credit to Jason Yang for bringing me this problem :)

`d` is a dictionary whose keys are all strings. The values can be anything, including dictionaries. `flatten_dictionary` should return a dictionary with non-dictionary values.

For example:

- If `d = {'a': 1, 'b': 2, 'c': {'d': 3, 'e': 4}}` then `flatten_dictionary(d)` should return the dictionary `{'a': 1, 'b': 2, 'c.d': 3, 'c.e': 4}`.
- If `d = {'a': 1, 'b': 2, 'c': {'d': 3, 'e': 4, 'f': {'g': 5}}}` then `flatten_dictionary(d)` should return the dictionary `{'a': 1, 'b': 2, 'c.d': 3, 'c.e': 4, 'c.f.g': 5}`.

Note: This problem doesn't follow the standard
    [Base case]
    [Recursive step]
format that we've used so far, but it's doable with your current knowledge. You should still follow the steps of:

1. Find out the recursive step. What recursive calls would you want to make?
2. What would you get back from those recursive calls?
3. How would you use the results from that recursive call to add to your new dictionary that you're returning?

Focus more on the recursive step and how to build a new dictionary.

get_nth_fibonacci_memoized(n, fibonaccis = {})

If we try to call `get_nth_fibonacci(50)`, it would take a very, very long time to get the results. The calls needed would be:

- `get_nth_fibonacci(49)`
  - Which would call `get_nth_fibonacci(48)`
    - Which would call `get_nth_fibonacci(47)`
      - Which would call `get_nth_fibonacci(46)`
      - ... and `get_nth_fibonacci(45)`
    - ... and `get_nth_fibonacci(46)`
      - Which would call `get_nth_fibonacci(45)`
      - ... and `get_nth_fibonacci(44)`
  - ... and `get_nth_fibonacci(47)`
    - Which would call `get_nth_fibonacci(46)`
      - And so on...
    - ... and `get_nth_fibonacci(45)`
      - And so on...
- `get_nth_fibonacci(48)`

    - Which would call `get_nth_fibonacci(47)`
        - Which would call `get_nth_fibonacci(46)`
            - Which would call `get_nth_fibonacci(45)`
            - ... and `get_nth_fibonacci(44)`
        - ... and `get_nth_fibonacci(45)`
            - Which would call `get_nth_fibonacci(44)`
            - ... and `get_nth_fibonacci(43)`
    - ... and `get_nth_fibonacci(46)`
        - Which would call `get_nth_fibonacci(45)`
            - And so on...
        - ... and `get_nth_fibonacci(44)`
            - And so on...

As you can see, there are repeated recursive calls on the same thing. Thus, suppose we want to keep a dictionary of results that gets passed along based on what we've calculated already. Call this dictionary `fibonaccis`, and if no dictionary is passed in, it should take the value of an empty dictionary by default.

Write an alternate version of `get_nth_fibonacci` called `get_nth_fibonacci_memoized` which adds the newly calculate Fibonacci number to the dictionary, and passes this dictionary along in each recursive call. If we've already calculated a Fibonacci number n, then we can just return that value instead of making additional recursive calls.

Calling get_nth_fibonacci_memoized(50) should return the number `12586269025` almost instantly. Below are some test cases:

```
assert get_nth_fibonacci_memoized(1) == 1
assert get_nth_fibonacci_memoized(2) == 1
assert get_nth_fibonacci_memoized(3) == 2
assert get_nth_fibonacci_memoized(10) == 55
assert get_nth_fibonacci_memoized(50) == 12586269025
assert get_nth_fibonacci_memoized(100) == 354224848179261915075
```

As an aside: This idea of remembering what we've gotten from previous recursive calls already is called memoization. We end up using additional memory in order to save on computation time, and for the case of Fibonacci numbers, we save a *lot* of time.