CSC148 - Thinking about Recursion

Students often have a hard time wrapping their minds around recursion. It is a challenging topic for two reasons: it is conceptually different from the iterative (looping) style of code we have used so far, and the usual technique of tracing each function call is much, much more time-consuming and error-prone for recursive functions. The goal of this worksheet is to give you a different way of *reasoning* about recursive code, so that when you're stuck debugging your recursive functions, you have a better approach than "brute-force tracing."

```
def nested_sum(obj):
    """Return the sum of the items in a nested list.

    @type obj: int | list
    @rtype: int
    """
    if isinstance(obj, int):
        return obj
    else:
        s = 0
        for lst_i in obj:
            s = s + nested_sum(lst_i)
        return s
```

---

1. Carefully trace through the function call `nested_sum(10)`. Which line(s) of code execute, and what is the output?

2. Do the same for `nested_sum(100)`. After this, you should be relatively convinced that `nested_sum` works properly on integers, aka "nested lists of depth 0".

3. Now consider the function call for a nested list of depth 1, `nested_sum([15, 105, 13])`. The loop executes, making one recursive call for each "sub-nested list" of `15, 105, 13`. Fill in the following table, which traces through the three different iterations of the loop. We have started it for you.

   Do not trace into the recursive calls! Remember that you are now convinced that `nested_sum` works correctly on integers, so you can predict what the return values are without doing any tracing.

Value of `lst_i`	Return value of `nested_sum(lst_i)`	Value of `s` at the *end* of the iteration
`15`		

4. So what does `nested_sum([15, 105, 13])` return? Confirm that this is what you expect it to return, based on the docstring of `nested_sum`.

5. Now assume that `nested_sum` works properly on nested lists of depth 0 or 1. Consider the call

```
>>> nested_sum([[15, 105, 13], [1, 2, 3], 4, [1]])
```

Fill in the following table, and then state what the function returns.

Value of `lst_i`	Return value of `nested_sum(lst_i)`	Value of `s` at the *end* of the iteration

You can try out a few more examples to ensure that `nested_sum` works on all lists of depth two. After that, you could move on to depth 3, and then depth 4, etc. The key idea is this: when considering a call to `nested_sum(obj)`, where `obj` is a nested list of depth d, you should *already* have convinced yourself of the correctness of `nested_sum` on nested lists of depth $< d$. So when you trace the call, you don't actually need to trace into each recursive call (wasting valuable time and mental energy), but instead predict its output based on the docstring of `nested_sum`.

Here is an exercise to see whether you get the point.

```
def nested_sum2(obj):
    """Return the sum of the items in <obj>, times 2.

    I.e., return the value of nested_sum(obj) * 2.
    """
    if isinstance(obj, int):
        # HIDDEN
    else:
        s = 0
        for lst_i in obj:
            s = s + nested_sum2(lst_i)
        return s
```

---

6. Consider the function call `nested_sum2([[2, [3, 1]], 4, [[1]], [10, 20]])`.

Assuming that `nested_sum2` works on lists of depth < 3, fill in the table below to verify that the recursive part of this function is correct. You should not need to trace any recursive calls.

Value of `lst_i`	Return value of `nested_sum2(lst_i)`	Value of `s` at the *end* of the iteration