CSC148 - Thinking about Recursion

With recursive code, the usual technique of tracing each function call is much, much more time-consuming and error-prone. The goal of this worksheet is to give you a different way of reasoning about recursive code, so that when you’re debugging recursive functions you have a better approach than “brute-force tracing.”

```python
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. Trace the function call `nested_sum(10)`. Which line(s) of code execute, and what value is returned? Is this the correct return value, according to the docstring?

2. Now trace the call `nested_sum(100)`. What value is returned and is it correct?

3. Are you convinced that `nested_sum` works properly on any integer, aka any nested list of depth 0? If not, trace more examples.

4. Now consider a call with 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. Complete the table that we started, which traces through the three iterations of the loop.

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

<table>
<thead>
<tr>
<th>Value of <code>lst_i</code></th>
<th>Return value of <code>nested_sum(lst_i)</code></th>
<th>Value of <code>s</code> at the end of the iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. So what does `nested_sum([15, 105, 13])` return? Is it correct, according to the docstring?
6. Are you convinced that `nested_sum` works properly on any nested list of depth 1? If not, trace more examples.

7. Now assume that `nested_sum` works properly on nested lists of depth 0 or 1, and consider this call with a nested list of depth 2:

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

Fill in the following table with a row for each iteration of the loop:

<table>
<thead>
<tr>
<th>Value of <code>lst_i</code></th>
<th>Return value of <code>nested_sum(lst_i)</code></th>
<th>Value of <code>s</code> at the end of the iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8. What does `nested_sum([[15, 105, 13], [1, 2, 3], 4, [1]])` return? Is it correct?

9. Are you convinced that `nested_sum` works properly on any nested list of depth 2? If not, trace more examples.

10. You could move on to lists of depth 3, and then depth 4, etc. But by now, you may be convinced that the pattern will hold, and that the function works on lists of any depth.

    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 the return value of the recursive call based on the docstring of `nested_sum`. 