CSC148 - Tracing and Debugging Recursive Functions

On this worksheet, you'll practice the *partial tracing* technique for recursive functions covered in this week's prep.

1. Here is a partial implementation of a nested list function that returns a brand-new list.

```
def flatten(obj: Union[int, List]) -> List[int]:
    """Return a (non-nested) list of the integers in <obj>.

    The integers are returned in the left-to-right order they appear
    in <obj>.

    >>> flatten(6)
    [6]
    >>> flatten([1, [-2, 3], -4])
    [1, -2, 3, -4]
    >>> flatten([[0, -1], -2, [[-3, [-5]]]])
    [0, -1, -2, -3, -5]
    """
    if isinstance(obj, int):
        # Base case omitted  return [obj]
    else:
        s = []
        for sublist in obj:
            s.extend(flatten(sublist))
        return s
```

Our goal is to determine whether the recursive step is correct *without* fully tracing (or running) this code.

Consider the function call `flatten([[0, -1], -2, [[-3, [-5], -7]]])`.

(a) What *should* `flatten([[0, -1], -2, [[-3, [-5], -7]]])` return, according to its docstring?

[0, –1, –2, –3, –5, –7]

(b) We'll use the table below to partially trace the call `flatten([[0, -1], -2, [[-3, [-5], -7]]])`. Complete the **first two columns** of this table, assuming that `flatten` works properly on each recursive call. Remember that filling out these two columns can be done *just* using the argument value and `flatten`'s docstring; you don't need to worry about the code at all!

Note: the input list `[[0, -1], -2, [[-3, [-5], -7]]]` has just *three* sub-nested-lists.

`sublist`	`flatten(sublist)`	Value of `s` at the *end* of the iteration
N/A	N/A	`[]` (initial value of `s`)
[0, –1]	[0, –1]	[0, –1]
–2	[–2]	[0, –1, –2]
[[–3, [–5], –7]]	[–3, –5, –7]	[0, –1, –2, –3, –5, –7]

(c) Use the third column of the table to complete the partial trace of the recursive code. Remember that every time you reach a recursive call, don't trace into it—use the value you calculated in the second column!

(d) Compare the final value of `s` with the expected return value of `flatten`. Does this match?

(e) Finally, write down an implementation of the *base case* of `flatten` directly on the code above.

2. Now consider the following function and partial implementation.

```
def uniques(obj: Union[int, List]) -> List[int]:
    """Return a (non-nested) list of the integers in <obj>, with no duplicates.

    >>> uniques([13, [2, 13], 4])
    [13, 2, 4]
    """
    if isinstance(obj, int):
        # Base case omitted
    else:
        s = []
        for sublist in obj:
            s.extend(uniques(sublist))    s.extend([item for item in unique(sublist) if item not in s])
        return s
```

It turns out that there is a problem with the recursive step in this function, and it has the insidious feature of being *sometimes correct, and sometimes incorrect.* To make sure you understand this, find *two* examples for initial arguments to `uniques`: one in which partial tracing would lead to us thinking there's no error, and one in which partial tracing would lead us to find an error.

Input that does NOT reveal an error:
[1, 2, 3]

Expected output: [1, 2, 3]   [1, [2, 2, [3, 4]]]

Partial trace table (fill it in and verify that the bottom-right corner matches the expected output; you might not need to use all the rows, depending on your chosen input)

`sublist`	`uniques(sublist)`	Value of `s` at the *end* of the iteration
N/A	N/A	[]
1	[1]	[1]
2	[2]	[1, 2]
3	[3]	[1, 2, 3]

Input that DOES reveal an error: [1, 2, [2, 2, [3, 4]]]

Expected output:

Partial trace table (fill it in and verify that the bottom-right corner *doesn't* match the expected output; you might not need to use all the rows, depending on your chosen input)

`sublist`	`uniques(sublist)`	Value of `s` at the *end* of the iteration
N/A	N/A	[]
1	[1]	[1]
1	[1]	[1, 1]
1	[1]	[1, 1, 1]

3. What you've provided above is a *counter-example* that shows that this recursive step is incorrect. This is a good start, but we'd like to go deeper. Analyse the recursive code above, and then describe in words *why* the code is incorrect, i.e., what the problem with the code is.