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.

```python
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
    else:
        s = []
        for sublist in obj:
            s.extend(flatten(sublist))
        return s
```

Our goal is to determine whether this 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?

(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`)

(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`. Do they 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.

```python
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))
        return s
```

It turns out that there is a problem with this recursive step, and it has the insidious feature of being *sometimes correct, and sometimes incorrect*. To make sure you understand this, find *two* example inputs: 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 looks CORRECT:

Expected output:

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	`[]`

Input that looks INCORRECT:

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	`[]`

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. Share your answer with your group, and collectively try to get the best wording you can!