CSC148 - Another Mutating Version of Quicksort

Last time, we learned how to implement an in-place version of `_partition` that mutated `lst` directly, and that helped us design an in-place version of quicksort.

We're going to do that again! We'll jump right to the version with `start` and `end` parameters, which we'll need for the quicksort itself.

```
def _in_place_partition(lst: List[int], start: int, end: int) -> int:
    """Mutate <lst[start:end]> so that it is partitioned with pivot lst[start].

    Let pivot = lst[start]. The elements of <lst> are moved around so that lst looks like

        [x1, x2, ... x_m, pivot, y1, y2, ... y_n],

    where each of the x's is <= pivot, and each of the y's is > pivot.

    The *new index of the pivot* is returned.

    >>> lst = [10, 3, 20, 5, -6, 30, 7]
    >>> _in_place_partition(lst, 0, 7)  # Pivot is 10
    >>> lst[4]  # The 10 is now at index 4
    10
    >>> set(lst[:4]) == {3, 5, -6, 7}
    True
    >>> set(lst[5:]) == {20, 30}
    True
    """
```

1. Again, we're going to divide up `lst` into three parts using indexes `small_i` and `big_i`. The way we're doing this is different than last time, though:

   - `lst[start+1:small_i]` contains the elements that are known to be `<= lst[start]` (the "smaller partition")
   - `lst[small_i:big_i]` contains the elements that are known to be `> lst[start]` (the "bigger partition")
   - `lst[big_i:end]` contains the elements that have not yet been compared to the pivot (the "unsorted section")

   `_in_place_partition` uses a loop to go through the elements of the list and compare each one to the pivot, building up either of the "smaller" or "larger partitions" and shrinking the "unsorted section." Next, we'll investigate how to translate this idea into code.

   (a) When we first begin the algorithm, we know that `lst[start]` is the pivot, but have not yet compared it against any other list element. What should the initial values of `small_i` and `big_i` be?

   (b) We need to check the items one at a time, until every item has been compared against the pivot. How do we know that we've finished checking every item?

   (c) How should the value of `small_i` or `big_i` change on each loop iteration, based on the current item that we're checking?

2. Next, implement `_in_place_partition` in the space below. Make sure you're confident in your answers to the questions on the previous page before writing the main loop!

```
def _in_place_partition(lst: List[int], start: int, end: int) -> int:
    # Initialize variables
    pivot = lst[start]

    small_i =

    big_i =


    # The main loop









    # At this point, all elements have been compared.
    # Now, move the pivot into its correct position in the list
    # (after the "smaller" partition, but before the "bigger" partition).


    # Return the new index of the pivot
```