CSC148 - A Mutating Version of Quicksort

On this worksheet, we're going to study how to implement an *in-place* version of `_partition` that simply mutates `lst` directly; this is the key ingredient to implementing an in-place version quicksort.

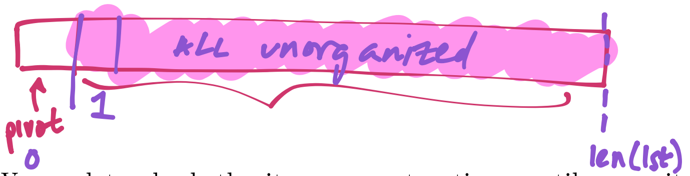
```python
def _in_place_partition(lst: List[int]) -> None:
    """Mutate <lst> so that it is partitioned with pivot lst[0].

    Let pivot = lst[0]. 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.

    >>> lst = [10, 3, 20, 5, -6, 30, 7]
    >>> _in_place_partition(lst)  # 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. The key idea to implement this function is to divide up `lst` into three parts using indexes `small_i` and `big_i`:

   - `lst[1:small_i]` contains the elements that are known to be `<=` `lst[0]` (the "smaller partition")
   - `lst[big_i:]` contains the elements that are known to be `>` `lst[0]` (the "bigger partition")
   - `lst[small_i:big_i]` contains the elements that have not yet been compared to the pivot (the "unsorted section")

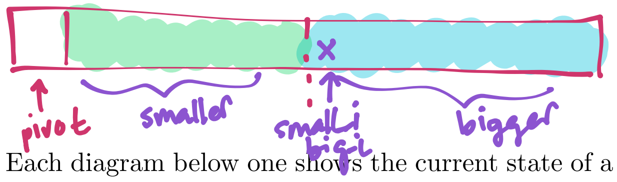
   `_in_place_partition` uses a loop to go through the elements of the list and compare each one to the pivot, building up either of the "smaller" or "larger partitions" and shrinking the "unsorted section." Next, we'll investigate how to translate this idea into code.

   (a) When we first begin the algorithm, we know that `lst[0]` 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?



small_i = 1
big_i = len(lst)

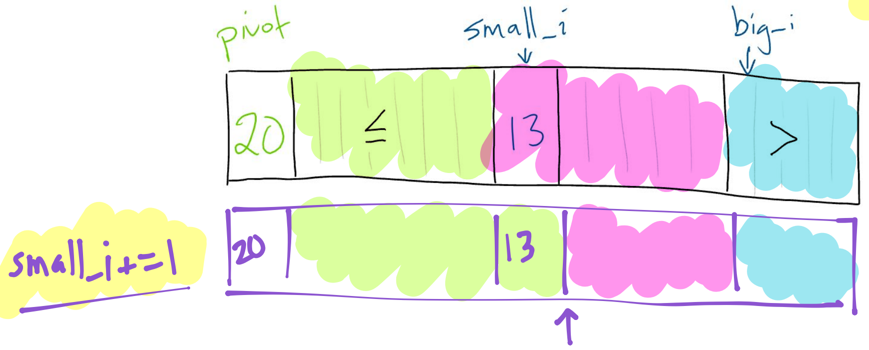
   (b) We need to check the items one at a time, until every item has been compared against the pivot. What is the relationship between `small_i` and `big_i` when we have finished checking every item?

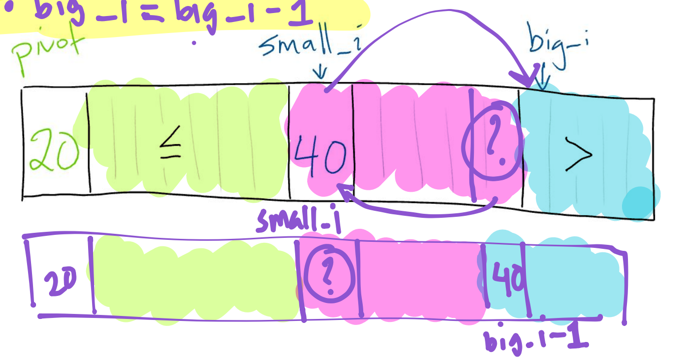


→ small_i == big_i
so we keep looping
↳ while small_i < big_i

   (c) Each diagram below one shows the current state of a list during this loop. The next element that is compared against the pivot is `lst[small_i]`. Show how the list sections should change after this item is compared. How should the value of `small_i` or `big_i` change?

• swap lst[small_i] with lst[big_i - 1]
• big_i = big_i - 1





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]) -> None:
    # Initialize variables
    pivot = lst[0]

    small_i = 1

    big_i = len(lst)

    # The main loop

    while small_i < big_i :
        if lst[small_i] <= pivot:
            small_i += 1

        else:
            lst[small_i], lst[big_i-1] =
                lst[big_i - 1] , lst[small_i]
            big_i -=1
    # At this point, all elements have been compared.
    # The final step is to move the pivot into its correct position in the list
    # (after the "smaller" partition, but before the "bigger" partition).

    lst[0], lst[small_i-1] =
        lst[small_i], lst[0]
```

3. Finally, we're going to need to make this helper a bit more general in order to use it with a modified quicksort algorithm. (Maybe on a separate sheet of paper, or your computer.)

- Modify your implementation so that it returns the *new index of the pivot* in the list (this is so that when _in_place_partition is called in quicksort, we know where the partitions start and end).
- Modify your implementation so that it takes in two additional parameters start and end, so that the function only partitions the range lst[start:end] rather than the entire list.