CMPS 2200 Recitation 05

Name (Team Member 1):_____
Name (Team Member 2):_____

In this lab, we'll continue working with sequence functions and sorting.

The sorting algorithms we've discussed so far all work by *comparing* numbers (e.g., `merge_sort`, `insertion_sort`, `selection_sort`). Today, we'll look at an algorithm that sorts without making any pairwise comparisons.

The algorithm is particularly suited for sorting lists with the following properties: - elements are non-negative integers - the maximum element is not too big - many elements are repeated

For example:

```
[2,2,1,0,1,0,1,3] -> [0,0,1,1,1,2,2,3]
```

In addition to the input list of length n, the algorithm also takes as input the maximum value in the list (k). E.g., k=*3* in the above example.

The algorithm proceeds by first counting how often each input value appears. Based on these counts, the algorithm figures out the range of output locations to place each value. For example, because there are two 0s and three 1s, we know that the first 1 goes in the 3rd position, and the final 1 goes in the 5th position. Finally, it uses this information to construct the final sorted list. In more detail, the three steps are:

- **Step 1:** the `count_values` function counts how often each value appears.
  - `count_values([2,2,1,0,1,0,1,3], 3) -> [2,3,2,1]`
- **Step 2:** the `get_positions` function determines where the first appearance of each value should be in the final, sorted output.
  - `get_positions([2,3,2,1]) -> [0, 2, 5, 7]` In this example, the first 0 should appear at index 0, the first 1 at index 2, the first 2 at index 5, and the first 3 at index 7.
- **Step 3:** the `construct_output` function combines the original input with the positions from step 2 to construct the final output. It does so by looping through the original input, finding the proper location for each value, and updating the counts array as values are added.
  - `construct_output([2,2,1,0,1,0,1,3], [0, 2, 5, 7]) -> [0,0,1,1,1,2,2,3]`

Here is a partial trace of step 3:

```
a=[2,2,1,0,1,0,1,3]
output=[-,-,-,-,-,-,-,-] counts=[0, 2, 5, 7]
```

- insert 2

```
output=[-,-,-,-,-,2,-,-] counts=[0, 2,6, 7]
```

- insert 2

```
output=[-,-,-,-,-,2,2,-] counts=[0, 2,7, 7]
```

- insert 1

```
output=[-,-,1,-,-,2,2,-] counts=[0,3, 7, 7]
```

- insert 0

```
output=[0,-,1,-,-,2,2,-] counts=[1, 3, 7, 7]
```

...

Below, we'll complete the algorithm `supersort` by implementing three functions, `count_values`, `get_positions`, and `construct_output`.

1. Implement a simple, sequential version of `count_values` (linear span and work) and test it with `test_count_values`.

.
.
.

2. Continuing our example, `counts` should now be `[2, 3, 2, 1]`. We next need to convert this into a list indicating the location of the first appearance of each value in the output.

E.g., `positions=[0, 2, 5, 7]` means that, in the final output, the first 0 appears at index 0, the first 1 appears at index 2, etc.

We can use `scan` to create the needed list. You may need to adjust slightly the output of scan to get the needed list. Complete `get_positions` and test with `test_get_positions`.

.
.
.

3. What is the work and span of `get_positions`? (assume our more efficient version of `scan` from class)

   put in answers.md

.
.
.

4. Finally, we'll use this positions array to construct the final output. First, we'll create the output list (n elements). Then, we will loop through the original input array once again. For each value, we'll look up in the `positions` array where the value should go. E.g., for the first value 2, we look up `positions[2]`, which tells us the 2 should go in index 5 in the output. To update `counts` for future iterations, we will then increment `counts` by one for the value we just read. E.g., `positions[2]` will increment from 5 to 6; the next 2 we read will be placed in index 6.

Implement `construct_output` with a simple for loop and test with `test_construct_output`.

.
.
.

5. What is the work and span of `construct_output`?

   put in answers.md

.
.
.

6. What is the work and span of `supersort`?

   put in answers.md

.
.
.

7. Our implementation of `count_values` has poor span. Let's instead implement it using map-reduce. Complete `count_map`, `count_reduce`, which are used by `count_values_mr` to construct the `counts` variable using map-reduce. Test with `test_count_values_mr`.

.
.
.

8. What is work and span of `count_values_mr`?

put in answers.md .
.
.