Lab 11 "Solution"

This page is meant to be *like* a solution to the lab, but more focused on explaining *how* to do the lab as opposed to providing a complete solution. Code in this section will be minimal, but the intent is for the explanations to provide you a starting point in solving the lab.

Task 1: Finding runs

In this task, you were asked to implement find_runs(): a function that takes in a list and returns a list of index pairs consisting of each of the runs. A "run" is a sorted slice of your list: equivalently, this means a run is any non-decreasing slice of your list.

The starter code provided the basic framework and asked two questions:

1. How can you tell if a run should continue?
2. How can you tell if a run is over?

The idea of a run being a 'non-decreasing slice' implies that 'we should add to the run as long as the current number is greater than or equal to the last one in the run'. Thus, the answers to the two questions are as follows:

1. How can you tell if a run should continue? **If the current item is >= the last one in our current run.**
2. How can you tell if a run is over? **If the current item is < the last one in our current run.**

With the starter code provided, we have defined run_end to start at 1. This means run_end is the index of the *next* item to add to your current run. As such, comparing to the last item in your current run is simply comparing to the item at index run_end - 1.

If you have this comparison implemented, then the remaining tasks are to:

1. Update runs when a run is over, using run_start and run_end
2. Update run_start when a run is over: when should a run start (compared to run_end)?
3. Update run_end regardless of whether the run is over or not

Finally, there's one last step to consider: what if run_end == len(lst) and the loop ends? You may still have one more run on-going, and you'll need to add that to your list of runs before returning it.

Task 2: Merging runs

Now that you have runs from find_runs(), the next step is to merge them. This "merge" step is exactly the same as the "merge" from mergesort! Our runs are sorted lists, which is precisely the _merge() function takes.

In regards to our makeshift Stack: using append() and pop() is sufficient for emulating a Stack without creating the entire Stack class. We pop the last 2 runs, merge the corresponding slices, and then add the resulting run -- which is larger and still sorted -- to the end.

Remember: our list of runs is in order, so that means we're merging from the end to the start. So if you have a list of runs like [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)], after the first set of merges you would have [(0, 1), (1, 2), (2, 3), (3, 5)], then the next merge would give you [(0, 1), (1, 2), (2, 5)], and so on.

Task 3: Descending runs

In identifying descending runs, you look at the first two values in a run: if the first value is larger than the second, then it's a decreasing run. Otherwise, it's an increasing run.

The code for this is similar to your original find_runs(), but with a few more additional tasks:

- You need a variable to keep track of whether your current run is ascending or descending.
- If this variable indicates your run is ascending, you continue as you would in your original find_runs(): if the current item is >= the last one in your current run, then the run continues. Otherwise, the run is over.
- If this variable indicates your run is descending, you do the opposite, continuing the run only if the current item is <= the previous one.

When starting a new run, you *must* make sure to update the variable that tracks whether the current run is ascending or descending!

Once you properly implement the code for identfying the descending runs, you'll have one more step to do: reverse those runs in-place, such that all runs returned are now ascending runs.

Suppose you have a run from index run_start to index run_end. To reverse it, you can use the reversed() function and re-assign that slice of your list. For example, doing something like lst[run_start:run_end] = reversed(lst[run_start:run_end])

There are other ways to reverse part of your list, too! This is just one possible solution.

Task 4: Minimum run length

Identifying whether a run meets some minimum length requires only a simple if-statement that checks the difference between run_end and run_start. Afterwards, you only need to call insertion sort on the slice of the list from run_start to however many elements you want in your slice (i.e. to run_start + MIN_RUN). Note that we provided a constant to you: MIN_RUN. This makes it easier to change the minimum run length, especially if it gets used in multiple places.

If we do end up with a run that's less than MIN_RUN and call insertion sort on it, make sure to update run_end as well! Additionally, consider the case where run_start + MIN_RUN extends beyond

the length of the list -- this is a possiblity now, and you must account for it, lest you run into IndexErrors.

Task 5: Optimizing merge

The algorithm for optimizing merge is mostly laid out for you already: when we merge two consecutive runs, store one of the runs in a temporary variable. For instance if you have: [… , 10, 15, 17, 8, 11, 12, 16, …] where [10, 15, 17] will be referred to as A, and [8, 11, 12, 16] will be referred to as B.

You would store A into its own variable (a "temporary list"). Afterwards, you perform the standard merge algorithm in-place: you'll need to keep track of three indexes -- one for your current index in the temporary list containing A's values, one for your current index in the other list, B, and the last index for your current index in your overall list. This last index will start as the start of your run -- i.e. the starting point of run A.

Afterwards, just perform the merge algorithm: look at the item in the temporary list with A and compare it to the current item in B, placing the smaller one in the list, and adjusting indexes as needed. For a refresher on merge, look back to the prep!

Task 6: Limiting the 'runs' stack (harder!)

The optimization in this task isn't one that saves *time*, but it saves *space*. In future CSC courses, you'll learn that saving space can save time -- or rather, using too much memory can hurt your time. As a simple explanation: suppose we have a list with millions of runs in it. With our current implementation, the list of runs we store would *always* contain those millions of runs, even though we only ever care to look at a few runs at a time. That's a lot of wasted memory!

We also have a preference for merging large runs (or rather, not having small runs), so we merge as needed in accordance with the properties outlined in the lab handout: you take a look at the lengths of the three newest runs, and then merge if 1) the right-most run (C) is larger than the run before it (B), or 2) the combined length of the two right-most runs (B and C) is larger than the run to the left of them (A).

In the first case, the two newest runs on our stack get merged. In the second, the smaller of the right-most (C) and third right-most (A) gets merged with the second right-most (B) -- we can't merge A and C together, as they're not consecutive runs.

This cycle of checking the three right-most runs and merging them continues until we don't have to do any merging, and then we continue onto finding our next run.

Quiz 11

In this quiz, you were asked to implement a function that returns a sorted list with the same values of a given list, lst, but without any duplicates in it.

Having a sorted list means we can very easily tell if an item has a duplicate or not. Suppose we have the list [1, 2, 2, 2, 3, 10, 10, 20]: we know 2 has duplicates because there's a *consecutive* chain of 2s! We don't have to look anywhere else in the list-- we just have to look at the elements around it.

So, consider an approach where you only add an item to your new list if the item before it is different (or, alternatively, if the item *after* it is different). For example, suppose we're looking at index 4 in the list above -- that is, the value 3. We'll add it to our new list because 3 is different from 2, and then we move on to index 5 (which has a value of 10). 10 is different from 3, so we add that, and move on to index 6 (which also has a value of 10). Since 10 is equal to the previous element, 10, we don't add that to the list.