Lab 8: Sorting Algorithms

In this lab, you will produce both an implementation of a sorting algorithm (for example, the “selection sort” described in detail below) and the logical statements that would be used in a correctness proof for it (that is, a pre- and post-condition for each function, and a loop_precondition, loop_invariant, and loop_postcondition for each loop, and a progress (or loop_progress) expression for each recursive function or loop).

The logical statements may be either comments or executable statements using our logic functions (the ones used to check bubblesort will be available for your use). If any logical statement is identical to an adjacent logical statement (e.g., an outer loop postcondition and the subsequent function postcondition), you may just give a comment to that effect rather than copying the statement. You will not need to provide the proof itself.

“Selection Sort” works as follows: Instead of swapping adjacent letters of the existing string, we’ll repeatedly select the next letter for the resulting (sorted) string, and put it directly in place. Each time through the outer loop, we use an inner loop to identify the index at which we’ll find the smallest of the unsorted letters — then we put this in place (and move the letter that was there into the place it had occupied).

For example, we’ll sort the letters of "joetheplumber" (written as a quoted string here, though in Python we’ll need to use a list of letters, list("joetheplumber")). In the first trip through the outer loop, we find that the smallest letter (the 'b') is in position 10. We then put this at the start of the list (position 0) and put the 'j' from position 0 where the 'b' was, producing "boetheplumjer". In the second iteration of the outer loop, we find the smallest letter after that 'b', which is the 'e' in position 2 (we could also have found the 'e' in position 5 or 10). We then switch this 'e' with the 'o' in position 1, producing "beotheplumjer". By repeating this “swap the smallest unsorted into the right position” process we eventually produce the sorted list "beeehjlmoprtu".

You are to create a working sort procedure or function, and all the logical statements. You may implement selection sort in the file selectionsort.py (which is completely empty in your starter project), or create a different algorithm if you have email approval from Dave or Suzanne to do so, but in any case your logical statements must correspond to the procedure/function you write. You may use the functions in checking_functions.py and helping_functions.py as much or as little as you wish.

You may approach this problem however you like, as long as you produce a working sort procedure/function and correct logical statements that correspond to the description above. We recommend building the “selection sort” algorithm as a set of loops using an algorithm design approach advocated in David Gries’ book “The Science of Computer Programming”:

a) Get the Lab 8 starter files, which include the files from the bubblesort/anagram checking program that was presented in lecture (in the files anagrams.py, bubblesort.py, and checking_functions.py).

b) Add a test suite for selectionsort.py (Hint — this should be very easy.), and (if you wish) add to the test suite for anagrams.py.

c) The starter version of the sort procedure is mostly comments, except for the creation of a copy (for use when checking the postcondition) and the standard sort postcondition. Run your tests, and confirm that all failing tests report a postcondition error rather than returning the wrong answer.
d) The sort procedures’s pre- and post-conditions may well serve as loop pre- and post-condition for the outer loop — think about what changes think about what is being changed by the outer loop (see the description above) and try to come up with an invariant and progress condition for the outer loop; write these into your function, together with a loop header that has no body except for the invariant statement. (If you like, put in a loop body that satisfies your progress condition but doesn’t actually solve the problem — running the test after doing so should result in the same failing cases as in the previous step, but this time with errors about invariants rather than just the postcondition.)

e) Think about what must be accomplished in one execution of the outer loop body, to maintain the invariant while making progress — write some Python statements to achieve this goal. For any step that requires work that grows with the size of the string, create an empty loop with loop pre- and post-conditions that claim to have achieved the goal, like the outer loop you wrote in the step above. (Once again, you are welcome to put in a loop body that makes progress in some trivial but unhelpful way, and then run the test suite — this time all your errors should be about the inner loop’s invariant or postcondition, since the rest of the program should be correct.)

f) Think about what must be accomplished in one execution of the inner loop body, to maintain the invariant while making progress — write some Python statements to achieve this goal. This time, each step should be something that can be expressed in a simple Python statement.

g) Test your program. If you did each step correctly, your program will work right the first time, and have all the correct logical statements.

We recommend that you submit after each non-trivial step. Whatever you do, please remember to commit your final files when you are done.