Daily Coding Problem #260

Problem

This problem was asked by Pinterest.

The sequence `[0, 1, ..., N]` has been jumbled, and the only clue you have for its order is an array representing whether each number is larger or smaller than the last. Given this information, reconstruct an array that is consistent with it. For example, given `[None, +, +, -, +]`, you could return `[1, 2, 3, 0, 4]`.

Solution

Notice that if there are no negative signs in our input, we can return the original sequence `[0, 1, ..., N]`. Furthermore, if we have just one run of consecutive negatives, we can reverse the corresponding entries in the original sequence to produce a decreasing run of numbers. For example, given `[None, +, -, -, -]` we reverse the last three entries of `[0, 1, 2, 3, 4]` to get `[0, 1, 4, 3, 2]`.

We can extend this trick to more complicated input, matching plus signs with elements of our original sequence and reversing subsequences whenever there is a run of minus signs.

To keep track of which numbers are being reversed, we can use a stack. As we traverse the input array, we keep track of the corresponding elements of the original sequence. For a run of positive signs, we keep the elements from the original sequence. For a run of negative signs, we push those elements onto the stack. When the run of negatives ends,

we can pop those elements off one by one to get a decreasing subsequence in our answer.

Since there is one fewer + or - sign than elements in our answer, the last element must be generated separately. Additionally, a run of negative signs can end with either a positive, or the end of the input array, so we must make sure to empty the stack at the end.

```
def reconstruct(array):
    answer = []
    n = len(array) - 1
    stack = []

    for i in range(n):
        if array[i + 1] == '-':
            stack.append(i)
        else:
            answer.append(i)
            while stack:
                answer.append(stack.pop())

    stack.append(n)

    while stack:
        answer.append(stack.pop())

    return answer
```

This algorithm runs in `O(N)` time and space, since in the worst case we are filling up a stack the same size as our original array. It may seem as if there is a higher time complexity, because of our inner loop over the stack, but the total number of items we pop off is bounded by the size of the input array.