CS 120: Intro to Computer Programming II

In-Class Activity - Advanced Topics - Sorting

In this ICA, we'll be writing and testing several sort functions. In each case, we'll write a function that takes an array as its only parameter, and returns nothing; it re-arranges the values inside the array. (This is called an **in-place sort.**)

Activity 1 - Turn in this one

First thing to write is a testing function! Write a function which does the following 100 times:

- Generates 20 random integers
- Duplicates the array (use slicing, it's easy!), so that you will have a "pristine" copy if you need to report an error.
- Call Python's standard `sorted()` function to sort the random data; save the result away as a third array.
- Call the function `bubble_sort()` (we'll change this function name each time we need to test a new function) to sort one of the copies of the random data.
- Compare the result of `sorted()` with the one that you sorted with your own function; if they are the same, then silently move on to the next iteration of this loop; however, if they don't match, print out the **original, unsorted** data (so that you can debug your sort).

Oh, and record the time when your function **begins and ends** (the entire function, including all 100 iterations) and print it out, at the end of the function...so that you can compare how long each of your algorithms took to run.

How do you test a testing function... ?

To test out the tester itself, go ahead and write `bubble_sort()`. Remember that bubble sort works as follows:

```
repeat n times
    scan through the entire array, comparing adjacent values
        if any value is greater than the one to its right, swap them
```

Activity 2 - Turn in this one

We've done a famously bad algorithm (Bubble Sort); now let's do a famously good one! In this Activity, implement Merge Sort.

You've seen Merge Sort discussed in a video, and you also analyzed the `merge()` function (one of the steps of Merge Sort) in a recent ICA. But just to remind you, Merge Sort is a **recursive** sort that works like this:

```
if the length of the array is 0 or 1
    return immediately

split the input array into two halves
recurse into merge_sort() to sort each one
call merge() to merge the two sorted halves back into the original array
```

Activity 3 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Bubble Sort isn't the only $O(n^2)$ sort algorithm. There are a lot of other algorithms - since they are $O(n^2)$, they definitely are **much** worse than the $O(n \lg n)$ sorts (Merge Sort, Quicksort, Heap Sort, etc.). However, if you're willing to tolerate an $O(n^2)$ sort, there are lots of better options than Bubble Sort!

Implement Selection Sort, which works as follows:

```
scan through the entire array, and find the minimum value.  Record where you found it.
Then, swap that value into the 1st slot of the array.
    (The old 1st will go into the slot where the minimum used to be.)

Repeat the above, but now start the search at the {\bf second} element in the array.
    Thus, you are finding the second-smallest
    When you are done, swap it into the second slot

Repeat for all slots
```

(activity continues on the next page)

Activity 4 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Let's do Quicksort. Or rather, because the `partition()` algorithm is complex, let's do a "cheating" version of it.

Quicksort works a little like Merge Sort - but in this algorithm, instead of splitting the array in half, you choose a "medium" value (called the pivot) and then sort the array into things that are less than (or equal to) the pivot, and other things that are greater.

Sometimes, the arrays will be (roughly) balanced in size; sometimes not. How well balanced they are depends on your luck in choosing a good pivot.

The cool thing about Quicksort is that it does **not** need a merge step at the end; by definition, you know everything in the "less than" array goes to the left of the pivot, and everything in the "greater than" array goes to its right.

In the **real** Quicksort, `partition()` shuffles values around inside the array itself - and so we never actually do any array duplication. Instead, when we recurse, we use that trick that we've seen before of recursing using `start,end` parameters. This is nice, because, when they return, it's never necessary to join the arrays back together!

But for this Activity, we'll actually split things. It's bad for performance, but it's good for code simplicity.

Implement Quicksort (hackish version) as follows:

```
if the length of the array is 0 or 1
    return immediately

take the {\bf middle value} of the array, we will designate it the pivot
    swap it with the very first value

scan through the entire array (except for the first slot, which contains the pivot)
    sort each value into one of two arrays: lesser and greater
    (values that are == to the pivot go into the lesser)

recurse into both arrays

copy values back into the input array, in the following order:
    1) the (sorted) ''lesser'' array
    2) the pivot
    3) the (sorted) ''greater'' array
```

(activity continues on the next page)

Activity 5 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Of the $O(n^2)$ sort algorithms, Insertion Sort is probably the fastest. Basically, it's an implementation of the old "insert into a sorted array" problem, that we've discussed several times this semester: for each new value, you scan the array to find the proper location to place the value; once you've found it, you shift lots of things over, and then drop it into its proper location.

The trick with Insertion Sort is that the length of the array never grows! Instead, the left part of the array is the "sorted" part; it grows a little longer each time that you insert a new value. But the rest of the array is where all of the unsorted data goes. So at a high level, the algrorithm is:

```
remove one value from the unsorted array
    thus it's one element shorter
insert that value into the sorted array
    shift things as needed
    obviously, this array gets one element longer
```

The only trick is that the two arrays actually live inside the same array in memory!

Implement Insertion Sort:

```
set a counter (initialized to 1) which tells us how many values are sorted (so far)

while counter < length of array
    copy the first ''unsorted'' value; store it into a temporary variable.
    scan the ''sorted'' values, to find the correct location to place the new value
    shift values over as necessary
        NOTE: the slot where the ''new value'' came from will get destroyed in this process
    write the new value into its proper location
    increment the counter
```

(activity continues on the next page)

Activity 6 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

We gave you the `partition()` function for Quicksort in the Optional Activities for ICA 12-3. Go look it up, and copy it into your code here. It actually has a serious missing part: it partitions the **entire** array, instead of just part of it.

Start by updating this function to include `start,end` parameters.

Then, rewrite your Quicksort from the previous Activity, making the following changes:

- The recursive function has `start,end` parameters.
- You use the new `partition()` function to move data around **inside** the array, instead of making two smaller arrays.
- You recurse into Quicksort, for each sub-array, by calling the function with **updated values** for `start,end`.