CS 120: Intro to Computer Programming II

In-Class Activity - 12 Complexity - Day 2

Activity 1 - Turn in this one

The following function runs on an array of values. First, determine what it does; then, figure out its time complexity. Finally, run some tests to see if your analysis (of its time complexity) is correct.

```
def mystery_one(data):
    for i in range(len(data)):
        for j in range(len(data)-1):
            if data[j] > data[j+1]:
                data[j],data[j+1] = data[j+1],data[j]
```

(Yes, some of you - maybe more than a few - may recognize this. Don't give the answer away immediately; let your fellow members of the group think about this and hopefully figure it out for themselves.)

Activity 2 - Turn in this one

The `append()` method, on Python arrays, takes `O(1)` time (with a small caveat we won't talk about here). Write a loop which does lots of `append()` calls; test to see if what I've told you is actually true.

Turn in your code, the times from your experiment, and your conclusions.

Activity 3 - Turn in this one

Measure the following code, to see how long it takes to run. Do **not** modify it - because if you remove the dummy variable, Python (some versions of Python, anyhow) will actually optimize your code, silently, to make it faster.

```
def concat_build(n):
    retval = ""
    dummy  = retval
    for i in range(n):
        retval += " "
        dummy   = retval
    return retval
```

This function ought to run in $O(n^2)$ time, can you confirm that it does? What would that tell you about the cost of the string-concat operation, as compared to appending to an array?

(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 figure out how `append()` works. To do this, I want you to write a simple class, as follows:

- Have a Python array as a private field. This class will **never** call `append()` on that array; however, this class may, from time to time, re-allocate the array, copy all of the values from the old array to the new array, and then switch over to using the new, larger array.
- Keep a private field which represents how many of the values in the array are "in use." This could be as little as zero (that is, this object represents and empty list) or as much as the current length of the private, internal array.
- Implement a `__len__(self)` method, which returns the current number of **active** elements in the curren array.
- Implement a `append(self, val)` method, which stores a new value into the private, internal array. If there is any free space in the array, then simply place it in the next avaialble slot, and increase the count of "active slots."

  But if the array is full, allocate a **brand-new array**, which must be **exactly** twice the size of the current one. Copy all of the values from the old array to the new, and then make the new array the current array for your class.

Test this out with a loop. Is your `append()` method $O(1)$ like we thought?