CS 120: Intro to Computer Programming II

In-Class Activity - 12 Complexity - Day 1

Activity 1 - Turn in this one

For this activity, we will need **(at least) two or three** students, all running the exact same code! Try running the following function, for various values of n. **Run the code on at least two different computers - and record all of the results, from all of the computers.**

Start very small (say, $n = 10$) and then multiply by 10 each time. Once you've gotten to a size large enough that it takes more than a couple seconds, start adding instead of multiplying. For example, if you stop at 100 thousand, then try 200 thousand, then 300 thousand, then 400 thousand, etc.

```
def foo(n):
    total = 0
    start = time.time()
    for i in range(n):
        total += i
    end = time.time()
    print(f"foo(): n={n} total={total}      elapsed: {end-start} secs")
    return total
```

Report your results (from all the students) to your TA.

Activity 2 - Turn in this one

Discuss, with your group, the results from the previous Activity. Hopefully, you found that **all** of the different computers showed a linear relationship between n and the elapsed time - at least, once n was large enough. Is that true in your group?

But did all of the computers report the same time? Or did some computers run in different time than others?

What does this tell you about the nature of an $O(n)$ algorithm? Can we predict, just from the code, how long it will take to run?

Solution: No, we cannot predict how long the code will take to run - because different computers run at different speeds. (In fact, even the **same** computer can vary, based on small difference int the load of the computer, random variation, etc.)

Activity 3 - Turn in this one

Now go back, to very small values of n (10, or maybe 100). Run the same code again, but run it **many** times with small n. What do you see about the time? Is it consistent across all of the runs? How much can it vary? Report your time results, and your discussion, to your TA.

Solution: When n is small, we somtimes notice significant differences in time - because tiny changes in the state of the computer can affect the performance.

When n is larger, these tiny variations and interruptions tend to average out, and have very little impact on performance (at least, compared to the **overall** time taken).

(activity continues on 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.

Set up a dictionary, filled with random data, with the following code:

```
table = {}
for i in range(1000*1000):
    val = random.randint(0, 1000)
    if val not in table:
        table[val] = 1
    else:
        table[val] += 1
```

Now, let's test out the performance of indexing into a dictionary, using the following loop:

```
def bar(n):
    sum = 0
    start = time.time()
    for i in range(n):
        val = random.randint(0,1000)
        if val not in table:
            continue
        sum += table[val]
    end = time.time()
    print(f"sum={sum}     elapsed seconds: {end-start}")
```

Run this code on one of the student computers; vary n until you can figure out how it performs. It **should** run in $O(n)$ time - do your numbers match up with that expectation?

And is this the same speed as the previous loop, or does it run at a different speed?

Solution: Your code should run in $O(n)$ time - but it will **not** be nearly as fast as the loop from Activity 1. This is because, in Activity 1, we had a trivial loop, which simply incremented `i` and then did a sum. But in this Activity, we have a much more complex loop, which does random number generation, dictionary lookups, etc.

So two $O(n)$ algorithms do **not** run at the same speed, even on the same computer!