CS 120: Intro to Computer Programming II

In-Class Activity - 05 Recursion, part 2 - Day 1

Don't discard your work from this ICA, when you turn it in. You will be using some of it again, in the next day.

Background: `time.time()`

One of the ways to check how long a program takes to run is with the `time` library. The following snippet will give you the current time, measured as the number of seconds since the Epoch:

```
import time
x = time.time()
```

While this can occasionally be useful, we will find it a lot more useful to measure **deltas** - that is, the amount of time **between** two points. So we're going to use the following code often:

```
import time
start = time.time()
... do something ...
end = time.time()
print(f"Elapsed time: {end-start} seconds")
```

This works because `time()` returns a floating-point number - meaning that it can measure fractions of seconds. I'm not clear on exactly how precise it can be - that probably varies from OS to OS - but we still get a pretty good picture. Remebmer, however, that all computer clocks are approximate - especially when we're measuring tiny bits of time. So when the deltas are very small, you have to remember that they might not be 100% accurate.

What is the Epoch?

- `https://en.wikipedia.org/wiki/Unix_time`
- `https://www.epochconverter.com/`

(activity continues on the next page)

Activity 1 - Turn in this one

Today, we're going to be needing some large lists of random numbers. Start the day by writing a function,

```
def gen_rand(n):
```

which will generate an array of `n` random numbers. Don't worry if the array might have a couple duplicates, but it's important that we **do not** have many of them. Therefore, the range of values you generate should be large enough to make sure that duplicates are rare.

Maybe you can just generate numbers between `0` and `10*n`, or something like that?

Activity 2 - Turn in this one

Let's see how long it takes to generate some random numbers. Run the following snippet for various values of `N`. Each time that you run this, measure the time-delta (by checking the start and end times, and subtracting them).

You will probably find, for very small `N`, that the delta is zero - this is because computer clocks don't have infinite precision; therefore, if you check the clock twice in close succession, you will see the same value.

```
N = ... you choose ...

print("Running")     # print this first, so you know when Python is up

vals = gen_rand(N)
print(f"Generated {N} random numbers")
```

Once you've found a reasonable range of `N` - one that reliably reports more than zero time - change `N` a bit. What happens if you multiply it by 10? Try a wide variety of values - but don't worry, you don't need to test anything longer than just a second or two.

Once you've collected a few data points, can you make a prediction? What is the time cost of the `gen_rand()` function, in Big-Oh?

Activity 3 - Turn in this one

Now, write a simple function which uses a loop to sum up all of the values in an array. Then, call this function - passing it the random numbers. Use `time.time()` to measure how long the function takes. (Make sure to pay attention to the difference between the time it takes to **generate** the numbers, and the time it takes to **sum** them.)

But **before you run this code,** make a prediction: in terms of big-Oh notation, predict how long it will take to sum up all of the numbers. Then test your program, using various values of `N`. Does the evidence match your predictions?

(activity continues on the next page)

Activity 4 - Turn in this one

In our next meeting, we're going to start working with a recursive sum function, which slices the array every time that it recurses. But for now, we're not going to actually do recursion; we're just going to simulate it with a loop.

Run the following function, for various sizes of input data. Can you determine what its runtime is, in Big-Oh notation?

```
def sumlist_pretend_slice_1(vals):
    sum = 0
    while len(vals) > 0:
        sum += vals[0]
        vals = vals[1:]
    return sum
```