CS 120: Intro to Computer Programming II

In-Class Activity - 12 Complexity - Day 4

Throughout this activity, we'll need to generate random numbers from time to time. Please use the following function to generate the numbers.

```
def gen_rand_data(n):
    retval = []
    for i in range(n):
        retval.append( random.randint(0,10*n) )
    return retval
```

If you have extra time at the end, explore what would go wrong if you used this alternate version. (This version is what a lot of students write at first - which is why I gave you a correct version above.)

```
def gen_rand_data(n):
    retval = []
    for i in range(n):
        retval.append( random.randint(0,100) )
    return retval
```

Why does this version mess up our experiment?

Activity 1 - Turn in this one

Compare these two functions. Notice that they are identical, except for the data structure that they use. Make a prediction about the big-Oh performance of each one. Then, run experiments to see if you are correct. (Hint: The `add()` and `in` operations on a `set` run in $O(1)$ time.)

```
def count_dups_1(data):
    so_far = []
    count  = 0
    for val in data:
        if val in so_far:
            count += 1
        else:
            so_far.append(val)
    return count

def count_dups_2(data):
    so_far = set()
    count  = 0
    for val in data:
        if val in so_far:
            count += 1
        else:
            so_far.add(val)
    return count
```

Solution: The first one runs in $O(n^2)$ time - because the `in` operator on arrays has to do a brute-force search of the array. Thus, each pass of the loop is $O(n)$, and thus the function is $O(n^2)$.

The second one runs in $O(n)$ time, because the `in` operator on sets runs in $O(1)$.

(activity continues on the next page)

Activity 2 - Turn in this one

Sometimes, there is more than one data structure involved. Consider the following function:

```
def all_pairs(data1, data2):
    retval = []
    for v1 in data1:
        for v2 in data2:
            retval.append( (v1,v2) )
    return retval
```

Discuss the following with your group (and record your notes for your TA). **This is taking big-Oh further than you've seen. Don't be afraid to make a mistake; just give this your best shot.**

- Suppose that `data1` and `data2` were both (roughly) the same size. Use n to represent that size. What would be the runtime of thise function?
- Suppose that you knew that `data1` was **very** small (let's say, no more than three elements). Would this change how you might describe the performance of the function, in terms of big-Oh?
- What if `data1` was large but `data2` was small, would that make any difference?

Solution: The runtime is $O(n^2)$, because we have nested loops, each of them (roughly) $O(n)$ passes.
But if either one of the input arrays are short, then the runtime drops to $O(n)$, because the runtime is "$O(n)$ times a constant."

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.

Sometimes, a little bit of pre-processing of your data will make your algorithm a lot more efficient. Doing reverse lookups into a dictionary is a classic example.

The following function takes a dictionary, and an array. For each value in the array, it hunts the dictionary to see if there is a **key** which maps to this value; if so, it puts the key into a result array. If not, it marks the entry with a `None` result. Analyze this function to find out its big-Oh runtime performance. Assume that the dictionary, and the input array, are (roughly) the same size; call that size n.

```
def lookup_in_dict(dic, search):
    retval = []
    for s in search:
        found = None
        for key in dic:
            if dic[key] == s:
                found = key
                break
        retval.append(found)
    return retval
```

Analyze this function to figure out its runtime performance. **Verify this experimentally.**

Solution: This function runs in $O(n^2)$ time, because it has nested loops, each of which are $O(n)$ in length.

(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.

Now, **rewrite the function** (from the previous step) so that it creates a "reverse" map for the dictionary: that is, it maps values to keys, instead of the other way around. (To simplify your function, you may assume that each value only shows up once in the input dictionary.)

Once you've rewritten your code, analyze it (it should be faster!), and then confirm that your code works better than before.

Solution:

```
def lookup_in_dict(dic, search):
    rev_dic = {}
    for key in dic:
        val = dic[key]
        rev_dic[val] = key

    retval = []
    for s in search:
        if s in rev_dic:
            retval.append(rev_dic[s])
        else:
            retval.append(None)
    return retval
```

This runs in $O(n)$ time, becauase the preliminary loop runs in $O(n)$ time, And the `in` operator on dictionaries runs in $O(1)$ time.