Work with your neighbor. (This will be graded for participation only.)

1. For each code fragment below, state its worst-case big-O complexity.

a) **ANS:** O(n)

```
n = int(input())
while n > 0:
    print(n)
    n -= 1
```

b) **ANS:** O(n) – Dividing n by 2 does not change the complexity. The division by 2 does not happen on each iteration of the loop.

```
n = int(input())
n = n/2
while n > 0:
    print(n)
    n -= 1
```

c) **ANS:** $O(n^2)$

```
n = int( input() )
if n % 2 == 0:
    for i in range(n):
        x = x + 1
else:
    for i in range(n):
        for j in range(n):
            x = x + 1
```

d) **ANS:** O(n) – m is a constant and so the inner loop is O(1). (You can think of writing out 100 print statements as the body of `for x in numlist1`)

```
m = 100
for x in numlist1:
    for y in range(m):
        print(x + y)
```

e) **ANS:** O(n)

```
def count(char, lc_str):
    num = 0
    for i in range(len(lc_str)):
        if char == lc_str[i]:
            num += 1
    return num
```

2. The following function takes a list of integers and performs some kind of mystery transformation on the list. First, determine what the function does. Second, determine its worst-case big-O complexity. (If you recognize what the function does immediately, don't give the answer to your fellow tablemates. Let them work it out for themselves.)

```
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]
```

ANS:
This function sorts the list. It consists of two nested loops, where each loop is O(n), so the function is $O(n^2)$.

3. Given a list `alist`, write a function `have_two_greater(alist)` that returns a *new* list that contains all of the elements in `alist` which have at least two elements that are greater than itself. *Do not use a sorting algorithm to first sort the list.*
Requirements:
   - *Do not use sorted() or sort() or sort the list yourself*
   - *Don't use built-in max()*
   - *Don't use pop() or delete – that is, don't modify the argument list passed in*

ANS:

```
def have_two_greater(alist):

    result = []
    for a in alist:
        count = 0
        for b in alist:
            if b > a:
                count += 1
        if count >= 2:
            result.append(a)
    return result
```

What is the complexity of your function? Why?
This function is $O(n^2)$. It consists of two nested loops, where each loop is O(n). The computations within the loops are: assignment, addition, if, greater than, and append. Each of those is O(1).

4. Given a list `alist`, write a function `second_largest(alist)` that finds the second largest element of the list argument `alist`.

   ANS:

```
def find_second_largest(alist):
    first = alist[0]
    second = alist[0]
    for elem in alist:
        if elem > first:
            second = first
            first = elem
        elif (elem > second and elem != first):
            second = elem
    return second
```

   What is the complexity of your function? Why?

   This can also be done in one loop, as shown above. This is O(n). The function has one loop which iterates over the elements of `alist`. All of the computations within the loop are O(1).

   You could also traverse the list twice in two consecutive loops (not nested). The first time, you find the maximum element. The second time, you can find the second largest element. Since the loops are not nested, this function is O(n).

5. Given your solution to problem 4, can you write a solution for `have_two_greater(alist)` that is O(n)?

   ANS:
   If you have an O(n) solution to problem 4, then you can use that function to find the second largest element, and given that, iterate through the list to find all elements less than the second largest. Such a solution is O(n).