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

Problem 1. Merging two lists. Determine the complexity of the `merge()` function below, which is an iterative solution to merging two sorted lists.

```
def merge(in1, in2, output):

    """merges two input lists into an output list
    in1, in2 – the sorted input lists
    output -  a list whose length accommodates both lists
              (the elements of output will be overwritten)
    """
    assert len(output) == len(in1) + len(in2)

    pos1 = 0
    pos2 = 0
    while pos1 < len(in1) and pos2 < len(in2):
        if in1[pos1] <= in2[pos2]:
            output[pos1+pos2] = in1[pos1]
            pos1 += 1
        else:
            output[pos1+pos2] = in2[pos2]
            pos2 += 1

    # if one of the lists was shorter, add any
    # of the leftover elements to the output list
    while pos1 < len(in1):
        output[pos1+pos2] = in1[pos1]
        pos1 += 1
    while pos2 < len(in2):
        output[pos1+pos2] = in2[pos2]
        pos2 += 1

    assert pos1+pos2 == len(output)
```

ANS:

Complexity of merge(): O(n)
The function has two while loops that are not nested. It writes to every position of output, whose length is the sum of both list lengths. The complexity is the O(n), where n is the length of output.

Problem 2. In this problem, you will write two different versions of a function and determine their run-time complexity. Write a function `has_dups(alist)` that takes a list of integers and returns `True` if `alist` contains duplicate values and `False` otherwise. If `alist` is empty, the function returns `False`.

a) In the first version, use nested loops. What is the complexity of your function?

ANS:

```
def has_dups(alist):
    for i in range(len(alist)):
        for j in range(len(alist)):
            if i != j and alist[i] == alist[j]:
                return True
    return False
```

The complexity of the function is $O(n^2)$.

b) In the second version, use a dictionary to keep track of whether a value has been seen before. Once a value has been seen, the function can immediately return.

ANS:

```
def has_dups(alist):
    counts = {}
    for elem in alist:
        if elem in counts:
            return True
        else:
            counts[elem] = 1
    return False
```

c) Since the function only iterates through `alist` once (worst case), its complexity appears to be O(n). What would we need do know (that we don't know yet!) about dictionary operations in order to give a thorough answer?

ANS:

To determine the complexity of the solution above, we would need to know the complexity of the operation

```
elem in counts
```

which is *looking up an element in a dictionary*, and the complexity of

```
counts[elem] = 1
```

which is *inserting an element into a dictionary*. Note: Later in the semester, we will explore the implementation of dictionaries (which are called hash tables in most other languages) and discover that both of these operations are O(1).

Wait until we have covered linked list complexity before doing the next problem.

Problem 3. Write a method `concat(self, list2)` that concatenates the linked list `list2` to the end of `self`.

ANS:

```
    def concat(self, list2):
        if self._head == None:
            self._head = list2._head
            return
        current = self._head
        prev = current
        while current != None:
            prev = current
            current = current._next
        prev._next = list2._head
```

What is the complexity of `concat()`?

O(n). We have to iterate to the end of the first list (self) in order to find the end and change the reference of the last element to refer to the first element of list2.