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

List comprehension problems:

1. Write a function `times_k(alist, k)` that takes as arguments a list `alist` and an integer `k` and returns a list consisting of the elements of `alist` each multiplied by `k`. Your solution should use a list comprehension.
   ANS:

```
def times_k(alist, k):
    return [ elem * k for elem in alist]
```

2. Write a function `first_matches_last(alist)` that takes as an argument a list of strings `alist` and returns a list consisting of those elements of `alist` whose first and last characters are the same. Your solution should use a list comprehension.
   ANS:

```
def first_matches_last(alist):
    return [ elem for elem in alist if len(elem) <= 1 or
                                      elem[0] == elem[-1] ]
```

3. Write a function `times_i(alist)` that takes as an argument a list `alist` and returns a list consisting of the elements of `alist` multiplied by the position number of the elements. In other words, if alist is the list

   $[L_0, L_1, L_2, ..., L_n]$

   then the value returned by `times_i(alist)` is the list

   $[L_0 \times 0, L_1 \times 1, L_2 \times 2, ..., L_n \times n]$

   Your solution should use a list comprehension.

   ANS:

```
def times_i(alist):
    return [ i * alist[i] for i in range(len(alist))]
```

Final Exam Review Problems:

4. (Trees) For the following problem, use the implementation of a `BinaryTree` class below:

```
class BinaryTree:
    def __init__(self, value):
        self._value = value
        self._left = None
        self._right = None
```

Also, assume that `value()`, `left()`, and `right()` are the usual getters for the attributes.

Write a recursive function `sum_evens(tree)` that takes a binary tree `tree` and returns the sum of the values in the tree that are *even*.

ANS:

```
def sum_evens(tree):
    if tree == None:
        return 0
    if tree.value() % 2 == 0:
        return tree.value() + sum_evens(tree.left()) +
                                   sum_evens(tree.right())
    else:
        return sum_evens(tree.left()) + sum_evens(tree.right())
```

5. (Complexity) For the bodies of code below, state their worst-case, big-O complexity.

(a)

```
m = len(numlist2)
for x in numlist1:
    for y in range(m):
        print(x + y)
```

ANS:
$O(n^2)$

(b)

```
n = len(arglist) // 2
sum = 0
while n > 0:
    sum += arglist[n]
    n -= 1
```

ANS:
O(n)

(c)

```
def has_overlaps(list_a, list_b):
    for item in list_b:
        if item in list_a:
            return True
    return False
```

ANS:

$O(n^2)$

6. Under what condition is searching for an item in a list O(log(n))?

   ANS: If the list is sorted, since then you can use binary search which is O(log(n)).

7. The following ordering of complexities is correct. (Answer True or False)

   O(n), O(log n), $O(n^2)$

   ANS: False
   The correct ordering is O(log n), O(n), $O(n^2)$

8. How would you design a stack which, in addition to `push()` and `pop()`, has a method `min()` which returns the current minimum value in the stack? The methods `push()`, `pop()`, and `min()` must operate in O(1) time. Describe your solution in English, don't write the code.

ANS:
For push(self, item), you could push not only the item on the stack, but also the current min. You would want to create a class such as StackItem that has attributes for the item and the current min. (Or you could simply push a tuple of the item and the current min.)

The push() method would check the min associated with the current top of the stack. It would then use the min of its value or the stack's current min, which ever is smaller, and push that on the stack with the item.