1a.)

```
def reverse1(lst):
    rev_lst = []
    i = 0
    while(i < len(lst)):
        rev_lst.insert(0, lst[i])
        i += 1
    return rev_lst
```

(n) (n)

the worst case run time for this function is n^2

the while loop runs n times because i starts at 0, increases by 1 with each iteration, and goes until it's (1) less than the length of the list

because .insert is being used to place the new number at the front of the list, each iteration it has to shift every element in the list to the right, giving it a run time n

$n \cdot n$ results in an overall run time of n^2

1b.)

```
def reverse2(lst):
    rev_lst = []
    i = len(lst) - 1
    while (i >= 0):
        rev_lst.append(lst[i])
        i -= 1
    return rev_lst
```

(n) (1)

the worst case run time would be n

the while loop runs n times (i decrements by 1 each iteration), giving it a time complexity of n

since .append is adding the value to the end of the list, the length of the list has no effect, so it has a time complexity of 1

$n \cdot 1$ results in an overall run time of n

2EC.) 1. Show that the following series of $2n$ operations takes $O(n)$ time: n `append` operations on an initially empty array, followed by n `pop` operations.

each time the array is resized, a new array is created that is 2·the capacity and the previous elements are added to the array. Since the new array is based on the length of the old array, the run time for resizing an array is n

appending to an array has a constant run time, however worst case is that the element that's being appended won't fit and causes a resizing (which costs n time)

therefore, n append operations has a run time of n

similar to the n append operations, usually a pop operation is constant (as long as it's removing the last index, otherwise it'd be n because it has to shift elements left) but a worst case pop operation is it drops the number of elements to below a quarter of the capacity and causes a resize, which as described has a run time of n

Since the n append operations are performed sequentially with the n pop operations, their run times can be added to give the overall run time: $n + n = 2n$, which simplifies to n

2. Consider a variant to our shrinking strategy, in which an array of capacity N, is resized to capacity precisely that of the number of elements, any time the number of elements in the array goes strictly below $N/2$.
Show that there exists a sequence of n `append` and/or `pop` operations on an initially empty `ArrayList` object, that requires $\Omega(n^2)$ time to execute.

if n/2 pop operations are performed, the number of elements goes below N/2 and so the capacity will change to N/2

then if n/2 append operations are performed, each time a new element is added to the list, the capacity will have to be resized to the new n, which takes n time

$\frac{1}{2}n$ operations, each of which costs n time due to resizing, results in a total time of $\frac{1}{2}n \cdot n = n^2$ (disregarding the $\frac{1}{2}$)

3b.)

zcd220_hw3_q2.py | zcd220_hw3_q3.py | zcd220_hw3_q4.p

```
def find_duplicates(lst):
    duplicates = []   (1)
    sorted_lst = sorted(lst)  (nlogn)*
    for i in range(1, len(sorted_lst)):  (n)*
        if sorted_lst[i] == sorted_lst[i - 1]:  (1)
            duplicates.append(sorted_lst[i])  (1)
    return duplicates
```

the worst case run time for my function is nlog(n)

the sorting method has a run time complexity of nlog(n)

the for loop iterates n times, giving it a time complexity of n

.append is not dependent on the size of the list, so it's time complexity is 1

since the sorting step and the for loop are sequential, their run times are added: $n\log(n) + n$

since nlog(n) grows faster than n, the +n can be taken off and the run time simplified to nlog(n)

4a.)

```
def remove_all(lst, value):
    end = False
    while(end == False):  ← n
        try:
            lst.remove(value)  ← n
        except ValueError:
            end = True
```

the worst case run time for this function is n^2

in the worst case, the entire list is made of the value to be removed which would mean the while loop would iterate n times

also in the worst case, when .remove is called, value is at the very last index and .remove would have to traverse through every element in the list (n)

therefore, the resulting worst case run time is $n \cdot n = n^2$

4b.)

```
def remove_all(lst, value):
    available_ind = 0
    for i in range(len(lst)):  (n)
        if lst[i] != value:  (1)
            lst[available_ind] = lst[i]  (1)
            available_ind += 1  (1)
    del lst[available_ind:]  (n)
    return lst
```

the worst case run time for this function is n

the for loop iterates n times so its time complexity is n

every operation within the for loop is constant

the del operation for a list has a run time of the range from the beginning of the splice to the end, which worst case is n

because the for loop and del operation are sequential, they are added to $2n$ which results in a time complexity of n for the function