Question 1:
Consider the following two implementations of a function that if given a list, `lst`, create and return a new list containing the elements of `lst` in reverse order.

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

$\theta(n^2)$

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

$\theta(n)$

If `lst` is a list of n integers,

1. What is the worst case running time of `reverse1(lst)`? Explain of your answer.
2. What is the worst case running time of `reverse2(lst)`? Explain of your answer.

1) The worst case running time of reverse1(lst) is $\theta(n^2)$. The insertion of elements at the beginnig of the list is ultimately a $\theta(n)$ operation and is performed $\theta(i)$ times.

2) The worst case running time of reverse2(lst) is $\theta(n)$. Instead of inserting at the beginning of the list, the elements are appended in reverse order. Appending has a runtime of $\theta(1)$ and this operation is performed $\theta(i)$ times

3(b)

```
def find_duplicates(lst):
    dups = []                                   # O(1)
    len_lst = len(lst)                          # O(1)
    for i in range(len_lst):                    # O(i)
        if lst[abs(lst[i])] >= 0:               # O(1)
            lst[abs(lst[i])] = -lst[abs(lst[i])]
        else:
            dups.append(abs(lst[i]))            # O(1)
    return dups                                 # O(1)
```

Worst case running time: $\Theta(n)$

4(c)

```
def remove_all(lst, value):
    l_lst = len(lst)                            # O(1)
    ctr = 0                                     # O(1)
    for x in range(l_lst):                      # O(x)
        if lst[x] != value:                     # O(1)
            lst[ctr] = lst[x]                   # O(1)
            ctr += 1                            # O(1)
    for x in range(ctr, l_lst):                 # O(x)
        lst.pop()                               # O(1)
```

$\Theta(2x)$

Worst case running time: $\Theta(n)$

Question 4:

The `remove(value)` method of the `list` class, removes the **first** occurrence of `value` from the list it was called on, or raises a `ValueError` exception, if `value` is not present.

Note: Since `remove` needs to shift elements, its worst-case running time is linear.

In this question we will look into the function `remove_all(lst, value)`, that removes **all** occurrences of `value` from `lst`.

a) Consider the following implementation of `remove_all`:

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

θ(1) θ(n) θ(n) θ(1) θ(n²)

Analyze the worst-case running time of the implementation above.

b) Give an implementation to `remove_all` that runs in worst-case linear time.

Notes:

1. Your implementation should **mutate the given list object** (in-place), without using an additional data structure.
2. Your implementation **should keep the relative order** of the elements that remain in the list

c) Analyze the worst-case running time of your implementation in (b).