1.1) The worst-case runtime of reverse1(lst) is $\Theta(n^2)$.

Inserting at the front of a list takes $\Theta(n)$ time for an n-element list. Thus the total runtime is

$$1 + 2 + 3 + 4 \ldots + (n-2) + (n-1) + n = \frac{n^2+n}{2}$$

or $\Theta(n^2)$

1.2) While append is not a constant time operation in the worst-case scenario, by amortization, we can show that consecutively appending n elements to an empty list takes $\Theta(n)$ time, so the worst-case runtime is $\Theta(n)$

2c. [1 time credit]

When a list of size $\frac{n}{2}$ is resized to size n, this takes $\Theta(n)$ time. However, it also guarantees that the next $\frac{n}{2}$ appends to the list will not require the list to be resized.

$$T = \underbrace{1}_{\text{capacity set 1}} + \underbrace{1}_{\text{add ele1}} + \underbrace{2}_{\text{resize to 2}} + \underbrace{1}_{\text{add ele2}} + \underbrace{4}_{\text{resize to 4}} + \underbrace{1}_{\text{add eles 3\&4}} + \underbrace{1}_{} \ldots + \underbrace{n}_{\text{resize to n}} + \underbrace{1 + 1 + \ldots + 1}_{\text{add another n/2 elems}}$$

Splitting this up we find

time spent resizing $= 1 + 2 + 4 \ldots + \frac{n}{2} + n = 2n - 1 = \Theta(n)$

time spent assigning $= \underbrace{1 + 1 + 1 + 1 \ldots + 1}_{n \text{ ones}} = n = \Theta(n)$

$\Theta(n) + \Theta(n) = \Theta(n)$, so n appends take $\Theta(n)$ time. This is true of pop too, which similarly assigns $n/2$ elements to none, before taking n time to shrink the list.

2c.2

The following sequence of appends & pops:
($\frac{n}{2}$ appends), append(), pop(), pop(), append(), ($\frac{n}{2}$ pops).
repeat $n/4$ times

append resizes $\frac{n}{2}$-length array to length n in $\Theta(n)$ time, then adds the $(\frac{n}{2}+1)^{th}$ element.

↓

pop removes the added element, leaving $\frac{n}{2}$ elements remaing. This occurs in $\Theta(1)$ time.

↓

pop again, leaving $\frac{n}{2}-1$ elements. $\frac{n}{2}-1 \leq \frac{n}{2}$ so the list is resized to size $\frac{n}{2}$ in $\Theta(n)$ time

↓

append adds an element in $\Theta(1)$ time, making $\frac{n}{2}$ elements in an $\frac{n}{2}$-capacity array

repeat $\frac{n}{4}$ times

time spent appending: $\frac{n}{2} + \frac{n}{4}(n+1+n+1)$ + time spent popping: $\frac{n}{2} = \frac{n^2}{2} + \frac{3n}{2}$, which is lower bounded by $\Omega(n^2)$

3b)
This implementation of find_duplicates runs in O(n) time
O(1) + O(n) + O(n) O(1) + O(n) + O(1) = O(n)

```
1   def find_duplicates(lst):
2       n = len(lst)                          O(1)
3
4       count = [0] * n                       O(n)
5
6       for i in lst:                         O(n)
7           count[i] += 1
8
9       out = []                              O(1)
10
11      for i in range(len(count)):           O(n)
12          if count[i] > 1:
13              out.append(i)
14
15      return out                            O(1)
```

4a)
The given implementation runs in $O(n^2)$ time. In the worst-case scenario, all n elements will need to be removed. Each removal takes linear time, since .remove() shifts elements. While the amount of elements that need to be shifted will be removed as the right-side of the list is approached, the time for each removal creates the following sequence:

$n + (n - 1) + (n - 2) + (n - 3) \ldots + 4 + 3 + 2 + 1 = (n(n + 1))/2$
Thus, the implementation still runs in $O(n^2)$ time in the worst-case scenario,

4c)
The below implementation of find_duplicates runs in O(n) time
$O(n) + O(n) + O(n) + O(n) + O(1) = O(n)$

```
1   def remove_all(lst, value):
2
3       back = [0] * len(lst)                    O(n)
4       removed = 0
5
6       for i in range(len(lst)):                O(n)
7           back[i] = removed
8           if lst[i] == value:
9               removed += 1
10
11      for i in range(len(lst)):                O(n)
12          lst[i - back[i]] = lst[i]
13
14      for i in range(removed):                 O(n)
15          lst.pop()
16
17      return lst                               O(1)
```