Homework #3
Due by Monday 2/19, 11:55pm

Submission instructions:

a) You should turn in 6 files:
   - 3 '.py' files: one, with all the code related, to each of the questions 2-4. Name your files: 'YourNetID_hw3_q2.py', 'YourNetID_hw3_q3.py', and 'YourNetID_hw3_q4.py'.
   - A '.pdf' file with all your written answers. Name your file: 'YourNetID_hw3.pdf'

   Note: your netID follows an abc123 pattern, not N12345678.

b) You should submit your homework via Gradescope. For Gradescope's autograding feature to work:
   a. Name all functions and methods exactly as they are in the assignment specifications.
   b. Make sure there are no print statements in your code. If you have tester code, please put it in a "main" function and do not call it.
   c. Do not use import statements that require external files.

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


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

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.

Question 2:

Consider the implementation we made in class for `MyList`, and its extensions you did in the lab.

In this question, we will add two more methods to this class: the `insert` method and the `pop` method.

a) Implement the method `insert(self, index, val)` that inserts `val` before `index` (shifting the elements to make room for `val`).

For example, your implementation should follow the behavior below:

```
>>> mlst = MyList()
>>> for i in range(1, 4+1):
...     mlst.append(i)
>>> mlst.insert(2, 30)
>>> mlst
[1, 2, 30, 3, 4]
```

Notes:

1. Make sure to double the capacity of the array, if there is no room for an additional element.
2. Your function should raise an `IndexError` exception in any case the index (positive or negative) is out of range.

b) Implement the method `pop(self)`, that removes the last element from the list. The `pop` method should return the element removed from the list, and put `None` in its place in the array. If `pop` was called on an empty list, an `IndexError` exception should be raised.

In order to maintain the linear memory usage of the list data structure, and its efficient amortized performance, we use the following shrinking strategy: When the number of elements in the list goes strictly below a quarter of the array's capacity, we shrink its capacity by half.

For example, your implementation should follow the behavior below:

```
>>> mlst = MyList()
>>> for i in range(1, 5+1):
...     mlst.append(i)
>>> mlst.pop()
5
>>> mlst.pop()
4
>>> mlst.pop()
3
>>> mlst.pop()
2
>>> mlst
[1]
```

Note: After the executing the code above, the capacity of the array in `MyList` will be 4.

c) *Extra Credit (You don't need to submit)*: The extending and shrinking strategies we use in our `MyList` implementation (doubling the capacity of the array each time there is no room to add an element, and shrinking the capacity of the array by a factor of 2 each time the number of elements is less than a quarter of the array's capacity), guarantees two important things:

i. In any given time, the memory used to store the elements is linear. That is, if there are *n* elements in the array, then: $n \leq (capacity\ of\ the\ array) \leq 4n$.

ii. Any sequence of *n* `append` and/or `pop` operations on an initially empty `MyList` object, takes *O(n)* time.

Proving these properties is out of the scope of this assignment, but we will show two supporting insights:

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.
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 `MyList` object, that requires $\Omega(n^2)$ time to execute.

d) *Extra Credit (You don't need to submit)*: Modify the `pop` method, so it could optionally get an index as a parameter. This index indicates the position of the element that is to be removed from the list.

Notes:

1. Make sure to use the same shrinking strategy described above in section (b).
2. Your function should raise an `IndexError` exception in any case the index (positive or negative) is out of range.

Question 3:

a) Give a **linear implementation** of the following function:

```
def find_duplicates(lst)
```

The function is given a list `lst` of *n* integers. All the elements of `lst` are from the domain: *{1, 2, …, n-1}*. Note that this restricted domain implies that there are element/s appearing in `lst` more than once.
The function should return a list containing all the elements that appear in `lst` more than once.

For example, if `lst=[2, 4, 4, 1, 2]`, the call `find_duplicates(lst)` could return `[2, 4]`.

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

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
```

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

b) Give an implementation to `remove_all` that runs in worst-case linear time.
Note: Your implementation should mutate the given list object (in-place).

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