NYU, Tandon School of Engineering

CS-UY 1134 Data Structures and Algorithms — Fall 2017

Homework #4
Due by Friday 10/13, 11:55pm

Submission instructions:

a) You should submit your homework in the NYU Classes system.

b) For this assignment you should turn in 2 files:
- One '.pdf' file for questions 3c, 4b, 5a and 5c. Name this file 'YourNetID_hw4.pdf'
- One '.py' file for the rest of the questions (1, 2, 3a-b, 4a and 5b). This file should contain all the functions and classes you were asked to implement in these question. Name this file 'YourNetID_hw4.py'

Question 1:

Give a **recursive** implement to the following function:

```
def split_by_sign(lst, low, high)
```

The function is given a list `lst` of non-zero integers, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should reorder the elements in `lst`, so that all the negative numbers would come before all the positive numbers.

Note: The order in which the negative elements are at the end, and the order in which the positive are at the end, doesn't matter, as long as all he negative are before all the positive.

Question 2:

Given an ordered list *L*. A ***permutation*** of *L* is a rearrangement of its elements in some order. For example *(1, 3, 2)* and *(3, 2, 1)* are two different permutations of *L=(1, 2, 3)*.

Implement the following function:

```
def permutations(lst, low, high)
```

The function is given a list `lst` of integers, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should return a list containing all the different permutations of the elements in `lst`. Each such permutation should be represented as a list.

For example, if `lst=[1, 2, 3]`, the call `permutations(lst, 0, 2)` could return `[[1, 2, 3], [2, 1, 3], [1, 3, 2], [3, 2, 1], [3, 1, 2], [2, 3, 1]]`

Hint: Use recursion! Think what the recursive call should return, and how to modify it in order to calculate the desired result.

<u>Question 3:</u>

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]
```

   <u>Notes</u>:

   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]
```

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

c) 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.

   <u>Notes</u>:
   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.

<u>Question 4:</u>

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: $\{1, 2, \ldots, n\text{-}1\}$. 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 5:

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.

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