Problem 1

Write a LinkedList class and a _Node class (note the leading underscore), representing a singly-linked, non-circular list. _Node should exactly 3 methods: __init__(val), get_next(), and get_val(). Note that this class does not allow users to change the value in a _Node; once set in __init__(val), this value is immutable.

LinkedList should have the following methods: __init__() (creates an empty list), add_head(val), remove_head(), peek_head(), is_empty(), and get_count(). Note that since _Node is a private class, the functions take and return values instead of node objects. This means that add_head(val) must construct a _Node object.

The “remove” and “peek” methods both return the value stored in the head of the list; the difference is that “remove” removes the node while returning, while “peek” keeps it on the list. Both the “remove” and “peek” must raise errors if the list is empty; discuss with your group if an assert or an exception is more appropriate, and write down your decision below. (Don’t forget to actually include the check in the code, too!)

Include a tail pointer in your LinkedList class. Update it in add_head(val); also change it to None, in remove_head(), if you remove the last element from the list.

(I’ve left the next page blank for your code.)
class _Node:
    def __init__(self, val):
        self._val = val
        self._next = None
    def get_next(self):
        return self._next
    def get_val(self):
        return self._val

class LinkedList:
    def __init__(self):
        self._head = self._tail = None
    def add_head(self, val):
        new = _Node(val)
        new._next = self._head
        self._head = new
        if self._tail is None:
            self._tail = self._head
    def remove_head(self):
        assert self._head is not None
        retval = self._head._val
        self._head = self._head._next
        if self._head is None:
            self._tail = None
        return retval
    def peek_head(self):
        assert self._head is not None
        return self._head._val
    def is_empty(self):
        return self._head is None
def get_count(self):
    count = 0
    cur = self._head
    while cur is not None:
        count += 1
        cur = cur._next
    return count

Problem 2(a)

Assume that you have the LinkedList and _Node classes from Problem 1. Write another method, for the LinkedList class, named add_sorted(val). This method will search through the list, and add the new value at the correct location (you want the head to be the least value, and the tail to be the greatest).

If the value already exists in the list, raise a ValueError exception. (NOTE: Some lists allow duplicates; others, like Python’s set class, allow you to add duplicates and silently ignore them. Not all lists throw an exception in this case - but you have to!)

# NOTE: we’re assuming that the list is sorted!
def add_sorted(self, val):
    if self._head is None or self._head._val > val:
        self.add_head(val)
        return
    cur = self._head
    while cur._next is not None and cur._next._val < val:
        cur = cur._next
    new = _Node(val)
    new._next = cur._next
    cur._next = new
    if new._next == None:
        self._tail = new
Problem 2(b)

What is the asymptotic (that is, big-Oh) time cost of `add_sorted(val)`? Explain.

O(n), because it has to scan through the elements of the list.

Suppose that you wanted to add n different values to the list, and so you simply called `add_sorted(val)` for each. What is the total asymptotic time cost of adding all of the values to the list?

O(n^2), because it does O(n) n many times.

Problem 3

Sometimes, instead of having a separate List class, we embed a list inside another, larger data structure. In this problem, write a `Person` class, with three fields (of your choosing), which are set in the `__init__()` method. Have a `_next` field in the class, with a `get_next()` getter method for it. No other methods are required.

Then, write a `ListOfPersons` class, which uses the `_next` fields in `Person` objects to form a linked list of `Person` objects. Implement an `__init__()` and `add_tail()` method; however, do NOT use a tail reference. (We’re doing this just for variety.)

Finally, add a `get_head()` method to `ListOfPersons`. It returns `None` if the list is empty.

class Person:
    def __init__(self, name, ssnum, favorite_color):
        self._name = name
        self._ssnum = ssnum
        self._favorite_color = favorite_color
        self._next = None
    def get_next(self):
        return self._next
class ListOfPersons:
    def __init__(self):
        self._head = None
    def add_tail(self, person):
        if self._head is None:
            self._head = person
            return
        cur = self._head
        while cur._next is not None:
            cur = cur._next
        cur._next = person
    def get_head(self):
        return self._head
Problem 4(a)

Finally, we will implement a list which has no wrapper class. Implement a ListNode class; this class represents both a single node, and also the list which begins at that node. The __init__() method must take exactly one parameter: the value (which cannot be changed after the node is created). Also a write a get_next() getter.

An empty list is represented by None. A list with one element is represented by a single ListNode object (which has a _next of None). A list with two elements is a pair of ListNode objects, with the first pointing at the second.

For this class, every function which modifies the list must return the new list:

- Write an add_head(val) method, which is a method of the ListNode class, and so is called on a certain ListNode object. It must:
  - Create a new ListNode object, containing the new value
  - Point the 'next' field of the new object at the old one
  - Return the new object
- Write a remove_head() method in the ListNode class. This method returns a tuple: the value and the new list. (If this remove_head() is called on a list with only one element, the "new list" is None.)
- Write an add_tail(val) method, which adds the value to the tail of the list:
  - Create a new ListNode object
  - Add the node to the tail of the list
  - Return the head of the list (which is the original self)

(I've left the next page blank, for you to write your code)
```python
class ListNode:
    def __init__(self, val):
        self._val = val
        self._next = None
    def get_next(self):
        return self._next
    def add_head(self, val):
        new = ListNode(val)
        new._next = self
        return new
    def remove_head(self):
        return (self._val, self._next)
    def add_tail(self, val):
        if self._next is None:
            self._next = ListNode(val)
        else:
            self._next = self._next.add_tail(val)
        return self

Problem 4(b)

add_tail(val) and add_head(val) are both methods of the ListNode class - so you can’t call them on an empty list. Discuss with your group how you would create the first element in a (previously empty) list, and write down what the group comes up with. (HINT: You don’t need any new methods.)

Finally, discuss with your group how to redesign add_tail() so that it might work with any list (even an empty one). What would need to change? Write down what you come up with.

The solution is to make many methods - at least add_head() and add_tail() - to be global functions instead of methods of the class. Then, instead of a ‘self’ parameter, we have ‘head’ parameter - and we will allow for the special case of head is None.
```