Question 1. [6 MARKS]

You are responsible for creating a class to represent an election riding. A riding has a name, and has voters, each identified by their social insurance number (we use single digits for simplicity in the example below). Once a voter has voted, they are not allowed to vote again. (Our simple class has no way to reset this.) Here is an example of how we want to use this class.

```
>>> r = Riding('Donlands')
>>> r.add_voters([5, 3, 9])
>>> r.add_voters([1, 2, 3, 4])   # Adding voter 3 again does nothing and raises no error.
>>> r.can_vote(2)
True
>>> r.can_vote(6)
False
>>> r.voted(2)
>>> r.can_vote(2)
False
```

Below and on the next page complete the implementation of this class so that the example code above will run as shown. You may choose any reasonable way to store the necessary data. Do not add anything to the public interface for this class beyond what is demonstrated in the example code.

Assume that the appropriate types have been imported from `typing`.

```
class Riding:
    """An election riding.
    === Attributes ===
    # TODO: Describe all instance attributes here.  Be clear and precise; this will
    # help us understand your code.









    """
    # TODO: Write type annotations for your attributes here.
```

```
    # TODO: Implement the initializer here.
    # The method header must include a type contract, but a docstring is NOT required.
```

```
    # TODO: Implement method add_voters here.
    # The method header must include a type contract, but a docstring is NOT required.
```

```
    # TODO: Implement method voted here.  You may assume a precondition that the
    # voter has already been added to the riding.
    # The method header must include a type contract, but a docstring is NOT required.
```

```
    # TODO: Implement method can_vote here.
    # The method header must include a type contract, but a docstring is NOT required.
```

Solution:

```
class Riding:
    """An election riding.

    _name: The name of the riding
    _voters: A dictionary mapping the social insurance number of a person
        eligible to vote in this riding to a boolean indicating whether or not
        they can vote in this riding.  Note to us: easy to get this bool
        backwards; include that in the grading scheme

    >>> r = Riding('Donlands')
    >>> r.add_voters([5, 3, 9])
    >>> r.add_voters([1, 2, 3, 4])
    >>> r.can_vote(2)
    True
    >>> r.can_vote(6)
    False
    >>> r.voted(2)
    >>> r.can_vote(2)
    False
    """
    _name: str
    _voters: Dict[int, bool]

    def __init__(self, name: str) -> None:
        self._name = name
        self._voters = {}

    def add_voters(self, voters: List[int]) -> None:
        """Add a list of voters to this Riding.  If a voter is already listed
        in this Riding, ignore them (do nothing to change the data about them.)

        Note to us: We could allow this to be used for resetting a voter, but
        that complicates the doctests too much, I think.
        """
        for voter in voters:
            if voter not in self._voters:
                self._voters[voter] = True

    def voted(self, voter: int) -> None:
        """Record the fact that <voter> has voted in this Riding.

        Precondition: <voter> is listed in this Riding.
        """
        self._voters[voter] = False

    def can_vote(self, voter: int) -> bool:
```

```
        if voter in self._voters:
            return self._voters[voter]
        else:
            return False
```

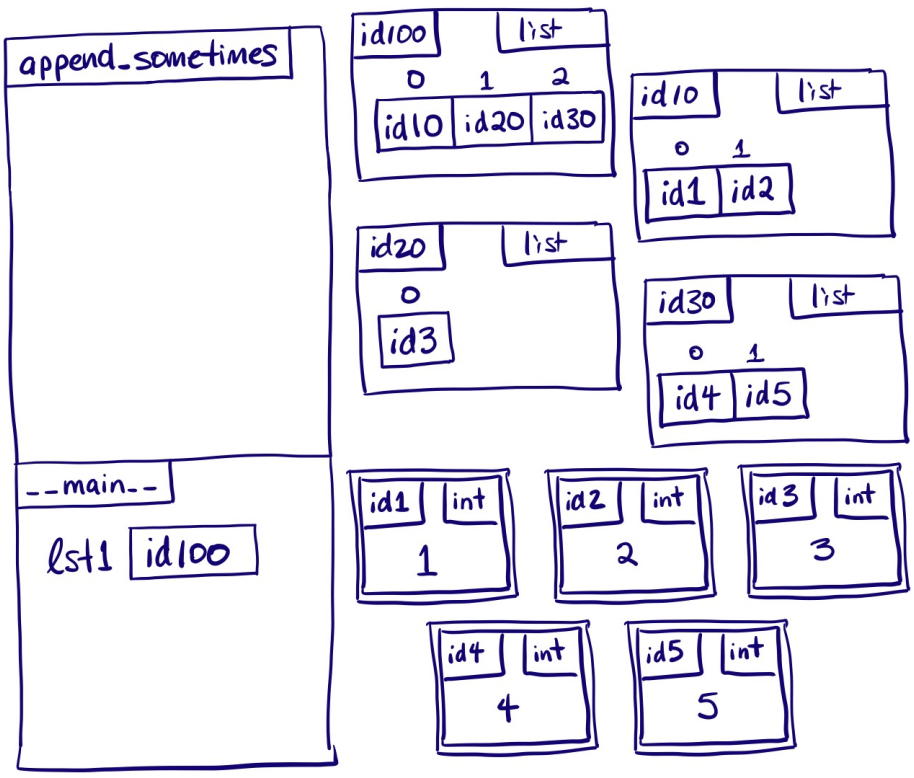
Question 2. [6 MARKS]

The function below has been implemented, but isn't working properly. The docstring is correct.

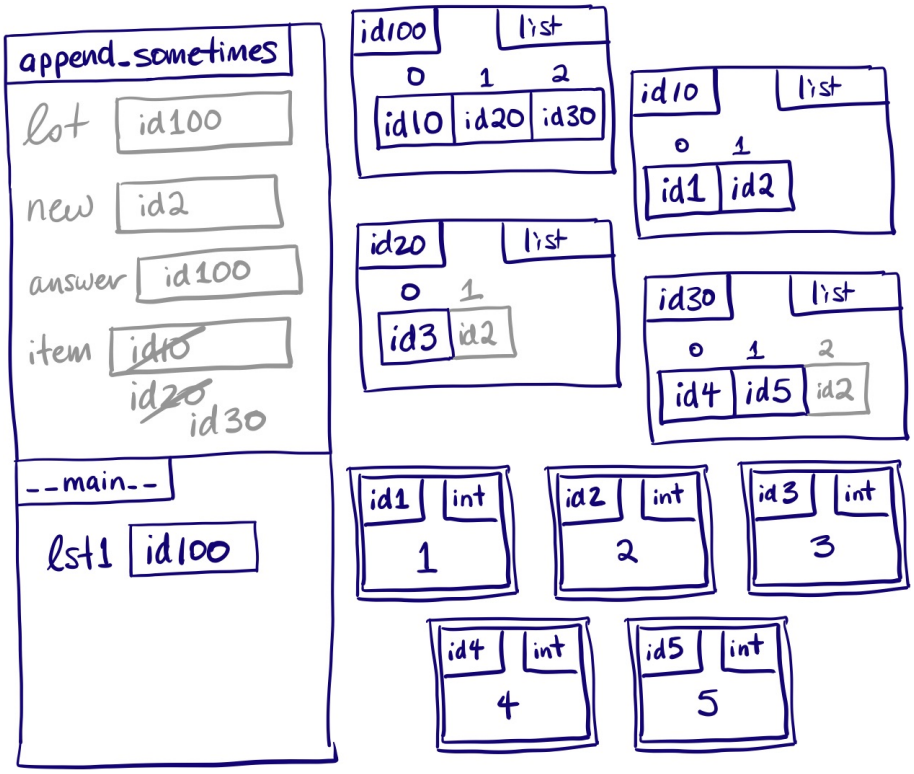
```
def append_sometimes(lst: List[List[int]], new: int) -> List[List[int]]:
    """Return a new list that is the same as <lst>, except that <new> has been
    appended to each sublist that didn't already contain it.

    >>> lst1 = [[1, 2], [3], [4, 5]]
    >>> new_lst1 = append_sometimes(lst1, 2)
    >>> new_lst1
    [[1, 2], [3, 2], [4, 5, 2]]
    >>> lst1
    [[1, 2], [3], [4, 5]]
    """
    answer = lst
    for item in answer:
        if new not in item:
            item.append(new)
    return answer
```

Part (a) [4 MARKS]

We have begun to trace the call to `append_sometimes` using the example from its docstring. Complete the diagram to show the state of memory immediately before `append_sometimes` returns. (We assume in the diagram that Python uses the shortcut that allows it to create aliases to the same immutable object.)



Solution:



Part (b) [1 MARK]

Which of the following doctests does the function pass? Circle pass or fail for each.

Solution:

```
>>> new_lst
[[1, 2], [3, 2], [4, 5, 2]]
```

[Pass] Fail

```
>>> lst
[[1, 2], [3], [4, 5]]
```

Pass [Fail]

Part (c) [1 MARK]

What is the problem with our code? Be specific and use correct terminology.

Solution:

The docstring does not say to mutate lst, so this should not happen. (And the docstest example makes this explicit.) But answer is defined to be an alias for lst. So when answer is mutated, so is lst — they are the same object.

Extra info (not required): To work, the function would need to make `answer` reference a copy of `lst`, copying both the top level and the nested lists.

Question 3. [4 MARKS]

Part (a) [2 MARKS]

From Assignment 1, name two classes that have a composition relationship: _____

Solution: There are many correct answers to this:

- CallHistory and Call,
- Contract and Bill,
- Customer and Phoneline,
- Phoneline and Contract
- Phoneline and Bill
- Phoneline and CallHistory

Where exactly in the code can you find proof that this is the intended relationship? Circle one answer.

Solution:

1. In the first line of one of these two classes (the line that says `class`).
2. In the first line of the other classes in the program.
3. In the parameters of the initializers for these two classes.
4. In the type annotations for the instance attributes of these two classes.

From Assignment 1, name two classes that have an inheritance relationship: _____

Solution: There are many correct answers to this:

- Contract and MTMContract (or Contract and any of the other types of contract).
- Filter and ResetFilter (or Filter and any of the other types of filter).

Where exactly in the code can you find proof that this is the intended relationship? Circle one answer.

Solution:

1. In the first line of one of these two classes (the line that says `class`).
2. In the first line of the other classes in the program.
3. In the parameters of the initializers for these two classes.
4. In the type annotation for the instance attributes of these two classes.

Part (b) [2 MARKS]

Recall that in class we worked on an abstract `Employee` class with two child classes, `SalariedEmployee` and `HourlyEmployee`. The following snippets of code all run without error. Which of them involve polymorphism? Circle Y or N for each.

Solution:

[Y] N
```
def f1(people: List[Employee]) -> None:
    for person in people:
        print(person)
```

[Y] N
```
def f2(people: List[Employee]) -> None:
    for person in people:
        person.pay(date(2019, 4, 16))
```

Y [N]
```
def f3(n: int) -> List[SalariedEmployee]:
    people = []
    for i in range(n):
        people.append(SalariedEmployee(i, 'Human' + str(i), 1000))
    return people
```

Y [N]
```
def f4(people: List[SalariedEmployee]) -> str:
    answer = ''
    for person in people:
        answer += str(person) + '\n'
    return answer
```

Question 4. [4 MARKS]

Recall the Stack class that we have used in the course. You can find an implementation of it on the provided aid sheet.

Suppose we want to define a new kind of Stack called a LimitedStack. It works like a regular Stack, except that it can hold at most k items. A LimitedStack has the same public interface as a regular Stack, except that this limit is provided when the LimitedStack is initialized. If a call to push is made when the number of items on the stack is already at its limit, nothing happens and no error is raised — the function merely returns. Otherwise, it behaves like a regular Stack.

Write this new class. Part of the mark will be for good design. In particular, avoid repeated code, even a single line.

You must write complete type contracts, but you do not need to write docstrings (but you can if you find it helpful!)

Solution:

```
class LimitedStack(Stack):
    """A Stack with a limited capacity.  Once it has reached its capacity,
    any new items that are to be pushed are simply not added to the stack.

    >>> s = LimitedStack(3)
    >>> s.push(11)
    >>> s.push(22)
    >>> s.push(33)
    >>> s.push(44)
    >>> s.pop()
    33
    """

    def __init__(self, limit: int) -> None:
        self.capacity = limit
        Stack.__init__(self)

    def push(self, item):
        """Push <item>, but only if this stack is not at capacity.
        """
        if len(self._items) < self.capacity:
            # Notice that we don't repeat the code for dequeuing, even though
            # it's simple.  We call the parent class to do it.
            Stack.push(self, item)
```

Question 5. [6 MARKS]

Write this function (outside any class) that takes a Stack and makes a Queue with the same items. It should also clear the Stack. The items should go into the Queue in such a way that the first one that went into the stack is the first one out of the queue. (Of course, any further items will go into the queue via the enqueue method, preserving this FIFO behaviour.)

You may make use of the Stack and Queue classes defined on the provided aid sheet to create temporary Stack and/or Queue objects, but you should not create any other new objects such as lists or dictionaries.

You must not access the instance attributes of any Stack or Queue. Use the public interface instead.

```
def make_queue_from(s: Stack) -> Queue:
    """Clear the Stack s, and return a Queue containing the items that were in s.
    Ensure that when the items are dequeued, the first item that went into the
    original Stack will be the first item out of the Queue.

    >>> nums = Stack()
    >>> nums.push(1)
    >>> nums.push(2)
    >>> nums.push(3)
    >>> q = make_queue_from(nums)
    >>> nums.is_empty()
    True
    >>> q.enqueue(4)
    >>> q.dequeue()
    1
    >>> q.dequeue()
    2
    >>> q.dequeue()
    3
    >>> q.dequeue()
    4
    >>> q.is_empty()
    True
    """
```

Solution:

```
    temp = Stack()
    while not s.is_empty():
        temp.push(s.pop())
    q = Queue()
    while not temp.is_empty():
        q.enqueue(temp.pop())
    return q
```

Question 6. [6 MARKS]

Part (a) [2 MARKS]

Say we have an arbitrarily long list, and we want to insert a new element in the middle. We are considering using a Python list or the LinkedList implementation on the aid sheet. Circle all of the statements below that are true. In all cases, assume we are talking about time in terms of big-oh.

Solution:

1. A Python list implementation will be faster than a LinkedList implementation.
2. A Python list implementation will be slower than a LinkedList implementation.
3. A Python list implementation will take about the same time as a LinkedList implementation.
4. A Python list implementation will have to access every item in the front half of the list.
5. A Python list implementation will have to access every item in the back half of the list.
6. A LinkedList implementation will have to access every node in the front half of the list.
7. A LinkedList implementation will have to access every node in the back half of the list.

Part (b) [2 MARKS]

Suggest a single new instance attribute we could add to our LinkedList implementation that would make inserting in the middle require fewer operations.

Does this new addition change the big-oh runtime of inserting into the middle of a LinkedList?

Circle one: Yes No

Does this new addition change the big-oh runtime of inserting **at the end** of a LinkedList?

Circle one: Yes No

Solution:

```
Answer option 1:
We could store a reference to the middle node of the linked list.
Yes, this would change the big-oh running time of inserting to the middle.
No, it would not change the big-oh running time of inserting at the end.

Answer option 2:
We could store the length of the linked list.
No, this would not change the big-oh running time of inserting to the middle.
No, it would not change the big-oh running time of inserting at the end.
```

Part (c) [2 MARKS]

Suppose we implemented the stack ADT using just one instance attribute: an instance of the LinkedList class as defined in the provided aid sheet. Which end of the stack should be at the front of the linked list?

Circle one: [The top of the stack] The bottom of the stack

Explain:

```
MARKING:

1 mark for circling "top of stack".
1 mark for a clear and correct answer.
    Nothing can be earned for this 1 mark if the circling part is incorrect.
```

Question 7. [6 MARKS]

The method below is being added to the LinkedList implementation as on the provided aid sheet. Assume the class also has a `__str__` method defined.

Fill in the boxes with the necessary code to complete the method according to its docstring. Do not add any code outside of the boxes. You must not create any new Node objects.

```
def reverse_nodes(self, i: int) -> None:
    """Reverse the nodes at index i and i + 1 by changing their next references
    (not by changing their items).

    Precondition: Both i and i + 1 are valid indexes in the list.

    >>> lst = LinkedList([5, 10, 15, 20, 25, 30])
    >>> print(lst)
    [5 -> 10 -> 15 -> 20 -> 25 -> 30]
    >>> lst.reverse_nodes(1)
    >>> print(lst)
    [5 -> 15 -> 10 -> 20 -> 25 -> 30]
    >>> lst = LinkedList([5, 10, 15, 20, 25, 30])
    >>> lst.reverse_nodes(0)
    >>> print(lst)
    [10 -> 5 -> 15 -> 20 -> 25 -> 30]
    >>> lst = LinkedList([5, 10, 15, 20, 25, 30])
    >>> lst.reverse_nodes(4)
    >>> print(lst)
    [5 -> 10 -> 15 -> 20 -> 30 -> 25]
    """
    if i == 0:  # special case of reversing the first two nodes
        temp = self._first
        self. [          ] = [          ]
        temp. [          ] = [          ]
        self. [          ] = [          ]
    else:  # general case
        # find the node before the pair to reverse
        curr = [          ]
        # traverse to the node at index i - 1
        for unused_ in range(i - 1):
            curr = [          ]
        # reverse the pair of nodes
        temp = curr.next
        curr. [          ] = [          ]
        temp. [          ] = [          ]
        curr. [          ] = [          ]
```

Solution:

```
if i == 0:
    temp = self._first
    self._first = self._first.next
    temp.next = self._first.next
    self._first.next = temp
else:
    curr = self._first
    for _ in range(i - 1):
        curr = curr.next
    temp = curr.next
    curr.next = curr.next.next
    temp.next = curr.next.next
    curr.next.next = temp
```

Question 8. [6 MARKS]

Write the body of the following recursive function according to its docstring. You can (and should!) use the Recursive code template from the provided aid sheet as a starting point.

```
def all_greater_than(obj: Union[int, List], n: int) -> bool:
    """Return True iff all the items in <obj> are greater than n.

    >>> all_greater_than(13, 10)
    True
    >>> all_greater_than(13, 40)
    False
    >>> all_greater_than([[1, 2, 3], 4, [[5]]], 0)
    True
    >>> all_greater_than([[1, 2, 3], 4, [[5]]], 3)
    False
    """
```

Solution:

```
    if isinstance(obj, int):
        return obj > n
    else:
        for sublist in obj:
            if not all_greater_than(sublist, n):
                return False
        return True
```