# Lab 5: Linked Lists

## 1) Practice with linked lists

For this task: we have commented out the doctests in the methods. You will not be able to run them until you finish step (3) of this task, at which point you may uncomment them. We recommend you read all of the steps in this task before you begin.

1. In the starter code, find and read the docstring of the method *__len__*, and then implement it.

   You already implemented this method in this week's prep, but it's good practice to implement it again. (And if you missed this week's prep, do it now!)

2. Then, implement the methods *count*, *index*, and *__setitem__*.

3. You might have noticed that all the doctests were commented out in the previous part.

   This is because they use a more powerful initializer than the one we've started with.

   Your final task in this section is to implement a new initializer with the following interface:

```
        def __init__(self, items: list) -> None:
            """Initialize a new linked list containing the given items.

            The first node in the linked list contains the first item
            in <items>.
            """

```

   The lecture notes suggest one way to do this using *append*; however, here we want you to try doing this without using *append* (or any other helper method).

   There are many different ways you could implement this method, but the key idea is that you need to loop through *items*, create a new *_Node* for each item, link the nodes together, and initialize *self._first*.

   Spend time drawing some pictures before writing any code!

# 2) Timing *__len__* for linked lists vs. array-based lists

1. Most methods take longer to run on large inputs than on small inputs, although this is not always the case.

   Look at your code for your linked list method *__len__*.

   Do you expect it to take longer to run on a larger linked list than on a smaller one?

2. Pick one the following terms to relate the growth of *__len__*'s running time vs. input size, and justify.

   - constant, logarithmic, linear, quadratic, exponential

3. Complete the code in *time_lists.py* to measure how running time for your *__len__* method changes as the size of the linked list grows. Is it as you predicted?

   Now's let's assess and compare the performance of Python's built-in *list*. You can do this by simply adding it to the list of types that *list_class* iterates over. What do you notice about the behaviour of calling *len* on a built-in *list*?

# 3) Augmenting our linked list implementation

It makes sense that our implementation of *LinkedList.__len__* is so slow; but how is the built-in *list.__len__* so much faster?

It turns out that built-in Python lists use an additional attribute to store their length, so that whenever *list.__len__* is called, it simply returns the value of this attribute.

The process of adding an extra attribute to an existing data structure is known as augmentation, and is very common in computer science. Every data structure augmentation poses a question of trade-offs:

- The benefit of augmenting is that the extra attribute makes certain operations simpler and/or more efficient to implement.
- The cost of augmenting is that this extra attribute increases the complexity of the data structure implementation.

  In particular, such attributes often have representation invariants associated with them that must be maintained every time the data structure is mutated.

1. Create a copy of your *LinkedList* class (you can pick a name for the copy), and add a new private attribute *_length* to the class documentation and initializer.

   Write down a representation invariant for this new attribute; you can use English here, but try to be precise without using the word "length" in your description. (Hint: how do we define length in terms of the nodes of a list?)

2. Update each mutating method to preserve your representation invariant for this new attribute. (Why don't we need to worry about the non-mutating methods?)

3. Now let's enjoy the benefit of this augmentation!

   Modify your new class' *__len__* method to simply return this new attribute.

   *Use doctests wisely to ensure you've made the correct changes for this and the previous step.*

4. Finally, perform some additional timing tests to demonstrate that you really have improved the efficiency of *__len__*.

# 4) Additional exercises

## Generalizing *__getitem__*

The implementation we've provided for *__getitem__* has many shortcomings compared to Python's built-in lists.

Two features that it doesn't currently support are negative indexes and slices (e.g., *my_list[2:5]*).

Your first task here is to investigate the different ways in which Python supports these operations for built-in Python lists; you can do this by experimenting yourself in the Python console, or by doing some reading online.

Then, modify the linked list implementation of *__getitem__* so that it handles both negative indexes and slices.

Note that a slice in Python is actually a class: the expression *my_list[2:5]* is equivalent to *my_list.__getitem__(slice(2, 5))*.

Use *isinstance* to determine whether the input to *__getitem__* is an integer or a slice.

The fully general method signature of *__getitem__* should become:

```
    def __getitem__(self, index: Union[int, slice]) -> Union[Any,
   LinkedList]
```

Note: slicing should always return a new *LinkedList* object.

This means that for a given slice, you'll need to create a *LinkedList* and new *_Nodes* as well, in a similar manner to how you implemented the more powerful initializer at the end of Task 1.

## Matplotlib Practice

Use *matplotlib* to plot the results of your timing experiments, using the same approach as last week (See matplotlib section in lab 4).