Lab 5: Linked Lists Solution

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?

   > Yes, because in order to count the number of nodes in the linked list, it has to traverse through all the nodes.

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

   > The growth of *__len__*'s running time is **linear**.
   >
   > First, we need to calculate the number of steps taken by the loop.
   >
   > The code tells us that the loop starts at 0^{th} node and increments by one until $n-1^{th}$ node in the linked list.
   >
   > Using this fact, we can calculate the loop has
   >
   > $$n - 1 - 0 + 1 = n \tag{1}$$
   >
   > iterations.
   >
   > Because we know each iteration takes a constant time (1 step), we can conclude the loop takes total of

$$n \cdot 1 = n \tag{2}$$

steps.

Finally, adding the constant time operations outside of the loop (1 step), we can conclude the algorithm has total running time of $n + 1$, which is $\mathcal{O}(n)$.

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?

Yes, the output is showing the running time grows proportionally to the size of input, and this is what we have predicted.

```
[LinkedList] Size   1000: 0.00010701700000000092
[LinkedList] Size   2000: 0.00019254800000000072
[LinkedList] Size   4000: 0.00036997500000000155
[LinkedList] Size   8000: 0.0007837910000000059
[LinkedList] Size  16000: 0.0015089149999999996

```

```
"""CSC148 Lab 5: Linked Lists

=== CSC148 Winter 2020 ===
Department of Computer Science,
University of Toronto

=== Module description ===
This module runs timing experiments to determine how the time taken
to call `len` on a Python list vs. a LinkedList grows as the list size grows.
"""
from timeit import timeit
from linked_list import LinkedList

NUM_TRIALS = 3000                         # The number of trials to run.
SIZES = [1000, 2000, 4000, 8000, 16000]  # The list sizes to try.


def profile_len(list_class: type, size: int) -> float:
    """Return the time taken to call len on a list of the given class and size.

    Precondition: list_class is either list or LinkedList.
    """
```

```
        # TODO: Create an instance of list_class containing <size> 0'
    s.
        my_list = LinkedList([0 for x in range(size)])

        # TODO: call timeit appropriately to check the runtime of len
     on the list.
        # Look at the Lab 4 starter code if you don't remember how to
     use timeit:
        # https://www.teach.cs.toronto.edu/~csc148h/winter/labs/
    w4_ADTs/starter-code/timequeue.py

        time = timeit('len(my_list)', number=1, globals=locals())

        return time


    if __name__ == '__main__':
        for list_class in [LinkedList]:
            # Try each list size
            for s in SIZES:
                time = profile_len(list_class, s)
                print(f'[{list_class.__name__}] Size {s:>6}: {time}')

```

Listing 1: task_2_step_3_solution.py

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*?

By using built-in list, I noticed that the time it takes for *__len__* is infinitesimally small, and it doesn't grow with size.

```
    [LinkedList] Size   1000: 1.1270000000035418e-06
    [LinkedList] Size   2000: 8.170000000001787e-07
    [LinkedList] Size   4000: 7.779999999998899e-07
    [LinkedList] Size   8000: 7.909999999999862e-07
    [LinkedList] Size  16000: 1.448000000001115e-06

```