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?

   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*?