CSC148 - Analysing the Running Time of Linked List Operations

Last week, we began to look at how to analyse the efficiency of Python code, first by counting "steps", and then using Big-Oh notation. On this worksheet, we'll first review what you know about the efficiency of operations for Python's `list` class, and then analyse our `LinkedList` class.

1. (*Review*) Suppose we have a Python (array-based) list of length n.

   (a) Complete the following table by using Big-Oh notation to describe the running time of each of the following operations. Remember: $\mathcal{O}(1)$ means "constant time", while $\mathcal{O}(n)$ means "linear time", both with respect to the list length n.

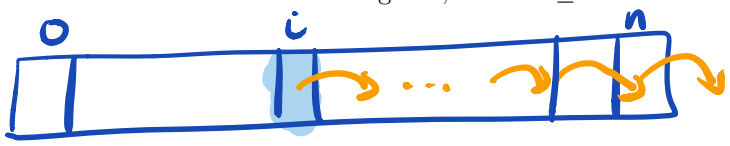
Operation	Running time
Insert at the front of the list	$O(n)$
Insert at the end of the list	$O(1)$
Look up the element at index i, where $0 \leq i < n$	$O(1)$

must shift all n elements over

Python keeps expansion room

items are contiguous

   (b) In general, Big-Oh notation means (roughly) "proportional to". We don't have to just use one variable n in Big-Oh notation! For example, $\mathcal{O}(n + i)$ translates to "proportional to $n + i$".
   Using this idea, write down a Big-Oh expression to capture the running time of the following operation: inserting a new item at index i into a list of length n, where $0 \leq i < n$.



$O(n-i)$

2. Now let's look at our linked list implementation. The key idea behind determining the efficiency of linked list operations is counting *how many nodes are traversed during the operation.*

   (a) Review your implementations of `LinkedList.append`, `LinkedList.insert`, and `LinkedList.__getitem__`. Then, complete the following table, assuming a linked list of length n.

Operation	Running time
Insert at the front of the linked list	$O(1)$
Insert at the end of the linked list	$O(n)$
Look up the element at index i, where $0 \leq i < n$	$O(n)$

no shifting! fixed amount of work

hop to end + fixed amount to insert

must hop

   (b) What is the Big-Oh running time of inserting an item at index i into a linked list of length n, where $0 \leq i < n$?

$O(i)$ hop to the spot fixed amount of work to update.

or $O(i)$ where $i \in n$

3. Suppose we have an array-based list of length 1,000,000, and a linked list of length 1,000,000. If we insert a new item at index 500,000 into each list, would it be:

   - significantly faster for the array-based list, or
   - significantly faster for the linked list, or
   - roughly the same amount of time for both lists?

Explain your answer!

	get to the middle	insert there
pythonlist	$O(1)$	shift 500,000
linked list	hop over ~ 500,000	$O(1)$

4. Consider this implementation of `LinkedList.__init__` from the lecture notes:

```
class LinkedList:
    def __init__(self, items: list) -> None:
        self._first = None
        for item in items:
            self.append(item)
```

✓ simple

(a) Suppose we call `LinkedList(items)` where `items` has length n. Calculate the total number of nodes traversed when we make this call. Note that the same node can be traversed more than once, and you should count *each time* the node is traversed. (You may or may not include "1 step" for creating each new node.)

when we add	the LL has this many node to traverse
items[0]	0
items[1]	1
items[2]	2
⋮	
items[n-1]	n-1

0+1+2 ... (n-1)

(b) Based on your calculation in the previous part, write down a Big-Oh expression for the running time of this operation in terms of n, the length of the input `items`.

$O(n^2)$

$O(n)$ is very possible.

different scenario, same list size

5. The last topic we'll cover today is looking at how running time can vary, even among inputs of the same size.

For example, consider `LinkedList.__contains__`, and suppose we have a linked list `lst` of length $n \geq 1$, and we are searching for the item 148. That is, we call `lst.__contains__(148)`.

(a) When would it be possible for `lst.__contains__(148)` to return after visiting just a single node?

if 148 is in the 1st node.

(b) When would it be necessary for `lst.__contains__(148)` to visit all n nodes in the linked list `lst`?

if 148 is in the last node. (+ not before)

if 148 is not in the LL.

worst case.

best case. ~ less interesting

average case ~ tricky.] csc263.