Module 5 Homework - Recursion

Part 1 - Recursion on a Linked List

Linked lists can be thought of as recursive structures - every node is really the "head" of a sub-linked list. This makes them a great playground for implementing various types of recursive functions. We provide starter code that demonstrates a few methods of recursion in a LinkedList, and you will implement a few more.

Starter Code

The `LinkedList` class is already fully implemented. It is essentially a wrapper - the `LLNode` class does all the interesting recursion.

In `LLNode`, we have implemented:

- `init`
- `__len__()`
- `__repr__()`
- `get_tail()`

`__len__()`, `__repr__()`, and `get_tail()` show various examples of recursive `LLNode` functions. Study them to understand how they work, then implement the following. Note that we have listed the parameters you should use - this is a great hint into how you can efficiently structure your code. Do not change these.

- `add_last(self, item)`
  - recursively call on linked node until you hit the tail, at which point you should add a new node
  - we are not keeping track of a tail node, so this is $\mathcal{O}(n)$
- `sum_all(self)`
  - recursively add all items in linked list and return the sum
  - $\mathcal{O}(n)$
- `reverse(self, prev)`
  - recursively calls itself on linked node until it gets to the tail
  - tail node updates it's link (to whatever was passed in as `prev`), then returns itself
  - every other node updates its link, then return **the old tail**
  - $\mathcal{O}(n)$

Part 2 - Recursion for branching paths

Model a linear board game consisting of numbered tiles using a list. The numbers represent how many tiles you can move forwards (FW) or backwards (BW). The goal is to reach the final tile (the tile at index `n-1`) in as few moves as possible.

For instance, we could solve the following puzzle by only moving forwards in 9 moves, but we could solve it with a strategic backwards step in just 4:

```
#idx: 0  1   2  3  4  5  6  7  8  9 10 11 12
L =  [1, 3, 10, 4, 2, 1, 1, 1, 1, 1, 1, 1, 1]

# nonoptimal path: [0, 1, 4, 6, 7, 8, 9, 10, 11, 12]
# optimal path:    [0, 1, 4, 2, 12]
```

Write a function that takes a list representing such a board as input, and returns a tuple of:

- a list representing the optimal path
- an integer representing the number of moves required.

Note that we do not count the starting tile towards our number of moves (it takes 0 moves to solve a puzzle that is only 1-item long).

```
>>> find_optimal_path([1, 3, 10, 4, 2, 1, 1, 1, 1, 1, 1, 1, 1])
[0, 1, 4, 2, 12], 4
```

If the puzzle is not solveable, return `None, None`.

Imports

No imports allowed on this assignment, with the following exceptions:

- Any modules you have written yourself
- `typing` - this is not required, but some students have requested it
- For testing only (do not use these for functionality in any other classes/algorithms):
  - unittest
  - random

Submission Requirements

Students must submit **individually** by the due date (Tuesday, October 10th at 11:59 pm EST).

At a minimum, you must submit the following files:

- `LinkedList.py`
- `SolvePuzzle.py`

Additionally, you must include any other files necessary for your code to run, such as modules containing data structures you wrote yourself.

Note that we are not requiring tests here. We have intead provided tests for you. We'll begin requiring tests again for the next homework, but we are prioritizing giving you multiple chances to practice recursion this week, since it's an area where we typically need a lot of practice. See here for some more practice problems.

You will still be manually graded on structure, readability, and correctness. Autograder scores are subject to review while manual grading. Egregiously poor structural descions, non-recursive solutions, or brute forcing unittests with something like

```
def find_optimal_path(L):
    return [0, 1, 4, 2, 12], 4
```

will not recieve any credit.