Mod 5 Homework - Recursion

Problem 1 - `trib.py`

Write a recursive function `trib(k)` that returns the kth tribonacci number. The tribonacci series (wiki) is a modification of the fibonacci series where each number is equal to the sum of the three previous numbers, rather than the sum of the previous two. We need to define the first three numbers of the series explicitly:

$$T_k = \begin{cases} 0 & k = 1 \\ 0 & k = 2 \\ 1 & k = 3 \\ T_{k-1} + T_{k-2} + T_{k-3} & k > 3 \end{cases}$$

You'll need to use memoization to avoid an exponential running time. **Do not use mutable objects, like empty lists or dictionaries, as default values**. Python only creates default values once, so all calls to the function use the same default objects - modifications to these mutables by one call to this function will still be there the next time you call it.

While this won't raise any errors here, since we are finding a fixed series, it will be the source of very tough to find bugs down the road. As good practice, we should always use a helper function instead of a mutable default, like `_trib` below:

```
# BAD - all calls to trib share the same default dictionary
def trib(k, solved=dict()):
    # Recursive work here

# GOOD - a new empty dictionary is created every time trib is called
def trib(k):
    return _trib(k, dict())

def _trib(k):
    # Recursive work here
```

`TestTrib.py`

Two unittests will suffice here.

1) Create a dictionary of the first 10 tribonacci numbers, iterate through it, and verify that your function gives the correct result for each:

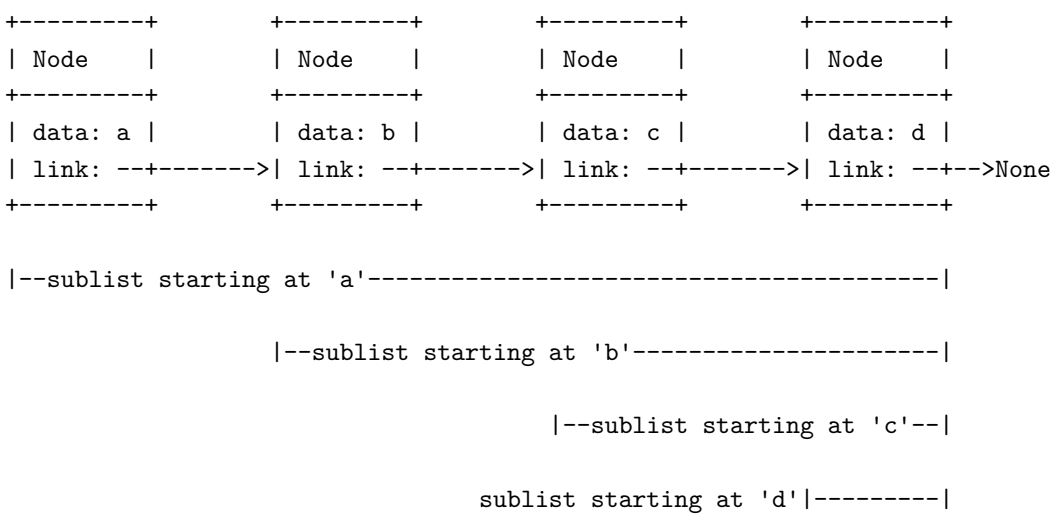
```
def test_first_ten(self):
    """Tests the first 10 numbers in the tribonacci series"""
    solutions = {1:???, 2:???, 3:???, ...} # Frist 10 tribs

    for k in solutions:
        self.assertEqual(trib(k), solutions[k])
```

2) The 100th item in the tribonacci series is `28,992,087,708,416,717,612,934,417` ($\sim 2.9 \times 10^{25}$). Write a unittest to ensure `trib(100)` returns this number. For easy copy/pasting, the number without commas is `28992087708416717612934417`.

Problem 2 - Implement a linked list with a recursive Node class

Linked list have elegant recursive implementations. We can think of each node as the head of a sublist:



Our general recursive algorithm in a linked list Node is:

1) Base case: this is the tail, return correct value or values
2) This is not the tail. Recursively call this function, and then return what that call returns

For instance, we can recursivley implement `__len__` in the `Node` class as:

```
def __len__(self):
    """Recursively calculates length of this sublist"""
    # Base case: this sublist has a length of 1
    if self.link is None: return 1

    # Recursive case: this sublist has a length of 1 + the length of its link
    return 1 + len(self.link)
```

Implement the following functionality **recursively** in the `Node` class in `recursive_node.py`:

- `get_tail(self)` - returns the data stored in the tail node
- `add_last(self, data)` - Adds a node containing `data` to the end of this list. Does not need to return anything.
- `total(self)` - Recursively calculates the sum of all items in list
- `remove_last(self)` - Removes the last node from this list, and returns the correct data. This function should always return a tuple of two values: the "new head" of this sublist, and the data from the tail.

  The "new head" should be `None` if this node is the tail, and `self` otherwise.
- `reverse(self, prev)` - Reverses the linked list. Every "link" should be redirected, and the final "head" of the list should be what was originally the "tail". `prev` is the "previous" node in the original linked list. We pass ourselves as prev to our link by calling `self.link.reverse(self)`, which is equivalent to `Node.reverse(self=self.link, prev=self)`.

  As with `remove_last`, we need to return the new head of this sublist. Unlike 'remove_last", the new head will always be the same node - the original tail gets passed back through every level of recursion.

We have provided a `RecurisveLinkedList` class that implements all the necessary methods from the `LinkedList` perspective. All of your work should be in `Node`.

You do not need to write your own test cases - we have provided a full suite of tests in `TestLinkedList.py`.

Note that all 4 methods above **must be implemented recursively** for you to get credit on this assignment. We will manually deduct points for non-recursive implementations that pass test cases.

Grading

As always, ensure:

- Every method and unittest has a good docstring
- Names are descriptive and consistent
- Whitespace is used to improve readability

`TestTrib.py` will be manually graded.

We will also remove any points earned for `recursive_node.py` by non-recursive methods.

Submission

At minimum, submit the following files with the classes and unittests listed. See above for more in-depth descriptions of each tests.

- `trib.py`
- `TestTrib.py`
- `recursive_node.py`
  - Contains `Node` class

Students must submit individually by the deadline (usuall Tuesdays at 11:59 PM EST) to recieve credit.