3. Complete the recursive `strHelper` function in the `__str__` method for our `OrderedList` class.

```python
def __str__(self):
    """Returns a string representation of the list with a space between each item. """
    ...
    return " + strHelper(current, getdata(current))"

return " + strHelper(self.head) + "(tail)"
```

4. Some mathematical concepts are defined by recursive definitions. One example is the Fibonacci series:

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots \]

After the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined recursively as:

- \( \text{Fib}_0 = 0 \)
- \( \text{Fib}_1 = 1 \)
- \( \text{Fib}_n = \text{Fib}_{n-1} + \text{Fib}_{n-2} \) for \( n \geq 2 \).

a) Complete the recursive function:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

b) Draw the call tree for `fib(5)`. 

![Call Tree for fib(5)](call_tree.png)
c) On my office computer, the call to fib(40) takes 22 seconds, the call to fib(41) takes 35 seconds, and the call to fib(42) takes 56 seconds. How long would you expect fib(43) to take?

d) How long would you guess calculating fib(100) would take on my office computer?

e) Why do you suppose this recursive fib function is so slow?
   redundant calculation

f) What is the computational complexity? $O(2^n)$ bad

g) How might we speed up the calculation of the Fibonacci series?

store answers on first calculation and then lookup as needed

5. A VERY POWERFUL concept in Computer Science is **dynamic programming**. Dynamic programming solutions eliminate the redundancy of divide-and-conquer algorithms by calculating the solutions to smaller problems first, storing their answers, and looking up their answers if later needed instead of recalculating them.

We can use a list to store the answers to smaller problems of the Fibonacci sequence.

To transform from the recursive view of the problem to the dynamic programming solution you can do the following steps:

1) Store the solution to smallest problems (i.e., the base cases) in a list
2) Loop (no recursion) from the base cases up to the biggest problem of interest. On each iteration of the loop we:
   - solve the next bigger problem by looking up the solution to previously solved smaller problem(s)
   - store the solution to this next bigger problem for later usage so we never have to recalculate it

a) Complete the dynamic programming code:

```python
def fib(n):
    """Dynamic programming solution to find the nth number in the Fibonacci seq.""
    # List to hold the solutions to the smaller problems
    fibonacci = [0, 1]

    # Step 1: Store base case solutions
    fibonacci.append(0)
    fibonacci.append(1)

    # Step 2: Loop from base cases to biggest problem of interest
    for position in range(2, n + 1):
        fibonacci.append(fibonacci[position - 1] + fibonacci[position - 2])

    # return nth number in the Fibonacci sequence
    return fibonacci[n]
```

Running the above code to calculate fib(100) would only take a fraction of a second.

b) One tradeoff of simple dynamic programming implementations is that they can require more memory since we store solutions to all smaller problems. Often, we can reduce the amount of storage needed if the next larger problem (and all the larger problems) don’t really need the solution to the really small problems, but just the larger of the smaller problems. In fibonacii when calculating the next value in the sequence how many of the previous solutions are needed? two
1. Consider the coin-change problem: Given a set of coin types and an amount of change to be returned, determine the fewest number of coins for this amount of change.

a) What "greedy" algorithm would you use to solve this problem with US coin types of \{1, 5, 10, 25, 50\} and a change amount of 29-cents?

\[
\begin{align*}
29 & \quad \downarrow \\
-25 & \\
-1 & \\
-3 & \\
-1 & \\
-1 & \\
\hline
5 & \quad \text{coin solution}
\end{align*}
\]

b) Do you get the correct solution if you use this algorithm for coin types of \{1, 5, 10, 12, 25, 50\} and a change amount of 29-cents?

\[
\begin{align*}
29 & \quad \downarrow \\
-25 & \\
-4 & \\
-1 & \\
\hline
5 & \quad \text{coin solution with greedy alg.}
\end{align*}
\]

Better: 3 coin solution: 12, 12, 5

2. One way to solve this problem in general is to use a divide-and-conquer algorithm. Recall the idea of **Divide-and-Conquer** algorithms.

Solve a problem by:
- dividing it into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

a) For the coin-change problem, what determines the size of the problem?

\[
\begin{align*}
29 & \quad \downarrow \\
\text{Changes} & \quad \# \text{coins}
\end{align*}
\]

b) How could we divide the coin-change problem for 29-cents into smaller problems?

\[
\begin{align*}
29 & \quad \downarrow \\
\text{Changes} & \quad \# \text{coins}
\end{align*}
\]

\[
\begin{align*}
29 & \quad \downarrow \\
\text{Changes} & \quad \# \text{coins}
\end{align*}
\]

b) How could we divide the coin-change problem for 29-cents into smaller problems?

c) If we knew the solution to these smaller problems, how would we be able to solve the original problem?
3. After we give back the first coin, which smaller amounts of change do we have?

Original Problem

\[ \begin{align*}
\text{Possible First Coin} & : + 1 \\
29 \text{ cents} & \quad 5-\text{cent coin} \quad 10-\text{cent coin} \quad 25-\text{cent coin} \\
28 & \quad 24 & \quad 19 & \quad 17 & \quad 4 \\
\text{Smaller problems} & : 4-\text{coin} \quad 2-\text{coin} \quad 1-\text{coin} \\
\end{align*} \]

4. If we knew the fewest number of coins needed for each possible smaller problem, then how could we determine the fewest number of coins needed for the original problem?

\[ \min \left( \begin{array}{c}
\text{4-coin} \\
\text{2-coin} \\
\text{1-coin} \\
\text{Smaller problems}
\end{array} \right) + 1 \]

5. Complete a recursive relationship for the fewest number of coins.

\[ \text{FewestCoins}(\text{change}) = \begin{cases} 
\min( \text{FewestCoins}(\text{change} - \text{coin})) + 1 & \text{if change} \notin \text{CoinSet} \\
0 & \text{if change} \in \text{CoinSet and coin} \leq \text{change}
\end{cases} \]

6. Complete a couple levels of the recursion tree for 29-cents change using the set of coins \{1, 5, 10, 12, 25, 50\}.
<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

- 0th Floor
- 1st Floor
- 2nd Floor
- 3rd Floor
- 4th Floor

**Floor Plan**

- 0th Floor: Room 0
- 1st Floor: Rooms 1-3
- 2nd Floor: Rooms 4-6
- 3rd Floor: Rooms 7-9
- 4th Floor: Rooms 10-12

**Floor Levels**

- 0th Floor: Level 0
- 1st Floor: Level 1
- 2nd Floor: Level 1
- 3rd Floor: Level 1
- 4th Floor: Level 1

**Floor Areas**

- 0th Floor: 200 sq ft
- 1st Floor: 300 sq ft
- 2nd Floor: 400 sq ft
- 3rd Floor: 500 sq ft
- 4th Floor: 600 sq ft

**Floor Plan Details**

- 0th Floor: Entryway, Hallway, Kitchen, Living Room
- 1st Floor: Bedrooms, Bathrooms, Office
- 2nd Floor: Playroom, Study, Guest Room
- 3rd Floor: Master Suite, Home Gym, Laundry
- 4th Floor: Roof Deck, Storage, Media Room

**Floor Plans**

- 0th Floor Plan
- 1st Floor Plan
- 2nd Floor Plan
- 3rd Floor Plan
- 4th Floor Plan

**Floor Features**

- 0th Floor: Hardwood floors, Central Heating
- 1st Floor: Granite Countertops, Stainless Steel Appliances
- 2nd Floor: Skylights, Ceiling Fans
- 3rd Floor: En-suite Bathrooms, Walk-in Closets
- 4th Floor: Rooftop Deck, Outdoor Kitchen

**Floor Layouts**

- 0th Floor Layout
- 1st Floor Layout
- 2nd Floor Layout
- 3rd Floor Layout
- 4th Floor Layout

**Floor Notes**

- 0th Floor: Access to Basement
- 1st Floor: Access to Garage
- 2nd Floor: Access to Terrace
- 3rd Floor: Access to Deck
- 4th Floor: Access to Roof Garden