3. The non-recursive \_str\_ method for \_orderedList\_ object below would return: "(head) a c m (tail)"

```python
def \_str\_(self):
    resultStr = "(head) "
    current = self._head
    while current != None:
        resultStr += str(current.getData()) + " "
        current = current.getNext()
    return resultStr + "(tail)"
```

We can think of building the string for the list as (a) + (string for the rest of the list)

a) Complete the recursive strHelper function in the \_str\_ method for our \_orderedList\_ class.

```python
def strHelper(current):
    if current == None:
        return """"
    else:
        return str(current.getData()) + " " + strHelper(current.getNext())
```

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, ...
```

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:

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

a) Complete the recursive function:

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

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

Why do you suppose this recursive fib function is so slow?

 Millions of years
  a lot of redundant calculations

What is the computational complexity? $O(2^n). 2^n = 2^{n-1} \times 2$

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

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:

- Store the solution to smallest problems (i.e., the base cases) in a list
- 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
# fib(n):
# Dynamic programming solution to find the nth number in the Fibonacci sequence.

# List to hold the solutions to the smaller problems
fibonacci = []

# 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 fibonacci when calculating the next value in the sequence how many of the previous solutions are needed?**

```
previous 1 | 1 | 2 | 3 | 5 | 8
current 0 | 1 | 2 | 3 | 5 | 8
next 54
```

$O(1)$ storage
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?

   \[
   \begin{array}{ccc}
   29 & 1 & 5 \\
   25 & 4 & 1 \\
   10 & 3 & 1 \\
   5 & 2 & 0 \\
   \end{array}
   \]

   Solution

   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?

   \[
   \begin{array}{ccc}
   29 & 1 & 5 \\
   25 & 4 & 1 \\
   12 & 3 & 0 \\
   \end{array}
   \]

   Solution

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?

- smaller change amount
- subsets of coins
- combination of both

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 be able to solve the original problem?

\[
\begin{array}{ccc}
29 & 28 & 20 \\
19 & 17 & 4 \\
10 & 12 & 25 \\
5 & 1 & 50 \\
\end{array}
\]
3. After we give back the first coin, which smaller amounts of change do we have?

**Original Problem**

```
29 cents
```

```
Possible First Coin: 1-cent coin, 5-cent coin, 10-cent coin, 25-cent coin, 50-cent coin
```

```
Smaller problems: 28, 24, 19, 17, 4
```

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

```
\text{FewestCoins}(\text{change}) = 1 + \min(\text{FewestCoins}(\text{change} - \text{coin}))
```

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

```
\text{FewestCoins}(\text{change}) = \begin{cases} 
 \text{min}(\text{FewestCoins}(\text{change} - \text{coin})) + 1 & \text{if change} \not\in \text{CoinSet} \\
 1 & \text{if coin} \in \text{CoinSet and coin} \leq \text{change} \\
 0 & \text{if change} = 0 
\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\}.

**Original Problem**

```
29 cents
```

```
Possible First Coin: 1-cent coin, 5-cent coin, 10-cent coin, 25-cent coin, 50-cent coin
```

```
Smaller problems: 28, 24, 19, 17, 4
```

```
Best solution so far: 10-cent solution
```

```
Bound on best
```

```
58
```

Lecture 10 - Page 2
1. The textbook solves the coin-change problem with the following code (note the “set-builder-like” notation):
   \[ \{ c | c \in \text{coinValueList} \text{ and } c \leq \text{change} \} \]
   
   ```python
   def recMC(change, coinValueList):
       global backtrackingNodes
       backtrackingNodes += 1
       minCoins = change
       if change in coinValueList:
           return 1
       else:
           for i in coinValueList:
               if i <= change:
                   numCoins = 1 + recMC(change - i, coinValueList)
                   if numCoins < minCoins:
                       minCoins = numCoins
       return minCoins
   ```

   I removed the fancy set-builder notation and replaced it with a simple if-statement check:
   
   ```python
   def recMC(change, coinValueList):
       global backtrackingNodes
       backtrackingNodes += 1
       minCoins = change
       if change in coinValueList:
           return 1
       else:
           for i in coinValueList:
               if i <= change:
                   numCoins = 1 + recMC(change - i, coinValueList)
                   if numCoins < minCoins:
                       minCoins = numCoins
       return minCoins
   ```

   Results of running this code:
   
   Change Amount: 63 Coin types: [1, 5, 10, 25]
   Run-time: 70.689 seconds
   Fewest number of coins 6
   Number of Backtracking Nodes: 67,716,925

   a) Why is the second version so much “faster”?

      only has single list of coins
      first builds new subset of coins

   b) Why does it still take a long time?

      redundant calculations

   2. To speed the recursive backtracking algorithm, we can prune unpromising branches. The general recursive backtracking algorithm for optimization problems (e.g., fewest number of coins) looks something like:

   ```python
   Backtrack( recursionTreeNode p ) {
       for each child c of p do
           if promising(c) then
               if c is a solution that's better than best then
                   best = c
               else
                   Backtrack(c)
           end if
       end if
   } // end Backtrack
   ```

   General Notes about Backtracking:
   
   - The depth-first nature of backtracking only stores information about the current branch being explored on the run-time stack, so the memory usage is “low” even though the # of recursion tree nodes might be exponential (2^n).
   - Each node of the search-space (recursive-call) tree maintains the state of a partial solution. In general the partial solution state consists of potentially large arrays that change little between parent and child. To avoid having 
     multiple copies of these arrays, a reference to a single “global” array can be maintained which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.

   a) For the coin-change problem, what defines the current state of a search-space tree node?

   ![Diagram of coin-change problem with tree structure]
b) When would a “child” tree node NOT be promising?
   
   - req. change amount

3. Consider the output of running the backtracking code with pruning (next page) twice with a change amount of 63 cents.

<table>
<thead>
<tr>
<th>Change Amount: 63 Coin types: [1, 5, 10, 25]</th>
<th>Change Amount: 63 Coin types: [25, 10, 5, 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time: 0.036 seconds</td>
<td>Run-time: 0.003 seconds</td>
</tr>
<tr>
<td>Fewest number of coins: 6</td>
<td>Fewest number of coins: 6</td>
</tr>
<tr>
<td>The number of each type of coins is:</td>
<td>The number of each type of coins is:</td>
</tr>
<tr>
<td>number of 1-cent coins is 3</td>
<td>number of 25-cent coins is 2</td>
</tr>
<tr>
<td>number of 5-cent coins is 0</td>
<td>number of 10-cent coins is 1</td>
</tr>
<tr>
<td>number of 10-cent coins is 1</td>
<td>number of 5-cent coins is 0</td>
</tr>
<tr>
<td>number of 25-cent coins is 2</td>
<td>number of 1-cent coins is 3</td>
</tr>
<tr>
<td>Number of Backtracking Nodes: 4831</td>
<td>Number of Backtracking Nodes: 310</td>
</tr>
</tbody>
</table>

a) Explain why ordering the coins from largest to smallest produced faster results.

b) For coins of [50, 25, 12, 10, 5, 1] typical timings:

<table>
<thead>
<tr>
<th>Change Amount</th>
<th>Run-Time (seconds)</th>
<th>Number of Tree Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>399</td>
<td>8.88</td>
<td>2,015,539</td>
</tr>
<tr>
<td>409</td>
<td>55.17</td>
<td>12,093,221</td>
</tr>
<tr>
<td>419</td>
<td>318.56</td>
<td>72,558,646</td>
</tr>
</tbody>
</table>

Why the exponential growth in run-time?

4. As with Fibonacci, the coin-change problem can benefit from dynamic program since it was slow due to solving the same problems over-and-over again. Recall the general idea of dynamic programming:

   - Solve smaller problems before larger ones
   - store their answers
   - look-up answers to smaller problems when solving larger subproblems, so each problem is solved only once

a) To solve the coin-change problem using dynamic programming, we need to answer the questions:

   - What is the smallest problem?

   - Where do we store the answers to the smaller problems?