3. The non-recursive \_str\_ method for \texttt{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 thing of building the string for the list as "a " + (string for the rest of the list)

a) Complete the recursive \texttt{strHelper} function in the \_str\_ method for our \texttt{OrderedList} class.

```python
def \_str\_(self):
    \"\"\" Returns a string representation of the list with a space between each item. \"\"\"
    def strHelper(current):
        if current == None:
            return ""
        else:
            return str(current.getData()) + " " + strHelper(current.getNext())

    # Start of \_str\_ method execution
    return "(head) " + strHelper(self._head) + "(tail)"
```

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

\[
\begin{align*}
0 & , 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots
\end{align*}
\]

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:

\[
\begin{align*}
Fib_0 &= 0 \\
Fib_1 &= 1 \\
Fib_N &= Fib_{N-1} + Fib_{N-2} \text{ for } N \geq 2.
\end{align*}
\]

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 \texttt{fib(5)}.
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?

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

g) 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:

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 = []

    # 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):
    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?
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?

\[
\begin{array}{c}
29 \\
- 25 \\
- 1 \\
- 1 \\
- 0
\end{array}
\]

Yes for US coins greedy gives true fewest # coins

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

No greedy - 5 coin solution, but better 12, 12, 5 3 coin 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?

change amount: \(29\)

# of coin types: \(\{1, 5, 10, 25, 50\}\)

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

Choices to consider:
1. smaller change amount(s) only
2. subset(s) of coins only
3. combination

\[
\begin{array}{c}
29 \\
5 \\
24 \\
19 \\
14 \\
9
\end{array}
\]

c) If we knew the solution to these smaller problems, how would 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

Possible First Coin
- 1-cent coin
- 5-cent coin
- 10-cent coin
- 25-cent coin

29 cents

28
24
19
17
4

1-cent coin
5-cent coin
10-cent coin
25-cent coin
50-cent coin

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?

add one coin to best smaller size problem

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

FewestCoins(change) = \[
\begin{cases}
\min(\text{FewestCoins}(\text{change - coin})) + 1 & \text{if change} \not\in \text{CoinSet} \\
\text{coin} \in \text{CoinSet} \text{ and coin} \leq \text{change} & \text{if change} \in \text{CoinSet}
\end{cases}
\]

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

Possible First Coin
- 1-cent coin
- 5-cent coin
- 10-cent coin
- 25-cent coin
- 50-cent coin

29 cents

28
24
19
17
4

Original Problem

Possible First Coin
- 1-cent coin
- 5-cent coin
- 10-cent coin
- 25-cent coin
- 50-cent coin

29 cents

28
24
19
17
4

1-cent coin
5-cent coin
10-cent coin
25-cent coin

50-cent coin

Smaller problems

4

not promising

\(O(\text{exponential})\)
1. The textbook solves the coin-change problem with the following code (note the “set-builder-like” notation):
\[ \{ c \mid c \in \text{coinValueList} \text{ and } c \leq \text{change} \} \]

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

I removed the fancy set-builder notation and replaced it with a simple if-statement check:

```
def recMC(change, coinValueList):
    global backtrackingNodes
    backtrackingNodes += 1
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= 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: 45.815 seconds
Fewest number of coins 6
Number of Backtracking Nodes: 67,716,925

a) Why is the second version so much “faster”?
1st builds a new list of coins before looping down it

b) Why does it still take a long time?
67 million nodes

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:

```
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 for
} // 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?
change amt remaining and previous coins to get there
b) When would a “child” tree node NOT be promising?

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</th>
<th>Coin types: {1, 5, 10, 25}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time: 0.036 seconds</td>
<td></td>
</tr>
<tr>
<td>Fewest number of coins 6</td>
<td></td>
</tr>
<tr>
<td>The number of each type of coins is:</td>
<td></td>
</tr>
<tr>
<td>number of 1-cent coins is 3</td>
<td></td>
</tr>
<tr>
<td>number of 5-cent coins is 0</td>
<td></td>
</tr>
<tr>
<td>number of 10-cent coins is 1</td>
<td></td>
</tr>
<tr>
<td>number of 25-cent coins is 2</td>
<td></td>
</tr>
<tr>
<td>Number of Backtracking Nodes: 4031</td>
<td></td>
</tr>
</tbody>
</table>

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

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

First solution found on left is 63 if coin set not too good for pruning. First solution found on right is the greedy 6-coin solution which is much better for pruning.

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?

First solution found is greedy which for 419 would be \((8 \times 50 + 1 \times 12 + 1 \times 5 + 2 \times 1)\) 12-coin solution. This limits the branch lengths, but tree is still exponential down to that depth.

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? charge amnt of 0 0 coins

- Where do we store the answers to the smaller problems?
backtrackingNodes = 0  # profiling variable to track number of state-space tree nodes

def solveCoinChange(changeAmt, coinTypes):
    def backtrack(changeAmt, numberOfEachCoinType, numberOfCoinsSoFar, solutionFound, bestFewestCoins, bestNumberOfEachCoinType):
        global backtrackingNodes
        backtrackingNodes += 1

        for index in range(len(coinTypes)):
            smallerChangeAmt = changeAmt - coinTypes[index]
            if promising(smallerChangeAmt, numberOfEachCoinType, numberOfCoinsSoFar, solutionFound, bestFewestCoins):
                if smallerChangeAmt == 0:
                    solutionFound = True
                    if not solutionFound or numberOfCoinsSoFar + 1 < bestFewestCoins:
                        # check if its best
                        bestFewestCoins = numberOfCoinsSoFar + 1
                        bestNumberOfEachCoinType = numberOfEachCoinType
                        bestNumberOfEachCoinType[index] = 1

                    solutionFound, bestFewestCoins, bestNumberOfEachCoinType = backtrack(smallerChangeAmt, numberOfEachCoinType, numberOfCoinsSoFar + 1, solutionFound, bestFewestCoins, bestNumberOfEachCoinType)

                    return solutionFound, bestFewestCoins, bestNumberOfEachCoinType

    def promising(changeAmt, numberOfCoinsReturned, solutionFound, bestFewestCoins):
        if changeAmt < 0:
            return False
        elif changeAmt == 0:
            return True
        else:
            # changeAmt > 0
            if solutionFound and numberOfCoinsReturned + 1 <= bestFewestCoins:
                return False
            else:
                return True

    # Body of solveCoinChange
    numberOfEachCoinType = []  # set-up initial "current state" information
    numberOfCoinsSoFar = 0
    solutionFound = False
    bestFewestCoins = -1
    bestNumberOfEachCoinType = None

    numberOfEachCoinType = []
    for coin in coinTypes:
        numberOfEachCoinType.append(0)
    numberOfCoinsSoFar = 0
    solutionFound = False
    bestFewestCoins = -1
    bestNumberOfEachCoinType = None

    solutionFound, bestFewestCoins, bestNumberOfEachCoinType = backtrack(changeAmt, numberOfEachCoinType, numberOfCoinsSoFar + 1, solutionFound, bestFewestCoins, bestNumberOfEachCoinType)

    return bestFewestCoins, bestNumberOfEachCoinType
Dynamic Programming Coin-change Algorithm:
I. Fills an array fewestCoins from 0 to the amount of change. An element of fewestCoins stores the fewest number of coins necessary for the amount of change corresponding to its index value.

For 29-cents using the set of coin types \{1, 5, 10, 12, 25, 50\}, the dynamic programming algorithm would have previously calculated the fewestCoins for the change amounts of 0, 1, 2, ..., up to 28 cents.

II. If we record the best, first coin to return for each change amount (found in the “minimum” calculation) in an array bestFirstCoin, then we can easily recover the actual coin types to return.

\[
fewestCoins[29] = \min(\text{fewestCoins}[28], \text{fewestCoins}[24], \text{fewestCoins}[19], \\
\text{fewestCoins}[17], \text{fewestCoins}[4]) + 1 = 2 + 1 = 3
\]

\[
\text{fewestCoins: } [0, 4, 2, 4, 2, 4, 3]
\]

\[
\text{bestFirstCoin: } [0, 12, 24, 29]
\]

Examine the coins in the solution for 29-cents from bestFirstCoin[29], bestFirstCoin[24], and bestFirstCoin[12]

b) Extend the lists through 32-cents.

\[
fewestCoins: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]
\]

\[
\text{bestFirstCoin: } [0, 1, 1, 1, 1, 5, 1, 1, 1, 10, 1, 12, 1, 1, 5, 1, 1, 10, 1, 10, 1, 12, 25, 1, 1, 1, 5, 1]
\]

c) What coins are in the solution for 32-cents?