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

```
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())

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

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

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

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:

$Fib_0 = 0$

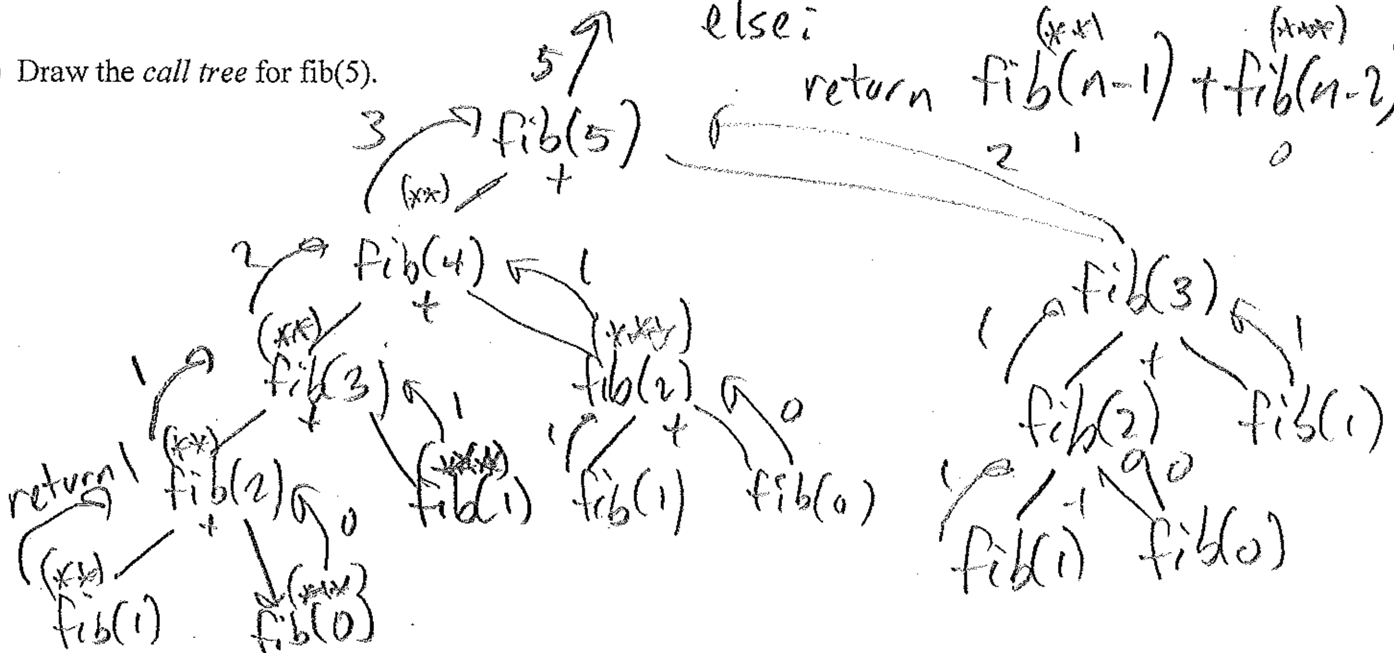
$Fib_1 = 1$

$Fib_N = Fib_{N-1} + Fib_{N-2}$ for $N \geq 2$.

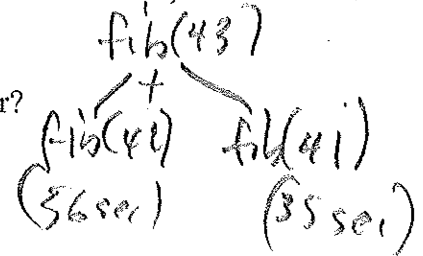
a) Complete the recursive function: `def fib (n):`

```
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).



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? 91 sec



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

over million years

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

recalculate answers many times

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:

```
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 ):
        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?

Data Structures (CS 1520) Lecture 10 Name:________________

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?

Next coin ≤ change & largest coin

29
-25
4
-1
3
-1
2
-1
1
-1
0

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?

29
25
4
-1
⋮

5 coin solution

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.

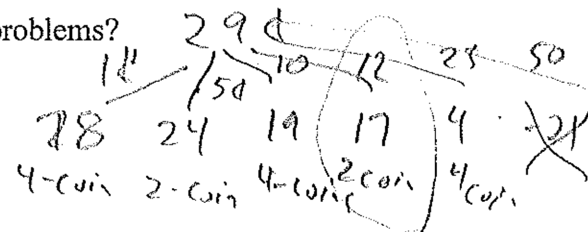
Solve a problem by:

- 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?

- amount of change
- ~~set of coin~~

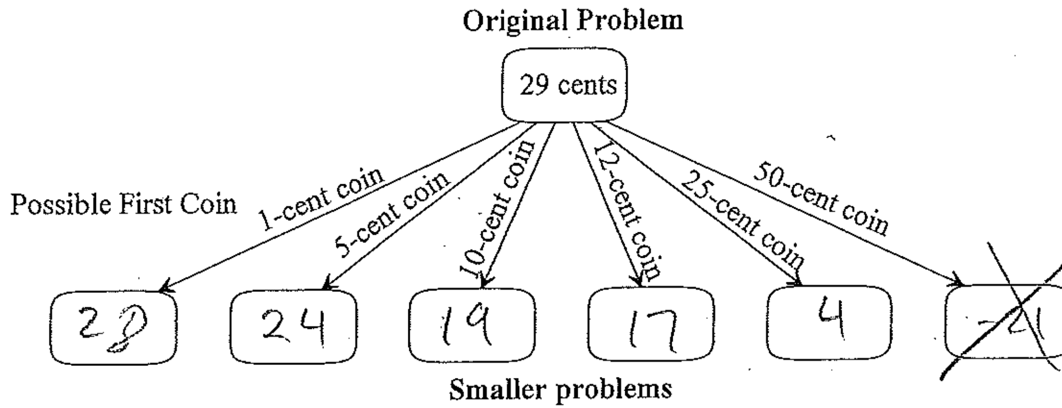
b) How could we divide the coin-change problem for 29-cents into smaller problems?



c) If we knew the solution to these smaller problems, how would be able to solve the original problem?

Take solution with fewest #coins among smaller problems and add 1 coin (coin to get to smaller problem)

3. After we give back the first coin, which smaller amounts of change do we have?



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?

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

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

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

29 cents

Possible First Coin: 1-cent coin, 5-cent coin, 10-cent coin, 12-cent coin, 25-cent coin, 50-cent coin

28 | 24 | 19 | 17 | 4 | 1

Smaller problems

Level 0

[12,12] 2 coins/level change 2

1. The textbook solves the coin-change problem with the following code (note the "set-builder-like" notation): $\{c \mid c \in \text{coinValueList and } c \leq \text{change}\}$

```
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: 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 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: 45.815 seconds
Fewest number of coins 6
Number of Backtracking Nodes: 67,716,925
```

a) Why is the second version so much "faster"?

top build list of coins ≤ change every call (67 million)

b) Why does it still take a long time?

recalculates smaller problems many time

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                          # each c represents a possible choice
        if promising(c) then                          # c is "promising" if it could lead to a better solution
            if c is a solution that's better than best then   # check if this is the best solution found so far
                best = c                              # remember the best solution
            else
                Backtrack(c)                          # follow a branch down the tree
            end if
        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" eventhough 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.

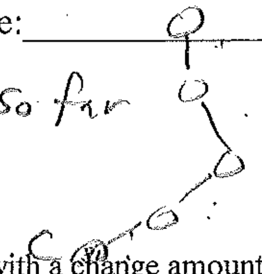
level 0 1 2 — 12 5 30 [12, 5]

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

30 level 2

change amount, level/# coins already returned, coin list of returned coins

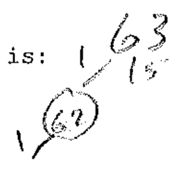
b) When would a "child" tree node NOT be promising? 5-coin solution best so far



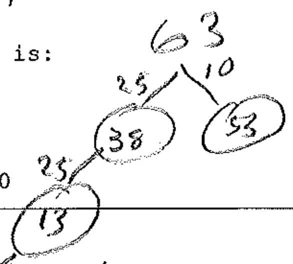
(best solution so far # coins) ≤ child's # coins/level

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

```
Change Amount: 63  Coin types: [1, 5, 10, 25]
Run-time: 0.036 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 1-cent coins is 3
number of 5-cent coins is 0
number of 10-cent coins is 1
number of 25-cent coins is 2
Number of Backtracking Nodes: 4836
```



```
Change Amount: 63  Coin types: [25, 10, 5, 1]
Run-time: 0.003 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 25-cent coins is 2
number of 10-cent coins is 1
number of 5-cent coins is 0
number of 1-cent coins is 3
Number of Backtracking Nodes: 310
```



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

code on left's first solution is 63 coins which is not helpful in pruning

code on right's first solution will be the greedy solution which is much better for pruning

6 coins

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

Change Amount	Run-Time (seconds)	Number of Tree Nodes
399	8.88	2,015,539
409	55.17	12,093,221
419	318.56	72,558,646

Why the exponential growth in run-time?

Even with pruning we still have a lot of nodes in the tree that get recalculated, for same change amounts.

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?