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)"
```

(**) n 0 1 2 3 4 5 ... 100

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

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:

$Fib_0 = 0$

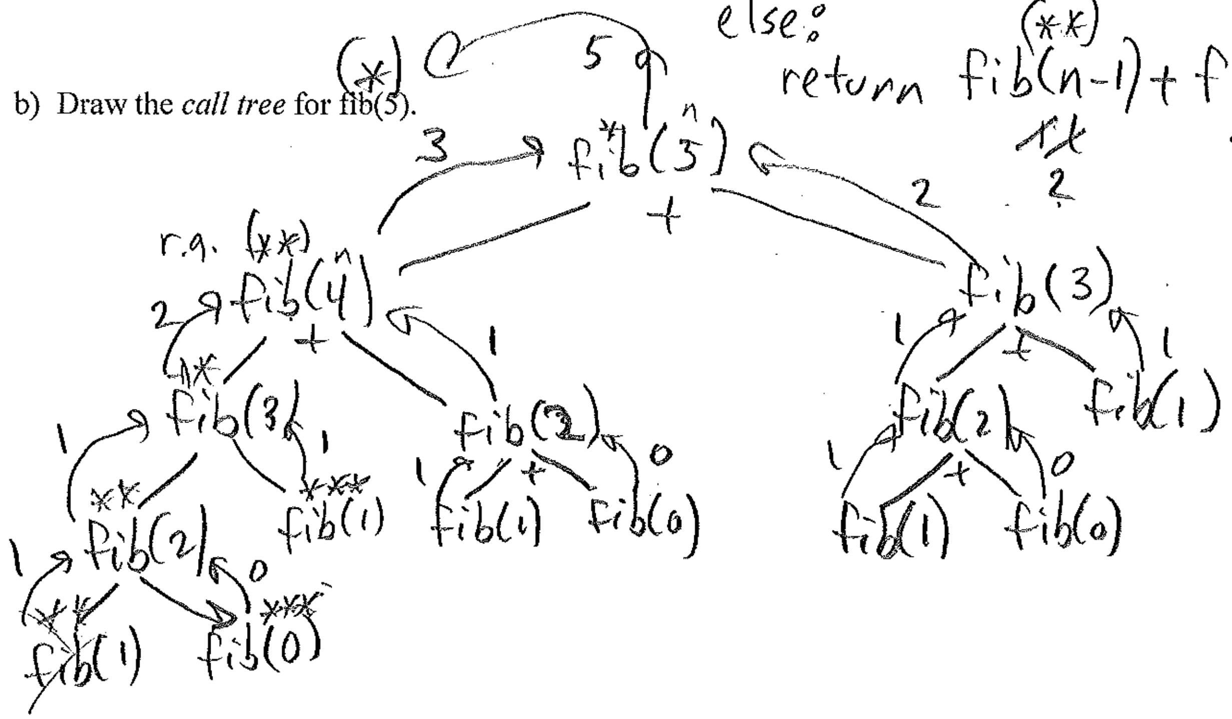
$Fib_1 = 1$

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

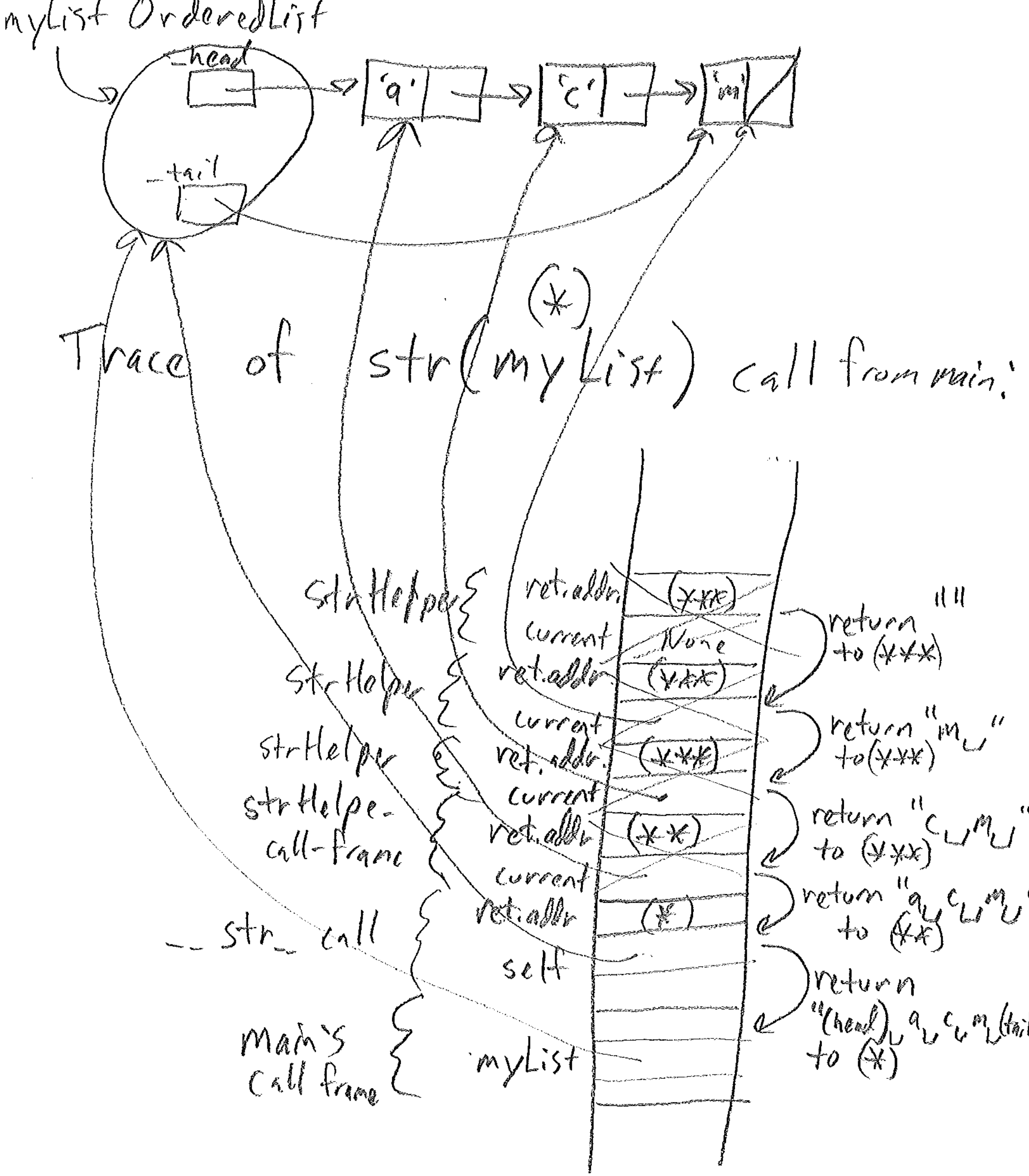
a) Complete the recursive function:

```
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).



Trace of str(myList) call from main:



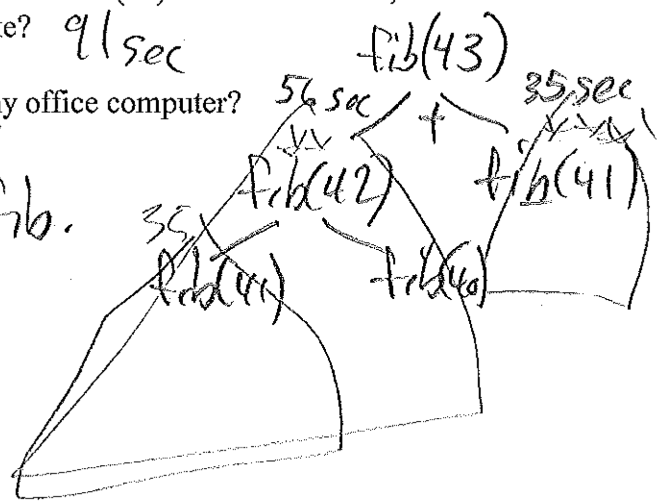
c) On my office computer, the call to fib(40) takes 22 seconds, the call to fib(41) takes 35 seconds, and the call to fib(42) takes 56 seconds. How long would you expect fib(43) to take? 91 sec

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

millions of years

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

redundant calculations of fib.




$2^n = 2^{n-1} * 2$

f) What is the computational complexity? $O(2^n)$ $1.5^n$

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

use dyn. programming

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]
```

O(n) running

0 1 2

[0, 1, 1]

n

O(n) storage

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? 2 use three variables a, b, n

a + b

O(1) storage

O(n) exec

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?

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

5 coin 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? No

/ 3 coin solution possible 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

29¢ change
[1, 5, 10, 12, 25, 50]

a) For the coin-change problem, what determines the size of the problem? ← amount of change and coin set size: [1, 5, 10, 12, 25, 50]

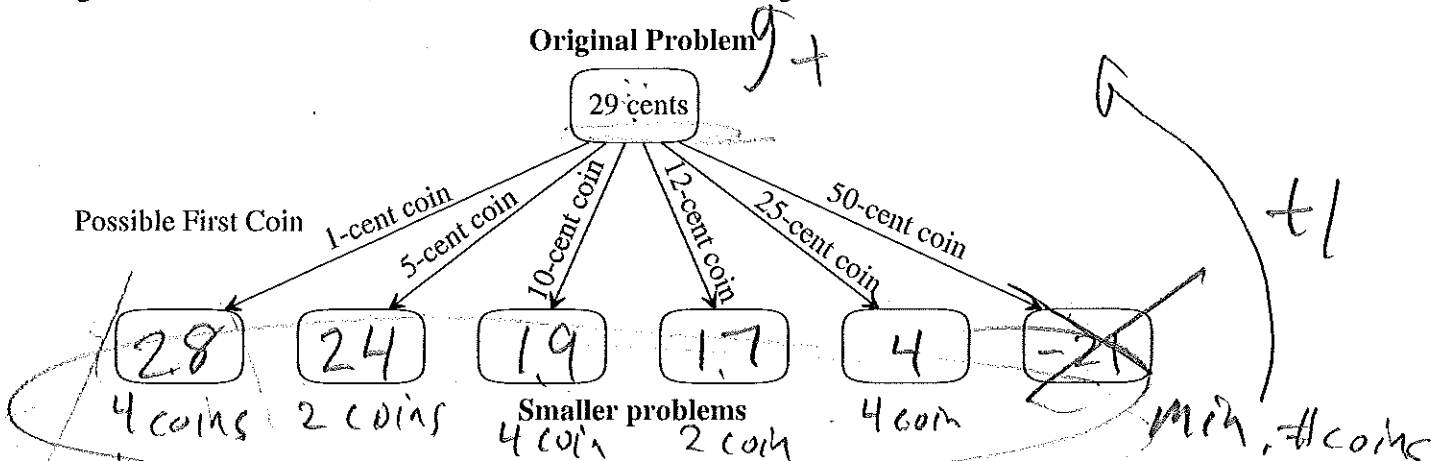
b) How could we divide the coin-change problem for 29-cents into smaller problems?

reduce amount of change

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

(see next page)

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.

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

6. Complete a couple levels of the recursion tree for 29-cents change using the set of coins {1, 5, 10, 12, 25, 50}.