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

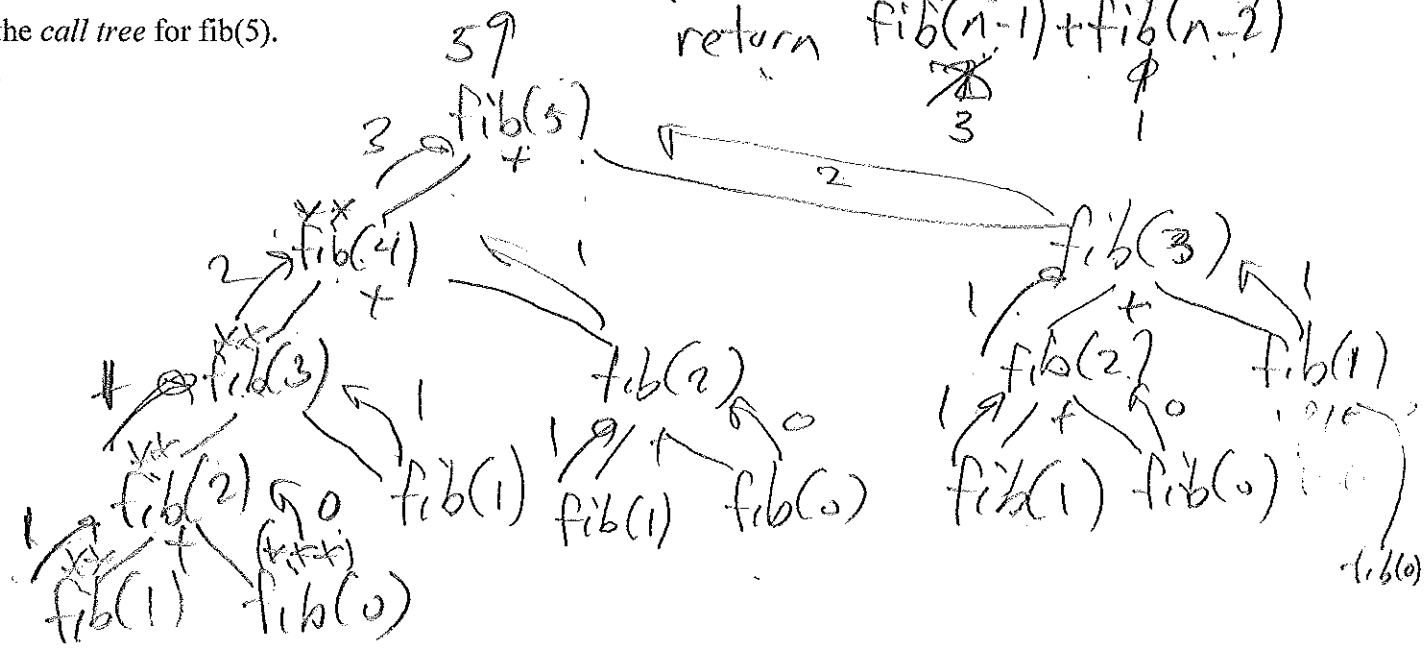
After the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined recursively as:

$Fib_0 = 0$
$Fib_1 = 1$
$Fib_N = Fib_{N-1} + Fib_{N-2}$ 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)
```

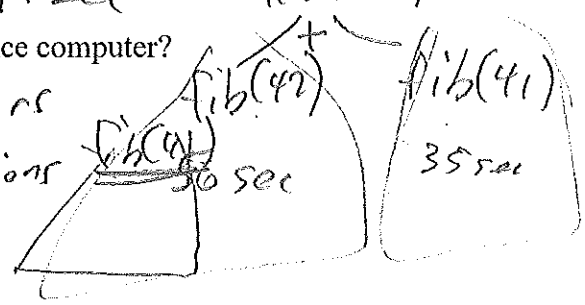
b) Draw the *call tree* for fib(5).



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? > 1 million years

e) Why do you suppose this recursive fib function is so slow?
many redundant fib(T calculations



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

g) How might we speed up the calculation of the Fibonacci series?
remember answer to smaller problems and look them up if needed again.

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, 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? 2

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
+1
0

5 coin solution
best

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 same 5 coin greedy solution

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:

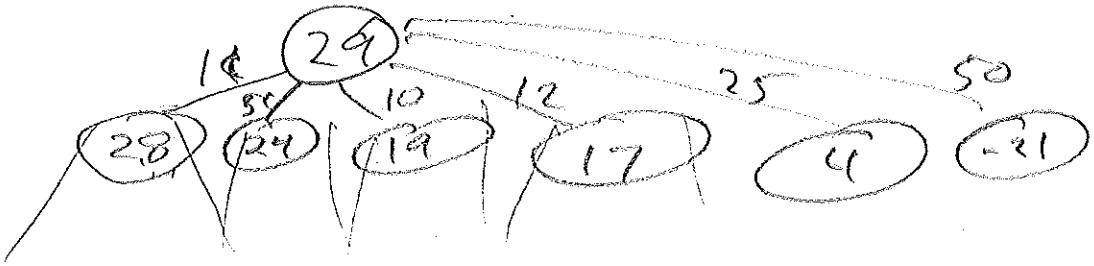
- dividing it into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

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

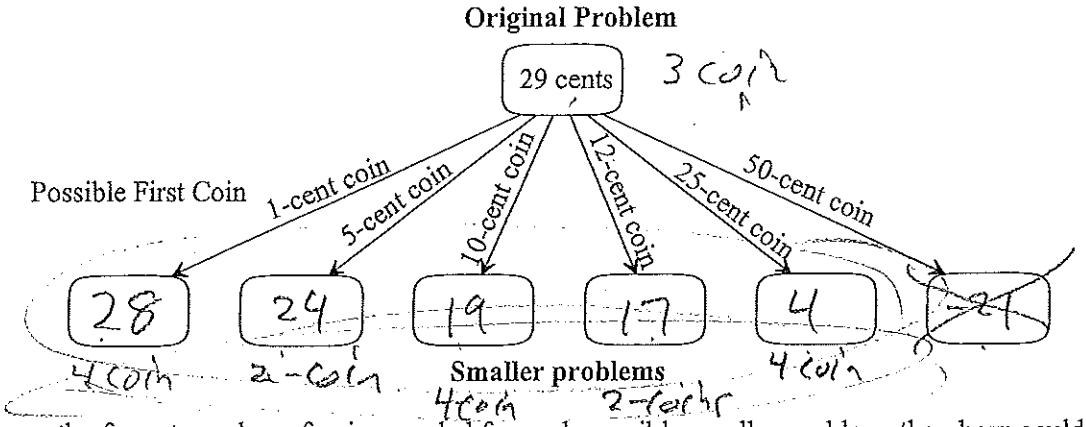
{1, 5, 10, 12, 25, 50}

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

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?



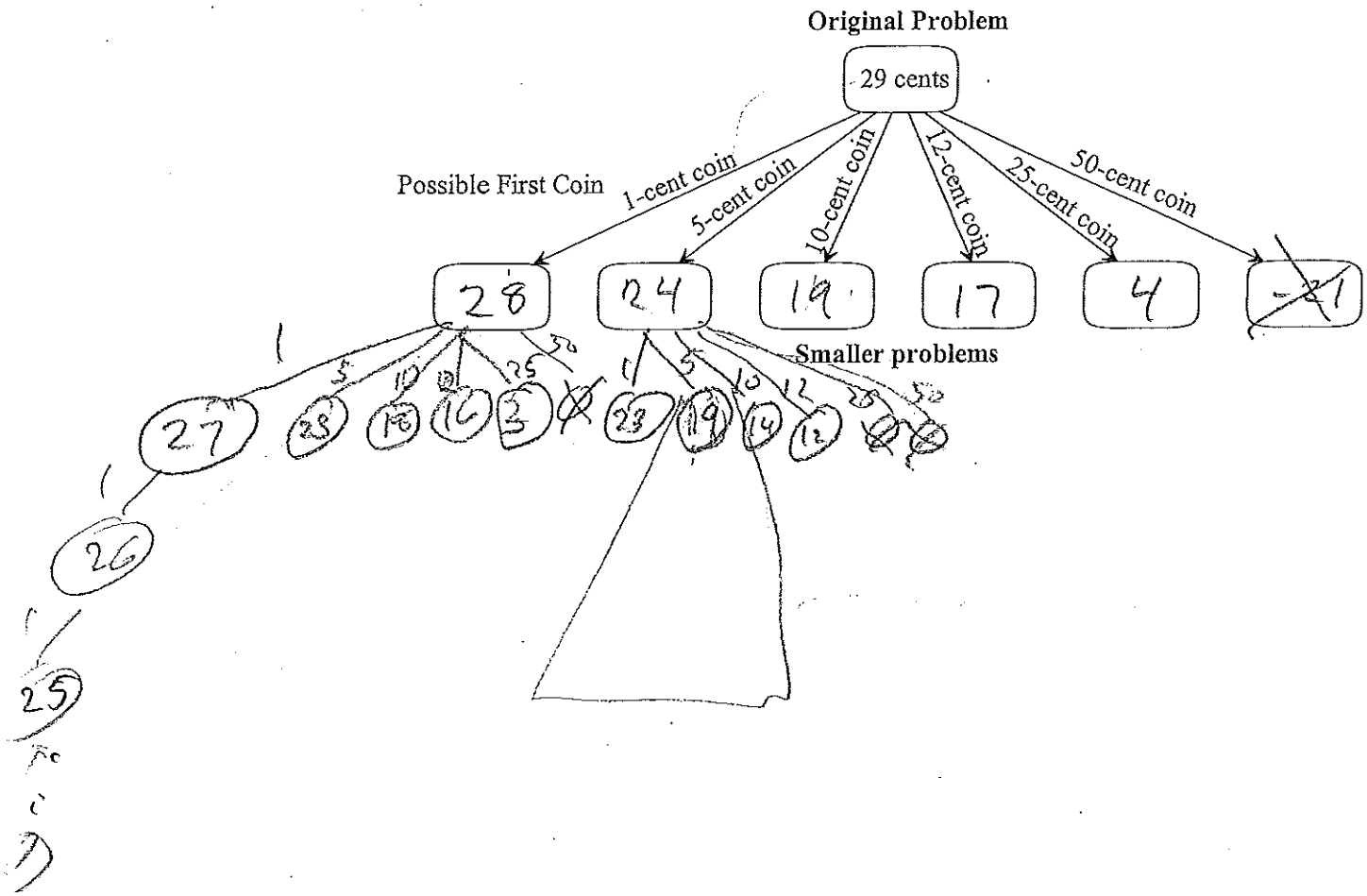
4. If we knew the fewest number of coins needed for each possible smaller problem, then how could determine the fewest number of coins needed for the original problem?

Take the minimum and add 1 since we gave back one coin to get to smaller problem.

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

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

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



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"? don't build a new list of coins that don't exceed change amount

b) Why does it still take a long time? redundant problems - whole recursion tree

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.

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