Intro. to Computing (810:051) **Lab 9** **Name:_____**

Objectives:

- Practice dictionary manipulation using dictionary methods and sequence operations
- Practice list manipulation using list methods and sequence operations
- Practice writing recursive functions

For today's lab you'll need several files, so start the lab by downloading the following file to your desktop:
http://www.cs.uni.edu/~fienup/cs051s10/labs/lab9.zip
Extract this file to the Desktop (or your flash drive) by right-clicking on lab9.zip icon and selecting Extract All.

Part A: Yesterday in lecture we talked about an algorithm for the function `printSortedByCount`, which solved the problem: "Write a function that takes the list-of-customer-field-lists from a previous lecture (by processing customerData.txt file), and generates a dictionary with a tally of the number of customers from each state. Then, write another function that takes this dictionary as a parameter and prints the number of customers from each state **sorted from most to least**." The file lab9/customerStatesDictionary.py contains a partial program:

```
def main():
    """ Open's file, reads customer information into a list, closes the file"""
    custFile = open('customerData.txt','r')
    customerList = generateList(custFile)
    custFile.close()
    statesCountDictionary = generateStatesCountDictionary(customerList)
    printSortedByState(statesCountDictionary)
    printSortedByCount(statesCountDictionary)

def generateList(custFile):
    """ Reads customer data from file and returns a list of customers"""
 ...

def generateStatesCountDictionary(customerList):
    """ Tallies the number of customers from each state and returns a dictionary
        with the state as a key and the count as the associated value."""
  ...

def printSortedByState(statesCountDictionary):
    """ Prints the tally of the number of customers from each state
        sorted alphabetically by the state abbreviation."""
    tupleList = statesCountDictionary.items()
    tupleList.sort()
    print "\n\n%5s   %-14s" % ("State","Customer Count")
    print "-"*25
    for item in tupleList:
        state, count = item
        print "%5s   %8d" % (state.center(5), count)

def printSortedByCount(statesCountDictionary):
    """ Prints the tally of the number of customers from each state sorted by count."""
```

The algorithm we talked about for the function `printSortedByCount` was:

- generate a list of tuples from the dictionary, tupleList = [....('IA', 40), ...]
- loop over the list to generate a new list of tuples with the elements reversed, [... (40, 'IA'),...]
- use the list sort method to sort from smallest to largest counts
- use the list reverse method to which to largest to smallest
- loop over the list and print a table sorted by count from largest to smallest

After you have the program for part A working correctly, raise your hand and we'll check your work.

Part B:

Yesterday in class we implementing a recursive function `fib` which takes as a parameter the position in the Fibonacci series, and returns the corresponding value in the series. For example, `fib(6)` is 8.

	Position in Series										
	0	1	2	3	4	5	6	7	8	9	10
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:

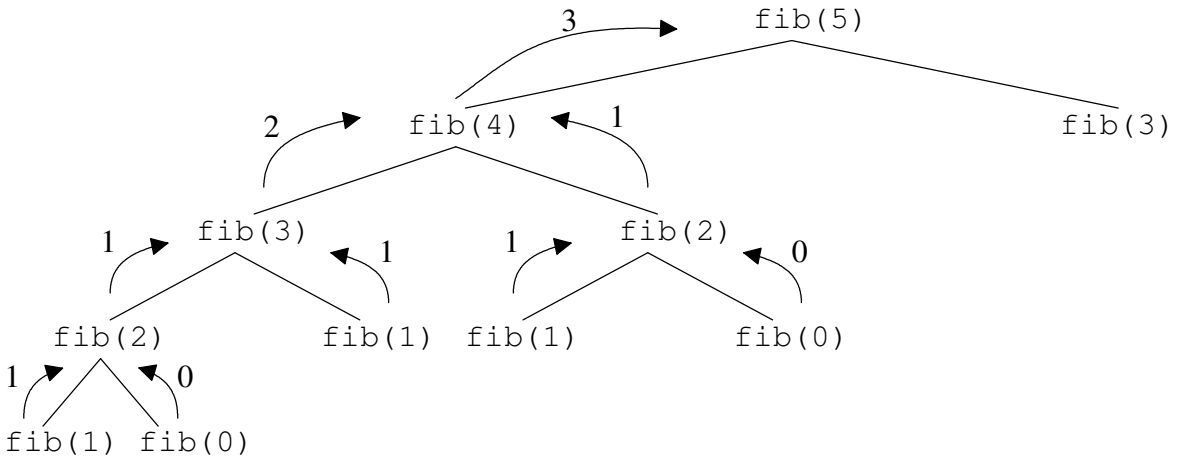
$Fib_0 = 0$

$Fib_1 = 1$

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

The `lab9.zip` file you downloaded and extracted contains a `fibonacci.py` program containing the recursive function `fib`.

a) Complete the recursion tree for fib(5).



b) Complete the following table of timings:

fib(38)	fib(39)	fib(40)

c) How long would you expect fib(41) to take?

d) How long would you guess fib(100) to take? (just make a wild guess -- DON'T TRY TO TIME IT)

e) Why do you suppose that the recursive fibonacci function, fib, is so slow?

After you have completed the timing table for the `fib` function and answered the questions, raise your hand and demonstrate your program.

Part C: As you might have guessed, the slowness of the recursive "divide-and-conquer" Fibonacci function, fib(n), is due to redundant calculations performed due to the recursive calls. 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 programmingsolution 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 needed) 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 problems
- store the solution to this next bigger problem for later usage so we never have to recalculate it

a) We'll reimplement fib(n) that calculates the n^{th} number in the Fibonacci sequence use dynamic programming. The `lab9.zip` file you downloaded and extracted contains a partial program in the file `fibDynPgm.py`. We'll use a list called, fibonacci, to store the solution to the "smaller" problems. Complete the dynamic programming code:

```
def 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(    )
    fibonacci.append(    )

    # Step 2:  Loop from base cases to biggest problem of interest
    for position in xrange(                ):

        fibonacci.append(                                          )

    # return nth number in the Fibonacci sequence
    return
```

b) Rerun the simple main function to calculate fib(40) or bigger. Note any "Wow" comments here:

c) 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 for example, when calculating the next value in the sequence we only need the previous two solution. Reimplement the `fib(n)` function a third time to eliminate the list `fibonacci` and replace it with three variables.

After you have implemented both dynamic programming versions of fibonacci, raise your hand and explain your code. If you do not get done today, then show me the completed lab in next lab period (after Spring Break). Make sure that you log off the computer before you leave.