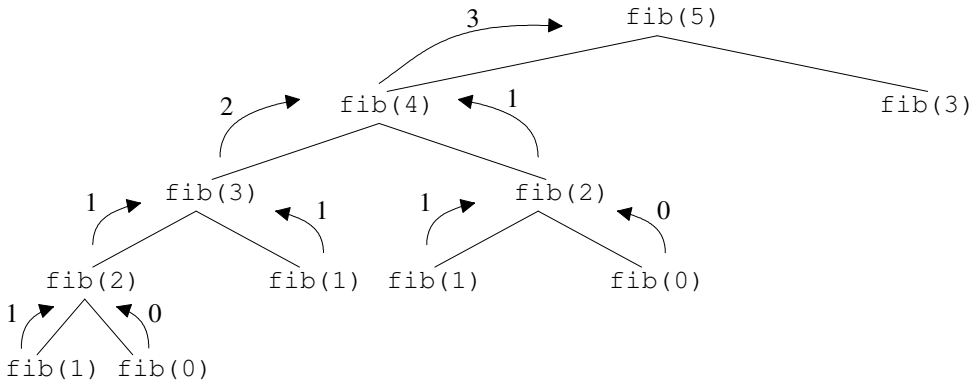
# Intro. to Computing

Name:_____

1. In lab 9 we experimented with the 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, ... |

The partial recursion tree for fib(5) was:




The recursive fibonacci function was very slow due to redundant calculations performed due to the recursive calls. I estimate that the fib(100) call would take about 16 million years to finish.

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) 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(    )
    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
```

Run 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 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.

2. A *function* is a procedural abstract (a named body of code to perform some action and return a resulting value). The syntax of a function definition is:

**def** functionName([parameter [, parameter]*]):
<functionBody>

where a parameter can be either:

- formal parameter name
- a keyword argument of the form: id = value which assigns the formal parameter id a specified default value. Note: keyword arguments can only appear as the last parameters in a parameter list
- a pseudo-argument of the form *args that captures all of the remaining non-keyword arguments in a tuple.
- a pseudo-argment keyword argument of the form **kwargs which captures all of the remaining keyword arguments into a dictionary

For example, the function definition "`def foo(x, y, *args, **kwargs):`" called with "`foo(1, 2, 3, 4, a=5, b=6)`" will result with *formal parameter* x containing 1, y containing 2, args containing (3, 4), and kwargs containing {'a':5, 'b':6}. Python uses pass-by-value parameter passing, which copies the value of the *actual parameters* to the *formal parameters*. Since variables associated with built-in collections and objects contain references, *actual parameters* to these only copy their reference values to corresponding formal parameters.

Predict the output of the following Python code segment.

```
def foo(a, b = 8, *args):
      a = 5
      b[2] = 99
      sum = 0
      for item in args:
            sum += item
      return sum


myInt = 4
myList = [ 1, 2, 3, 4 ]
total = foo(myInt, myList, 10, 11)
print 'myInt =', myInt
print 'myList =', myList
print 'total =', total
```

3. In Python, functions are *first-class data objects* which means that they can be:
- assigned to variables,
- passed as arguments to other functions,
- returned as the value of another function, or
- stored in a data structure such as a list or diectionary.

Thus, we write *higher-order functions* that expect a function and a set of data values as arguments. Python has the following predefined higher-order functions with are often useful:

| Function | General Syntax | Example | Description |
|---|---|---|---|
| `apply` | apply(object[, args[, kwargs]]) | apply(pow, (2, 3))<br>8 | Call a callable object with positional arguments taken from the tuple args, and keyword arguments taken from the optional dictionary kwargs |
| `map` | map(fn, sequence[, sequence, ...]) | map(len, ['cat', 'i', 'at'])<br>[3, 1, 2] | Return a list of the results of applying the function to the items of the argument sequence(s). |
| `filter` | filter(fn or None, sequence) | def odd(n):<br>  return n % 2 == 1<br>filter(odd, range(9))<br>[1, 3, 5, 7] | Return those items of sequence for which function(item) is true. If function is None, return the items that are true |
| `reduce` | reduce(fn, sequence[, initial]) | def add(x, y):<br>  return x + y<br>reduce(add, [2, 3, 4, 5])<br>14 | Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. |

A *lambda* is an anonymous function with no name that can be used to avoid defining a function and then passing it as a parameter to the higher-order functions. The general syntax of a lambda is:

**lambda** <argname-1, argname-2, ..., argname-n> **:** <expression using argname's>

The reduce example above would be: reduce(lambda x, y: x + y, [2, 3, 4, 5])

a) The built-in absolute value function (e.g., abs(-5)) takes a number as an argument and return the absolute value of the argument. Write code for a mapping that generates a list of the absolute values of the numbers in a list named `numbers`.

b) Write the code for a filtering that generates a list of the positive numbers in a list named `numbers`. You should first define You should use a lambda to create the auxiliary function.

c) Write the code for a reducing that creates a single string from a list of strings named `words`.