CS313E: Elements of Software Design

Recursion

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 26, 2012 at 15:42

What is Recursion?

Simply speaking, a recursive function is one that calls itself.

```
def fact (n):
    # Naive factorial routine.  Do you see anything
    # wrong and how would you fix it?
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Any recursive procedure must have:

- one or more *base cases* that return an answer without calling the procedure recursively;
- one or more *recursive cases* that call the method on arguments that *move the computation in the direction of the base case*.

Recursive Procedures

What is wrong with the following method? How could you fix it?

```
def isEven(n):
    if n == 0:
        return True
    else:
        return isEven(n - 2)
```

There must be a "well-founded" relation on the arguments of the method that decreases in each recursive call, but cannot decrease indefinitely. Otherwise, the method will run forever (on some inputs).

Thinking Recursively: Length of a List

Recursion is also a *way of thinking about computing problems*: Solve a "big" problem by solving "smaller" instances of the *same* problem. The simplest instances must be solvable directly.

Problem: How many items are in a list L?

Base case: If L is empty, length is 0.

Recursive case: If I knew the length of L[1:], then I could compute the length of L. How?

What must be true for the recursive case to "apply"?

What is the "well founded relation" that guarantees that this recursion terminates?

Does this work for any list?

Recursive Solution: Length of a List

We can naturally turn that recursive schema into Python code.

```
def listLen (lst):
    if not lst:
        return 0
    else:
        return 1 + listLen( lst[1:] )
```

Running the code:

```
> python
>>> from listLen import *
>>> listLen( ['a', 'b', 'c', 'd'], 0)
4
```

What's Actually Happening?

I instrumented the code to print out a "trace."

```
def listLen (lst, k):
    if not lst:
        print(" 1 +"*k, "+ 0 =")
        return 0
    else:
        print(" 1 +"*k, "1 + listLen(", lst[1:], ") =")
        return 1 + listLen( lst[1:], k+1 )
```

Now we get:

```
>>> from listLen import *
>>> listLen( ['a', 'b', 'c', 'd'], 0)
 1 + listLen( ['b', 'c', 'd'] ) =
 1 + 1 + listLen( ['c', 'd'] ) =
 1 + 1 + 1 + listLen( ['d'] ) =
 1 + 1 + 1 + 1 + listLen( [] ) =
 1 + 1 + 1 + 1 + + 0 =
4
```

Using Traceback to Show the Stack

```
def listLen2 (lst):
   if not lst:
      assert False                      # line 16
      return 0
   else:
      return 1 + listLen2( lst[1:] ) # line 19

try:
   listLen2( ['a', 'b', 'c'] )       # line 23
except:
   traceback.print_exc(limit=10)

>>> from listLen import *
Traceback (most recent call last):
  File "listLen.py", line 23, in <module>
    listLen2( ['a', 'b', 'c'] )
  File "listLen.py", line 19, in listLen2
    return 1 + listLen2( lst[1:] )
  File "listLen.py", line 19, in listLen2
    return 1 + listLen2( lst[1:] )
  File "listLen.py", line 19, in listLen2
    return 1 + listLen2( lst[1:] )
  File "listLen.py", line 16, in listLen2
    assert False
```

Some Recursive Problems

Many computing problems are naturally thought of as recursive:

- the length of a list
- the sum of a list of numbers
- the number of occurences of an element in a list
- the reverse of a list
- the append of two lists
- linear search
- the number of nodes in a tree
- binary search

For each problem, think about the well founded relation on the arguments that ensures termination.

Some Recursive Programs: Sum a List

Sum a list of Numbers:

What is the recursive schema? What is the base case? the recursive case?

Some Recursive Programs: Sum a List

Sum a list of Numbers:

What is the recursive schema? What is the base case? the recursive case?

```
def sumList (lst):
    if not lst:
        return 0
    else:
        return lst[0] + sumList( lst[1:] )
```

Some Recursive Programs: Count Occurrences

Count occurrences of an item in a list:

What is the recursive schema? What is the base case? the recursive case?

Some Recursive Programs: Count Occurrences

Count occurrences of an item in a list:

What is the recursive schema? What is the base case? the recursive case?

```
def countItem (lst, item):
    if not lst:
        return 0
    else:
        if ( lst[0] == item ):
            return 1 + countItem( lst[1:], item )
        else:
            return countItem( lst[1:], item )
```

Some Recursive Programs: Reverse a List

Reverse a list:

What is the recursive schema? What is the base case? the recursive case?

Some Recursive Programs: Reverse a List

Reverse a list:

What is the recursive schema? What is the base case? the recursive case?

```
def revList (lst):
    if not lst:
        return []
    else:
        return revList( lst[1:] ) + [ lst[0] ]
```

Some More Recursive Programs

Append of two lists:

```
def app (lst1, lst2):
    if not lst2:
        return lst1
    else:
        return app ( lst1 + [ lst2[0] ], lst2[1:] )
```

Linear search in a list:

```
def search (lst, item):
    if not lst:
        return False
    else:
        if ( lst[0] == item ):
            return True
        else:
            return search( lst[1:], item )
```

Running Our Recursive Programs

```
>>> listLen( [ 1, 2, 3, 4 ] )
4
>>> sumList( [ 1, 2, 3, 4, 5 ])
15
>>> countItem( [1, 2, 2, 3, 2, 5, 5, 1 ], 2)
3
>>> revList( [ 1, 2, 3, 4, 2 ] )
[2, 4, 3, 2, 1]
 >>> app( [ 1, 2, 3], [ 4, 5, 6 ])
[1, 2, 3, 4, 5, 6]
>>> search( [1, 2, 3], 2)
True
>>> search( [1, 2, 3], 4)
False
```

The Overhead of Recursion

Though recursion is a wonderful conceptual tool, *it's not free.* There is a cost to any recursive solution.

Consider the computation of the nth Fibonnaci number.

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2), \text{ for } n \geq 2$$

We call such a set of equations a *recurrence relation*. It is typically quite easy to implement a function in Python directly from the recurrence relation.

Some of the Fibonacci Numbers:

n	0	1	2	3	4	5	6	7	8	9	10	11
$F(n)$	0	1	1	2	3	5	8	13	21	34	55	89

BTW: often the sequence is started at 1 rather than at 0.

Fibonnaci in Python

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

This is a very nice transcription of the recurrence relation and works fine. Sort of.

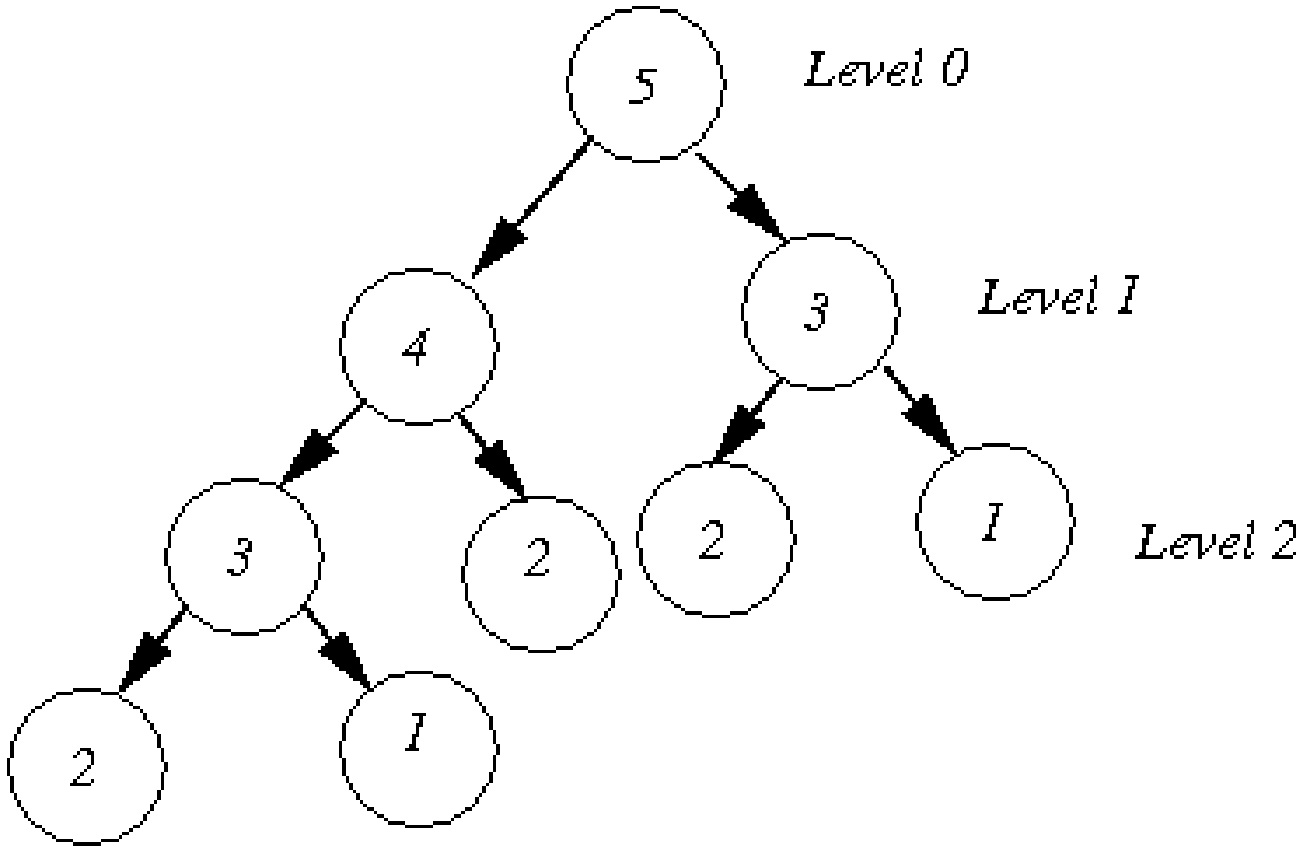
The Computation

You can see that that values go up quickly. But how much computation is being done?

```
>>> from Recursion import *
>>> fibCaller()
Input an integer (negative to exit):10
fib(10) = 55
Input an integer (negative to exit):20
fib(20) = 6765
Input an integer (negative to exit):30
fib(30) = 832040
Input an integer (negative to exit):40
fib(40) = 102334155
Input an integer (negative to exit):-10
```

How Bad Is It?

If n is 0 or 1, you only make one call to `fib`. But suppose $n = 5$, you do a lot of work, much of it repeated multiple times.



Counting Calls

How many calls to fib are made for a given n?

Counting Calls

How many calls to fib are made for a given n?

The recurrence relation for this is:

$$C(0) = 1$$
$$C(1) = 1$$
$$C(n) = 1 + C(n-1) + C(n-2), \text{ for } n \geq 2$$

We can easily write a Python function to compute this:

```
def fibCountCalls( n ):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return 1 + fibCountCalls( n-1 ) \
                 + fibCountCalls( n-2 )
```

Let's See How Bad It Is

Here's some code to compute and print out the number of calls for any given user-supplied input:

```
import time
def fibCaller():
    while True:
        n = int( input("Input an integer (negative to exit): ") )
        if n < 0:
            break
        tStart = time.clock(); ans = fib(n); tEnd = time.clock()
        interval = tEnd - tStart
        calls = fibCountCalls( n )
        print ("fib(" + str(n) +") = " + str(ans), end = "")
        print (" with " + str( calls ) + " recursive calls")
        print ("time = " + str(interval) + " seconds to execute")
```

Recursive Fib

```
>>> from Fib import *
>>> fibCaller()
Input an integer (negative to exit): 10
fib(10) = 55 with 177 recursive calls
time = 0.0 seconds to execute
Input an integer (negative to exit): 20
fib(20) = 6765 with 21891 recursive calls
time = 0.01 seconds to execute
Input an integer (negative to exit): 30
fib(30) = 832040 with 2692537 recursive calls
time = 2.33 seconds to execute
Input an integer (negative to exit): 40
fib(40) = 102334155 with 331160281 recursive calls
time = 282.53 seconds to execute
Input an integer (negative to exit): -1
>>>
```

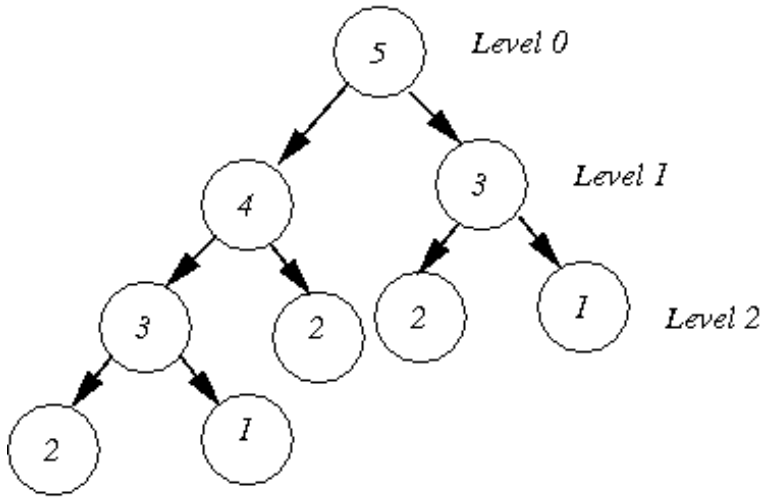
How Many Calls

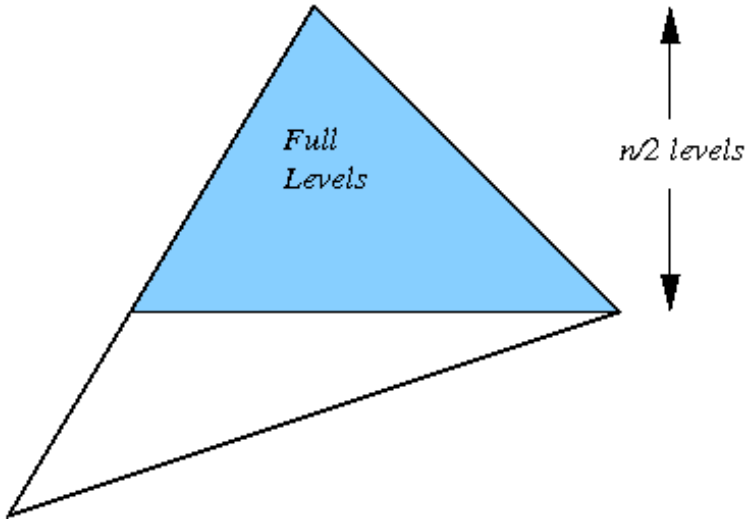
```
>>> from Recursion import *
>>> fibTallyCalls( 20 )
i:  0   fib(i):     0   calls:         1
i:  1   fib(i):     1   calls:         1
i:  2   fib(i):     1   calls:         3
i:  3   fib(i):     2   calls:         5
i:  4   fib(i):     3   calls:         9
i:  5   fib(i):     5   calls:        15
i:  6   fib(i):     8   calls:        25
i:  7   fib(i):    13   calls:        41
i:  8   fib(i):    21   calls:        67
i:  9   fib(i):    34   calls:       109
i: 10   fib(i):    55   calls:       177
i: 11   fib(i):    89   calls:       287
i: 12   fib(i):   144   calls:       465
i: 13   fib(i):   233   calls:       753
i: 14   fib(i):   377   calls:      1219
i: 15   fib(i):   610   calls:      1973
i: 16   fib(i):   987   calls:      3193
i: 17   fib(i):  1597   calls:      5167
i: 18   fib(i):  2584   calls:      8361
i: 19   fib(i):  4181   calls:     13529
```

What do you think is the complexity of this algorithm? Why?

Complexity of Naive Fib

This algorithm is *exponential*. To see this consider the following trees.





The number of nodes in the "full" levels of the recursion tree are $2^k - 1$, where $k = 1/2n$ for `fib( n )`. Since this *underestimates* the number of nodes in the tree to the right, `fib` is at least exponential.

Can We Do Better?

Take another look at the initial values of the Fibonnaci sequence:

n	0	1	2	3	4	5	6	7	8	9	10	11
$F(n)$	0	1	1	2	3	5	8	13	21	34	55	89

Surely we can do better than our horrible exponential solution. Perhaps instead of computing backwards from n down to 0, we can *compute forwards from 0 to n*.

A Better Implementation

```
def fibHelper(k, limit, ans, ansSub1):
    if k >= limit:
        return ans
    else:
        return fibHelper( k+1, limit, ans + ansSub1, ans)

def fibBetter(n):
    return fibHelper(1, n, 1, 0)
```

Why was the `fibHelper` function needed?

Better Performance

After changing `fibCaller` to call `fibBetter`:

```
>>> from Fib import *
>>> fibCaller()
Input an integer (negative to exit): 10
fib(10) = 55 with 10 recursive calls
time = 0.0 seconds to execute
Input an integer (negative to exit): 20
fib(20) = 6765 with 20 recursive calls
time = 0.0 seconds to execute
Input an integer (negative to exit): 30
fib(30) = 832040 with 30 recursive calls
time = 0.0 seconds to execute
Input an integer (negative to exit): 500
fib(500) = 139423224561697880139724382870407283950070256587697307
264108962948325571622863290691557658876222521294125 with 500 recursive calls
time = 0.0 seconds to execute
```

Better Performance

Is there any limit to how big an argument we can give? Yes, because the runtime stack will overflow when we reach the "recursion depth."

```
Input an integer (negative to exit): 900
fib(900) = 54877108839480000051413673948383714443800519309123592
724494953427039811201064341234954387521525390615504949092187441 2
182466791047314424730220139801604070070171756973179004832752466 5
2938800 with 900 recursive calls
time = 0.0 seconds to execute

Input an integer (negative to exit): 1000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "Recursion.py", line 66, in fibCaller
    tStart = time.clock(); ans = fibBetter(n); tEnd = time.clock()

   ... many, many lines deleted

  File "Recursion.py", line 51, in fibHelper
    if k >= limit:
RuntimeError: maximum recursion depth exceeded in comparison
```

Iterative Version

You can replace the recursive code by *iterative* code (i.e., a loop) and you won't have this problem. Try this as an exercise.

Writing the fib algorithm iteratively, it's possible to compute some pretty large numbers.

```
>>> from Recursion import *
>>> iterativeFib( 5000 )
3878968454388325633701916308325905312082127714646245106160597214895550139044037097010822916462210669479293452858882973813483102008954982940361430156911478938364216563944106910214505634133706558656238254656700712525929903854933813928836378347518908762970712033337052923107693008518093849801803847813996748881765554653788291644268912980384613778969021502293082475666346224923071883324803280375039130352903304505842701147635242270210934637699104006714174883298422891491273104054328753298044273676822977244987749874555691907703880637046832794811358973739993110106219308149018570815397854379195305617510761053075688783766033667355445258844886241619210553457493675897849027988234351023599844663934853256411952221859563060475364645470760330902420806382584929156452876291575759142343809142302917491088984155209854432486594079793571316841692868039545309545388698114665082066862897420639323438488465240988742395873801976993820317174208932265468879364002630797780058759129671389634214252579116872755600360311370547754724604639987588046985178408674382863125
```

Closed Form Solution

It turns out that there is a *closed form* solution for the nth Fibonacci number.

$$fib(n) = (1/\sqrt{5})[(1+\sqrt{5})/2]^n - (1/\sqrt{5})[(1-\sqrt{5})/2]^n.$$

What is the performance of this version?

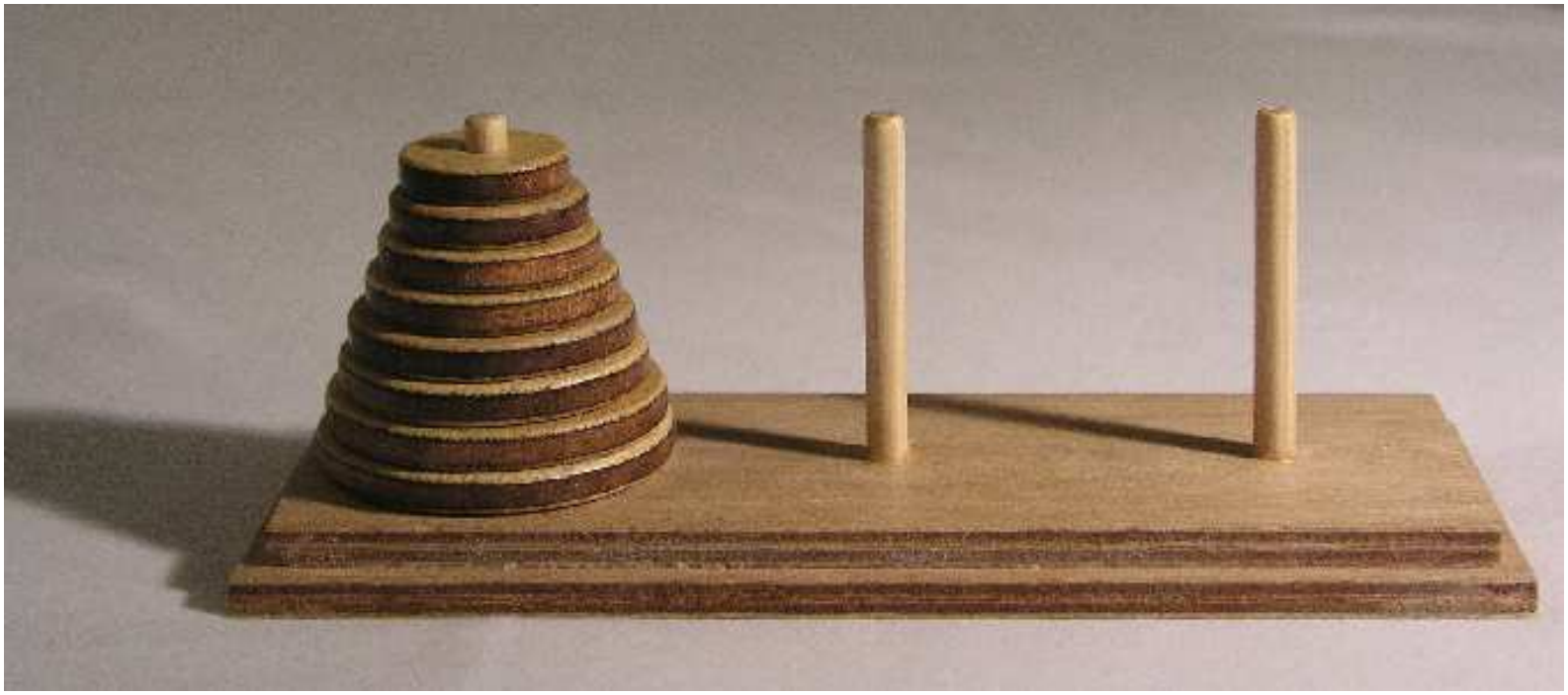
Naturally Recursive Problems

Some problems (like Fibonnaci) have a very natural recursive solution, but can easily be recoded in a non-recursive fashion.

Some other problems are very difficult to solve in any way but recursively.

Example: the *Towers of Hanoi* consists of three rods and a number of disks of different sizes that can slide onto any rod. The puzzle starts with the disks neatly stacked in order of size on one rod, the smallest at the top, thus making a conical shape.

Towers of Hanoi



Towers of Hanoi

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- No disk may be placed atop a smaller disk

How should you think about this problem in a recursive way?

From Wikipedia

There is a legend about a Vietnamese temple which contains a large room with three time-worn posts in it holding 64 golden disks. The priests of Hanoi, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the rules of the puzzle, since that time. According to the legend, when the last move of the puzzle is completed, the world will end. Are we in danger?

From Wikipedia

There is a legend about a Vietnamese temple which contains a large room with three time-worn posts in it holding 64 golden disks. The priests of Hanoi, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the rules of the puzzle, since that time. According to the legend, when the last move of the puzzle is completed, the world will end. Are we in danger?

If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64} - 1$ seconds or roughly 600 billion years; it would take 18,446,744,073,709,551,615 moves to finish.

Towers of Hanoi

```
def makeMove(frm, to):
    print ("Move disk from " + frm + " to " + to)

def towersOfHanoi(n, frm, using, to):
    if n == 1:
        makeMove(frm, to)
    else:
        towersOfHanoi(n-1, frm, to, using)
        makeMove(frm, to)
        towersOfHanoi(n-1, using, frm, to)

def towersMoveCount(n):
    < what should this be? >

def callTowers(n):
    if n < 0:
        print ("Bad value input")
        return
    towersOfHanoi(n, "a", "b", "c")
    moves = towersMoveCount(n)
    print ("This took " + str(moves) + " moves")
```

Calling It

```
>>> callTowers(4)
  Move disk from a to b
  Move disk from a to c
  Move disk from b to c
  Move disk from a to b
  Move disk from c to a
  Move disk from c to b
  Move disk from a to b
  Move disk from a to c
  Move disk from b to c
  Move disk from b to a
  Move disk from c to a
  Move disk from b to c
  Move disk from a to b
  Move disk from a to c
  Move disk from b to c
This took 15 moves
```

Calling It

```
>>> for i in range(20): print i, towersMoveCount(i)
...
0 0
1 1
2 3
3 7
4 15
5 31
6 63
7 127
8 255
9 511
10 1023
11 2047
12 4095
13 8191
14 16383
15 32767
16 65535
17 131071
18 262143
19 524287
```

How Many Calls

Solving the problem for n disks requires $2^n - 1$ moves. This algorithm is *exponential* in the number of disks.

But how many times was TowersOfHanoi called as a function of the input n?

Each recursive call actually only moved one disk, so there were the same number of calls to the method as there were moves!