Consider the factorial function:

\[ k! = 1 \times 2 \times \ldots \times k. \]

This is often defined mathematically by the following recurrence relation:

\[
\begin{align*}
1! &= 1 \\
n! &= n \times (n-1)!, \text{ for } n > 1
\end{align*}
\]

It is typically quite easy to implement a function in Python directly from the recurrence relation.

For example, to compute 5! we do the following:

\[
\begin{align*}
5! &= 5 \times 4! \\
&= 5 \times (4 \times 3!) \\
&= 5 \times (4 \times (3 \times 2!)) \\
&= 5 \times (4 \times (3 \times (2 \times 1!))) \\
&= 5 \times (4 \times (3 \times (2 \times 1))) \\
&= 5 \times (4 \times (3 \times 2)) \\
&= 5 \times (4 \times 6) \\
&= 5 \times 24 \\
&= 120
\end{align*}
\]
Factorial Function

Mathematical definition:

\[
1! = 1 \\
n! = n \times (n - 1)!, \text{ for } n > 1
\]

Here's a straightforward implementation in Python.

```python
def fact(n):
    """ Factorial function. ""
    if n == 1:
        return 1
    else:
        return n * fact(n-1)  # note recursive call
```

This function is recursive because it calls itself.

Can you see anything wrong with this? How might you fix it?

In programming, recursion just means that a program calls itself, either directly or indirectly.

Functions A and B are mutually recursive if A calls B and B calls A.

```python
def isNonnegativeEven(n):
    print("In isNonnegativeEven(", n, ")")
    if (n < 0):
        return False
    elif (n == 0):
        return True
    else:
        return isNonnegativeOdd(n - 1)
def isNonnegativeOdd(n):
    print("In isNonnegativeOdd(", n, ")")
    if (n < 1):
        return False
    elif (n == 1):
        return True
    else:
        return isNonnegativeEven(n - 1)
```

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 can be solved directly.

**Example:** Suppose I decide to walk to the store.

**Base case:** I’m already there; there’s nothing to do.

**Recursive case:** I take one step; now I’ve reduced the problem to the “smaller” but structurally identical problem of walking from where I am now to the store.

What can go wrong:

- I don’t know how to take a step.
- There is no store.
- I can’t recognize when I get to the store.
- I walk in the wrong direction.
- I walk right past the store and keep going forever.

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 can be solved directly.

**Example:** Suppose I decide to walk to the store.

**Base case:** I’m already there; there’s nothing to do.

**Recursive case:** I take one step; now I’ve reduced the problem to the “smaller” but structurally identical problem of walking from where I am now to the store.

What can go wrong:

- I don’t know how to take a step.
- There is no store.
- I can’t recognize when I get to the store.
- I walk in the wrong direction.
- I walk right past the store and keep going forever.
Any recursive function must have:
- one or more base cases that return an answer without calling the procedure recursively;
- one or more recursive cases that call the function on arguments that move the computation in the direction of some base case;
- assurance that you will eventually hit one of the base cases.

```python
def fact(n):
    if n <= 1: # the base case
        return 1
    else:
        return n * fact(n-1) # the recursive case
```

How do you know that this eventually terminates?

### Some Faulty Examples

```python
def factBad(n):
    return n * factBad(n-1)

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

What's wrong and how would you fix these?

### Recursive Thinking: Some Examples

**Example:** 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?

```python
def countItemsInList(L):
    """ Recursively count the number of items in list. ""
    if not L: # empty list counts as False
        return 0
    else:
        return 1 + countItemsInList(L[1:])
```

```python
>>> l1 = [1, 2, 3, "red", 2.9]
>>> countItemsInList(l1)
5
```

Does this work for any list?

### What's Actually Happening?

I instrumented the code to print out an “execution trace.”

```python
def countItems2(lst, k):
    if not lst:
        print("1+"*k, "0 =")
        return 0
    else:
        print("1+"*k, "1 + countItems(", lst[1:], ") =")
        return 1 + countItems2(lst[1:], k+1)
```

```python
>>> lst = [4, 5, 2, 5, 9, 2, 8]
>>> countItems2(lst, 0)
1 + countItems([5, 2, 5, 9, 2, 8]) =
1 + 1 + countItems([5, 9, 2, 8]) =
1 + 1 + 1 + countItems([9, 2, 8]) =
1 + 1 + 1 + 1 + countItems([2, 8]) =
1 + 1 + 1 + 1 + 1 + countItems([8]) =
1 + 1 + 1 + 1 + 1 + 1 + countItems([ ]) =
1 + 1 + 1 + 1 + 1 + 1 + 0 =
7
```
Recursive Thinking: Some Examples

Example: how can you sum a list of numbers?

Base case: If L is empty, the sum is 0.

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

def sumItemsInList(L):
    """ Recursively sum the items in a list. """
    if not L: # empty list counts as False
        return 0
    else:
        return L[0] + sumItemsInList(L[1:])

>>> lst = [5, 6, 14, -3, 0, -70]
>>> sumItemsInList(lst)
-48

Example: how can you count the occurrences of key in list L?

Base case: If L is empty, the count is 0.

Recursive case: If L starts with key, then it’s 1 plus the count in the rest of the list; otherwise, it’s just the count in the rest of the list.

def countOccurrencesInList(key, L):
    """ Recursively count the occurrences of key in L. """
    if not L: # empty list counts as False
        return 0
    elif key == L[0]:
        return 1 + countOccurrencesInList(key, L[1:])
    else:
        return countOccurrencesInList(key, L[1:])

>>> lst = [5, 6, 14, -3, 0, -70]
>>> countOccurrencesInList(3, lst)
0
>>> countOccurrencesInList(6, lst)
2

Example: how can you reverse a list L?

Base case: If L is empty, the reverse is [].

Recursive case: If L is not empty, remove the first element and append it to end of the reverse of the rest.

def reverseList(L):
    """ Recursively reverse a list. """
    if not L: # empty list counts as False
        return []
    else:
        return reverseList(L[1:]) + [L[0]]

>>> lst = [1, 5, "red", 2.3, 17]
>>> print(reverseList(lst))
[17, 2.3, 'red', 5, 1]
Recursive Thinking: Some Examples

How would Jason do this recursively?

```
COUNT = 500
STRING = "I will not throw paper airplanes in class."

def blackboard(n):
    if n <= 0:  # base case
        return
    else:
        print(STRING)  # recursive case
        blackboard(n-1)

blackboard(COUNT)
```

Recursion

An algorithm that dates from Euclid finds the greatest common divisor of two positive integers:

\[
gcd(a, b) = a, \text{ if } a = b
\]

\[
gcd(a, b) = gcd(a, b - a), \text{ if } a < b
\]

\[
gcd(a, b) = gcd(a - b, b), \text{ if } b < a
\]

```
def gcd(a, b):
    """Euclid's algorithm for GCD."""
    print("Computing gcd(", a, ",", b, ")")
    if a < b:
        return gcd(a, b-a)
    elif b < a:
        return gcd(a-b, b)
    else:
        print("Found gcd:", a)
        return a

print("gcd (68, 119) =", gcd(68, 119))
```

What is assumed about \(a\) and \(b\)? What is the base case? The recursive cases?

Running GCD

```bash
> python gcd.py
Computing gcd (68 , 119 )
Computing gcd (68 , 51 )
Computing gcd (17 , 51 )
Computing gcd (17 , 34 )
Computing gcd (17 , 17 )
gcd (68 , 119 ) = 17
```

Some Exercises for You to Try

- Write a recursive function to append two lists.
- Write a recursive version of linear search in a list.
- Write a recursive version of binary search in a list.
- Write a recursive function to sum the digits in a decimal number.
- Write a recursive function to check whether a string is a palindrome.

It’s probably occurred to you that many of these problems were already solved with built in Python methods or could be solved with loops.

That’s true, but our goal is to teach you to think recursively!
For some recursive solutions you’ll need a helper function.

For example, remember binary search from slideset 10.

```python
def BinarySearch( lst, key):
    ''' Search for key in sorted list lst. '''
    low = 0
    high = len(lst) - 1
    while (high >= low):
        mid = (low + high) // 2
        if key < lst[mid]:
            high = mid - 1
        elif key == lst[mid]:
            return mid
        else:
            low = mid + 1
    # What's true here? Why this value?
    return (-low - 1)
```

It’s clear how to think of this recursively.

Recursion

Here’s one version:

```python
def BinarySearchRecursive( lst, key):
    ''' Search for key in sorted list lst. '''
    if lst == []:
        return False
    mid = (len(lst) - 1) // 2
    if key == lst[mid]:
        return True
    elif key < lst[mid]:
        return BinarySearchRecursive( lst[:mid], key)
    else:
        return BinarySearchRecursive( lst[mid+1:], key)
```

This is inferior to the iterative version for at least two reasons:

- it creates a new copy of half of the list in each recursive call (by slicing);
- it returns a Boolean rather than an index. Can you see why returning an index would be difficult?

A better approach would be to preserve the original list and add some parameters to the recursive function. This is often done with a helper function.

```python
def BinarySearchHelper( lst, key, low, high):
    if low > high:
        return -low - 1
    mid = (low + high) // 2
    if key < lst[mid]:
        return BinarySearchHelper( lst[:mid], key, low, mid - 1)
    elif key == lst[mid]:
        return mid
    else:
        return BinarySearchHelper( lst[mid+1:], key, mid + 1, high)
```

```python
>>> from BinarySearch import *
>>> lst = [ 2, 4, 7, 9, 10 , 12 , 14 , 17 , 20 ]
>>> BinarySearchRecursive( lst, 10 )
True
>>> BinarySearchRecursive( lst, 11 )
False
>>> BinarySearchRecursive( lst, 21 )
False
```

Compare the helper to the nonrecursive version.
Recursion vs. Iteration

For some problems a recursive solution is simpler to code and to understand than an iterative solution.

You can always convert a recursive solution to an iterative solution, and vice versa. But it may not be easy!

```python
def gcdIter(a, b):
    """ Iterative version of GCD. """
    while a != b:
        if a < b:
            b = b - a
        elif b < a:
            a = a - b
    return a
```

The Overhead of Recursion

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

Consider the computation of the nth Fibonacci number.

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

Some of the Fibonacci Numbers:

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(n)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
</tr>
</tbody>
</table>

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

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

def fib(n):
    """ Compute the nth Fibonnaci number. """
    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. What’s wrong and how would you fix it?

Instrumenting the Code

```python
import time

def fibCaller():
    while True:
        n = int(input("Input an integer (negative to exit): "))
        if n < 0:
            break
        # Time the call
        tStart = time.clock(); ans = fib(n); tEnd = time.clock()
        interval = tEnd - tStart
        intervalStr = format(interval, "9.4f")
        # How many recursive calls?
        calls = fibCountCalls(n)
        print("fib(" + str(n) +") = ", ans)
        print("with " + str(calls) + " recursive calls")
        print("time = " + intervalStr + " seconds to execute")

>>> fibCaller()
Input an integer (negative to exit): 10
fib(10) = 55 with 177 recursive calls
time = 0.0000 seconds to execute
Input an integer (negative to exit): 20
fib(20) = 6765 with 21891 recursive calls
time = 0.0053 seconds to execute
Input an integer (negative to exit): 30
fib(30) = 832040 with 2692537 recursive calls
time = 0.2702 seconds to execute
Input an integer (negative to exit): 40
fib(40) = 102334155 with 331160281 recursive calls
time = 31.0691 seconds to execute
< I tried it for 50 but got tired of waiting >
Input an integer (negative to exit): -10
```

The Computation

You can see that the values go up quickly.

```
>>> fibCaller()
Input an integer (negative to exit): 10
fib(10) = 55 with 177 recursive calls
time = 0.0000 seconds to execute
Input an integer (negative to exit): 20
fib(20) = 6765 with 21891 recursive calls
time = 0.0053 seconds to execute
Input an integer (negative to exit): 30
fib(30) = 832040 with 2692537 recursive calls
time = 0.2702 seconds to execute
Input an integer (negative to exit): 40
fib(40) = 102334155 with 331160281 recursive calls
time = 31.0691 seconds to execute
< I tried it for 50 but got tired of waiting >
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?

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:

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

```
>>> 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
```

Can We Do Better?

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

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(n)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
</tr>
</tbody>
</table>

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

```python
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)
```

```
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)
```

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(n)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
</tr>
</tbody>
</table>

Why was the fibHelper function needed?
Better Performance

After changing \texttt{fibCaller} to call \texttt{fibBetter}:

\begin{verbatim}
>>> fibBetterCaller()
Input an integer (negative to exit): 10
fib(10) = 55 with 10 recursive calls
time = 0.0000 seconds to execute
Input an integer (negative to exit): 20
fib(20) = 6765 with 20 recursive calls
time = 0.0000 seconds to execute
Input an integer (negative to exit): 30
fib(30) = 832040 with 30 recursive calls
time = 0.0000 seconds to execute
Input an integer (negative to exit): 40
fib(40) = 102334155 with 40 recursive calls
time = 0.0000 seconds to execute
Input an integer (negative to exit): 500
fib(500) = 139423224561697880139724382870407283950070256587697307
2641089629432557162268329069155765887622252194125 with 500
recursive calls
time = 0.0004 seconds to execute
Input an integer (negative to exit): -10
>>>
\end{verbatim}

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."

\begin{verbatim}
>>> fibBetterCaller()
Input an integer (negative to exit): 900
fib(900) = 5487710883948000005141367394838371444380051930912359272
4494953427039811201064341234964387521525390615054949902187
4412182466791047314424730220139801604070071715766973179004
8327524662593880 with 900 recursive calls
time = 0.0007 seconds to execute
Input an integer (negative to exit): -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
< some lines omitted >
  File "/u/byoung/cs303e/slides/RecursionExamples.py", line
    73, in fibHelper
      if k >= limit:
RecursionError: maximum recursion depth exceeded in comparison
\end{verbatim}

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.

\begin{verbatim}
>>> iterativeFib( 5000 )
3878968453832563370191630382590530512082127714646426451061605
9721489555013904403709701082291646221066947929345285882973
8134381020089549829403614301569114789383642165639441691021
450563413370658668623282546467070172592909384933839288363
78347518908769270120333705292310769300818093849801803847
813996748881765556537882916442689192808346137787969020152029
308247566634626269230718833248032803750391303529033045058427
0114763524227021093463769910400671417488822898422891491273104
05432875392804427367682297744987794784556919077038063704
6832798411358973739993110162193018490185708153978543791953
0651871076105307568873768033667355445258844886241619210553
4574936758978490279882343510235998446639485325641195222185
9563060475364644547076030390242080638285489219164528762915757
591423483809142032917491088984155209854432486594079793571316
84169288603954530955438869811466508206686289742063933243848
84652098874239587380197699382031717420893226564688793640026
307977805875912967313986342125205791168772755600360311370547
7547246046439987588046985178408674382863125
\end{verbatim}

Closed Form Solution

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

\[
fib(n) = \frac{1}{\sqrt{5}} \left[ \left(1 + \sqrt{5}\right) / 2 \right]^n - \frac{1}{\sqrt{5}} \left[ \left(1 - \sqrt{5}\right) / 2 \right]^n.
\]
Naturally Recursive Problems

Some problems (like Fibonacci) 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.

Towers of Hanoi

The *Towers of Hanoi* consists of three pegs and a number of disks of different sizes that can slide onto any peg. The puzzle starts with the disks neatly stacked in order of size on one peg, the smallest at the top.

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

- Only one disk may be moved at a time.
- No disk may be placed on top of a smaller disk.

This is a problem which is easy to solve recursively, but very hard to solve iteratively.
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, for centuries.

According to the legend, when the last move of the puzzle is completed, the world will end. Are we in danger?

The smallest solution for 64 disks requires:

\[18,446,744,073,709,551,615\] moves.

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.

Thinking Recursively

**Problem:** Move \(n\) disks from peg A to peg C, using peg B as the intermediate peg.

- What's the base case?
- What are the recursive cases?

What problem are we solving? Solving the puzzle means moving the disks from the start state to the ending state, while following the rules.

Our program is solving a slightly different problem: print out a legal list of moves that will take us from start state to end state.

So, what's a move?

```python
def makeMove(peg1, peg2):
    print ("Move disk from " + peg1 + " to " + peg2)
```
Towers of Hanoi

Our main function has four parameters:

- \( n \) : how many disks to move (int);
- \( A \) : the start peg (str);
- \( B \) : the intermediate peg (str);
- \( C \) : the destination peg (str).

```python
def towersOfHanoi(n, A, B, C):
    """ Prints a list of legal moves to solve the Tower of Hanoi problem for \( n \) disks. ""
    if n == 1:
        makeMove(A, C)
    else:
        towersOfHanoi(n-1, A, C, B)
        makeMove(A, C)
        towersOfHanoi(n-1, B, A, C)
```

Calling It

```python
>>> towersOfHanoi(4, 'A', 'B', 'C')
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 A 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 # /
```

How Many Moves

```python
def towersOfHanoi(n, frm, using, to):
    """ Prints a list of legal moves to solve the Tower of Hanoi problem for \( n \) disks. ""
    if n == 1:
        makeMove(frm, to)
    else:
        towersOfHanoi(n-1, frm, to, using)
        makeMove(frm, to)
        towersOfHanoi(n-1, using, frm, to)
```

It’s pretty clear that the number of moves (and recursive calls) is defined by:

\[
M(1) = 1 \\
M(n) = 1 + 2 \times M(n - 1), \text{ for } n > 1
\]

But this has solution:

\[
M(n) = 2^n - 1
\]

The number of moves is also the number of recursive calls.

```python
>>> for i in range(0, 64, 5):
...    print( format(i, "3d"), format(towersMoveCount(i), "20d"))
...
... 0 0
... 5 31
20 1048575
25 33554431
30 1073741823
35 34359738367
40 109951162775
45 35184372088831
50 1125899906842623
55 36028797018963967
60 1152921504606846975
```