Recursion and Complexity

How do we reason about the complexity of a recursive function?
Formal tool: recurrence relations.
Plus more informal counting arguments.

Instead of "how many times around the loop?":
"how many recursive calls?"
def fact(n):
    if n <= 1:
        result = 1
    else:
        result = n * fact(n-1)
    return result

let T(n) = amount of time to compute fact(n).
Don't need exact solution for T(n), just big-O

Base case: n ≤ 1
takes small constant amount of time to compute

Give that constant an arbitrary name:
T(n) = a, n ≤ 1
def fact(n):
    if n <= 1:
        result = 1
    else:
        result = n * fact(n-1)
    return result

suppose n > 1.
What must we do to compute factorial(n)?
• compare n <= 1
• call fact(n-1)
• multiply by n
• return result

Time to call fact(n-1)? T(n-1)

So T(n) = T(n-1) + b, some constant b.
Our formula for $T(n)$ is a recurrence relation:

- $T(n) = a$, $n \leq 1$
- $T(n) = T(n-1) + b$, $n > 1$

for some constants $a$ and $b$

$T(n)$ is well-defined, but how do we get the big-$O$? We need a \textit{closed-form solution}: no $T$'s on the right hand side.
Solving Recurrence Relations

In general, not easy to do.
Some recurrence relations have no known solution.
In 121: fairly simple recurrence relations.

To solve:
1. Guess a solution by "unrolling" the relation.
2. (Optional) Use mathematical induction to prove your solution is correct.
"Unrolling" a Recurrence Relation

Start with 1 and work up to n

\[ T(n) = a, \quad n \leq 1 \]
\[ T(n) = T(n-1) + b, \quad n > 1 \]

\[
\begin{align*}
T(1) &= a \\
T(2) &= T(1) + b = a + b \\
T(3) &= T(2) + b = a + b + b = a + 2b \\
T(4) &= T(3) + b = a + 2b + b = a + 3b \\
T(5) &= T(4) + b = a + 3b + b = a + 4b \\
\end{align*}
\]

A pattern is emerging!

\[ T(n) = a + (n-1)b \]

So \( T(n) \) is \( O(n) \)

Recursive factorial has \textit{linear} complexity
Another Way To "Unroll"

Start with $n$ and work down to 1

$T(n) = a, \ n \leq 1$

$T(n) = T(n-1) + b, \ n > 1$

$T(n) = T(n-1) + b$
$\quad = [T(n-2)+b] + b = T(n-2) + 2b$
$\quad = [T(n-3)+b] + 2b = T(n-3) + 3b$
$\quad = T(n-4) + 4b$

A pattern is emerging!
After $k$ steps, we get $T(k) = T(n-k) + kb$.

Unroll until we have $T(1)$ on the right hand side.
Goal $n-k = 1$, means $k = n-1$.
With formula above, $T(n) = T(1) + (n-1)b = a + (n-1)b$

Same conclusion as before.
$T(n) = a + (n-1)b = O(n)$
def fact(n):
    if n <= 1:
        result = 1
    else:
        result = n * fact(n-1)
    return result

Chain of calls:
    fact(n) calls fact(n-1) calls fact(n-2) calls.... fact(2) calls fact(1).

How many calls? n-1
How long does each call take? constant time.
So the function takes O(n) time.
Recursive Binary Search

def binSearch(lis, target, low=0, high=None):
    if high == None:
        high = len(lis) - 1

    if low > high:
        return -1

    mid = (low+high)//2
    if lis[mid] == target:
        return mid # we found it

    elif target < lis[mid]:
        return binSearch(lis, target, low, mid-1)

    else: # target > lis[mid]
        return binSearch(lis, target, mid+1, high)
def binSearch(lis, target, low=0, high=None):
    if high == None:
        high = len(lis) - 1

    if low > high:
        return -1

    mid = (low+high)/2
    if lis[mid] == target:
        return mid # we found it

    elif target < lis[mid]:
        return binSearch(lis, target, low, mid-1)

    else: # target > lis[mid]
        return binSearch(lis, target, mid+1, high)

For a list section of size n:
How much time is taken outside of recursive calls?

Answer: O(1) -- constant.
Recursive Binary Search

Time for binary search of list section of size $n$:
small constant + time for recursive calls
call the constant $a$

Time for binary search of list section of size $n$:
Case 1: if $n = 0$: time is $a$
Case 2: if the target is at the midpoint: time is $a$
Case 3: $a +$ time taken by recursive call.
Worst-case scenario: ignore Case 2.
Now reason about sequence of recursive calls.

Critical question: how many calls?
Worst case: $\log_2 n$. So total time is $a*\log_2 n$.
Function is $O(\log n)$. 
Reversing a String

def reverse(s):
    if len(s) <= 1:
        return s
    else:
        return reverse(s[1:]) + s[0]

To analyse complexity, need to know complexity of string operations. Assumptions:
• taking slice of length n is O(n)
• string concatenation (+) is O(n) where n is total number of characters
Reversing a String

def reverse(s):
    if len(s) <= 1:
        return s
    else:
        return reverse(s[1:]) + s[0]

Given assumptions, how much time does function take with string of length n, besides the recursive call?

O(n) time – for slice & concatenation.

So there's a constant b such that the work done inside the method (without the recursive call) takes <= bn time.
Recursion Relation For reverse

T(n) = a, n <= 1
T(n) = T(n-1) + bn + c, n > 1

Unroll from 1 to n:
T(1) = a
T(2) = T(1) + 2b + c = a + 2b + c
T(3) = T(2) + 3b + c = a + (2+3)b + 2c
T(4) = T(3) + 4b + c = a + (2+3+4)b + 3c
T(5) = T(4) + 5b + c = a + (2+3+4+5)b + 4c
....
T(n) = a + (2+3+...+n)b + (n-1) b
     = O(1) + O(n^2) + O(n) = O(n^2)
def reverse(s):
    if len(s) <= 1:
        return s
    else:
        return reverse(s[1:]) + s[0]

Sequence of calls:

Total time is:

\[ c \sum_{i=1}^{n} i = c \frac{n(n+1)}{2} = O(n^2) \]
Tower of Hanoi

def hanoi(numDisks, startPeg=1, destPeg=3, tempPeg=2):
    if numDisks == 1:
        print "move disk from", startPeg, "to", destPeg
    else:
        hanoi(numDisks-1, startPeg, tempPeg, destPeg)
        print "move disk from", startPeg, "to", destPeg
        hanoi(numDisks-1, tempPeg, destPeg, startPeg)

Time for a function call minus recursive calls?

O(1) -- constant

hanoi with n disks calls hanoi with n-1 disks twice (with different pegs)
Tower of Hanoi: Call Tree

Each call takes constant time. How many calls?
Tower of Hanoi: Number of Calls

Total number of calls:
\[1 + 2 + 4 + 8 + \ldots + 2^{n-2} + 2^{n-1} = 2^n - 1\]
So Tower of Hanoi is \(O(2^n)\)

Useful summation formula:
\[
\sum_{i=0}^{n} b^i = \frac{b^{n+1} - 1}{b - 1}
\]
Summation Facts You Need To Know

\[ \sum_{i=1}^{n-1} i^p = O\left(n^{p+1}\right) \]

\[ \sum_{i=0}^{n} b^i = O\left(b^n\right) \]