CS234
MODULE 14 - RECURSION

• Why? We all know this!
• Correctness
• Efficiency

Updated: 2018-07-09
Recursion: What is it?

In CS: **Recursion** is when a function calls itself in order to solve a problem.

**e.g.** All of CS115/135

In Math: Recursion is when a function is defined in terms of itself.

**E.g.** the Fibonacci sequence
Recursion: Why?

Often, when you have a problem, a smaller of the version will recur (that means “happen again”).

If you know this recurrence, you can phrase your answer in terms of the recurring problem’s answer.

E.g: “The smallest thing in the list is either the first thing, or the smallest thing in the rest of the list”
Recursion: The Base Case

To write a recursive function, you need one thing besides a recurring sub-problems: A base case. Or base cases.

Otherwise, it will never end! Problems usually cannot get smaller forever. A list is eventually empty. But with real numbers, they absolutely can get smaller forever (floats are approximate reals so they can only get small approximately forever)
Recursion: Basics

- **Base Case:** A small problem with a known answer (for lists: a list with 0 or 1 elements, usually)

- **General Case:** A larger problem
  - **Recursive Case:** A sub-problem that had recurred
  - **Recursive Answer:** The value returned by a recursive call in the recursive case (we’re saying recursion a lot)
Recursion: Making it work

The recursive case should be closer to the base case. If not, you are caught in infinite regress.

It’s not always easy to gauge progress.
Termination

The Collatz Function: Does it terminate?

def f(n):
    if n == 1: return
    if n % 2 == 0:
        f(n // 2)
    else:
        f(3*n+1)
The Collatz Conjecture

As far as we know, it always terminates. That is, if you start with any positive integer \( n \), it will always eventually reach 1 (the base case) and terminate.

However, it has so far resisted all attempts to prove why.
Termination of Loops

Proving that loops terminates is just as hard.

def Collatz(n):
    while n > 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = n * 3 + 1
Recursion: Why Not?

Recall: Every time you call a function, Python (or C, C++, Racket, Pascal, Visual Basic, blah blah blah) needs to:

- Allocate space for parameters and local variables
- Remember what it was doing so it can resume when the function returns

Meaning: A recursive function might look like it allocates O(1) memory, but it could be O(n)
Recursion: Memory

def in_order(node):
    if node is not None:
        in_order(node.left)
        print(node.item)
        in_order(node.right)

Memory required: $O(1)$ per call
Maximum calls at once: $O(h)$
Total Memory: $O(1) \times O(h) = O(h)$
So why?

Recursion can be more natural!
Recursion can be more elegant!

Or, at the very least, it can leave the programmer feeling smug and self-satisfied.

Remember doing in-order without recursion? That’s why.
Recursion: Not inefficient by definition

The non-recursive in-order still required a stack. The stack holds all nodes that we are currently to the left of. (Or, that should have been your solution!)

I.e. the stack required $O(h)$ memory.
Analysing Recursion

As you recall, if your CS function recurs, the Math function $T(n)$ will also recur.

def add(a, b): # b must be a positive int
    if b == 0: return a
    return 1 + add(a, b-1)

$T(n) = O(1) + T(n-1)$ : (where n is b)
$T(0) = O(1)$
Closed Form

We don’t need our precious lookup sheet anymore.

\[ T(n) = O(1) + T(n-1) \]
\[ T(n) = O(1) + O(1) + T(n-2) \]
\[ T(n) = 3 \times O(1) + T(n-3) \]

... 
\[ T(n) = n \times O(1) + T(0) = O(n) + O(1) = O(n) \]
def bin_search(lst, item, start, end):
    if start > end: return False
    mid = (start + end) // 2
    if lst[mid] == item: return True
    elif lst[mid] > item:
        return bin_search(lst,item,start,mid-1)
    else:
        return bin_search(lst,item,mid+1,end)
Binary Search, Closed Form

T(n) = O(1) + T(n/2) \quad T(1) = O(1)
T(n) = 2 * O(1) + T(n/4)
T(n) = 3 * O(1) + T(n/8)

\ldots

T(n) = k * O(1) + T(n / 2^k)

If k = \log_2 n then:

T(n) = O(\log n) + T(1) = O(\log n)