# Reminders

- Academic integrity is important
  - Do not plagiarize, we will run plagiarism detection software
  - Do not give away your code!
  - Penalties can be 0 in the course and academic suspension

- Instead, use all resources we provide you
  - My office hours
  - Other instructors' office hours
  - Tutorials (time permitting)
  - Help Centre
  - Piazza

# CSC 148 Winter 2017

## Week 4

## Recursion

Bogdan Simion

bogdan@cs.toronto.edu

http://www.cs.toronto.edu/~bogdan

University of Toronto, Department of Computer Science

# Recursion is complicated ...

*"In order to understand recursion ...*

*you must first understand recursion"*

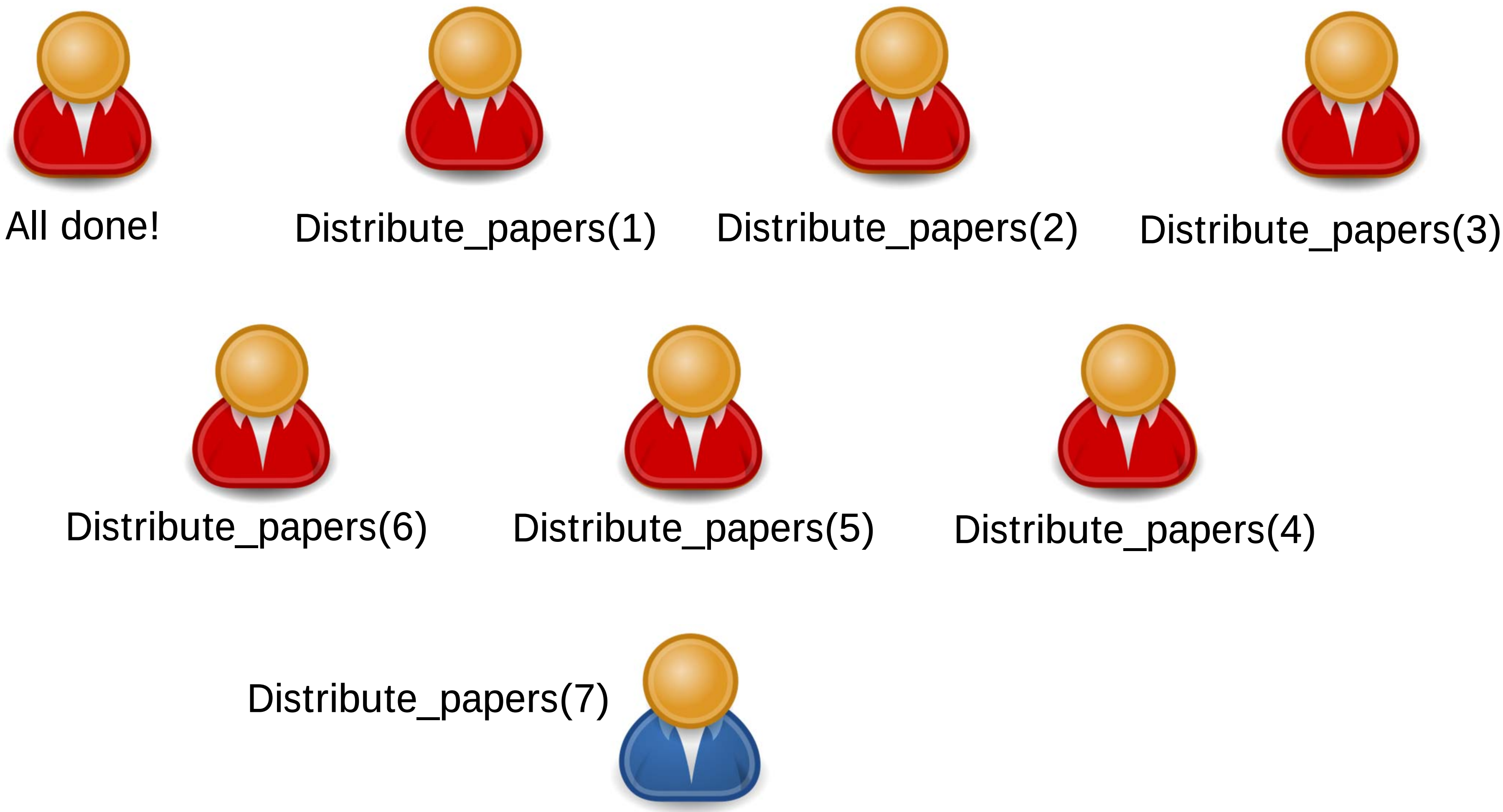
# What is recursion?

- Solve a problem by using an algorithm that calls itself on a smaller problem

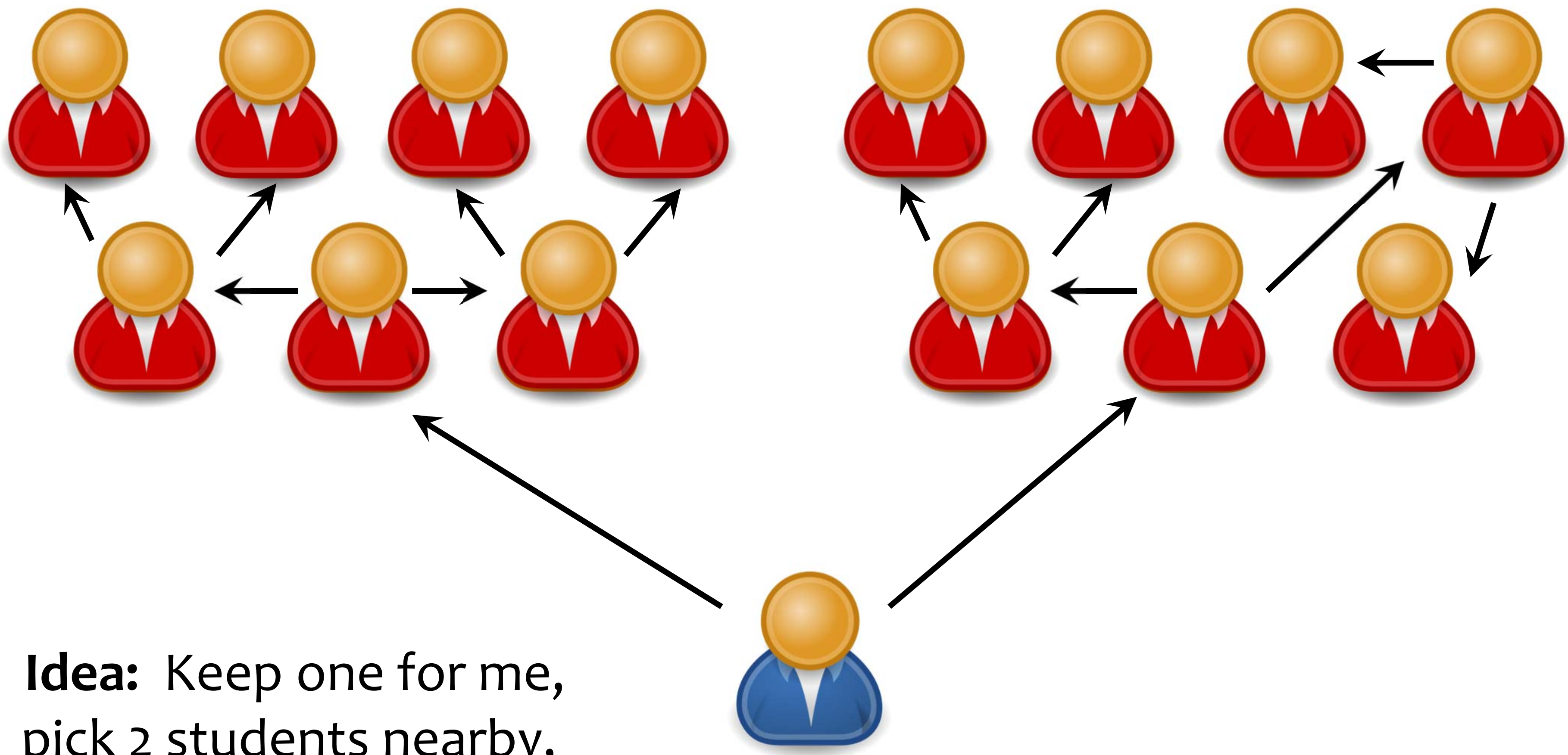
- With each call, the problem becomes simpler

- At some point, the problem becomes trivial!

# Using recursion – example 1

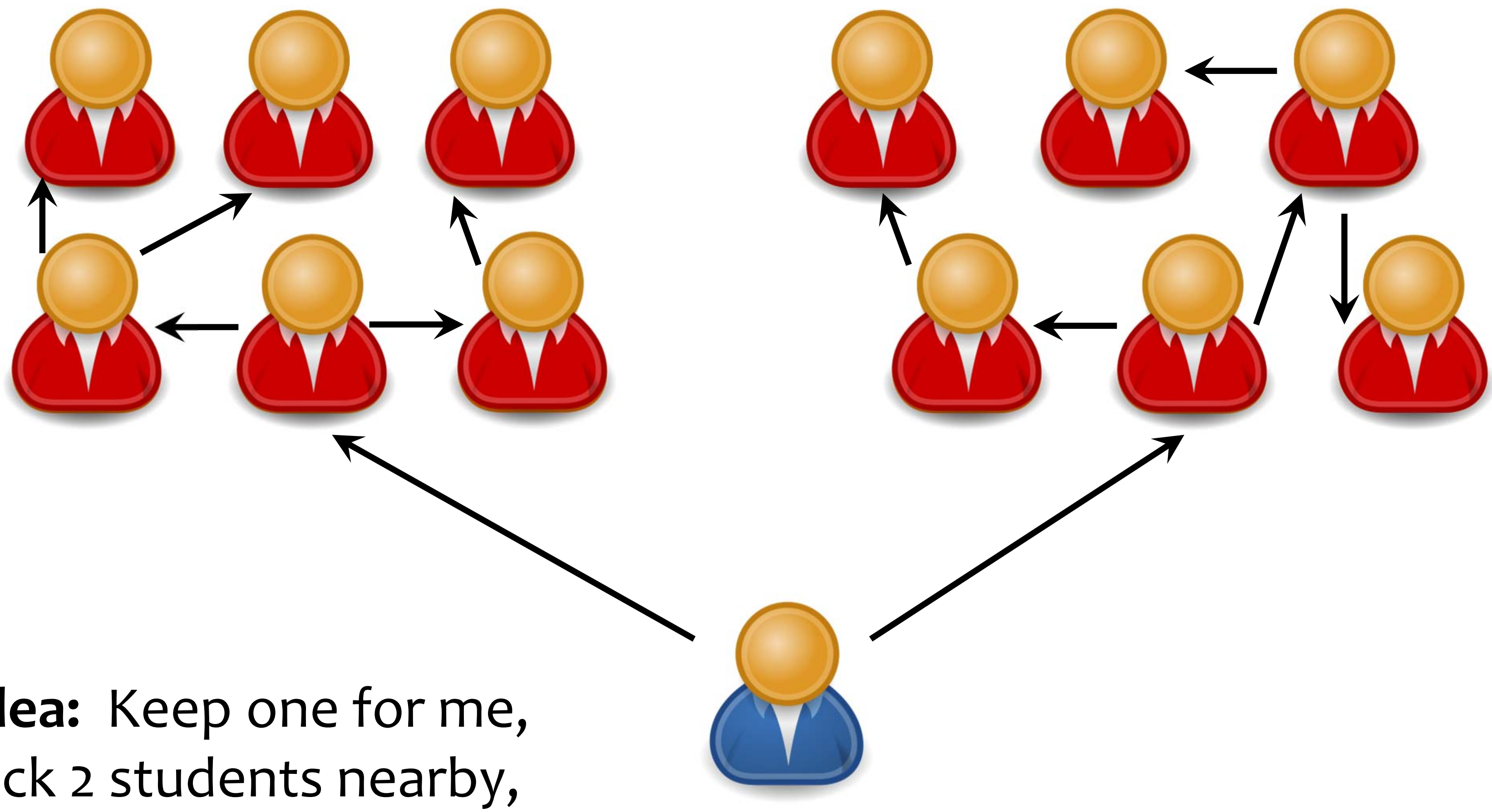


# Using recursion – example 2



**Idea:** Keep one for me, pick 2 students nearby, and give each half of your remaining pile

# Using recursion – example 2



**Idea:** Keep one for me, pick 2 students nearby, and give each half of your remaining pile

# What is recursion?

- *“In order to understand recursion ...*

  *you must first understand recursion”*

- Actually, to understand recursion, one must understand its recursive components ...

# Recursion types

- Depends on how we split the problem
  - "N-1" approach: handle one entity, then call the recursion for N-1 entities
  - Divide in 2 or more subproblems: apply recursion for each half, quarter, etc. of the problem
  - Other ways (more later..)

# Programmer perspective

- Recursion is when a function calls itself directly
  - (mostly .. we won't talk about indirect recursion)
- Goal:
  - Calls itself to solve a smaller part of the problem, using the same function/algorithm
- In some cases, we need to combine the solution!

# Recursion – more formally

- Recursion has 2 phases/steps:

  1. Base case
     - Simplest problem, cannot break it down further
     - This is where we stop recursing
  2. Recursive decomposition step
     - Breaks down the problem into smaller, “similarly-solvable” subproblems
     - Must guarantee to eventually get to the base case

# Sum of list elements

List of integer elements: List = [3, 4, 5]
What's the sum of elements? Solve recursively.

```
def sum_list(L):
    if len(L) == 0:
        return 0
    else:
        return L[0] + sum_list (L[1:])

# main program
...
print(sum_list(List))
```

Sure, we could just use predefined sum(List), or use a simple for loop.

Assume for now that we want an alternative solution using recursion.

# Tracing recursion

List = [3, 4, 5]
What's the sum of elements? Solve recursively.

```
def sum_list(L):
    if len(L) == 0:
        return 0
    else:
        return L[0] + sum_list (L[1:])

# main program
...
print(sum_list(List))
```

Main program sum_list([3,4,5]) ?

sum_list([3,4,5]) → 3 + sum_list([4,5])

sum_list([4,5]) → 4 + sum_list([5])

sum_list([5]) → 5 + sum_list([])

sum_list([]) → 0

# More complex problems

- Why do all this? This could simply be solved with predefined 'sum' function
- What if L's elements can be lists themselves?
  - L = [ 1, [5,3], 8, [4,[9,7]] ]

Will this work?

```
s = 0
for elem in L:
    s += elem
```

What about this?

```
s = 0
for elem in L:
    if isinstance(elem, list):
        for subelem in elem:
            s += subelem
    else:
        s += elem
```

- Nested lists can occur at any depth ⇒ complicated!

# Sum of list elements – nested lists

- L = [1, [5,3], 8, [4,[9,7]]]

recursive step

base case

```
def sum_list(L):
    if  isinstance(L, list):
        s = 0
        for elem in L:
            # calculate the sum of the sublist “elem” recursively
            s += sum_list(elem)
        return s
    else:
        return L
```

# Sum of list elements – nested lists

- L = [1, [5,3], 8, [4,[9,7]]]

recursive step

base case

```
def sum_list(L):
    if isinstance(L, list) :
        s = 0
        for elem in L:
            # calculate the sum of the sublist "elem" recursively
            s += sum_list(elem)
        return s
    else:
        return L
```

⇨ We could rewrite the recursive step to be more "python-style":

```
if isinstance(L, list) :
    return sum([sum_list(elem) for elem in L])
```

# Sum of list elements – nested lists

- L = [1, [5,3], 8, [4,[9,7]]]

recursive step

base case

```
def sum_list(L):
    if isinstance(L, list) :
        return sum([sum_list(elem) for elem in L])
    else:
        return L
```


Exercise sheet ...
Let's trace a few examples ..

# Tracing to understand recursion

- 1. What helper methods does this function call?
  - `sum(...), isinstance(...), sum_list(...) itself`
- 2. Trace the call: sum_list(27)
  - `27`
- 3. Trace this call: sum_list([4, 1, 8])
  - `--> sum([ sum_list(4), sum_list(1), sum_list(8) ])`
  - `--> sum([      4,            1,            8      ])`
  - `--> 13`
- 4. Trace this call: sum_list([4])
  - `--> sum([ sum_list(4) ])`
  - `--> sum([4])`
  - `--> 4`
- 5. Trace this call: sum_list([])
  - `--> sum([])`
  - `--> 0`

# Tracing to understand recursion

- 6. Trace this call: sum_list( [4, [1, 2, 3], 8] )

```
--> sum([ sum_list(4), sum_list([1,2,3]), sum_list(8) ])
--> sum([        4,              6,                 8 ])
--> 18
```

- 7. Trace this call: sum_list( [[1, 2, 3], [4, 5], 8] )

```
--> sum([ sum_list([1,2,3]), sum_list([4,5]), sum_list(8) ])
--> sum([          6,               9,                8 ])
--> 23
```

- 8. Trace this call: sum_list( [1,  [2, 2],  [2, [3, 3, 3], 2]] )

```
-->sum([sum_list(1), sum_list([2,2]), sum_list([2,[3,3,3],2])])
-->sum([    1,             4,        sum_list(2),  sum_list([3,3,3]),  sum_list(2)] )
-->18
```

- 9. Trace this call: sum_list( [1,  [2, 2],  [2, [3, [4, 4], 3, 3], 2]] )

```
-->sum([ sum_list(1), sum_list([2,2]),
         sum_list([2, [3, [4, 4], 3, 3], 2])  ])
-->sum([ 1, 4, 21])
-->26
```

- Let's try depth 37 ...

# Announcements ...

- A1 partnerships
- Special consideration form
- Remember: anonymous feedback form

# Recursion (continued)

# Depth of a list

- Examples:

```
>>> L1  = [1, 3, 4, 2, 9]
>>> depth(L1)
1
>>> L2  = [1, 3, [4, 2], 9]
>>> depth(L2)
2

>>> L3 = [1, [5,3], 8, [4,[9,7]]]
>>> depth(L3)
???
```

- How can we calculate the depth of any list?

- Depth of a list = 1 plus the maximum depth of L's elements if L is a list, otherwise its depth is 0.

# Depth of a list

- Depth of a list = 1 plus the maximum depth of L's elements if L is a list, otherwise its depth is 0.
- The definition is almost exactly the Python code you write!
- Example: L = [1, [5,3], 8, [4,[9,7]]]

base case

recursive step

```
def depth(L):
    if not isinstance(L, list) :   # L is not a list
        return  0
    else:
        return  1 + max([depth(elem)  for elem in L])
```

Any problems here?
- max not defined on an empty list

Make sure we dealt with the special case of a non-list ..
Ensure problem is broken down into a smaller one, that a base exists, etc...

Let's trace a few examples ..

# Trace to understand recursion

- Trace is increasing complexity; at each step, fill in values for recursive calls that have (basically) **already been traced**.

```
Trace depth(17)   => 0

Trace depth([])   => 1

Trace depth([3, 17, 1])
=> 1 + max([depth(3), depth(17), depth(1)])
=> 1 + max([0, 0, 0])
=> 1
Trace depth([5, [3, 17, 1], [2, 4], 6])
=> 1 + max([depth(5), depth([3,17,1]), depth([2,4]), depth(6)])
=> 1 + max([0, 1, 1, 0])
=> 2
```

When tracing on paper, no need to recurse on these again, we already know how to handle single-nested lists, so just replace call with the result!

```
Trace depth([14, 7, [5, [3, 17, 1], [2, 4], 6], 9])
=> Practice!
```

# Maximum number in a (nested) list

- Maximum of a list = the maximum out of all of L's elements.
- Use the built-in max, much like sum
- Logical steps:

```
How would you find the max of non-nested list?
  max([...])

How would you build a new list using a comprehension?
 [do_something_to_element(elem) for elem in L]

What should you do with list items that were
themselves lists?
  max([max_list(elem) for elem in L])
```

- Get some intuition by tracing through flat lists, lists nested one deep, then two deep, etc..

# Maximum of a nested list

- Maximum of a list = the maximum out of all of L's elements.
- Base case?

    If some element x is not a list, then x's maximum is the element itself
- Recursive step?

    If some element x is a list, then calculate its maximum recursively with the same algorithm applied for L.

- L = [1, [5,3], 8, [4,[9,7]]]

base case / recursive step

```
def rec_max (L):
    if  not isinstance(L, list) :
        return L
    else:
        return max([rec_max (x)   for x in L])
```

Any problems here?

- max not defined on an empty list
- special case if L is None..

Preconditions?

# Trace to understand recursion

- Trace from simple to complex; fill in already-solved recursive calls

```
Trace rec_max([3, 5, 1, 3, 4, 7])

 => max([rec_max(3), rec_max(5), ...])
 => max([3, 5, 1, ...])
 => 7

Trace rec_max([4, 2, [3, 5, 1, 3, 4, 7], 8])
 => max([ rec_max(4), rec_max(2),
          rec_max([3, 5, 1, 3, 4, 7]),
          rec_max(8) ])
 => max([ 4, 2, 7, 8 ])
 => 8
```

As calculated above!
Reminder: **When you trace, only** unroll what we don't know how to solve yet!

```
Trace rec_max([6, [4, 2, [3, 5, 1, 3, 4, 7], 8], 5])
=> max([ rec_max(6),
         rec_max([4, 2, [3, 5, 1, 3, 4, 7], 8]),
         rec_max(5) ])
=> max([ 6, 8, 5 ])
=> 8
```

As above! **When tracing on paper**, we now know how to calculate rec_max on a list of depth 2, so don't unroll it

# Infinite lists

- What if depth, rec_max have a base case, but the input is an infinite list?
- How can we generate an infinite list?

```
L = [1, 2, 3]
L.append(L)
print(depth(L))
```

# Concatenate strings in a (nested) list

- Concatenate all the strings in a list with possible nesting at any level
- Logical steps:

```
How would you concatenate strings in non-nested list?


How would you build a new list using a comprehension?


What should you do if list items are themselves lists?
```

- Let's implement this ..
- Get some intuition by tracing through flat lists, lists nested one deep, then two deep, etc..

# Trace to understand recursion

- Trace from simple to complex; fill in already-solved recursive calls

```
Trace concat("The cow goes moo!")
 => "The cow goes moo!"

Trace concat(["The", "cow", "goes", "moo", "!"])
 => " ".join([concat("The"), concat("cow"), ...])
 => " ".join(["The", "cow", ...])
 => "The cow goes moo !"

Trace concat(["This", "sentence", "is actually",
"constructed", ["from", "other smaller"], "strings"])
=> " ".join([ "This", "sentence", "is actually", "constructed",
              concat(["from", "other smaller"]),
              "strings"])
```

We know how to calculate this already, no need to unroll it more, for the purposes of tracing on paper!

```
=> "This sentence is actually constructed from other smaller
strings"
```

# Distributing papers ...

- How would we write code for this ?
  - Assume: our pile of papers is a list of papers, each represented by their unique paper number. Each student takes 1 paper, then distributes further two halves of their remaining pile to two other students who will do the exact same thing. When no more papers left or only 1 left, then the problem becomes trivial.

# Assignment 1

- Quick demo ..
- More on this next time ..