

Lecture 11: Reversing List and Sorting

Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

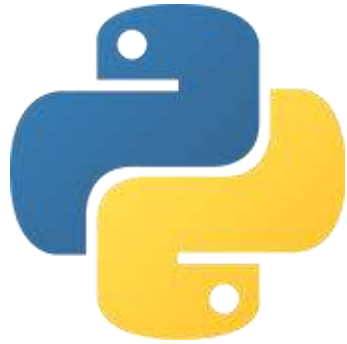
Previously in CSE 1729...



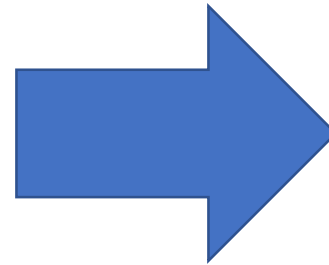
```
1 | (define x (list 1 2 3 ))  
2 | x
```

```
(define (square-list k)  
  (if (= k 0)  
      (list 0)  
      (cons (* k k)  
             (square-list (- k 1)))))
```

How to iterate through a list?



```
1 l = [1, 2, 3, 4 ,5]
2 #iterate through the loop
3 for i in range(0, len(l)):
4     print(l[i])
```



```
C:\WINDOWS\system32\cmd.exe
1
2
3
4
5
```

MAPPING A FUNCTION OVER A LIST

- Applying function to each element of a list is called *mapping*. It's a powerful tool.

```
(define (map f items)
  (if (null? items)
      '()
      (cons (f (car items))
            (map f (cdr items)))))
```

- Then, for example:

```
> (map (lambda (x) (* x x)) '(0 1 2 3 4 5 6))
'(0 1 4 9 16 25 36)
> (map (lambda (x) (* x x)) '())
'()
```

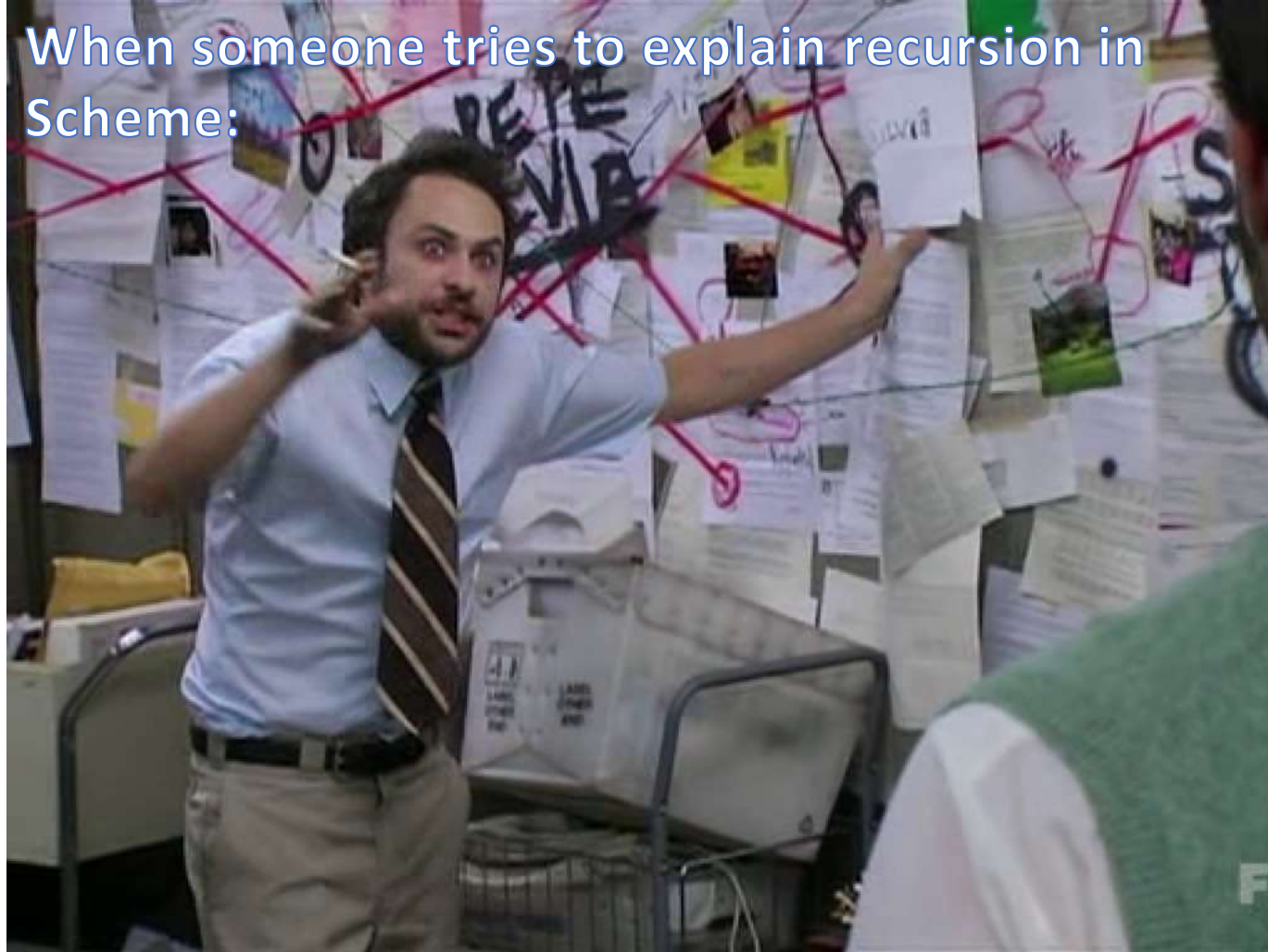
BACK TO ASYMMETRY: REVERSING A LIST. NOT AS EASY AS YOU THOUGHT...


- Reversing a list. One strategy: peel off the first element; reverse the rest; append the first element to the end. This yields:

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```

Get ready for some complicated pictures...

When someone tries to explain recursion in Scheme:



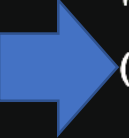


```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2)))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```

reverse call 1

Items = (1,2,3)

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



reverse call 1


Items = (1,2,3)

append call 1



(reverse (cdr items)) **+** (list (car items))=1


```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
               (list (car items)))))
```



reverse call 1

Items = (1,2,3)

append call 1


(reverse (cdr items)) + (list (car items))=1



reverse call 2

Items = (2,3)

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



append call2

(reverse (cdr items)) **+** (list (car items))=2

reverse call 1

Items = (1,2,3)


append call 1

(reverse (cdr items)) **+** (list (car items))=1

reverse call 2

Items = (2,3)

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



reverse call 1

Items = (1,2,3)


append call 1

(reverse (cdr items)) **+** (list (car items))=1



reverse call 2

Items = (2,3)



append call 2


(reverse (cdr items)) **+** (list (car items))=2



reverse call 3

Items = (3,)

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



reverse call 1

Items = (1,2,3)

append call 1

(reverse (cdr items)) **+** (list (car items))=1

reverse call 2

Items = (2,3)

append call 2

(reverse (cdr items)) **+** (list (car items))=2

reverse call 3

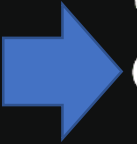
Items = (3,)

append call 3

(reverse (cdr items)) **+** (list (car items))=3

Now focus on append call 3...

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



append call 3


(reverse (cdr items)) **+** (list (car items))=3

Items = (3,)

(list (car items))=3

(reverse (cdr items)) = The empty list

Now we go to reverse call 4



```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```

append call 3


(reverse (cdr items)) **+** (list (car items))=3

reverse call 4

Items = '()

Now we go to reverse call 4

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



append call 3


(reverse (cdr items)) **+** (list (car items))=3

reverse call 4

Items = '()

Items are indeed null so we can
finally return!

Back to append call 3!



```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```

append call 3


()' **+** (list (car items))=3

reverse call 4

Items = '()

Back to append call 3!

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```




append call 3

()' **+** (list (car items))=3

List 1 is the empty list so just return the second argument...which is 3

Remember this confusing picture? Let's start at append all 3 which is now ready to return....

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



reverse call 1

Items = (1,2,3)

append call 1

(reverse (cdr items)) + (list (car items))=1

reverse call 2

Items = (2,3)

append call 2

(reverse (cdr items)) + (list (car items))=2

reverse call 3


Items = (3,)

append call 3

(reverse (cdr items)) + (list (car items))=3

Remember this confusing picture? Let's start at append call 3 which is now ready to return....

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



reverse call 1

Items = (1,2,3)

append call 1

(reverse (cdr items)) + (list (car items))=1

reverse call 2

Items = (2,3)

append call 2

(reverse (cdr items)) + (list (car items))=2

reverse call 3

Items = (3,)


append call 3

(reverse (cdr items)) + (list (car items))=3

Returns (3,)

Remember this confusing picture? Let's start at append call 3 which is now ready to return....

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



reverse call 1

Items = (1,2,3)

append call 1

(reverse (cdr items)) + (list (car items))=1

reverse call 2

Items = (2,3)

append call 2

(reverse (cdr items)) + (list (car items))=2

reverse call 3


Returns (3,)

append call 3

(reverse (cdr items)) + (list (car items))=3

Returns (3,)

Now let's just focus on append call 2



```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```

append call 2

(reverse (3,)) + (list (car items))=2

All arguments are lists so we can actually evaluate the method (by recursively calling append).

Now let's just focus on append call 2

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```


append call 2

(rev (3,) ems)) + (list (car items))=2

I will not draw out the recursion but you can see that when are appending two lists, we have to recurse until we hit the end of list 1.

Now we can trace all the way back to reverse call 1 and get the solution

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```



reverse call 1

Items = (1,2,3)

Returns (3,2,1)

append call 1

Returns (3,2)

⊕ (list (car items))=1

reverse call 2

Returns (3,2)

append call 2

Returns (3,)

⊕ (list (car items))=2

reverse call 3

Returns (3,)

append call 3

(reverse (reverse (cdr items)))=3

Returns (3,)

Is this a good code in terms of runtime?

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2)))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```


EVEN AFTER ALL THAT WORK: THIS REVERSE HAS A SERIOUS PROBLEM


- How long does it take to reverse a list?
 - (One good way to measure the running time of a Scheme function is to measure the total number of procedure calls it generates.)
- If the list has n elements, reverse is called on each suffix.
 - There are about n of these, which looks OK.
- However, each reverse also calls append.
 - If reverse is called with a list of k elements, the append needs to step all the way through this list in order to get to the end, generating k total calls to append.
- All in all, this is roughly $n + (n - 1) + \dots + 1$ calls; about $\frac{n^2}{2}$.
 - Surely we can reverse a list in roughly n steps!

APPENDING THE CAR ONCE THE REST OF THE LIST IS REVERSED IS COSTLY...

- ...what if we pass the `car` along as a parameter, asking our next-in-line to take care of the job of appending it to the resulting list?
- Specifically, consider the function (`reverse-and-append list rest`):
 - it should reverse list, append rest onto the end, and return the result.

```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
      (reverse-and-append (cdr r-items)
                          (cons (car r-items) rest))))
```

- Note: this simply generates $\sim n$ recursive calls!



```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
      (reverse-and-append (cdr r-items)
                          (cons (car r-items) rest))))
```

reverse call 1


r-items = (1,2,3)
rest = '()

```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
      (reverse-and-append (cdr r-items)
                          (cons (car r-items) rest))))
```

RA call 1

r-items = (1,2,3)
rest = '()

(cdr r-items) = (2,3) (car r-items) = 1



```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
      (reverse-and-append (cdr r-items)
                          (cons (car r-items) rest))))
```

RA call 1

r-items = (1,2,3)
rest = '()

(cdr r-items) = (2,3) (car r-items) = 1

RA call 2

r-items = (2,3)
rest = (1,)

```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
      (reverse-and-append (cdr r-items)
                          (cons (car r-items) rest))))
```

RA call 1

r-items = (1,2,3)
rest = '()


(cdr r-items) = (2,3) (car r-items) = 1

RA call 2

r-items = (2,3)
rest = (1,)

(cdr r-items) = (3,) (car r-items) = 2

```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
      (reverse-and-append (cdr r-items)
                          (cons (car r-items) rest))))
```



RA call 1

r-items = (1,2,3)
rest = '()

(cdr r-items) = (2,3) (car r-items) = 1

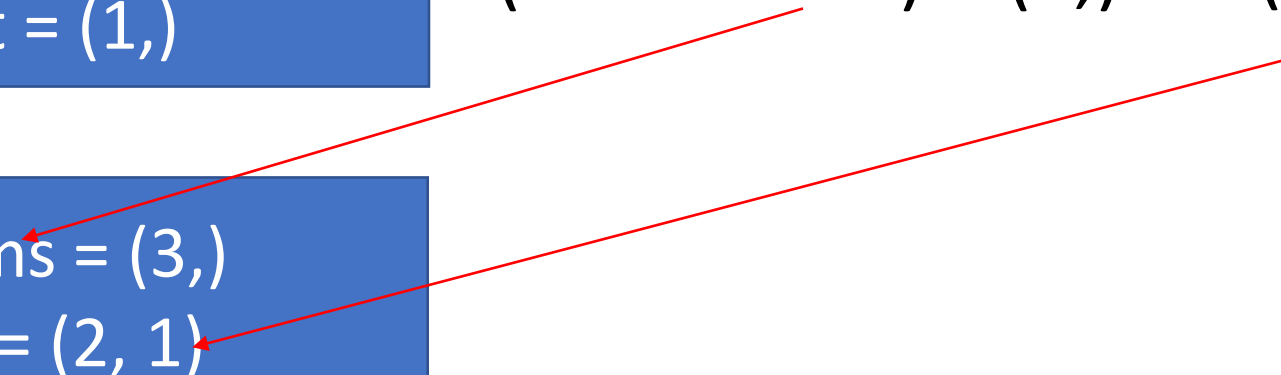
RA call 2


r-items = (2,3)
rest = (1,)

(cdr r-items) = (3,) (car r-items) = 2

RA call 3

r-items = (3,)
rest = (2, 1)



```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
       (reverse-and-append (cdr r-items)
                                                                (cons (car r-items) rest))))
```

RA call 1

r-items = (1,2,3)
rest = '()

(cdr r-items) = (2,3) (car r-items) = 1

RA call 2


r-items = (2,3)
rest = (1,)

(cdr r-items) = (3,) (car r-items) = 2

RA call 3

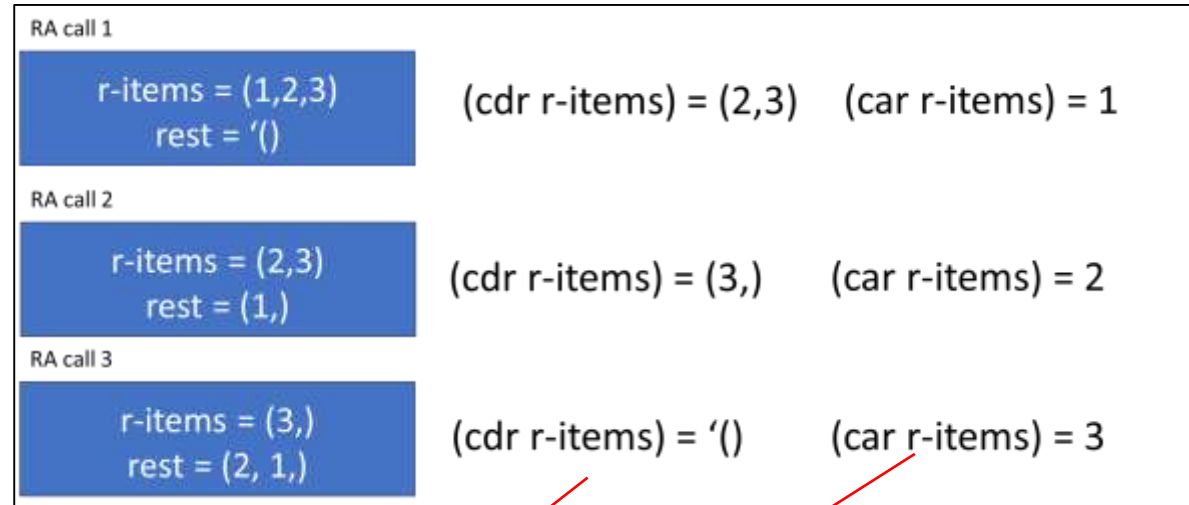
r-items = (3,)
rest = (2, 1)

(cdr r-items) = '() (car r-items) = 3

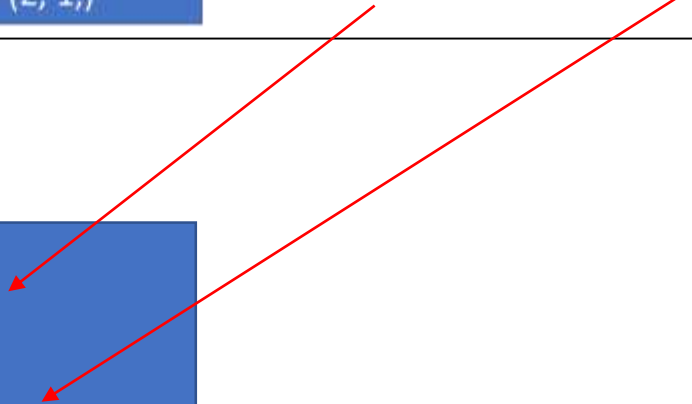



```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
      (reverse-and-append (cdr r-items)
                          (cons (car r-items) rest))))
```

We are now calling RA with r-items as the empty list. This means we can just return rest.



RA call 4



```
r-items = '()
rest = (3,2, 1)
```

What was the difference again between these two codes?

```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
      (reverse-and-append (cdr r-items)
                          (cons (car r-items) rest))))
```

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
                (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
                (list (car items)))))
```

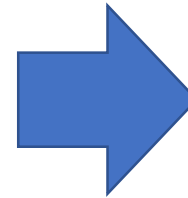
Answer: No need for the append function to combine lists, we build the reverse list as we recurse.

SORTING A LIST: SELECTION SORT

- Goal
 - Sort a list of value (integers) in increasing order
- Idea
 - Find the minimum,
 - Extract it (remove it from the list),
 - Sort the remaining elements,
 - Add the minimum back in front!

First just think about finding the smallest in simple Python code...

```
1 import sys #Used for getting the maximum float value
2
3 def FindMin(inputList):
4     currentMin = sys.float_info.max
5     #Go through all elements in the list
6     for i in range(0, len(inputList)):
7         #Check if the current element is the smallest
8         if l[i] < currentMin:
9             currentMin = l[i]
10    return currentMin
11
12 l = [1, 3 , 0, 5]
13 print(FindMin(l))
```

A screenshot of a Windows command prompt window. The title bar shows the path C:\WINDOWS\system32\cmd.exe. The command prompt shows the output of the Python code: 0, followed by the prompt Press any key to continue . . .

```
C:\WINDOWS\system32\cmd.exe
0
Press any key to continue . . .
```

Now let's go to Scheme...

First just think about finding the smallest between two numbers:

```
(define (smaller a b) (if (< a b) a b))
```

Now start to think about lists. If a list has only one element by default that is the smallest element:

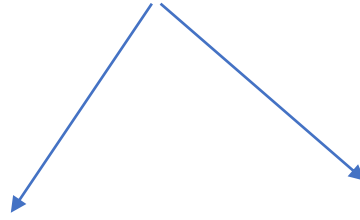
```
(define (smallest l)
  (if (null? (cdr l))
      (car l)
      (smallest (cdr l))))
```

Last step in finding the smallest...

- Compare the current head with the rest of the elements. Pick the smallest.
- Base case: Only one element, so that by default is the smallest.

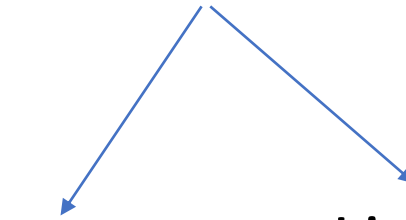
```
(define (smallest l)
  (define (smaller a b) (if (< a b) a b))
  (if (null? (cdr l))
      (car l)
      (smaller (car l) (smallest (cdr l)))))
```

List = (1, 4, 0, 3)



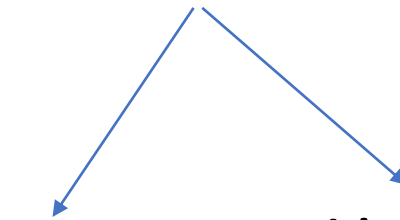
Recur 1: Who is smallest?

List = (4, 0, 3)



Recur 2: Who is smallest?

List = (0, 3)

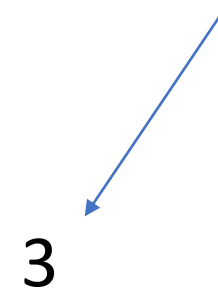


Recur 3: Who is smallest?

0

List = (3)

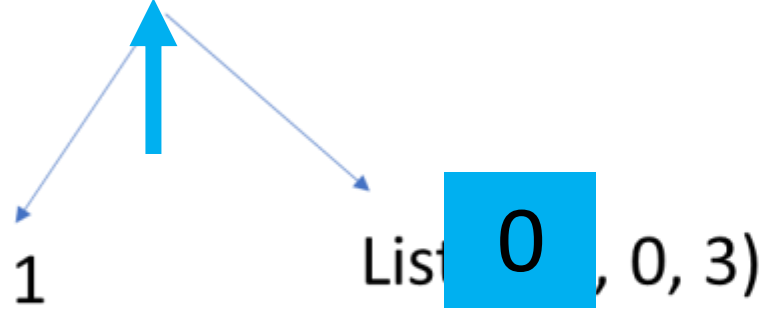
Recur 3: Hit case where (cdr l) is null, return 3



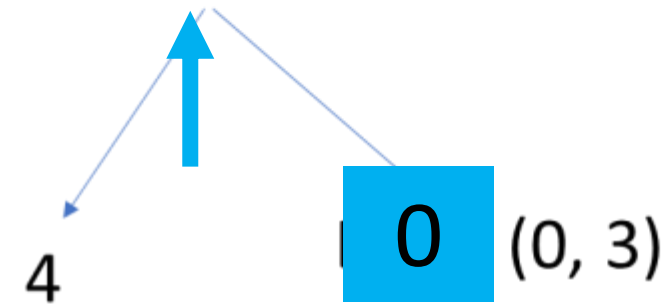
3

List = **0** 4, 0, 3)

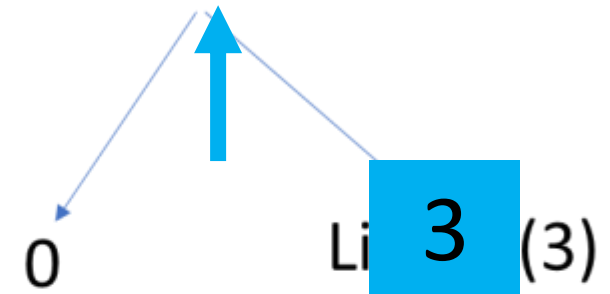
Recur 1: Who is smallest?



Recur 2: Who is smallest?



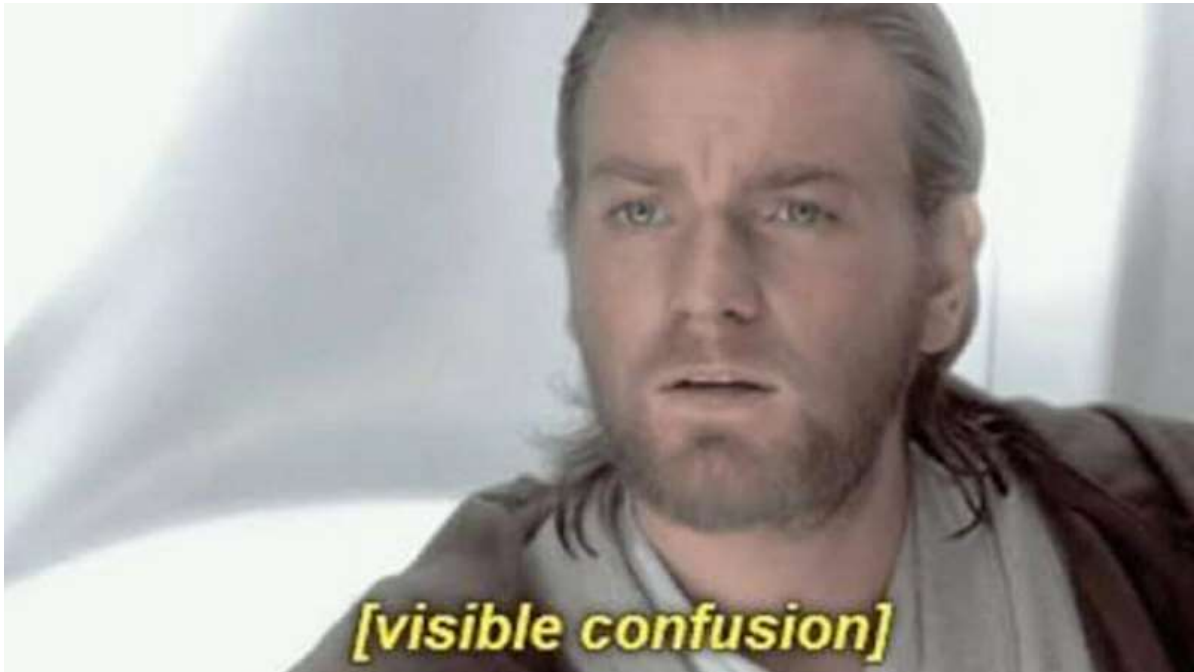
Recur 3: Who is smallest?



Recur 3: Hit case where (cdr l) is null, return 3



Does anyone remember what we were actually trying to do?



- We were trying to sort a list. Things we need to do to sort a list:
 1. Find the smallest element in a list.
 2. Remove the element.

REMOVING FROM A LIST

- Goal
 - Remove a *single occurrence* of a value from a list
- Inductive definition
 - Base case:
 - Easy: empty list
 - Induction:
 - If we have a match: done! Just return the tail.
 - If we don't: remove from the tail and preserve the head.

THE SCHEME CODE

- One plain induction on the list.
 - *v*: the value to remove
 - *elements*: the list to remove it from

```
(define (remove v elements)
  (if (null? elements)
      elements
      (if (equal? v (car elements))
          (cdr elements)
          (cons (car elements)
                (remove v (cdr elements))))))
```

Next time: We'll put all the pieces together and make our awesome sorting algorithm!

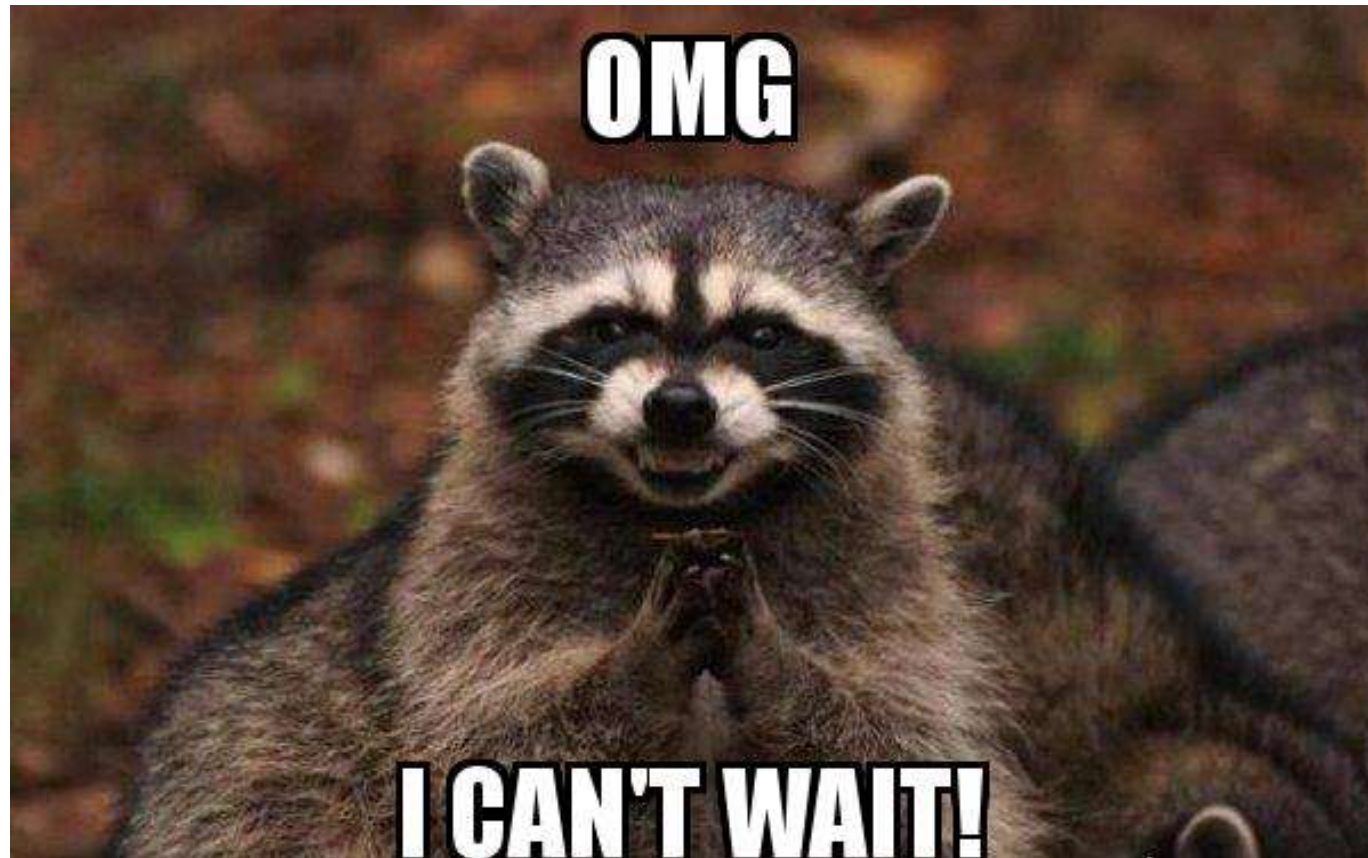


Figure Sources

- https://i.kym-cdn.com/entries/icons/original/000/022/524/tumblr_o16n2kBlpX1ta3qyvo1_1280.jpg
- <https://i.kym-cdn.com/entries/icons/original/000/031/197/visible.jpg>