Attempt 1

```
(define (2nd-max lon)
  (second (sort lon >)))
```

Short, sweet, and O(n log n)

•

Attempt 2

1 Find the largest value in the list.
2 Remove that item from the list.
3 Find the largest value in what's left.

•

1 Find the largest value in the list.
2 Remove that item from the list.
3 Find the largest value in what's left.

```
(define (2nd-max lon)
  (apply max                          ; step 3
         (remove                      ; step 2
             (apply max lon)          ; step 1
             lon)))
```

A little longer, and only O(n).
But it makes three passes...
.

Attempt 3

Our argument contains at least two numbers:

```
(<number> <number> . <list-of-numbers>)
```

... with the usual definition for a list:

```
<list-of-numbers>
  ::= ()
    | (<number> . <list-of-numbers>)
```

.

If we could write a loop, we might...

- Create two local variables, **largest** and **2nd-largest**.

- Initialize the variables using the first two items in the list.

- **Then** look at each item in the rest of the list to see if it is greater than either of the two variables and, if so, update the variables.

.

Create an interface procedure to initialize the variables and start the processing:

```
(define (2nd-max lon)
  (2nd-max-tr
    (max (first lon) (second lon))
    (min (first lon) (second lon))
    (rest (rest lon))))
```

Now we have to write 2nd-max-tr.

.

```
(6 1 2 -3 9 4 -1 2 8 1 2 4)


largest          6
2nd-largest      1
rest             (2 . (-3 9 4 -1 2 8 1 2 4))



.
```

```
(6 1 2 -3 9 4 -1 2 8 1 2 4)


largest         6
2nd-largest     2
rest            (-3 . (9 4 -1 2 8 1 2 4))


.
```

```
(6 1 2 -3 9 4 -1 2 8 1 2 4)


largest          6
2nd-largest      2
rest             (9 . (4 -1 2 8 1 2 4))


.
```

```
(6 1 2 -3 9 4 -1 2 8 1 2 4)


largest         9
2nd-largest     6
rest            (4 . (-1 2 8 1 2 4))


.
```

We could write a 'cond' or an 'if' expression to handle the three cases:

```
(define (2nd-max-tr largest 2nd-largest lon)
  (cond ((null? lon) 2nd-largest)
        (( case #1 ) ...)
        (( case #2 ) ...)
        (( case #3 ) ...) ))
```

The ...'s are ugly and repeat (rest lon) three times.

.

Instead, let's use our interface procedure as inspiration:

```
(define (2nd-max-tr largest 2nd-largest lon)
  (if (null? lon)
      2nd-largest
      (2nd-max-tr
         new value of largest
         new value of second
         new value of lon    ) )))
```

.

Handle (first lon) in first two arguments:

```
(define (2nd-max-tr largest 2nd-largest lon)
  (if (null? lon)
      2nd-largest
      (2nd-max-tr
         new value of largest
         new value of second
         new value of lon      )))
```

•

Handle (rest lon) in the third argument:

```
(define (2nd-max-tr largest 2nd-largest lon)
  (if (null? lon)
      2nd-largest
      (2nd-max-tr
         new value of largest
         new value of second
         new value of lon      )))
```

•

```
(define (2nd-max-tr largest 2nd-largest lon)
  (if (null? lon)
      2nd-largest
      (2nd-max-tr
        (max largest
             (first lon))
        (max 2nd-largest
             (min largest (first lon)))
        (rest lon))))
```

This is order O(n), makes only **one** pass,
and uses only one stack frame.
.

How might we compare these solutions?

- length of the code
- space used at run-time
- time used at run-time
- time to create the program
- ...
- complexity of the code
- ...
- familiarity

.

We have learned a new language in order to:

- learn a new way to think about languages

We have learned a new language in order to:

- learn a new way to think about languages
- learn a new style of programming

-

We have learned a new language in order to:

- learn a new way to think about languages
- learn a new style of programming
- learn patterns of recursive programs

•

We have learned a new language in order to:

- learn a new way to think about languages
- learn a new style of programming
- learn patterns of recursive programs

Now, we use all three to:

- learn how programming languages work

.

Static Properties of Variables

A property is **static** when its value can be determined by looking at the text of a program.

-

Static Properties of Variables

A property is **static** when its value can be determined by looking at the text of a program.

A property is **dynamic** when the program must be executed in order to determine its value.

.

example: data type

```
int sumOfSquares(int m, int n)
{
    return m*m + n*n;
}
```

```
def sum_of_squares(m, n):
    return m*m + n*n
```

•

A Little Language

```
<exp> ::= <varref>
        | (lambda (<var>) <exp>)
        | (<exp> <exp>)
```

•

free and **bound** variables

```
int sumOfSquares( int m, int n )
{
    // m and n are bound
    // to formal parameters

    return m*m + n*n;
}
```

•

A variable **is bound** or **occurs bound** in an expression if it refers to the formal parameter in the expression.

A variable **is bound** or **occurs bound** in an expression if it refers to the formal parameter in the expression.

A variable **is bound** or **occurs bound** in an expression if it refers to the formal parameter in the expression.

A variable **is free** or **occurs free** in an expression if it is not bound.

•

```
<exp> ::= <varref>
        | (lambda (<var>) <exp>)
        | (<exp> <exp>)
```

Free and bound variables in this language:

•

A function definition with no free variables is called a **combinator**.

```
(lambda (f)           ; combinator
  (lambda (x)
    (f (f x))))
```

```
(lambda (f)           ; not a combinator
  (lambda (x)
    (f (g x))))
```

.

This is **not** a combinator:

```
(define sum-of-applications
  (lambda (f x y)
    (+ (f x) (f y))))
```

•