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

Short, sweet, and O(n log n)

.

```
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
  (lambda (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...
.

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>)
```

So we will want an interface procedure.

.

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.

.

```
(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   <- replace w/ (max 6 2)
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      1   <- replace w/ (max 1 2)
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)


.
```

IN PYTHON

```
def second_max(lst):
    largest = max(lst[0], lst[1])
    second  = min(lst[0], lst[1])

    for n in lst[2:]:
        larger  = max(largest, n)
        smaller = min(largest, n)

        largest = larger
        second  = max(second, smaller)

    return second
```

.

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

This is order O(n) and makes only **one** pass!
But, man, 2nd-max-tr is ugly...

.

Use our interface procedure as inspiration:

```
(define 2nd-max-tr
  (lambda (largest 2nd-largest lon)
    (if (null? lon)
        2nd-largest
        (2nd-max-tr
           ; -- handle (first lon)
           new value of largest
           new value of second
           ; handle (rest lon)
           new value of lon      ))))
```

.

```
(define 2nd-max-tr
  (lambda (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.
This is dandy solution, IMHO.
.

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

.

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: 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.

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

Compilers can use static properties of a program to detect errors and to improve program performance.

.

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 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:

.

This is **not** a combinator:

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

.

Quiz 1

60 points total

=> 54 A
=> 48 B
=> 42 C
=> 36 D

quiz average = 45
"What is my course grade?"
.

	pair	list
type predicate	pair?	list?
access procedures	car	first
	cdr	rest
constructor	cons	list

.