Max of a List

Implement the function \texttt{max-item} which returns the biggest number in a list of numbers
Data: list-of-num, obviously

Contract:

; max-item : list-of-num -> num
Examples

(check-expect (max-item '(2 7 5)) 7)
(check-expect (max-item empty) ...) 

Problem: max-item makes no sense on an empty list
Data and Contract, Again

**Data:** nonempty-list-of-num

; A nonempty-list-of-num is either
; - (cons num empty)
; - (cons num nonempty-list-of-num)

**Contract:**

; max-item : nonempty-list-of-num -> num
Examples, Again

(check-expect (max-item '(2 7 5)) 7)
(check-expect (max-item '(2)) 2)
Implementation

No existing functions on non-empty lists, so start with the template

; A nonempty-list-of-num is either
;  - (cons num empty)
;  - (cons num nonempty-list-of-num)

(define (max-item nel)
  (cond
   [(empty? (rest nel)) ... (first nel) ...]
   [else
    ... (first nel)
    ... (max-item (rest nel)) ...]))
(define (max-item nel)
  (cond
    [(empty? (rest nel)) (first nel)]
    [else
      (cond
        [(> (first nel) (max-item (rest nel)))
          (first nel)]
        [else
          (max-item (rest nel))]))))

Implementation Complete
Test

(check-expect (max-item '(2)) 2)

works fine

(check-expect
 (max-item '(1 2 3 4 5 6 7 8 9 10))
10)

works fine

(check-expect
 (max-item '(1 2 3 4 5 6 7 8 9 10
  11 12 13 14 15 16 17 18 19 20
  21 22 23 24 25 26 27 28 29 30))
30)

answer never appears!
The Speed of max-item

Somewhere around 20 items, the max-item function starts to take way too long

Even if you buy a computer that’s 10 times faster, the problem shows up with about 23 items...

How long does a program take to run?
Counting Steps

How long does

\[(+ 1 (* 6 7))\]

take to execute?

Computer speeds differ in “real time,” but we can count steps:

\[(+ 1 (* 6 7)) \rightarrow (+ 1 42) \rightarrow 43\]

So, evaluation takes 2 steps
Steps for max-item and 1 Element

How long does this expression take?

\[(\text{max-item } '(2))\]

\[(\text{max-item } '(2))\]
\[\rightarrow (\text{cond } [(\text{empty? } \text{rest } '(2))) \ (\text{first } '(2))] \ldots)\]
\[\rightarrow (\text{cond } [(\text{empty? } \text{empty}) \ (\text{first } '(2))] \ldots)\]
\[\rightarrow (\text{cond } [\text{true} \ (\text{first } '(2))] \ldots)\]
\[\rightarrow (\text{first } '(2))\]
\[\rightarrow 2\]

5 steps — and any list with one item will take five steps
Steps for max-item and 2 Elements

How long does this expression take?

\[(\text{max-item } '(2 1))\]

\[(\text{max-item } '(2 1))\]
\[\rightarrow (\text{cond} [(\text{empty? } (\text{rest } '(2 1))) \ (\text{first } '(2 1))] \ [\text{else } \ldots])\]
\[\rightarrow (\text{cond} [(\text{empty? } '(1)) \ (\text{first } '(2 1))] \ [\text{else } \ldots])\]
\[\rightarrow (\text{cond} [\text{false} \ (\text{first } '(2 1))] \ [\text{else } \ldots])\]
\[\rightarrow (\text{cond} [\text{else} \ (\text{cond} [(> \ (\text{first } '(2 1)) \ \ldots) \ \ldots] \ [\text{else } \ldots)])]\]
\[\rightarrow (\text{cond} [(> \ (\text{first } '(2 1)) \ (\text{max-item } (\text{rest } '(2 1)))) \ \ldots] \ [\text{else } \ldots])\]
\[\rightarrow (\text{cond} [(> \ 2 \ (\text{max-item } (\text{rest } '(2 1)))) \ \ldots] \ [\text{else } \ldots])\]
\[\rightarrow (\text{cond} [(> \ 2 \ (\text{max-item } '(1))) \ \ldots] \ [\text{else } \ldots])\]
\[\rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots\]
\[\rightarrow (\text{cond} [(> \ 2 \ 1) \ (\text{first } '(2 1))] \ [\text{else } \ldots])\]
\[\rightarrow (\text{first } '(2 1))\]
\[\rightarrow 2\]

14 steps — where 5 came from the recursive call

Are all 2-element lists the same?
Steps for max-item and 2 Elements

\[
\text{max-item } '(1 \ 2)\n\]

\[
\begin{align*}
\text{(max-item } '(1 \ 2)) & \\
\rightarrow & \text{ (cond ((empty? (rest } '(1 \ 2))) (first } '(1 \ 2))) [\text{else ...}]) \\
\rightarrow & \text{ (cond ((empty? } '(2)) (first } '(1 \ 2))) [\text{else ...}]) \\
\rightarrow & \text{ (cond [false (first } '(1 \ 2))] [\text{else ...}]) \\
\rightarrow & \text{ (cond [else (cond [(> (first } '(1 \ 2)) ...) ...] [\text{else ...}])]) \\
\rightarrow & \text{ (cond [(> (first } '(1 \ 2)) (max-item (rest } '(1 \ 2)))) ...) [\text{else ...}]) \\
\rightarrow & \text{ (cond [(> 1 (max-item (rest } '(1 \ 2)))) ...) [\text{else ...}]) \\
\rightarrow & \text{ (cond [(> 1 (max-item } '(2)) ...) [\text{else ...}]) \\
\rightarrow & \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \\
\rightarrow & \text{ (cond [(> 1 \ 2) ...) [\text{else ...}]]) \\
\rightarrow & \text{ (cond [else (max-item (rest } '(1 \ 2))))]) \\
\rightarrow & \text{ (max-item (rest } '(1 \ 2))) \\
\rightarrow & \text{ (max-item } '(2)) \\
\rightarrow & \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \\
\rightarrow & 2
\end{align*}
\]

20 steps — where 10 came from two recursive calls
Steps for max-item and N Elements

In the worst case, the step count \( T \) for an \( n \)-element list passed to *max-item* is

\[
T(n) = 10 + 2T(n-1)
\]

\[
T(1) = 5
\]
\[
T(2) = 10 + 2T(1) = 20
\]
\[
T(3) = 10 + 2T(2) = 50
\]
\[
T(4) = 10 + 2T(3) = 110
\]
\[
T(5) = 10 + 2T(4) = 230
\]

...  

• In general, \( T(n) > 2^n \)

• Note that \( 2^{30} \) is \( 1,073,741,824 \) — which is why our last test never produced a result
Repairing max-item

In the case of **max-item**, the problem is easily fixed with **local**

```
(define (max-item nel)
  (cond
    [(empty? (rest nel)) (first nel)]
    [else
     (local [(define r (max-item (rest nel)))]
       (cond
        [>(first nel) r) (first nel)]
        [else r])))))
```

With this definition, there’s always one recursive call

```
(max-item '(1 2)) takes 17 steps
```
Steps for new max-item and N Elements

In the worst case, now, the step count $T$ for an $n$-element list passed to max-item is

$$T(n) = 12 + T(n-1)$$

$T(1) = 5$
$T(2) = 12 + T(1) = 17$
$T(3) = 12 + T(2) = 29$
$T(4) = 12 + T(3) = 41$
$T(5) = 12 + T(4) = 53$

...$

• In general, $T(n) = 5 + 12(n-1)$

• So our last test takes only 343 steps
Using Local to Reduce Complexity

Before, we used `local` to either make the code nicer or to support abstraction

Now we’re using `local` to avoid redundant calculations, which avoids evaluation complexity

Fortunately, these reasons reinforce each other

Where a value is definitely computed and possibly computed multiple times, always give it a name and compute it once
We once wrote a sort-list function:

; sort-list : list-of-num -> list-of-num
(define (sort-list l)
  (cond
   [(empty? l) empty]
   [(cons? l) (insert (first l) (sort-list (rest l)))]))

How long does it take to sort a list of \( n \) numbers?

We have only one recursive call to sort-list, so it doesn’t have the same problem as before...
Insertion Sort

... but what about \texttt{insert}?

\begin{verbatim}
; sort-list : list-of-num -> list-of-num
(define (sort-list l)
  (cond
   [(empty? l) empty]
   [(cons? l) (insert (first l) (sort-list (rest l)))]))

; insert : num list-of-num -> list-of-num
(define (insert n l)
  (cond
   [(empty? l) (list n)]
   [(cons? l)
    (cond
     [(< n (first l)) (cons n l)]
     [else (cons (first l) (insert n (rest l)))])))))
\end{verbatim}

On each iteration of \texttt{sort-list}, there’s a call to \texttt{sort-list}
and a call to \texttt{insert}
Insert Time

insert itself is like the repaired max-item:

; insert : num list-of-num -> list-of-num
(define (insert n l)
  (cond
    [(empty? l) (list n)]
    [(cons? l)
      (cond
        [(< n (first l)) (cons n l)]
        [else (cons (first l) (insert n (rest l)))]))])

In the worst case, insert into a list of size n takes $k_1 + k_2n$

The variables $k_1$ and $k_2$ stand for some constant
Insertion Sort Time

Given that the time for \textit{insert} is $k_1 + k_2n$...

\begin{verbatim}
; sort-list : list-of-num -> list-of-num
(define (sort-list l)
  (cond
    [(empty? l) empty]
    [(cons? l) (insert (first l) (sort-list (rest l))))])
\end{verbatim}

The time for \texttt{sort-list} is defined by

\[
\begin{align*}
T(0) &= k_3 \\
T(n) &= k_4 + T(n-1) + k_1 + k_2n
\end{align*}
\]
Insertion Sort Time

\[ T(0) = k_3 \]
\[ T(n) = k_4 + T(n-1) + k_1 + k_2n \]

Even if each \( k \) were only 1:

\[ T(0) = 1 \]
\[ T(1) = 4 \]
\[ T(2) = 8 \]
\[ T(3) = 13 \]
\[ T(3) = 19 \]

\[ \ldots \]

- In the long run, \( T(n) \) is a lot like \( n^2 \)

- This is a lot better than \( 2^n \) — but sorting a list of 10,000 items takes more than 100,000,000 steps
Sorting Algorithms

• The list-of-num template leads to the insertion sort algorithm
  ○ It’s not practical for large lists

• Algorithms such as quick sort and merge sort are faster
Merge Sort

\[
(\text{define } (\text{merge-sort } l) \\
(\text{cond} \\
[(\text{or } (\text{empty? } l) (\text{empty? } (\text{rest } l))) ] l] \\
[\text{else} \\
(\text{local } [(\text{define } a\text{-half } (\text{even-items } l)) \\
(\text{define } b\text{-half } (\text{odd-items } l))] \\
(\text{merge-lists} \\
(\text{merge-sort } a\text{-half}) \\
(\text{merge-sort } b\text{-half}))))))
\]

- **even-items** and **odd-items** each take \(k_5 + k_6n\) steps
- **merge-lists** takes \(k_7 + k_8n\) steps
- So, for **merge-sort**:

\[
\begin{align*}
T(0) & = k_9 \\
T(1) & = k_{10} \\
T(n) & = k_{11} + 2T(n/2) + 2k_5 + 2k_6n + k_7 + k_8n
\end{align*}
\]
Merge Sort Time

Simplify by collapsing constants:

\[ T(n) = k_{12} + 2T(n/2) + k_{13}n \]

Setting constants to 1:

\[ \ldots \]

\[ T(5) = 21 \]
\[ T(6) = 27 \]
\[ T(7) = 33 \]
\[ T(8) = 39 \]
\[ T(9) = 46 \]
\[ \ldots \]

In the long run, \( T(n) \) is a lot like \( n\log_2 n \)

- Sorting a list of 10,000 items takes something like 100,000 steps
  (which is 1,000 times faster than insertion sort)
The Cost of Computation

The study of execution time is called *algorithm analysis*, and the theoretical bound for a given problem is the subject of *complexity theory*

Practical points:

1. Use **local** to avoid redundant computations
   - Something you can always do to tame evaluation
2. Algorithms like **merge-sort** are in textbooks
   - You mostly learn them, not invent them

Other courses teach you more about the second category