

## Max of a List

- Implement the function `max-item` which returns the biggest number in a list of numbers

## Data and Contract

**Data:** `list-of-num`, obviously

**Contract:**

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

## Examples

```
(max-item '(2 7 5)) "should be" 7
```

```
(max-item empty) "should be" ...
```

**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

```
(max-item '(2 7 5)) "should be" 7
(max-item '(2)) "should be" 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)) ...]))
```

## Implementation Complete

```
(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))]]]))
```

## Test

```
(max-item '(2)) "should be" 2
```

works fine

```
(max-item '(1 2 3 4 5 6 7 8 9 10))
"should be" 10
```

works fine

```
(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))
"should be" 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 can we understand how long a program takes 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)) → (+ 1 42) → 43
```

So, evaluation takes 2 steps

## Steps for max-item and 1 Element

How long does this expression take?

```
(max-item '(2))
```

```
(max-item '(2))
→ (cond [(empty? (rest '(2))) (first '(2))] ...)
→ (cond [(empty? empty) (first '(2))] ...)
→ (cond [true (first '(2))] ...)
→ (first '(2))
→ 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?

```
(max-item '(2 1))
```

```
(max-item '(2 1))
→ (cond [(empty? (rest '(2 1))) (first '(2 1))] [else ...])
→ (cond [(empty? '(1)) (first '(2 1))] [else ...])
→ (cond [false (first '(2 1))] [else ...])
→ (cond [else (cond [(> (first '(2 1)) ...) ...] [else ...])])
→ (cond [(> (first '(2 1)) (max-item (rest '(2 1)))) ...] [else ...])
→ (cond [(> 2 (max-item (rest '(2 1)))) ...] [else ...])
→ (cond [(> 2 (max-item '(1))) ...] [else ...])
→ ... → ... → ... → ...
→ (cond [(> 2 1) (first '(2 1))] [else ...])
→ (first '(2 1))
→ 2
```

14 steps — where 5 came from the recursive call

Are all 2-element lists the same?

## Steps for max-item and 2 Elements

```
(max-item '(1 2))
```

```
(max-item '(1 2))
→ (cond [(empty? (rest '(1 2))) (first '(1 2))] [else ...])
→ (cond [(empty? '(2)) (first '(1 2))] [else ...])
→ (cond [false (first '(1 2))] [else ...])
→ (cond [else (cond [(> (first '(1 2)) ...) ...] [else ...])])
→ (cond [(> (first '(1 2)) (max-item (rest '(1 2)))) ...] [else ...])
→ (cond [(> 1 (max-item (rest '(1 2)))) ...] [else ...])
→ (cond [(> 1 (max-item '(2))) ...] [else ...])
→ ... → ... → ... → ...
→ (cond [(> 1 2) ...] [else ...])
→ (cond [else (max-item (rest '(1 2)))]])
→ (max-item (rest '(1 2)))
→ (max-item '(2))
→ ... → ... → ... → ...
→ 2
```

20 steps — where 10 came from *two* recursive calls

## 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 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

## 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

## Sorting

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 `insert`?

```
; 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)))]))]))
```

On each iteration of `sort-list`, there's a call to `sort-list` and a call to `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 `insert` is  $k_1 + k_2n$ ...

```
; 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)))]))
```

The time for `sort-list` is defined by

$$T(0) = k_3$$
$$T(n) = k_4 + T(n-1) + k_1 + k_2n$$

## 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:

$$T(0) = 1$$
$$T(1) = 4$$
$$T(2) = 8$$
$$T(2) = 13$$
$$T(3) = 19$$

...

- 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

```
(define (merge-sort l)
  (cond
    [(or (empty? l) (empty? (rest l))) l]
    [else
     (local [(define a-half (even-items l))
              (define b-half (odd-items l))]
       (merge-lists
        (merge-sort a-half)
        (merge-sort b-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`:

$$T(0) = k_9$$
$$T(1) = k_{10}$$
$$T(n) = k_{11} + 2T(n/2) + 2k_5 + 2k_6n + k_7 + k_8n$$

## Merge Sort Time

Simplify by collapsing constants:

$$T(n) = k_{12} + 2T(n/2) + k_{13}n$$

Setting constants to 1:

```
...
T(5) = 21
T(6) = 27
T(7) = 33
T(8) = 39
T(9) = 46
...
```

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)

## Vectors

The **Advanced** language provides **vectors**, which is similar to lists:

```
> (list 1 2 3)
(list 1 2 3)
> (vector 1 2 3)
(vector 1 2 3)
```

Differences:

- There's nothing like `cons` for vectors
- The `vector-ref` function extracts an element from anywhere in the vector in constant time

## The Cost of Computation

The study of execution time is called **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 learn them, not invent them

Other courses teach you more about the second category

Is there anything else in the first category (things you can do now)?

soon...

## List-Ref versus Vector-Ref

```
; list-ref : list-of-X nat -> X
(define (list-ref l n)
  (cond
    [(zero? n) (first l)]
    [else (list-ref (rest l) (sub1 n))]))

(list-ref '(a b c d) 1) "should be" 'b
```

In general, `(list-ref l n)` takes about  $n$  steps

## List-Ref versus Vector-Ref

```
; vector-ref : vector-of-X nat -> X
(define (vector-ref l n)
  ...)

(vector-ref (vector 'a 'b 'c 'd) 1)
"should be" 'b
```

In general, `(vector-ref v n)` takes 1 step

You can't actually define `vector-ref` yourself

Eventually, we'll use vectors when we need "random access" among arbitrarily many components

More generally, each kind of data comes with operations that have a certain cost — a programmer has to pick the right data