Patterns of recursion

Readings: none.

Topics:

- Simple recursion
- Using accumulative recursion
- Recognizing generative recursion

Simple vs. general recursion

All of the recursion we have done to date has followed a pattern we call "**simple** recursion".

The templates we have been using have been derived from a data definition and specify the form of the recursive application.

We will now learn to use a new pattern of recursion, **accumulative recursion**, and learn to recognize **generative recursion**.

For the next several lecture modules we will use simple recursion and accumulative recursion. We will avoid generative recursion until the end of the course.

Simple recursion

Recall from Module 06:

In simple recursion, every argument in a recursive function application (or applications, if there are more than one) are either:

- unchanged, or
- *one step* closer to a base case according to a data definition

The limits of simple recursion

```
;; (max-list lon) produces the maximum element of lon
;; max-list: (listof Num) → Num
;; requires: lon is nonempty
(define (max-list lon)
  (cond [(empty? (rest lon))  (first lon)]
        [(> (first lon) (max-list (rest lon))) (first lon)]
        [else  (max-list (rest lon))]))
```

There may be two recursive applications of max-list.

The code for max-list is correct.

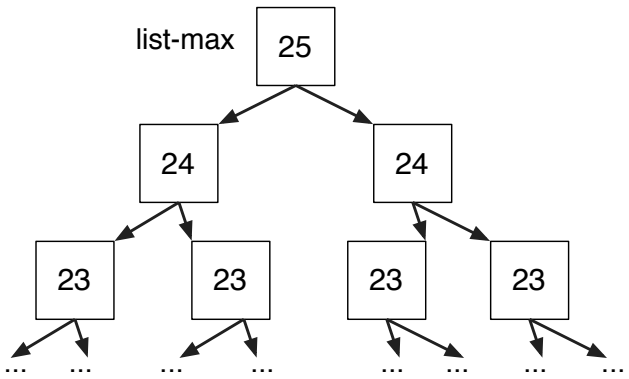
But computing (max-list (countup-to 1 25)) is very slow.

Why?

The initial application is on a list of length 25.

There are two recursive applications on the rest of this list, which is of length 24.

Each of those makes two recursive applications.



max-list can make up to $2^n - 1$ recursive applications on a list of length n.

Measuring efficiency

We can take the number of recursive applications as a rough measure of a function's efficiency. list-max can take up to $2^n - 1$ recursive applications.

length makes n recursive applications on a list of length n.

length is clearly more efficient than this version of list-max.

We say that length's efficency is proportional to n and max-list's efficiency is proportional to 2^n. We express the former as $O(n)$ and the later as $O(2^n)$.

There are "families" of algorithms with similar efficiencies.
Examples, from most efficient to least:

"Big-Oh"	**Example**
$O(1)$	no recursive calls
$O(\lg n)$	divide things in half; binary-search on a *balanced tree*
$O(n)$	one recursive appplication for each item; length
$O(n^2)$	an $O(n)$ application for each item; insertion-sort
$O(2^n)$	two recursive applications for each item; max-list

Much more about "Big-Oh" notation and efficiency in later courses.

Accumulative recursion

Intuitively, we can find the maximum of a list of numbers by scanning it, remembering the largest value seen so far.

Computationally, we can pass down that largest value seen so far as a parameter called an **accumulator**.

This parameter accumulates the result of prior computation, and is used to compute the final answer that is produced in the base case.

This approach results in the code on the next slide.

```
;; (max-list/acc lon max-so-far) produces the largest
;;     of the maximum element of lon and max-so-far
;; max-list/acc: (listof Num) Num → Num
(define (max-list/acc lon max-so-far)
  (cond [(empty? lon) max-so-far]
        [(> (first lon) max-so-far)
         (max-list/acc (rest lon) (first lon))]
        [else (max-list/acc (rest lon) max-so-far)]))
```



```
(define (max-list2 lon)
  (max-list/acc (rest lon) (first lon)))
```

Now even (max-list2 (countup-to 1 2000)) is fast.

```
(max-list2 (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
⇒ (max-list/acc (cons 2 (cons 3 (cons 4 empty))) 1)
⇒ (max-list/acc (cons 3 (cons 4 empty)) 2)
⇒ (max-list/acc (cons 4 empty) 3)
⇒ (max-list/acc empty 4)
⇒ 4
```


This technique is known as **accumulative recursion**.

It is more difficult to develop and reason about such code, which is why simple recursion is preferable if it is appropriate.

HtDP discusses it much later than we are doing (after material we cover in lecture module 10) but in more depth.

Indicators of the accumulative recursion pattern

- All arguments to recursive function applications are:
  - unchanged, or
  - one step closer to a base case in the data definition, or
  - a partial answer (passed in an *accumulator*).
- The value(s) in the accumulator(s) are used in one or more base cases.
- The accumulatively recursive function usually has a wrapper function that sets the initial value of the accumulator(s).

Another accumulative example: reversing a list

Using simple recursion:

```
;; my-reverse: (listof X) → (listof X)
(define (my-reverse lst)
  (cond [(empty? lst) empty]
        [else (append (my-reverse (rest lst)) (list (first lst)))]))
```

Intuitively, append does too much work in repeatedly moving over the produced list to add one element at the end.

This has the same worst-case behaviour as insertion sort, $O(n^2)$.

Reversing a list with an accumulator

```
(define (my-reverse lst)  ; primary function
  (my-rev/acc lst empty))

(define (my-rev/acc lst acc)  ; helper function
  (cond [(empty? lst) acc]
        [else (my-rev/acc (rest lst)
                          (cons (first lst) acc))]))
```

This is $O(n)$.

A condensed trace

(my-reverse (cons 1 (cons 2 (cons 3 empty))))
⇒ (my-rev/acc (cons 1 (cons 2 (cons 3 empty))) empty)
⇒ (my-rev/acc (cons 2 (cons 3 empty)) (cons 1 empty))
⇒ (my-rev/acc (cons 3 empty) (cons 2 (cons 1 empty)))
⇒ (my-rev/acc empty (cons 3 (cons 2 (cons 1 empty))))
⇒ (cons 3 (cons 2 (cons 1 empty)))

Generative Recursion: GCD

In Math 135, you learn that the Euclidean algorithm for Greatest Common Divisor (GCD) can be derived from the following identity for $m > 0$:

$$gcd(n, m) = gcd(m, n \bmod m)$$

We also have $gcd(n, 0) = n$.

We can turn this reasoning directly into a Racket function.


```
;; (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm
;; euclid-gcd: Nat Nat → Nat
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

This function does not use simple or accumulative recursion.

The arguments in the recursive application were **generated** by doing a computation on m and n.

The function euclid-gcd uses **generative recursion**.

Once again, functions using generative recursion are easier to get wrong, harder to debug, and harder to reason about.

We will return to generative recursion in a later lecture module. Avoid generative recursion until then.

Simple vs. accumulative vs. generative recursion

In **simple recursion**, all arguments to the recursive function application (or applications, if there are more than one) are either unchanged, or *one step* closer to a base case in the data definition.

In **accumulative recursion**, parameters are as above, plus parameters containing partial answers used in the base case.

In **generative recursion**, parameters are freely calculated at each step. (Watch out for correctness and termination!)

Goals of this module

You should be able to recognize uses of simple recursion, accumulative recursion, and generative recursion.

You should be able to write functions using simple and accumulative recursion.

You should know that some functions are much more efficient than others, that efficiency is expressed with “Big-Oh” notation, and that you’ll learn more about this in future courses.