# Module 4: Generative Recursion

Topics:

- Generative Recursion
- Sorting Algorithms
- Analysis of Sorting Algorithms
- Designing generative recursive code

Readings: HtDP 25, 26, Intermezzo 5



# Types of recursion so far

- Structural recursion
  - Based on recursive definition of input data values
  - Standard templates
- Accumulative recursion
  - Builds up intermediate results on recursive calls
  - Specialized template
- Some algorithms do not fall into either of these categories



# Example: `gcd`

- The greatest common divisor (*gcd*) of two natural numbers is the largest natural number that divides evenly into both.
  - *gcd (10, 25) = 5*
  - *gcd (20, 22) = 2*
  - *gcd (47, 21) = 1*
- Exercise: Write **`gcd`** function using the standard count down template.

## Euclid's Algorithm for **gcd**

- gcd(m,0) = m
- gcd(m,n) = gcd(n, m mod n)

```
(define (gcd m n)
  (cond
    [(zero? n) m]
    [else (gcd n (remainder m n))]))
```



## Tracing **gcd**

```
(gcd 25 10)
⇒(gcd 10 (remainder 25 10))
⇒(gcd 10 5)
⇒(gcd 5 (remainder 10 5))
⇒(gcd 5 0)
⇒ 5
```



## Comments on **gcd**

- Not structural (not counting up or down by 1)
- Not accumulative

⇒Generative recursion

- Still has a base case
- Still has a recursive case – but problem is broken down in a new way

# Why generative recursion?

- Allow more creativity in solutions
- Remove restrictions on solutions
- May allow for improved efficiency
- May be more intuitive for some problems
  - Breaking into more “natural” subproblems
- We need to “generate” (figure out) the subproblems



# Steps for Generative Recursion

1. Divide the problem into subproblem(s)
2. Determine base case(s)
3. Figure out how to combine subproblem solutions to solve original problem
4. Use local constants and helper functions to make division more understandable
5. TEST! TEST! TEST!



# Example: removing duplicates

```
;; singles: (listof X) -> (listof X)
;; Produces a list like lst, but
;; containing only the first
;; occurrences of each element in lst
;; Examples:
;; (singles empty) => empty
;; (singles (list 1 2 1 3 4 2))
;;    => (list 1 2 3 4)
```

```
(define (singles lst)
 (cond
  [(empty? lst) empty]
  [else (cons (first lst)
         (singles
           (filter
            (lambda (x)
             (not (equal? x
                   (first lst))))
            (rest lst))))]))
```



# Example: reversing a number

Write a function **backwards** that consumes a natural number and produces a new number with the digits in reverse order.

For example,

- `(backwards 6) => 6`
- `(backwards 89) => 98`
- `(backwards 10011) => 11001`
- `(backward 5800) => 85`



# A Possible Approach

Consider the number n = 5678

- Divide the number into:
  - Last digit: 8
  - Everything else: 567
- Next, reverse 567
  - Take last digit (7) and "add to" 8 => 87
  - What's left? 56
- Repeat the process until all digits processed.

## Start with the accumulative template

```
(define (backwards n)
  (local
    [;; bw-acc: nat nat -> nat
     ;; produces the number resulting from
     ;; adding the reversed digits of
     ;; res to the end of so-far
     (define (bw-acc so-far res)
       (cond
         [(zero? res) so-far]
         [else (bw-acc … …)]))]
    …))
```



```
(define (backwards n)
  (local
    [;; bw-acc: nat nat -> nat
     ;; produces the number resulting from
     ;; adding the reversed digits of
     ;; res to the end of so-far
     (define (bw-acc so-far res)
       (cond
         [(zero? res) so-far]
         [else (bw-acc
                  (+ (* so-far 10)
                     (remainder res 10))
                  (quotient res 10))]))]
    (bw-acc 0 n)))
```



## The Sorting Problem

```
;; sort-list:
;; (listof num) -> (listof num)
;; Produces a list with all of the same
;; elements as lst, but in sorted order
;; from smallest to largest
;; Assumption: No duplicate values.
;; Example:
;; (sort-list (list 1 4 3 2))
;; => (list 1 2 3 4)
```

## Insertion Sort (from CS115)

- An empty list is already sorted
- If the rest of the list was already sorted
  - Just find the correct spot to insert the first value in the list



## Insertion Sort from CS115

```
(define (insert-sort lst)
  (cond
    [(empty? lst) empty]
    [else (insert (first lst)
                  (insert-sort
                     (rest lst)))]))
```



## The **insert** helper function

```
;; insert: num (listof num) -> (listof num)
;; produces a new strictly increasing list
;; containing item and all elements in
;; strictly increasing sortedlist
(define (insert item sortedlist)
  (cond
    [(empty? sortedlist) (list item)]
    [(< item (first sortedlist))
         (cons item sortedlist)]
    [else (cons (first sortedlist)
         (insert item (rest sortedlist)))]))
```

# Running time of `insert-sort`

- Sorting `(list` $x_1$ $x_2$...$x_n$`)`:
  - n calls to **`insert`**
  - n calls to **`insert-sort`**
  - => Overall running time depends on `insert`
- Insert $x_j$ into sorted `(list` $x_{j+1}$ $x_{j+2}$...$x_n$`)`:
  - Best case: $x_j$ into first position
  - Worst case: $x_j$ into last position
  - ⇒**`insert`** has linear running time

⇒**`insert-sort`** has quadratic running time



# Selection sort: another sorting algorithm

Consider this approach to sorting:
- Find the smallest value
- Put it at beginning of list
- Sort what's left by repeating this process



```
(define (selection-sort lst)
  (cond
    [(empty? lst) empty]
    [else
    (local
     [(define smallest
        (foldr min (first lst) (rest lst)))
      (define sm-removed
        (filter (lambda (x)
               (not (= x smallest))) lst))]
     (cons smallest
         (selection-sort sm-removed)))]))
```

# Running Time of `selection-sort` (assume list has length `n`)

| List Size | | `smallest` | `sm-removed` | `cons` | Total steps |
|---|---|---|---|---|---|
| n | | n | n | 1 | 2n+1 |
| n-1 | | n-1 | n-1 | 1 | 2n-1 |
| n-2 | | n-2 | n-2 | 1 | 2n-3 |
| … | | | | | |
| 1 | | 1 | 1 | 1 | 3 |



# Generative Recursion: more room for errors

There are more places where things can go wrong when using generative recursion:

- Base case condition or answer
- Recursive case:
  - Breaking into different subproblems (not just **`first`** and **`rest`**)
  - Recursive calls
  - Combining subproblem solutions into overall solution

=> Continue with testing and documentation



# Quicksort – another sorting algorithm

Consider the following approach

- Choose a “pivot” value
- Find all values less than the pivot: sort them
- Find all values greater than the pivot: sort them
- Put the results together:
  - Less then sorted
  - Pivot
  - Greater than sorted

=> Done!

## Quicksort questions

- How to choose pivot?
  - Choose any value in the list
- How to sort smaller lists?
  - Use same idea again (quicksort recursively)
- When to stop recursion?
  - When the list is empty
- How to combine the parts?
  - append



```
(define (quick-sort lst)
  (cond
    [(empty? lst) empty]
    [else
     (local [(define pivot (first lst))
             (define less-than-p
               (filter (lambda (x)
                         (< x pivot)) lst))
             (define more-than-p
               (filter (lambda (x)
                         (> x pivot))lst))]
      (append (quick-sort less-than-p)
              (list pivot)
              (quick-sort more-than-p)))]))
```



## Running time of `quick-sort`: Worst case

- What pivot value gives the worst partitioning?
  - Largest or smallest
- How many recursive calls in worst case?
  - About 2N
  - Half of those on empty lists
- What is the running time of the partitioning?
  - Linear
- Overall Running Time:
  - Quadratic

## More on Quicksort

- In practice,
  - partitioning usually produces sublists which are close in size to each other
  - Quicksort usually uses fewer levels of recursion than its extreme cases
  - ⇒Quicksort is usually much faster than insertion and selection sort



## Built-in **quicksort**

```
;; quicksort: (listof X)
;;     (X X -> boolean) -> (listof X)
;; produces a list with elements from
;; lst using the order given by compare
;; Example: (quicksort (list 3 0 7 -1) <)
;;            => (list -1 0 3 7)
(define (quicksort lst compare) …)
```

Exercise: Use **quicksort** to sort a list of **posn** values into decreasing order of x coordinates.



## Comments on Generative Recursion

- More choices increases chances for errors
- Design recipe:
  - No general template for generative recursion
  - Contract, purpose, examples, testing are still important
- Structural and accumulative recursion remain best choice for many problems
  - Templates are still important!
- An algorithm can use combinations of different types of recursion

# Goals of Module 4

- Understand how generative recursion is more general than structural or accumulative recursion
- Understand how insertion sort, selection sort and quicksort work
- Be able to compare running times of sorting algorithms
- Be aware that generatively recursive solutions may be harder to debug