Module 7: Advanced Recursion

Working with portions of a list

Consider the following functions.

```
;; (sum L) add up all the values in L
;; sum: (listof Num) -> Num
;; Example:
(check-expect (sum (list 2 3 6)) 11)

(define (sum L) (foldr + 0 L))


;; (portions L) divide each value in L by sum of L.
;; portions: (listof Num) -> (listof Num)
;; Examples:
(check-expect (portions (list 1 1 2)) (list 0.25 0.25 0.5))
(check-expect (portions (list 6 1 3)) (list 0.6 0.1 0.3))

(define (portions L)
  (map (lambda (x) (/ x (sum L)))
       L))
```

Working with portions of a list

It is pretty easy to rewrite `sum` using only recursion:

```
(define (sum L)
  (cond [(empty? L) 0]
        [else (+ (first L) (sum (rest L)))]))
```

Rewriting `portions` is more difficult.

Exercise

Test this code and see how it behaves:

```
;; (portions-bad L) Try to divide each value in L by sum of L.
;; portions-bad: (listof Num) -> (listof Num)
(define (portions-bad L)
  (cond [(empty? L) '()]
        [else (cons (/ (first L) (sum L))
                    (portions-bad (rest L)))]))
```

Each time we call `(sum L)` it is working with a smaller list. The very first item gets divided by the total, but the remaining items are divided by smaller and smaller numbers!

A non-recursive "wrapper" around a recursive function

To solve this problem, make a non-recursive function that calls a separate recursive function.

Exercise

Write a recursive function `divide-each` that allows `portions-r` to achieve its purpose.

```
;; (portions-r L) divide each value in L by sum of L.
;; portions-r: (listof Num) -> (listof Num)
;; Examples:
(check-expect (portions-r (list 1 1 2)) (list 0.25 0.25 0.5))
(check-expect (portions-r (list 6 1 3)) (list 0.6 0.1 0.3))

(define (portions-r L)
  (divide-each L (sum L)))
```

Working with wrappers

If a function cannot accomplish everything required in a single recursion, we may need to

- modify the data before doing our recursion,
- modify the answer that a recursive function returns, or
- create additional data to be used by the recursive function.

We can do this with a "wrapper" function that calls the recursive function just once.

Exercise

Write a function `(add-first-each L)` that consumes a `(listof Int)` and adds to each value in the list the first in the list.
Your function should be a wrapper around a recursive helper function.

```
(add-first-each (list 3 2 7 6 5)) => (list 6 5 10 9 8)
```

Adding a counter using a wrapper

Consider the function `add-index`:

```
;; (add-index L) to each item in L, add the distance from the front of L.
;; add-index: (listof Num) -> (listof Num)
;; Examples:
;;(add-index (list 0   0   0))
;;                  +0  +1  +2
;; =>         (list 0   1   2)

;; (add-index (list 2   3   5   7  11))
;;                  +0  +1  +2  +3  +4
;; =>          (list 2   4   7  10  15)
```

Simply recursing through the list we have no way to determine how many steps we have already taken. Solution: make `add-index` be a wrapper around a recursive function.

Adding a counter using a wrapper

We can make a new function, with an extra parameter, to count how far we have already proceeded from the front of the list:

```
;; (add-counter-from L counter) add counter to first of L, and
;;   numbers counting up to subsequent items.
;; add-counter-from: (listof Num) Nat -> (listof Num)
;; Example:
(check-expect (add-counter-from (list 2   7   4) 3) (list 5 11 9))
```

`add-index` then need only start the parameter at zero:

```
(define (add-index L)
  (add-counter-from L 0))
```

`add-counter-from` will do the work from there.

Ex. Complete `add-counter-from`.

Counting Exercises

Exercise

Write a function `(non-decreasing L)` that consumes a non-empty `(listof Num)`, and returns a `(listof Num)` containing only those values at least as big as all the values that came before.

For example,

```
(non-decreasing (list 2 3 1 6 8 6 2 4 8 1 9))
=> (list 2 3 6 8 8 9)
```

Exercise

Write a function `(at-index L)` that consumes a `(listof Int)` and returns all the values in `L` so item *i* is at location *i*.

For example,

```
(at-index (list 0 6 2 3 5 6 0 7)) => (list 0 2 3 7)
; . . . . . 0 1 2 3 4 5 6 7
```

Sorted Data

A `(listof Int)` is said to be sorted in increasing order if every item in the list is greater than or equal to the value that comes before it.
For example, `(list 2 3 3 5 7)` is sorted in increasing order, but `(list 2 3 5 3 7)` is not.

Exercise

Complete `sorted?`.

```
;; (sorted? L) return #true if every value in L is >= the one before.
;; sorted? (listof Int) -> Bool
;; Examples:
(check-expect (sorted? (list 42)) #true)
(check-expect (sorted? (list 2 3 3 5 7)) #true)
(check-expect (sorted? (list 2 3 5 3 7)) #false)
```

What is the base case?

Sorted Data

Suppose we have a sorted `(listof Int)`, and we wish to add a new value, keeping it sorted.

- What should we do if the list is empty?
- What should we do if the item is less than or equal to the first item?
- What should we do if the item is greater than the first item?

Exercise

Complete `insert`.

```
;; (insert item L) Add item to L so L remains sorted in increasing order.
;; insert: Int (listof Int) -> (listof Int)
;; Requires: L is sorted in increasing order.
;; Examples:
(check-expect (insert 6 (list 7 42)) (list 6 7 42))
(check-expect (insert 81 (list 3 9 27)) (list 3 9 27 81))
(check-expect (insert 5 (list 2 3 7)) (list 2 3 5 7))
```

Using `insert`, sort a list that is not sorted

Note that `insert` requires `L` to be sorted, but there are no restrictions on its length. It could be an empty list.

We can use this to sort a list that is not already sorted.

Suppose we have an unsorted list: `(list 2 9 7 4 6)`.

Start with an empty list, and construct the answer there. Insert one value into the (empty) answer list. Then insert the next value into the result from this, and continue this process for each value in the list.

How? **`foldr`**!

Tracing Insertion Sort

```
;; (insertion-sort L) return a copy of L, sorted in increasing order.
;; insertion-sort: (listof Int) -> (listof Int)
;; Examples:
(check-expect (insertion-sort (list 3 9 7 4)) (list 3 4 7 9))

(define (insertion-sort L)
  (foldr insert '() L))

(insertion-sort (list 3 9 7 4))
=> (foldr insert '() (list 3 9 7 4))
=> (insert 3 (insert 9 (insert 7 (insert 4 '()))))
=> (insert 3 (insert 9 (insert 7 (list 4))))
=> (insert 3 (insert 9 (list 4 7)))
=> (insert 3 (list 4 7 9))
=> (list 3 4 7 9)
```

Ex. Try this out with your `insert` function.

Recursion can do everything – but it may be harder

Anything that is possible with any combination of higher order functions (**`map`**, **`filter`**, and **`foldr`**) can be achieved using only recursion. Some more things are also possible!
The recursive code may be harder to write or to read, but not always.

Exercise

Rewrite `insertion-sort` to use recursion instead of **`foldr`**.
(You will still use `insert`.)

```
;; (insertion-sort L) return a copy of L, sorted in increasing order.
(define (insertion-sort L)
  (foldr insert '() L))
```

It would be difficult or impossible to write `insert` using only higher order functions. Yet it is not too difficult to write using recursion.

Always start by considering: can I do this using higher order functions? If you can, it will usually be easier.

Simulating Higher Order Functions using Recursion: map

The following program walks through an entire list, without doing anything to it:

```
(define (do-nothing L)
  (cond [(empty? L) '()]
        [else (cons (first L)
                    (do-nothing (rest L)))]))
```

Previously, we used **map** to transform each item in a list using a given function. Similarly, using recursion:

```
;; (double-each L) multiply each value in L by 2.
;; double-each: (listof Int) -> (listof Int)
(define (double-each L)
  (cond [(empty? L) '()]
        [else (cons (* 2 (first L))
                    (double-each (rest L)))]))
```

Exercise

Use recursion to write a function that duplicates the following function:

```
(define (f L) (map (lambda (x) (+ (sqr x) x)) L))
```

Simulating Higher Order Functions using Recursion: `filter`

The following program walks through an entire list, without doing anything to it:

```
(define (do-nothing L)
  (cond [(empty? L) '()]
        [else (cons (first L)
                    (do-nothing (rest L)))]))
```

This uses **cons** to include every value from the input. If we remove the `(cons (first L) ...)` it will recurse on the rest of the values, without keeping any.

Using **filter** we could keep some values and discard others. Similarly, using recursion:

```
;; (keep-evens L) return all values of L that are even.
;; keep-evens: (listof Int) -> (listof Int)
(define (keep-evens L)
  (cond [(empty? L) '()]
        [(even? (first L)) (cons (first L) (keep-evens (rest L)))]
        [else (keep-evens (rest L))]))
```

Ex. Write a recursive function that duplicates the following function:

```
(define (g L) (filter (lambda (x) (= 0 (remainder x 3))) L))
```

Simulating Higher Order Functions using Recursion: `foldr`

Recall how **`foldr`** works. It has three parameters: a combining function, a base value, and a list.

```
;; (sum L) return the sum of the values in L
;; sum: (listof Int) -> Int
(define (sum L) (foldr + 0 L))

(foldr + 0 (list 3 5 7))
=> (+ 3 (+ 5 (+ 7 0)))
```

We can use recursion to combine the **`first`** value with the result of a recursive call on **`rest`**.

```
(define (rsum L)
  (cond [(empty? L) 0]
        [else (+ (first L) (rsum (rest L)))]))
```

- The empty list is a base case, so it returns the base value; in this case, `0`.
- Otherwise, it combines `(first L)` with a recursive call on `(rest L)`, using the combining function; in this case, `+`.

```
(rsum (list 3 5 7)) => (+ 3 (rsum (list 5 7))) => (+ 3 (+ 5 (rsum (list 7))))
=> (+ 3 (+ 5 (+ 7 (rsum '())))) => (+ 3 (+ 5 (+ 7 0)))
```

Processing two lists simultaneously

Sometimes we have data in two or more lists, and need to do computation on the lists together. We identify three important cases, whcih we will discuss in further detail:

A list "going along for the ride" E.g. appending two lists:

```
(my-append (list 1 2 3) (list 4 5 6)) => (list 1 2 3 4 5 6)
```

Processing "in lockstep" E.g. adding items in one list to corresponding items in another:

```
(add-pairs (list 1 2 3) (list 5 8 6)) => (list 6 10 9)
```

Processing at different rates E.g. merging two sorted lists:

```
(merge (list 2 3 7) (list 4 6 8 9)) => (list 2 3 4 6 7 8 9)
```

Adding to a list

Inserting an item at the front of a list is easy: `(cons 7 (list 5 3 2)) => (list 7 5 3 2)`

Appending an item at the back can be done with a little recursion:

```
;; (add-end n L) add n at the end of L.
;; add-end: (listof Any) Any -> (listof Any)
;; Example:
(check-expect (add-end (list 2 3 5) 7) (list 2 3 5 7))

(define (add-end L n) (cond [(empty? L) (cons n '())]
                            [else (cons (first L) (add-end (rest L) n))]))
```

Ex. Experiment with `add-end` using various values. Try to understand how it works.

Adding to a list

Consider what `(add-end (list 2 3 5) 7)` does.

- We are given `(cons 2 (cons 3 (cons 5 '())))`
- We return `(cons 2 (cons 3 (cons 5 (cons 7 '()))))`

So we "replaced" `'()` with `(cons 7 '())`. Could we replace it with something else?
How much harder would it be to substitute a longer list?

Exercise

Use recursion to complete `append-lists`.

```
;; (append-lists L1 L2) form a list of the items in L1 then L2, in order.
;; append-lists: (listof Any) (listof Any) -> (listof Any)
;; Example:
(check-expect (append-lists (list 3 7 4) (list 6 8)) (list 3 7 4 6 8))
```

Template for a list "going along for the ride"

We do not need to recurse through `L2` in order to append it to `L1`. `L2` is present in the recursion, and is passed to the next recursive call.

We use **`first`** and **`rest`** on `L1`, just like in single-list recursion.

The template looks like this:

```
(define (my-alongforride-template L1 L2)
  (cond
    [(empty? L1) ... ]
    [else (... (first L1) ...
               ... (my-alongforride-template (rest L1) L2) ...)]))
```

Another list "going along for the ride"

We can instead recurse on a number, with an unchanged list:

```
;; (duplicate-thing L n) return a list with n copies of L.
;; duplicate-thing: (listof Any) Nat -> (listof (listof Any))
;; Example:
(check-expect (duplicate-thing (list 42 6 7) 3)
              (list (list 42 6 7) (list 42 6 7) (list 42 6 7)))
```

Ex. Complete `duplicate-thing`.

Dot-product

We may process two lists of the same length, at the same time.

The dot product of two vectors is the sum of the products of the corresponding elements of the vectors. (This works for vectors of any dimension.)
E.g. if $\vec{u} = [2, 3, 5]$ and $\vec{v} = [7, 11, 13]$, then $\vec{u} \cdot \vec{v} = 2 \cdot 7 + 3 \cdot 11 + 5 \cdot 13 = 112$.

Exercise

Complete `dot-product`.

```
;; A Vector is a (listof Num).

;; (dot-produce u v) return the dot product of u and v.
;; dot-product: Vector Vector -> Num
;; Requires: u and v have the same length.
;; Example:
(check-expect (dot-product (list 2 3 5) (list 7 11 13)) 112)
```

Dot-product

Here is one solution:

```
;; A Vector is a (listof Num).

;; (dot-produce u v) return the dot product of u and v.
;; dot-product: Vector Vector -> Num
;; Requires: u and v have the same length.
;; Example:
(check-expect (dot-product (list 2 3 5) (list 7 11 13)) 112)

(define (dot-product u v)
  (cond [(empty? u) 0]
        [else (+ (* (first u) (first v))
                 (dot-product (rest u) (rest v)))]
        ))
```

Now we'll generalize this idea of examining two lists at the same rate.

Lockstep template

Dot product consumes the two lists at the same rate, and they are of the same length.
When one becomes empty, the other does too.
Code will often resemble this template:

```
(define (lockstep-template L1 L2)
  (cond
   [(empty? L1) ... ] ; if L1 is empty, so is L2.
   [else (... (first L1) ... (first L2) ... ; We use both firsts.
              ... (lockstep-template (rest L1) (rest L2)) ... )]))
              ; We make a recursive call on both rests.
```

Exercise

Write a recursive function `vector-add` that adds two vectors.

```
(vector-add (list 3 5) (list 7 11)) => (list 10 16)
(vector-add (list 3 5 1 3) (list 2 2 9 3)) => (list 5 7 10 6)
```

Merging two sorted lists

So we've left one list unchanged, and consumed two lists at the same rate.
It remains to consider functions that consume two lists, but not at the same rate.

Suppose I have two lists, each sorted, and I wish to create a sorted list that contains the items from both lists.

```
(merge (list 2 3 7) (list 4 6 8))  => (list 2 3 4 6 7 8)
```

OK, but how to do that???

Idea: look at the **first** item in both lists. Take the smaller one; then run recursively on the **rest** of the list that provided the smaller value, and the whole of the other list.

Idea: look at the `first` item in both lists. Take the smaller one; then run recursively on the `rest` of the list that provided the smaller value, and the whole of the other list.

Let's take a closer look at that example `(merge (list 2 3 7) (list 4 6 8))`.

- Given `(list 2 3 7)` and `(list 4 6 8)`. The `first` items are 2 and 4. The smaller is 2. Answer is 2, followed by the answer from a recursive call on `(list 3 7)` and `(list 4 6 8)`.
- Given `(list 3 7)` and `(list 4 6 8)`. The `first` items are 3 and 4. The smaller is 3. Answer is 3, followed by the answer from a recursive call on `(list 7)` and `(list 4 6 8)`.
- Given `(list 7)` and `(list 4 6 8)`. The `first` items are 7 and 4. The smaller is 4. Answer is 4, followed by the answer from a recursive call on `(list 7)` and `(list 6 8)`.
- Given `(list 7)` and `(list 6 8)`. The `first` items are 7 and 6. The smaller is 6. Answer is 6, followed by the answer from a recursive call on `(list 7)` and `(list 8)`.
- Given `(list 7)` and `(list 8)`. The `first` items are 7 and 8. The smaller is 7. Answer is 7, followed by the answer from a recursive call on `'()` and `(list 8)`.
- Given `'()` and `(list 8)`. One of the lists is empty, so return the non-empty list. Answer is (list 8).

Total answer is `(cons 2 (cons 3 (cons 4 (cons 6 (cons 7 (cons 8 '()))))))`.

Merging two sorted lists

Exercise

Carefully reason through what to do with `(list 5 7 15 17)` and `(list 1 2 6 12)`.

Merging two sorted lists: code

Ex. There are two base cases; what are they?

Exercise

Complete `merge`.

```
;; (merge L1 L2) return the list of all items in L1 and L2, in order.
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: L1 is sorted; L2 is sorted.
;; Example:
(check-expect (merge (list 2 3 7) (list 4 6 8 9)) (list 2 3 4 6 7 8 9))
```

Generic two-list template

More generally, we may need to consider if (1) both lists are empty; (2) just the first is empty; (3) just the second is empty; or (4) both are non-empty.

```
(define (my-two-list-template L1 L2)
  (cond
    [(and (empty? L1)
          (empty? L2))
     ... ]
    [(and (empty? L1)
          (not (empty? L2)))
     ( ... (first L2) ... (rest L2) ...)]
    [(and (not (empty? L1))
          (empty? L2))
     ( ... (first L1) ... (rest L1) ...)]
    [(and (not (empty? L1))
          (not (empty? L2)))
     ( ... my-two-list-template ...) ]))
```

If `L` is a list, `(cons? L)` gives the same answer as `(not (empty? L))`. You may use either.

Full Names

Suppose we have two `(listof Str)`: one of first names, and one of matching last names:

```
(define gnames (list "Joseph" "Burt" "Douglas" "James" "David"))
(define snames (list "Hagey" "Matthews" "Wright" "Downey" "Johnston"))
```

Exercise

Complete `join-names`.

```
;; (join-names G S) Make a list of full names from G and S.
;; join-names: (listof Str) (listof Str) -> (listof Str)
;; Example:
(check-expect (join-names gnames snames)
              (list "Joseph Hagey" "Burt Matthews" "Douglas Wright"
                    "James Downey" "David Johnston"))
```

Hint

Each name is formed from one value from each list; use the lockstep template!

List equality

How can we tell if two lists are the same?
The built-in function **`equal?`** will do it, but let's write our own.
Things to consider:

- Base case: if one list is empty, and the other isn't, they're not equal.
- If the first items aren't equal, the lists aren't equal.
- The empty list is equal to itself.

Exercise

Complete `list=?`

```
;; (list=? a b) return #true iff a and b are equal.
;; list=?: (listof Any) (listof Any) -> Bool
;; Examples:
(check-expect (list=? (list 6 7 42) (list 6 7 42)) #true)
```

Ex.

For added enjoyment (!), rewrite `list=?` without using **`cond`**.

Using lists to speed up computations

Suppose I have a series of numbers that I use frequently, but which take work to compute, such as the Catalan numbers (used in combinatorics; https://oeis.org/A000108):

$$C_n = \frac{\binom{2n}{n}}{n+1} \qquad C = [1, 1, 2, 5, 14, 42, \ldots]$$

You may assume you have a function to compute a Catalan number:

```
;; (catalan n) return the n-th Catalan number.
;; catalan: Nat -> Nat
```

If every time my program needs one of these, it computes it, it may compute the same number many times. This takes time. Instead, I can calculate each just once, and save them in a list.

```
;; (catalans-interval bottom top) return all the catalan numbers
;;   starting at index bottom, and ending before index top.
;; catalans-interval: Nat Nat -> (listof Nat)

(define (catalans-interval bottom top)
  (cond [(= bottom top) '()]
        [else (cons (catalan bottom) (catalans-interval (+ bottom 1) top))]))
```

(You could get the same result by `(map catalan (range bottom top 1))`.)

We can make a list of numbers, but can we get them back out?

Exercise

Complete `n-th-item`.

```
;; (n-th-item L n) return the n-th item in L, where (first L) is the 0th.
;; n-th-item: (listof Any) Nat -> Any
;; Example:
(check-expect (n-th-item (list 3 7 31 2047 8191) 0) 3)
(check-expect (n-th-item (list 3 7 31 2047 8191) 3) 2047)
```

By creating a list to store a sequence of numbers, then extracting the *n*th item of the list, we can speed computations, sometimes significantly.

There is a built-in function `list-ref` that behaves exactly like `n-th-item`. In real code, it is almost always better to use the built-in function. Avoid writing your own!

```
(list-ref (list 3 7 31 2047 8191) 0) => 3
(list-ref (list 3 7 31 2047 8191) 3) => 2047
```

A few reminders about `first` and `rest`

Consider a few `(listof Nat)`:

- `(first (list 1 2 3))` => `1`, which is a `Nat`.
- `(rest (list 1 2 3))` => `(list 2 3)`, which is a `(listof Nat)`.

- `(first (list 2 3))` => `2`, which is a `Nat`.
- `(rest (list 2 3))` => `(list 3)`, which is a `(listof Nat)`.

- `(first (list 3))` => `3`, which is a `Nat`.
- `(rest (list 3))` => `'()`, (the same as `empty`), which is a `(listof Nat)`.

If `L` is a non-empty `(listof X)`, for any type `X`:

- `(first L)` returns a `X`
- `(rest L)` returns a `(listof X)`.

! Never use `first` or `rest` on empty lists. Each requires a non-empty list.

Two-dimensional data

You may know how to compute binomial coefficients, used in combinatorics:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

If I have n items, this tells me how many ways there are to choose k of them. Requires: $k \leq n$. Suppose we want to save these, instead of recomputing as needed. How can we store the data? A list of lists!

```
(binomials 4) => (list
                  (list 1)            ; 0 choose 0
                  (list 1 1)          ; 1 choose 0, 1 choose 1
                  (list 1 2 1)        ; 2 choose 0, 2 choose 1, 2 choose 2
                  (list 1 3 3 1)      ; ...
                  (list 1 4 6 4 1))   ; ...
```

I can get one row out of this: `(n-th-item 4 binomials) => (list 1 4 6 4 1)`

...and an item out of that row: `(n-th-item 2 (n-th-item 4 binomials)) => 6`

Computing binomials

For reference, you may use the following functions to compute $\binom{n}{k}$:

```
;; (factorial n) return n!.
;; factorial: Nat -> Nat
;; Example:
(check-expect (factorial 4) 24)

(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))

;; (binomial n k) return n choose k.
;; binomial: Nat Nat -> Nat
;; Example:
(check-expect (binomial 4 1) 4)
(check-expect (binomial 4 2) 6)

(define (binomial n k)
  (/ (factorial n) (* (factorial k) (factorial (- n k)))))
```

Creating two-dimensional data

How can I build a table like this?

```
(binomials 4) => (list
                  (list 1)           ; 0 choose 0
                  (list 1 1)         ; 1 choose 0, 1 choose 1
                  (list 1 2 1)       ; 2 choose 0, 2 choose 1, 2 choose 2
                  (list 1 3 3 1)     ; ...
                  (list 1 4 6 4 1))  ; ...
```

We did this kind of thing earlier using **map**.

Since binomial has two parameters, use **lambda** to fill in the extra.

To build one row:

```
;; (make-binomial-row r) return the r-th row of the binomial table.
;; make-binomial-table: Nat -> (listof Nat)
;; Example:
(check-expect (make-binomial-row 4) (list 1 4 6 4 1))

(define (make-binomial-row r)
  (map (lambda (k) (binomial r k)) (range 0 (+ r 1) 1)))
```

Creating two-dimensional data

Now that we have a way to build one row, use **map** a second time to build all the rows:

```
;; (binomials n) return the binomial table up to n choose n.
;; binomials: Nat -> (listof (listof Nat))
;; Example:
(check-expect (binomials 2) (list
                  (list 1)           ; 0 choose 0
                  (list 1 1)         ; 1 choose 0, 1 choose 1
                  (list 1 2 1)))     ; 2 choose 0, 2 choose 1, 2 choose 2

(define (binomials n)
  (map make-binomial-row (range 0 (+ n 1) 1)))
```

Creating two-dimensional data

How can I use recursion to build a table like this?

```
(binomials 4) => (list
                  (list 1)           ; 0 choose 0
                  (list 1 1)         ; 1 choose 0, 1 choose 1
                  (list 1 2 1)       ; 2 choose 0, 2 choose 1, 2 choose 2
                  (list 1 3 3 1)     ; ...
                  (list 1 4 6 4 1))  ; ...
```

As before, start by building a function to create just one row of the table.

```
;; (make-binomial-row r i) make the rest of the r-th row of the
;;   binomial table, starting from i.
;; make-binomial-row: Nat Nat -> (listof Nat)
;; Example:
(check-expect (make-binomial-row-from 4 0) (list 1 4 6 4 1))

(define (make-binomial-row-from r i)
  (cond [(> i r) '()]
        [else (cons (binomial r i) (make-binomial-row-from r (+ 1 i)))]))
```

Creating two-dimensional data

Since `make-binomial-row-from` makes one row of the table, now I just need to call it repeatedly, once for each row. I can do this with another count up recursion.

```
;; (binomial-rows low high) make all the rows of binomials from low to high.
;; binomial-rows: Nat Nat -> (listof (listof Nat))

(define (binomial-rows low high)
  (cond [(= low high) '()]
        [else (cons (make-binomial-row-from low 0)
                    (binomial-rows (+ 1 low) high))]))
```

Recap: we break the task into two parts:

- Write a function that creates a single row.
- Write code that creates many rows, by repeatedly calling the single-row function.

Exercise

Using recursion, create a function (and necessary helper functions) to create the times tables up to a given value. For example,

```
(times-tables 4) => (list (list 0 0 0 0)
                          (list 0 1 2 3)
                          (list 0 2 4 6)
                          (list 0 3 6 9))
```

Remember: write the function for one row first!

Module Summary

- Become comfortable writing code that uses recursion in more complex ways.
- Understand how recursion can replace any use of higher order functions, and do things that are impossible with only higher order functions.
- Be able to design recursive functions that recurse on two values.
- Use recursion to build lists to store data, and to extract it again.