If you have not already, make sure you

- Read *How to Design Programs* Section 17.
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))
It is pretty easy to rewrite \texttt{sum} using only recursion:

\begin{verbatim}
(define (sum L)
 (cond [(empty? L) 0]
 [else (+ (first L) (sum (rest L)))]))
\end{verbatim}
Working with portions of a list

Rewriting portions is more difficult.

Copy this code and see how it behaves:

```scheme
;;;; (portions L) divide each value in L by sum of L.
;;;; portions: (listof Num) -> (listof Num)
(define (portions L)
  (cond [(empty? L) '()] [else (cons (/ (first L) (sum L))
    (portions (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 portions be a non-recursive function that calls a different function that is recursive:

Exercise

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

;;; (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)
  (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.

Exercise

Using recursion, write a function \( \text{(add-first-each L)} \) that consumes a \( \text{(listof Int)} \) and adds to each value in the list the first in the list.

\( \text{(add-first-each (list 3 2 7 6 5)) \Rightarrow (list 6 5 10 9 8)} \)
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:
(check-expect (add-index (list 0 0 0)) (list 0 1 2))
(check-expect (add-index (list 2 3 5 7 11)) (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.
We can make a new function, with an extra parameter, to count how far we have already proceeded from the front of the list:

```scheme
;; (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:

```scheme
(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 \((\text{non-decreasing } L)\) that consumes a \((\text{listof Num})\), and returns a \((\text{listof Num})\) containing only those values at least as big as all the values that came before.
For example,
\[(\text{non-decreasing } (\text{list } 2 3 1 6 8 6 4 8 1 9))\]
\[\Rightarrow (\text{list } 2 3 6 8 8 9)\]

Exercise
Write a function \((\text{at-index } L)\) that consumes a \((\text{listof Int})\) and returns all the values in \(L\) so item \(i\) is at location \(i\).
For example,
\[(\text{at-index } (\text{list } 0 6 2 3 5 6 0 7)) \Rightarrow (\text{list } 0 2 3 7)\]
; . . . . . . . . . . . . . 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)) #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?
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 \texttt{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: \((\texttt{list} \ 2 \ 9 \ 7 \ 4 \ 6)\).

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? \texttt{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)

It works!
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:

```scheme
(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:

```scheme
;; (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)))]))
```

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

```scheme
(def (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))
```
Recall how `foldr` works. It has three parameters: a combining function, a base value, and a list.

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

```scheme
(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`.

```scheme
(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, +.

```scheme
(rsum (list 3 5 7)) => (+ 3 (rsum (list 5 7))) => (+ 3 (+ 5 (rsum (list 7))))
=> (+ 3 (+ 5 (+ 7 (rsum '())))) => (+ 3 (+ 5 (+ 7 0)))
```
Sometimes we have data in two or more lists, and need to do computation on the lists together. We identify three important cases:

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

\[
\text{my-append} (\text{list } 1 \ 2 \ 3) (\text{list } 4 \ 5 \ 6) \Rightarrow (\text{list } 1 \ 2 \ 3 \ 4 \ 5 \ 6)
\]

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

\[
\text{add-pairs} (\text{list } 1 \ 2 \ 3) (\text{list } 5 \ 8 \ 6) \Rightarrow (\text{list } 6 \ 10 \ 9)
\]

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

\[
\text{merge} (\text{list } 2 \ 3 \ 7) (\text{list } 4 \ 6 \ 8 \ 9) \Rightarrow (\text{list } 2 \ 3 \ 4 \ 6 \ 7 \ 8 \ 9)
\]
Adding to a list

Inserting an item at the front of a list is easy: \( \text{cons} \ 7 \ (\text{list} \ 5 \ 3 \ 2) \) \( \Rightarrow \) \( \text{list} \ 7 \ 5 \ 3 \ 2 \)

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

\[
\text{add-end} \ n \ L \ \text{add} \ n \ \text{at the end of} \ L.
\]

\[
\text{add-end}: \ (\text{listof} \ \text{Any}) \ \text{Any} \ \Rightarrow \ (\text{listof} \ \text{Any})
\]

;; Example:

(check-expect (add-end (list 2 3 5) 7) (list 2 3 5 7))

(\text{define} \ (\text{add-end} \ L \ n) \ (\text{cond} \ [\ (\text{empty?} \ L) \ (\text{cons} \ n \ '())] \ [\ \text{else} \ (\text{cons} \ (\text{first} \ L) \ (\text{add-end} \ (\text{rest} \ L) \ n))]))

How much harder would it be to append a list instead of just a number?

Exercise

Use recursion to complete append-lists.

\[
\text{append-lists} \ L1 \ L2 \ \text{form a list of the items in} \ L1 \ \text{then} \ L2, \ \text{in order.}
\]

\[
\text{append-lists}: \ (\text{listof} \ \text{Any}) \ (\text{listof} \ \text{Any}) \ \Rightarrow \ (\text{listof} \ \text{Any})
\]

;; Example:

(check-expect (append-lists (list 3 7 4) (list 6 8)) (list 3 7 4 6 8))
We do not need to recurse through \( L_2 \) in order to append it to \( L_1 \). \( L_2 \) is present in the recursion, and is passed to the next recursive call.

We use \texttt{first} and \texttt{rest} on \( L_1 \), just like in single-list recursion.

The template looks like this:

\[
\text{(define \ (my-alongforride-template \ L_1 \ L_2)}
\begin{array}{l}
\text{(cond} \\
\text{\ \ [(empty? \ L_1) ... ]} \\
\text{\ \ [else \ ... \ (first \ L_1) ...} \\
\text{\ \ \ \ ... \ (my-alongforride-template \ (rest \ L_1 \ L_2) \ ...))]})
\end{array}
\]

\text{21 CS 115 Module 7: Advanced Recursion}
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.
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-produce (list 2 3 5) (list 7 11 13)) 112)
Here we are consuming the two lists at the same rate, and they are of the same length. When one becomes empty, the other does too.

```
(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

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

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. 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))
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, \((\text{cons? } L)\) gives the same answer as \((\text{not } (\text{empty? } L))\). You may use either.
Some examples using prime factor decomposition (pfd)

;; A PFD, or prime factor decomposition, is a (listof Nat)
;; Requires:
;;   the elements are in ascending order
;;   the elements are prime numbers.

;; (factorize n) return the prime factor decomposition of n.
;; factorize: Nat -> PFD
;; Examples:
(check-expect (factorize 1) '())
(check-expect (factorize 17) (list 17))
(check-expect (factorize 24) (list 2 2 2 3))
(check-expect (factorize 42) (list 2 3 7))

Exercise: Complete factorize. It may be helpful to consider the count-up template for recursion on a Nat, starting at 2.
The gcd of two numbers $a$, $b$ is the largest number that divides both numbers. Given the prime factor decomposition of two numbers, it is relatively easy to compute the gcd. We want to keep each prime factor that is in both lists. This can be solved using the generic two-list template.

;; (pfd-gcd p1 p2) return the PFD of the gcd of p1 and p2.
;; pfd-gcd: PFD PFD -> PFD
;; Examples:
(check-expect (pfd-gcd (list 2 2 3) (list 2 3 3 5)) (list 2 3))
(check-expect (pfd-gcd (list 2 3 5) (list 3 3 7)) (list 3))
(check-expect (pfd-gcd (list 5 7) (list 3 11)) '())
(check-expect (pfd-gcd (list 5 7) '()) '())
From pfd-gcd to pfd-lcm

;; (pfd-gcd p1 p2) return the PFD of the gcd of p1 and p2.
;; pfd-gcd: PFD PFD -> PFD
;; Examples:
(check-expect (pfd-gcd (list 2 2 3) (list 2 3 3 5)) (list 2 3))

(define (pfd-gcd p1 p2)
  (cond
   [(or (empty? p1) (empty? p2)) '()]
   [ (= (first p1) (first p2))
     (cons (first p1) (pfd-gcd (rest p1) (rest p2)))]
   [(< (first p1) (first p2))
     (pfd-gcd (rest p1) (pfd-gcd p1 (rest p2)))]
   [(> (first p1) (first p2))
     (pfd-gcd p1 (rest p2))])))

Exercise

Complete pfd-lcm.

;; (pfd-lcm L1 L2) return the lcm of p1 and p2.
;; pfd-lcm: PFD PFD -> PFD
;; Example:
(check-expect (pfd-lcm (list 2) (list 2)) (list 2))
(check-expect (pfd-lcm (list 2 2 3) (list 2 3 3 5)) (list 2 2 3 3 5))
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"))

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):

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

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

```.scheme
;; (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.

```scheme
;; (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 (add1 bottom) 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?

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 \((\text{listof Nat})\):

- \((\text{first } (\text{list } 1 2 3)) \Rightarrow 1\), which is a Nat.
- \((\text{rest } (\text{list } 1 2 3)) \Rightarrow (\text{list } 2 3)\), which is a \((\text{listof Nat})\).

- \((\text{first } (\text{list } 2 3)) \Rightarrow 2\), which is a Nat.
- \((\text{rest } (\text{list } 2 3)) \Rightarrow (\text{list } 3)\), which is a \((\text{listof Nat})\).

- \((\text{first } (\text{list } 3)) \Rightarrow 3\), which is a Nat.
- \((\text{rest } (\text{list } 3)) \Rightarrow '()\), (the same as empty), which is a \((\text{listof Nat})\).

If \(L\) is a non-empty \((\text{listof } X)\), for any type \(X\):

- \((\text{first } L)\) returns a \(X\)
- \((\text{rest } L)\) returns a \((\text{listof } X)\).

Never use \text{first} or \text{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!

\[(\text{binomials 4}) \Rightarrow (\text{list}
\begin{array}{l}
(\text{list 1}) \quad ; \quad 0 \text{ choose } 0 \\
(\text{list 1 1}) \quad ; \quad 1 \text{ choose } 0, 1 \text{ choose } 1 \\
(\text{list 1 2 1}) \quad ; \quad 2 \text{ choose } 0, 2 \text{ choose } 1, 2 \text{ choose } 2 \\
(\text{list 1 3 3 1}) \quad ; \quad ... \\
(\text{list 1 4 6 4 1})) \quad ; \quad ...
\end{array})\]

I can get one row out of this: \((\text{n-th-item 4 binomials}) \Rightarrow (\text{list 1 4 6 4 1})\)

...and an item out of that row: \((\text{n-th-item 2 (n-th-item 4 binomials)}) \Rightarrow 6\)
Computing binomials

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

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

(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (sub1 n)))]))

;; (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?

\[
\text{binomials 4) => (list (list 1) ; 0 choose 0 } \\
\text{(list 1 1) ; 1 choose 0, 1 choose 1 } \\
\text{(list 1 2 1) ; 2 choose 0, 2 choose 1, 2 choose 2 } \\
\text{(list 1 3 3 1) ; ... } \\
\text{(list 1 4 6 4 1)) ; ...}
\]

We did this kind of thing earlier using \text{map}.

Since \text{binomial} has two parameters, use \text{lambda} to fill in the extra.

To build one row:

\[
\text{;; (make-binomial-row r) return the r-th row of the binomial table.} \\
\text{;; make-binomial-table: Nat -> (listof Nat)} \\
\text{;; Example:} \\
\text{(check-expect (make-binomial-row 4) (list 1 4 6 4 1))}
\]

\[
\text{(define (make-binomial-row r) } \\
\text{ (map (lambda (k) (binomial r k)) (range 0 (+ r 1) 1)))}
\]
Creating two-dimensional data

Now that we have a way to build one row, use \texttt{map} a second time to build all the rows:

\[
; (\text{binomials } n) \text{ return the binomial table up to } n \text{ choose } n.
; \text{ binomials: } \text{Nat} \rightarrow (\text{listof } (\text{listof } \text{Nat}))
; \text{ Example:}
\]
\[
(\text{check-expect } (\text{binomials } 2) \ (\text{list}
(\text{list } 1) \quad ; \ 0 \text{ choose } 0
(\text{list } 1 \ 1) \quad ; \ 1 \text{ choose } 0, \ 1 \text{ choose } 1
(\text{list } 1 \ 2 \ 1)) \quad ; \ 2 \text{ choose } 0, \ 2 \text{ choose } 1, \ 2 \text{ choose } 2

(\text{define } (\text{binomials } n)
 (\text{map } \text{make-binomial-row } (\text{range } 0 (+ n 1) 1)))
\]
How can I use recursion to build a table like this?

\[
\text{(binomials 4) => (list }
\begin{array}{c}
\text{(list 1)} ; 0 \text{ choose } 0 \\
\text{(list 1 1)} ; 1 \text{ choose } 0, 1 \text{ choose } 1 \\
\text{(list 1 2 1)} ; 2 \text{ choose } 0, 2 \text{ choose } 1, 2 \text{ choose } 2 \\
\text{(list 1 3 3 1)} ; \ldots \\
\text{(list 1 4 6 4 1))} ; \ldots
\end{array}
\]

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

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

\text{(define (make-binomial-row-from r i)}
\text{ (cond [(< i r) '()]
\text{ [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.

```scheme
;; (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)]))
```

Exercise

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

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

Before we begin the next module, please

- Read *How to Design Programs* Sections 6, 7.