<table>
<thead>
<tr>
<th>Folding Wrong with Filtered Maps</th>
<th>Drawing Swords</th>
<th>Long Live fold-right!</th>
<th>Substituting Regents</th>
<th>TIARA Is A Recursive Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>
Suppose \( x \) is bound to the list \((1\ 2\ 3\ 4\ 5\ 6\ 7)\). Using map, filter, and/or fold-right, write an expression involving \( x \) that returns:

\[(1\ 4\ 9\ 16\ 25\ 36\ 49)\]
Suppose \( x \) is bound to the list \((1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7)\). Using map, filter, and/or fold-right, write an expression involving \( x \) that returns:

\((1 \ 4 \ 9 \ 16 \ 25 \ 36 \ 49)\)

\((\text{map} \ (\lambda (x) (* x x)) \ x)\)
Suppose \( x \) is bound to the list \((1 2 3 4 5 6 7)\). Using map, filter, and/or fold-right, write an expression involving \( x \) that returns:

\[
\left(\left(1 \ 1\right)\left(3 \ 3\right)\left(5 \ 5\right)\left(7 \ 7\right)\right)
\]
Suppose \( x \) is bound to the list \((1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7)\). Using map, filter, and/or fold-right, write an expression involving \( x \) that returns:

\[
((1 \ 1) \ (3 \ 3) \ (5 \ 5) \ (7 \ 7))
\]

\[
(map \ (lambda \ (x) \ (list \ x \ x)) \ (filter \ odd? \ x))
\]
Suppose x is bound to the list (1 2 3 4 5 6 7). Using map, filter, and/or fold-right, write an expression involving x that returns:

(((2) ((4) ((6) #f))))
Suppose \( x \) is bound to the list \((1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7)\). Using map, filter, and/or fold-right, write an expression involving \( x \) that returns:

\[
((2) \ ((4) \ ((6) \ \#f)))
\]

\[
(fold-right
  \lambda \( x \ acum \)
    (cons (list \( x \)) (list \( acum \)))
  \#f
  (filter even? \( x \))
)
Suppose \( x \) is bound to the list \((1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7)\). Using map, filter, and/or fold-right, write an expression involving \( x \) that returns:

\[
\text{The maximum element of } x: 7
\]
Suppose \( x \) is bound to the list (1 2 3 4 5 6 7). Using map, filter, and/or fold-right, write an expression involving \( x \) that returns:

**The maximum element of \( x \): 7**

\[
(fold-right \quad \text{max} \quad \text{car} \quad x \quad (cdr \quad x))
\]
Suppose \( x \) is bound to the list (1 2 3 4 5 6 7). Using map, filter, and/or fold-right, write an expression involving \( x \) that returns:

The last pair of \( x \): (7)
Suppose $x$ is bound to the list 
(1 2 3 4 5 6 7). Using map, filter, and/or fold-right, write an expression involving $x$ that returns:

The last pair of $x$: (7)

**Answer:** trick question! It’s not possible. Map, filter, and fold-right do not give you access to the original list’s backbone, they only let you see the values.
Draw a box-and-pointer diagram for the value produced by the following expression:

\[
(\text{cons} \ (\text{cons} \ "x" \ \text{nil})
\hspace{1cm}
(\text{cons} \ "y" \ (\text{cons} \ "z" \ \text{nil})))
\]
Draw a box-and-pointer diagram for the value produced by the following expression:

\[
(\text{cons} \ (\text{cons} \ "x" \ \text{nil}) \\
(\text{cons} \ "y" \ (\text{cons} \ "z" \ \text{nil})))
\]
What code will produce the following box-and-pointer diagram?
What code will produce the following box-and-pointer diagram?

```scheme
(define null-null (cons '() '()))
(define x (cons (cons (cons '() null-null) '()) null-null))
```

Write code that will cause the following to be printed:

\((1 2 (3 (((4))) 5))\)
Write code that will cause the following to be printed:

\[(1 \ 2 \ (3 \ (((4))) \ 5))\]
Draw a box-and-pointer diagram for the value produced by the following expression:

\[
\text{(map car}
\text{(list (list 3)}
\text{(list 4)}
\text{(list 5)))}
\]
Draw a box-and-pointer diagram for the value produced by the following expression:

\[
(\text{map car} \\
(\text{list} \ (\text{list} \ 3) \\
(\text{list} \ 4) \\
(\text{list} \ 5)))
\]
Draw the box-and-pointers diagram for the value of the following expression:

```
(fold-right
  append
  '()
  (list (list "a" "b")
       (list "c")
       (list "d" "e" "f")))
```
Draw the box-and-pointers diagram for the value of the following expression:

```lisp
(fold-right
  append
  '()
  (list (list "a" "b")
        (list "c")
        (list "d" "e" "f")))
```

```
NC  NC  NC  NC  NC  NC  NV
```

```
"a"  "b"  "c"  "d"  "e"  "f"
```
Write the following procedure using fold-right:

; Creates a new list with the same elements as lst
(define (copy-list lst) )
Write the following procedure using fold-right:

```
; Creates a new list with the same elements as lst
(define (copy-list lst)
  (fold-right cons '() lst))
```
Write the following procedure using fold-right:

```
(define (append list1 list2)
)
```
Write the following procedure using fold-right:

(define (append list1 list2)
  (fold-right cons list2 list1))
Write a procedure to reverse a list using fold-right (you may also use length, append, list, and/or cons):

```
(define (reverse lst)
  )
```
Write a procedure to reverse a list using fold-right (you may also use length, append, list, and/or cons):

\[
\text{(define (reverse lst)}
\]
\[
\text{ (fold-right}
\]
\[
\text{ (lambda (new accum)
\]
\[
\text{ (append accum
\]
\[
\text{ (list new))}
\]
\[
\text{ '())}
\]
\[
\text{ lst))}
\]
Write the **for-all?** procedure using **fold-right**. It should return `#t` if applying the procedure **pred** to each element of **lst** evaluates to `#t`.

```scheme
;; for-all ?
;; list <A>, (A -> boolean) -> boolean
;; Examples:
;; (for-all? (list 1 3 5 7) odd?) => #t
;; (for-all? (list 1 3 5 6) odd?) => #f
(define (for-all? lst pred) ... )
```
Write the **for-all?** procedure using **fold-right**. It should return `#t` if applying the procedure **pred** to each element of **lst** evaluates to `#t`.

```scheme
;; for-all? :
;;   list<A>, (A -> boolean) -> boolean
;; Examples:
;;   (for-all? (list 1 3 5 7) odd?) => #t
;;   (for-all? (list 1 3 5 6) odd?) => #f
(define (for-all? lst pred)
  (fold-right
   (lambda (x accum)
     (and accum (pred x)))
   #t
   lst))
```
Write the procedure `map` in terms of `fold-right`.

```scheme
(define (map pred lst) ...)
```
Write the procedure `map` in terms of `fold-right`.

```
(define (map pred lst)
  (fold-right
   (lambda (x accum)
     (cons (pred x) accum))
   '()
   lst))
```
Write the value of the final Scheme expression. Assume the expressions are evaluated in order. Use unspecified, error, or procedure where appropriate.

```
((lambda (x) (+ x x)) 5)
```
Write the value of the final Scheme expression. Assume the expressions are evaluated in order. Use unspecified, error, or procedure where appropriate.

\[
(((\text{lambda} \ (x) \ (+ \ x \ x)) \ 5) \implies 10)
\]
Write the value of the final Scheme expression. Assume the expressions are evaluated in order. Use unspecified, error, or procedure where appropriate.

```
(define x 5)
(define y 6)
(let ((x 7)
      (y x))
  (+ x y))
```
Write the value of the final Scheme expression. Assume the expressions are evaluated in order. Use unspecified, error, or procedure where appropriate.

```
(define x 5)
(define y 6)
(let ((x 7)
      (y x))
  (+ x y)) => 12
```
Write the value of the final Scheme expression. Assume the expressions are evaluated in order. Use unspecified, error, or procedure where appropriate.

```
(define x 10)
(define y 20)
(define (foo x)
    (lambda (y) (- x y)))
(((foo y) x)
```
Write the value of the final Scheme expression. Assume the expressions are evaluated in order. Use unspecified, error, or procedure where appropriate.

```
(define x 10)
(define y 20)
(define (foo x)
    (lambda (y) (- x y)))
(((foo y) x) => 10)
```
Write the value of the final Scheme expression. Assume the expressions are evaluated in order. Use unspecified, error, or procedure where appropriate.

```
(define (inc x)
  (lambda (y) (+ y 1)))
(inc 1)
```
Write the value of the final Scheme expression. Assume the expressions are evaluated in order. Use unspecified, error, or procedure where appropriate.

\[
\text{(define (inc x)}
\text{  (lambda (y) (+ y 1)))}
\text{(inc 1) } \Rightarrow \text{ compound procedure}
\]
Write the value of this Scheme expression. Assume the expressions are evaluated in order. Use \textbf{unspecified}, \textbf{error}, or \textbf{procedure} where appropriate.

\[(\text{lambda} (x \ y) (x \ y)) \ \\
(\text{lambda} (z) \ \\
 (\text{lambda} (a) \ \\
 (\text{+} a z)))\]
Write the value of this Scheme expression. Assume the expressions are evaluated in order. Use unspecified, error, or procedure where appropriate.

(((lambda (x y) (x y))
  (lambda (z)
    (lambda (a)
      (+ a z)))
  *)
⇒ compound procedure (note: the procedure will generate an error if evaluated)
What is the time order of growth of the following procedure? You may assume that \( x \) and \( y \) are non-negative integers.

```
(define (bar x y)
  (if (< x 0)
    #t
    (bar (+ x 1) (+ y y))))
```
What is the time order of growth of the following procedure? You may assume that \( x \) and \( y \) are non-negative integers.

\[
\begin{align*}
(\text{define} & \ (\text{bar} \ x \ y)) \\
(\text{if} & \ (\leq \ x \ 0) \\
\#t & \\
(\text{bar} & \ (+ \ x \ 1) \ (+ \ y \ y)))
\end{align*}
\]

\[\Rightarrow \text{infinite (bad test condition)}\]
What is the time order of growth of set-difference?

; set-contains? : set<A>, A \rightarrow boolean \quad \Theta(\log n)
; set->list : set<A> \rightarrow list<A> \quad \Theta(n)
; list->set : list<A> \rightarrow set<A> \quad \Theta(n \log n)
;; Returns the set containing all elements in a that are not in b
(define (set-difference a b)
    (let ((a-list (set->list a)))
        (list->set
            (filter
                (lambda (x)
                    (not (set-contains? b x)))
                a-list))))

; example:
(define a (list 1 2 3 4 5))
(define b (list 3 4 5 6))
(set-difference a b); -\rightarrow (1 2)
What is the time order of growth of set-difference?

; set-contains? : set<A>, A → boolean \( \Theta(\log n) \)
; set->list : set<A> → list<A> \( \Theta(n) \)
; list->set : list<A> → set<A> \( \Theta(n \log n) \)
;; Returns the set containing all elements in a that are not in b
(define (set-difference a b)
  (let ((a-list (set->list a)))
    (list->set
      (filter
        (lambda (x)
          (not (set-contains? b x)))
        a)))))

Time OOG => \( \Theta(n \log n) \)

Note: \( n + (n \log n) + (n \log n) \)
What's the longest time it will take to guess the number?

```
(define (make-adversary number)
  (lambda (x)
    (cond ((< x number) "bigger")
          ((= x number) "found it")
          ((> x number) "smaller"))))

(define (guess-number adversary min max)
  (let* ((mid (quotient (+ min max) 2))
         (reply (adversary mid)))
    (cond ((equal? reply "smaller")
           (guess-number adversary min mid))
          ((equal? reply "found it") mid)
          ((equal? reply "bigger")
           (guess-number adversary mid max))))

;; Usage example:
(guess-number (make-adversary 7) 1 100)
```
What's the longest time it will take to guess the number?

\[
\text{(define (make-adversary number) \ldots)}
\]

\[
\text{(define (guess-number adversary min max)}
\text{  (let* ((mid (quotient (+ min max) 2))
\text{     (reply (adversary mid))))
\text{  (cond ((equal? reply "smaller" )
\text{     (guess-number adversary min mid)))
\text{  ((equal? reply "found it") mid))
\text{  ((equal? reply "bigger" )
\text{     (guess-number adversary mid max))))))}
\]

\text{Answer: \(\Theta (\log (max - min))\)
Write a procedure, **fold-left**, that works like **fold-right**, but processes elements of the list in left-to-right order and is iterative.

```
(define (fold-right op init lst)
  (if (null? lst)
      init
      (op (car lst)
           (fold-right op init (cdr lst))))

(define (fold-left op init lst)
  (if (null? lst)
      init
      (fold-left op (op (car lst) init) (cdr lst))))

(fold-right cons '() (list 1 2 3 4 5))
; => (1 2 3 4 5)

(fold-left  cons '() (list 1 2 3 4 5))
; => (5 4 3 2 1)
```
Write a procedure, **fold-left**, that works like **fold-right**, but processes elements of the list in left-to-right order and is iterative.

\[
\begin{align*}
\text{(define (fold-right op init lst)} & \text{)} \\
& \text{(if (null? lst)} \\
& \quad \text{init} \\
& \quad (op (car lst)} \\
& \quad \text{(fold-right op init (cdr lst))))) \\
\text{(define (fold-left op init lst)} & \text{)} \\
& \text{(if (null? lst)} \\
& \quad \text{init} \\
& \quad \text{(fold-left op (op (car lst) init)} \\
& \quad \text{(cdr lst))))}
\end{align*}
\]
What is the order-of-growth in time and space for unknown-costs?

```
(define (costs-n-n n)
  (if (<= n 0)
      0
      (+ n (costs-n-n (- n 1))))
)

(define (costs-n-1 n)
  (if (<= n 0)
      0
      (costs-n-1 (- n 1))))
)

(define (unknown-costs n)
  (define (helper n1 n2)
    (if (>= n1 (* n2 n2 n2))
        (costs-n-n (costs-n-1 n1))
        (helper (+ n1 2) n2)))
  (helper 1 n))
```
What is the order-of-growth in time and space for unknown-costs?

```
(define (costs-n-n n)
  (if (<= n 0)
      0
      (+ n (costs-n-n (- n 1))))

(define (costs-n-1 n)
  (if (<= n 0)
      0
      (costs-n-1 (- n 1))))

(define (unknown-costs n)
  (define (helper n1 n2)
    (if (>= n1 (* n2 n2 n2))
      (costs-n-n (costs-n-1 n1))
      (helper (+ n1 2) n2)))
  (helper 1 n))

;; OOG time: n^3  (Notes: n^3 + n^3 + 1)
;; OOG space: 1    (Notes: 1 + 1 + 1 - final call to costs-n-n is passed 0)
```