**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**; CS442 Assignment 1**

**; h55li 20300403**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**; Question 1**

**;PART A**

(define foldr

(lambda (f i L)

(if (null? L)

i

(f (car L) (foldr f i (cdr L)))

))

)

**;PART B**

**; 1**

(foldr + 0 L)

**; 2**

(foldr

(lambda (unused i)

(+ i 1))

0 L)

**; 3**

(foldr

(lambda (new max)

(if (> new max) new max))

0 L)

**; 4**

(foldr

(lambda (a b)

(if (not b) (p a) b))

#f L)

**; 5**

(foldr

(lambda (arg tail)

(cons (f arg) tail))

'() L)

**; 6**

(foldr cons M L)

**;PART C**

1.

A simple implementation of foldr may not use tail-recursion. So foldl would use less (O(1)) memory to run. However we can implement one in terms of the other by first reversing L. This means that an O(n) operation can make these two functions equivalent. Therefore, there's no (significant) difference between the two.

2.

This implementation is more efficient because it stops evaluating when one of the elements of the list evaluates to true. So this implementation's average runtime is n/2 while the fold implementation has an average runtime of n.

**; Question 2**

**; a1q2a.scm**

(define kill3

(lambda (L)

(set! L

(set-cdr! L

(cons (car (cdr L))

(cdr (cdr (cdr L))))

)

))

)

**; a1q2b.scm**

(define counter 0)

(define countme

(lambda ()

(set! counter (+ counter 1)) counter

)

)

**; a1q2c.scm**

(define L '(0 0 0 0))

(map

(let ((x 0))

(lambda (y) (set! x (+ x 1)) (+ y x))

)

L

)