Problem 1

In recursive procedures each procedure calls itself usually with simpler arguments but also with pending operations.

Since there are pending operations the call stack has to grow each time the function calls itself, this results in linear space complexity.

In iterative procedures there are still self-calls, but since there are no pending operations the callstack doesnt have to get bigger.

The return value of the inital call changes to the newly called procedure, eliminating the need to grow the callstack.

Usually, recursive implementations are easier to code and they tend to require less lines of code to implement, however, their linear space complexity is linear, wheres the linear complexity of iterative approaches are constant.

This results in iterative approaches being more performant compared to recurisive ones.

Problem 2a

```
(define (even? n)
  (= (modulo n 2) 0))


(define even-odd
  (lambda (lst)
    (if (eq? lst '())
      0
      (if (even? (car lst))
         (+ (car lst) (even-odd (cdr lst)))
         (- (even-odd (cdr lst)) (car lst))
      )
    )
  )
)

;(display (even-odd '(1 2 3 4)))
;(display (even-odd '(1 2 3 5 8 13)))
```

Problem 2b

```
(define even-odd-iter
  (lambda (list)
    (even-odd-iter-helper 0 list)
  )
)

(define even-odd-iter-helper
  (lambda (n list)
    (if (eq? list '())
```

```
      n
      (if (even? (car list))
        (even-odd-iter-helper (+ n (car list)) (cdr list))
        (even-odd-iter-helper (- n (car list)) (cdr list))
      )
    )
  )
)
;(display (even-odd-iter '(1 2 3 4)))
;(display (even-odd-iter '(1 2 3 5 8 13)))
```

Problem 3

```
;Previous idx_getter from ps1
(define idx_getter
  (lambda (L i) ; L: list, i: index
    (cond
      ((null? L) '())
      ((eq? i 0) (car L)) ;Found i'th index, return its element
      (else (idx_getter (cdr L) (- i 1))) ;Iterate through the list untill we reach the index
    )
  )
)

(define newLst '())

(define swap
  (lambda (i j lst)
    (cond
      ((eq? i 0) (append newLst (idx_getter lst j)))
      ((eq? j 0) (append newLst (idx_getter lst i)))
      (else (append newLst (list (car lst))) (swap (- i 1) (- j 1)) (cdr lst))
    )
  )
)

(define swap-left (lambda (a list) (swap-left-helper 0 a list '() ) ) )
(define swap-left-helper
  (lambda (n a lst newlist)
    (if (eq? n a)
        newlist
        (swap-left-helper (+ n 1) a (cdr lst) (append newlist (list (car lst)))))))
```