Problem 1

Top Down (recursive) Approach:

In the top down approach we reduce the problem untill we hit a base case. When the base case is hit in the call stack and a value is finally gotten from a procedure call, the call stack collapses. During this collapse all of the pending operations are calculated and we receive our final value.

Definition:

Square number is a number that can be expressed as the square of an integer.

S is the set of square numbers.

For any integer, if n is a integer, n^2 is in S.

Code Implementation:

```
(define (top-down-squares n)
  (if (eq? n 0)
      0
      (cons (top-down-squares (- n 1)) (* n n))
  )
)
```

Bottom-Up (Iterative) approach:

In the bottom up approach we do not have any pending operations. The recursive calls are done through passing parameters and not having any pending operations.

Definition:

Let S represent the set of square numbers.

If an integer x can be written as n^2 by another n(integer) x is in this set.

Code Implementation:

```
(define (bottom-up-helper lst n)
  (if (eq? n 0)
      lst
      (bottom-up-helper (cons (* n n) lst) (- n 1))
  )
)

(define (bottom-up-squares n)
  (bottom-up-helper '() n)
)
```

Rules of Inference approach:

By using the hint we can establish a solution.

First we use axioms, 0 and 1 are square numbers.

Then, we use $(x + y)^2$ is a square number, if x and y are both square numbers.

Then, we can conclude 4 is a square number since $(1 + 1)^2$ is (1 + 2 + 1) from the formula.

Since 2 is available now, we can conclude that $(1+ 2)^2$ 9 is a square number.

Code Implementation:

```
(define (infer-new-square x y)
 (let ((x-squared (* x x))
      (y-squared (* y y)))
   (let ((two-xy (* 2 x y))
        (x-plus-y (+ x y)))
     ; calculate (x+y)^2 using the equation: (x+y)^2 = x^2 + 2xy + y^2
     (let ((new-square (+ x-squared two-xy y-squared)))
      new-square))))

(define known-x 2)
(define known-y 3)

(display (infer-new-square known-x known-y))
```

Problem 2

```
(define repeatN (lambda (lst n) (repeatN-helper lst n 0 '() )))

(define repeatN-helper (lambda (lst n cnt newLst) (if (eq? lst '())

                                        newLst

                                        (if (eq? n cnt) (repeatN-helper (cdr lst) n 0 newLst) (repeatN-helper lst n
(+ cnt 1) (append newLst (list (car lst))))))

                                        ))
```

Problem 3

```
(define (even? n)

 (= (modulo n 2) 0))


(define sum-even (lambda (lst) (foldl + 0 (filter even? lst))))
```

Problem 4

```
(define isIn? (lambda (lst n ) (if (eq? lst '()) #f

            (if (list? (car lst)) (isIn? (car lst) n) (if (eq? n (car lst))

               #t

               (isIn? (cdr lst) n))

               )

            )

           ))

(define myProc (lambda (pred lst1 lst2 op init) (if (not(eq? lst1 '()))

                                      (if (isIn? lst2 (car lst1)) (myProc isIn? (cdr lst1) lst2 op (op init (car lst1)))

                                         (myProc isIn? (cdr lst1) lst2 op init)

                                      )

                                      init

                                      )))

(myProc isIn? '(1 5 8 13) '(1 2 3 4 5) + 0) ;returns 6 (1,5 isIn the lst2 1+5=6)

(myProc isIn? '(1 5 8 13) '(1 5 10 12) * 1) ; returns 5

(myProc (lambda (ls2 n) (not (isIn? ls2 n))) '(1 5 8) '(1 2 5) + 0) ; returns 8

(myProc isIn? '(1 5 8 13) '(1 (2 (3 4) 8)) / 1) ; returns 1/8
```