```
"""#-------------------------------------------------------------------------------------#"""
"""#----------------------      METU Cognitive Sciences         ----------------------#"""
"""#----------------------          Turgay Yıldız                ----------------------#"""
"""#----------------------      yildiz.turgay@metu.edu.tr       ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""



"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.1                ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""
```

Write LAMBDA expressions that

returns the greatest of two integers.

```
(lambda (x y) (if   (> x y)  x   y ) )
```

given two integers, returns T if one or the other divides the other without remainder.

```
(lambda (x  y) (cond    ( (zerop (rem x y))             t)
                        ( (zerop (rem y x))             t)
                        ( t                             nil)
    ))
```

given a list of integers, returns the mean.

```
(lambda (lst)  (/ (reduce #'+ lst) (length lst)) )
```

given a list of integers, returns the sum of their factorials – use your factorial solution.

```
(lambda (lst)  (reduce #' + (mapcar #' factorial lst)) ; assume we have factorial
)
```

```
"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.2                ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""
```

Define a procedure PAIR-PROD using MAPCAR and LAMBDA, which takes a list of two element lists of integers and returns a list of products of these pairs. E.g. an input like

((7 8) (1 13) (4 1)) should yield (56 13 4).

```
(defun pair-prod (lst)

    (mapcar #' (lambda (x) (* (car x) (car (cdr x))) )  lst)

    )

"""#----------------------          Second Way              ----------------------#"""

(defun pair-prod2 (lst)

    (mapcar #' (lambda (x)

        (reduce #' *  x))
      lst)
)
```

```
"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.3                ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""
```

Define a procedure that takes two lists as input and returns the list of their pairwise averages. Use only MAPCAR, LAMBDA and arithmetic operations in your definition.

```
(defun func (x y)

    (mapcar  #' (lambda (a b)  (/ (+ a b) 2)  )  x  y )
)
```

```
"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.4                ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""
```

Define your own REMOVE-IF.

```
* (remove-if #' oddp '(1 2 3 4 5 6) )

                        (2 4 6)


(defun  my_remove_if (func1 lst)

    (if (endp lst)
        nil
        (if     (funcall func1 (car lst))
                (my_remove_if func1 (cdr lst))
                (cons (car lst) (my_remove_if func1 (cdr lst)))
        )
    )
)
"""#----------------------          Second Way              ----------------------#"""

(defun  my_remove_if2 (func1 lst result)

    (dolist (i lst result)

        (if     (funcall func1 i)
                nil
                (setf result (cons i result))
        )
    )
)

"""#--------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.5                ----------------------#"""
"""#--------------------------------------------------------------------------------------#"""

Define LENGTH using MAPCAR, LAMBDA, + and APPLY.

'(1 2 3 4 5) -> 5

(defun my_length (lst)

    (apply  #' + (mapcar #' (lambda (x)
                                    1
                            )
                            lst
                )
    )
)

"""#--------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.6                ----------------------#"""
"""#--------------------------------------------------------------------------------------#"""

Define a procedure that takes an integer n and gives a list of n random single digit
numbers. Use the built-in RANDOM, MAKE-LIST, MAPCAR and LAMBDA in your solu-
tion. Check the definition of the builtins you are not familiar with from reference
books on the website or on the web.

3   ->   '(1 9 4)

(defun  make_random_list (n)

        (mapcar  #' (lambda (x)
                    (random 10)
                    )
                    (make-list n))
    )

"""#--------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.7                ----------------------#"""
"""#--------------------------------------------------------------------------------------#"""

Define a procedure that takes two lists: a list N of numbers and a list P of symbols
with function bindings, i.e. symbols used to define some single argument mathe-
matical procedure with DEFUN. Your procedure should return a list with the same
size as N, whose elements are lists consisting of values obtained by applying all
the procedures in P to the corresponding element in N. For example, if you pro-
vide your procedure with a list of symbols naming square, absolute value and
float functions,

e.g. (sqr abs float), and the list (1 -2 3),

it should return:
                ((1 1 1.0) (4 2 -2.0) (9 3 3.0))

You are NOT allowed to use any procedure (built-in or user-defined)
other than #', DEFUN, MAPCAR, LAMBDA and FUNCALL


(defun   my_func (lst1    lst2)

            (mapcar  #' (lambda (x)
                (mapcar  #'  (lambda  (y)
                    (funcall y x)                   ; since y will be send like 'y  otherwise unbound. no need for #'
                                                    ; mapcar sends values like '(a b c d)   ->   'a 'b 'c 'd

                            )
                        lst2)
                        )
                lst1)
)


(my_func '(1 -2 3) '(sqrt abs float)  )    output value is

((1.0 1 1.0) (#C(0.0 1.4142135) 2 -2.0) (1.7320508 3 3.0))
```

```
"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.8                -----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

Define a procedure APPLIER that takes a procedure "proc", an input "input" and a
count "cnt", and gives the result of applying proc to input cnt times. For instance,

(APPLIER #'CDR '(1 2 3) 2) should give (3)


(defun  applier (proc  input_ count_ )

    (dotimes (i count_ input_)

        (setf input_ (funcall proc input_))
    )
)
"""#----------------------          Second Way              -----------------------#"""

(defun  applier2 (proc  input_ count_)

    (last (mapcar #' (lambda (x)

        (setf input_ (funcall proc input_))
        )

        (make-list count_ :initial-element proc))
    )
)

"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.9                -----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

Define a procedure MOST, which takes a list and a procedure argument, and returns
the element in the list that gives the highest score when provided as an argument
to the given procedure packed in a list with its score. To get full credit, solve the
task WITHOUT using recursion or iteration, you can use MAPCAR, REDUCE
and LAMBDA besides other built-ins you would need.

* ( most '(0.3 0.5 0.2) : proc # '( lambda ( x ) (* 2 ( log x ))))

            (0.5 -1.3862944)

(defun most (lst &key (proc #'(lambda (x) (* 2 (log x)))))

    (reduce #' (lambda (x y)

        (if     (> (cadr x) (cadr y))
                x
                y
        )       )

        (mapcar #' (lambda (z)

            (list z (funcall proc z))
            )
        lst)
    )
)

* (most '(0.3 0.5 0.2))

(0.5 -1.3862944)

"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.10               -----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

The built-in FIND-IF returns the first element in its second argument that returns T
for its first argument:

* ( find-if #'  (lambda ( x ) ( > x 3))     '(1 3 9 0 4))

9

Define your own version of FIND-IF, which returns the index together with
the element. Remember that indexing starts with 0. For instance your procedure
should return (2 9) for the above invocation, where 2 is the index of 9.

(defun find-if-2 (proc lst)

    (let    ( (result                   nil)
              (len                      (length lst))
              (flag                     0)
            )

        (dotimes  (i len result)

            (if     (and (= flag 0) (funcall proc (nth i lst)))  ; computation efficiency
                    (and (setf result (list i (nth i lst))) (setf flag 1))
                    nil
            )
        )
    )
)

* ( find-if-2  #'(lambda ( x ) ( > x 3))    '(1 3 9 0 4)  )

(2 9)


"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.11               ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

Define a procedure REPLACE-IF, which takes three arguments: a list LST, an item
ITEM and a function TEST, and replaces every element of LST that passes the TEST
with ITEM. You may find using keyword arguments useful (see the lecture notes).
Make use of MAPCAR, LAMBDA and FUNCALL in your solution.

(replace-if '(1 0 1 0 1 0 1 1 1 0 0 1)  '1    'zerop)


(defun REPLACE-IF (lst arg test)

    (mapcar #' (lambda (x)

        (if     (funcall test x)
                arg
                x
        )
                )

    lst)
)

* (replace-if '(1 0 1 0 1 0 1 1 1 0 0 1)  '1    'zerop)

(1 1 1 1 1 1 1 1 1 1 1 1)

"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.12               ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

MAPCAR can work on any number of lists; you only need to be careful to provide a
function with the correct number of arguments. For instance

( mapcar # '( lambda ( x y ) (+ x y )) '(1 2 3) '(4 5 6))

gives (5 7 9). Don't worry if lists are not of equal length, MAPCAR goes as far as
the shortest list.

Define procedures that use MAPCAR and LAMBDA and

• zip two lists together – (zip '(a b) '(1 2)) should give ((A 1) (B 2)).

(defun zip (lst1 lst2)

    (mapcar #' (lambda (x y)

        (list x y)
                )
    lst1 lst2)
)

(zip '(a b c) '(1 2 3) )

((A 1) (B 2) (C 3))

• take three lists: first two will be lists of integers, and the third is a list of
functions. Apply the corresponding function to corresponding arguments.

(func12 '(1 2 3 4) '(1 2 3 4)   '(+ * / - )  )     ->    (2  4  1  0)

(defun func12 (lst1 lst2 lst3)

    (mapcar #' (lambda (x y z)

        (apply z (list x y))
                )
    lst1 lst2 lst3)

)

(func12 '(1 2 3 4) '(1 2 3 4)   '(+ * / - )  )

                                (2 4 1 0)


"""#-----------------------------------------------------------------------------------#"""
"""#----------------------      Differences Between Them        ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""
Funcall sees the elements one-by-one , even lists ! No loop !
```

```
* (funcall 'list  '(a b c) )                        ->          (list '(a b c))              ->  (A B C)
* (funcall 'list  '(a b c)  '(1 2 3) )              ->          (list '(a b c) '(1 2 3))     ->  (  (A B C) (1 2 3)  )

* (funcall 'car  '(a b c) )                         ->          A
* (funcall 'car  '(a b c) '(1 2 3) )                ->          ERROR !     Car takes one element.

Apply sees all the elements in the LAST list at once : like parallel working
If there is only one list. It sees everything in it together
Apply is stingy/miser. It wants everything. But if the outputs is too much.
It knows it can not eat all of that. It just wants the last aggregated data.
Loop for the last element if list !

* (apply 'list  '(a b c) )                          ->          (list 'a 'b 'c)              ->      (A B C)
* (apply 'list  '(a b c)  '(1 2 3) )                ->          (list '(a b c) '1 '2 '3)     ->      ((A B C) 1 2 3)


* (apply 'car  '(a b c) )                           ->          ERROR !     Car takes one element.

Mapcar is different. It takes and gives back. It is very generous/bountiful

* (mapcar 'list  '(a b c) )                         ->              ((A) (B) (C))
* (mapcar 'list  '(a b c)  '(1 2 3) )               ->              ((A 1) (B 2) (C 3))


* (funcall '+   1 2 3 )                                 ->      6
* (apply   '+   1 2 3 '(4 5 6)  )                       ->      21   funcall would give error for this input.

* (mapcar  '+   '(1 2 3 )   )                           ->      (1 2 3)
* (mapcar  '+   '(1 2 3 )  '(4 5 6 )   )                ->      (5 7 9)
* (mapcar  '+   '(1 2 3 )  '(4 5 6 ) '(8 9 19))         ->      (13 16 28)

mapcar takes only lists !

"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.13               ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""

                WHEN REAL MATHEMATICS COMES INTO THE GAME !

The way to toss a fair coin in LISP is to do (random 2), which would evaluate to
0 or 1 with a fifty-fifty chance.
Study the following procedure and indicate what the parameters n, f, c and s
stand for. In other words, describe what this procedure computes.

(lets assume n=3 and f=1 , and n=number of heads f=1 for head , then this function computes
many random coin tosses. If it reached the number of succeses = 3 . Then it returns th
number of trials
    )

(defun h (n f &optional (c 0) (s 0))

    (if     (= c n )
            s
            (h n f (+ c (if (= (random 2) f) 1 0)) (+ s 1))))



(defun func (number_success head_or_tail &optional (counter 0) (number_trials 0))

    (if     (= counter number_success)
            number_trials
            (func number_success head_or_tail (+ counter (if (= (random 2) head_or_tail) 1 0)) (+ number_trials 1))
    )
)
Lets try to get 10 Heads in x tosses !

* (func 10 1 )
19              ; found in 19 tosses.
* (func 10 1 )
28
* (func 10 1 )
17
* (func 10 1 )
19
* (func 10 1 )
17
* (func 10 1 )
16
* (func 10 1 )
19
* (func 10 1 )
22
* (func 10 1 )
18
* (func 10 1 )
19
* (func 10 1 )
22
* (func 10 1 )
23
* (func 10 1 )
17
```

```
Lets prove the function !


(defun prove (n counter storage)

    (if (= counter n)
        (float (/ (apply #'+  storage) (length storage)))
        (prove n (+ counter 1) (cons (func 10 1) storage))
    )
)

(prove 100 0 nil )
20.83

* (prove 1000 0 nil )
20.156

* (prove 10000 0 nil )
19.9234

We proved that to get 10 Heads we need to toss the coin nearly 20 times.
Because the probability is 0.5 .

"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.14               ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""

Find the numbers in a given range that have the same Collatz length using the
techniques of this section.

(range = "[1 6)"    -> assume ( (1 1)  (2 5) (3 8) (4 99) (5 8) (6 99))    ->   (3 and 5) (6 and 4))

( defun collatz_generate ( n )

        ( if     (= n 1)
                '(1)
                ( cons n ( collatz_generate ( if    ( evenp n )
                                                    (/ n 2)
                                                    (+ (* n 3) 1))))
        )
)

* (collatz_generate 5)          ->      (5 16 8 4 2 1)


(defun collatz_length ( n )

    (- (length (collatz_generate n )) 1))

(collatz_length 5)              ->      5


I need a func like (range 2 6)    ->   (2 3 4 5)

(defun range (lower upper  storage)

    (if     (= lower upper)
            (reverse storage)
            (range (+ lower 1) upper (cons lower storage))
    )
)

(defun main (lower upper counter)


    (if     (= counter (- upper lower))
            nil

            (let*   ( (rnge                       (range lower upper nil))
                      (result                     nil)
                      (nt                         (nth counter rnge))
                    )

                (cons

                    (dolist (i rnge result)
                        (if     (= (collatz_length i) (collatz_length nt))
                                (setf result (cons i result))
                                nil
                        )
                    )
                    (main lower upper (+ counter 1))

                )
            )
    )
)

* (main 1 20 0)

((1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (13 12)
(13 12) (15 14) (15 14) (16) (17) (19 18) (19 18))
```

```
(defun temp (lst) (if (= (length lst) 1) t nil))


(defun main2 (lower upper)

    (remove-if #' temp (main lower upper 0))
)



"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.15               -----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

* (sort '( (1 2) (1 3) (1 1) (1 6)) #'< :key #'cadr)

(sort   input     func_for_comparison       func_for_transform_input)

((1 1) (1 2) (1 3) (1 6))


( defun collatz_generate ( n )

        ( if     (= n 1)
                '(1)
                ( cons n ( collatz_generate ( if    ( evenp n )
                                                    (/ n 2)
                                                    (+ (* n 3) 1))))
        )
)

(defun collatz_length ( n )

    (- (length (collatz_generate n )) 1))


(defun sorted (lst)  (sort lst #'> :key #'collatz_length))

* (sorted '(1 5 4 3 2 6 7 8 9 ) )

(9 7 6 3 5 8 4 2 1)

"""#-----------------------------------------------------------------------------------#"""
"""#----------------------                  END                 -----------------------#"""
"""#-----------------------------------------------------------------------------------#"""
```