Algorithms: sort

Sort

- Reorder the elements of a vector **`v`** in order to guarantee that

`v[i+1] >= v[i]`

- dozen of algorithms devised since mid XXth century
- algorithm differ in
  - the number of operations necessary to complete the task
  - the influence of the starting conditions (e.g. *quasi ordered* vs *completely* disordered) on the number of operations
  - worst case behavior
  - average behavior
  - best case behavior

Selection sort: a very easy solution

- Idea 1
  - find the minimum value of a subset of the vector and exchange it with the one in the head of the subset of the vector
- Idea 2
  - iterate starting with the whole vector
  - continue disregarding the first element (that now contains the minimum), then the first two, the first three and so on

Trace of the algorithm

Iteration	v[1]	v[2]	v[3]	v[4]
0	200	150	300	100
1	100	150	300	200
2	100	150	300	200
3	100	150	200	300

Shaded areas are not considered in next iteration, since they are already sorted

see an animation of how the algorithm works: https://youtu.be/wnKQsow7ERI

Find the minimum of the positions from i to last

- set the i position as the temporary minimum
- for all the positions j from i+1 to last
    - if position j is smaller than the current temporary minimum
        - set the temporary minimum to j
- at the end the temporary minimum is the position of the minimum of the portion of vector

Selection sort: algo

- for all the positions **i** from the **first** to the **last** minus **1**
  - find position **mini** of the minimum of positions from **i** to **last**
  - swap the positions **i** and **mini**

Selection sort – R solution

algo to R prog

```
selectionsort <- function(v)
{
   lenv <- length(v)
   if (lenv < 2)
      return(v) # no need to sort
   # continuing only if lenv >= 2
   for(i in 1:(lenv-1)){
      mini <- i
      for (j in (i+1):lenv)
        if (v[j] < v[mini]) mini <- j
```

```
# swap values in position i and mini,
#    if different
if (i != mini){
         tmp <- v[i]
         v[i] <- v[mini]
         v[mini] <- tmp
      }
   }
   return(v)
}
```

Selection sort - testing

```
x = sample(1:20000)
n <-1000
ptm <- proc.time()
for(i in 1:n)
      y <- sort(x)
print(proc.time()- ptm)
```

fills a vector with random values

reads machine time before starting

repeats n times the operations, to make numbers readable

shows difference between end time and starting time

Sorting a data frame

- reuse the algorithm developed for the vector
- only two changes
  - specify a parameter with the number of column to be sorted
  - the references to the variable to be sorted must consider the column index

```
selectionSortDf <- function(x, sc) #sc=sorting column
{
  lenx <- nrow(x)
  if (lenx < 2)
    return(x) # no need to sort
  # continuing only if lenx >= 2
  for (i in 1:(lenx - 1)) {
    mini <- i
    for (j in (i + 1):lenx)
      if (x[j,sc] < x[mini,sc])
        mini <- j
    # swap values in position i and mini, if different
    if (i != mini) {
      tmp <- x[i,]
      x[i,] <- x[mini,]
      x[mini,] <- tmp
    } # swap - end
  }
  return(x)
}
```

Bubble sort: another easy solution

- Idea 1
  - compare each element whit the following, from the first to the one before last, if they are unordered swap them
  - bigger elements will move towards the end
- Idea 2
  - repeat if the last loop has changed something
  - if the last loop did not change anything then the vector is sorted

Trace of the algorithm

Swapped values are in *italic*

iteration	v[1]	v[2]	v[3]	v[4]
0	200	150	300	100
1	*150*	*200*	*100*	*300*
2	150	*100*	*200*	300
3	*100*	*150*	200	300
4	100	150	200	300

No change in the last iteration: the algorithm ends

[15 Sorting Algorithms in 6 Minutes - YouTube](https://www.youtube.com/watch?v=kPRA0W1kECg) https://www.youtube.com/watch?v=kPRA0W1kECg

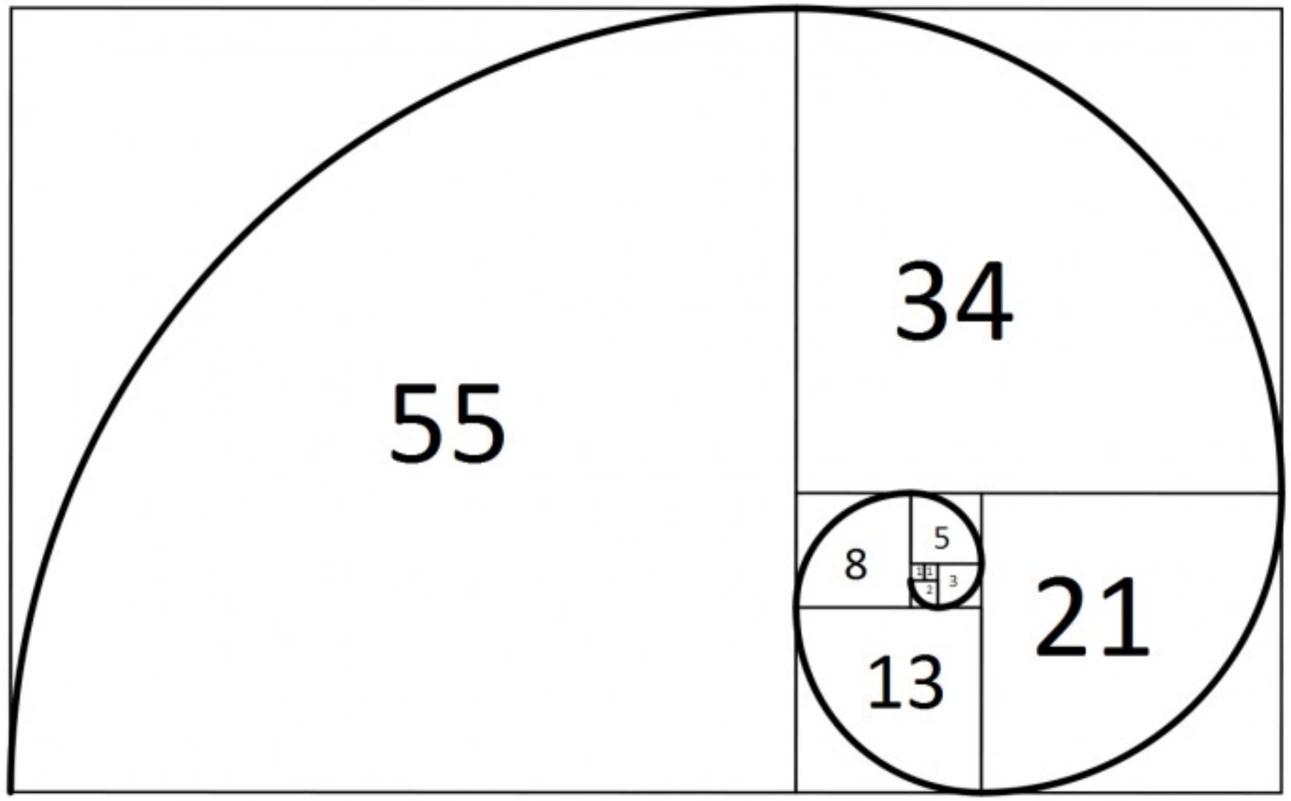
Programming: Recursive functions

Recursive function

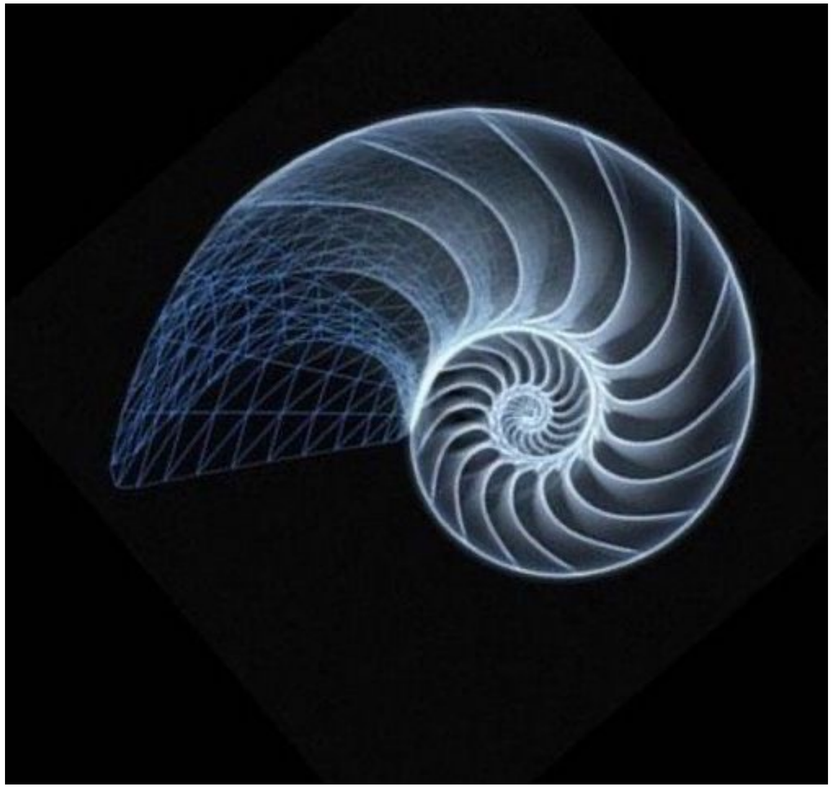
- A function that *calls itself*
- In mathematics there are many examples of functions with a recursive definition
  - n! = n * (n-1)!   if n>1
    0! = 1
  - fibonacci(n) = fibonacci(n-1)+fibonacci(n-2)   if n>2
    fibonacci(1) = 1   fibonacci(2) = 1
    - *curiosity: in most of the flowers the number of petals belongs to the Fibonacci's sequence*
  - determinant of a square matrix with Laplace formula

$$\det M = \sum_{j=1}^{n} (-1)^{i+j} \det M_{ij}$$

Curiosity: graphical representation of the Fibonacci series



Curiosity: graphical representation of the Fibonacci series



Recursive function (ii)

- The recursive call has always *smaller parameters* (in some sense)
  - The parameter of the recursive call is a smaller number
  - The parameter of the recursive call is a smaller matrix
  - …
- There is always at least one *base case* when the function is computed without recursion
- It is quite easy write a program directly from the recursive definition

Factorial – iterative version

n! = n * (n-1)!   if n>1
0! = 1

parameter: n non negative integer
use: i: counter
fact: accumulator
algo:

```
- catch special case n=0
- set fact to 1 (neutral for *)
- repeat varying i from 2 to n
  - set fact to fact * i
- return fact
```

```
factorialIte <- function(n){
  if (n<0)
    return(NULL)
  if (n == 0)
    return(1)
  f <- 1
  for (i in 1:n)
     f <- f*i
  return(f)
}
```

Factorial – recursive version

n! = n * (n-1)!   if n>1
0! = 1

```
factorialRec <- function(n){
  if (n<0)
    return(NULL)
  if (n == 0)
    return(1)
  return(n * factorialRec(n-1))
}
```

Fibonacci – recursive version

if n

    fibonacci(n)
      = fibonacci(n-1)
       +fibonacci(n-2)

else

fibonacci(1) = 1

fibonacci(2) = 1

```
fibonacciRec <- function(n){
if (n <= 0)
    return(NULL)
  if (n == 1 | n == 2)
    return(1)
  return(fibonacciRec(n-1) +
         fibonacciRec(n-2)
        )
}
```

Calls to compute fibonacci(6)

- fib(6)
    - fib(5)
        - fib(4)
            - fib(3)
                - fib(2)
                    - fib(1)
                    - fib(0)
                - fib(1)
            - fib(2)
                - fib(1)
                - fib(0)
        - fib(3)
            - fib(2)
                - fib(1)
                - fib(0)
            - fib(1)
    - fib(4)
        - fib(3)
            - fib(2)
                - fib(1)
                - fib(0)
            - fib(1)
        - fib(2)
            - fib(1)
            - fib(0)

not efficient, since many computations are repeated

Recursive vs iterative

- Recursive programming is easy to understand
  - once you have understood the "trick"
- In general, a problem that is recursively defined allows an iterative solution
  - sometimes it requires some work to be designed
- It is not straightforward to say if the iterative solution is better or worse than the recursive one….
- Sometimes the recursive one is by far worse
- e.g. fibonacci

Fibonacci – iterative version

- the recursive calls are substituted by a loop
- compute fibonacci(6)

n_2	n_1	n=n_1+n_2				
1	1	2	3	5	8	13

Fibonacci – iterative version

- the recursive calls are substituted by a loop

	n_2	n_1	n=n_1+n_2			
1	**1**	**2**	**3**	**5**	**8**	**13**

Fibonacci – iterative version

- the recursive calls are substituted by a loop

		n_2	n_1	n=n_1+n_2		
1	1	2	3	5	8	13

Fibonacci – iterative version

- the recursive calls are substituted by a loop

			n_2	**n_1**	**n=n_1+n_2**	
1	**1**	**2**	**3**	**5**	**8**	**13**

Fibonacci – iterative version

- the recursive calls are substituted by a loop
- compare the number of iterations with the number of calls in the iterative version

				n_2	**n_1**	**n=n_1+n_2**
1	**1**	**2**	**3**	**5**	**8**	**13**

Fibonacci – iterative version

- the recursive calls are substituted by a loop
- compare the number of recursive calls and the number of loop executions…

```
Fibonacci - iterative version
Parameter: n, non-negative integer
Uses:
- n_1, n_2: two previous Fibonacci's
numbers
Algo:
- if n is non positive returns NULL
- if n is 1 or 2 returns 1
- initialize n_1 and n_2 to 1
- repeat n-2 times
  - compute f as n_1 + n-2
  - set n_2 to n_1
  - set n_1 to f
- return f
```

Merge Sort
A recursive algorithm for sorting

Example 037

divide and conquer strategy

parameter: m - vector to sort

result: a sorted vector

use:

- left, right - the two parts of the vector

algo:

```
- if length of m is not more than 1
    return m
- compute middle position using integer arithmetic
- split m in two parts of equal size(+/-1)
- sort left part, if necessary
- sort right part, if necessary
- merge left and right into result
- return the result
```

Merge two sorted vectors

Example **`mergeSort`**

merge two sorted vectors

parameters:

left, right - two sorted vectors
of the same type

result: a sorted vector containing all the elements of left and right

uses:

ll, lr - lengths of the two input vectors

pl, pr, pt - pointers to current position
of left, right, result

```
algo:
  initialize ll and lr
  initialize the result vector with the appropriate length and type
  initialize the pointers to 1
  repeat if both pl and pr still have not reached the end
    if the current position of pl is not more
          than the current position of pr
      copy the current position of left in the result
      advance the current position of left
    else
      copy the current position of right in the result
      advance the current position of right
  if current position of left has not reached the end
      copy the remainder of left in the result
  if current position of right has not reached the end
      copy the remainder of right in the result
  return the result
```

Merge two sorted vectors

Left	Right	Total
3	2	
7	5	
9	8	
15	11	
22	17	

Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	
9	8	
15	11	
22	17	

Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	3
9	8	
15	11	
22	17	

Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	3
9	8	5
15	11	
22	17	

Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	3
9	8	5
15	11	7
22	17	

Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	3
9	8	5
15	11	7
22	17	8

Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	3
9	8	5
15	11	7
22	17	8
		9

Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	3
9	8	5
15	11	7
22	17	8
		9
		11

Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	3
9	8	5
15	11	7
22	17	8
		9
		11
		15

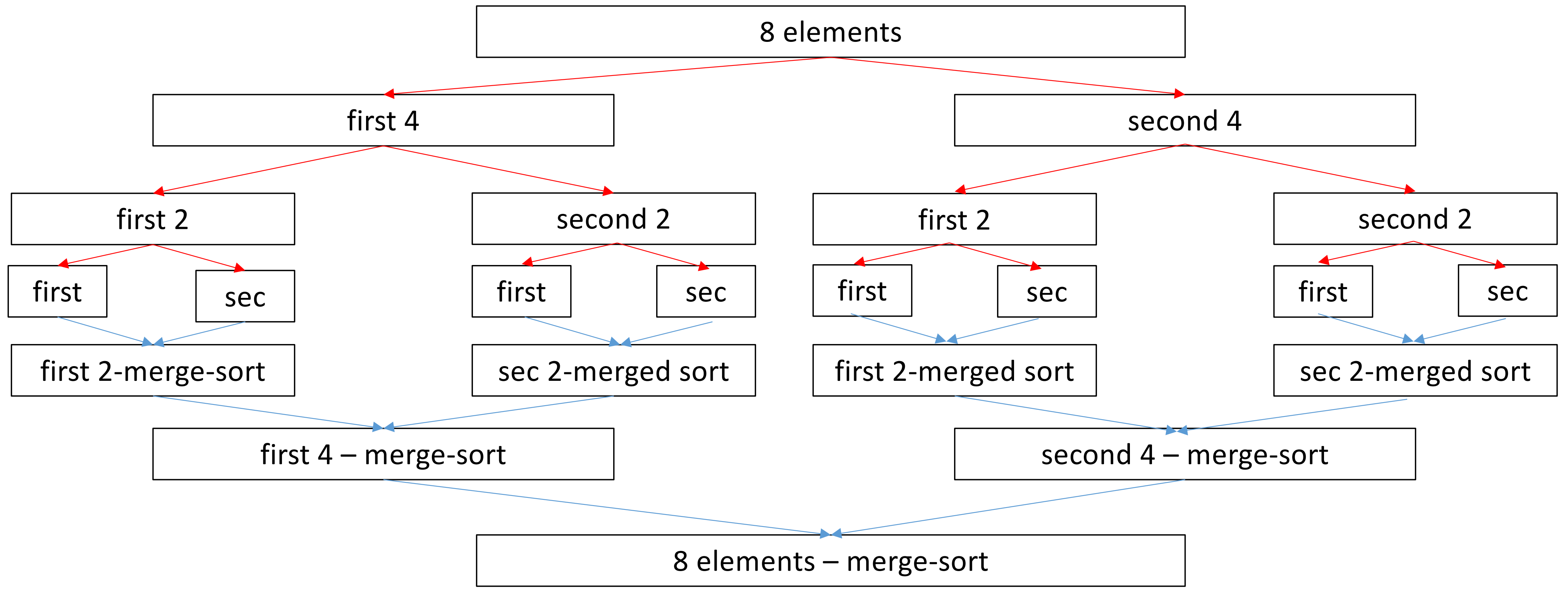
Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	3
9	8	5
15	11	7
22	17	8
		9
		11
		15
		17

Merge two sorted vectors

Left	Right	Total
3	2	2
7	5	3
9	8	5
15	11	7
22	17	8
		9
		11
		15
		17
		22

mergeSort diagram (for 8 elements)



red arrows indicate recursion

blue arrows indicate sorted merge at the end of recursion

Matrix algebra

Matrix echelon form

- A row in a matrix is said to be a *zero row* if it has all zero entries (i.e. it is the zero vector)
- A matrix is said to be in *row echelon form* if
  - all nonzero rows are above all zero ones, and
  - the *leading entry* (i.e. the first nonzero entry from the left of a nonzero row) is always strictly to the right of the leading entry of the row above it
    - the leading entries are also called *pivots*

Check echelon form

1. Prepare a vector containing the column position of the leading element of each row
2. Check the vector of the leading elements to decide if the matrix is echelon

- Write the algorithm
- See example **`rref`**

columns of pivots ?

	[,1]	[,2]	[,3]	[,4]
[1,]	30	10	0	10
[2,]	0	10	20	0
[3,]	0	0	30	10
[4,]	0	0	0	0
[5,]	0	0	0	0

Find columns of pivots - initialization

	[,1]	[,2]	[,3]	[,4]	pivot col
[1,]	30	10	0	10	5
[2,]	0	10	20	0	5
[3,]	0	0	30	10	5
[4,]	0	0	0	0	5
[5,]	0	0	0	0	5

Find columns of pivots

	[,1]	[,2]	[,3]	[,4]	pivot col
[1,]	30	10	0	10	1
[2,]	0	10	20	0	2
[3,]	0	0	30	10	3
[4,]	0	0	0	0	5
[5,]	0	0	0	0	5

- Parameter: matrix m
- Use
  - norows, nocols: initialized to dimensions of m
  - pivots: vector of the leading positions, initialized to nocols+1
- Body of the operations
  - find the pivots
  - repeat varying i from 2 to norows
    - if pivots[i]<=pivots[i-1] AND pivots[i]!=nocols+1
      - Return FALSE
  - return TRUE

necessary to accept the cases with two or more empty rows are at the bottom of the matrix

Transformation to echelon form

- Each matrix can be transformed to echelon form with an adequate sequence of *elementary row operations*
- The elementary row operations do not change the *row space*

Elementary row operations

- row swapping: a row is swapped with another

  $r_i \leftrightarrow r_j$

- scalar multiplication: each element of the row is multiplied by a nonzero scalar

  $r_i \leftarrow \boldsymbol{\alpha}\, r_j$

- row combination: a row is replaced by the sum of itself and the multiple of another row

  $r_i \leftarrow r_i + \boldsymbol{\alpha}\, r_j$

Two matrices are row equivalent if it is possible to change one to another by a finite sequence of elementary row operations

Each matrix can be transformed to row echelon form by means of a finite sequence of elementary row operations, that is each matrix is row equivalent to a matrix in echelon form

Row canonical form

also known as reduced row echelon form (RREF)

- All zero rows, if any, are at the bottom of the matrix
- Each leading nonzero entry in a row is to the right of the leading nonzero entry in the preceding row
- Each pivot (leading nonzero entry) is equal to 1
- Each pivot is the only nonzero entry in its column

See the document **`rref_explained-square.pdf`**

```
# #################################### #
# myRref - my reduced row echelon form #
# #################################### #
```

Algorithm sketch

1. set the current row to 1
2. among the rows from the current to the end choose the one with the leftmost non-zero entry and switch it with the current row; the leftmost non-zero is the pivot
3. with scalar multiplication on the current row set the pivot to 1
4. with row combination, reduce to zero all the entries above and below the current pivot
5. consider as current row the next one and repeat steps 2, 3 and 4, till the end of the rows

```
parameters
  m matrix of numbers
use
  pivot: the column of the current pivot,
         initialized to 1
  norow, nocolumn: number of rows and columns
         in the matrix
  r: index of the row under consideration
  i: index to look for nonzero
         in current pivot column
```

```
algo
get norow and nocol from m
initialize pivot to 1
repeat varying r over rows of m
  if pivot is beyond the right limit of columns
       exit from the loop
  i <- r starts to look for the first nonzero entry in pivot column
  repeat if position i, pivot contains zero
    increment i
    if i is beyond the lower limit of the rows
      (it means that there is no nonzero in the current pivot column)
      start again the search with i<-r and incremented pivot
      if pivot is beyond the right limit of columns
         exit from the function returning m that is now in REF
  (continued from "repeat varying r over rows of m")
  (now nonzero found in i,pivot)
   swap rows i and r (row swapping)
   divide the pivot row (now r) by the pivot element
           to obtain a pivot equal to 1
      (scalar multiplication)
   all the rows but the current pivot row
      are linearly combined with the
      current pivot row
      the linear combination is such that:
            - all the elements to the left and below
              the current pivot remain zero
            - the pivot is the only nonzero
              in its column
   increment pivot
return m
```