//this will be used in the size function for counting the element array

fun int incr (int x, int y) = x+y

//this functions returns the greatest value of a and b

fun int max (int a, int b) = if (b < a) then a else b

fun [int] writeIntArr([int] x) = map(writeInt, x)

fun int writeInt ( int i ) =

let s = write ("\t") in

let y = write (i) in

let z = write ("\t") in y

//create an array with all elements with column 1

fun [int] ones ([int] a) = let result = 0 in map (fn int (int x) => result+1, a)

//this calcuate the size of the array a

fun int size ([int] a) = reduce (incr, 0, ones(a))

//this calculate the length of each row in the bidimensional array b

//columns number

fun [int] sizeB ([[int]] b) = map (fn int ([int] x) => size (x), b)

//this calcuate the module

fun int mod (int n, int q) = n - ((n/q)\*q)

//this function take a as array and returns the greatest value in the array

fun int maxArr ([int] a) = reduce (max, a[0], a)

//this function return an array with in a[0] the row index and in a[1] the column index

fun [int] getIndex (int index, [[int]] array) =

let s = sizeB(array) in

let m = mod(index,s[0]) in

let row = (index/s[0])+1 in

let column = m+1 in

{row, column}

//this functions returns i+j

fun int getPlusIndex ([int] array) =

array[0]+array[1]

//returns the value in the bidimensional array correspondant to the index {i,j}

//where i is the row and j is the column.

//Mind that i and j doesn't start from 0 (as index array in Fasto)

fun int getValue ([int] index, [[int]] array) =

let i = index[0]-1 in

let row = array[i] in

let column = index[1]-1 in

row[column]

fun int checkValue ([int] index, [[int]] array) =

let value = getValue({index[0], index[1]}, array) in

if value == 0 then 1 else 0

//returns 1 if j and j are the same value.

fun int diagIndex ([int] index) =

if index[0] == index [1] then 1 else 0

//this works only for squared matrix

fun int maxIndex ([[int]] matrix) =

size(sizeB(matrix))\*maxArr(sizeB(matrix)) //rownumber\*the max of the column numbers basically the size of the matrix

fun [[int]] createPosMatrix ([[int]] matrix) =

map (fn [int] (int i) =>

getIndex(i,matrix), iota(maxIndex(matrix)))

fun [[int]] diagIndexScan ([[int]] matrix) =

let pos = createPosMatrix(matrix) in

map (fn [int] ([int] x) => if x[0] == x [1] then 1::x else 0::x, pos)

fun int checkResult (int x, int i) =

if (x==1 && i==1) then 1 else 0

fun int product (int a, int b) =

a\*b

fun int signAlgebComplement (int i) =

if (mod(i,2) == 0) then 1 else ~1

//return a partion of the matrix by row, where s is the number of requested rows

fun [[int]] rowselection ([[int]] matrix, int s, int indexRow) =

map (fn [int] (int x) => matrix[indexRow+x], iota(s))

fun [[int]] columnselection ([[int]] matrix, int s, int indexColumn) =

map (fn [int] (int x) => ColumnElement(matrix, indexColumn+x), iota(s))

fun [int] ColumnElement ([[int]] matrix, int c) = map (fn int ([int] x) => x[c], matrix)

fun [[int]] firstRowBlock ([[int]] matrix, int s, int indexRow) =

if s < 1 then secondRowBlock (matrix, s, indexRow) else // controllare se serve if anche sotto

rowselection (matrix, s, 0)

fun [[int]] firstColumnBlock ([[int]] matrix, int s, int indexColumn) =

if s < 1 then secondColumnBlock (matrix, s, indexColumn) else

columnselection (matrix, s, indexColumn)

fun [[int]] secondRowBlock ([[int]] matrix, int s, int indexRow) =

if s < 1 then {{0000000000000000000}} else

rowselection (matrix, s, indexRow)

fun [[int]] secondColumnBlock ([[int]] matrix, int s, int indexColumn) =

columnselection (matrix, s, indexColumn)

fun [[int]] VectorColumnMatrix ([[int]] matrix) =

map (fn [int] (int x) => traspose (x, matrix) , iota(maxArr(sizeB(matrix))) )

fun [int] traspose (int x, [[int]] matrix) =

map (fn int ([int] y) => y[x], matrix)

fun [int] addset ([int] q, [int] b) =

let z = {b[0]} in

map (fn [int] (int x) => q[x]::z, iota(size(q)))

fun [[int]] composeBlocks ([[int]] matrix, int i) =

let address = (getIndex(i, matrix)) in

let firstColumns = address[1]-1 in

if firstColumns < 1 then

let secondColumns = size(sizeB(matrix))-1 in

let b1 = secondColumnBlock (matrix, secondColumns, firstColumns+1) in

let b = VectorColumnMatrix(b1) in

let firstRows = 0 in //address[0]-1 as I need to take all the rows before the index

let secondRows = size(sizeB(matrix))-firstRows-1 in

let result = secondRowBlock (b, secondRows, firstRows+1) in

result

else

let b0 = firstColumnBlock (matrix, firstColumns, 0) in

let secondColumns = size(sizeB(matrix))-1-firstColumns in

let b1 = secondColumnBlock (matrix, secondColumns, firstColumns+1) in

let b = VectorColumnMatrix(b1) in

let a = VectorColumnMatrix(b0) in

let resCol = map (fn [int] (int x)=> calcVal (a[x], b[x]), iota(size(sizeB(a)))) in

let firstRows = address[0]-1 in //address[0]-1 as I need to take all the rows before the index

let secondRows = size(sizeB(matrix))-firstRows-1 in

if firstRows < 1

then let result = secondRowBlock (resCol, secondRows, firstRows+1) in

result

else

let result = secondRowBlock (resCol, secondRows, firstRows+1) in

result

fun [int] calcVal ([int] ael, [int] bel) =

if size(ael) < 2 then ael[0]::bel else

let s = size(ael)-1 in

let res = ael[s]::bel in

let aelnew = map (fn int (int x) => ael[x], iota(size(ael)-1)) in

calcVal(aelnew, res)

fun [int] determinant ([[int]] matrix) =

// if size(sizeB(matrix)) < 3 then (getValue (1,matrix)\*getValue(4,matrix))-(getValue(3,matrix)\*getValue(2,matrix))

// else

let m2 = composeBlocks(matrix,0) in

let sign = map (fn int (int x)=> signAlgebComplement(getPlusIndex(getIndex(x,matrix))), iota(size(sizeB(matrix)))) in

m2

fun [int] main () =

let matrix = {{1,2,3},{4,5,6},{7,8,9}} in

determinant (matrix)

// let matrix = {{1,2,3,4}, {9,6,7,8}, {0,0,1,0}, {0,0,0,1}} in

//map (fn [[int]] (int x) => composeBlocks(matrix), iota(size(sizeB(matrix))))

// composeBlocks(matrix,2)

// let address = getIndex (index, matrix) in

// //getPlusIndex = i+j

// let iPlusj = getPlusIndex(address) in

// let sign = signAlgebComplement(iPlusj) in

// sign\*determinant(matrix, index)

//composeBlocks (matrix, index)

//instead of calculate (-1)^(i+j) this function

//returns 1 if i+j is even or -1 if i+j is odd

// fun int main () =

// let i = signAlgebComplement(3) in

// write(i) //i sarebbe la sequenza di 1 e -1 da moltiplicare per il minore complementare

realizzazione di programmi scritti in linguaggio Fasto per l'elaborazione di matrici ed il calcolo delle loro operazioni.   
Individuazione di matrici quadrate, diagonali, nulle.  
Prodotto per scalari, somma di matrici, prodotto di matrici.   
Calcolo di matrici trasposte, decomposizione a blocchi, calcolo del determinante.