UMA - LCC

Fundamentals of Programming

`typedef array<array<int, NCOL>,NROW> TMat;`

Prof. Dr. Juan Falgueras

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

IV Matrices

The declaration we use `typedef array<array<int, NCOL>,NROW> TMat;`
For square matrices, easier: `typedef array<array<int, N>,N> TSqMat;`

1 Build a procedure `printMat(TMat m)` that prints the matrix `m` on the screen.

2 Build a procedure

```
void fillRandMat(TMat& m)
```

that fills the matrix `m` on with random numbers. Use the procedure `srand(time(0))` to change the seed of the random sequence and the function `rand()` that returns a pseudorandom number between 0 and `RAND_MAX`. Use later `printMat` from 1

3 Build a procedure that returns a identity square matrix:

```
TSqMat identSqMat();
```

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

4 Build a function to know whether the `TSqMat` that receives is or not diagonal (a **diagonal** matrix has all its elements 0 except those on the main diagonal):

```
bool isDiagonal(TSqMat m);
```

5 Build a function that returns the **sum** of its two `TMat` parameters:

```
TMat sumMats(TMat a, TMat b);
```

6 Build a function that transposes the `TSqMat` that receives, that is the `[i,j]` element swaps with the `[j,i]`

```
TSqMat transpose(TSqMat a);
```

HINT: Be careful not to swap the elements twice. That is, while swapping $a_{ij} \leftrightarrow a_{ji}$. You can traverse only elements over the diagonal (j from $i+1$ to N)

7 Build a function to know whether the `TSqMat` that receives is or not **symmetric** (a matrix is symmetric if coincides with its transpose):

```
bool isSymmetric(TSqMat m);
```

8 Build a procedure to swap two rows of a `TMat`:

```
void swapRows(TMat& m, int r1, int r2);
```

Try declaring the `TMat` as an array of `TRows` and swap the corresponding rows

9 Build a procedure that returns the coordinates (x, y) of its first cell that is a local maximum, that is, is greater than its 8 neighbours or if at the boundaries, greater than its 5 neighbours or if at a corner, greater than its 3 neighbours. Return the first one, running from the top-left to the bottom-right element of the `TMat`. If no cell is maximum, return `-1` in both coordinates.

```
void firstMax(TMat m, int& x, int& y);
```

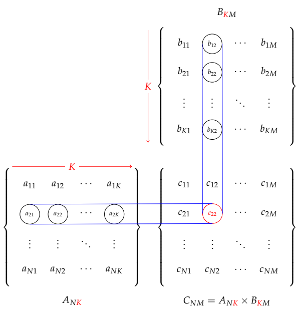
Use the next auxiliary function that returns true to the comparison just when the indexes of the value to compare are outside the matrix:

```
bool gt(TMat m, int v, int i, int j)
{
    return (i>=0 and i<NROWS and j>=0 and j<NCOLS)?
        v > m[i][j]: true;
}
```

10 Multiply two matrices, returning the resulting matrix product:

```
TMatNM multMat(TMatNK a, TMatKM a);
```

Remember that the cell i, j of the result is the scalar product of the i-row of the first by the j-column of the second. As we are not using `TSqMat` we need to declare 3 types of matrices, of dimensions $N \times M$: `TMatNM`; $N \times K$: `TMatNK`; and $K \times M$: `TMatKM`

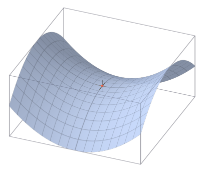


11 Find the greatest sum for the elements of all the possible diagonals of a `TMat`. Take into account that diagonals can go $\searrow$ or $\nearrow$. Some diagonals are shorter than others.

$$\begin{pmatrix} \searrow & \searrow & \dots & a_{0m} \\ \searrow & \searrow & \dots & \searrow \\ \vdots & \vdots & \ddots & \vdots \\ a_{n0} & \searrow & \dots & \searrow \end{pmatrix} \qquad \begin{pmatrix} a_{00} & \nearrow & \dots & \nearrow \\ \nearrow & \nearrow & \dots & \nearrow \\ \vdots & \vdots & \ddots & \vdots \\ \nearrow & \nearrow & \dots & a_{nm} \end{pmatrix}$$

12 A `TMat` has a saddle point when if it is a local maximum respect to its neighbours in the row but a local minimum with respect to its neighbours in the column, or vice versa.

A vertical/horizontal local maximum/minimum is a point that is greater/smaller than its two above-belowleft-right neighbours. If at a border, only the existing neighbour counts.



A `TMat` can have more than one saddle point so, to get them all, build a function that traverses the whole `TMat` and returns another `TMat` full of 0s except at the points where there were saddle points. Local saddle point with maximum in the row: $+1$; local saddle point with maximum in the column: -1; no saddle point: 0

13 A chess square is a `TSqMat` with $N = 8$ with 16 pieces: one king ♔, one queen ♕, two rooks ♖, two knights ♘, two bishops ♗, and eight pawns ♙. They can easily be denoted by an `enum` C++ type or an `int`eger with `const`ants). But we are here interested in a classical problem that is the eight queen puzzle, for which we want to build a check function. The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other

```
bool noThreaten(TSqMat m);
```

You can simply have `TSqMat` defined with booleans instead of integers, just for this problem, so a true means a queen in that cell.

14 Magic Squares

A magic square is a `TSqMat` of $N \times N$ distinct integer numbers (i.e. each number used once), where the numbers in each row, and in each column, and the numbers in the main and secondary diagonals, all add up to the same number, a *magic constant* (that magically is $(n(n^2+1))/2$. A more general formula can be found here).

Except for the size $N = 2$, there can be magic squares for any N. A well known one (the oldest known, dating from 650 BC) is:

8	1	6
3	5	7
4	9	2

The next is as the previous one but with 19 added to each number:

23	28	21
22	24	26
27	20	25

There is an algorithm able to build simple square matrices for odd Ns by H. Coxeter:

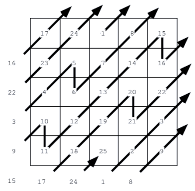
(a) Start in the middle of the upper row with 1.

(b) Go up and right assigning consecutive numbers to the empty cells.

(c) When crossing the matrix boundaries, reset the index(es) to go to the other side of the matrix as if it were repeated indefinitely

(d) toward the fourth directions.

(e) If a cell is already occupied, step down a cell and continue.



Build `void coxeter(TSqMat& magic);` Try with matrices of different odd sizes and print them.

15 Chess knight tour

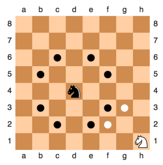
A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square only once.

In this exercise we are not endeavouring to build such kind of tours (it is not so difficult, but left for a future exercise). What we want is to check if a tour, after done, is correct.

To represent the tour of a knight on a chessboard we can use a `TSqMat` of integers (suppose `{N=8}`). The tour starts in a cell with the integer number 1, from that cell the knight has jumped to a legal cell (se below) in which there must be a number 2, and so on. For the tour being complete and correct, every jump must be legal and the last cell to check contain $8 \times 8 = 64$. Build:

```
bool okTour(TSqMat m);
```

A knight can jump to a maximum of 8 positions from where it is: two cells horizontally (either left or right) and one vertical (either up or down); or two cells vertically (either up or down) and one horizontal (either left or right).



As an example of a valid tour see the next:

```
TSqMat m = {{
  {{ 1, 64, 61, 42, 37, 40, 55, 52}},
  {{60, 43,  2, 63, 56, 53, 36, 39}},
  {{ 3, 62, 59, 20, 41, 38, 51, 54}},
  {{44, 19,  4, 57, 50, 21, 32, 35}},
  {{ 5, 58, 45, 18, 31, 34, 49, 22}},
  {{14, 17,  6, 25, 46, 11, 30, 33}},
  {{ 7, 26, 15, 12,  9, 28, 23, 48}},
  {{16, 13,  8, 27, 24, 47, 10, 29}}
}};
```

16 Game of Life

The Conway's Game of Life is cellular automaton in which a initial state of a matrix of cells determines, based on the state of the neighbours of each cell, its next state, or generation.

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, `ALIVE` or `DEAD`. Every cell interacts with its 8 neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, or generation, the following transitions occur:

(a) An alive cell with **fewer than 2** alive neighbours will **die**: underpopulation.

(b) An alive cell with **2 or 3** alive neighbours will remain **alive**.

(c) An alive cell with **more than 3** alive neighbours will **die**: overpopulation.

(d) An dead cell with **exactly 3** alive neighbours will be born alive next generation: reproduction.

So depending on the contents:

no. of neighb.	state		new state
< 2	ALIVE	$\to$	DEAD
$= 2, 3$	ALIVE	$\to$	ALIVE
$= 3$	DEAD	$\to$	ALIVE
> 3	ALIVE	$\to$	DEAD

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

In order to achieve this conditions in a simple C++ program, a `TMat` can be defined. This matrix can be of a size not bigger than a typical terminal screen (around 40 lines by 100 columns). The matrix can be initialised using a `rand()` as in 2 and the matrix can be redraw each generation after erasing the terminal window (what can be done sending to it `cout` the string:

```
const string CLRSCR = "\x1B[H\x1B[J";
```

Yet, perhaps in Windows, you have to call the operating system to do it: `system("cls")`

The main process is:

```
void nextGeneration(TMat& m)
```

A good idea here is to **modify** the matrix instead of copying it modify. The reason is this boards tend to be large to huge, and the pace of producing new boards can be critical. But in order not to interfere previous states with the pretty next, we can state temporary states for each cell as the evaluation of each of the original cell is being taken. After this analysis, a final double-for loop will definitely set the value for each cell. The states of the cells and the temporary states can be described either with `enum TState {DEAD, ALIVE, TODIE, TOBEBORN};` or with

```
typedef int TState;
const int DEAD      = 0;
const int ALIVE     = 1;
const int TODIE     = 2;
const int TOBEBORN  = 3;
```

The borders of the matrix can be considered surrounded either by *nothing* (dead cells) or having as neighbours just the cells of their opposite side as if they were bent over themselves in an infinite repeated surface.

After a generation is displayed, the program can wait a keyboard action or automatically, after `usleep(100000);` (100000 msecs), be erased and substituted by the drawing of the next generation. To initially fill the contents of the `TMat` you can use

```
void arrAleat(TMat& a)
{
  for (int i = 0; i < NROWS; i++)
    for (int j = 0; j < NCOLS; j++)
      a[i][j] = (TState)(rand() % 2);
}
```