Esercizio 1

Scrivere un programma che riceve in ingresso il contenuto di una matrice quadrata di interi con N^2 elementi.
Il programma calcola e visualizza i massimi locali della matrice.
Il massimo locale e' definito come un elemento della matrice i cui adiacenti sono tutti minori o uguali ad esso.

Esempio: data la matrice `M[5][5]` (i cui massimi locali sono rappresentati in grassetto):

$$M = \begin{bmatrix} 7 & 4 & 3 & 2 & 1 \\ 1 & \mathbf{9} & 6 & 4 & 3 \\ 5 & \mathbf{9} & 3 & 1 & \mathbf{6} \\ \mathbf{9} & 8 & 2 & 4 & 3 \\ 4 & 2 & \mathbf{8} & 4 & 1 \end{bmatrix}$$

il programma dovra' stampare a video:

```
M[1,1] = 9
M[2,1] = 9
M[3,0] = 9
M[4,2] = 8
M[2,4] = 6
```

Nota: attenzione al calcolo dei massimi lungo i bordi della matrice.

ESERCIZIO 2 (Game of Life)

Una versione semplificata del *Gioco della Vita* (*Game of Life*) si svolge su una griglia di caselle (cellule) rappresentate con una matrice `N × N`, `N > 2`.
Ogni cellula può essere in 2 stati: "viva" (valore 1) o "morta" (valore 0).
L' intorno di una cellula e' definito come l' insieme dalle sue cellule circostanti.
Lo stato di tutte le cellule evolve simultaneamente a intervalli di tempo discreti: lo stato di una cellula all' istante successivo dipende dallo stato del suo intorno all' istante precedente secondo le regole seguenti.

1. Una cellula morta diventa viva se nell'istante precedente ha esattamente 3 cellule vive nel suo intorno.
2. Una cellula viva muore se nell' istante precedente ha meno di 2 cellule vive nel suo intorno (morte per isolamento) o piu' di 3 cellule vive nel suo intorno (morte per sovraffollamento).

a. Si definisca un sotto-programma `...aliveNeighbours(...)` che valutando la posizione `[riga][colonna]` di una casella in una matrice, restituisca il numero di "vicini vivi".

b. Si scriva un sotto-programma `...GoL(...)` avente come parametri le matrici rappresentanti la griglia di cellule prima e dopo un passo di evoluzione, rispettivamente.

ESERCIZIO 3

```
#define DIM 10
typedef struct Intv { unsigned int r, c; } Casella;
typedef Casella TappetoElastico[DIM][DIM];
```

Pare che tra le discipline degli ottocenteschi "circhi delle pulci" non ci fosse il *tappeto elastico* (non in quello del Prof. Heckler, almeno). Ma avrebbe potuto funzionare così: **la pulce salta sulla prima cella (0,0)** del tappeto elastico (quadrato), atterrandovi legge le coordinate (riga e colonna) della prossima cella su cui saltare, e da lì continua a saltare, ogni volta leggendo le coordinate della cella successiva verso cui saltare. Se/quando le coordinate lette indicano un punto esterno al tappeto, la pulce scende (applausi).

Si implementino le seguenti funzioni in C e si spieghi brevemente come funzionano gli algoritmi usati.

`... ciclico(...)` che riceve in input un tappeto elastico e restituisce `1` se esso obbliga a saltare indefinitamente; restituisce `0` se invece a un certo punto la pulce potra' scendere.

`...contasalti(...)` che riceve un tappeto elastico e misura il numero di salti che la pulce compie prima di scendere (se il tappeto non e' ciclico), oppure `-1`.

`... dicoppia(...)` che controlla se un tappeto e' adatto all'esibizione di coppia: **una seconda pulce inizia a saltare dalla casella `(DIM-1,DIM-1)` contemporaneamente alla prima pulce**, ed esse continuano a saltare e atterrare in perfetta sincronia fino a uscire assieme dal tappeto, senza mai "scontrarsi" – cioe' atterrare contemporaneamente sulla stessa casella.

ESERCIZIO 4

Considerata una matrice `A` di `N x M` interi, definiamo *claque* una sottomatrice `2 x 2` in cui la somma algebrica dei valori di una diagonale sia pari a quella dell'altra diagonale. In figura sono evidenziate le claque.

Le funzioni `...contaclaque( ... )` e `...senzaclaque(...)` ricevono in ingresso una matrice di interi di dimensione `N x M` e restituiscono, rispettivamente, il numero di claque della matrice e `1` o `0` a seconda che la matrice sia o meno priva di claque.

4	-1	7	0	0
-4	-9	-1	0	0
2	8	16	1	4
-1	7	5	2	5

ESERCIZIO 5

Quando due topini si sfidano al *tunnel delle tentazioni* attraversano dei percorsi costituiti da tratti liberi (L), occupati da formaggio (F), da gatti (G) o da topine (T). Vince il topino che per primo arriva alla fine del suo tunnel (tutti i tunnel sono diversi e generati casualmente).

La struttura dati e' definita come segue:

```
typedef struct top {
    int  peso_cor, peso_max, peso_min, voracita;
} Topo;
typedef enum { L, G, F, T; } tratto;
typedef tratto[100] Tunnel;
```

Ogni topino inizia la corsa con il suo peso corrente, e impiega 5s (secondi) per attraversare un tratto libero (trascuriamo il dimagrimento dovuto alla corsa). I tratti non liberi causano i seguenti comportamenti:

(G): il topino si spaventa, perde 10g (grammi) di peso, e accelera, percorrendo il tratto in soli 2 secondi;

(F): il topino mangia fino a raggiungere il suo `peso_max`, alla velocita' espressa (in g/s) dalla sua voracita';

(T): il topino indugia per 30 secondi nelle varie pratiche connesse al corteggiamento, e perde 4g di peso.

Attenzione: se, in ogni momento, un topino scende sotto il suo `peso_min`, perde i sensi (e la sfida).

(a) Si codifichi la funzione

```
int sfida(Topo *tp1, Topo *tp2, Tunnel tn1, Tunnel Tn2)
```

che restituisce 1 se vince il topino tp1, 2 se vince il topino tp2, 0 in caso di pareggio (stesso tempo o svenimento di entrambi).

Ci si avvalga della funzione

```
int tempo(Topo *tp, Tunnel tn)
```

che restituisce il tempo in secondi impiegato da `tp` a percorrere `tn`, o `-1` se il topino perde i sensi.

(b) Si codifichi anche la funzione `...tempo(...)`