ESERCIZIO 1

Abbiamo una matrice M di interi di dimensione $n \times n$ con $n > 1$. Una *discesa* su questa matrice è una sequenza di n celle della matrice con i seguenti vincoli

- le celle appartengono a righe diverse della matrice
- la prima cella appartiene alla prima riga della matrice
- ogni altra cella è adiacente (in verticale o in diagonale) alla cella che la precede.

Il *valore* di una discesa è dato dalla somma dei valori delle sue n celle.

Ad esempio, per $M = \begin{pmatrix} 12 & 10 & 3 & 14 & 9 \\ 0 & 1 & 13 & 15 & 13 \\ 8 & 10 & 1 & 2 & 7 \\ 7 & 11 & 10 & 5 & 7 \\ 18 & 4 & 6 & 10 & 0 \end{pmatrix}$

sono evidenziate due possibili discese (la prima vale 29 e la seconda vale 31).

Progettare un algoritmo che, data la matrice M, trova il valore massimo tra i valori delle possibili discese di M.

Per la matrice dell'esempio precedente l'algoritmo deve restituire 66 (la discesa di valore massimo in M è infatti $14, 13, 10, 11, 18$)

L'algoritmo deve avere complessità $O(n^2)$. **Motivare bene la correttezza e la complessità dell'algoritmo proposto**.

Utilizziamo una tabella bidimensionale di dimensione $n \times n$ dove
$T[i,j] =$ il valore massimo per tutte le discese che terminano nella cella $M[i,j]$.
La soluzione sarà $\max_{j=0}^{n}(T[n-1,j])$.

La formula ricorsiva che permette di di ricavare $T[i,j]$ dalle celle precedentemente calcolate è la seguente:

$$T[i,j] = \begin{cases} M[i,j] & i = 0 \\ \max(T[i-1,j-1], T[i-1,j]) + M[i,j] & j = 0 \\ \max(T[i-1,j], T[i-1,j+1]) + M[i,j] & j = n-1 \\ \max(T[i-1,j-1], T[i-1,j], T[i-1,j+1]) + M[i,j] & \text{altrimenti} \end{cases}$$

la ricorrenza segue dal fatto che una discesa che termina nella cella $M[i,j]$ è il prolungamento di una discesa che termina in una delle tre celle $M[i-1,j-1], M[i-1,j]$ o $M[i-1,j+1]$. Il meglio che si può ottenere in termini di valore è dunque dato da $M[i,j] + \max(T[i-1,j-1],\ T[i-1,j]\ T[i-1,j+1])$.
I restanti casi della ricorrenza tengono conto del fatto che la cella sia la prima del cammino o che appartenga alla prima o all'ultima colonna della matrice.

```
def es(M):
    n=len(M)
    T=[[0 for _ in range(n)] for _ in range(n)]
    for j in range(n): T[0][j]=M[0][j]
    for i in range(1,n):
        for j in range(0,n):
            if j==0: T[i][j]=max(T[i-1][j],T[i-1][j+1])
            elif j==n-1:max(T[i-1][j-1],T[i-1][j])
            else: T[i][j]=max(T[i-1][j-1],T[i-1][j],T[i-1][j+1])
            T[i][j]+=M[i][j]
    return max(T[n-1])
```

Complessità $\Theta(n^2)$

ESERCIZIO 2 Il display di un telefonino si presenta come di seguito indicato:

1	2	3
4	5	6
7	8	9
$*$	0	#

Cerchiamo un particolare numero telefonico e sappiamo che:

- il numero è composto da n cifre.
- non contiene cifre uguali adiacenti
- nel comporre il numero sul tastierino basta spostarsi solo tra tasti adiacenti in orizzontale o verticale

Ad esempio, per $n = 7$, la combinazione 12108586996 non è di certo il numero telefonico che cerchiamo a causa della presenza delle seguenti tre coppie di cifre adiacenti 10 e 86 e 99.

Progettare un algoritmo che, dato n, enumera tutte le combinazioni possibili per il numero telefonico da ricercare.
Ad esempio:

- per $n = 1$ l'algoritmo deve stampare le 10 seguenti combinazioni $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ (non necessariamente in quest'ordine)
- per $n = 2$ l'algoritmo deve stampare le 26 seguenti combinazioni : $08, 12, 14, 21, 23, 25, 32, 36, 41, 45, 47, 52, 54, 56, 58, 63, 65, 67, 74, 78,$ $80, 85, 87, 89, 96, 98$ (non necessariamente in quest'ordine).

La complessità dell'algoritmo proposto deve essere $O(nS(n))$ dove $S(n)$ è il numero di combinazioni da stampare.
Motivare bene la complessità del vostro algoritmo.

Algoritmo:

- per ognuno delle 10 cifre decimali eseguiremo una funzione di backtracking che enumera tutte le combinazioni possibili lunghe n che partono da quella cifra.

- la funzione di backtraking poi produce le possibili combinazioni facendo attenzione ad accodare a ciascuna combinazione parziale solo le cifre adiacenti in orizzontale o verticale all'ultima cifra inserita.

```python
def es(n):
    prossimi={
    0:[8],
    1:[2,4],
    2:[1,3,5],
    3:[2,6],
    4:[1,5,7],
    5:[2,4,6,8],
    6:[3,5,9],
    7:[4,8],
    8:[0,5,7,9],
    9:[6,8]
    }
    for x in range(10):
        esR(n,[x],prossimi)

def esR(n,sol,prossimi):
    if len(sol)==n:
        print(sol)
        return
    for y in prossimi[sol[-1]]:
        sol.append(y)
        esR(n,sol,prossimi)
        sol.pop()
```

- Nota che nell'albero di ricorsione prodotto dall'esecuzione di esR un nodo viene generato solo se porta ad una foglia da stampare.

- Possiamo quindi dire che la complessità dell'esecuzione di $esR([i])$ richiederà tempo

$$O(S(n) \cdot h \cdot f(n) + S(n) \cdot g(n))$$

  dove:

  - $h = n$ è l'altezza dell'albero.
  - $f(n) = O(1)$ è il lavoro di un nodo interno.
  - $g(n) = O(n)$ è il lavoro di una foglia

- Quindi il costo di $esR(n, [x], prossimi)$ è $O(nS(n))$.

- Nota che $es(n)$ esegue 10 volte la procedura $esR(n, [i], prossimi)$, otteniamo quindi che il costo di $es(n)$ è $O(nS(n))$.