ESERCIZIO 1. Progettare un algoritmo che, data una sequenza A ternaria sull'alfabeto $\{0,1,2\}$ calcola il numero di sottosequenze di A del tipo $0^i1^j2^k$, $i,j,k \geq 1$.

Ad esempio:
per $A = [2,0,1,0,1,2,0]$ l'algoritmo deve restituire 5. Le sottosequenze presenti sono: $[-,0,1,-,-,2,-]$, $[-,0,-,-,1,2,-]$, $[-,0,1,-,1,2,-]$, $[-,0,-,0,1,2,-]$, e $[-,-,-,0,1,2,-]$.

L'algoritmo proposto deve avere complessità $O(n)$ dove n è la lunghezza di A. **Motivare BENE la correttezza e la complessità dell'algoritmo proposto**.

Utilizziamo una tabella T di dimensione $(n+1) \times 3$ dove:

- $T[t,0]$ = numero di sottosequenze del tipo 0^i, $i \geq 1$ presenti nelle prime t posizioni di A.
- $T[t,1]$ = numero di sottosequenze del tipo 0^i1^j, $i,j \geq 1$ nelle prime t posizioni di A.
- $T[t,2]$ = numero di sottosequenze del tipo $0^i1^j2^k$, $i,j,k \geq 1$ nelle prime t posizioni di A.

La soluzione sarà in $T[n,2]$.

Per $t = 0$ si ha $T[0,1] = T[0,2] = 0$ mentre $T[0,0] = 1$ se $A[0] = 0$, $T[0,0] = 0$ altrimenti.
Nel caso $t > 0$. Sia $A[t] = a$, possiamo ragionare come segue:

- per $T[t,a]$:
  - *(per $a = 0$) si ha $2T[t-1,a]+1$ (posso aggiungere o meno il simbolo 0 a tutte le sottosequenze che terminavano con 0 e posso avere il solo nuovo simbolo 0)*
  - (per $a > 0$) si ha $T[t,a] = 2T[t-1,a] + T[t-1,a-1]$ (posso aggiungere o meno il simbolo a a tutte le sottosequenze che terminavano con a e posso aggiungere il simbolo a a tutte le sottosequenze che terminavano con $a-1$)
- per $b \neq a$ allora $T[t,b] = T[t-1,b]$ (il numero di sottosequenze che terminano con b non cambia se $A[t] \neq b$)

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

$$T[t,a] = \begin{cases} 1 & \text{se } t = 0 \text{ e } a = 0 \\ 0 & \text{se } t = 0 \\ 2T[t-1,a]+1 & a = 0 \\ 2T[t-1,a]+T[t-1,a-1] & \text{altrimenti} \end{cases}$$

```
def es1(A):
    T=[[0 for _ in range(3)]  for _ in range(len(A))]
    for t in range(len(A)):
        for a in range(2):
            print(t,a,T[t][a])
    if A[0]==0: T[0][0]=1
    for t in range(1,len(A)):
        for a in range(3):
            if A[t]==a and a==0: T[t][a]=2*T[t-1][a]+1
            elif A[t]==a:T[t][a]=2*T[t-1][a]+T[t-1][a-1]
            else: T[t][a]=T[t-1][a]
    return T[len(A)-1][2]
```

```
>>> A=[2,0,1,0,1,2,0]
>>> es1(A)
5
>>> A=[0,1,2,0,1,2]
>>> es1(A)
7
>>> A[0,0,1,1,2,2]
>>> es1(A)
27
```

Complessità $\Theta(n)$

Progettare un algoritmo che, data una sequenza A di cifre decimali lunga n, stampa tutte le sequenze lunghe n che possono ottenersi da A incrementando di uno almeno una cifra diversa da '9' o decrementando di uno almeno una cifra diversa da '0'.
L'algoritmo proposto deve avere complessità $O(nD(n))$ dove $D(n)$ è il numero di sequenze da stampare.
Ad esempio per $A = [1,9,0]$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti sequenze:

$[2,8,1]$, $[2,8,0]$, $[2,9,1]$, $[2,9,0]$, $[0,8,1]$, $[0,8,0]$, $[0,9,1]$, $[0,9,0]$, $[1,8,1]$, $[1,8,0]$, $[1,9,1]$.

Motivare BENE la correttezza e la complessità dell'algoritmo proposto.

Algoritmo:

- Per la stampa delle stringhe eseguiremo una funzione di backtracking.
- Al passo ricorsivo i-esimo possiamo potenzialmente inserire nella soluzione che andiamo costruendo una delle tre cifre $A[i]+1$, $A[i]-1$ e $A[i]$. Per ottenere una complessità proporzionale alle $D(n)$ sequenze da stampare la funzione di backtraking controlla che la scelta di inserire uno delle tre cifre venga fatta solo se la soluzione parziale che si ottiene può essere completata poi come una stringa da stampare. Si inserisce $A[i]+1$ se e solo se $A[i] \neq 9$ mentre si inserisce $A[i]-1$ se e solo se $A[i] \neq 0$. Infine si inserisce $A[i]$ solo se $A[i]$ non è l'ultima cifra della sequenza oppure nella soluzione parziale c'è una cifra modifica. Per quest'ultimo controllo utilizziamo una variabile booleana che viene posta a $True$ se nella stringa è stato modificato almeno un simbolo.

```
def es2(A,i=0, Sol=[],t=False):
    if i==len(A):
        print(Sol)
        return
    if A[i]!=9:
        Sol.append(A[i]+1)
        es2(A,i+1,Sol, True)
        Sol.pop()
    if A[i]!=0:
        Sol.append(A[i]-1)
        es2(A,i+1,Sol, True)
        Sol.pop()
    if t==True or i!=len(A)-1:
        Sol.append(A[i])
        es2(A,i+1,Sol, t)
        Sol.pop()
```

- Nota che nell'albero di ricorsione prodotto dall'esecuzione di $es2$ un nodo viene generato solo se porta ad una foglia da stampare.
- Possiamo quindi dire che la complessità dell'esecuzione di $es(A)$ richiederà tempo

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

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 $es2(A)$ è $O(nD(n))$.

```
>>> A=[1,9,0]
>>> es2(A)
[2, 8, 1]
[2, 8, 0]
[2, 9, 1]
[2, 9, 0]
[0, 8, 1]
[0, 8, 0]
[0, 9, 1]
[0, 9, 0]
[1, 8, 1]
[1, 8, 0]
[1, 9, 1]
```

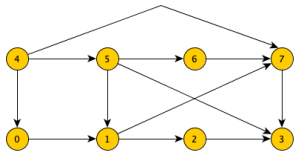
```
>>> A=[9]
>>> es2(A)
[8]
```


ESERCIZIO 3

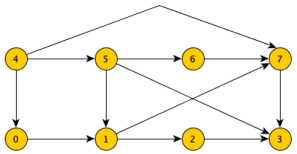
Considerate il grafo nella figura che segue e calcolate un ordine topologico usando l'algoritmo basato su una visita in profondità a partire dal vertice 0. Durante la visita, nel caso l'algoritmo debba esplorare il vicinato di un vertice, lo farà sempre in ordine crescente rispetto all'indice.

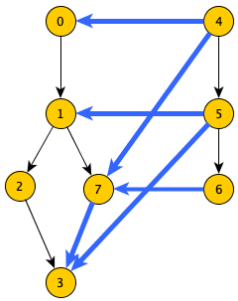
Elencare poi in ordine di visita gli archi visitati; dire infine quali sono gli archi di attraversamento, gli archi in avanti e gli archi all'indietro che si incontrano durante la visita.

G=



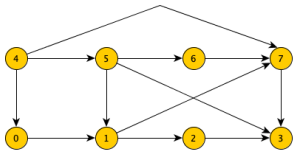
G=

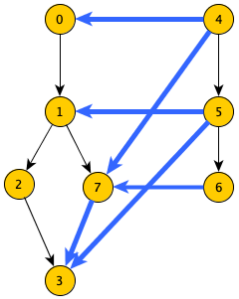




Gli archi di visita di visita incontrati sono nell'ordine:
$0 \to 1$, $1 \to 2$, $2 \to 3$, $1 \to 7$, $4 \to 5$, $5 \to 6$.
Gli archi blu sono archi di attraversamento
Non ci sono archi in avanti né all'indietro.

G=





L'ordine con cui termina la visita dfs dei nodi del grafo è: **3,2,7,1,0,6,5,4**

Il sort topologico prodotto dalla visita dfs è: **4,5,6,0,1,7,2,3**