ESERCIZIO 1. Abbiamo diversi tipi di stringhe binarie lunghe al più 2, concatenando stringhe siffatte possiamo ottenere stringhe binarie di lunghezza arbitraria.
Progettare un algoritmo che prende in input l'insieme I coi tipi di stringhe disponibili ed una stringa binaria S lunga n e, in tempo $O(n)$, conta i diversi modi con cui e' possibile ottenere S concatentando le stringhe dei tipi disponibili.

Ad esempio per $I = \{'0', '01', '10'\}$

- se $S =' 001010'$ la risposta dell'algoritmo è 3, infatti è possibile ottenere S nei seguenti modi:
  $'0' +' 0' +' 10' +' 10'$,
  $'0' +' 01' +' 01' +' 0'$
  $'0' +' 01' +' 0' +' 10'$

- se $S =' 0011010'$ la risposte dell'algoritmo deve essere 0 infatti non c'è modo di ottenere S concatenando stringhe del tipo specificato dall' insieme I.

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

Utilizziamo una tabella di dimensione n dove
$T[i] =$ il numero di modi di ottenere $S_0, \ldots S_i$ utilizzando le stringhe dei tipi disponibili.
La soluzione sarà in $T[n-1]$.

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

$$T[i] = \begin{cases} 1 & \text{se } i = 0 \text{ e } S_0 \in I \\ 0 & \text{se } i = 0 \\ 2 & \text{se } i = 1 \text{ e } S_0S_1,\ S_0 \text{ e } S_1 \in I \\ 1 & \text{se } i = 1 \text{ e } S_0S_1 \in I \text{ oppure } (S_0 \text{ e } S_1 \text{ in } I) \\ 0 & \text{se } i = 1 \\ T[i-1] + T[i-2] & \text{se } S_i \in I \text{ ed } S_{i-1}S_i \in I \\ T[i-1] & \text{se } S_i \in I \\ T[i-2] & \text{se } S_{i-1}S_i \in I \\ 0 & \text{altrimenti} \end{cases}$$

```
def es1(S, I):
    n=len(S)
    T=[0 for _ in range(n)]
    if S[0] in I: T[0]=1
    if len(S)==1: return T[0]
    if S[1] in I and T[0]==1 :T[1]+=1
    if S[0:2] in I :T[1]+=1
    for i in range(2,n):
           if S[i] in I : T[i]+=T[i-1]
           if S[i-1:i+1] in I: T[i]+=T[i-2]
    return T[n-1]

>>> I={'0','01','10'}
>>> S='001010'
>>> es1(S,I)
3

>>> I1={'0','1','01','10'}
>>> S1='0101'
>>> es1(S1,I1)
5
```

Complessità $\Theta(n)$

ESERCIZIO 2: Progettare un algoritmo che, dato l'intero n, stampa tutte le sequenze lunghe n sull'alfabeto ternario $\{a, b, c\}$ dove il simbolo a è sempre seguito da almeno due simboli b.
L'algoritmo proposto deve avere complessità $O(nS(n))$ dove $S(n)$ è il numero di sequenze da stampare.

Ad esempio per $n = 4$ l'algoritmo deve stampare, non necessariamente in quest'ordine, le seguenti 20 sottosequenze:

abbb abbc babb bbbb bbbc bbcb bbcc bcbb bcbc bccb

bccc cabb cbbb cbbc cbcb cbcc ccbb ccbc cccb cccc.

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

Algoritmo:

Eseguiremo una funzione di backtracking che enumera tutte le sottosequenze da stampare facendo ben attenzione a generare le sole sequenze parziali che porteranno ad almeno una sequenza da stampare.

Al passo ricorsivo i-esimo verifichiamo se in uno dei due passi precedenti è stata inserita una a, in questo caso dovremo necessariamente inserire una b. Se in nessuno dei due passi precedenti è stata inserita una a potenzialmente siamo liberi di inserire uno qualunque dei tre simboli a, b o c, bisogna solo fare attenzione che se si decide di inserire a ci siano a destra ancora due posti per potere inserire i due simboli b.

```
def es2(n):
        es2R(0,n,[])

def es2R(i,n,sol):
    if i==n:
        print(''.join(sol))
        return
    if (i>=1 and sol[i-1]=='a') or (i>=2 and sol[i-2]=='a'):
        sol.append('b')
        es2R(i+1,n,sol)
        sol.pop()
    else:
        simboli=['b','c']
        if i+2<n:simboli.append('a')
        for c in simboli:
            sol.append(c)
            es2R(i+1,n,sol)
            sol.pop()

>>> es2(4)
bbbb
bbbc
bbcb
bbcc
bcbb
bcbc
bccb
bccc
babb
cbbb
cbbc
cbcb
cbcc
ccbb
ccbc
cccb
cccc
cabb
abbb
abbc
```

- Nota che nell'albero di ricorsione prodotto dall'esecuzione di $es2R$ un nodo viene generato solo se porta ad una foglia da stampare.
- Possiamo quindi dire che la complessità dell'esecuzione di $es2R(0, n, [\,])$ 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 $es2R(0, n, [\,])$ è $O(nS(n))$.

Algoritmo alternativo:

possiamo fare a meno di inserire nella soluzione *sol* il simbolo -1 nei casi in cui decidiamo di non prendere il simbolo S_i, in questo modo avremo due vantaggi:

1. al termine tutto quello che è stato inserito in *sol* potrà essere direttamente stampato (senza dover filtrare i simboli -1)

2. non ci sarà bisogno di ricorrere al parametro *ultimo* per tener traccia dell'ultimo elemento inserito in *sol* (se $sol! = [\,]$ possiamo direttamente ottenere l'ultimo valore inserito con $sol[-1]$)

```
def es2b(S):
        es2bR(0,S,[])

def es2bR(i,S,sol):
    if i==len(S):
        print(sol)
        return
    if i!=len(S)-1 or sol!=[]:
        es2bR(i+1,S,sol)
    if sol==[] or S[i]>sol[-1]:
        sol.append(S[i])
        es2bR(i+1,S,sol)
        sol.pop()

>>> es2b(S)
[4]
[5]
[4]
[4, 5]
[2]
[2, 4]
[2, 5]
[2, 4]
[2, 4, 5]
[3]
[3, 4]
[3, 5]
[3, 4]
[3, 4, 5]
```

- Nota che nell'albero di ricorsione prodotto dall'esecuzione di $es2bR$ un nodo viene generato solo se porta ad una foglia da stampare.
- Possiamo quindi dire che la complessità dell'esecuzione di $es2bR(0, S, [\,])$ 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 $es2R(0, S, [\,])$ è $O(nD(n))$.
- Otteniamo quindi che il costo di $es2b(n)$ è $O(nD(n))$.