ESERCIZIO 1. Progettare un algoritmo che, data una sequenza decimale S lunga n, conta il numero di sottosequenze di S strettamente crescenti. L'algoritmo proposto deve avere complessità $O(n)$.

Ad esempio per $S = [3, 2, 4, 5, 4]$ l'algoritmo deve rispondere 14, infatti S presenta le seguenti sottosequenze crescenti:

$[3],\ [2],\ [4],\ [3, 4],\ [2, 4],\ [5],\ [3, 5],\ [2, 5],\ [4, 5],\ [3, 2, 5],\ [3, 4, 5],\ [4],\ [3, 4],\ [2, 4]$

Notate che possono esserci diverse sottosequenze contenenti gli stessi elementi (ad esempio la sottosequenza [4] viene contata due volte perché il 4 compare in S nella seconda e nella quarta posizione).
Motivare bene la correttezza e la complessità dell'algoritmo proposto.

Utilizziamo una tabella bidimensionale di dimensione $n \times 10$ dove
$T[i,j]$ = il numero di sottosequenze crescenti in $S_0, \ldots S_i$ che terminano con la cifra j.
La soluzione sarà in $\sum_{j=0}^{9}(T[n-1,j])$.

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

$$T[i,j] = \begin{cases} 1 & \text{se } i = 0 \text{ e } j = S_0 \\ 0 & \text{se } i = 0 \\ T[i-1][j] & \text{se } j \neq S_i \\ 1 + \sum_{k=0}^{j} T[i-1][k] & \text{altrimenti} \end{cases}$$

```
def es1(S):
    n=len(S)
    T=[[0 for _ in range(10)] for _ in range(n)]
    T[0][S[0]]=1
    for i in range(1,n):
        for j in range(10):
            T[i][j]=T[i-1][j]
            if S[i]==j:
                T[i][j]+=1
                for k in range(j):
                    T[i][j]+=T[i][k]
    return sum(T[n-1])

>>>S=[3,2,4,5,4]
>>>es1(S)
14
```

Complessità $\Theta(n)$

ESERCIZIO 2. Progettare un algoritmo che, data una sequenza decimale S lunga n, stampa le sottosequenze di S strettamente crescenti.
L'algoritmo proposto deve avere complessità $O(nD(n))$ dove $D(n)$ è il numero di sequenze da stampare.

Ad esempio per $S = [3, 2, 4, 5, 4]$ l'algoritmo deve stampare, non necessariamente in quest'ordine, le seguenti sottosequenze:

$[3]$, $[2]$, $[4]$, $[3, 4]$, $[2, 4]$, $[5]$, $[3, 5]$, $[2, 5]$, $[4, 5]$, $[3, 2, 5]$, $[3, 4, 5]$, $[4]$, $[3, 4]$, $[2, 4]$

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

Algoritmo:

- Eseguiremo una funzione di backtracking che enumera tutte le sottosequenze da stampare di S.
- Al passo ricorsivo i-esimo possiamo prendere o non prendere un elemento di S_i (nella soluzione inseriamo l'elemento preso S_i o il simbolo -1 ad indicare che l'elemento non é stato preso, al termine stamperemo della soluzione solo gli elementi diversi da -1). Per ottenere una complessità proporzionale alle $D(n)$ sottosequenze da stampare la a funzione di backtraking controlla che la scelta di prendere (o non prendere) l'elemento S_i venga fatta solo se la soluzione parziale che si ottiene porta ad un elemento da stampare. A questo proposito utilizziamo due semplici funzioni di taglio:
  - **prendiamo l'elemento** S_i solo se questo è il primo elemento che inseriamo nella soluzione o se l'ultimo elemento inserito nella soluzione ha un valore inferiore ad S_i
  - **lasciamo l'elemento** S_i solo se questa scelta non produce la soluzione finale in cui non è stato preso nulla.

```
def es2(S):
        es2R(0,S,[],-1)

def es2R(i,S,sol,ultimo):
    if i==len(S):
        sol1=[x for x in sol if x!=-1]
        print(sol1)
        return
    if i!=len(S)-1 or ultimo!=-1:
        sol.append(-1)
        es2R(i+1,S,sol,ultimo)
        sol.pop()
    if S[i]>ultimo:
        sol.append(S[i])
        es2R(i+1,S,sol,S[i])
        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 $es2R$ un nodo viene generato solo se porta ad una foglia da stampare.
- Possiamo quindi dire che la complessità dell'esecuzione di $es2R(0, S, [\,], -1)$ 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 $es2R(0, S, [\,], -1)$ è $O(nD(n))$.
- Otteniamo quindi che il costo di $es2(n)$ è $O(nD(n))$.

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 $O(nD(n))$.