

Corso di laurea in Informatica
Progettazione d'algoritmi
Didattica blended

Tecnica Programmazione Dinamica 3

Angelo Monti



Dato un intero n bisogna contare le stringhe binarie lunghe n in cui non compaiono 2 zeri consecutivi.

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiamo il contenuto delle celle come segue:
 $T[i]$ = numero di stringhe binarie lunghe i dove non compaiono 2 zeri consecutivi.
- Una volta riempita la tabella la soluzione al nostro problema la troveremo nella locazione $T[n]$.

ESERCIZIO 1

Dato un intero n vogliamo contare quante sono le stringhe binarie lunghe n in cui non compaiono 2 zeri consecutivi.
L'algoritmo proposto deve avere complessità $O(n)$.

Ad esempio:

- per $n = 1$ sono 2 (0, 1)
- per $n = 3$ sono 5 (010, 011, 101, 110, 111)

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiamo il contenuto delle celle come segue:
 $T[i]$ = numero di stringhe binarie lunghe i dove non compaiono 2 zeri consecutivi.

- Resta da definire la regola ricorsiva con cui calcolare i valori $T[i]$ nella tabella.

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

la ricorrenza viene fuori dal seguente ragionamento:

- il conteggio può essere visto come la somma delle stringhe da contare lunghe i che terminano con 1 e le stringhe da contare lunghe i che terminano con 0:
 1. le stringhe da contare lunghe i che terminano con 1 si ottengono accodando la cifra 1 a quelle da contare lunghe $i-1$. Queste stringhe sono $T[i-1]$
 2. le stringhe da contare lunghe i che terminano con 0 devono avere al penultimo posto la cifra 0, si ottengono dunque accodando 10 alle stringhe da contare lunghe $i-2$. Queste stringhe sono dunque $T[i-2]$

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

Implementazione:

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

Complessità $\Theta(n)$

```
>>> es1(3)
5
```

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

ESERCIZIO esame:

Dato un intero n vogliamo contare quante sono le stringhe binarie lunghe n in cui non compaiono 3 zeri consecutivi.

Ad esempio

- per $n = 1$ sono 2 (0,1)
- per $n = 4$ sono 13 (0100,0010,0011,0101,0110,1010,1001,1100,0111,1011,1101,1110,1111)

L'algoritmo proposto deve avere complessità $O(n)$.

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

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

Dato un intero n bisogna contare le stringhe binarie lunghe n in cui non compaiono 3 zeri consecutivi.

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiamo il contenuto delle celle come segue:
 $T[i]$ = numero di stringhe binarie lunghe i in cui non compaiono 3 zeri consecutivi.
- Una volta riempita la tabella la soluzione al nostro problema la troveremo nella locazione $T[n]$.

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiamo il contenuto delle celle come segue:
 $T[i]$ = numero di stringhe binarie lunghe i in cui non compaiono 3 zeri consecutivi.

- Resta da definire la regola ricorsiva con cui calcolare i valori $T[i]$ nella tabella.

$$T[i] = \begin{cases} 1 & \text{se } i = 0 \\ 2 & \text{se } i = 1 \\ 4 & \text{se } i = 2 \\ T[i-1] + T[i-2] + T[i-3] & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- possiamo partizionare tutte le stringhe da contare in tre differenti tipologie: quelle che terminano con 1, quelle che terminano con 10 e quelle che terminano con 100.
 1. le stringhe da contare lunghe i che terminano con 1 si ottengono accodando la cifra 1 a quelle da contare lunghe $i-1$. Queste stringhe sono $T[i-1]$
 2. le stringhe da contare lunghe i che terminano con 10 si ottengono accodando le 2 cifre 10 a quelle da contare lunghe $i-2$. Queste stringhe sono $T[i-2]$
 3. le stringhe da contare lunghe i che terminano con 100 si ottengono accodando le 3 cifre 100 a quelle da contare lunghe $i-3$. Queste stringhe sono $T[i-3]$

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

$$T[i] = \begin{cases} 1 & \text{se } i = 0 \\ 2 & \text{se } i = 1 \\ 4 & \text{se } i = 2 \\ T[i-1] + T[i-2] + T[i-3] & \text{altrimenti} \end{cases}$$

Implementazione:

```
def es2(n):
    T = [0 for _ in range(n + 1)]
    T[0], T[1], T[2] = 1, 2, 4
    for i in range(3, n + 1):
        T[i] = T[i-1] + T[i-2] + T[i-3]
    return T[n]

>>> es2(4)
13
```

La complessità è $\Theta(n)$

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

Dato un intero n vogliamo contare quanti modi diversi ci sono di sistemare n persone in un albergo che dispone di camere singole e camere doppie.

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiremo il contenuto delle celle come segue:

$T[i]$ = numero di modi in cui è possibile sistemare i persone nell'albergo.

- Una volta riempita la tabella la soluzione al nostro problema la troveremo nella locazione $T[n]$.

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

ESERCIZIO 2

Dato un intero n vogliamo contare quanti modi diversi ci sono di sistemare n persone in un albergo che dispone di camere singole e camere doppie.

L'algoritmo proposto deve avere complessità $O(n)$.

Ad esempio:

- per $n = 2$ sono 2:
 - $\{[1],[2]\}$
 - $\{[1,2]\}$
- per $n = 4$ sono 10:
 - $\{[1],[2],[3],[4]\}$,
 - $\{[1,2],[3],[4]\}$,
 - $\{[1,3],[2],[4]\}$,
 - $\{[1,4],[2],[3]\}$,
 - $\{[2,3],[1],[4]\}$,
 - $\{[2,4],[1],[3]\}$,
 - $\{[3,4],[1],[2]\}$,
 - $\{[1,2],[3,4]\}$,
 - $\{[1,3],[2,4]\}$,
 - $\{[1,4],[2,3]\}$

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiremo il contenuto delle celle come segue:

$T[i]$ = numero di modi in cui è possibile sistemare i persone nell'albergo.

- Resta da definire la regola ricorsiva con cui calcolare il valore $T[i]$ calcolati i valori $T[j]$ che precedono nella tabella.

$$T[i] = \begin{cases} 1 & \text{se } i = 0 \\ 1 & \text{se } i = 1 \\ T[i-1] + (i-1) \cdot T[i-2] & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- possiamo vedere le disposizioni delle i persone come la somma di due diverse tipologie: quelle in cui la persona i finisce in camera da sola e quelle in cui la persona i finisce in camera con una delle altre $i-1$ persone.
 1. le differenti disposizioni in cui i finisce da solo sono $T[i-1]$, sono infatti tutti i possibili modi di disporre le altre $i-1$ persone.
 2. le differenti disposizioni in cui i finisce in camera con una delle altre $i-1$ persone sono $(i-1) \cdot T[i-2]$ infatti ci sono $i-1$ modi di scegliere il compagno di stanza e una volta scelto il compagno ci sono $T[i-2]$ modi di disporre tutte le altre $i-2$ persone nelle varie camere.

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

$$T[i] = \begin{cases} 1 & \text{se } i = 0 \\ 1 & \text{se } i = 1 \\ T[i-1] + (i-1) \cdot T[i-2] & \text{altrimenti} \end{cases}$$

Implementazione:

```
def es3(n):
    T = [0 for _ in range(n + 1)]
    T[0]=T[1]=1
    for i in range(2, n + 1) :
        T [i] = T[i-1] + (i-1)* T[i-2]
    return T[n]

>>> es3(4)
10
>>> es3(5)
26
```

La complessità è $\Theta(n)$

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

Abbiamo un gruppo di n amici che debbono pernottare presso un hotel. L'hotel dispone di stanze da 1, 2 o 3 posti letto. Progettare un algoritmo che dato l'intero n calcola quante diverse allocazioni delle n persone sono possibili.

Utilizzeremo una tabella T di dimensione $n + 1$ e definiamo il contenuto delle celle come segue:

$T[i]$ = il numero di allocazioni per i persone, $1 \leq i \leq n$.

Una volta riempita la tabella la soluzione al nostro problema la troveremo in $T[n]$.

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

ESERCIZIO esame:

Abbiamo un gruppo di n amici che debbono pernottare presso un hotel. L'hotel dispone di stanze da 1, 2 e 3 posti letto.

Progettare un algoritmo che dato l'intero n calcola quante diverse allocazioni delle n persone sono possibili.

Ad esempio per $n = 4$ la risposta deve essere 14.

Infatti mettendo tra parentesi le persone che finiscono in una stessa stanza ecco di seguito le 14 possibilità:

```
{1}{2}{3}{4}
{1}{2,3,4}
{1}{2,3}{4}
{1}{2,4}{3}
{1}{3,4}{2}
{1,2}{3,4}
{1,2}{3}{4}
{1,3}{2,4}
{1,3}{2}{4}
{1,4}{2,3}
{1,4}{2}{3}
{1,2,3}{4}
{1,2,4}{3}
{1,3,4}{2}
```

L'algoritmo proposto deve avere complessità $O(n)$.

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

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiamo il contenuto delle celle come segue:

$T[i]$ = numero di modi in cui è possibile sistemare i persone nell'albergo.

Resta da definire la regola ricorsiva con cui calcolare i valori $T[i]$ della tabella

$$T[i] = \begin{cases} 1 & \text{se } i = 1 \\ 2 & \text{se } i = 2 \\ 5 & \text{se } i = 3 \\ T[i-1] + (i-1)T[i-2] + \frac{(i-1)(i-2)}{2}T[i-3] & \text{altrimenti} \end{cases}$$

La ricorrenza viene fuori dal seguente ragionamento:

Possiamo considerare tre casi in base alla sistemazione della prima persona:

- la prima persona finisce in camera da sola. Questo può accadere in un solo modo e poi ci sarà da considerare le possibili allocazioni delle altre $n - 1$ persone. Quindi le allocazioni di questo tipo sono $T[n - 1]$
- la prima persona finisce in una camera da due. Questo può accadere in $n - 1$ modi (in base al compagno che gli capita) e poi ci sarà da considerare le possibili allocazioni delle altre $n - 2$ persone. Quindi le allocazioni di questo tipo sono $(n - 1) \cdot T[n - 2]$
- la prima persona finisce in una camera da tre. Questo può accadere in $\binom{n-1}{2} = \frac{(n-1)(n-2)}{2}$ modi (in base ai due compagni che gli capitano) e poi ci sarà da allocare le altre $n - 3$ persone quindi le allocazioni di questo tipo sono $\frac{(n-1)(n-2)}{2} \cdot T[n - 3]$

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

$$T[i] = \begin{cases} 1 & \text{se } i = 1 \\ 2 & \text{se } i = 2 \\ 5 & \text{se } i = 3 \\ T[i-1] + (i-1)T[i-2] + \frac{(i-1)(i-2)}{2}T[i-3] & \text{altrimenti} \end{cases}$$

Implementazione:

```
def es1(n):
    m=max(4,n+1)
    T=[0 for _ in range(m)]
    T[1],T[2],T[3]=1,2,5
    for i in range(4,n+1):
        T[i]= T[i-1]+ (i-1)*T[i-2] + (i-1)*(i-2)*T[i-3]//2
    return T[n]

>>> es1(4)
14
>>> es1(5)
46
```

Complessità $\Theta(n)$

ESERCIZIO 3

Data una sequenza S di elementi una sottosequenza di S si ottiene eliminando zero o più elementi da S (Nota: non necessariamente le eliminazioni devono avvenire in testa o in coda alla sequenza)

Ad esempio:

data la sequenza 9, 3, 2, 4, 1, 5, 8, 6, 7, 2, una sua sottosequenza è 3, 1, 5, 6
(si ottiene eliminando dalla sequenza gli elementi indicati in rosso: 9, 3, 2, 4, 1, 5, 8, 6, 7, 2)

NOTA: le sottosequenze possibili per una sequenza di n elementi sono $\Theta(2^n)$.

Il problema della sottosequenza crescente: Data una sequenza S di n interi, vogliamo trovare una sottosequenza crescente (vale a dire i cui elementi risultano ordinati in modo crescente) di lunghezza massima.

Ad esempio:

data la sequenza $S = 9, 3, 2, 4, 1, 5, 8, 6, 7, 2$, una sottosequenza crescente di lunghezza massima per S è 3, 4, 5, 6, 7.

(non è l'unica, un'altra possibile soluzione è 2, 4, 5, 6, 7).

Progettare un algoritmo basato sulla programmazione dinamica che data una lista contenente una sequenza S di n elementi, in tempo $O(n^2)$ restituisce una sottosequenza crescente di lunghezza massima di S .

- come è tipico della *programmazione dinamica* progettiamo un algoritmo in grado di trovare la lunghezza massima della sottosequenza di S (e solo in un secondo tempo modificheremo l'algoritmo in modo da ottenere anche la sottosequenza che ha quella lunghezza).

- Utilizzeremo una tabella monodimensionale di dimensioni n e definiamo il contenuto delle celle come segue:
 $T[i]$ = la lunghezza massima per una sottosequenza crescente che termina con l'elemento di S in posizione i .

- Poiché la sottosequenza di lunghezza massima da qualche parte dovrà pur terminare la soluzione al nostro problema sarà data da:

$$\max_{0 \leq i < n} (T[i])$$

- Utilizzeremo una tabella monodimensionale di dimensioni n e definiamo il contenuto delle celle come segue:
 $T[i]$ = la lunghezza massima per una sottosequenza crescente che termina con l'elemento di S in posizione i .

- Resta da definire la regola ricorsiva con cui calcolare i valori $T[i]$ nella tabella.

$$T[i] = \begin{cases} 1 & \text{se } S_j > S_i \text{ per } 0 \leq j < i \\ \max_{0 \leq j < i \text{ con } S_j < S_i} (T[j]) + 1 & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- se tutti gli elementi che precedono S_i sono maggiori di S_i allora l'unica sottosequenza crescente che termina in posizione i è data dal solo elemento S_i .
- in caso contrario la sottosequenza che termina con S_i avrà come prefisso una sottosequenza crescente che termina in una posizione j con $j < i$ e $S_j \leq S_i$ e tra tutti i possibili "agganci" prendo quello che produce la sottosequenza più lunga.

$$T[i] = \begin{cases} 1 & \text{se } S_j > S_i \text{ per } 0 \leq j < i \\ \max_{0 \leq j < i \text{ con } S_j < S_i} (T[j]) + 1 & \text{altrimenti} \end{cases}$$

Implementazione:

```
def SSM(lista):
    T=[0]*len(lista)
    for i in range(len(lista)):
        m=0
        for j in range(i):
            if lista[j]<lista[i]:
                m=max(m,T[j])
        T[i]=m+1
    return max(T)

>>> lista=[9,3,2,41,5,8,6,7]
>>> SSM(lista)
5
```

La complessità è $O(n^2)$

$$T[i] = \begin{cases} 1 & \text{se } S_j > S_i \text{ per } 0 \leq j < i \\ \max_{0 \leq j < i \text{ con } S_j < S_i} (T[j]) + 1 & \text{altrimenti} \end{cases}$$

una volta ottenuta la lunghezza $l = \max(T)$ della sottosequenza crescente più lunga e disponendo della tabella T possiamo ripercorrere all'indietro le scelte che hanno portato alla sottosequenza di lunghezza massima e da queste ricavare gli elementi della sottosequenza:

- scoprire la posizione j in lista in cui termina la sequenza più lunga
(per j deve aversi $T[j] == l$)
- a questo punto possiamo scoprire qual'è la posizione t in cui si trova il penultimo elemento della soluzione
(per t deve aversi $t < j$ e $lista[t] < lista[j]$ e $T[t] == l - 1$)
- ora possiamo scoprire qual'è la posizione t' in cui si trova il terzultimo elemento della soluzione
(per t' deve aversi $t' < t$ e $lista[t'] < lista[t]$ e $T[t'] == l - 2$)
- e così via per l passi

Implementazione:

```
def SSMI(Lista):
    T=[0]*len(Lista)
    for i in range(len(Lista)):
        m=0
        for j in range(i):
            if Lista[j]<Lista[i]:
                m=max(m,T[j])
        T[i]=m+1
    # in T la lunghezza della soluzione
    l=max(T)
    #ora calcolo la soluzione
    for i in range(1,len(T)):
        if T[i]==l: break
    # in j la posizione in cui termina la soluzione
    j=i
    sol=[]
    while True:
        sol.append(Lista[j])
        if l==1: break
        #cerco la posizione j in cui si
        #trova l'elemento precedente dalla soluzione
        l = l - 1
        t = j - 1
        while lista[t]>lista[j] or T[t]!=l: t=t-1
        j=t
    #in sol c'è la soluzione all'inverso
    sol.reverse()
    return len(sol),sol

>>> lista=[9,3,2,41,5,8,6,7]
>>> SSMI(lista)
(5, [2, 4, 5, 6, 7])
```

- il calcolo della tabella richiede $O(n^2)$
- il calcolo della lunghezza della soluzione richiede $O(n)$
- il calcolo della posizione j dell'ultimo elemento della sequenza richiede $O(n)$
- Il calcolo delle posizioni degli altri elementi della sequenza richiede $O(n)$ infatti abbiamo due *while* annidati, consideriamo il costo dei due *while* separatamente:
 - il primo *while* viene eseguito esattamente l volte ma $l \leq n$ quindi la complessità è $O(n)$
 - con il secondo *while* ad ogni esecuzione mi sposto verso sinistra di una posizione, se sono all'inizio in posizione j in totale le iterazioni di quel *while* non potranno mai superare j , ma $j < n$ quindi la complessità è $O(n)$

La complessità dell'implementazione è $O(n^2)$.

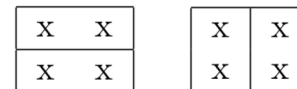
E' noto un diverso algoritmo che permette di risolvere il problema in tempo $O(n \log n)$.

ESERCIZIO d'esame

Dato l'intero n vogliamo contare il numero di differenti tassellamenti di una superficie di dimensione $n \times 2$ tramite tessere di domino di dimensione 1×2 .

Ad esempio:

- per $n = 1$ la risposta dell'algoritmo deve ovviamente essere 1
- per $n = 2$ la risposta deve essere 2 perché sono possibili i soli due seguenti tassellamenti:



L'algoritmo deve avere complessità $O(n)$

Dato l'intero n vogliamo contare il numero di differenti tassellamenti di una superficie di dimensione $n \times 2$ tramite tessere di domino di dimensione 1×2 .

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiremo il contenuto delle celle come segue:

$T[i]$ = numero di tassellamenti possibili per la superficie di dimensione $i \times 2$.

- Una volta riempita la tabella la soluzione al nostro problema la troveremo nella locazione $T[n]$.

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiremo il contenuto delle celle come segue:

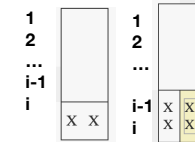
$T[i]$ = numero di tassellamenti possibili per la superficie di dimensione $i \times 2$.

- Resta da definire la regola ricorsiva con cui calcolare i valori $T[i]$ nella tabella.

$$T[i] = \begin{cases} 1 & \text{se } i = 1 \\ 2 & \text{se } i = 2 \\ T[i-1] + T[i-2] & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- possiamo vedere i tassellamenti della superficie di dimensione $i \times 2$ come la somma di due diverse tipologie: i tassellamenti in cui il tassello che copre l'ultima cella in basso a sinistra è posizionato in orizzontale e i tassellamenti in cui il tassello che copre l'ultima cella in basso a sinistra è posizionato in verticale:



- i tassellamenti in cui il tassello che copre l'ultima cella in basso a sinistra è posizionato in orizzontale sono $T[i-1]$, infatti il tassello occupa l' i -esima riga della superficie e resta da tassellare una superficie di dimensione $(i-1) \times 2$ (le prime $i-1$ righe della superficie) in tutti i modi possibili. Questi modi sono appunto $T[i-1]$.
- i tassellamenti in cui il tassello che copre l'ultima cella in basso a sinistra è posizionato in verticale sono $T[i-2]$ infatti in questo caso il vertice in basso a destra deve essere anch'esso coperto da un tassello in verticale, questi due tasselli lasciano da coprire una superficie di dimensione $(i-2) \times 2$ (la prime $i-2$ righe della superficie iniziale) in tutti i modi possibili. Questi modi sono appunto $T[i-2]$.

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

$$T[i] = \begin{cases} 1 & \text{se } i = 1 \\ 2 & \text{se } i = 2 \\ T[i-1] + T[i-2] & \text{altrimenti} \end{cases}$$

Implementazione:

```
def es4(n):
    T = [0 for _ in range(n + 1)]
    T[1], T[2] = 1, 2
    for i in range(3, n + 1):
        T[i] = T[i-1] + T[i-2]
    return T[n]
```

```
>>> es4(3)
3
```

Complessità $\Theta(n)$

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

ESERCIZIO d'esame

Dato un intero n vogliamo sapere quante sequenze di cifre decimali non decrescenti lunghe n è possibile trovare.

Ad esempio:

- per $n = 1$ la risposta dell'algoritmo deve ovviamente essere 10
- per $n = 2$ la risposta deve essere 55.
Infatti alla cifra x al primo posto possono seguire $10-x$ cifre diverse. Quindi si ha

$$\sum_{x=0}^9 (10-x) = \sum_{i=1}^{10} i = \frac{10 \cdot 11}{2} = 55$$

Progettare un algoritmo che trova la risposta in tempo $O(n)$.

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

Dato un intero n vogliamo sapere quante sequenze di cifre decimali non decrescenti lunghe n è possibile trovare.

- Utilizzeremo una tabella bidimensionale di dimensioni $n \times 10$ e definiamo il contenuto delle celle come segue:

$T[i, j]$ = numero di sequenze decimali non decrescenti lunghe i che terminano con la cifra j .

- Una volta riempita la tabella la soluzione al nostro problema sarà data dalla somma degli elementi nell'ultima riga: $\sum_{j=0}^9 T[n, j]$.

- Utilizzeremo una tabella bidimensionale di dimensioni $n \times 10$ e definiamo il contenuto delle celle come segue:
 $T[i, j]$ = numero di sequenze decimali non decrescenti lunghe i che terminano con la cifra j .

- Resta da definire la regola ricorsiva con cui calcolare i valori $T[i, j]$ della tabella

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

la ricorrenza viene fuori dal seguente ragionamento:

- esiste un'unica sequenza lunga 1 che termina con la cifra j , la cifra j ed è ovviamente non decrescente.
- la sequenza lunga i non decrescente che termina con j è composta da una sequenza non decrescente lunga $i-1$ cui si accoda la cifra j . Perché la sequenza risultante sia non decrescente la sequenza non decrescente lunga $i-1$ deve terminare con una qualunque cifra k non superiore a j (vale a dire: deve essere una della sequenza contata con $T[i-1, k]$)

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

Implementazione:

```
def es4(n):
    T=[[0 for _ in range(10)] for _ in range(n+1)]
    for j in range(10): T[1][j]=1
    for i in range(1,n+1):
        for j in range(0,10):
            for k in range(j+1):
                T[i][j]+=T[i-1][k]
    return sum(T[n])

>>> es4(1)
10
>>> es4(2)
55
>>> es4(3)
220
```

- NOTA CHE:**
- inizializzare la tabella costa $\Theta(n)$ (nota che la tabella ha $n+1$ righe ma un numero costante di colonne: 10)
 - dei 3 *for* annidati il primo viene iterato esattamente n volte, il secondo viene ogni volta iterato esattamente 10 volte ed il terzo viene iterato ogni volta al più 10 volte. Questo significa che il costo totale dei 3 *for* è $\Theta(n)$.

Complessità $\Theta(n)$

ESEMPIO

Il problema della massima sottosequenza comune. Date due sequenze di simboli X e Y , vogliamo trovare una sottosequenza comune X e Y che abbia lunghezza massima.

Progettate un algoritmo che risolva il problema in tempo $O(n \cdot m)$ dove n sono i simboli in X ed m quelli in Y .

Ad esempio:

per $X = \text{ABCBDAB}$ e $Y = \text{BDCABA}$

- una delle possibili sottosequenze comuni di lunghezza 3:

x = A B C B D A B
y = B D C A B A

- ci sono due diverse sottosequenze comuni di lunghezza massima B,C,A,B e B,C,B,A:

x = A B C B D A B
y = B D C A B A

x = A B C B D A B
y = B D C A B A

Una terza stringa di lunghezza 4 è B D A B

Concentriamoci preliminarmente sul calcolare il valore della soluzione ottima:

Sia $X = x_1, x_2, \dots, x_n$ e $Y = y_1, y_2, \dots, y_m$.

Definiamo una tabella bidimensionale T di dimensioni $(n + 1) \times (m + 1)$ dove:

• $T[i, j]$ = massima lunghezza possibile per sottosequenze comuni al prefisso di X di lunghezza i ed al prefisso di Y di lunghezza j .

$T[i, j]$ con $i, j \geq 1$ conterrà dunque la lunghezza massima per sottosequenze comuni alle due stringhe

$X' = x_1 x_2 \dots x_i$

$Y' = y_1 y_2 \dots y_j$

La soluzione al nostro problema sarà nella cella $T[n, m]$

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

$T[i, j]$ = massima lunghezza possibile per sottosequenze comuni al prefisso di X di lunghezza i ed al prefisso di Y di lunghezza j .

resta ora da definire la formula ricorsiva che permette di calcolare la cella $T[i, j]$ a partire dalle celle precedenti già calcolate:

$$T[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ T[i-1][j-1] + 1 & \text{se } X[i-1] = Y[j-1] \\ \max\{T[i-1][j], T[i][j-1]\} & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- se non possiamo usare nessun simbolo di una delle due sequenze è ovvio che la sottosequenza comune avrà lunghezza 0.
- se le due sequenze terminano con lo stesso simbolo allora la sottosequenza comune più lunga terminerà con quel simbolo e per il resto sarà la sequenza più lunga tra x_1, x_2, \dots, x_{i-1} e y_1, y_2, \dots, y_{j-1} , vale a dire $T[i, j] = T[i-1, j-1] + 1$
- se le due sequenze terminano con simboli diversi, diciamo a per X e b per Y , allora deve verificarsi uno dei seguenti tre casi:
 - la sottosequenza più lunga termina con a . In questo caso l'ultimo simbolo di Y non influisce e si ha: $T[i, j] = T[i, j-1]$
 - la sottosequenza più lunga termina con b . In questo caso l'ultimo simbolo di X non influisce e si ha: $T[i, j] = T[i-1, j]$
 - la sottosequenza più lunga termina con un simbolo c diverso da a e b . In questo caso l'ultimo simbolo di X e l'ultimo simbolo di Y non influiscono e si ha: $T[i, j] = T[i-1, j-1]$

e dovendo prendere il caso che produce la sottosequenza più lunga deve aversi:

$$T[i, j] = \max\{T[i, j-1], T[i-1, j], T[i-1, j-1]\} = \max\{T[i, j-1], T[i-1, j]\}$$

dove l'ultimo passaggio segue dal fatto che $T[i-1, j-1] \leq T[i, j-1]$ e $T[i-1, j-1] \leq T[i-1, j]$.

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

$T[i, j]$ = massima lunghezza possibile per sottosequenze comuni al prefisso di X di lunghezza i ed al prefisso di Y di lunghezza j .

resta ora da definire la formula ricorsiva che permette di calcolare la cella $T[i, j]$ a partire dalle celle precedenti già calcolate:

$$T[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ T[i-1][j-1] + 1 & \text{se } X[i-1] = Y[j-1] \\ \max\{T[i-1][j], T[i][j-1]\} & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- se non possiamo usare nessun simbolo di una delle due sequenze è ovvio che la sottosequenza comune avrà lunghezza 0
- se le due sequenze terminano con lo stesso simbolo allora la sottosequenza comune più lunga terminerà con quel simbolo e per il resto sarà la sequenza più lunga tra x_1, x_2, \dots, x_{i-1} e y_1, y_2, \dots, y_{j-1} , vale a dire $T[i, j] = T[i-1, j-1] + 1$.
- se le due sequenze terminano con simboli diversi, diciamo a per X e b per Y , allora deve verificarsi uno dei seguenti tre casi:
 - la sottosequenza più lunga termina con a . In questo caso l'ultimo simbolo di Y non influisce e si ha: $T[i, j] = T[i, j-1]$
 - la sottosequenza più lunga termina con b . In questo caso l'ultimo simbolo di X non influisce e si ha: $T[i, j] = T[i-1, j]$
 - la sottosequenza più lunga termina con un simbolo c diverso da a e b . In questo caso l'ultimo simbolo di X e l'ultimo simbolo di Y non influiscono e si ha: $T[i, j] = T[i-1, j-1]$

e dovendo prendere il caso che produce la sottosequenza più lunga deve aversi:

$$T[i, j] = \max\{T[i, j-1], T[i-1, j], T[i-1, j-1]\} = \max\{T[i, j-1], T[i-1, j]\}$$

dove l'ultimo passaggio segue dal fatto che $T[i-1, j-1] \leq T[i, j-1]$ e $T[i-1, j-1] \leq T[i-1, j]$.

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

$$T[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ T[i-1][j-1] + 1 & \text{se } X[i-1] = Y[j-1] \\ \max\{T[i-1][j], T[i][j-1]\} & \text{altrimenti} \end{cases}$$

Implementazione:

```
def lcs(X, Y):
    n, m = len(X), len(Y)
    T = [[0 for _ in range(m+1)] for _ in range(n+1)]
    for i in range(n+1):
        for j in range(m+1):
            if i==0 or j==0:
                T[i][j] = 0
            elif X[i-1] == Y[j-1]:
                T[i][j] = T[i-1][j-1] + 1
            else:
                T[i][j] = max(T[i-1][j], T[i][j-1])
    return T[n][m]

>>> X = 'ABCBDABO'
>>> Y = 'BDCABAO'
>>> lcs(X, Y)
5
```

Complessità $\Theta(n \cdot m)$

Corso di Progettazione di Algoritmi – Prof. Angelo Monti

Una volta costruita la tabella T , se si vuole ricostruire la sottosequenza comune, come al solito, possiamo, partendo dalla locazione $T[n, m]$ (con il valore della lunghezza massima) percorrere T a ritroso seguendo le scelte che hanno portato a calcolare la lunghezza massima.

$$T[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ T[i-1][j-1] + 1 & \text{se } X[i] = Y[j] \\ \max\{T[i-1][j], T[i][j-1]\} & \text{altrimenti} \end{cases}$$

Durante il percorso all'indietro sulla tabella T , per com'è fatta la ricorrenza, quando ci si trova sulla cella $T[i, j]$:

- se $X[i-1] = Y[j-1]$: il simbolo $X[i-1]$ appartiene alla sottosequenza di lunghezza ottima, e ci si sposta sulla cella $T[i-1, j-1]$
- se $X[i-1] \neq Y[j-1]$: ci si sposta in $T[i-1, j]$ se $T[i-1, j] = T[i, j]$, in caso contrario ci si sposta in $T[i, j-1]$.

quando si giunge ad una cella $T[i, j]$ con $i = 0$ o $j = 0$ l'algoritmo termina.

Nell'algoritmo proposto verrà restituita una terna:

- la prima componente è una lista con i simboli della sottosequenza più lunga,
- la seconda componente è una lista con le posizioni della sottosequenza in X ,
- la terza componente è una lista con le posizioni della sottosequenza in Y .

Ad esempio:
per $X = \text{ABCBDABO}$ e $Y = \text{BDCABAO}$ dove la soluzione ottima ha lunghezza 5 verrà restituita la terna:

$(['B', 'C', 'B', 'A', 'O'], [1, 2, 3, 5, 7], [0, 2, 4, 5, 6])$

Implementazione:

```
def sol_lcs(X, Y):
    n, m = len(X), len(Y)
    T = [[0 for _ in range(n+1)] for _ in range(m+1)]
    for i in range(n+1):
        for j in range(m+1):
            if i==0 or j==0:
                T[i][j]=0
            elif X[i-1]==Y[j-1]:
                T[i][j]=T[i-1][j-1]+1
            else:
                T[i][j]=max(T[i-1][j], T[i][j-1])
    #dopo aver costruito la tabella ricaviamo la soluzione:
    i, j, sol, posX, posY = len(X), len(Y), [], [], []
    while i>0 and j>0:
        if X[i-1]==Y[j-1]:
            sol.append(X[i-1])
            posX.append(i-1)
            posY.append(j-1)
            i-=1
            j-=1
        else:
            if T[i][j]==T[i-1][j]: i-=1
            else: j-=1
    sol.reverse()
    posX.reverse()
    posY.reverse()
    return sol, posX, posY

>>> X = 'ABCBDABO'
>>> Y = 'BDCABAO'
>>> sol_lcs(X, Y)
(['B', 'C', 'B', 'A', 'O'], [1, 2, 3, 5, 7], [0, 2, 4, 5, 6])
```

- l'inizializzazione della tabella costa $\Theta(n \cdot m)$
- il calcolo dei valori della tabella costa $\Theta(n \cdot m)$
- L'esecuzione del *while* per il calcolo della soluzione costa tempo $O(n + m)$. Infatti, ad ogni passo del *while*, almeno uno degli indici i e j si decrementa.

La complessità dell'algoritmo è $\Theta(n \cdot m)$