Algoritmi e Strutture Dati (Mod. B)

Programmazione Dinamica
(Parte II)

③ *Calcolo del valore di una soluzione ottima*

Il terzo passo consiste nel *calcolare* il *valore della soluzione ottima* (alla parentesizzazione) *in termini* delle *soluzioni ottime* (alle parentesizzazioni) *dei sottoproblemi*.

③ *Calcolo del valore di una soluzione ottima*

A partire dall'equazione sotto, sarebbe facile definire un *algoritmo ricorsivo* che calcola il *costo minimo* *C(1,n)* di $A_{1\ldots n}$

Purtroppo vedremo che tale approccio porta ad un *algoritmo di costo esponenziale*, non migliore dell'*enumerazione esaustiva*.

$$C(l,r)=\begin{cases}0 & \text{se } l \geq r\\ \min\limits_{l\leq k<r}\{C(l,k)+C3(k+1,r)+c_{l-1}c_kc_r\} & \text{se } l<r\end{cases}$$

L \ R	1	2	3	4	5	6
1	0					
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

$C(l,r) = 0$ se $l = r$,

$C(l,r) = min_{1 \le k < r}\{ C(l,k) + C(k+1,r) + c_{l-1}c_k c_r \}$ altrimenti

$$C(1,2) = min_{1 \le k < 2}\{ C(1,k) + C(k+1,2) + c_1 c_k c_2 \}$$
$$= C(1,1) + C(2,2) + c_0 c_1 c_2$$

L \ R	1	2	3	4	5	6
1	0					
2	-					
3	-	-	0			
4	-	-	-			
5	-	-	-	-	0	
6	-	-	-	-	-	0

$C(l,r) = 0$ se $l = r$,

$C(l,r) = \min_{1 \le k < r}\{ C(l,k) + C(k+1,r) + c_{l-1}c_k c_r \}$ altrimenti

$$C(2,4) = \min_{2 \le k < 4}\{ C(2,k) + C(k+1,4) + c_1 c_k c_4 \}$$
$$= \min \{ C(2,2) + C(3,4) + c_1 c_2 c_4 ,$$
$$C(2,3) + C(4,4) + c_1 c_3 c_4 \}$$

L \ R	1	2	3	4	5	6
1	0					
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

$C(l,r) = 0$ se $l = r$,

$C(l,r) = min_{1 \le k<r} \{ C(l,k) + C(k+1,r) + c_{l-1} c_k c_r \}$ altrimenti

$$C(2,5) = min_{2 \le k<5} \{ C(2,k) + C(k+1,5) + c_1 c_k c_5 \}$$
$$= min \{ C(2,2) + C(3,5) + c_1 c_2 c_5 ,$$
$$C(2,3) + C(4,5) + c_1 c_3 c_5 ,$$
$$C(2,4) + C(5,5) + c_1 c_4 c_5 \}$$

L \ R	1	2	3	4	5	6
1						
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-		
6	-	-	-	-	-	0

$C(l,r) = 0$ se $l = r$,

$C(l,r) = min_{1 \le k<r}\{ C(l,k) + C(k+1,r) + c_{l-1}c_k c_r \}$ altrimenti

$$
\begin{aligned}
C(1,5) &= min_{1 \le k<5}\{ C(1,k) + C(k+1,5) + c_0 c_k c_5 \} \\
&= min \{ C(1,1) + C(2,5) + c_0 c_1 c_5 , \\
&\qquad C(1,2) + C(3,5) + c_0 c_2 c_5 , \\
&\qquad C(1,3) + C(4,5) + c_0 c_3 c_5 , \\
&\qquad C(1,4) + C(5,5) + c_0 c_4 c_5 \}
\end{aligned}
$$

L \ R	1	2	3	4	5	6
1	0					
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

$C(l,r) = 0$ se $l = r$,

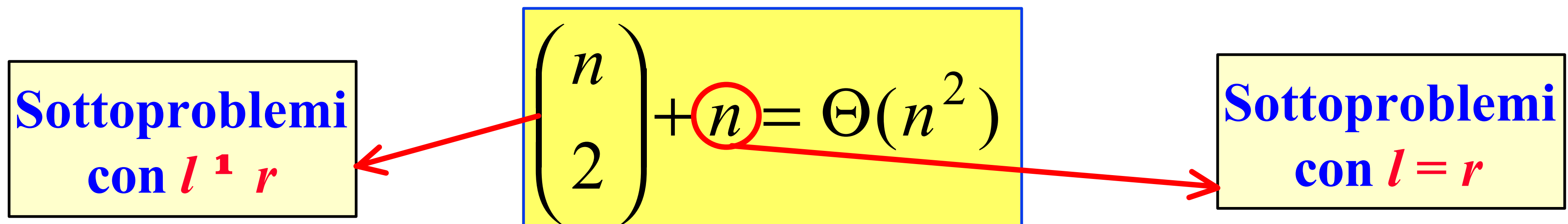
$C(l,r) = min_{1 \le k < r}\{ C(l,k) + C(k+1,r) + c_{l-1}c_k c_r \}$ altrimenti

$$C(1,6) = min_{1 \le k < 6}\{ C(1,k) + C(k+1,6) + c_0 c_k c_6 \}$$

$$= min \{ C(1,1) + C(2,6) + c_0 c_1 c_6 ,$$
$$C(1,2) + C(3,6) + c_0 c_2 c_6 ,$$
$$C(1,3) + C(4,6) + c_0 c_3 c_6 ,$$
$$C(1,4) + C(5,6) + c_0 c_4 c_6 ,$$
$$C(1,5) + C(6,6) + c_0 c_5 c_6 \}$$

③ *Calcolo del valore di una soluzione ottima*

L'importante osservazione è che ci sono un *numero limitato di sottoproblemi*, *uno per ogni scelta di l* e r (con $1 \le l \le r \le n$), quindi in totale



corrispondenti alle celle riempite di una matrice $n \times n$.

Utilizzando quindi una matrice $C[n \times n]$ possiamo risolvere facilmente il problema *in tempo polinomiale*.

③ *Calcolo del valore di una soluzione ottima*

Utilizzando quindi una matrice *$C[n,n]$* possiamo risolvere facilmente il problema *in tempo polinomiale*.

Ogni *sottoproblema* è risolvibile *utilizzando solo le soluzioni di sottoproblemi* ricorrenti più volte che possono essere *calcolati prima e memorizzati nella matrice $C[n,n]$*.

In tal modo *non si calcola mai più di una volta la soluzione ad un sottoproblema*.

Questa è l'*idea chiave della programmazione dinamica*.

③ *Calcolo del valore di una soluzione ottima*

L'algoritmo *MCO* (*M*atrix-*C*hain-*O*rder)

- **prende in *ingresso* un array *c*[] contenente le dimensioni delle matrici (*c*[0] è il numero di righe della prima matrice, *c*[i] è il numero di colonne della matrice A_i**
- **utilizza (e *ritorna*) due matrici $n \times n$ ausiliarie:**
  - ***C*[*l*,*r*] che contiene i costi minimi dei sottoproblemi $A_{l...r}$**
  - ***S*[*l*,*r*] che contiene i valori delle scelte ottime per i *k*, cioè quelle che minimizzano il costo dei sottoproblemi generati.**

③ *Calcolo del valore di una soluzione ottima*

```
MCO(c[]: array di interi)
   n = lunghezza[c] - 1
   for i = 1 to n
      do C[i,i] = 0
   for l = 2 to n
      do for i = 1 to n - l + 1
           do j = i + l - 1
              C[i,j] = ∞
              for k = i to j - 1
                 do q = C[i,k] + C[k+1,j] + c[i-1] c[k] c[j]
                    if q < C[i,j]
                       then C[i,j] = q
                            S[i,j] = k
   return
```

Il costo è zero nel caso di una sola matrice

l varia sulle diagonali sopra quella principale

i e *j* assumono i valori delle celle nella diagonale *l*

Calcola tutti i possibili valori e conserva solo il più piccolo

$C[\,]$

L \ R	1	2	3	4	5	6
1	0	224	176	218	276	350
2	-	0	64	112	174	250
3	-	-	0	24	70	138
4	-	-	-	0	30	90
5	-	-	-	-	0	90
6	-	-	-	-	-	0

i	c_i
0	7
1	8
2	4
3	2
4	3
5	5
6	6

$$
\begin{aligned}
C(1,4) &= \min_{1 \le k \le 3}\{\, C(1,k) + C(k+1,4) + c_0 c_k c_4 \,\} \\
&= \min \{\; C(1,1) + C(2,4) + c_0 c_1 c_4 \,, \\
&\qquad\quad C(1,2) + C(3,4) + c_0 c_2 c_4 \,, \\
&\qquad\quad C(1,3) + C(4,4) + c_0 c_3 c_4 \;\} \\
&= \min \{\; 0 + 112 + 7 * 8 * 3, \\
&\qquad\quad 224 + 24 + 7 * 4 * 3, \\
&\qquad\quad 176 + 0 + 7 * 2 * 3 \;\} \\
&= \min \{\, 280, 332, 218 \,\} = 218
\end{aligned}
$$

$Tempo = O(n^3)$

$S[\,]$

L \ R	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

i	c_i
0	7
1	8
2	4
3	2
4	3
5	5
6	6

$$C(1,4) = min_{1 \le k \le 3}\{\, C(1,k) + C(k+1,4) + c_0 c_k c_4 \,\}$$

$$= min\{\; C(1,1) + C(2,4) + c_0 c_1 c_4,\; C(1,2) + C(3,4) + c_0 c_2 c_4,\; C(1,3) + C(4,4) + c_0 c_3 c_4 \;\}$$

$$= min\{\; 0 + 112 + 7 * 8 * 3,\; 224 + 24 + 7 * 4 * 3,\; 176 + 0 + 7 * 2 * 3 \;\}$$

$$= min\{\, 280, 332, 218 \,\} = 218$$

③ *Calcolo del valore di una soluzione ottima*

- **L'algoritmo *MCO* calcola i costi minimi riempiendo le matrici partendo dalla diagonale principale (tutti 0) e proseguendo in ordine con le diagonali successive alla principale.**
- **Ad ogni passo il valore di $C[i,j]$ dipende solo dai valori, calcolati precedentemente, delle celle $C[i,k]$ e $C[k+1,j]$, che infatti stanno *tutti nelle diagonali sottostanti* a quella di $C[i,j]$**
- **Il valore di ogni cella $C[i,j]$ viene calcolato una sola volta durante l'elaborazione della diagonale (*indicata dall'indice l*) in cui compare.**

④ *Costruzione della soluzione ottima*

Il quarto passo consiste nel *costruire* il *la soluzione ottima* (al prodotto delle *n* matrici), cioè quello il cui *costo ottimo* è stato calcolato al *terzo passo*.

- **Infatti terzo passo calcola il *costo* (e gli indici *k* ottimali delle scelte ad ogni livello) della *soluzione ottima*, ma *non* calcola il *prodotto corrispondente*.**

④ *Costruzione della soluzione ottima*

- **Possiamo definire un *algoritmo* che costruisce la soluzione a partire dalla informazione calcolata da *MCO*.**
- **La matrice *S*[ ] ci permette di determinare il modo migliore di moltiplicare le matrici.**
- ***S*[*i,j*] contiene infatti il valore di *k* su cui dobbiamo spezzare il prodotto $A_{i \ldots j}$ ...**
- **… ci dice cioè che per calcolare $A_{i \ldots j}$ dobbiamo prima calcolare $A_{i \ldots k}$ e $A_{k+1 \ldots j}$ e poi moltiplicare I due risultati tra di loro.**
- **Ma questo è un processo facilmente realizzabile tramite un *algoritmo ricorsivo* che ricalca proprio la sottostruttura ricorsiva definita al *passo uno***

④ Costruzione della soluzione ottima

```
MCM(A[]:array; S[]:matrice; i,j: intero)
  if j>i
     then k=S[i,j]
          X=MCM(A,S,i,k)
          Y=MCM(A,S,k+1,j)
          return Prod-Matrici(X,Y)
      else return A[i]
```

- *A*[] un array di lunghezza *n* che contiene le matrici $[A_1, A_2, \ldots, A_n]$
- *S*[] la matrice $n \times n$ che contiene il valore di *k* ottimo per ogni coppia di indici *(l,r)*

④ Costruzione della soluzione ottima

$$A_{1\ldots6} = A_{1\ldots k} \times A_{k+1\ldots6}$$
$$= A_{1\ldots3} \times A_{4\ldots6}$$
$$A_{1\ldots3} = A_{1\ldots k} \times A_{k+1\ldots3}$$
$$= A_1 \times A_{2\ldots3}$$
$$A_{4\ldots6} = A_{4\ldots k} \times A_{k+1\ldots6}$$
$$= A_{4..5} \times A_6$$
$$A_{2\ldots3} = A_{2\ldots k} \times A_{k+1\ldots3}$$
$$= A_2 \times A_3$$
$$A_{4\ldots5} = A_{4\ldots k} \times A_{k+1\ldots5}$$
$$= A_4 \times A_5$$

S[ ]

L \ R	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

$$A_{1\ldots6} = ((A_1 \times (A_2 \times A_3)) \times ((A_4 \times A_5) \times A_6))$$

Versione ricorsiva di MCO

$$C(l,r) = \begin{cases} 0 & \text{se } l \geq r \\ \min_{l \leq k < r} \{C(l,k) + C(k+1,r) + c_{l-1}c_k c_r\} & \text{se } l < r \end{cases}$$

```
R-MCO(c[]: array di interi; i,j:intero)
 if i = j
   then return 0
   else C[i,j] = ∞
        for k = i to j-1
          do Costo = R-MCO(c,i,k) + R-MCO(c,k+1,j) +
                + c[i-1]c[k]c[j]
             if Costo < C[i,j] then C[i,j] = Costo
        return C[i,j]
```

Versione ricorsiva di MCO

- **Definiamo l'*equazione di ricorrenza* per *C*[1,*n*] dell'algoritmo ricorsivo appena visto:**

$$T(n) \geq \begin{cases} 1 & \text{se } n = 1 \\ 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) & \text{se } n > 1 \end{cases}$$

assumendo che i *due* `if` abbiano costo almeno unitario (comunque costante).

Versione ricorsiva di MCO

- **Definiamo l'*equazione di ricorrenza* per $C[1,n]$ dell'algoritmo ricorsivo appena visto:**

$$T(n) \geq \begin{cases} 1 & \text{se } n = 1 \\ 1 + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + 1 \right) & \text{se } n > 1 \end{cases}$$

$$T(n) \geq 1 + n - 1 + \sum_{k=1}^{n-1} 2T(k) \quad \text{se } n > 1$$

$$T(n) \geq n + \sum_{k=1}^{n-1} 2T(k) \quad \text{se } n > 1$$

Versione ricorsiva di MCO

- Dimostriamo che l'equazione di ricorrenza ha soluzione almeno esponenziale $\Omega(2^n)$ [$T(n) \geq 2^{n-1}$] :

$$T(n) \geq n + \sum_{k=1}^{n-1} 2\,T(k) \quad \text{se } n > 1$$

- *Caso Base*: $T(1) \geq 1 = 2^0 = 2^{1-1}$
- *Caso Induttivo*:

$$T(n) \geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n$$

$$\geq 2 \sum_{i=0}^{n-2} 2^{i} + n$$

Versione ricorsiva di MCO

- **Dimostriamo che l'equazione di ricorrenza ha soluzione almeno esponenziale $\Omega(2^n)$ [$T(n) \geq 2^{n-1}$] :**

$$T(n) \geq n + \sum_{k=1}^{n-1} 2\,T(k) \quad \text{se } n > 1$$

- ***Caso Base*: $T(1) \geq 1 = 2^0$**
- ***Caso Induttivo*:**

$$T(n) \geq 2 \sum_{i=0}^{n-2} 2^i + n$$

$$\geq 2(2^{n-1} - 1) + n$$

$$\geq 2^n + n - 2 \geq 2^n \quad \text{per } n \geq 2$$

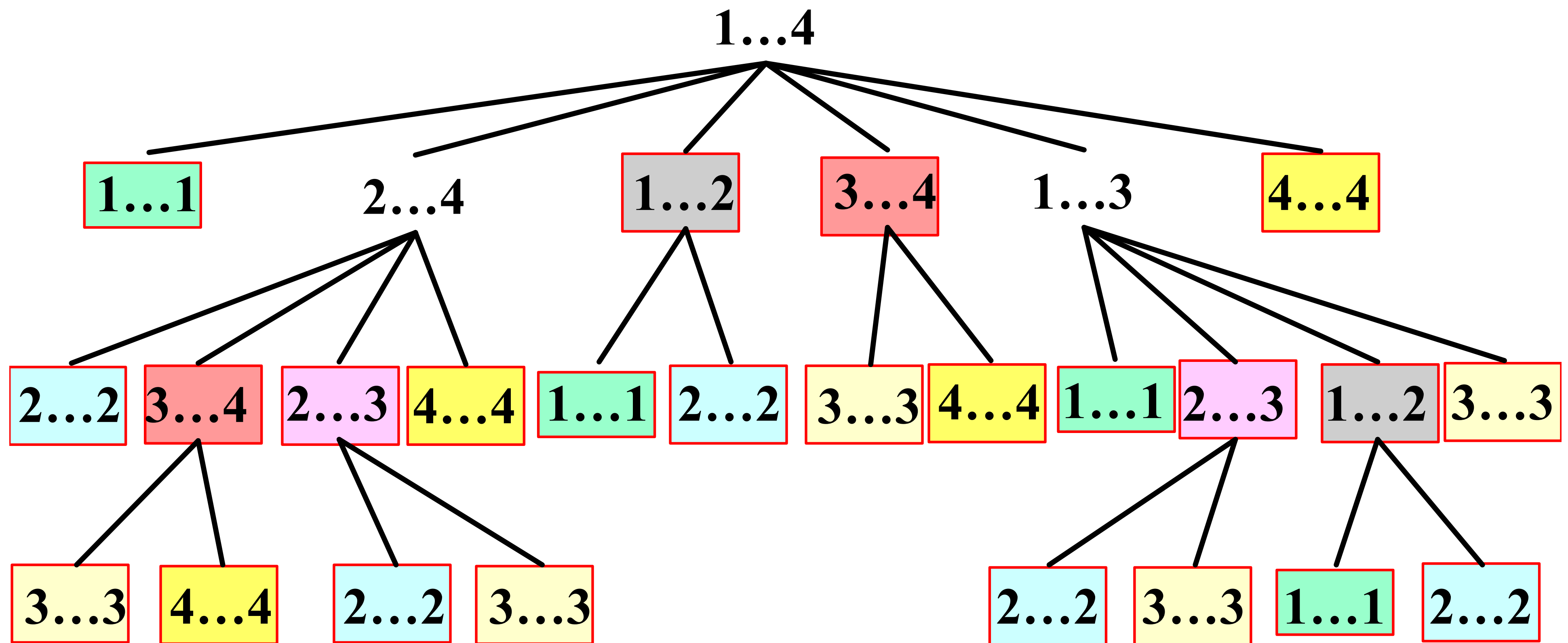
Versione ricorsiva di MCO

- **Quindi il tempo di esecuzione di *R-MCO* è almeno esponenziale nel numero di matrici da moltiplicare, cioè:**

$$T(n) = \Omega(2^n)$$

Da dove deriva l'inefficienza dell'algoritmo ricorsivo?

Versione ricorsiva di MCO



Ricorsione e Programmazione Dinamica

- **Ancora una volta, l'*inefficienza* della versione ricorsiva deriva dal fatto che i sottoproblemi *non* danno luogo a computazioni indipendenti.**
- **cioè, *molti sottoproblemi ricorrono più volte* e devono quindi essere *risolti ogni volta*.**
- **Questo fenomeno viene anche indicato con: *sovrapposizione dei sottoproblemi*.**

Programmazione Dinamica

- **La *Programmazione Dinamica* evita di ripetere la computazione per i sottoproblemi che si ripetono.**
- **col supporto di *memoria aggiuntiva* (tabella) risolve i sottoproblemi *al più una sola volta*.**

La *sovrapposizione di sottoproblemi* è un altro degli indicatori che la *Programmazione Dinamica* *potrebbe fornire una soluzione efficiente*.

Programmazione Dinamica

- **La *Programmazione Dinamica* sfrutta in maniera determinante *struttura della soluzione ottima*.**
- **Affinché la *Programmazione Dinamica* sia applicabile, *è necessario* che la soluzione ottima del problema esibisca la *proprietà della sottostruttura ottima***

La *sottostruttura ottima* è il secondo indicatore che la *Programmazione Dinamica* *potrebbe fornire una soluzione efficiente*.

Programmazione Dinamica

La *sottostruttura ottima* è il secondo indicatore che la *Programmazione Dinamica potrebbe fornire una soluzione efficiente*.

- **Prima di tentare di definire l'algoritmo, si deve però dimostrare la proprietà per il problema dato.**
- **Una tecnica standard per dimostrare la proprietà**
  - **si assume che esista una soluzione migliore per un sottoproblema arbitrario e**
  - **si dimostra che da tale assunzione segue una contrad-dizione con l'ipotesi di ottimalità**

Ricorsione con memorizzazione

- **É una variante della *Programmazione Dinamica*.**
- **L'idea è quella di fondere la *memorizzazione* in tabella tipico della *Progr. Dinamica* con l'approccio ricorsivo tipico del *Divede-et-Impera*.**
- **Quando un sottoproblema viene risolto per la prima volta, viene *memorizzato in una tabella*:**
  - **ogni volta che si deve *risolvere un sottoproblema*, si *controlla* nella *tabella* se è già stato risolto precedentemente**
  - **se lo è già stato, si usa il risultato della tabella**
  - **altrimenti lo si calcola e si memorizza il risultato**

Ricorsione con memorizzazione

- **Quando un sottoproblema viene risolto per la prima volta, viene *memorizzato in una tabella*:**
  - **ogni volta che si deve *risolvere un sottoproblema*, si *controlla* nella *tabella* se è già stato risolto precedentemente**
  - **se lo è già stato, si usa il risultato della tabella**
  - **altrimenti lo si calcola e si memorizza il risultato**
- **In tal modo, ogni sottoproblema viene calcolato una sola volta e memorizzato come nel caso precedente.**
- **Dal momento che abbiamo solo $\Theta(n^2)$ sottoproblemi, il tempo di esecuzione sarà identico.**

Versione ricorsiva con memorizzazione

```
Mem-MCO(c[]: array di intero)
 n=lunghezza[c]-1
 for i=1 to n
   do for j=i to n
        do C[i,j]=∞
 return Cerca-MC(c,1,n)

Cerca-MC(c[]: array of intero; i,j:intero)
  if C[i,j]<∞
   then return C[i,j]
  if i=j
   then C[i,j]=0
   else for k = i to j-1
         do Costo=Cerca-MC(c,i,k)+Cerca-MC(c,k+1,j)
                  +c[i-1]c[k]c[j]
            if Costo<C[i,j] then C[i,j]=Costo
  return C[i,j]
```

Confronto tra i due approcci

- ***Iterativo Bottom-up*** **:**
  - **quando *tutti i sottoproblemi devono essere risolti almeno una volta*, è più efficiente, non avendo il problema di gestire le chiamate ricorsive.**
  - **in alcuni casi, la regolarità degli accessi alla tabella può essere sfruttata per ridurre ulteriormente tempo e/o spazio (come per *Numeri di Fibonacci*)**

- ***Ricorsione con memorizzazione*****:**
  - **quando *alcuni sottoproblemi possono non essere risolti affatto*, può essere vantaggiosa perché risolve solo i sottoproblemi richiesti per ottenere la soluzione finale.**

Confronto con Divide-et-Impera

La soluzione dei problemi è costruita a partire dalle soluzione dei sottoproblemi.

- ***Divide-et-Impera*** **: Merge Sort**
  - **Ogni suddivisione del problema in sottoproblemi porta alla stessa soluzione**
  - **I sottoproblemi sono disgiunti**
  - **Il calcolo avviene in maniera "top-down"**

Confronto con Divide-et-Impera

La soluzione dei problemi è costruita a partire dalle soluzione dei sottoproblemi.

- ***Divide-et-Impera*** **: Merge Sort**
  - **Ogni suddivisione del problema in sottoproblemi porta alla stessa soluzione**
  - **I sottoproblemi sono disgiunti**
  - **Il calcolo avviene in maniera "top-down"**
- ***Programmazione Dinamica*** **: Moltiplicazioni tra Matrici**
  - **Differenti suddivisioni del problema portano a differenti soluzioni. La maggior parte di queste non porta ad una soluzione ottima.**
  - **I sottoproblemi si sovrappongono (non sono indipende-nti)**
  - **Il calcolo avviene in maniera "bottom-up"**

Sottosequenza

Una sequenza *$S'=(a_1,\ldots,a_m)$* è una *sottosequenza* della sequenza *S* se *S'* è ottenuta da *S rimuovendo uno o più elementi*.

Gli elementi rimanenti devono comparire nello *stesso ordine* nella sequenza risultante, anche se *non* devono essere *necessariamente contigui* in *S*.

- **La *sequenza nulla* (di *lunghezza* 0) è una *sottosequenza* di ogni sequenza.**
- **Una *sequenza* è una *lista* di elementi ma….**
- **… una *sottosequenza non è* necessariamente una *sottolista* (poiché deve essere contigua).**

Sottosequenza più lunga comune

Definizione: **Date due sequenze** $S_1=(a_1,\ldots,a_m)$ **e** $S_2=(b_1,\ldots,b_n)$**, una** ***sottosequenza comune Z*** **di** S_1 **e** S_2 **è una sequenza che è sottosequenza di entrambe le sequenze** S_1 **e** S_2**.**

Definizione: **Date due sequenze** $S_1=(a_1,\ldots,a_m)$ **e** $S_2=(b_1,\ldots,b_n)$**, la** ***più lunga sottosequenza comune Z*** **(*****LCS*****) di** S_1 **e** S_2 **è la** ***sottosequenza comune*** **di** S_1 **e** S_2 **che ha** ***lungezza massima*****.**

Sottosequenza più lunga comune

- **Input**
  - **2 sequenze di simboli, S_1 e S_2**
- **Output**
  - **Trovare la più lunga sottosequenza comune (*LCS*) di S_1 e S_2**

Esempio

$S_1 = \underline{AAA}A\underline{T}TG\underline{A}$ $\quad$ $S_2 = T\underline{AA}CG\underline{ATA}$

$LCS[S_1,S_2] = AAATA$

Soluzione esaustiva

```
LCS-Esaustivo(X[],Y[]: array di char)
 l = 0
 LCS = Nil
 /* Enumera tutte le sottosequenza di X */
 for each "sottosequenza K di X"
   do if "K è una sottosequenza di Y"
         then ls = lunghezza[K]
              if ls > l
               then l = ls
                    LCS = K
 return LCS
```

Miglioramento possibile:

Se lunghezza[*Y*] < lunghezza[*X*] enumera solo le sotto-sequenze di *lunghezza minore o uguale* a lunghezza[*Y*].

Soluzione esaustiva

Ma data una sequenza di lunghezza n, quante sono le possibili sottosequenze?

Ogni sottosequenza può avere lunghezza k con $0 \le k \le n$.

Cioè il numero di sottosequenze è pari al numero di *possibili sottoinsiemi* di elementi *distinti* di un insieme degli n elementi nella sequenza originaria

$$\sum_{k=0}^{n} \binom{n}{k} = \Theta(2^n)$$

Soluzione impraticabile!

① *Carattedizzazione della soluzione ottima*

Iniziamo col caratterrizzare la struttura della soluzione ottima.

Data una sequenza $X=(x_1,\dots,x_n)$, denoteremo con X_i, l'i-esimo *prefisso* di X, cioè la sotto-sequenza $(x_1,\dots,x_i)$. X_0 denota la sotto-sequenza nulla.

Esempio:

- X = ABDCCAABD
- X_3 = ABD
- X_6 = ABDCCA

1) $x_m = y_n$

$X[1,\ldots,m]$	X	$X[1,\ldots,m\text{-}1]$	
$Y[1,\ldots,n]$	X	$Y[1,\ldots,n\text{-}1]$	
$Z[1,\ldots,k]$	X	$Z[1,\ldots,k\text{-}1]$	

2) $x_m \neq y_n$ e $z_k \neq x_m$

$X[1,\ldots,m]$	X	$X[1,\ldots,m\text{-}1]$	
$Y[1,\ldots,n]$	Y	$Y[1,\ldots,n]$	Y
$Z[1,\ldots,k]$	?	$Z[1,\ldots,k]$	?

3) $x_m \neq y_n$ e $z_k \neq y_m$

$X[1,\ldots,m]$	X	$X[1,\ldots,m]$	X
$Y[1,\ldots,n]$	Y	$Y[1,\ldots,n\text{-}1]$	
$Z[1,\ldots,k]$	?	$Z[1,\ldots,k]$	?

① *Carattedizzazione della soluzione ottima*

Teorema: **Date le due sequenze $X=(x_1,\ldots,x_m)$ e $Y=(y_1,\ldots,y_n)$, sia $Z=(z_1,\ldots,z_k)$ una LCS di X e Y.**

1 **se $x_m = y_n$, allora $z_k = x_n = y_n$ e Z_{k-1} è una *LCS* di X_{m-1} e Y_{n-1}.**

2 **se $x_m \neq y_n$ e $z_k \neq x_m$, allora Z è una *LCS* di X_{m-1} e Y.**

3 **se $x_m \neq y_n$ e $z_k \neq y_n$, allora Z è una *LCS* di X e Y_{n-1}.**

① *Carattedizzazione della soluzione ottima*

Teorema: **Date le due sequenze** $X=(x_1,\ldots,x_m)$ **e** $Y=(y_1,\ldots,y_n)$**, sia** $Z=(z_1,\ldots,z_k)$ **una LCS di** X **e** Y**.**

1 **se** $x_m = y_n$**, allora** $z_k = x_m = y_n$ **e** $Z_{k\text{-}1}$ **è una** ***LCS*** **di** $X_{m\text{-}1}$ **e** $Y_{n\text{-}1}$**.**

2 **se** x_m **¹** y_n **e** z_k **¹** x_m**, allora** Z **è una** ***LCS*** **di** $X_{m\text{-}1}$ **e** Y**.**

3 **se** x_m **¹** y_n **e** z_k **¹** y_n**, allora** Z **è una** ***LCS*** **di** X **e** $Y_{n\text{-}1}$**.**

Dimostrazione:

1 **Suponiamo** $x_m = y_n$ **ma** z_k **¹** x_m**.**

Poiché Z **è una** ***sottosequenza comune*** **di** X **e** Y **di lunghezza** k**,**

Ma allora concatenando x_m **a** Z **otterremmo una** ***sotto-sequenza comune*** **di** X **e** Y **di lunghezza** $k+1$**.**

Questo però contraddice l'assunzione che Z **sia una** ***LCS*** **di** X **e** Y**.**

Quindi deve essere $z_k = x_m = y_n$

① *Carattedizzazione della soluzione ottima*

Teorema: **Date le due sequenze $X=(x_1,\ldots,x_m)$ e $Y=(y_1,\ldots,y_n)$, sia $Z=(z_1,\ldots,z_k)$ una LCS di X e Y.**

1. **se $x_m = y_n$, allora $z_k = x_m = y_n$ e Z_{k-1} è una *LCS* di X_{m-1} e Y_{n-1}.**
2. **se $x_m \neq y_n$ e $z_k \neq x_m$, allora Z è una *LCS* di X_{m-1} e Y.**
3. **se $x_m \neq y_n$ e $z_k \neq y_n$, allora Z è una *LCS* di X e Y_{n-1}.**

Dimostrazione:

Quindi deve essere $z_k = x_m = y_n$

Ma allora eliminando dalle 3 sequenze l'ultimo carattere, otteniamo che Z_{k-1} deve essere una *sottosequenza comune* di X_{m-1} e Y_{n-1} di lunghezza k-1

Ma Z_{k-1} è anche una *LCS* di X_{m-1} e Y_{n-1}?

(*Per assurdo*) Supponiamo che esista una *sottosequenza comune* W di X_{m-1} e Y_{n-1} di *lunghezza maggiore* di k-1.

Allora concatenando z_k a W otterremmo ancora una *sotto-sequenza comune più lunga* di Z, contraddicendo l'ipotesi.

① Carattedizzazione della soluzione ottima

Teorema: **Date le due sequenze $X=(x_1,\ldots,x_m)$ e $Y=(y_1,\ldots,y_n)$, sia $Z=(z_1,\ldots,z_k)$ una LCS di X e Y.**

1. **se $x_m = y_n$, allora $z_k = x_m = y_n$ e $Z_{k\text{-}1}$ è una *LCS* di $X_{m\text{-}1}$ e $Y_{n\text{-}1}$.**
2. **se $x_m \neq y_n$ e $z_k \neq x_m$, allora Z è una *LCS* di $X_{m\text{-}1}$ e Y.**
3. **se $x_m \neq y_n$ e $z_k \neq y_n$, allora Z è una *LCS* di X e $Y_{n\text{-}1}$.**

Dimostrazione:

2 **Se $x_m \neq y_n$ e $z_k \neq x_m$, allora Z è una *LCS* di $X_{m\text{-}1}$ e Y_n**

Infatti, se esistesse una *sottosequenza comune* W di $X_{m\text{-}1}$ e Y di lunghezza maggiore di k, allora (essendo $x_m \neq y_n$ e $z_k \neq x_m$) W sarebbe pure una *sottosequenza comune* di X_m e Y.

Ma questo contraddice l'ipotesi che Z sia una *LCS* di X e Y.

3 **Se $x_m \neq y_n$ e $z_k \neq y_m$, allora Z è una *LCS* di X e $Y_{n\text{-}1}$**

La dimostrazione è simmetrica a quella del punto 2

② *Definizione ricorsiva della soluzione ottima*

- $c(i,j)$ = ***lunghezza*** **della** ***LCS*** **di** X_i **e** Y_j.
- **Vogliamo trovare** $c(m,n)$ **dove** m **e** n **sono le lunghezze delle (sotto-)sequenze** X **e** Y, **rispet-tivamente.**

$c(i,j) = 0$ **se** i=0 **o** j=0

$c(i,j)$ = ? **se** i,j ¹ 0

1) $x_m = y_n$

$X[1,\ldots,i]$ X — $X[1,\ldots,i\text{-}1]$

$Y[1,\ldots,j]$ X — $Y[1,\ldots,j\text{-}1]$

$Z[1,\ldots,k]$ X — $Z[1,\ldots,k\text{-}1]$

$c(i,j) = c(i\text{-}1,j\text{-}1) + 1$ se $i,j > 0$ e $x_i = y_j$

2) $x_m \neq y_n$ (e $z_k \neq x_m$)

$X[1,\ldots,i]$ X — $X[1,\ldots,i\text{-}1]$

$Y[1,\ldots,j]$ Y — $Y[1,\ldots,j]$ Y

$Z[1,\ldots,k]$? — $Z[1,\ldots,k]$?

3) $x_m \neq y_n$ (e $z_k \neq y_m$)

$X[1,\ldots,i]$ X — $X[1,\ldots,i]$ X

$Y[1,\ldots,j]$ Y — $Y[1,\ldots,j\text{-}1]$

$Z[1,\ldots,k]$? — $Z[1,\ldots,k]$?

$c(i,j) = max\ \{\ c(i\text{-}1,j),\ c(i,j\text{-}1)\ \}$ se $i,j > 0$ e $x_i \neq y_j$

② *Definizione ricorsiva della soluzione ottima*

- ***c(i,j) = lunghezza*** **della** ***LCS*** **di** X_i **e** Y_j**.**
- **Vogliamo trovare** ***c(m,n)*** **dove** ***m*** **e** ***n*** **sono le lunghezze di** ***X*** **e** ***Y*****, rispettivamente.**

$$c(i,j) = \begin{cases} 0 & se\, i = 0\, o\; j = 0 \\ c[i-1, j-1] + 1 & se\;\; i, j > 0\;\; e\;\; x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & se\;\; i, j > 0\;\; e\;\; x_i \neq y_j \end{cases}$$

② Definizione ricorsiva della soluzione ottima

```
RLCS-length(X[],Y[]:array of char;i,j:intero)
 if i=0 or j=0
  then return 0
 else
   if X[i] ¹ Y[j]
    then
       return max(RLCS-length(X,Y,i,j-1),
                  RLCS-length(X,Y,i -1,j))
    else
       return 1+RLCS-length(b,X,i-1,j-1)
```

② Definizione ricorsiva della soluzione ottima

i,j

$i,j-1$ | $i-1,j$

$i,j-2$ | $i-1,j-1$ | $i-1,j-1$ | $i-2,j$

$i,j-3$ | $i-1,j-2$ | $i-1,j-2$ | $i-2,j-2$ | $i-1,j-2$ | $i-2,j-1$ | $i-2,j-1$ | $i-3,j$

$i-1,j-3$ | $i-2,j-2$ | $i-1,j-3$ | $i-2,j-2$ | $i-2,j-2$ | $i-3,j-1$

③ *Calcolo del valore della soluzione ottima*

Utilizziamo *due* tabelle di dimensione *m´n*

- ***c*[*i,j*] conterrà il *valore della lunghezza della massima sottosequenza comune* dei prefissi X_i e Y_j.**
  - ***c*[*m,n*] conterrà quindi *lunghezza della massima sottosequenza comune* di *X* e *Y*.**
- ***b*[*i,j*] conterrà un "*puntatore*" alla entry della tabella stessa che identifica il *sottoproblema ottimo* scelto durante il calcolo del valore *c*[*i,j*].**
  - **i valori possibili saranno ↖, - e ¬**

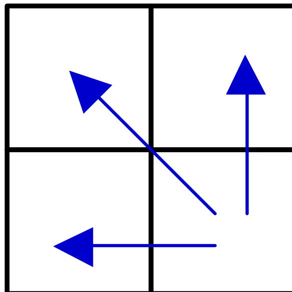
③ *Calcolo del valore della soluzione ottima*

Utilizziamo *due* tabelle di dimensione *m´n*

- ***b*[*i*,*j*] conterrà un "*puntatore*" alla entry della tabella stessa che identifica il *sottoproblema ottimo* scelto durante il calcolo del valore *c*[*i*,*j*].**
  - i valori possibili saranno ↖, - e ¬
    - ↖ **punta alla cella con indici *i*-1,*j*-1**
    - - **punta alla cella con indici *i*-1,*j***
    - ¬ **punta alla cella con indici *i*,*j*-1**

- **TACCBT**
- **ATBCBD**

↖	punta a *i*-1,*j*-1
-	punta a *i*-1,*j*
¬	punta a *i*,*j*-1



i \ j		0	1	2	3	4	5	6
			A	T	B	C	B	D
0		0	0	0	0	0	0	0
1	T	0	↑ 0	↖ 1	← 1	← 1	← 1	← 1
2	A	0	↖ 1	↑ 1	↑ 1	↑ 1	↑ 1	↑ 1
3	C	0	↑ 1	↑ 1	↑ 1	↖ 2	← 2	← 2
4	C	0	↑ 1	↑ 1	↑ 1	↑ 2	↑ 2	↑ 2
5	B	0	↑ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
6	T	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3

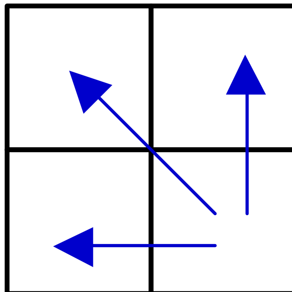
$c(i,j) = 0$ se $i=0$ o $j=0$

$c(i,j) = c(i-1,j-1) + 1$ se $i,j > 0$ e $x_i = y_j$

$c(i,j) = \max\{ c(i-1,j), c(i,j-1) \}$ se $i,j > 0$ e x_i ¹ y_j

- **ABDDBE**
- **BEBEDE**

↖	**punta a** ***i*-1,*j*-1**
-	**punta a** ***i*-1,*j***
¬	**punta a** ***i*,*j*-1**



i \ j		0	1	2	3	4	5	6
			B	E	B	E	D	E
0		0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
2	B	0	↖ 1	← 1	↖ 1	← 1	← 1	← 1
3	D	0	↑ 1	↑ 1	↑ 1	↑ 1	↖ 2	← 2
4	D	0	↑ 1	↑ 1	↑ 1	↑ 1	↖ 2	↑ 2
5	B	0	↖ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
6	E	0	↑ 1	↖ 2	↑ 2	↖ 3	← 3	↖ 3

$c(i,j) = 0$ se i=0 o j=0

$c(i,j) = c(i-1,j-1) + 1$ se $i,j > 0$ e $x_i = y_j$

$c(i,j) = \max\{ c(i-1,j), c(i,j-1) \}$ se $i,j > 0$ e x_i ¹ y_j

③ Calcolo del valore della soluzione ottima

```
LCS-Length(X[],Y[]: array of char)
 m=lunghezza[X]
 n=lunghezza[Y]
 for i=1 to m
   do c[i,0]=0
 for j=1 to n
   do c[0,j]=0
 for i=1 to m
   do for j=1 to n
        do if xi=yj
            then c[i,j]=1+c[i-1,j-1]
                 b[i,j]="↖"
           else if c[i-1,j] ³ c[i,j-1]
                 then c[i,j]=c[i-1,j]
                      b[i,j]="- "
                else c[i,j]=c[i,j-1]
                      b[i,j]="¬ "
 return
```

Inizalizzazione prima riga e prima colonna della marice

I carattri finali di X_i e Y_j sono identici!

I caratteri finali diversi
- *LCS* di X_{j-1} e Y_j è non peggiore
- *LCS* di X_j e Y_{j-1} è migliore

③ Calcolo del valore della soluzione ottima

```
LCS-Length(X[],Y[]: array of char)
 m=lunghezza[X]
 n=lunghezza[Y]
 for i=1 to m
   do c[i,0]=0
 for j=1 to n
   do c[0,j]=0
 for i=1 to m
   do for j=1 to n
        do if xi=yj
             then c[i,j] = 1+c[i-1,j-1]
                  b[i,j]="↖"
             else if c[i-1,j] ³ c[i,j-1]
                  then c[i,j]=c[i-1,j]
                       b[i,j]="-"
                  else c[i,j]=c[i,j-1]
                       b[i,j]="¬"
 return
```

Θ(m+n)

Θ(n)

Θ(nm)

④ *Costruzione della soluzione ottima*

Per costruire la *LCS* possiamo quindi utilizzare la tabella *b[i,j]* che contiene i "*puntatori*" ai *sottoproblemi ottimi* per ogni coppia di indici *i* e *j*.

Partiamo dalla entry *b[m,n]* e procediamo a ritroso seguendo le "*frecce*" della tabella.

Ricordate che *LCS-Length* assegna *b[i,j]* = ↖ solo quando $x_i = y_j$.

Quindi, partendo dalla fine della tabella, ogni volta che in *b[i,j]* troviamo ↖ sappiamo che x_i ($=y_j$) è contenuto nella *LCS* che cerchiamo.

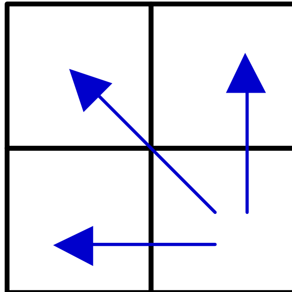
I valori vengono stampati nell'ordine corretto.

④ Costruzione della soluzione ottima

```
Print-LCS(b[]: array;X[]: array;i,j: intero)
 if i = 0 or j = 0
  then return
 if b[i,j] = "↖"
  then Print-LCS(b,X,i-1,j-1)
       print xi
  else if b[i,j] = "-"
        then Print-LCS(b,X,i-1,j)
        else Print-LCS(b,X,i,j-1)
```

- TACCBT
- ATBCBD

i \ j		0	1	2	3	4	5	6
			A	T	B	C	B	D
0		0	0	0	0	0	0	0
1	T	0	0	1	1	1	1	1
2	A	0	1	1	1	1	1	1
3	C	0	1	1	1	2	2	2
4	C	0	1	1	1	2	2	2
5	B	0	1	1	2	2	3	3
6	T	0	1	2	2	2	3	3



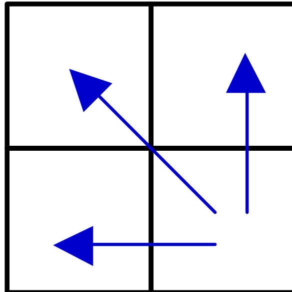
$c(i,j) = 0$ se $i=0$ o $j=0$

$c(i,j) = c(i-1,j-1) + 1$ se $i,j > 0$ e $x_i = y_j$

$c(i,j) = \max\{ c(i-1,j), c(i,j-1) \}$ se $i,j > 0$ e $x_i \neq y_j$

- **TACCBT**
- **ATBCBD**

- **TCB**



i \ j		0	1	2	3	4	5	6
			A	T	B	C	B	D
0		0	0	0	0	0	0	0
1	T	0	0	1	1	1	1	1
2	A	0	1	1	1	1	1	1
3	C	0	1	1	1	2	2	2
4	C	0	1	1	1	2	2	2
5	B	0	1	1	2	2	3	3
6	T	0	1	2	2	2	3	3

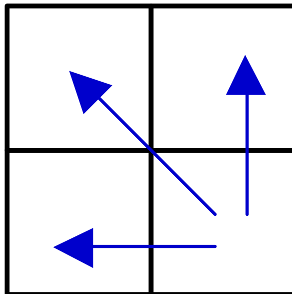
$c(i,j) = 0$ se $i=0$ o $j=0$

$c(i,j) = c(i-1,j-1) + 1$ se $i,j > 0$ e $x_i = y_j$

$c(i,j) = \max\{ c(i-1,j), c(i,j-1) \}$ se $i,j > 0$ e $x_i \neq y_j$

- **ABDDBE**
- **BEBEDE**

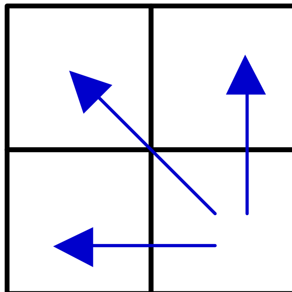
i \ j		0	1	2	3	4	5	6
			B	E	B	E	D	E
0		0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↑0	↑0	↑0
2	B	0	↖1	←1	↖1	←1	←1	←1
3	D	0	↑1	↑1	↑1	↑1	↖2	←2
4	D	0	↑1	↑1	↑1	↑1	↖2	↑2
5	B	0	↖1	↑1	↖2	←2	↑2	↑2
6	E	0	↑1	↖2	↑2	↖3	←3	↖3



$c(i,j) = 0$	se $i=0$ o $j=0$
$c(i,j) = c(i-1,j-1) + 1$	se $i,j > 0$ e $x_i = y_j$
$c(i,j) = \max\{ c(i-1,j), c(i,j-1) \}$	se $i,j > 0$ e $x_i \neq y_j$

- **ABDDBE**
- **BEBEDE**

- **BDE**



i \ j		0	1	2	3	4	5	6
			B	E	B	E	D	E
0		0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↑0	↑0	↑0
2	B	0	↖1	←1	↖1	←1	←1	←1
3	D	0	↑1	↑1	↑1	↑1	↖2	←2
4	D	0	↑1	↑1	↑1	↑1	↖2	↑2
5	B	0	↖1	↑1	↖2	←2	↑2	↑2
6	E	0	↑1	↖2	↑2	↖3	←3	↖3

$c(i,j) = 0$ se $i=0$ o $j=0$

$c(i,j) = c(i-1,j-1) + 1$ se $i,j > 0$ e $x_i = y_j$

$c(i,j) = \max\{ c(i-1,j), c(i,j-1) \}$ se $i,j > 0$ e $x_i \neq y_j$

④ Costruzione della soluzione ottima

```
Print-LCS(b[]: array;X[]: array;i,j: intero)
 if i = 0 or j = 0
  then return
 if b[i,j] = "↖"
  then Print-LCS(b,X,i-1,j-1)
       print x_i
  else if b[i,j] = "-"
        then Print-LCS(b,X,i-1,j)
        else Print-LCS(b,X,i,j-1)
```

Poiché ad ogni passo della ricorsione *almeno uno* tra i e j *viene decrementato*, il tempo di esecuzione è pari alla *somma delle lunghezze* delle due sequenze: $\Theta(m+n)$

Ottimizzazioni possibili

- **La tabella $b[i,j]$ può essere eliminata, risparmiando spazio di un fattore costante.**
  - **Il valore di $c[i,j]$ dipende solo dai valori $c[i-1,j-1]$, $c[i,j-1]$ e $c[i-1,j]$. In tempo costante si può quindi determinare quale di questi tre è stato usato, e quindi quale sia il tipo di freccia. (Esercizio 16.3-2)**
- **Se ci serve solo calcolare la lunghezza della *LCS*, possiamo ancora ridurre lo spazio della tabella $c[i,j]$ a due sole righe di lunghezza $min\{n, m\}$**
  - **Ad ogni istante (cioè per ogni coppia i,j), ci servono i valori $c[i-1,j-1]$, $c[i,j-1]$ e $c[i-1,j]$, che stanno nella riga corrente e in quella precedente. (Esercizio 16.3-4)**

Esercizi

Esercizi:

- **16.3-2**
- **16.3-3**
- **16.3-4**

Problemi:

- **16.2**
- **16.3**
- **16.4**