```
INIT_LCS (X, Y)                    (senza l'info addizionale)
    m ← length (X)
    n ← length (Y)
    if m=0 OR n=0 then return 0
    for i ← 0 to m do
        L[i,0] ← 0
    for j ← 1 to n do
        L[0,j] ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            L[i,j] ← -1
    return R_LCS (X, Y, m, n)
```

Questo di fianco è l'algoritmo memoizzato, a metà tra ricorsione e algoritmo ricorsivo.
Dopo i singoli cicli di inizializzazione, tra i e j metto -1, ritornando i singoli pezzi ricorsivamente.
Proseguo nella ricerca dei prefissi, non buttando via il risultato dopo i confronti negli indici"i" e "J".
Si nota come rappresenta una ottimizzazione del codice presente sopra.

L'algoritmo sotto è più ottimizzato temporalmente e comprende anche la fase top-down della versione ricorsiva, ma evita di riempire la tabella come farebbe l'altro.
Infatti l'istanza non verrebbe più ricalcolata vista l'uguaglianza.

```
R_LCS (X, Y, i, j)
    if L[i,j] = -1 then
        if (x_i = y_j) then
            L[i,j] ← R_LCS (X, Y, i-1, j-1) + 1
        else if R_LCS (X, Y, i-1, j) ≥ R_LCS (X, Y, i, j-1) then
                L[i,j] ← L[i-1, j]
            else
                L[i,j] ← L[i, j-1]
    return L[i,j]
```

Complessità: $O(m \cdot n)$

↳ è al caso pessimo; ma nel caso migliore?

↳ questo alg. ha complessità più bassa

Osservazione: - alg. iterativo fa sempre $m \cdot n$ confronti

- nell'alg. memoizzato, se $x_i = y_j$ non calcola i sottoproblemi $i-1, j$ e $i, j-1$ ($\Rightarrow$ nelle loro posizioni in L rimane il valore di default)

Esempio:

Nella tabella evito ogni singola istanza di i-1 ricorsivamente, tenendo quindi solamente quelle che fanno match. Il discorso dell'ottimizzazione riguarda togliere tutti gli elementi della riga dell'elemento dove c'è un match, togliendo tutti gli altri elementi della riga perché non mi servono.

$X = \langle A, A, B, B, C \rangle$

$Y = \langle B, B, C \rangle$

istanze generate:

$$(5,3) \xrightarrow{C=C} (4,2) \xrightarrow{B=B} (3,1) \xrightarrow{B=B} (2,0)$$

$$\begin{cases} \cancel{(4,3)} \\ \cancel{(5,2)} \end{cases} \qquad \cancel{(4,1)} \quad \cancel{(3,2)} \qquad \cancel{(3,1)} \quad \cancel{(3,0)}$$

sottoproblemi non risolti

Caso Y = suffisso di $n \leq m$ caratteri di X:

$T_{best}(m, n) = n$ $\rightarrow$ addirittura sublineare!

* Esempio codice bottom-up: *

$X = \langle B, D, C, D \rangle$

$Y = \langle A, B, C, B, D \rangle$

segue $LCS(X, Y)$

		A	B	C	B	D	← Y
	0	1	2	3	4	5	
	0	0	0	0	0	0	0
B	1	0	0/↑	1/↖	1/←	1/↖	1/←
D	2	0	0/↑	1/↑	1/↑	1/↑	2/↖
C	3	0	0/↑	1/↑	2/↖	2/←	2/↑
D	4	0	0/↑	1/↑	2/↑	2/↑	3/↖

↑ X

Nella tabella qui di fianco, vediamo come i numeri, per es. 0 o 1 rappresenta l'aggiunta di un carattere o meno, confrontando sempre massimo o minimo. DI fatto, si aggiunge 1 all'el. diagonale qualora io trovi l'elemento massimo tra elementi adiacenti.
Quando c'è un match, mi sposto in diagonale, restituendolo qualora match c'è.

L/B

B C D ← la LCS

↳ seleziono il carattere solo quando nel cammino da $B(m,n)$ trovo ↖

Codice per stampare la LCS: non fa altro che seguire il cammino da $B(m,n)$

```
PRINT_LCS (B, X, i, j)
    if i=0 or j=0 then return ε
    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, j-1)
        else PRINT_LCS (B, X, i-1, j)
return
```

Complessità: $O(m)$ o, più precisamente, Θ(taglia della LCS)