Università di Torino – Facoltà di Scienze MFN
Corso di Studi in Informatica
Curriculum SR (Sistemi e Reti)

Algoritmi e Laboratorio a.a. 2006-07
Lezioni

prof. Elio Giovannetti

Parte 26 – Programmazione dinamica:
la più lunga sottosequenza comune.
(versione 12/02/2007)

Definizioni preliminari

- sottosequenza di una sequenza $a_0, a_1, \ldots, a_{m-1}$:
  è una qualunque sequenza $a_{i_0}, a_{i_1}, \ldots, a_{i_k}$ con $i_k < i_{k+1}$;

  cioè una sottosequenza di una sequenza S è una qualunque sequenza di elementi di S non necessariamente contigui ma nello stesso ordine che in S;

  cioè una sottosequenza di una data sequenza è la sequenza stessa alla quale sono stati tolti zero o più elementi;
- in particolare sono sottosequenze di una sequenza S sia la sequenza vuota che la sequenza stessa S.
- Non confondere le sottosequenze con i segmenti (vedi def. Lez 04): tutti i segmenti di S sono sottosequenze di S, ma non tutte le sottosequenze di S sono segmenti di S.

Il problema della più lunga sottosequenza comune (Longest Common Subsequence o LCS)

Date due sequenze S1 ed S2, trovare la più lunga sequenza S che è sottosequenza sia di S1 che di S2, cioè trovare una sottosequenza comune di lunghezza massima.
Se vi sono più sottosottosequenze comuni di lunghezza massima, trovarne una, non importa quale.

Esempi del problema:
- biologia: trovare la più lunga sottosequenza comune a due sequenze di DNA;
- sicurezza informatica: individuare, in un log costituito da una sequenza di comandi, quelle sottosequenze che indicano la presenza di un possibile attacco al sistema.

Esempio

Date le due sequenze di DNA:

AGCCGGATCGAGT
TCAGTACGTTA

una sottosequenza comune di lunghezza massima è:

AGCGTA

Un'altra sottosequenza comune di lunghezza massima, per le stesse due sequenze, è:

AGTCGA

Infatti:

AGCCGGATCGAGT
TCAGTACGTTA

L'algoritmo ingenuo (o "a forza bruta").

Siano S1 ed S2 le due sequenze.
Si generano tutte le sottosequenze di S1, e per ognuna di esse si controlla se è anche sottosequenza di S2, tenendo traccia della più lunga trovata.
Ogni sottosequenza di S1 corrisponde a un sottoinsieme dell'insieme {0, 1, 2, ..., m-1} degli indici di S1; vi sono quindi 2^m sottosequenze di S1 (vedi Logica o Matematica Discreta)

Per chi non si ricorda: Data una sequenza $a_0, a_1, \ldots a_{m-1}$, per costruire una sua sottosequenza si hanno le seguenti scelte:
- a_0 può esserci oppure no: 2 possibilità;
- per ognuna di esse, a_1 può esserci oppure no (2^2 possibilità);
- per ognuna di esse, a_2 può esserci oppure no (2^3 possibilità);
- ecc.

L'algoritmo ingenuo o "a forza bruta".

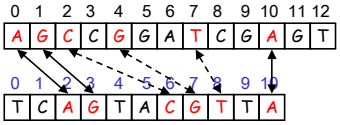
- Per generare ed esaminare 2^m sottosequenze occorre quindi un tempo $\Theta(2^m)$. Più precisamente, per ogni sottosequenza di S1 occorrono nel caso peggiore n passi per controllare se è anche sottosequenza di S2. La complessità è quindi $2^m n$.
- L'algoritmo è dunque esponenziale ! è inutilizzabile !
- Esiste un algoritmo migliore, che sfrutta la tecnica detta della programmazione dinamica.

Notazione.

Una sottosequenza comune a due sequenze verrà indicata da una sequenza di coppie di indici rispettivamente delle due sequenze, corrispondenti agli elementi uguali.

Esempio:

La più lunga sottosequenza comune di:



è la sottosequenza comune AGCGTA indicata da:

[(0, 2), (1, 3), (2, 6), (4, 7), (7, 8), (10, 10)]

L'algoritmo: anteprima.

Per trovare la più lunga sottosequenza comune di due sequenze di lunghezze rispettivamente m ed n, consideriamo una tabella (matrice) **LCS** con m+1 righe corrispondenti agli elementi della prima sequenza, ed n+1 colonne corrispondenti agli elementi della seconda sequenza.

La casella **LCS[i, j]** dovrà virtualmente contenere la più lunga sottosequenza comune dei due segmenti iniziali di lunghezze rispettive i e j;

la riga e la colonna in più rappresentano i rispettivi segmenti iniziali vuoti;

ogni casella della prima riga o della prima colonna contiene quindi una sottosequenza vuota, di lunghezza 0 (perché la LCS di due sequenze di cui una vuota è vuota).

Esempio: lcs("BACATBA", "ATCBAB")

Φ = lcs("BAC", "ATCB")

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0						
A	-- 0						
C	-- 0				Φ		
A	-- 0						
T	-- 0						
B	-- 0						
A							

Riempiamo la tabella calcolando successivamente il contenuto di ogni casella, partendo dall'angolo a sinistra in alto, e percorrendo la tabella riga per riga (sarebbe indifferente percorrerla colonna per colonna).

L'ultima casella che riempiremo, cioè quella nell'angolo a destra in basso, conterrà la soluzione.

lcs("B", "ATCB") = "B", di lunghezza 1

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1		
A	-- 0						
C	-- 0						
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

casella successiva:

lcs("B", "ATCBA")

poiché la casella corrisponde a due caratteri diversi nella riga e nella colonna, non può corrispondere a una LCS con un elemento in più; si ha quindi:

lcs("B", "ATCBA") = lcs("B", "ATCB");

mettiamo nella casella una freccia, per indicare che essa contiene la stessa lcs della cella contigua in orizzontale.

lcs("B", "ATCBA") = lcs("B", "ATCB") , lunghezza 1

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	
A	-- 0						
C	-- 0						
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("B", "ATCBAB") = "B", lunghezza 1

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	**B** 1
A	-- 0						
C	-- 0						
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BA", "A") = "A", lunghezza 1

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	**B** 1
A	-- 0	**A** 1					
C	-- 0						
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BA", "AT") = lcs("BA", "A") = "A", lunghezza 1

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	**B** 1
A	-- 0	**A** 1	← 1				
C	-- 0						
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BA", "ATC") = lcs("BA", "AT") = ... = "A", lungh. 1

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	**B** 1
A	-- 0	**A** 1	← 1	← 1			
C	-- 0						
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BA", "ATCB") = lcs("BA", "ATC") = ... = "A", lunghezza 1
lcs("BA", "ATCB") = lcs("B", "ATCB") = "B", lunghezza 1
è indifferente, scegliamo la seconda

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	**B** 1
A	-- 0	**A** 1	← 1	← 1	↑ 1		
C	-- 0						
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

Consideriamo la casella successiva, cioè

lcs("BA", "ATCBA")

Riga e colonna hanno lo stesso carattere, cioè i due segmenti terminano con lo stesso carattere: allora si deve aggiungere tale carattere alla lcs("B", "ATCB"), che è la casella nella riga e colonna precedenti, cioè la casella precedente "in diagonale".

lcs("BA", "ATCBA") = lcs("B", "ATCB") + A = "BA", lunghezza 2

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	1	**B** 1
A	-- 0	**A** 1	1	1	1	BA 2	
C	-- 0						
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BA", "ATCBAB") = lcs("BA", "ATCBA") = ... = "BA", lun. 2

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	1	**B** 1
A	-- 0	**A** 1	1	1	1	BA 2	2
C	-- 0						
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BAC", "A") = lcs("BA", "A") = "A", lun. 1

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	1	**B** 1
A	-- 0	**A** 1	1	1	1	BA 2	2
C	-- 0	1					
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BAC", "AT") = lcs("BA", "AT") =... = "A", lun. 1

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	1	**B** 1
A	-- 0	**A** 1	1	1	1	BA 2	2
C	-- 0	1	1				
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BAC", "ATC") = lcs("BA", "AT") + C = ... = "AC", lun. 2

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	1	**B** 1
A	-- 0	**A** 1	1	1	1	BA 2	2
C	-- 0	1	1	AC 2			
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BAC", "ATCB") = lcs("BAC", "ATC") = "AC", lun. 2

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	**B** 1
A	-- 0	**A** 1	← 1	← 1	↑ 1	BA 2	← 2
C	-- 0	↑ 1	↑ 1	AC 2	← 2		
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BAC", "ATCBA") = lcs("BA", "ATCBA") = "BA", lun. 2

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	**B** 1
A	-- 0	**A** 1	← 1	← 1	↑ 1	BA 2	← 2
C	-- 0	↑ 1	↑ 1	AC 2	← 2	↑ 2	
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BAC", "ATCBAB") = lcs("BA", "ATCBAB") = "BA", lun. 2

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	**B** 1
A	-- 0	**A** 1	← 1	← 1	↑ 1	BA 2	← 2
C	-- 0	↑ 1	↑ 1	AC 2	← 2	↑ 2	↑ 2
A	-- 0						
T	-- 0						
B	-- 0						
A	-- 0						

Osserva l'algoritmo usato:

A ciascuna nuova casella:

- se si ha lo stesso carattere sulla riga e sulla colonna:
  - si mette una "freccia diagonale";
  - si aumenta di 1 la lunghezza rispetto alla casella puntata dalla freccia;
- se sulla riga e sulla colonna ci sono due caratteri diversi:
  - si mette una freccia verso l'alto o verso sinistra, cioè verso la cella di lunghezza maggiore fra la cella contigua sopra e la cella contigua a sinistra (se hanno uguale lunghezza si sceglie ad es. quella sopra)
  - la lunghezza è la stessa di quella della cella puntata dalla freccia.

La lcs "contenuta" in ciascuna casella si ottiene, dall'ultimo elemento al primo, seguendo le frecce a partire dalla casella.

andiamo avanti di alcuni passi

lcs("BACA", "ATCBA") = lcs("BAC", "ATCB") + A = "ACA", lun. 3

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	← 1	**B** 1
A	-- 0	**A** 1	← 1	← 1	↑ 1	BA 2	← 2
C	-- 0	↑ 1	↑ 1	AC 2	← 2	↑ 2	↑ 2
A	-- 0	**A** 1	↑ 1	↑ 2	↑ 2	ACA 3	
T	-- 0						
B	-- 0						
A	-- 0						

Osservazione

Si noti che non è necessario memorizzare i caratteri nelle celle; l'informazione necessaria in ciascuna cella è:

- la freccia, che permette di ricostruire all'indietro la lcs;
- la lunghezza, che permetterà di calcolare le successive celle contigue.

Se si è interessati solo alla lunghezza della lcs, ma non alla effettiva sequenza, le frecce non servono, e basta costruire la matrice delle lunghezze.

lcs("BACA", "ATCBAB") = lcs("BACA", "ATCBA") = "ACA" lun. 3

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	1	**B** 1
A	-- 0	**A** 1	1	1	1	BA 2	2
C	-- 0	1	1	AC 2	2	2	2
A	-- 0	**A** 1	1	2	2	ACA 3	3
T	-- 0						
B	-- 0						
A	-- 0						

lcs("BACAT", "A") = lcs("BACA", "A") = "A", lun. 3

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	1	**B** 1
A	-- 0	**A** 1	1	1	1	BA 2	2
C	-- 0	1	1	AC 2	2	2	2
A	-- 0	**A** 1	1	2	2	ACA 3	3
T	-- 0	1					
B	-- 0						
A	-- 0						

Alla fine:

lcs("BACATBA", "ATCBAB") = ... = ACAB, lun. 4

		A	T	C	B	A	B
	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0	-- 0
B	-- 0	-- 0	-- 0	-- 0	**B** 1	1	**B** 1
A	-- 0	**A** 1	1	1	1	BA 2	2
C	-- 0	1	1	AC 2	2	2	2
A	-- 0	**A** 1	1	2	2	ACA 3	3
T	-- 0	1	AT 2	2	2	3	3
B	-- 0	1	2	2	ACB 3	3	**4** ACAB
A	-- 0	1	2	2	3	4 ACBA	4

Nota

- Si noti che, osservando la tabella, si individuano le altre sequenze massimali ACBA e ATBA, corrispondenti a scelte diverse in punti in cui era indifferente scegliere la freccia verticale o la freccia orizzontale.

Giustificazione rigorosa dell'algoritmo.
(formulazione per sequenze realizzate come array)

Il problema: Dati due array a e b, di lunghezze risp. m ed n, che indichiamo con a[0 .. m-1] e b[0 .. n-1], trovare una più lunga sottosequenza comune (Longest Common Subsequence)

lcs(a[0 .. m-1], b[0 .. n-1]) = [(i_0, j_0), (i_1, j_1), ... (i_h, j_h)]

Generalizziamo il problema nel modo ovvio: trovare una più lunga sottosequenza comune di due segmenti iniziali di a e b (i segmenti iniziali sono detti anche prefissi):

lcs(a[0 .. i], b[0 .. j])

Il problema originale è ovviamente un caso particolare del problema più generale, cioè per i = m-1 e j = n-1.

Semplifichiamo la notazione.

- Poiché tutte le più lunghe sottosequenze comuni che consideriamo sono del genere lcs(a[0 .. i], b[0 .. j]), possiamo alleggerire la notazione ponendo semplicemente

  LCS(i, j) = lcs(a[0 .. i], b[0 .. j]).

- Si ricordi quindi che, nelle prossime slides, LCS(i, j) è la più lunga sottosequenza comune di a[0 .. i] e b[0 .. j].

Risoluzione

Ragionando per induzione (completa), mostriamo come risolvere il problema generale, cioè mostriamo:

- base: come risolvere il problema in un opportuno insieme di casi-base;
- passo: come risolvere il problema se si sa risolvere lo stesso problema per qualunque dimensione inferiore (induzione completa).

Come vedremo, nella programmazione dinamica, a differenza che nel divide-et-impera, ciascun problema di dimensione inferiore è utilizzato nella risoluzione di più problemi di dimensione superiore; le soluzioni di tutti i problemi di dimensione inferiore devono quindi essere memorizzate, e non semplicemente restituite ricorsivamente al chiamante.

I casi base

- Se una delle due sequenze è vuota, la loro LCS è vuota. Nella nostra notazione i prefissi vuoti di a e b sono rispett.:

  a[0 .. –1] e b[0 .. –1]

- Si ha quindi:

  $LCS(-1, j) = []$ per $-1 \le j \le n-1$

  $LCS(i, -1) = []$ per $-1 \le i \le m-1$

Esempio:

Quindi lunghezza di LCS(-1, j) = lunghezza di LCS(i, -1) = 0

Il passo induttivo

- Supponiamo di aver trovato LCS(i', j') per tutte le coppie di indici (i', j') tali che i'< i or j'< j.
- Facciamo vedere come da tali soluzioni (in realtà da solo tre di esse) si può ricavare la sequenza LCS(i, j).

Cioè, un po' più formalmente:

- Ip. Induttiva: Per tutte le coppie (i', j') tali che i'+ j' < i +j, la sequenza LCS(i', j') è nota (è stata trovata).
- Tesi Induttiva: Si sa calcolare LCS(i, j).

Per trovare la LCS(i, j) occorre esaminare a[i] e b[j] e considerare diversi casi possibili. Vedi la prossima slide.

Dimostrazione (= idea centrale dell'algoritmo)

- **caso 1**: a[i] = b[j]

Come si vede chiaramente dal disegno,

se $LCS(i-1, j-1) = [(i_0, j_0), (i_1, j_1), \dots (i_k, j_k)]$

allora $LCS(i, j) = [(i_0, j_0), (i_1, j_1), \dots (i_k, j_k), (i, j)]$.

Chiamando lun(S) la lunghezza di una sequenza S, si ha inoltre:

$lun(LCS(i, j)) = lun(LCS(i-1, j-1)) + 1$

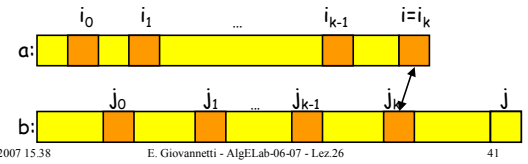
Dimostrazione (= idea centrale dell'algoritmo)

- **caso 2**: $a[i] \neq b[j]$

  In tal caso a[i] e b[j] non possono essere entrambi l'ultimo elemento di una sottosequenza comune, tuttavia è possibile che uno dei due elementi a[i] e b[j] lo sia, e sia quindi uguale ad un elemento (non l'ultimo) dell'altra sequenza;

  cioè, più rigorosamente:

  l'elemento finale di una sottosequenza comune non può essere la coppia (i, j), ma può essere una coppia (i, j') con j'< j, oppure (i', j) con i'< i. Esempio:



Dimostrazione (= idea centrale dell'algoritmo)

- **caso 2**: $a[i] \neq b[j]$ (continua)

  Le sottosequenze comuni di a[0 .. i] e b[0 .. j] sono quindi sottosequenze comuni di a[0 .. i-1] e b[0 .. j], oppure sottosequenze comuni di a[0 .. i] e b[0 .. j-1] (o di entrambi, si tratta di un or non esclusivo).

Quindi LCS(i, j) è la più lunga fra le due sequenze

LCS(i-1, j) e LCS(i, j-1).

Inoltre, per quanto riguarda le lunghezze, si ha ovviamente:

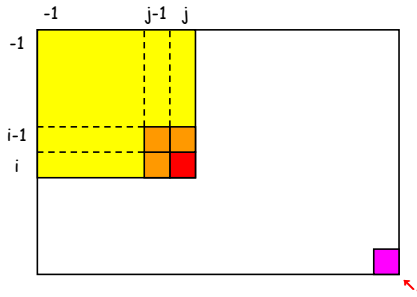
$lun(LCS(i, j)) = \max(lun(LCS(i-1, j)), lun(LCS(i, j-1)))$

Riassumendo:

- Sappiamo trovare LCS(-1, j) ed LCS(i, -1).
- Sappiamo in ogni caso trovare LCS(i, j) se abbiamo trovato LCS(i-1, j-1), LCS(i-1, j), LCS(i, j-1) (vedi Figura 1).
- Siamo allora in grado di costruire incrementalmente la matrice di tutte le LCS(i, j) per $-1 \le i \le m-1$ e $-1 \le j \le n-1$;
  - Nota: la matrice è dunque di dimensione $(m+1)\times(n+1)$, perché righe e colonne sono indiciate a partire da –1;
- Non si può usare lo schema divide-et-impera con la ricorsione, perché ogni soluzione LCS(i, j) può venire utilizzata tre volte e quindi deve venire memorizzata per non essere ricalcolata (vedi Figura 2).
- Basterebbe tenere in memoria solo tre elementi della matrice per volta, aggiornandoli ad ogni passo; è tuttavia più semplice, come vedremo, memorizzare l'intera matrice.

Costruzione incrementale della matrice: Figura 1.

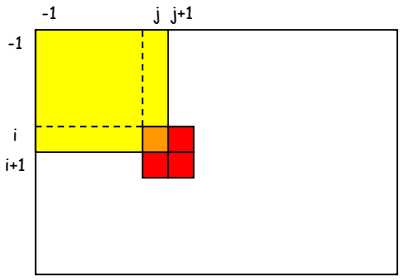
- Per trovare la LCS(i, j) basta conoscere le tre LCS contigue, cioè: LCS(i-1, j-1), LCS(i-1, j), LCS(i, j-1).



Costruzione incrementale della matrice: Figura 2.

- Reciprocamente, ogni LCS(i, j) può essere utilizzato per il calcolo di tre altri LCS:

  LCS(i+1, j+1), LCS(i, j+1), LCS(i+1, j).



Verso un'implementazione.

- Se si memorizza l'intera matrice, non è più necessario tenere in ciascun elemento (i, j) della matrice l'intera sequenza LCS(i, j): basta memorizzarne solo la lunghezza, il riferimento alla LCS utilizzata per trovarla, e l'eventuale nuovo ultimo elemento. Lo spazio necessario è quindi O(mn).
- Tali informazioni vengono espresse in modo sintetico, usando inoltre per semplicità due matrici "in parallelo" invece di una:
  - la matrice L delle lunghezze delle LCS(i, j);
  - la matrice F dei riferimenti alla LCS "precedente".
- Nei linguaggi derivati dal C gl'indici devono partire da 0, quindi andranno rispettivamente da 0 a m e da 0 a n invece da –1 a m-1 e da –1 a n-1, il che rende leggermente meno chiara la procedura.

Nota

Nel programma reale, se si vuole, si possono dare alle due matrici L ed F dei nomi più significativi (e con iniziale minuscola, in obbedienza alle convenzioni Java), ad esempio *lunghezze* e *frecce*, oppure *lunghezze* e *precLCS*, ecc.

Nelle slides seguenti, per ragioni di spazio, si sono invece usati i nomi L ed F costituiti da una sola lettera maiuscola, in accordo con l'uso matematico, ma in contrasto con le convenzioni Java.

L è l'iniziale di *lunghezze*, F è l'iniziale di *frecce*.

Le due matrici

- Siano, come al solito, m ed n le dimensioni rispettive dei due array di input a e b.
- L e F sono due matrici di dimensione $(m+1)\times(n+1)$, con le righe indiciate da 0 a m e le colonne da 0 a n.
- L è una matrice di interi, F una matrice di coppie di interi, definite come segue:

  L[i][j] = lun(LCS(i-1, j-1))
  cioè lunghezza della lcs(a[0 .. i-1], b[0 .. j-1])

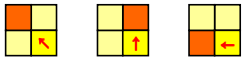
$$F[i][j] = \begin{cases} (i-1, j-1) & \text{se } LCS(i-1, j-1) = LCS(i-2, j-2), (i-1, j-1); \\ (i, j-1) & \text{se } LCS(i-1, j-1) = LCS(i-1, j-2); \\ (i-1, j) & \text{se } LCS(i-1, j-1) = LCS(i-2, j-1); \end{cases}$$

Se si potessero indiciare le matrici L e F a partire da –1, oppure gli array a e b a partire da 1, le definizioni di L e F sarebbero più semplici e chiare, senza la differenza di 1 fra i due membri.

Realizzazione alternativa della matrice F

Poiché per ciascun elemento di F vi sono solo tre scelte possibili, invece di memorizzare in F coppie di interi possiamo codificare ogni scelta mediante uno di tre simboli, che di fatto saranno i numeri 0, 1, 2, ma che rappresentiamo pittograficamente in modo da evidenziarne il significato. Poiché i è il numero di riga, e j il numero di colonna, si ha:

F[i, j]= ↖ se LCS(i-1, j-1) = LCS(i-2, j-2) seguita da (i-1, j-1)
← se LCS(i-1, j-1) = LCS(i-1, j-2) stessa riga, col-1
↑ se LCS(i-1, j-1) = LCS(i-2, j-1) stessa col., riga-1



L'implementazione: inizializzazione

L[i, 0] = lun(LCS(i-1, -1)) = 0 per ogni 0 ≤ i ≤ n
L[0, j] = lun(LCS(-1, j-1)) = 0 per ogni 0 ≤ j ≤ n

↖ ↑ ←

```
enum Freccia {DIAG, VERT, ORIZ}
...
int m = a.length;
int n = b.length;
int[][] L = new int[m+1][n+1];
Freccia[][] F = new Freccia[m+1][n+1];

for(int i = 0; i <= m; i++) L[i][0] = 0;
for(int j = 0; j <= n; j++) L[0][j] = 0;
```


L'implementazione: la costruzione delle due matrici.

Si effettua una banale iterazione sui due indici mediante due cicli for annidati.

Per scrivere tali cicli l'invariante non è quindi molto utile; esso afferma semplicemente che gli elementi della matrice con indici inferiori agl'indici correnti hanno i valori corretti, cioè sono stati correttamente calcolati.

INVARIANTE

∀i'≤ i . ∀j'≤ j . L[i'][j'] = lun(LCS(i'-1, j'-1))
∀i'≤ i . ∀j'≤ j . F[i'][j'] = DIAG se LCS(i'-1,j'-1) = ...
stessa riga, col. prec. ORIZ se LCS(i'-1, j'-1) = LCS(i'-1, j'-2)
stessa col., riga prec. VERT se LCS(i'-1, j'-1) = LCS(i'-2, j'-1)

L'implementazione: la costruzione delle due matrici.

```
for(int i = 1; i <= m; i++) {
  for(int j = 1; j <= n; j++) {
    if(a[i-1] == b[j-1]) {
      L[i][j] = L[i-1][j-1] + 1;
      F[i][j] = Freccia.DIAG;
    }
    else if(L[i][j-1] > L[i-1][j]) {
      L[i][j] = L[i][j-1]; // stessa riga, colonna preced.
      F[i][j] = Freccia.ORIZ;
    }
    else {
      L[i][j] = L[i-1][j]; // stessa colonna, riga preced.
      F[i][j] = Freccia.VERT;
    }
  }
}
```


Il problema più semplice di trovare solo la lunghezza della LCS è risolto.

- La lunghezza della più lunga sottosequenza comune di a e b si trova in `L[m][n]`; se si è interessati solo alla lunghezza, nella procedura precedente si può eliminare la matrice F e la sua costruzione, e l'algoritmo è terminato.
- Se invece si vuole sapere qual è la sottosequenza, occorre ricostruirla utilizzando la matrice F.

Ricostruzione della LCS.

- Esempio: lcs("GACATGC", "ATCGAG") = "ACAG"

matrice F

	0	1	2	3	4	5	6
0							
1		↑	↑	↑	↖	←	↖
2		↖	←	←	↑	↖	↑
3		↑	↑	↖	←	↑	↑
4		↖	↑	↑	↑	↖	←
5		↑	↖	↑	↑	↑	↑
6		↑	↑	↑	↖	↑	↖
7		↖	↑	↑	↑	↖	↑
		A	T	C	G	A	G
	-1	0	1	2	3	4	5

matrice L

		0	1	2	3	4	5	6
	-1							
G	0	0	0	0	0	0	0	0
A	1	0	0	0	0	1	1	1
C	2	0	1	1	1	1	2	2
A	3	0	1	1	2	2	2	2
T	4	0	1	1	2	2	3	3
G	5	0	1	2	2	2	3	3
C	6	0	1	2	2	3	3	4
	7	0	1	2	2	3	4	4
			A	T	C	G	A	G

Ottimizzazioni dello spazio.

Si noti che la prima riga e la prima colonna della matrice F sono inutili perché non contengono alcuna informazione.

Si potrebbe quindi usare come matrice F una matrice m×n; si è preferito usare una matrice (m+1)×(n+1) per avere gli stessi indici della matrice L.

Alternativamente, ma complicando un po' il codice, si potrebbe usare anche per L una matrice m×n, poiché la prima riga e la prima colonna di L sono sempre costituite da tutti zeri.

Una ottimizzazione di spazio talvolta usata consiste nel tenere solo due righe per volta della matrice L.

Altre più sofisticate ottimizzazioni sono possibili.

Ricostruzione della sottosequenza risultato

- Si parte dalla casella-soluzione (m, n) e si va all'indietro seguendo le frecce; in corrispondenza di ogni freccia diagonale si ha un elemento della sequenza risultato;
- in tal modo si ottiene però la sequenza invertita, a partire dall'ultimo elemento fino al primo;
- se la sequenza si vuole semplicemente stamparla, il modo più semplice per ottenere la stampa nell'ordine giusto è quello di definire una procedura ricorsiva, che effettua la stampa dopo la chiamata ricorsiva;
- se invece si vuole restituire la sequenza sotto forma di array, basta riempire iterativamente l'array a partire dal fondo.

Correttezza della ricostruzione della sequenza

- F[i][j] = ↖: l'ultimo elemento della LCS è a[i-1] (= b[j-1]); gli elementi precedenti sono quelli di LCS(i-2, j-2), corrispondente a F[i-1] [j-1];
- F[i][j] = ↑: non vi è un elemento della LCS da prelevare, gli elementi precedenti sono quelli di LCS(i-1, j-2), corrispondente a F[i][j-1];
- F[i][j] = ←: non vi è un elemento della LCS da prelevare, gli elementi precedenti sono quelli di LCS(i-2, j-1), corrispondente a F[i-1][j];

Si noti che solo nel primo caso si mette nell'array-risultato (oppure si stampa) un elemento; negli altri due casi ci si sposta semplicemente "all'indietro" nella matrice.

La ricostruzione termina quando si raggiunge F[0][j] o F[i][0], corrispondenti alle sequenze iniziali vuote.

Esercizio

Si completi l'implementazione in Java, con le due versioni della ricostruzione della sequenza risultato:

- una versione iterativa che restituisce la sequenza sotto forma di array;
- una versione ricorsiva che scrive la sequenza sullo schermo;

Si realizzi l'implementazione in modo che le due sequenze di input e la sequenza risultato siano delle String; si usino a tale scopo il metodo e il costruttore della classe String che permettono di trasformare una stringa in un array di caratteri e viceversa (consultare la documentazione Java).

Soluzione dell'esercizio: versione iterativa

```
int lungLCS = L[m][n];
char[] LCS = new char[lungLCS];
int i = m;
int j = n;
int k = lungLCS - 1;

while(i > 0 && j > 0) {
  switch(F[i][j]) {
    case DIAG: LCS[k] = a[i-1];
               k--; i--; j--; break;
    case VERT: j--; break;
    case ORIZ: i--; break;
  }
}

return LCS;
```


Complessità

- tempo: $T(m, n) = \Theta(mn)$ (costruzione di matrici m×n); assumendo come al solito m ~ n, l'algoritmo è quadratico: molto meglio dell'algoritmo ingenuo esponenziale !
- spazio: $S(m, n) = \Theta(mn)$ (memorizzazione di matrici m×n); se si vuole calcolare solo la lunghezza della LCS, con le ottimizzazioni citate (cioè non tenendo in memoria l'intera matrice) si ottiene una complessità lineare: $S(m, n) = \Theta(n)$