Progettazione di Algoritmi **Anno Accademico 2021–2022**

Note per la Lezione 15

Ugo Vaccaro

In questa lezione studiamo il problema della più Lunga Sottosequenza Comune. Date due sequenze di simboli

$$a = a[1] \dots a[m], \quad b = b[1] \dots b[n]$$

il problema consiste nel trovare la piú lunga sottosequenza comune ad a e b; vale a dire, trovare una sottosequenza $a[i_1] \dots a[i_k]$ di a, una sottosequenza $b[j_1] \dots b[j_k]$ di b, tali che $i_1 < \dots < i_k$, $j_1 < \dots < j_k$, e

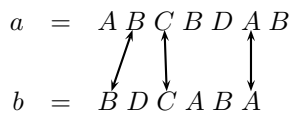
$$a[i_1] = b[j_1], \dots, a[i_k] = b[j_k]$$

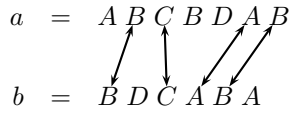
con k piú grande possibile.

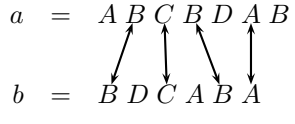
Denotiamo con $LCS(a, b)$ una tale piú lunga sottosequenza comune, e con $|LCS(a, b)|$ la sua lunghezza.

Perchè mai vorremmo risolvere questo problema? Perchè esso ha applicazioni in vari campi. Ad esempio in Biologia molecolare, In tale ambito, sequenze di DNA (geni) possono essere rappresentati come sequenze su di un alfabeto di quattro lettere {`A,C,G,T`}, corrispondenti alle quattro sottomolecole che formano il DNA. Quando i biologi trovano una nuova sequenza, in genere vogliono sapere a quali altre sequenze essa è simile, in quanto spesso sequenze simili hanno funzioni simili. Un modo di valutare come due sequenze sono simili è quello di determinare la lunghezza del loro più lunga sottosequenza comune, in quanto se la sottosequenza comune è "grande", allora e due sequanxe in questione condividono molti simpboli e quindi sno simili. Un altro ambito di applicazione della LCS è nel confronto di file. Il programma di Unix `diff` viene utilizzato per confrontare due diverse versioni dello stesso file per determinare quali modifiche sono state eventualmente apportate al file. Il programma `diff` funziona trovando la più lunga sottosequenza comune *delle linee* dei due file. In altri termini, è come se considerasse ciascuna linea come un singolo "carattere", e produce in output le linee che sono differenti nelle due versioni, ovvero il "complemento" della LCS delle due versioni.

Ad esempio, se $a = A\ B\ C\ B\ D\ A\ B$ e $b = B\ D\ C\ A\ B\ A$, avremmo







da cui $|LCS(a, b)| = 4$ e $LCS(a, b)$ è una qualunque tra B, C, B, A e B, C, A, B.

Ricordando la filosofia di base della tecnica Programmazione Dinamica, al fine di progettare un algoritmo che risolva il problema della più lunga sottosequenza comune, iniziamo dando una formulazione ricorsiva del calcolo della soluzione in termini di calcolo di sottosoluzioni ad opportuni sottoproblemi del problema di partenza. In altri termini, vogliamo esprimere la lunghezza della più lunga sottosequenza del problema originale in termini di lunghezze della più lunga sottosequenza comune di sottoproblemi del problema originale.

Date le sequenze

$$a = a[1] \dots a[m], \quad b = b[1] \dots b[n]$$

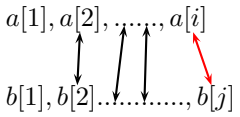
sia $c[i, j]$ la lunghezza di una più lunga sottosequenza comune di $a[1] \dots a[i]$ e $b[1] \dots b[j]$, dove $1 \leq i \leq m$ e $1 \leq j \leq n$. Definiamo inoltre

$$c[i, 0] = 0 = c[0, j], \quad \forall 0 \leq i \leq m,\ 0 \leq j \leq n.$$

Deriviamo, quindi, una equazione di ricorrenza per $c[i, j]$, ovvero, deriviamo una equazione che esprima $c[i, j]$ in termini di $c[p, t]$, dove p e t sono "più piccoli" di i e j.

Distinguiamo due casi:

- **Caso 1**: $a[i] \neq b[j]$.
  In tal caso, *non* é possibile che $a[i]$ e $b[j]$ siano *entrambi* presenti nella più lunga sottosequenza comune di $a[1] \dots a[i]$, e $b[1] \dots b[j]$ (altrimenti le sottosequenza scelte non sarebbero nemmeno *comuni!*)

  Visto che non è possibile che $a[i]$ e $b[j]$ siano entrambi presenti nella LCS(a, b), ne segue che una più lunga sottosequenza comune di $a[1] \dots a[i]$, e $b[1] \dots b[j]$ o non contiene $a[i]$ oppure non contiene $b[j]$. In altre parole, una tale più lunga sottosequenza comune o è una più lunga sottosequenza comune di

  $$a[1] \dots a[i-1], \text{ e } b[1] \dots b[j]$$

  oppure di

  $$a[1] \dots a[i], \text{ e } b[1] \dots b[j-1].$$

  E qual è delle due? Non lo sappiamo a priori, ma visto che ne cerchiamo una più lunga, sicuramente varrà

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

- **Caso 2**: $a[i] = b[j]$
  Sia $d[1] \dots d[k]$ una più lunga sottosequenza comune di

  $$a[1] \dots a[i], \text{ e } b[1] \dots b[j],$$

  e quindi vale anche $c[i, j] = k$. Allora affermiamo che $d[1], \dots, d[k-1]$ è una più lunga sottosequenza comune di

  $$a[1] \dots a[i-1], \text{ e } b[1] \dots b[j-1].$$

  Infatti, se per assurdo ne esistesse una comune α di lunghezza $\geq k$, allora appendendo alla fine di α il simbolo $a[i]$, otterremo una sottosequenza *comune* di

  $$a[1] \dots a[i], \text{ e } b[1] \dots b[j]$$

  di lunghezza $\geq k+1$, contraddicendo il fatto che $c[i, j] = k$. Abbiamo quindi provato che $c[i-1, j-1] = k-1$, ovvero

  $$c[i, j] = c[i-1, j-1] + 1.$$

Riassumendo, i valori $c[i,j]$ sono definiti dalla equazione di ricorrenza

$$c[i,j] = \begin{cases} 0 & \text{se } i=0 \text{ o se } j=0, \\ c[i-1,j-1]+1 & \text{se } i,j>0 \text{ e } a[i]=b[j] \\ \max\{c[i-1,j], c[i,j-1])\} & \text{se } i,j>0 \text{ e } a[i]\neq b[j]. \end{cases}$$

Un algoritmo di PD per il calcolo dei $c[i,j]$ basato sulle tecnica della memoization può essere il seguente:

```
Mem-Rec(i,j) %fà uso di una tabella c(i,j)
1. IF((i==0)||(j==0)) {
2.    RETURN 0
3.    } ELSE IF (c(i,j) non è definito) {
4.        IF (a[i]==b[j]) {
5.        c(i,j)= Mem-Rec(i-1,j-1)+1
6.        } ELSE {c(i,j)= max(Mem-Rec(i-1,j), Mem-Rec(i,j-1))
         }
      }
7. RETURN (c(i,j))
```

A noi interessa il valore ritornato da `Mem-Rec(m,n)`. Poichè ogni entrata della tabella `c(m,n)` viene calcolata esattamente una sola volta, e per il suo calcolo l'algoritmo impiega tempo costante, vien fuori che l'algoritmo `Mem-Rec(m,n)` impiega tempo $O(mn)$.

Supponendo che le due sequenze siano a=GDVEGTA e b =GVCEKST, la tabella $c[\cdot,\cdot]$ risultante sarebbe la seguente:

			G	V	C	E	K	S	T
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
G	1	0	1	1	1	1	1	1	1
D	2	0	1	1	1	1	1	1	1
V	3	0	1	2	2	2	2	2	2
E	4	0	1	2	2	3	3	3	3
G	5	0	1	2	2	3	3	3	3
T	6	0	1	2	2	3	3	3	4
A	7	0	1	2	2	3	3	3	4

L'algoritmo appena visto calcola $c[m,n] = |LCS(a,b)|$. Come fare a calcolare la più lunga sottosequenza comune ad a e b? Possiamo usare la tabella $c[\cdot,\cdot]$ prima calcolata, ragionando nel modo seguente:

- Se $c[m,n]=0$ allora ritorniamo la stringa vuota,
- altrimenti, se $c[m,n]=c[m-1,n]$, allora calcoliamo (ricorsivamente) la LCS tra $a[1]\dots a[m-1]$ e $b[1]\dots b[n]$,
- se $c[m,n]=c[m,n-1]$, allora calcoliamo (ricorsivamente) la LCS tra $a[1]\dots a[m]$ e $b[1]\dots b[n-1]$,
- Se entrambe le condizioni di sopra sono false, allora vuol dire che $a[m]=b[n]$, di conseguenza l'algoritmo stampa $a[m]$ e ricorre su $a[1]\dots a[m-1]$ e $b[1]\dots b[n-1]$

Date le due sequenze $a = a[1] \dots a[m]$ e $b = b[1] \dots b[n]$, denotiamo con $a(i) = a[1] \dots a[i]$ e $b(j) = b[1] \dots b[j]$, per $1 \leq i \leq m$, $1 \leq j \leq n$. Un algoritmo per il calcolo di LCS(a, b) può essere il seguente:

```
LCS_print(a,b,c)   % prende in input le sequenze a,b e la matrice c[,]
1. IF(c[m,n]==0) {
2.   RETURN ()
3.   } ELSE {
4.     IF(c[m,n]==c[m-1,n]) {
5.       LCS_print(a(m-1),b,c)
6.     } ELSE {IF (c[m,n]=c[m,n-1]) {
7.       LCS_print(a,b(n-1),c)
8.     } ELSE {
9.      LCS_print(a(m-1),b(n-1),c)
10.     print(a[m])
        }
     }
```

L' algoritmo `LCS_print(a,b,c)` ad ogni chiamata ricorsiva decrementa n, oppure decrementa m, o addirittura entrambi. Di conseguenza, termina in tempo $O(n + m)$

$\diamond$

Consideriamo ora un altro problema tra sequenze, che và sotto il nome di del calcolo della distanza di edit tra sequenze. Esso può essere formulato nel modo seguente. Date due sequenze di lettere $s = s[1] \dots s[m]$ e $t = t[1] \dots [n]$, la distanza di edit tra s e t (denotata con $\texttt{dist}(s, t)$) è pari al minimo numero di inserzioni di lettere, cancellazioni di lettere, o sostituzioni di lettere necessarie per trasformare s in t. Anche la distanza di edit tra sequanze può essere interpretata come una misura di similarità tra sequenze, e trova varie applicazioni, ad esempio nella correzione automatica nei text editor.

Vediamo un esempio. Vogliamo trasformare la sequenza di lettere `presto` in `peseta`. Potremmo procedere nel seguente modo:

$$\texttt{presto} \overset{\text{Cancella } \texttt{r}}{\longrightarrow} \texttt{pesto} \overset{\text{Inserisci } \texttt{e}}{\longrightarrow} \texttt{peseto} \overset{\text{Sostituisci } \texttt{o}}{\longrightarrow} \texttt{peseta}$$

Quindi, nell'esempio di sopra, con l'uso di 3 operazioni si riesce a trasformare `presto` in `peseta`. Si vede agevolmente che non è possibile utilizzare un numero di operazioni inferiore a 3, per cui $\texttt{dist}(\texttt{presto}, \texttt{peseta}) = 3$.

Osserviamo che potremmo sempre trasformare una qualsiasi sequenza $s = s[1] \dots s[m]$ in una qualsiasi $t = t[1] \dots [n]$ cancellando tutte le m lettere di s ed inserendo una ad una tutte le n lettere di t (ciò mostra che $\texttt{dist}(s, t) \leq n + m$). Il nostro problema è calcolare *esattamente* $\texttt{dist}(s, t)$, utilizzanzo la tecnica della Programmazione Dinamica.

A tale scopo, ricordiamo il primo passo della tecnica PD, e chiediamoci: Chi sono i sottoproblemi del problema di calcolare $\texttt{dist}(s, t)$? Essi corrispondono naturalmente al calcolo delle distanze di edit tra *sottosequenze* di s e t di lunghezze inferiori rispetto a quelle di s e t.

Sia $\texttt{dist}(s[1] \dots s[i], t[1] \dots t[j])$, per $1 \leq i < m$ e $1 \leq j < n$, la distanza di edit tra le sottosequenze $s[1] \dots s[i]$, e $t[1] \dots t[j]$, rispettivamente. La Programmazione Dinamica ci suggerisce di determinare una equazione di ricor-

renza per le quantità $\texttt{dist}(s[1]\dots s[i], t[1]\dots t[j])$, per $1 \le i < m$ e $1 \le j < n$, indi di calcolarle, per valori di i e j crescenti a partire dai casi base, memorizzando i valori intermedi in una tabella.

L'equazione di ricorrenza per $\texttt{dist}(s[1]\dots s[i], t[1]\dots t[j])$ la possiamo ottenere effettuando le seguenti considerazioni. Vogliamo calcolare il minimo numero di operazioni $\texttt{dist}(s[1]\dots s[i], t[1]\dots t[j])$ per trasformare $s[1]\dots s[i]$ in $t[1]\dots t[j]$. Iniziamo dalla fine: come può l'ultima lettera di $s[1]\dots s[i]$, ovvero $s[i]$, trasformarsi in $t[j]$, l'ultima lettera di $t[1]\dots t[j]$? Possiamo usare una delle tre seguenti operazioni:

- Sostituire $s[i]$ con $t[j]$.
  Ciò ci lascia poi con il problema di trasformare la sottosequenza $s[1]\dots s[i-1]$ in $t[1]\dots t[j-1]$, che richiederà ulteriori $\texttt{dist}(s[1]\dots s[i-1], t[1]\dots t[j-1])$ operazioni. In totale, useremo
  $$1 + \texttt{dist}(s[1]\dots s[i-1], t[1]\dots t[j-1])$$
  operazioni.
- Cancellare $s[i]$ e poi trasformare $s[1]\dots s[i-1]$ in $t[1]\dots t[j]$.
  Ciò richiederà $\texttt{dist}(s[1]\dots s[i-1], t[1]\dots t[j])$ operazioni. In totale, useremo
  $$1 + \texttt{dist}(s[1]\dots s[i-1], t[1]\dots t[j])$$
  operazioni.
- Inserire $t[j]$ alla fine di $s[1]\dots s[i]$.
  Ciò ci lascia poi con il problema di trasformare $s[1]\dots s[i]$ in $t[1]\dots t[j-1]$, che richiederà un numero di operazioni pari a $\texttt{dist}(s[1]\dots s[i], t[1]\dots t[j-1])$. In totale, useremo
  $$1 + \texttt{dist}(s[1]\dots s[i], t[1]\dots t[j-1])$$
  operazioni.

Vi è un caso speciale da considerare. Se $s[i] = t[j]$, allora nel primo passo sopra considerato non occorre trasformare $s[i]$ in $t[j]$, e quindi basteranno $\texttt{dist}(s[1]\dots s[i-1], t[1]\dots t[j-1])$ operazioni per trasformare $s[1]\dots s[i]$ in $t[1]\dots t[j]$. Per trattare questo caso, definiamo la quantità $\texttt{diff}(x, y) = 1$ se $x \neq y$, 0 altrimenti.

Mettendo tutto insieme, e visto che intendiamo trasformare $s[1]\dots s[i]$ in $t[1]\dots t[j-1]$ con il *minor* numero di operazioni, abbiamo la seguente equazione di ricorrenza per $\texttt{dist}(s[1]\dots s[i], t[1]\dots t[j])$ che cercavamo, $\forall\, i, j \ge 1$:

$$\begin{aligned}\texttt{dist}(s[1]\dots s[i], t[1]\dots t[j]) \;=\; \min\{&\texttt{dist}(s[1]\dots s[i-1], t[1]\dots t[j-1]) + \texttt{diff}(s[i], t[j]),\\ &\texttt{dist}(s[1]\dots s[i-1], t[1]\dots t[j]) + 1,\\ &\texttt{dist}(s[1]\dots s[i], t[1]\dots t[j-1]) + 1\}\end{aligned}$$

Per i casi base $i = 0 = j$, effettueremo la ovvia assunzione che $s[0] = t[0] =$*sequenza vuota*, e che quindi

$$\texttt{dist}(s[0], t[1]\dots t[j]) = j, \texttt{dist}(s[1]\dots s[i], t[0]) = i, \quad \text{e} \quad \forall\, i, j \ge 1.$$

L'algoritmo di Programmazione Dinamica (nella versione iterativa)[1] per il calcolo di

$$\texttt{dist}(s, t) = \texttt{dist}(s[1]\dots s[m], t[1]\dots t[n])$$

sarà quindi:

[1] Un utile esercizio è darne la versione ricorsiva basata sulla tecnica della Memoization

```
Edit_distanza(s,t)   %Fà uso di una matrice dist[.,.]
1. FOR (i=0, i<m+1, i=i+1) {
2.     dist[i,0]= i % (inizializzazione prima colonna)
   }
3.   FOR (j= 0, j<n+1, j=j+1) {
4.     dist[0,j]= j % (inizializzazione prima riga)
   }
5.   FOR (i=1, i<m+1, i=i+1) {
6.     FOR (j= 1, j<n+1, j=j+1) {
7.       IF (s[i]==t[j]) {    % (calcolo di dist[i,j] )
8.           dist[i,j]=min(dist[i-1,j-1], dist[i-1,j]+1, dist[i,j-1]+1)
9.         } ELSE { dist[i,j]= min(dist[i-1,j-1]+1, dist[i-1,j]+1, dist[i,j-1]+1)
         }
       }
10. RETURN dist[m,n]
```

Per l'analisi dell'algoritmo, notiamo che il `FOR` sulle linee 1. e 2. prende tempo $O(m)$, il `FOR` sulle linee 3. e 4. prende tempo $O(n)$, le istruzioni sulle linee 7-9 prendono tempo $O(1)$, il `FOR` sulle linee 6-9 prende tempo $O(n)$, il `FOR` sulle linee 5-9 prende tempo $O(nm)$. In totale, l'algoritmo `Edit_distanza`(s,t) prende tempo $O(nm)$.

Vediamo un esempio di applicazione dell'algoritmo appena visto. Sia a =`presto`, b =`peseta`. L'algoritmo costruirebbe la seguente matrice. Il quadrato colorato a fianco ricorda che ogni nuova entrata della matrice (in blu) viene calcolato sulla base dei valori contenuti nelle tre entrate in grigio, precedentemente calcolate.

		0	1	2	3	4	5	6
			p	e	s	e	t	a
0		0	1	2	3	4	5	6
1	p	1	0	1	2	3	4	5
2	r	2	1	1	2	3	4	5
3	e	3	2	1	2	2	3	4
4	s	4	3	2	1	2	3	4
5	t	5	4	3	2	2	2	3
6	o	6	5	4	3	3	3	3