Progettazione di Algoritmi - lezione 21

Discussione dell'esercizio [prezzi]

Dato un vettore *P* di *n* interi positivi in cui *P*[*t*] rappresenta il prezzo di una certa merce nel giorno *t*, si vuole sapere qual'è il giorno *i* in cui conviene comprare la merce ed il giorno *j*, con *j* > *i*, in cui conviene rivenderla in modo da massimizzare il profitto o in alternativa minimizzare la perdita. In altre parole siamo interessati a conoscere la coppia di giorni (*i*, *j*) con *i* < *j* per cui risulta massimo il valore *P*[*j*] − *P*[*i*].

Si potrebbe pensare di cercare la posizione *j* del massimo del vettore *P* e poi trovare il minimo delle posizioni che precedono *j*. Simmetricamente, si potrebbe cercare la posizione *i* del minimo e poi il massimo tra le posizioni che seguono *i*. Ma, come mostra l'esempio qui sotto, entrambi gli approcci possono fallire (contemporaneamente):

```
P:    5   10   3   9   2   7
```

Quindi proviamo con la Programmazione Dinamica. Trattandosi di un vettore, consideriamo come sotto-problemi quelli relativi ai prefissi del vettore: per ogni *k* = 2,…,*n*

$$M[k] = \text{massima differenza } P[j] - P[i] \text{ per } 1 \leqslant i < j \leqslant k$$

C'è però una difficoltà, il calcolo di *M*[*k*] richiede di considerare due casi: o la massima differenza non usa *P*[*k*] e questa è allora uguale a *M*[*k* - 1], oppure usa *P*[*k*]. Ma in quest'ultimo caso dovremmo conoscere il minimo delle posizioni che precedono *k* e il calcolo di ciò porterebbe ad un algoritmo di complessità quadratica. Dobbiamo modificare la definizione dei sotto-problemi in modo analogo a come abbiamo fatto per il problema del massimo sottovettore:

$$M[k] = \text{massima differenza } P[k] - P[i] \text{ per } 1 \leqslant i < k$$

Però, riflettendoci un po', ci accorgiamo che è sufficiente il calcolo del prezzo minimo del prefisso:

$$M[k] = \text{minimo prezzo del prefisso } P[1 \cdots k-1]$$

Così la massima differenza del tipo *P*[*k*] - *P*[*i*] per 1 ≤ *i* < *k*, è *P*[*k*] - *M*[*k*]. Il caso base è *M*[2] = *P*[1] e il calcolo per *k* = 3,…,*n* è facilissimo:

$$M[k] = \min\{P[k-1], M[k-1]\}$$

Il programma non deve necessariamente mantenere l'intera tabella *M*, basta l'ultimo elemento:

```
PREZZI(P: array di n prezzi)
    m <- P[1]
    max <- P[2] - m
    FOR k <- 3 TO n DO
        m <- min{P[k - 1], m}
        IF P[k] - m > max THEN
            max <- P[k] - m
    RETURN max
```

Se vogliamo calcolare il giorno di acquisto *i* e quello di vendita *j* basterà registrare gli ultimi aggiornamenti di *m* e *max*:

```
PREZZI_SOL(P: array di n prezzi)
    im <- 1       /* Indice del minimo */
    i <- 1        /* Giorno d'acquisto */
    j <- 2        /* Giorno di vendita */
    FOR k <- 3 TO n DO
        IF P[im] > P[k - 1] THEN
            im <- k - 1
        IF P[k] - P[im] > P[j] - P[i] THEN
            i <- im
            j <- k
    RETURN i, j
```

Ovviamente entrambi i programmi hanno complessità ottimale O(*n*).

Il problema della massima sottosequenza comune (LCS)

La maggior parte degli strumenti per confrontare file, come ad esempio l'utility *diff*, sono basati sul calcolo della massima sottosequenza comune dei file. Nell'esempio qui sotto i caratteri della massima sottosequenza comune delle due linee di testo sono stati sottolineati:

La massima sotto-sequenza in comune.

Il problema della Massima Sottosequenza Comune.

La formulazione del problema è la seguente:

> Date due sequenze $x = x[1],\ldots,x[n]$ e $y = y[1],\ldots,y[m]$, trovare una sottosequenza comune di lunghezza massima.

In inglese è conosciuto con il nome di *Longest Common Subsequence* (*LCS*) problem. Una sottosequenza comune (di lunghezza k) è data da una sequenza di indici di x, $1 \le i_1 < i_2 < \cdots < i_k \le n$, e una sequenza di indici di y, $1 \le j_1 < j_2 < \cdots < j_k \le m$, tali che

$$x[i_1] = y[j_1],\ x[i_2] = y[j_2], \ldots,\ x[i_k] = y[j_k]$$

È chiaro che una ricerca esaustiva prende tempo esponenziale in *n* e *m*. Infatti tutte le possibili sottosequenze di *x* sono esattamente 2^n (se consideriamo anche la sottosequenza vuota) perchè i sottoinsiemi di {1,2,…,*n*} corrispondono biunivocamente alle sottosequenze di *x* (ogni sottoinsieme è la sequenza degli indici di una sottosequenza). La stessa cosa vale ovviamente per *y* che ha quindi 2^m sottosequenze.

Come al solito, preoccupiamoci di calcolare solamente la massima lunghezza di una sottosequenza comune e poi vedremo come ricostruire una tale sottosequenza. Volendo applicare la Programmazione Dinamica, dobbiamo individuare i sotto-problemi da considerare. Una possibilità che viene subito in mente sono i prefissi delle due sequenze, cioè consideriamo i sotto-problemi relativi ai prefissi $x[1 \cdots i]$ e $y[1 \cdots j]$ per $i = 0, \ldots, n$ e $j = 0, \ldots, m$:

$$L[i,j] = \text{massima lunghezza di una sottosequenza comune a } x[1 \cdots i] \text{ e } y[1 \cdots j]$$

dove $x[1 \cdots 0]$ e $y[1 \cdots 0]$ denotano i prefissi vuoti. Chiaramente il valore che ci interessa è $L[n,m]$. I casi base si hanno quando $i = 0$ o $j = 0$: per $i = 0, \ldots, n$ e $j = 0, \ldots, m$,

$$L[0,j] = 0 \quad \text{e} \quad L[i,0] = 0$$

Vediamo come calcolare $L[i,j]$ quando sia i che j sono almeno 1. Cerchiamo di ricondurre tale calcolo a valori di L per prefissi più piccoli. Una massima sottosequenza comune a $x[1 \cdots i]$ e $y[1 \cdots j]$ come può essere fatta? Se non

comprende l'ultimo elmento, $x[i]$, di $x[1 \cdots i]$, $L[i,j]$ è uguale a $L[i-1,j]$. Simmetricamente, se non comprende $y[j]$, $L[i,j]$ è uguale a $L[i,j-1]$. Altrimenti comprende sia $x[i]$ che $y[j]$. Ma questo è possibile solo se $x[i] = y[j]$. In quest'ultimo caso la sottosequenza è formata da una sottosequenza comune a $x[1 \cdots i-1]$ e $y[1 \cdots j-1]$ e dall'elemento $x[i] = y[j]$. Quindi $L[i,j]$ è uguale a $L[i-1,j-1]+1$. Ricapitolando abbiamo determinato il seguente metodo di calcolo: per $i = 1, \ldots, n$ e $j = 1, \ldots, m$

$$L[i,j] = \begin{cases} \max\{L[i-1,j],\ L[i,j-1],\ L[i-1,j-1]+1\} & \text{se } x[i] = y[j] \\ \max\{L[i-1,j],\ L[i,j-1]\} & \text{altrimenti} \end{cases}$$

È facile vedere che se *x*[*i*] = *y*[*j*], allora

$$L[i-1,j] \leqslant L[i-1,j-1]+1 \quad \text{e} \quad L[i,j-1] \leqslant L[i-1,j-1]+1$$

Quindi il calcolo di *L*[*i*, *j*] può essere leggermente semplificato:

$$L[i,j] = \begin{cases} L[i-1,j-1]+1 & \text{se } x[i] = y[j] \\ \max\{L[i-1,j],\ L[i,j-1]\} & \text{altrimenti} \end{cases}$$

Il programma che calcola la tabella procede dai prefissi più corti verso quello più lunghi:

```
LCS(x: sequenza lunga n, y: sequenza lunga m)
    L: tabella (n+1)x(m+1)
    FOR i <- 0 TO n DO L[i, 0] <- 0
    FOR j <- 0 TO m DO L[0, j] <- 0
    FOR i <- 1 TO n DO
        FOR j <- 1 TO m DO
             IF x[i] = y[j] THEN
                 L[i, j] <- L[i-1, j-1] + 1
             ELSE
                 L[i, j] <- max{L[i-1, j], L[i, j-1]}
    RETURN L[n, m]
```

La complessità dell'algoritmo è O(*nm*). Se siamo interessati solamente alla lunghezza della massima sottosequenza, è sufficiente mantenere solamente le ultime due righe della tabella *L*, quindi possiamo usare memoria O(*m*), anziché O(*nm*). Se invece si vuole ricostruire la sottosequenza comune, come al solito, possiamo percorrere la tabella seguendo a ritroso le scelte che hanno portato a calcolare la lunghezza massima.

```
LCS_SOL(x: sequenza lunga n, y: sequenza lunga m, L: tabella)
    SOL <- lista vuota
    i <- n
    j <- m
    WHILE i > 0 AND j > 0 DO
        IF x[i] = y[j] THEN
            SOL <- (i, j, X[i]) + SOL   /* Aggiunge in testa alla lista SOL */
            i <- i - 1
            j <- j - 1
        ELSE IF L[i, j] = L[i - 1, j] THEN
            i <- i - 1
        ELSE
            j <- j - 1
    RETURN SOL
```

La lista ritornata contiene per ogni elemento della sottosequenza l'indice di *x*, l'indice di *y* e l'elemento. La complessità è O(*n* + *m*) perchè ad ogni iterazione del `WHILE` uno almeno dei due indici `i` , `j` è decrementato.

Pensando a una delle principali applicazioni della massima sottosequenza comune, il confronto di file, sorge

spontanea la domanda di come possa essere calcolata nella pratica per file di dimensione medio grande. Ad esempio, sei i due file hanno dimensione 100KB = 10^5B, la memoria richiesta dalla tabella è di 10^{10}B = 10GB. C'è un raffinamento dell'algoritmo che abbiamo visto, chiamato algoritmo di Hirschberg, che trova una sottosequenza comune di lunghezza massima usando spazio lineare O(min{*n*, *m*}) e tempo O(*nm*).

Edit Distance

La massima sottosequenza comune, LCS, è un modo di misurare la vicinanza o distanza di due sequenze o stringhe. Più la LCS è lunga e più sono vicine le due stringhe. Vista come distanza tra stringhe, la LCS è un po' strana perché cresce al diminuire della distanza, quando le due stringhe sono uguali raggiunge la massima lunghezza e quando sono massimamente distanti è di lunghezza zero. Però c'è un modo di vedere la LCS di due stringhe x e y che la rende una vera distanza. Partiamo dalla stringa x ed eliminiamo tutti gli elementi di x che non fanno parte della LCS. Questi sono $|x| - LCS(x, y)$, dove $|x|$ denota la lunghezza di x e $LCS(x, y)$ è la lunghezza della LCS di x e y. Poi inseriamo, a partire dalla LCS, tutti gli elementi che permettono di formare la stringa y. Questi sono $|y| - LCS(x, y)$. In totale abbiamo effettuato $|x| - LCS(x, y) + |y| - LCS(x, y) = |x| + |y| - 2LCS(x, y)$ operazioni di eliminazione ed inserimento per passare da x a y. Ad esempio, per trasformare `ALBERO` in `LIBRI`:

```
ALBERO      eliminiamo A E O
 LBR        LCS di ALBERO e LIBRI
LIBRI       inseriamo I I
```

Vale anche il viceversa, cioè il numero minimo di operazioni di inserimento ed eliminazione per trasformare x in y è uguale a $|x| + |y| - 2LCS(x, y)$. Infatti, siccome non avrebbe senso eliminare un elemento e poi reinserirlo o inserire un elemento e poi eliminarlo, si può pensare di effettuare tutte le operazioni di eliminazione prima di tutte quelle di inserimento. Quindi, ciò che rimane dopo aver fatto tutte le eliminazioni deve necessariamente essere la LCS di x e y perché se fosse più corta allora potremmo risparmiare qualche eliminazione. Quindi, denotando con

$$d(x, y) = \text{numero minimo di eliminazioni ed inserimenti per trasformare } x \text{ in } y$$

abbiamo che

$$d(x, y) = |x| + |y| - 2LCS(x, y)$$

Relativamente all'esempio, $d($`ALBERO`, `LIBRI`$) = 5$. È facile vedere che questa è proprio una distanza o metrica tra stringhe dato che soddisfa le proprietà: per tutte le stringhe x, y e w,

$$\begin{aligned}
&d(x, y) \geqslant 0 && \text{(non negatività)}\\
&d(x, y) = 0 \Leftrightarrow x = y && \text{(identità degli indiscernibili)}\\
&d(x, y) = d(y, x) && \text{(simmetria)}\\
&d(x, y) \leqslant d(x, w) + d(w, y) && \text{(disuguaglianza triangolare)}
\end{aligned}$$

Eliminazioni ed inserimenti sono operazioni eseguite tipicamente quando si modifica o edita un testo. Un'altra operazione comune è la sostituzione di un elemento con un altro. Se aggiungiamo anche la possibilità di effettuare operazioni di sostituzione otteniamo un'altra distanza tra stringhe nota come *distanza di Levenshtein* o più comunemente *edit distance*:

$$edit(x, y) = \text{numero minimo di eliminazioni, inserimenti e sostituzioni per trasformare } x \text{ in } y$$

Relativamente alle stringhe `ALBERO` e `LIBRI` abbiamo

```
ALBERO      eliminiamo A E
LIBRO       inseriamo I
LIBRI       sostituiamo O con I
```

Quindi $edit($`ALBERO`, `LIBRI`$) = 4$. Si può dimostrare che anche $edit(\cdot, \cdot)$ è una distanza tra stringhe. Il calcolo della edit distance ha varie applicazioni. Nello string matching approssimato per trovare le migliori occorrenze

approssimate di una stringa in un testo, nei controllori ortografici per trovare la o le parole del dizionario più vicine a una stringa, in alcuni sistemi di traduzione automatica (*translation memories*) per trovare segmenti di testo che approssimano un dato segmento in una base di dati di segmenti dotati di traduzione.

Data la parentela con la LCS non sorprende che anche per il calcolo della edit distance la Programmazione Dinamica permetta di ottenere un algoritmo efficiente. Parimenti non sorprende che per calcolare la edit distance di due stringhe x e y i sotto-problemi da considerare sono i prefissi delle due stringhe: per ogni $i = 0, \ldots, |x|$ e $j = 0, \ldots, |y|$,

$$E[i,j] = \text{edit distance dei prefissi } x[1 \cdots i] \text{ e } y[1 \cdots j]$$

I casi base relativamente ai prefissi vuoti sono immediati: per ogni $i = 0, \ldots, |x|$ e $j = 0, \ldots, |y|$,

$$E[0,j] = j \quad \text{e} \quad E[i,0] = i$$

Come calcolare $E[i,j]$ in funzione delle distanze relative ai prefissi più piccoli? Pensiamo a una possibile sequenza di operazioni che trasformano $x[1 \cdots i]$ in $y[1 \cdots j]$. Chiediamoci come può essere stato prodotto l'ultimo elemento di $y[1 \cdots j]$, cioè $y[j]$. Sono possibili tre casi.

1. $y[j]$ è derivato direttamente da un elemento (di eguale valore) di $x[1 \cdots i]$: questo caso si suddivide in due sottocasi.
   a. $x[i] = y[j]$: allora $E[i,j] = E[i-1,j-1]$
   b. $x[i] \neq y[j]$: allora $y[j]$ non può derivare da $x[i]$ per cui $x[i]$ è stato eliminato. Quindi $E[i,j] = E[i-1,j] + 1$, dove il $+1$ è dovuto all'eliminazione.
2. $y[j]$ è stato inserito: allora $E[i,j] = E[i,j-1] + 1$, dove il $+1$ è dovuto all'inserimento di $y[j]$.
3. $y[j]$ è stato prodotto per sostituzione: si può assumere che sia stato prodotto sostituendo $x[i]$, quindi $E[i,j] = E[i-1,j-1] + 1$.

Riassumendo, $E[i,j]$ deve essere uguale a uno dei seguenti valori:

$$E[i-1,j-1] \text{ (solo se } x[i] = y[j]) \quad \vee \quad E[i-1,j] + 1 \quad \vee \quad E[i,j-1] + 1 \quad \vee \quad E[i-1,j-1] + 1$$

Inoltre, ognuna di queste possibilità è sempre realizzabile. Infatti,

1. Se $x[i] = y[j]$, allora $E[i,j] \leq E[i-1,j-1]$ dato che le operazioni che trasformano $x[1 \cdots i-1]$ in $y[1 \cdots j-1]$ trasformano anche $x[1 \cdots i]$ in $y[1 \cdots j]$.
2. $E[i,j] \leq E[i-1,j] + 1$, perché basta eliminare $x[i]$ e poi trasformare $x[1 \cdots i-1]$ in $y[1 \cdots j]$.
3. $E[i,j] \leq E[i,j-1] + 1$, perchè dopo aver trasformato $x[1 \cdots i]$ in $y[1 \cdots j-1]$ basta inserire $y[j]$.
4. $E[i,j] \leq E[i-1,j-1] + 1$, dato che possiamo trasformare $x[1 \cdots i-1]$ in $y[1 \cdots j-1]$ e poi sostituire $x[i]$ con $y[j]$.

Tenendo conto di tutto ciò, concludiamo che

$$E[i,j] = \begin{cases} \min\{E[i-1,j-1],\ E[i-1,j]+1,\ E[i,j-1]+1\} & \text{se } x[i] = y[j] \\ 1 + \min\{E[i-1,j-1],\ E[i-1,j],\ E[i,j-1]\} & \text{altrimenti} \end{cases}$$

Questa regola di calcolo può essere semplificata perchè si può dimostrare che vale sempre:

$$E[i-1,j-1] \leqslant \min\{E[i-1,j]+1,\ E[i,j-1]+1\}$$

Quindi abbiamo che

$$E[i,j] = \begin{cases} E[i-1,j-1] & \text{se } x[i] = y[j] \\ 1 + \min\{E[i-1,j-1],\ E[i-1,j],\ E[i,j-1]\} & \text{altrimenti} \end{cases}$$

A questo punto possiamo scrivere il programma che calcola la edit distance di due stringhe:

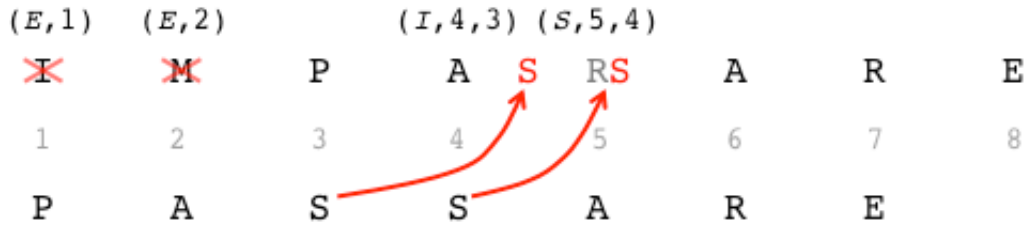
```
EDIT(x: stringa di lunghezza n, y: stringa di lunghezza m)
    E: tabella (n+1)x(m+1)
    FOR i <- 0 TO n DO E[i, 0] <- i
    FOR j <- 0 TO m DO E[0, j] <- j
    FOR i <- 1 TO n DO
        FOR j <- 1 TO m DO
            IF x[i] = y[j] THEN
                E[i, j] <- E[i-1, j-1]       /* Nessuna operazione */
            ELSE
                E[i, j] <- E[i-1, j-1] + 1       /* Sostituzione */
                IF E[i, j] > E[i-1, j] + 1 THEN
                    E[i, j] <- E[i-1, j] + 1     /* Eliminazione */
                IF E[i, j] > E[i, j-1] + 1 THEN
                    E[i, j] <- E[i, j-1] + 1     /* Inserimento */
    RETURN E[n, m]
```

Chiaramente la complessità è O(nm). Se non si è interessati a conoscere le operazioni che trasformano una stringa nell'altra è possibile risparmiare memoria perché basta mantenere le ultime due righe della tabella quindi memoria O(m) (o comunque O($\min\{n, m\}$), eventualmente scambiando x con y). Se invece si vogliono determinare le operazioni ottimali che permettono di trasformare x in y, come al solito, basta percorrere la tabella E seguendo a ritroso le scelte che sono state fatte per calcolarla.

```
EDIT_SOL(x: stringa di lunghezza n, y: stringa di lunghezza m, E: tabella)
    SOL <- lista vuota
    i <- n
    j <- m
    WHILE i > 0 OR j > 0 DO
        IF i = 0 THEN
            SOL <- SOL + (I, 0, j)    /* Inserimento dopo posizione 0 di y[j] */
            j <- j - 1
        ELSE IF j = 0 THEN
            SOL <- SOL + (E, i)       /* Eliminazione elemento in posizione i */
            i <- i - 1
        ELSE
            IF x[i] = y[j] AND E[i, j] = E[i-1, j-1] THEN
                i <- i - 1
                j <- j - 1
            ELSE IF E[i, j] = E[i-1, j-1] + 1 THEN
                SOL <- SOL + (S, i, j)  /* Sostituzione elemento in posizione i con y[j] */
                i <- i - 1
                j <- j - 1
            ELSE IF E[i, j] = E[i-1, j] + 1 THEN
                SOL <- SOL + (E, i)     /* Eliminazione elemento in posizione i */
                i <- i - 1
            ELSE
                SOL <- SOL + (I, i, j)  /* Inserimento dopo posizione i di y[j] */
                j <- j - 1
    RETURN SOL
```

Data la tabella E, la costruzione della lista delle operazioni prende tempo O($n + m$). Ecco un esempio per le stringhe $x =$ `IMPARARE` e $y =$ `PASSARE`:

	j	0	1	2	3	4	5	6	7	
i			**P**	**A**	**S**	**S**	**A**	**R**	**E**	
0		**0**	1	2	3	4	5	6	7	
1	**I**	**1**	1	2	3	4	5	6	7	*Eliminazione* 1
2	**M**	**2**	2	2	3	4	5	6	7	*Eliminazione* 2
3	**P**	3	**2**	3	3	4	5	6	7	
4	**A**	4	3	**2**	**3**	4	4	5	6	*Inserimento* (4, 3)
5	**R**	5	4	3	3	**4**	5	4	5	*Sostituzione* (5, 4)
6	**A**	6	5	4	4	4	**4**	5	5	
7	**R**	7	6	5	5	5	5	**4**	5	
8	**E**	8	7	6	6	6	6	5	**4**	



Un raffinamento simile a quello per il calcolo della LCS permette di ottenere la sequenza ottimale di operazioni usando solamente memoria O($\min\{n,m\}$) e tempo O(nm).

Altre operazioni oltre a inserimento, eliminazione e sostituzione sono importanti. Ad esempio, aggiungendo l'operazione di trasposizione di due elementi adiacenti si ottiene la cosidetta *distanza di Damerau–Levenshtein* che ha applicazioni anche in biologia per confrontare sequenze di DNA. La distanza di Damerau–Levenshtein è calcolabile con un algoritmo simile a quello per la edit distance e quindi anch'esso basato sulla programmazione dinamica.

Esercizio [senza spazi]

Abbiamo una stringa *t* di *n* caratteri che sospettiamo possa essere un testo da cui sono stati eliminati tutti gli spazi e la punteggiatura. Ad esempio potrebbe essere `"questaèunafrasesenzaspazi"`. Abbiamo a disposizione una subroutine `DICT` che prende in input una stringa *s* e ritorna *true* se *s* è una parola, *false* altrimenti. Se *t* è la stringa d'esempio, `DICT(t[1…6])` ritorna *true* mentre `DICT(t[7…10])` ritorna *false*. Vogliamo un algoritmo efficiente che usando la subroutine `DICT` (che si assume abbia costo O(1)) permetta di determinare se una stringa *t* è una concatenazione di parole.