Capitolo 5

PROGRAMMAZIONE DINAMICA

Le tecniche di programmazione dinamica si applicano a certi problemi di ottimizzazione, cioè problemi che richiedono di massimizzare o minimizzare certe quantità, che si possono spezzare in sottoproblemi. Puó capitare che lo stesso sottoproblema compaia più volte nella scomposizione del problema principale. A differenza delle strategie di tipo *Divide et Impera*, che risolverebbero piú volte lo stesso sottoproblema, l'idea in un algoritmo di programmazione dinamica è di risolvere prima i sottoproblemi base (strategia bottom-up) mantenendo in memoria i risultati parziali ottenuti, e quindi passare alla risoluzione a livelli sempre più alti utilizzando i risultati già in memoria. La tecnica della programmazione dinamica è stata introdotta nel 1957 da Bellman.

5.1 Prodotto in cascata di matrici

Si supponga di voler moltiplicare fra loro una serie di matrici. Il numero di prodotti scalari da effettuare varia, e puó variare anche di molto, a seconda dell'ordine in cui si effettuano le moltiplicazioni. Per esempio, se per moltiplicare una matrice A 4x2 con B 2x3 e con C 3x1 si fa il prodotto ((AxB)xC), si calcolano 24+12 prodotti scalari. Se invece si sceglie di procedere da destra verso sinistra, cioé facendo (Ax(BxC)), i prodotti saranno 6+16. Vale quindi la pena di perdere un po' di tempo alla ricerca di una buona sequenza di moltiplicazione.

Si supponga di voler moltiplicare N matrici $M_1 M_2 M_3 ... M_N$, dove la matrice M_i abbia r_i righe e r_{i+1} colonne. Si cercherà la sequenza di moltiplicazione (la parentesizzazione completa) che minimizza il numero dei prodotti scalari.

Per comodità si scriva $M_{i..j}$ per indicare la matrice che risulta dal prodotto $M_i M_{i+1}$ ... M_j. Si utilizzerà una matrice m: in m[i,j] si memorizzerà il minimo numero di moltiplicazioni necessarie per calcolare $M_{i..j}$. I valori di m sulla diagonale saranno tutti uguali a 0 ($M_{i..i} = M_i$) e si userà solo la parte sopra alla diagonale. Nell'array r si memorizzino le dimensioni delle matrici da moltiplicare.

Si inizia calcolando il costo (= numero di moltiplicazioni) per la moltiplicazione delle coppie di matrici $M_{1..2} = M_1M_2$, $M_{3..4} = M_3M_4$, $M_{5..6} = M_5M_6$, ..., $M_{N-1..N} = M_{N-1}M_N$. Poi si passa a cercare il modo migliore per calcolare le triple successive di matrici, utilizzando i risultati già trovati per le coppie. Per esempio, le parentesizzazioni possibili per la tripla $M_1M_2M_3$ sono due: $((M_1M_2)M_3)$ oppure $((M_1M_2)M_3)$. Il costo di (M_1M_2) è già stato calcolato e si trova in m[1,2]. Quindi si ha:

costo di $((M_1M_2)M_3)$ = m[1,2] + r[1]*r[2]*r[3]

In seguito si prendono in considerazione i gruppi successivi di quattro matrici, e cosí via. Nella matrice m, in m[1,2] si avrà il numero di moltiplicazioni minimo per calcolare M_1M_2, in m[1,3] il numero di moltiplicazioni minimo per calcolare $M_1M_2M_3$, in m[1,N] il numero di moltiplicazioni minimo per calcolare $M_1M_2...M_N$.

Lo pseudo codice è:

```
1.  MATRIX-CHAIN (M_{1..N});
2.  for i ← 1 to N;
3.  |  do for j ← i+1 to N;
4.  L  L  do m[i,j] ← ∞;
5.  for i ← 1 to N;
6.  L  do m[i,j] ← 0;
7.  for j ← 1 to N;
8.  |  do for i ← 1 to N-j;
9.  |  |  do for k ← i+1 to i+j;
10. |  |  |  do tmp ← m[i,k-1] + m[k,i+j] + r_i r_k r_{i+j+1};
11. |  |  |  |  if tmp < m[i,i+j];
12. |  |  |  |  |  then m[i,i+j] ← tmp;
13. L  L  L  L  L  s[i,i+j] ← k;
```

Le righe 3 e 4 nello pseudocodice semplicemente inizializzano la matrice m al massimo numero ammissibile. Il successivo ciclo for pone a zero tutti gli elementi sulla diagonale di m.

Notare l'uso di una matrice di supporto s: quando si è trovata la migliore suddivisione in due del prodotto $M_{i..i+j}$, cioè un numero k tale che la moltiplicazione di $M_{i..k-1}M_{k..i+j+1}$ sia la migliore per k che varia tra i e i+j, si memorizza k in s[i,i+j]. Assieme alla matrice m, questa serve per ricostruire la migliore parentesizzazione del prodotto che si vuole calcolare. Lo si fa con una procedura ricorsiva come segue:

```
1.  MATRIX-CHAIN-ORDER (i, j, s);
```

```
if i = j
    then out ← M(i);
    else
        out ← "(";
        MATRIX-CHAIN-ORDER (i, s[i][j]-1, s);
        MATRIX-CHAIN-ORDER (s[i][j], j, s);
        out ← ")";
```

m

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

s

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

Figura 5.1: Moltiplicazione di matrici: come risultano le matrici di supporto m ed s dopo l'applicazione dell'algoritmo sulle seguenti matrici: $A_1 = 30\text{x}35, A_2 = 35\text{x}15, A_3 = 15\text{x}5, A_4 = 5\text{x}10, A_5 = 10\text{x}20, A_6 = 20\text{x}25$.

5.2 String matching approssimato

Un'altro problema a cui può essere applicato un algoritmo di programmazione dinamica è il cosiddetto **String Matching Approssimato**.

Siano P e T due sequenze di m ed n caratteri rispettivamente, con $m <= n$. Il problema consiste nel trovare una occorrenza di P in T ammettendo anche un numero limitato di differenze. Le differenze ammesse saranno:

Sostituzione: un carattere è diverso.

Inserimento: in P c'è un carattere che non compare in T.

Cancellazione: un carattere in T non compare in P.

In genere, per comodità di implementazione dell'algoritmo, si suppone che che sia P che in T abbiano un primo carattere vuoto. Come al solito, si usa una matrice D in cui memorizzare i risultati parziali. Di seguito lo pseudocodice:

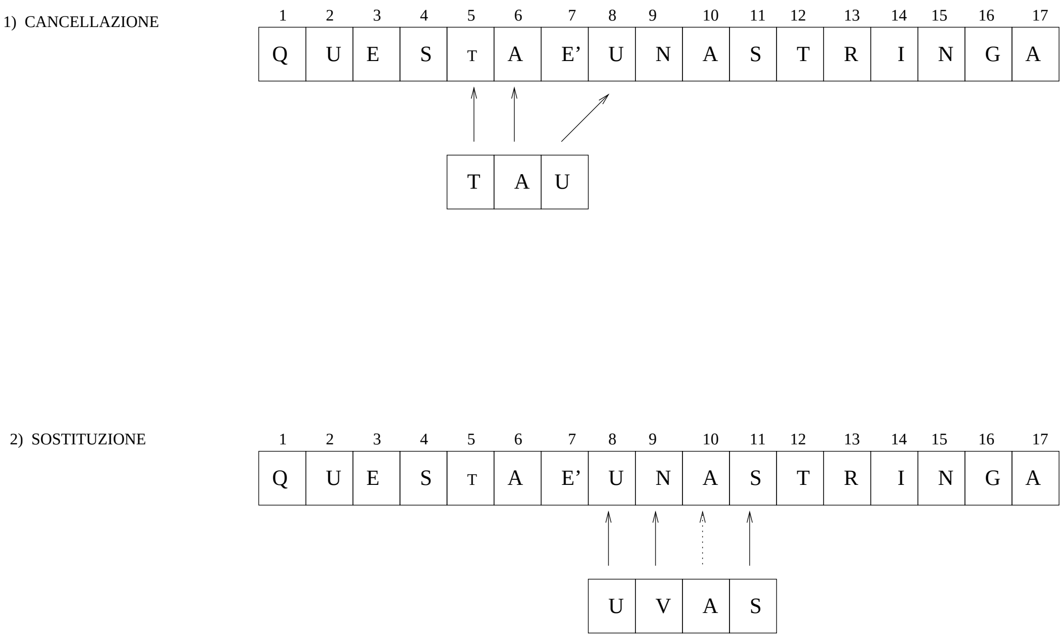

Figura 5.2: String matching approssimato: in 1) la stringa "TAU" ha una occorrenza a partire dalla posizione 5 di T con un differenza di tipo cancellazione. In 2) la stringa "UVAS" ha una occorrenza dalla posizione 8 con una sostituzione. In entrambi i casi si parla di **occorrenza 1-approssimata**.

```
STRING-MATCHING-APPROX (P, T);
for i ← 0 to n
  D[0,i] ← 0;
for i ← 0 to m
  D[i,0] ← i;
for i ← 1 to m
  for j ← 1 to n
    if P[i] = T[j]
      min ← D[i-1, j-1];
    else
      min ← D[i, j-1] + 1;
    if D[i-1, j]+1 < min
      min ← D[i-1, j]+1;
    if D[i, j-1]+1 < min
      min ← D[i, j-1]+1;
min ← D[m, 0];
for j ← 1 to n
  if D[m, j] < min
```

$min \leftarrow D[m, j]$;
$i \leftarrow j$;
return (i);