Fabrizio Luccio. Informatica per le Biotecnologie

6 Programmazione dinamica

La programmazione dinamica è un paradigma per la costruzione di algoritmi alternativo alla ricorsività. Esso si usa nei casi in cui esiste una definizione ricorsiva del problema, ma la trasformazione diretta di tale definizione in un algoritmo genera un programma di complessità esponenziale a causa del calcolo ripetuto, sugli stessi sottoinsiemi di dati, da parte delle diverse chiamate ricorsive.

Per comprendere questo punto consideriamo i due algoritmi ricorsivi RICERCA-BINARIA e MERGE-SORT visti nelle dispense 3 e 4. Il primo comprende due chiamate ricorsive nella sua *formulazione*, ma ogni *esecuzione* dell'algoritmo ne richiede una sola a seconda che il dato da ricercare sia minore o maggiore di quello incontrato nel passo della ricerca: in tal modo il calcolo si ripete su un solo sottoinsieme grande $n/2$, poi su un sottoinsieme di questo grande $n/4$, e così via: dividere il sottoinsieme per due fino a ridurlo a un singolo elemento implica, come abbiamo visto, l'esecuzione di $O(\log n)$ passi (si noti l'ordine O e non Θ: infatti l'algoritmo potrebbe terminare prima se il dato cercato si incontra in uno dei primi tentativi).

MERGE-SORT comprende anch'esso due chiamate ricorsive nella sua formulazione e ogni esecuzione dell'algoritmo le richiede entrambe per ordinare le metà "destra" e "sinistra" dell'insieme. Anche in questo caso in $\log n$ cicli i sottoinsiemi si riducono a un singolo elemento, ma poiché entrambi i sottoinsiemi vengono esaminati e il tempo per la loro fusione è lineare, il tempo richiesto, come già visto, è nel complesso $\Theta(n \log n)$.

La grande efficienza di questi due algoritmi rispetto ad altri realizzati in modo meno sofisticato è dovuta alla loro struttura ricorsiva, formulata in modo che chiamate ricorsive diverse agiscano su dati disgiunti: cioè l'algoritmo non è chiamato a ripetere operazioni già eseguite sugli stessi dati. Se ciò non si verifica, l'effetto della ricorsività sulla complessità di un algoritmo può essere disastroso.

Due esempi elementari sono il calcolo dei *numeri di Fibonacci* e il calcolo dei *coefficienti binomiali*. I primi sono definiti come:

$$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2}. \tag{1}$$

Per curiosità questi numeri furono introdotti nel XIII secolo da Leonardo figlio di Bonaccio (Fibonacci) da Pisa come numero di coppie di conigli presenti in un allevamento ideale in cui i conigli non muoiono mai, ogni coppia ne genera un'altra in un intervallo di tempo prefissato, ma una coppia appena generata deve aspettare il prossimo intervallo prima di essere fertile e generarne un'altra. Detto F_i il numero di coppie presenti al tempo i, l'allevamento è vuoto al tempo iniziale $i = 0$ (dunque $F_0 = 0$); vi si introduce una coppia appena nata al tempo $i = 1$ (dunque $F_1 = 1$) e questa potrà generarne un'altra a partire dal tempo $i = 2$ secondo una legge per la quale al tempo i il numero di coppie esistenti F_i è pari al numero presente al tempo precedente $i - 1$, più le nuove nate che sono tante quante ne esistevano al tempo $i - 2$ perchè solo queste sono fertili al tempo i. Da qui nasce la legge generale

$F_i = F_{i-1} + F_{i-2}$, che genera la progressione:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ...

Innescata dalla coppia 0,1 la crescita iniziale è piuttosto lenta, ma poi i valori aumentarno sempre più velocemente e infatti la loro crescita è esponenziale in i: si può infatti dimastrare che risulta $F_i \approx \frac{1}{\sqrt{5}}\Phi^i$, ove $\Phi = \frac{1+\sqrt{5}}{2}$ è un numero irrazionale pari a circa 1,62 noto in matematica come *sezione aurea*.

Si potrebbe utilizzare l'espressione (1) per formulare direttamente un algoritmo ricorsivo che, per un arbitrario ingresso i, calcola F_i come:

```
FIB(i)
// Calcolo ricorsivo dell'i-esimo numero di Fibonacci generato attraverso
   il costrutto return
   if (i == 0) {return 0;}
      else {if (i == 1) {return 1;}
         else {return (FIB(i − 1)+FIB(i − 2));}
            }
```

Il calcolo di F_n si innesca con la chiamata esterna FIB(n). Detto $T(n)$ il tempo richiesto dall'algoritmo avremo:

$T(0) = c_0,\ T(1) = c_1,$ con c_o e c_1 costanti;

$T(n) = T(n-1) + T(n-2) + c_2,$ con c_2 costante, per $n > 1$.

Possiamo semplificare il calcolo notando che $T(n) > 2T(n-2)$ da cui otteniamo:

$$\begin{aligned} T(n) &> 2\,T(n-2) > 2\,(2\,T(n-4)) = 2^2\,T(n-4) \\ &> 2^2\,(2\,T(n-6)) = 2^3\,T(n-6) > \dots \\ &> 2^k\,T(n-2k) > \dots \end{aligned}$$

da cui per n pari otteniamo $T(n) > 2^{n/2}T(0) = 2^{n/2}c_0$, e per n dispari otteniamo $T(n) > 2^{(n-1)/2}T(1) = 2^{(n-1)/2}c_1$. In ogni caso $T(n)$ è limitato inferiormente da una funzione esponenziale in n, quindi l'algoritmo BIN è esponenziale.

Questo fenomeno è probabilmente inaspettato per chi non abbia dimestichezza con la ricorsività degli algoritmi. Esso dipende dal fatto che le due chiamate ricorsive sono risolte con calcoli indipendenti tra loro, per cui il calcolo di FIB($i-1$) deve essere completato prima di iniziare il calcolo di FIB($i-2$), e il fenomeno si ripete in ogni chiamata successiva. Poiché il calcolo di F_{i-1} richiede per ricorrenza il calcolo di F_{i-2}, questo sarà effettuato per tale scopo, ma il suo valore non sarà più disponibile nella successiva chiamata ricorsiva FIB($i-2$): quindi F_{i-2} sarà calcolato due volte. Esaminando le chiamate successive possiamo renderci conto che F_{i-3} viene calcolato tre volte, F_{i-4} viene calcolato cinque volte, e in genere F_{i-k} viene calcolato un numero F_{k+1} di volte, pari cioè proprio a un numero di Fibonacci. Per esempio l'algoritmo ripete il calcolo di $F_{n-n} = F_0$ un numero F_{n+1} di volte: valore, come

abbiamo visto, esponenziale in n. E qui interviene la programmazione dinamica che si propone di conservare valori già calcolati per utilizzarli di nuovo se ciò è richiesto.

Secondo questo paradigma il calcolo di F_n è efficientemente eseguito con un algoritmo iterativo che costruisce la successione dei numeri di Fibonacci fino al punto n, ove ciascun elemento è direttamente calcolato come la somma dei due precedenti. Poiché ogni elemento della successione si calcola in tempo costante mediante un'addizione, il tempo complessivo è di ordine $\Theta(n)$.[1]

Un problema simile è quello del calcolo dei coefficienti binomiali definiti per due interi $0 \leq m \leq n$ come:

$$\binom{n}{m} = 1, \quad \text{per } m = 0 \text{ e } m = n;$$
$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}, \quad \text{per } 0 < m < n. \tag{2}$$

Per un insieme arbitrario, $\binom{n}{m}$ è il numero di combinazioni di n elementi presi in gruppi di m. Per esempio per l'insieme di cinque elementi $\{a, b, c, d, e\}$ vi sono i $\binom{5}{2} = 10$ gruppi di due elementi : $ab, ac, ad, ae, bc, bd, be, cd, ce, de$. Come noto i coefficienti binomiali appaiono nel *triangolo di Tartaglia*, che si può rappresentare in un'area triangolare all'interno di una matrice M di dimensioni $(n+1) \times (n+1)$, ove i valori di n e m sono relativi rispettivamente alle righe e alle colonne. Per $n = 5$ la M risulta:

	0	1	2	3	4	5
0	**1**					
1	**1**	**1**				
2	**1**	**2**	**1**			
3	**1**	**3**	**3**	1		
4	1	**4**	**6**	4	1	
5	1	5	**10**	10	5	1

ove per esempio $\binom{5}{2} = 10$ è l'elemento in riga 5 e colonna 2.

La matrice è calcolata in base alla formula (2): si noti il valore limite uguale a 1 nella colonna zero e nella diagonale, e gli altri valori calcolati come $M[i, j] = M[i-1, j] + M[i-1, j-1]$. Ciò suggerisce immediatamente di calcolare $\binom{n}{m}$ con un algoritmo iterativo ove gli elementi della matrice sono calcolati progressivamente per righe, determinando gli elementi della riga i per addizioni sulla riga $i-1$ fino a raggiungere $M[n, m]$. E in effetti è sufficiente calcolare gli elementi in una zona romboidale di larghezza $m+1$ e altezza $n-m+1$ come indicato in grassetto nella matrice qui sopra per il calcolo di $\binom{5}{2}$. Poiché ogni elemento della matrice si calcola

[1] Per n grande la cosa non è così semplice perché il numero di cifre di F_n cresce come $\log n$ e non può quindi, da un certo punto in poi, essere contenuto in una sola cella di memoria. Si noti anche che per il calcolo di F_n si può usare la formula approssimata $\frac{1}{\sqrt{5}}\Phi^n$ con opportune correzioni per farla convergere sull'intero richiesto.

in tempo costante, l'algoritmo richiede tempo proporzionale a $(m+1)(n-m+1)$, cioè di ordine $\Theta(nm) = O(n^2)$ (si ricordi che $m \leq n$).

Anche questo è un esempio di programmazione dinamica: nato dalla definizione ricorsiva (2), l'algoritmo calcola progressivamente i valori di $M[i,j]$ mantenendoli memorizzati fino al loro impiego per valori successivi degli indici. Uno spontaneo algoritmo ricorsivo avrebbe invece la forma:

```
BINOMIALE(i, j)
// Calcolo ricorsivo di (i j), generato dall'algoritmo attraverso la frase return
   if (j == 0) {return 1;}
      else {if (i == j) {return 1;}
         else {return (BINOMIALE(i − 1, j)+BINOMIALE(i − 1, j − 1));}
           }
```

ove il calcolo di $\binom{n}{m}$ si innesca con la chiamata esterna BINOMIALE(n,m). Detto $T(n,m)$ il tempo richiesto da questo algoritmo avremo:

$T(n,0) = c_0,\ T(n,n) = c_1,$ con c_o e c_1 costanti, per qualsiasi n;

$T(n,m) = T(n-1,m) + T(n-1,m-1) + c_2,$ con c_2 costante, per $0 < m < n$.

Come nel caso dei numeri di Fibonacci possiamo semplificare il calcolo notando che $T(n,m) > 2T(n-1,m-1)$ da cui otteniamo:

$$T(n,m) > 2\,T(n-1,m-1) > 2^2\,T(n-2,m-2) > ... > 2^m\,T(n-m,0) = 2^m c_0.$$

Un tempo, dunque, limitato inferiormente da una funzione esponenziale, per gli stessi motivi discussi per i numeri di Fibonacci.

Applicazioni classiche della programmazione dinamica si incontrano nel confronto tra sequenze di caratteri: problemi nati nell'ambito delle basi di dati e degli editori di testo, e oggi importantissimi nelle applicazioni algoritmiche in biologia molecolare (analisi del DNA ecc). Ne vedremo lo schema di base in una prossima dispensa.