Capitolo 3: Procedure e funzioni ricorsive

L'uso di procedure ricorsive (o “di ricorrenza” o “ricorrenti”) permette spesso di descrivere un algoritmo in maniera semplice e concisa, mettendo in rilievo la tecnica adottata per la soluzione del problema e facilitando quindi la fase di progettazione. Inoltre l'analisi risulta in molti casi semplificata poiché la valutazione del tempo di calcolo si riduce alla soluzione di equazioni ricorsive.
Molti classici algoritmi possono essere descritti mediante procedure ricorsive. Di conseguenza l'analisi dei relativi tempi di calcolo è ridotta alla soluzione di una o più equazioni ricorsive nelle quali si esprime il termine n-esimo di una sequenza in funzione dei precedenti.
Esistono varie tecniche utilizzate per risolvere queste equazioni o almeno per ottenere una soluzione approssimata.
E’ importante osservare che l’esecuzione di procedure ricorsive prevede la sua conversione in procedura iterativa. Su macchine RAM tale conversione è evidente perchè avviene manualmente a cura del programmatore mentre su macchine reali a singolo processore tale conversione viene eseguita automaticamente.

La conversione richiede l’uso di uno stack nel quale memorizzare l’istantanea dello stato della macchina prima di ciascuna chiamata ricorsiva. Il rischio, pertanto, è quello di esaurire, su lunghi cicli ricorsivi, le risorse dello stack disponibili.

Spesso, quindi, sarà strategia vincente quella di convertire il problema inizialmente rappresentato ricorsivamente in un problema analogo che non usa metodi ricorsivi.

§.16 Procedure ricorsive

Una procedura che chiama se stessa, direttamente o indirettamente, viene detta ricorsiva.

Consideriamo ad esempio il problema di determinare il numero massimo di parti in cui n rette dividono il piano.

Detto $p(n)$ tale numero, il disegno di un algoritmo ricorsivo per calcolare $p(n)$ è basato sulla seguente proprietà:

1. Una retta divide il piano in due parti, cioè $p(1) = 2$;
2. Sia $p(n)$ il numero di parti in cui n rette dividono il piano; aggiungendo una nuova retta, è facile osservare che essa interseca le precedenti in al più n punti, creando al più $n+1$ nuove regioni.

Vale quindi la relazione:

(36) $$p(n+1) = p(n) + (n+1)$$

da cui è facile ricavare la corrispondente procedura ricorsiva:

Procedura $P(n)$
if $n=1$ **then** **return** 2
else $x := P(n-1)$
return $(x+n)$

Questa tecnica di disegno cerca quindi di esprimere il valore di una funzione su un dato in dipendenza di valori della stessa funzione su dati "più piccoli".

Molti problemi si prestano in modo naturale ad essere risolti con procedure ricorsive, ottenendo algoritmi risolutivi in generale semplici e chiari.

Noi ci limiteremo ad affrontare il problema di analisi degli algoritmi ricorsivi: dato un algoritmo ricorsivo, stimare il tempo di calcolo della sua esecuzione su macchina RAM.

Tale stima può essere fatta agevolmente attraverso l'analisi della procedura ricorsiva stessa.

Si procede come segue, supponendo che l'algoritmo sia descritto da M procedure $P_1, P_2, \ldots, P_M$ comprendenti il programma principale:

1. Si associa ad ogni indice k con $1 \le k \le M$ la funzione (incognita) $T_k(n)$ che denota il tempo di calcolo di P_k in funzione della dimensione dell'ingresso.
2. Si esprime $T_k(n)$ in funzione dei tempi delle procedure chiamate da P_k, valutati negli opportuni valori dei parametri; a tal riguardo osserviamo che il tempo di esecuzione della istruzione $Z := P_j(a)$ è la somma del tempo di esecuzione della procedura P_j sull'istanza (ingresso) a, del tempo di chiamata di P_j (necessario a predisporre il record di attivazione) e di quello di ritorno.

Si ottiene in tal modo un sistema di M equazioni ricorsive che, risolto, permette di stimare $T_k(n) \quad \forall k = 1, \cdots, M$ ed in particolare per il programma principale.

A scopo esemplificativo, effettuiamo una stima del tempo di elaborazione dell'algoritmo per la generazione dei numeri di Fibonacci.

Per semplicità supporremo che la macchina sia in grado di effettuare una chiamata in una unità di tempo (più un'altra unità per ricevere il risultato).

§.17 I numeri di Fibonacci

Consideriamo la successione di interi $0,1,1,2,3,5,8,13,\ldots$ in cui ogni termine è ottenuto sommando i due precedenti. Tale successione, detta di Fibonacci, può essere rappresentata tramite una equazione ricorsiva con le relative condizioni iniziali:

(37)
$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_0 = 0 \\ F_1 = 1 \end{cases}$$

Per valutare il tempo di elaborazione (su macchina RAM) $T_{Fib}(n)$ della procedura ricorsiva $Fib(n)$ analizziamo i tempi richiesti dalle singole istruzioni:

Procedura $Fib(n)$		
if $n \le 1$		$[T = \Theta(1)]$
then	**return** n	$[T = \Theta(1)]$
else	$a := n-1$	$[T = \Theta(1)]$
	$x := Fib(a)$	$[T = \Theta(1) + T_{Fib}(n-1)]$
	$b := n-2$	$[T = \Theta(1)]$
	$y := Fib(b)$	$[T = \Theta(1) + T_{Fib}(n-2)]$
	return $(x+y)$	$[T = \Theta(1)]$

Vale allora la seguente equazione ricorsiva:

(38)
$$T_{Fib}(n) = \begin{cases} \Theta(1) & n = 0,1 \\ \Theta(1) + T_{Fib}(n-1) + T_{Fib}(n-2) & n \ge 2 \end{cases}$$

Ribadiamo che l'implementazione dell'algoritmo nella sua forma ricorsiva è altamente inefficiente perchè, come vedremo, $T_{Fib}(n) = \Theta\left(2^{cn}\right)$ con c costante.

Se evitiamo di usare procedure ricorsive, esteticamente belle e semplici da individuare a partire da un problema che naturalmente si delinea in forma ricorsiva, scopriamo che i tempi di elaborazione sono notevolmente ed evidentemente più contenuti.

Procedura $Fib2(n)$

if $n \leq 1$ $\quad [T = \Theta(1)]$

then **return** n $\quad [T = \Theta(1)]$

else $a := 0$ $\quad [T = \Theta(1)]$

$b := 1$ $\quad [T = \Theta(1)]$

for $j = 2, \ldots, n$

$c := a + b$ $\quad [T = \Theta(1)]$

$a := b$ $\quad [T = \Theta(1)]$

$b := c$ $\quad [T = \Theta(1)]$

return c $\quad [T = \Theta(1)]$

Poichè le istruzioni contenute nel ciclo sono complessivamente $\Theta(1)$ e il ciclo viene eseguito $n-1$ volte è immediato concludere che $T_{Fib2}(n) = \Theta(n)$.

Esiste però un terzo algoritmo che, sfruttando alcune considerazioni di teoria dei numeri, è in grado di abbattere ulteriormente i tempi di elaborazione.

Infatti, detta $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$, è possibile dimostrare che l'elemento a_{12} di A^n coincide proprio con il termine n-esimo della successione di Fibonacci.

Poichè esiste un algoritmo, noto come Fast Power, che elabora la potenza n-esima di una matrice in un tempo $\Theta(\lg n)$ ecco che, riconducendo il problema di calcolare il termine n-esimo della successione di Fibonacci al problema di calcolare la potenza n-esima della matrice A, i tempi di elaborazione vengono ricondotti a $T_{Fib3}(n) = \Theta(\lg n)$.

§.18 Ricorsione terminale

Il metodo generale per la traduzione iterativa della ricorsione può essere semplificato e reso più efficiente quando la chiamata a una procedura è l'ultima istruzione eseguita dal programma chiamante. In questo caso infatti, una volta terminata l'esecuzione della procedura chiamata, non occorre restituire il controllo a quella chiamante.

Per descrivere la traduzione iterativa di questa ricorsione, denotiamo rispettivamente con A e B la procedura chiamante e quella chiamata e supponiamo che la chiamata di B sia l'ultima istruzione del programma A. Possiamo allora eseguire la chiamata a B semplicemente sostituendo il record di attivazione di A con quello di B nella pila e aggiornando opportunamente l'indirizzo di ritorno alla procedura che ha chiamato A; il controllo passerà così a quest'ultima una volta terminata l'esecuzione di B. In questo modo si riduce il numero di record di attivazione mantenuti nella pila, si risparmia tempo di calcolo e spazio di memoria rendendo quindi piùefficiente l'implementazione.

Questo tipo di ricorsione viene chiamata ricorsione terminale. Un caso particolarmente semplice si verifica quando l'algoritmo è formato da un'unica procedura che richiama se stessa all'ultima istruzione. In tale situazione non occorre neppure mantenere una pila

per implementare la ricorsione perché non mai è necessario riattivare il programma chiamante una volta terminato quello chiamato.
Il seguente schema di procedura rappresenta un esempio tipico di questo caso. Consideriamo una procedura F, dipendente da un parametro x, definita dal seguente programma:

Procedura $F(x)$
if $C(x)$ **then** D
else E
$y := g(x)$
$F(y)$

Qui $C(x)$ è una condizione che dipende dal valore di x, mentre E e D sono opportuni blocchi di istruzioni. La funzione $y = g(x)$ invece determina un nuovo valore del parametro di input per la F di dimensione ridotta rispetto a quello di x.
Allora $F(x)$ è equivalente alla seguente procedura:

Procedura $\hat{F}(x)$
$x := a$
while $\neg C(x)$
E
$x := g(x)$
D

§.19 Equazioni ricorsive

Supponiamo di dover analizzare dal punto di vista della complessità un algoritmo definito mediante un insieme di procedure $P_1, P_2, \ldots, P_m$ che si richiamano ricorsivamente fra loro.

L'obiettivo dell'analisi è quello di stimare la funzione $T_i(n) \quad \forall i = 1, \ldots, m$ che rappresenta il tempo di elaborazione della procedura i-esima su dati di dimensione n.
Se ogni procedura richiama le altre su dati di dimensione minore, sarà possibile esprimere $T_i(n)$ come funzione dei valori $T_j(k)$ tali che $j \in \{1, \ldots, m\}$ e $k < n$.

Per fissare le idee, supponiamo di avere una sola procedura P che chiama sè stessa su dati di dimensione minore. Sia $T(n)$ il tempo di calcolo richiesto su dati di dimensione n nell'ipotesi "caso peggiore" oppure nell'ipotesi "caso medio". Sarà in generale possibile determinare opportune funzioni f_k in $k > 0$ variabili, tali che:

(39) $$T(n) = f_n\left(T(n-1), \ldots T(1), T(0)\right)$$

o almeno tali che:

(40) $$T(n) \leq f_n\left(T(n-1), \ldots T(1), T(0)\right)$$

Relazioni del precedente tipo sono dette relazioni ricorsive e, in particolare, in caso di uguaglianza, sono dette equazioni ricorsive.
Si osservi che data la condizione al contorno $T(0) = a$, esiste un'unica funzione $T(n)$ che soddisfa l'equazione (39).

Consideriamo per esempio il problema di valutare il tempo di calcolo delle seguenti procedure assumendo il criterio di costo uniforme:

```
Procedura B(n)
S := 0
for i = 0,…,n
        S := S + A(i)
return S
```

```
Procedura A(n)
if n = 0 then   return 0
         else   u := n − 1
                b := n + A(u)
                return b
```

Osserviamo innanzitutto che la procedura B richiama A, mentre A richiama se stessa. Per semplicità, assumiamo uguale a c il tempo di esecuzione di ogni istruzione ad alto livello e denotiamo con $T_B(n)$ e $T_A(n)$ rispettivamente il tempo di calcolo dell'esecuzione di B e A su input n. Allora si ottengono le seguenti equazioni:

$$T_B(n) = c + \sum_{i=0}^{n} \big(c + T_A(i)\big) + c$$

(41)

$$T_A(n) = \begin{cases} 2c & n = 0 \\ 2c + T_A(n-1) & n \geq 1 \end{cases}$$

Il problema di analisi è allora ridotto alla soluzione dell'equazione di ricorrenza relativa ai valori $T_A(n) \quad \forall n \in \mathbb{N}$
Lo sviluppo di tecniche per poter risolvere equazioni o relazioni di ricorrenza è quindi un importante e preliminare strumento per l'analisi di algoritmi e le prossime sezioni sono dedicate alla presentazione dei principali metodi utilizzati.

§.20 Metodo dei fattori sommanti

Con questa sezione iniziamo lo studio delle equazioni di ricorrenza più comuni nell'analisi degli algoritmi. In generale il nostro obiettivo è quello di ottenere una valutazione asintotica della soluzione oppure, più semplicemente, una stima dell'ordine di grandezza. Tuttavia sono stati sviluppati in letteratura vari metodi che permettono di ricavare la soluzione esatta di una ricorrenza. In alcuni casi poi l'equazione di ricorrenza è particolarmente semplice e si può ottenere la soluzione esatta iterando direttamente l'uguaglianza in esame.

Esempio 1. Consideriamo la seguente equazione di ricorrenza:

$$T(n)=\begin{cases}0 & n=0\\ T(n-1)+2n & n\geq 1\end{cases}$$

Poichè, per ogni $n>1$, $T(n-1)=T(n-2)+2(n-1)$ sostituendo questa espressione nell'equazione precedente si ricava $T(n)=2n+2(n-1)+T(n-2)$. Iterando n volte si ricava

$$T(n)=2\sum_{j=1}^{n} j+T(0)=n(n+1)$$

Esempio 2. Data l'equazione ricorsiva:

$$T(n)=\begin{cases}1 & n=1\\ T\left(\left\lfloor \frac{n}{2}\right\rfloor\right)+1 & n\geq 2\end{cases}$$

osservando che $T(n)=k+T\left(\left\lfloor \frac{n}{2^k}\right\rfloor\right)$ e che $\left\lfloor \frac{n}{2^k}\right\rfloor=1$ per $k:2^k\leq n<2^{k+1}$ ovvero per $k=\lfloor \lg n\rfloor$ si conclude che $T(n)=\lfloor \lg n\rfloor+1$.

Esempio 3. Data l'equazione ricorsiva:

$$T(n)=\begin{cases}1 & n=0\\ \frac{n+k-1}{n}T(n-1) & n\geq 1\end{cases}$$

si ricava immediatamente che

$$T(n)=\frac{(n+k-1)!}{n!(k-1)!}T(0)=\binom{n+k-1}{k-1}.$$

Esempio 4. Data l'equazione ricorsiva:

$$T(n)=\begin{cases}0 & n=0\\ 2T(n-1)+2n & n\geq 1\end{cases}$$

osservando che $T(n-1)=2T(n-2)+2(n-1)$ e sostituendo n volte otteniamo:

$$T(n)=2^nT(0)+\sum_{k=1}^{n}2^k(n-k+1)=\sum_{k=1}^{n}2^k(n-k+1)$$

Manipoliamo la sommatoria per ricondurci ad una forma notevole:

$$\sum_{k=1}^{n}2^k(n-k+1)=2^{n+1}\sum_{k=1}^{n}\frac{n-k+1}{2^{n-k+1}}=2^{n+1}\sum_{k=1}^{n}\frac{k}{2^k}=2^n\sum_{k=1}^{n}k\left(\frac{1}{2}\right)^{k-1}=2^n\sum_{k=0}^{n}k\left(\frac{1}{2}\right)^{k-1}$$

Ora, per proseguire, dobbiamo notare che la sommatoria è la derivata della progressione geometrica.

Più precisamente, detta $f_n(x) = \sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}$, abbiamo che:

$$\sum_{k=0}^{n} k\left(\frac{1}{2}\right)^{k-1} = \left.\frac{df_n}{dx}\right|_{x=\frac{1}{2}} = \left.\frac{d}{dx}\frac{1-x^{n+1}}{1-x}\right|_{x=\frac{1}{2}} = \left.\frac{1-(n+1)x^n+nx^{n+1}}{(1-x)^2}\right|_{x=\frac{1}{2}}$$

da cui

$$2^n\sum_{k=0}^{n} k\left(\frac{1}{2}\right)^{k-1} = 2^{n+2}\left(1-2^{-n}(n+1)+2^{-n-1}n\right) = 2^{n+2}-4(n+1)+2n = 2^{n+2}-2(n+2)$$

§.21 Equazioni "divide et impera"

Un'importante classe di equazioni di ricorrenza è legata all'analisi di algoritmi del tipo "divide et impera". Ricordiamo che un algoritmo di questo tipo suddivide il generico input di dimensione n in un certo numero m di sottoistanze del medesimo problema, ciascuna di dimensione a inferiore a quella iniziale; quindi richiama ricorsivamente se stesso su tali istanze ridotte e poi ricompone i risultati parziali ottenuti per determinare la soluzione cercata.
Il tempo di calcolo di un algoritmo di questo tipo è quindi soluzione di una equazione di ricorrenza della forma

(42) $$T(n) = mT\left(\frac{n}{a}\right) + g(n)$$

dove $g(n)$ è il tempo necessario per ricomporre i risultati parziali in un'unica soluzione.

Per alcune funzioni $g(n)$ esistono formule risolutive standard.

Siano $m, a, b, c \in \mathbb{R}^+$ con $a > 1$ e sia

(43) $$T(n) = \begin{cases} b & n = 1 \\ mT\left(\frac{n}{a}\right) + bn^c & n \geq 2 \end{cases}$$

definita per ogni n che sia potenza intera di a.
Allora $T(n)$ soddisfa le seguenti relazioni:

(44) $$T(n) = \begin{cases} \Theta\left(n^c\right) & m < a^c \\ \Theta\left(n^c \lg n\right) & m = a^c \\ \Theta\left(n^{\log_a m}\right) & m > a^c \end{cases}$$

A tale risultato si perviene sviluppando la ricorrenza fino alla forma $T(n) = bn^c \sum_{j=0}^{\log_a n} \left(\frac{m}{a^c} \right)^j$. Da qui, risolvendo esattamente la sommatoria, è anche possibile ricavare le espressioni asintotica e di rapporto limitato per $T(n)$.

Poiché la formula precedente è valida solo per valori di n che siano potenze intere di $a > 1$, ora costruiremo una formula che con qualche limitazione su $T(n)$ vale per ogni n intero qualsiasi.

Come esempio consideriamo l'algoritmo Mergesort che, sostanzialmente, ordina un vettore di n elementi spezzandolo in due parti di dimensione $\left\lfloor \frac{n}{2} \right\rfloor$ e $\left\lceil \frac{n}{2} \right\rceil$ rispettivamente; quindi richiama se stesso sui due sottovettori e poi compone le due soluzioni.
Si può verificare che il numero $M(n)$ di confronti eseguiti per ordinare n elementi, dove n in questo caso è un qualsiasi intero positivo, soddisfa la seguente ricorrenza:

(45)
$$M(n) = \begin{cases} 0 & n = 1 \\ M\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + M\left(\left\lceil \frac{n}{2} \right\rceil \right) + n - 1 & n \geq 2 \end{cases}$$

In generale le equazioni del tipo “divide et impera” nelle quali compaiono le parti intere possono essere trattate usando il seguente risultato che estende l'analoga valutazione asintotica ottenuta in precedenza.

Siano $a, b, c \in \mathbb{R}^+$ con $a > 1$; consideriamo inoltre $m_1, m_2 \in \mathbb{N}$: $m_1 + m_2 > 0$ e definiamo $T(n)$ mediante la seguente equazione:

(46)
$$T(n) = \begin{cases} b & n = 1 \\ m_1 T\left(\left\lfloor \frac{n}{a} \right\rfloor \right) + m_2 T\left(\left\lceil \frac{n}{a} \right\rceil \right) + bn^c & n \geq 2 \end{cases}$$

Allora, posto $m = m_1 + m_2$, $T(n)$ soddisfa le relazioni (44).
Per arrivare a tale risultato si devono considerare separatamente i tre casi $m < a^c, m = a^c, m > a^c$ previsti dalle (44). Inoltre si deve porre $m_1 = 0, m_2 = m > 0$ per ottenere una maggiorazione e $m_2 = 0, m_1 = m > 0$ per ottenere una minorazione. Per ognuno di questi sei casi complessivi si osservi che $k = \lfloor \log_a n \rfloor$ conduce a $T(a^k) \leq T(n) \leq T(a^{k+1})$ essendo $T(n)$ monotona non decrescente. A questo punto possiamo valutare $T(a^k)$ e $T(a^{k+1})$ usando i risultati del punto (44) e concludere, ad esempio, che $T(n) = \Theta(n^c)$.