Algoritmi e Strutture Dati

Capitolo 1

Un'introduzione informale agli algoritmi: ancora sulla sequenza di Fibonacci

Copyright © 2004 - The McGraw - Hill Companies, srl

Punto della situazione

- Stiamo cercando di calcolare efficientemente l'*n*-esimo numero della sequenza di Fibonacci
- Abbiamo progettato 2 algoritmi:
  - `fibonacci1`, non corretto (su modelli di calcolo realistici) in quanto approssima la soluzione
  - `fibonacci2`, che impiega tempo esponenziale in *n*

- Dovevate dimostrare che per `fibonacci2(n)`

$$T(n) = F_n + 2\,(F_n - 1) = 3F_n - 2$$

Foglie # Nodi interni

Copyright © 2004 - The McGraw - Hill Companies, srl

Dimostrazione del Lemma 1

Lemma 1: Il numero di foglie dell'albero della ricorsione di `fibonacci2(n)` è pari a F_n

Dim: Per induzione su n:

- Caso base n=1 (e anche n=2): in questo caso l'albero della ricorsione è costituito da un unico nodo, che è quindi anche una foglia; poiché $F_1=1$, il lemma segue.
- Caso n>2: supposto vero fino ad n-1, dimostriamolo vero per n; osserviamo che l'albero della ricorsione associato ad n è formato da una radice etichettata F(n) e da due sottoalberi etichettati F(n-1) e F(n-2). Per l'ipotesi induttiva, tali sottoalberi hanno rispettivamente F_{n-1} ed F_{n-2} foglie, e quindi l'albero della ricorsione associato ad n avrà $F_{n-1} + F_{n-2} = F_n$ foglie, come volevasi dimostrare. □

Copyright © 2004 - The McGraw - Hill Companies, srl

Dimostrazione del Lemma 2

Lemma 2: Il numero di nodi interni di un albero strettamente binario è pari al numero di foglie – 1.

Dim: Per induzione sul numero di nodi interni, sia detto k:

- Caso base k=1: se c'è un solo nodo interno, poiché per ipotesi deve avere due figli, tali figli saranno foglie, e quindi il lemma segue.
- Caso k>1: supposto vero fino a k-1, dimostriamolo vero per k nodi interni; osserviamo che poiché k>1, e l'albero è strettamente binario, abbiamo due possibilità:
  1. Uno dei due sottoalberi della radice è una foglia: in tal caso l'altro sottoalbero (strettamente binario) contiene k-1 nodi interni, e quindi per ipotesi induttiva avrà k foglie; allora, il numero totale di foglie è k+1, da cui segue il lemma;
  2. Entrambi i sottoalberi (strettamente binari) contengono nodi interni, in numero totale di $k-1=k_1+k_2$; ma allora, per ipotesi induttiva, conterranno rispettivamente k_1+1 e k_2+1 foglie, e quindi il numero totale di foglie è $k_1+k_2+2=k+1$, come volevasi dimostrare.

Copyright © 2004 - The McGraw - Hill Companies, srl

Algoritmo `fibonacci3`

- Perché l'algoritmo `fibonacci2` è lento? Perché continua a ricalcolare ripetutamente la soluzione dello stesso sottoproblema. Perché non memorizzare allora in un array le soluzioni dei sottoproblemi?

algoritmo `fibonacci3`*(intero n)* $\rightarrow$ *intero*
  sia Fib un array di n interi
  Fib[1] $\leftarrow$ *Fib[2]* $\leftarrow$ *1*
  for i = 3 **to** n **do**
    Fib[i] $\leftarrow$ *Fib[i-1] + Fib[i-2]*
  return *Fib[n]*

Correttezza? Corretto per definizione**!**

Copyright © 2004 - The McGraw - Hill Companies, srl

La struttura dati vettore o array

- Il vettore o array è una struttura dati utilizzata per rappresentare sequenze di elementi omogenei
- Un vettore è visualizzabile tramite una struttura unidimensionale di celle; ad esempio, un vettore di 5 interi ha la seguente forma

5	23	1	12	5

- Ciascuna delle celle dell'array è identificata da un valore di indice
- Gli array sono (generalmente) allocati in celle contigue della memoria del computer, alle quali si può accedere direttamente

Copyright © 2004 - The McGraw - Hill Companies, srl

Calcolo del tempo di esecuzione

- Linee 1, 2, e 5 eseguite una sola volta
- Linea 3 eseguita $n - 1$ volte (una sola volta per n=1,2)
- Linea 4 eseguita $n - 2$ volte (non eseguita per n=1,2)
- T(n): numero di linee di codice mandate in esecuzione da `fibonacci3`

$$T(n) = n - 1 + n - 2 + 3 = 2n \qquad n > 1$$

$$T(1) = 4$$

$$T(45) = 90$$

$$T(100) = 200$$

Per n=45, circa 38 milioni di volte più veloce dell'algoritmo `fibonacci2`!

Per n=100, F_{100}=354224848179261915075, quindi circa 10^{19} volte più veloce!

Copyright © 2004 - The McGraw - Hill Companies, srl

Calcolo del tempo di esecuzione

- L'algoritmo `fibonacci3` impiega tempo proporzionale a n invece che esponenziale in n, come accadeva invece per `fibonacci2`
- Tempo effettivo richiesto da implementazioni in C dei due algoritmi su piattaforme diverse (un po' obsolete ☺):

	`fibonacci2(58)`	`fibonacci3(58)`
Pentium IV 1700MHz	15820 sec. ($\simeq$ 4 ore)	0.7 milionesimi di secondo
Pentium III 450MHz	43518 sec. ($\simeq$ 12 ore)	2.4 milionesimi di secondo
PowerPC G4 500MHz	58321 sec. ($\simeq$ 16 ore)	2.8 milionesimi di secondo

Copyright © 2004 - The McGraw - Hill Companies, srl

Occupazione di memoria

- Il tempo di esecuzione non è la sola risorsa di calcolo che ci interessa. Anche la quantità di memoria necessaria può essere cruciale.
- Se abbiamo un algoritmo lento, dovremo solo attendere più a lungo per ottenere il risultato
- Ma se un algoritmo richiede più spazio di quello a disposizione, non otterremo mai la soluzione, indipendentemente da quanto attendiamo!
- È il caso di `Fibonacci3`, la cui correttezza è subordinata alla dimensione della memoria allocabile

Copyright © 2004 - The McGraw - Hill Companies, srl

Algoritmo `fibonacci4`

- `fibonacci3` usa un array di dimensione n (per il momento ignoriamo il fatto che il contenuto di tali celle cresce esponenzialmente)
- In realtà non ci serve mantenere tutti i valori di F_n precedenti, ma solo gli ultimi due, riducendo lo spazio a poche variabili in tutto:

```
algoritmo fibonacci4(intero n) → intero
    a ← b ← 1
    for i = 3 to n do
        c ← a+b
        a ← b
        b ← c
    return b
```

Copyright © 2004 - The McGraw - Hill Companies, srl

Correttezza? Corretto per definizione!

Efficienza?

- Per la risorsa tempo, calcoliamo ancora una volta il numero di linee di codice $T(n)$ mandate in esecuzione
  - Se $n \leq 2$: tre sole linee di codice
  - Se $n \geq 3$: $T(n) = 2+(n-1)+3\cdot(n-2) = 4n-5$ (per `Fibonacci3` avevamo $T(n)=2n$)
- Per la risorsa spazio, contiamo il numero di variabili di lavoro utilizzate: $S(n)=4$ (per `Fibonacci3` avevamo $S(n)=n+1$) [**NOTA:** stiamo assumendo che ogni locazione di memoria può contenere un valore infinitamente grande!]

Copyright © 2004 - The McGraw - Hill Companies, srl

Notazione asintotica

- Misurare T(n) come il numero di linee di codice mandate in esecuzione è una misura molto approssimativa del tempo di esecuzione, poiché trascura il tempo necessario per eseguire le operazioni soggiacenti
- Potremmo quindi erroneamente essere portati a pensare che due linee di codice che contengono una semplice assegnazione siano più costose di una linea di codice che contiene un'operazione matematica complessa!
- Per evitare questo problema, descriveremo l'ordine di grandezza di T(n) ignorando dettagli "inessenziali" come le costanti moltiplicative, additive e sottrattive
- Useremo a questo scopo la notazione asintotica Θ

Copyright © 2004 - The McGraw - Hill Companies, srl

Notazione asintotica Θ (definizione informale)

Supponiamo che f e g siano due funzioni definitivamente diverse da zero per $n \to \infty$, e che

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L, \quad 0 < L < \infty$$

Allora, scriveremo che f(n) = Θ(g(n)).

Copyright © 2004 - The McGraw - Hill Companies, srl

Esempi:

Sia $f(n) = 2n^2 + 3n$, allora $f(n)=\Theta(n^2)$

Sia $f(n) = n^2 – n \log n$, allora $f(n)=\Theta(n^2)$

Sia $f(n) = n^3 -2n^2+3n$, allora $f(n)=\Theta(n^3)$

Sia $f(n) = 23$, allora $f(n)=\Theta(1)$

Sia $f(n) = 3^n +2^n$, allora $f(n)=\Theta(3^n)$

Copyright © 2004 - The McGraw - Hill Companies, srl

Andamento asintotico per i `Fibonacci`

- `Fibonacci2` $T(n)=3F_n-2 \Rightarrow T(n)=\Theta(F_n) \Rightarrow T(n)=\Theta(\Phi^n)$, poiché

$$\lim_{n\to\infty} \frac{1/\sqrt{5}\left(\phi^n-\hat{\phi}^n\right)}{\phi^n} = 1/\sqrt{5}$$

- `Fibonacci3` $T(n)=2n \Rightarrow T(n)=\Theta(n)$, $S(n)=\Theta(n)$
- `Fibonacci4` $T(n)=4n-5 \Rightarrow T(n)=\Theta(n)$, $S(n)=\Theta(1)$

Copyright © 2004 - The McGraw - Hill Companies, srl

Un nuovo algoritmo

Possiamo sperare di calcolare F_n in tempo inferiore a $\Theta(n)$? Sembrerebbe impossibile…

Copyright © 2004 - The McGraw - Hill Companies, srl

Potenze ricorsive

- `Fibonacci4` non è il miglior algoritmo possibile
- È possibile dimostrare per induzione la seguente proprietà di matrici:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \underbrace{\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \ldots \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}}_{\text{n volte}} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

- Useremo questa proprietà per progettare un algoritmo più efficiente

Copyright © 2004 - The McGraw - Hill Companies, srl

Prodotto di matrici righe per colonne

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix} \qquad B = \begin{pmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,n} \end{pmatrix}$$

$$(\mathrm{AB})_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

$$i = 1, \ldots, n$$
$$j = 1, \ldots, n$$

Dimostrazione per induzione

Base induzione: n=2

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} F_3 & F_2 \\ F_2 & F_1 \end{bmatrix}$$

Hp induttiva: $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix}$

$$\Rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

□

Copyright © 2004 - The McGraw - Hill Companies, srl

Algoritmo `fibonacci5`

algoritmo `fibonacci5`$(intero\ n) \rightarrow intero$

1. $M \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. **for** $i = 1$ **to** $n - 1$ **do**
3. $\quad M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
4. **return** $M[1][1]$

- Osserva che il ciclo arriva fino ad n-1, poiché come abbiamo appena dimostrato, $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$ e quindi $M[1][1]=F_n$

- Il tempo di esecuzione è $T(n)=2+n+n-1 = \Theta(n)$
- Possiamo migliorare?

Copyright © 2004 - The McGraw - Hill Companies, srl

Calcolo di potenze

- Possiamo calcolare la n-esima potenza elevando al quadrato la $\lfloor n/2 \rfloor$ - esima potenza
- Se n è dispari eseguiamo una ulteriore moltiplicazione
- Esempio: se devo calcolare 3^8:

  $3^8 = (3^4)^2 = [(3^2)^2]^2 = [(3 \cdot 3)^2]^2 = [(9)^2]^2 = [(9 \cdot 9)]^2 = [81]^2 = 81 \cdot 81 = 6561$

  $\Rightarrow$ Ho eseguito solo 3 prodotti invece di 8
- Esempio: se devo calcolare 3^7:

  $3^7 = 3 \cdot (3^3)^2 = 3 \cdot (3 \cdot (3)^2)^2 = 3 \cdot (3 \cdot (3 \cdot 3))^2 = 3 \cdot (3 \cdot 9)^2 = 3 \cdot (27)^2 = 3 \cdot (27 \cdot 27) = 3 \cdot (729) = 2187$

  $\Rightarrow$ Ho eseguito solo 4 prodotti invece di 7

Copyright © 2004 - The McGraw - Hill Companies, srl

Algoritmo `fibonacci6`

algoritmo `fibonacci6`$(\textit{intero}\ n) \rightarrow \textit{intero}$

1. $A \leftarrow \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
2. $M \leftarrow$ `potenzaDiMatrice`$(A, n-1)$
3. **return** $M[1][1]$

passaggio per **valore**

funzione `potenzaDiMatrice`$(\textit{matrice}\ A,\ \textit{intero}\ k) \rightarrow \textit{matrice}$

4. **if** $(k \leq 1)$ **then** $M \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
5. **else** $M \leftarrow$ `potenzaDiMatrice`$(A, \lfloor k/2 \rfloor)$
6. $\quad M \leftarrow M \cdot M$
7. **if** (k è dispari) **then** $M \leftarrow M \cdot A$
8. **return** M

Copyright © 2004 - The McGraw - Hill Companies, srl

Tempo di esecuzione

- Tutto il tempo è speso nella funzione `potenzaDiMatrice`
  - All'interno della funzione si spende tempo costante
  - Si esegue una chiamata ricorsiva con input $\lfloor n/2 \rfloor$
- L'equazione di ricorrenza è pertanto:

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$$

Copyright © 2004 - The McGraw - Hill Companies, srl

Metodo dell'iterazione

Si può dimostrare che

$$T(n) = \Theta(\log_2 n)$$

Infatti: $T(n)=T(\lfloor n/2 \rfloor)+\Theta(1)=(T(\lfloor n/2^2 \rfloor)+\Theta(1))+\Theta(1)=$

$$=((T(\lfloor n/2^3 \rfloor)+\Theta(1))+\Theta(1))+\Theta(1)=\ldots$$

e per $k = \lfloor \log_2 n \rfloor$ si ha $\lfloor n/2^k \rfloor = 1$ e quindi

$$T(n)=((\ldots(T(\lfloor n/2^k \rfloor)+\Theta(1))+\ldots+\Theta(1))+\Theta(1))+\Theta(1)$$

$$= T(1)+k\cdot\Theta(1) = \Theta(1) + \lfloor \log_2 n \rfloor \cdot \Theta(1) = \Theta(\log n)$$

`fibonacci6` è quindi esponenzialmente più veloce di `fibonacci5`!

Copyright © 2004 - The McGraw - Hill Companies, srl

Riepilogo

	Numero di linee di codice	Occupazione di memoria
`fibonacci1`	$\Theta(1)$	$\Theta(1)$
`fibonacci2`	$\Theta(\Phi^n)$	$\Theta(\Phi^n)$*
`fibonacci3`	$\Theta(n)$	$\Theta(n)$
`fibonacci4`	$\Theta(n)$	$\Theta(1)$
`fibonacci5`	$\Theta(n)$	$\Theta(1)$
`fibonacci6`	$\Theta(\log n)$	$\Theta(\log n)$*

* per le variabili di lavoro delle chiamate ricorsive

Copyright © 2004 - The McGraw - Hill Companies, srl