Algoritmi e Strutture Dati

Capitolo 1
Un'introduzione informale agli algoritmi: ancora sulla sequenza di Fibonacci

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*, più precisamente:

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

Foglie (Lemma 1) — # Nodi interni (Lemma 2)

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.

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) → intero
    sia Fib un array di n interi
    Fib[1] ← Fib[2] ← 1
    for i = 3 to n do
        Fib[i] ← Fib[i-1] + Fib[i-2]
    return Fib[n]
```

NOTA: Assumiamo che il primo elemento dell'array sia in Fib[1]

Correttezza? Corretto per definizione!

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

- 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

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`

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

$$T(1) = 4$$

$$T(45) = 90$$

$$T(100) = 200$$

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!

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

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

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
```

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!]

Possiamo sperare di calcolare F_n in tempo inferiore a n? Sembrerebbe impossibile…

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

Prodotto di matrici righe per colonne

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

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

$i = 1, \ldots, n$

$j = 1, \ldots, n$

Dimostrazione per induzione

Base induzione: n=2

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

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

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

□

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 $T(n)=2+n+n-1 = 2n+1$, mentre $S(n)=5$
- Possiamo migliorare?

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

Algoritmo `fibonacci6`

algoritmo `fibonacci6`(*intero* n) $\rightarrow$ *intero*

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

funzione `potenzaDiMatrice`(*matrice* A, *intero* k) $\rightarrow$ *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. $M \leftarrow M \cdot M$
7. **if** (k è dispari) **then** $M \leftarrow M \cdot A$
8. **return** M

Generica funzione che calcola la potenza di una matrice 2×2

Tempo di esecuzione

- Tutto il tempo è speso nella funzione `potenzaDiMatrice`
  - All'interno della funzione si spende tempo costante (si eseguono al più 4 istruzioni)
  - Si esegue una chiamata ricorsiva con input $\lfloor n/2 \rfloor$
- L'equazione di ricorrenza è pertanto:

$$T(n) \leq T(\lfloor n/2 \rfloor) + 4$$

Metodo dell'iterazione

Si può dimostrare che

$T(n) \le 4 \cdot (1+\log n)$ (se ometto la base del logaritmo essa è pari a 2)

Infatti: $T(n) \le T(\lfloor n/2 \rfloor)+4 \le (T(\lfloor n/2^2 \rfloor)+4)+4 \le$
$((T(\lfloor n/2^3 \rfloor)+4)+4)+4 \le \ldots$

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

$T(n) \le ((\ldots(T(\lfloor n/2^{\lfloor \log n \rfloor} \rfloor)+4)+\ldots+4)+4)+4$

$\le T(1)+\lfloor \log n \rfloor \cdot 4 = 4 + \lfloor \log n \rfloor \cdot 4 \le 4 \cdot (1+\log n)$

- `fibonacci6` è quindi esponenzialmente più veloce di `fibonacci5`!
- Si osservi infine che $T(n) \ge \log n$, poiché chiaramente vengono eseguite almeno $\log n$ chiamate ricorsive

Riepilogo

	Numero di linee di codice	Occupazione di memoria
fibonacci1	1	1
fibonacci2	$3F_n-2 \approx \Phi^n$	$\approx \Phi^n$ (*)
fibonacci3	2n	n+1
fibonacci4	4n-5	4
fibonacci5	2n+1	5
fibonacci6	$\log n \leq T(n) \leq 4(1+\log n)$	$\approx \log n$ (*)

* per le variabili di lavoro delle chiamate ricorsive