Capitolo 1

Introduzione informale agli algoritmi

Questo capitolo si basa sul Capitolo 1 del libro di testo [1].

Definizione 1.1. Un *algoritmo* è un insieme ordinato di operazioni non ambigue ed effettivamente computabili che, quando eseguito, produce un risultato e si arresta in un tempo finito.

Gli algoritmi che esegue un individuo si possono esprimere in *linguaggio naturale*. Gli algoritmi che esegue un calcolatore si devono esprimere in un *linguaggio di programmazione* (C, C++, Java...). In questo corso useremo uno *pseudocodice*: simile ad un linguaggio di programmazione, ma con meno restrizioni sintattiche. Introduciamo già una convenzione che manterremo in tutti i capitoli.

Convenzione: *in tutti i capitoli assumiamo che gli indici degli array e delle matrici partano da* 1 *(e non da* 0*, come usualmente avviene nei linguaggi di programmazione).* Ad esempio, in un array A di n elementi indicheremo le celle con $A[1], A[2], \dots A[n]$.

1.1 I numeri di Fibonacci

Iniziamo con un semplice esempio per mostrare cosa significa *risolvere un problema con un algoritmo.* Si vuole scrivere un algoritmo per il calcolo dell'n-esimo numero di Fibonacci, che è definito dalla seguente formula:

$$F_n = \begin{cases} 1 & \text{se } n = 1, 2 \\ F_{n-1} + F_{n-2} & \text{se } n \geq 3 \end{cases} \tag{1.1}$$

Saranno presentati diversi algoritmi che calcolano i numeri di Fibonacci. Evidenzieremo pregi e difetti di ognuno.

1.1.1 Algoritmo numerico

Come primo passo, sfruttiamo un risultato matematico per il calcolo dei numeri di Fibonacci. Esso si basa sul seguente[1]

Teorema 1.1 (di Just)**.** *Se t è soluzione della equazione*

$$x^2 - x - 1 = 0 \tag{1.2}$$

allora per ogni naturale $n \geq 2$ risulta:

$$t^n = t \cdot F_n + F_{n-1}$$

Dimostrazione. Osserviamo innanzitutto che se t è soluzione dell'equazione (1.2) allora vale $t^2 - t - 1 = 0$, quindi

$$t^2 = t + 1 \tag{1.3}$$

Procediamo ora per induzione su n.

[1] Ringrazio Marco Lavazza Seranto per avermi suggerito questa elegante soluzione.

Caso base: $n = 2$. Per l'osservazione in (1.3) abbiamo $t^2 = t + 1 = F_1 + F_0$.

ipotesi induttiva: sia $n > 2$ e supponiamo che il lemma sia verificato $n - 1$, ovvero

$$t^{n-1} = t \cdot F_{n-1} + F_{n-2}.$$

Passo induttivo: dimostriamo la tesi per $n > 2$. Partiamo dall'ipotesi induttiva, moltiplichiamo ambo i membri per t e continuiamo con semplici passaggi algebrici.

$$\begin{aligned}
t^n &= t^2 \cdot F_{n-1} + t \cdot F_{n-2} \\
&= (t+1) \cdot F_{n-1} + t \cdot F_{n-2} && \text{sfrutiamo l'osservazione (1.3) e sostituiamo } t^2 = t + 1 \\
&= t \cdot F_{n-1} + F_{n-1} + t \cdot F_{n-2} && \text{svolgiamo il prodotto} \\
&= t \cdot (F_{n-1} + F_{n-2}) + F_{n-1} && \text{raccogliamo } t \\
&= t \cdot F_n + F_{n-1} && \text{concludiamo, grazie alla definizione di } F_n
\end{aligned}$$

□

Ora è facile calcolare le soluzioni di (1.2), che sono:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx +1.618 \qquad \hat{\varphi} = \frac{1-\sqrt{5}}{2} \approx -0.618 \tag{1.4}$$

Per il Teorema 1.1 di Just si ha:

$$\varphi^n = \varphi \cdot F_n + F_{n-1} \qquad \hat{\varphi}^n = \hat{\varphi} \cdot F_n + F_{n-1}$$

Sottraendo membro a membro le due uguaglianze otteniamo

$$\varphi^n - \hat{\varphi}^n = \varphi \cdot F_n - \hat{\varphi} \cdot F_n = F_n \cdot (\varphi - \hat{\varphi})$$

e quindi abbiamo

$$F_n = \frac{\varphi^n - \hat{\varphi}^n}{\varphi - \hat{\varphi}}$$

Osservando che $\varphi - \hat{\varphi} = \sqrt{5}$, otteniamo la Formula di Binet che esprime i numeri di Fibonacci senza ricorrere alla ricorsione:

$$F_n = \frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}} \tag{1.5}$$

Possiamo quindi scrivere un primo algoritmo che utilizza proprio questa di una funzione matematica per calcolare direttamente i numeri di Fibonacci.

algoritmo `fibonacciNumerico` (*intero n*) → *intero*

```
1  return 1/√5 · (φ^n − φ̂^n)
```

Il limite di tale algoritmo è dato dal fatto che si è costretti ad operare con numeri reali, rappresentati nei calcolatori con precisione limitata, e quindi si possono fornire risposte errate dovute ad errori di arrotondamento. Ad esempio, se arrotondiamo i valori φ e $\hat{\varphi}$ a tre cifre decimali, come descritto in (1.4) otteniamo:

`fibonacciNumerico`(3)	=	1.99992	≈	2	corretto
`fibonacciNumerico`(16)	=	986.698	≈	897	corretto
`fibonacciNumerico`(18)	=	2583.10	≈	2583	**errato**, poiché $F_{18} = 2584$.

1.1.2 Algoritmo ricorsivo

Data la natura ricorsiva della relazione di Fibonacci, si può pensare di realizzare un algoritmo ricorsivo, come il seguente:

algoritmo `fibonacciRicorsivo` (*intero n*) → *intero*

```
1  if(n <= 2) then return 1
2  else return fibonacciRicorsivo(n - 1) + fibonacciRicorsivo(n - 2)
```

Chiediamoci ora se quello che abbiamo scritto rappresenta un algoritmo *efficiente*. Le metriche per valutare l'efficienza di un algoritmo possono essere riassunte in:

1. tempo di esecuzione
2. quantità di memoria utilizzata
3. tempo di programmazione (per scrivere il codice)
4. lunghezza del codice prodotto

Le ultime due sono studiate dall'ingegneria del software. Noi ci occuperemo delle prime due.

Per analizzare le prestazioni di un algoritmo, noi valuteremo il tempo richiesto dalla sua esecuzione (*complessità temporale*) o la quantità di memoria occupata (*complessità spaziale*): consideriamo ora la prima. Ci chiediamo: "Quanto tempo richiede l'esecuzione di questo algoritmo?"

Per rispondere a questa domanda dobbiamo prima risolverne un'altra: "Come misuriamo il tempo?" Se lo misurassimo in secondi, le prestazioni dipenderebbero dalla macchina che esegue l'algoritmo. Noi cerchiamo invece una *risposta indipendente dalle tecnologie*. In prima approssimazione possiamo contare le linee di codice eseguite da una chiamata dell'algoritmo.

Calcoliamo le linee di codice eseguite dall'algoritmo `fibonacciRicorsivo`(n) per alcuni valori di n:

- Se $n = 2$ viene eseguita una linea di codice.
- Se $n = 3$ vengono eseguite 2 righe di codice più quelle eseguite dalla chiamate ricorsive su $n = 2$ e su $n = 1$, quindi in totale: $2 + 1 + 1 = 4$ righe di codice.
- Se $n = 4$ vengono eseguite 2 righe di codice più quelle eseguite dalla chiamate ricorsive su $n = 3$ e su $n = 2$, quindi in totale: $2 + 4 + 1 = 7$ righe di codice.
- Se $n = 5$ vengono eseguite 2 righe di codice più quelle eseguite dalla chiamate ricorsive su $n = 4$ e su $n = 3$, quindi in totale: $2 + 7 + 4 = 13$ righe di codice.
- e così via...

Indichiamo con $T(n)$ il numero di righe di codice eseguite dalla chiamata `fibonacciRicorsivo`(n). Possiamo allora intuire che vale la seguente relazione

$$T(n) = 2 + T(n-1) + T(n-2) \tag{1.6}$$

Questa relazione si dice *relazione di ricorrenza*.

In generale, ogni algoritmo ricorsivo può essere analizzato mediante una *relazione di ricorrenza*: il tempo speso da una routine è pari al tempo speso all'interno della routine più quello speso dalle chiamate ricorsive.

Possiamo rappresentare le chiamate ricorsive con una struttura ad albero, detta *albero di ricorsione*: si usa un nodo, la radice dell'albero, per la prima chiamata e generiamo un figlio per ogni chiamata ricorsiva. Vediamo un esempio per la chiamata[2] `fibonacciRicorsivo`(5)

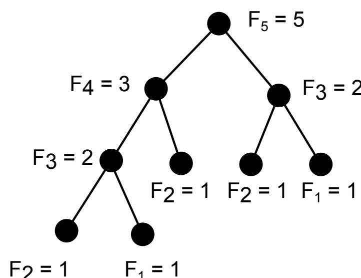

Figura 1.1: Albero di ricorsione di `fibonacciRicorsivo` per il calcolo di F_5

Sfruttando la struttura dell'albero possiamo contare il numero di righe di codice mandate in esecuzione. Ci basta contare:

[2] vi rimando alla Figura 1.5 del libro di testo [1] per la chiamata `fibonacciRicorsivo`(8).

- 2 righe di codice per ognuno dei nodi interni;
- 1 righe di codice per ognuna delle foglie.

Nel caso di `fibonacciRicorsivo`(5) abbiamo 4 nodi interni e 5 foglie. Quindi $4 \cdot 2 + 5 \cdot 1 = 13$ righe di codice.

Per calcolare le righe di codice dobbiamo semplicemente contare i nodi interni e le foglie dell'albero di ricorsione che rappresenta $T(n)$. Ci serviamo di alcuni risultati. Indichiamo con T_n l'albero delle chiamate ricorsive della funzione `fibonacciRicorsivo`(n).

Lemma 1.1. *Il numero di foglie in T_n è pari al numero di Fibonacci F_n.*

Dimostrazione. Procediamo per induzione su n.

Caso base: è banalmente verificato. Per $n = 1$ l'albero T_1 contiene un solo nodo, e dunque una sola foglia ($F_1 = 1$). Per $n = 2$ l'albero T_2 contiene anch'esso un solo nodo, e quindi una sola foglia ($F_2 = 1$).

Ipotesi induttiva: sia $n > 2$ e supponiamo che il lemma sia verificato per ogni k tale per cui $1 \leq k \leq n - 1$.

Passo induttivo: usando l'ipotesi, dimostriamo che il lemma vale per n. L'albero della ricorsione T_n ha come sottoalbero sinistro T_{n-1} e come sottoalbero destro T_{n-2}: per ipotesi essi hanno, rispettivamente, F_{n-1} e F_{n-2} foglie, dunque T_n ha $F_{n-1} + F_{n-2} = F_n$ foglie. □

Lemma 1.2. *Sia T un albero binario in cui ogni nodo interno ha esattamente due figli: allora il numero di nodi interni di T è pari al numero di foglie diminuito di uno.*

Dimostrazione. Procediamo per induzione sul numero di nodi dell'albero T considerato.

Caso base: se T ha esattamente un nodo, allora T ha una sola foglia e nessun nodo interno, quindi la condizione è verificata.

Ipotesi induttiva: sia T albero binario in cui ogni nodo interno ha esattamente due figli, inoltre T abbia più di un nodo, supponiamo che la condizione valga per tutti gli alberi con un numero di nodi strettamente minore di T.

Passo induttivo: proviamo che la condizione vale anche per T. Dobbiamo provare

$$\mathit{nodi_interni}(T) = \mathit{foglie}(T) - 1$$

Consideriamo la radice r di T. Poiché T ha più di un nodo, r deve avere almeno un figlio. Quindi la radice è un nodo interno. Poiché tutti i nodi interni di T devono avere esattamente due figli, possiamo concludere che T è composto dalla radice r e da due sottoalberi, che chiamiamo T_1 e T_2, entrambi non vuoti. Possiamo quindi affermare che

$$\mathit{nodi_interni}(T) = 1 + \mathit{nodi_interni}(T_1) + \mathit{nodi_interni}(T_2) \quad (1.7)$$
$$\mathit{foglie}(T) = \mathit{foglie}(T_1) + \mathit{foglie}(T_2) \quad (1.8)$$

I due sottoalberi T_1 e T_2 hanno le stesse proprietà di T, ma un numero minore di nodi rispetto a T. Possiamo quindi applicare l'ipotesi induttiva e otteniamo:

$$\mathit{nodi_interni}(T_1) = \mathit{foglie}(T_1) - 1 \qquad \mathit{nodi_interni}(T_2) = \mathit{foglie}(T_2) - 1 \quad (1.9)$$

Partendo dall'equazione (1.7) possiamo ottenere la proprietà cercata:

$$\begin{aligned}
\mathit{nodi_interni}(T) &= 1 + \mathit{nodi_interni}(T_1) + \mathit{nodi_interni}(T_2) \\
&= 1 + \mathit{foglie}(T_1) - 1 + \mathit{foglie}(T_2) - 1 && \text{per } (1.9) \\
&= \mathit{foglie}(T_1) + \mathit{foglie}(T_2) - 1 \\
&= \mathit{foglie}(T) - 1 && \text{per } (1.8)
\end{aligned}$$

□

Corollario 1.1. *Il numero di nodi interni in T_n è pari a $F_n - 1$.*

Dimostrazione. Osserviamo intanto che l'albero di ricorsione T_n soddisfa alle ipotesi del Lemma 1.2. Possiamo quindi applicare tale lemma e otteniamo: $nodi_interni(T) = foglie(T) - 1$. Ora, grazie al Lemma 1.1, concludiamo $nodi_interni(T) = F_n - 1$. □

Riepilogando quanto visto finora. Per contare le righe di codice eseguite da `fibonacciRicorsivo`(n) dobbiamo contare:

- due righe di codice per ogni nodo interno di T_n, ovvero $2 \cdot (F_n - 1)$ righe per il Corollario 1.1;
- una riga di codice per ogni foglia di T_n, ovvero $1 \cdot F_n$ righe per il Lemma 1.1.

In totale $3 \cdot F_n - 2$ righe di codice. Una soluzione assai inefficiente!

1.1.3 Algoritmo iterativo

La lentezza di `fibonacciRicorsivo` è dovuta al fatto che continua a ricalcolare ripetutamente la soluzione dello stesso sottoproblema (vedi, ad esempio, F_3 nell'albero di ricorsione in figura 1.1). Per fare di meglio, si potrebbe risolvere ogni sottoproblema una volta sola, memorizzarne la soluzione per utilizzarla nel seguito invece di ricalcolarla ogni volta che serve: questa è l'idea che sta alla base della tecnica chiamata *programmazione dinamica.*

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

```
1 Fib: array interi
2 Fib[1] = Fib[2] = 1
3 for i = 3 to n do
4   Fib[i] = Fib[i - 1] + Fib[i - 2]
5 return Fib[n]
```

Per quanto riguarda l'analisi temporale, non possiamo semplicemente contare quante righe di codice (distinte) vengono mandate in esecuzione. Infatti il ciclo `for` della riga 3 manda in esecuzione più volte la stessa riga (4). Occorre quindi operare in maniera diversa: per ogni linea di codice, calcoliamo quante volte essa è eseguita, esaminando a quali cicli appartiene e quante volte essi sono eseguiti. Definiamo allora:

$$T(n) \stackrel{def}{=} \sum_{k=1}^{5} \text{“ numero esecuzioni della riga } k \text{ da parte di } \texttt{fibonacciIterativo}(n)\text{”}$$

Calcoliamolo in alcuni casi. Nella seguente tabella riportiamo in numero di esecuzioni di ogni singola riga e il valore di $T(n)$ al variare di n. Vi ricordo che la riga 3 esegue il controllo $i <= n$ e viene eseguita finché tale controllo fallisce (ovvero la prima volta che $i > n$, che nel nostro caso è $i = n + 1$). La riga 4 viene eseguita solo quando il controllo fatto in 3 ha successo. Dopo l'esecuzione della riga 4, si aggiorna `i=i+1`.

n	riga 1	riga 2	riga 3	riga 4	riga 5	$T(n)$
1	1	1	1	0	1	4
1	1	1	1	0	1	4
3	1	1	2	1	1	6
4	1	1	3	2	1	8
5	1	1	4	3	1	10
6	1	1	5	4	1	12
⋮						
n	1	1	$n-1$	$n-2$	1	$2n$

In generale:

$$T(n) = \begin{cases} 4 & \text{se } n = 1, 2 \\ 1 + 1 + (n-1) + (n-2) + 1 = 2n & \text{se } n > 2 \end{cases}$$

Si tratta di una soluzione decisamente più efficiente dell'algoritmo ricorsivo proposto.

Un piccolo miglioramento

Analizziamo la quantità di memoria occupata dagli ultimi due algoritmi proposti. Vi faccio notare che conoscere la quantità di memoria utilizzata da un algoritmo è importante. Infatti, se un programma richiede una quantità di memoria eccessiva, non si ha la garanzia di ottenere i risultati della sua elaborazione: potrebbe essere richiesto più spazio rispetto a quello offerto dal disco su cui stiamo lavorando.

L'analisi della memoria occupata dipende dall'algoritmo che stiamo studiando. Per gli *algoritmi iterarivi* basta esaminare le variabili dichiarate e le chiamate di allocazione. Per gli *algoritmi ricorsivi* bisogna ricordare che lo spazio di memoria occupato in un certo momento è lo spazio totale usato da tutte le chiamate ricorsive *attive*.

Analizziamo la memoria occupata dall'algoritmo `fibonacciRicorsivo`. Osserviamo che le chiamate ricorsive aperte in un certo istante formano un *cammino* nell'albero della ricorsione associato alla chiamata `fibonacciRicorsivo`(n). Per valutare la memoria massima occupata dobbiamo quindi considerare il cammino massimo (che parte dalla radice) presente nell'albero. In pratica dobbiamo calcolare l'altezza dell'albero. Con riferimento alla Figura 1.1.5, si vede che il cammino massimo radice-foglia è quello che 'gira' sempre a sinistra scendendo lungo l'albero. Ogni volta che scendiamo di livello, verso sinistra, il sottoproblema affrontato è di una unità più piccolo rispetto al padre (di due unità se scendiamo verso destra). Possiamo quindi affermare che un cammino è lungo al più n.

Osserviamo inoltre che ogni chiamata ricorsiva richiede uno spazio costante per le variabili locali e uno spazio costante per l'indirizzo di ritorno. Quindi ogni chiamata ricorsiva richiede spazio di memoria costante. Abbiamo visto che le chiamate aperte contemporaneamente sono dell'ordine di n. Quindi concludiamo che lo spazio di memoria richiesto da `fibonacciRicorsivo`(n) è proporzionale a n (modulo fattori moltiplicativi costanti). In notazione asintotica (che studieremo nel prossimo capitolo) diremo che la memoria occupata è $\Theta(n)$.

L'algoritmo `fibonacciIterativo`(n) dichiara un array di n interi. Possiamo quindi affermare che `fibonacciIterativo` richiede una quantità di spazio di memoria linearmente proporzionale alla dimensione dell'input n. I due algoritmi, ricorsivo e iterativo, sono equivalenti dal punto di vista della memoria occupata.

Possiamo ancora migliorare le prestazioni. Osserviamo che ogni iterazione `fibonacciIterativo` utilizza solo i due valori precedenti a `fib[i]`, ovvero `fib[i - 1]` e `fib[i - 2]`. Non serve ricordare tutti i valori precedenti. Possiamo quindi rimpiazzare l'array con due variabili, come proposto nel seguente algoritmo. Otteniamo un notevole risparmio di memoria. Lo spazio richiesto diventa costante, per ogni valore di n.

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

```
1 a = 1, b = 1
2 for i = 3 to n do
3   c = a + b
4   a = b
5   b = c
6 return b
```

Analizziamo il tempo di esecuzione di questo algoritmo contando quante righe vengono mandate in esecuzione. Definiamo ancora:

$$T(n) \stackrel{def}{=} \sum_{k=1}^{6} \text{“numero esecuzioni della riga } k \text{ da parte di } \texttt{fibonacciIterativoModificato}(n)\text{”}$$

Calcoliamolo in alcuni casi

n	riga 1	riga 2	riga 3	riga 4	riga 5	riga 6	$T(n)$
1	1	1	0	0	0	1	4
2	1	1	0	0	0	1	4
3	1	2	1	1	1	1	7
4	1	3	2	2	2	1	11
5	1	4	3	3	3	1	15
$\vdots$							
n	1	$n-1$	$n-2$	$n-2$	$n-2$	1	$4n-5$

In generale:

$$T(n) = \begin{cases} 4 & \text{se } n = 1, 2 \\ 1 + (n-1) + 3(n-2) + 1 = 4n - 5 & \text{se } n > 2 \end{cases}$$

L'algoritmo `fibonacciIterativoModificato` gestisce in maniera più efficiente la memoria ma richiede più tempo di `fibonacciIterativo`. Visto che comunque il tempo richiesto rimane proporzionale a n, possiamo affermare che `fibonacciIterativoModificato` è il più efficiente tra gli algoritmi proposti finora.

1.1.4 Algoritmo basato su potenze ricorsive

Cerchiamo algoritmi per il calcolo dei numeri di Fibonacci che siano ancora più efficienti. Sfrutteremo il seguente

Lemma 1.3. *Poniamo $F_0 \stackrel{def}{=} 0$ per convenzione. Preso $n \geq 2$ vale la seguente proprietà:*

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

Dimostrazione. Procediamo per induzione su n.

Caso base: Per $n = 2$ il caso base è banalmente verificato:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{1} = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$$

Ipotesi induttiva: supponiamo valido il lemma per $n - 1$, ossia:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix}$$

Passo induttivo: dimostriamo la validità del lemma per n:

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} && \text{per ipotesi induttiva} \\ &= \begin{pmatrix} F_{n-1} + F_{n-2} & F_{n-1} \\ F_{n-2} + F_{n-3} & F_{n-2} \end{pmatrix} \\ &= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} && \text{per definizione} \end{aligned}$$

□

Sfruttiamo il Lemma 1.3 per definire un nuovo algoritmo: calcoliamo F_n computando la potenza $(n-1)$-esima della matrice indicata.

Definiamo innanzitutto una funzione che calcola il prodotto di due matrici applicando semplicemente la definizione. Ricordo che data A, matrice $n \times p$, e B, matrice $p \times m$, il prodotto $A \cdot B$ è rappresentato da C, matrice $n \times m$, definita da

$$C[i,j] \stackrel{def}{=} \sum_{k=1}^{p} A[i,k] \cdot B[k,j] \qquad \text{per } i = 1, \ldots, n \text{ e } j = 1, \ldots, m$$

algoritmo `prodottoMatrici` (*matrici A,B*) → *matrice*

```
1 for i = 1 to n
2     for j = 1 to m do
3         C[i,j] = 0
4         for k = 1 to p do
5             C[i,j] = C[i,j] + ( A[i,k] · B[k,j] )
6 return C
```

Esercizio 1.1. Date le matici $A_{n\times p}$ e $B_{p\times m}$, provate la chiamata `prodottoMatrici`(A,B) impiega un tempo proporzionale a $n{\cdot}m{\cdot}p$. In particolare, se A e B sono entrambe matrici $n\times n$ allora il tempo diventa proporzionale a n^3.

Possiamo ora scrivere l'algoritmo che sfrutta il Lemma 1.3.

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

```
1  M = (1 0; 0 1), A = (1 1; 1 0)
2  for i = 1 to n - 1 do
3      M = prodottoMatrici(M, A)
4  return M[1, 1]
```

Osserviamo che:

- la riga 1 impiega tempo costante (indipendente da n);
- l'iterazione della riga 2 viene ripetuta esattamente n volte;
- la riga 3 esegue sempre il prodotto di matrici 2×2, quindi il tempo impiegato è indipendente dal valore di n e possiamo considerarlo costante;
- la riga 4 impiega tempo costante.

Possiamo quindi concludere che il tempo impiegato dalla chiamata `fibonacciMatrice`(n) è proporzionale a n, ovvero $\Theta(n)$ in notazione asintotica.

Per quanto riguarda la memoria utilizzata, osserviamo che il numero di variabili (matrici e indici) allocate da `fibonacciMatrice`(n) è indipendente da n. Quindi lo spazio utilizzato è costante, ovvero $\Theta(1)$ in notazione asintotica.

1.1.5 Metodo dei quadrati ripetuti

Alla luce delle osservazioni fatte, possiamo dire che gli algoritmi `fibonacciIterativoModificato`(n) e `fibonacciMatrice`(n) sono equivalenti dal punto di vista delle prestazioni, infatti richiedono entrambi tempo proporzionale a n e occupano memoria costante. L'algoritmo `fibonacciMatrice` può essere ulteriormente migliorato sfruttando il metodo dei quadrati ripetuti. In pratica per calcolare la potenza n-esima della matrice si può ribilanciare la parentesizzazione del prodotto sfruttandone la proprietà associativa. Per esempio per calcolare A^8 l'algoritmo `fibonacciMatrice` calcola in sequenza

$$(\,(\,(\,(\,(\,(\,(\,A\cdot A\,)\cdot A\,)\cdot A\,)\cdot A\,)\cdot A\,)\cdot A\,)\cdot A\,)$$

ed eseguendo quindi 8 prodotti.

Possiamo invece rimaneggiare le parentesi del prodotto nel modo seguente

$$(\,(\,A\cdot A\,)\cdot(\,A\cdot A\,)\,)\cdot(\,(\,A\cdot A\,)\cdot(\,A\cdot A\,)\,).$$

Ci serve quindi calcolare 3 prodotti: $A^2=(\,A\cdot A\,)$, $A^4=(\,A^2\cdot A^2\,)$ e infine $A^8=(\,A^4\cdot A^4\,)$. Un risparmio notevole di tempo!

In generale definiamo

$$\lfloor n/2\rfloor \overset{def}{=} \begin{cases} n/2 & \text{se } n \text{ pari} \\ (n-1)/2 & \text{se } n \text{ dispari} \end{cases}$$

osservate che in ogni caso $\lfloor n/2\rfloor$ è un intero.

La potenza A^n può essere quindi definita in maniera ricorsiva come una serie di quadrati ripetuti:

$$A^n = \begin{cases} A^{\lfloor n/2\rfloor}\cdot A^{\lfloor n/2\rfloor} & \text{se } n \text{ pari} \\ A^{\lfloor n/2\rfloor}\cdot A^{\lfloor n/2\rfloor}\cdot A & \text{se } n \text{ dispari} \end{cases}$$

Utilizzando il metodo dei quadrati ripetuti per il calcolo della potenza n-esima della matrice, si ottiene il seguente algoritmo:

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

```
1 M = (1 1
       1 0)
2 potenzaDiMatrice(M, n - 1)
3 return M[0][0]

4 potenzaDiMatrice(matrice M, intero n)
5 if(n > 1) then
6   potenzaDiMatrice(M, n / 2)
7   M = M * M
8   if(n dispari)
9     M = M * (1 1
               1 0)
```

Esercizio 1.2. Assumendo che la divisione tra interi (`n/2`) arrotondi per difetto, provate la correttezza della procedura `potenzaDiMatrice`.

È evidente che il tempo impiegato da `fibonacciQuadratiRipetuti` è determinato dalla procedura `potenzaDiMatrice`. Analizziamo quindi il tempo di esecuzione di `potenzaDiMatrice`. La chiamata di `potenzaDiMatrice`(n) apre una (e solo una) chiamata ricorsiva su $n/2$. Il costo locale di una chiamata è costante, escludendo il costo delle chiamate ricorsive aperte. La complessità temporale di `potenzaDiMatrice` può essere quindi rappresentata dalla seguente relazione di ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n/2) + \Theta(1) & \text{se } n > 1 \end{cases}$$

dove abbiamo rappresentato le costanti con notazione asintotica. Tale relazione di ricorrenza è rappresentata dal seguente albero di ricorsione

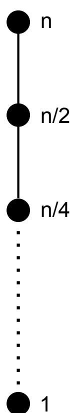

Figura 1.2: Albero di ricorsione di `potenzaDiMatrice`(n)

Quanto è alto questo albero? Osserviamo che un nodo a profondità p rappresenta un sottoproblema di dimensione $n\ /\ 2^p$. Il nodo di profondità massima rappresenta il sottoproblema di dimensione 1 (quello che non apre ulteriori chiamate ricorsive). Quindi, per calcolare l'altezza h dell'albero dobbiamo risolvere la seguente equazione

$$n\ /\ 2^h = 1$$

che ha soluzione $h = \log n$. Possiamo affermare che tutti nodi hanno un costo costante e il loro numero è proporzionale a $\log n$. Concludiamo quindi che il tempo richiesto da `potenzaDiMatrice` è proporzionale a $\log n$, ovvero in notazione asintotica

$$T(n) = O(\log n).$$

1.1.6 Esercizi Proposti

Sul libro di testo [1]: Esercizi 1.2 e 1.3; Problema 1.3.