Algoritmi e Strutture Dati

Capitolo 1
Un’introduzione informale
agli algoritmi

Ancora un esempio di problema e soluzioni algoritmiche: i numeri di Fibonacci

…verso un modello di calcolo
più simile a un computer
(rispetto alla bilancia o alla padella)

…ragionare in modo più qualitativo
sulla complessità temporale
degli algoritmi

L'isola dei conigli

Leonardo da Pisa (anche noto come Fibonacci) si interessò di molte cose, tra cui il seguente problema di dinamica delle popolazioni:

Quanto velocemente si espanderebbe una popolazione di conigli sotto appropriate condizioni?

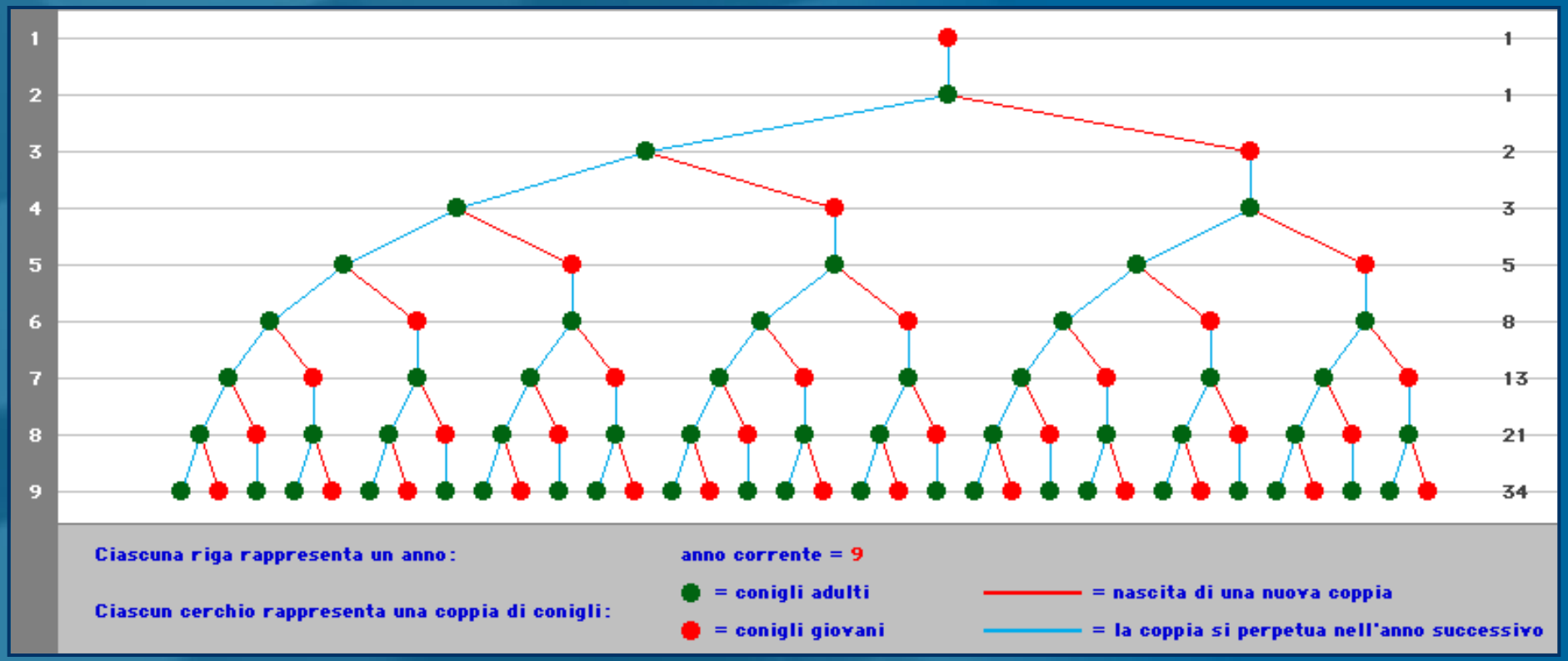
In particolare, partendo da una coppia di conigli in un'isola deserta, quante coppie si avrebbero nell'anno n?

Le regole di riproduzione

- Una coppia di conigli concepisce due coniglietti di sesso diverso ogni anno, i quali formeranno una nuova coppia
- La gestazione dura un anno
- I conigli cominciano a riprodursi soltanto al secondo anno dopo la loro nascita
- I conigli sono immortali

L'albero dei conigli

La riproduzione dei conigli può essere descritta in un albero come segue:



La regola di espansione

- Nell'anno n, ci sono tutte le coppie dell'anno precedente, e una nuova coppia di conigli per ogni coppia presente due anni prima
- Indicando con F_n il numero di coppie dell'anno n, abbiamo la seguente relazione di ricorrenza:

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

Il problema

Primi numeri della sequenza di Fibonacci:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, F_{18}=2584,…

Come calcoliamo F_n?

Un approccio numerico

- Possiamo usare una funzione matematica che calcoli direttamente i numeri di Fibonacci.
- Si può dimostrare che:

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$$

dove:

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \approx +1.618 \\ \hat{\phi} &= \frac{1-\sqrt{5}}{2} \approx -0.618 \end{aligned}$$

Algoritmo `fibonacci1`

algoritmo `fibonacci1`$(intero\ n) \rightarrow intero$

return $\frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$

Correttezza?

- Qual è l'accuratezza su Φ e $\hat{\Phi}$ per ottenere un risultato corretto?
- Ad esempio, con 3 cifre decimali:

$$\phi \approx 1.618 \text{ e } \hat{\phi} \approx -0.618$$

n	`fibonacci1`(n)	arrotondamento	F_n
3	1.99992	2	2
16	986.698	987	987
18	2583.1	2583	2584

Algoritmo `fibonacci2`

Un altro approccio: utilizzare direttamente la definizione ricorsiva:

tecnica del
divide et impera

algoritmo `fibonacci2`*(intero n)* $\rightarrow$ *intero*
  if (n$\leq$2) **then return** 1
  else return `fibonacci2`(n-1) + `fibonacci2`(n-2)

Corretto? sì! efficiente?

Tempo di esecuzione

- Un modello di calcolo rudimentale: ogni linea di codice costa un'unità di tempo

- Calcoliamo il numero di linee di codice mandate in esecuzione
  - misura indipendente dalla piattaforma utilizzata

- Se $n \leq 2$: una sola linea di codice
- Se $n=3$: quattro linee di codice, due per la chiamata `fibonacci2(3)`, una per la chiamata `fibonacci2(2)` e una per la chiamata `fibonacci2(1)`

Relazione di ricorrenza

T(n) = #linee di codice eseguite (nel caso peggiore) dall'algoritmo su input n

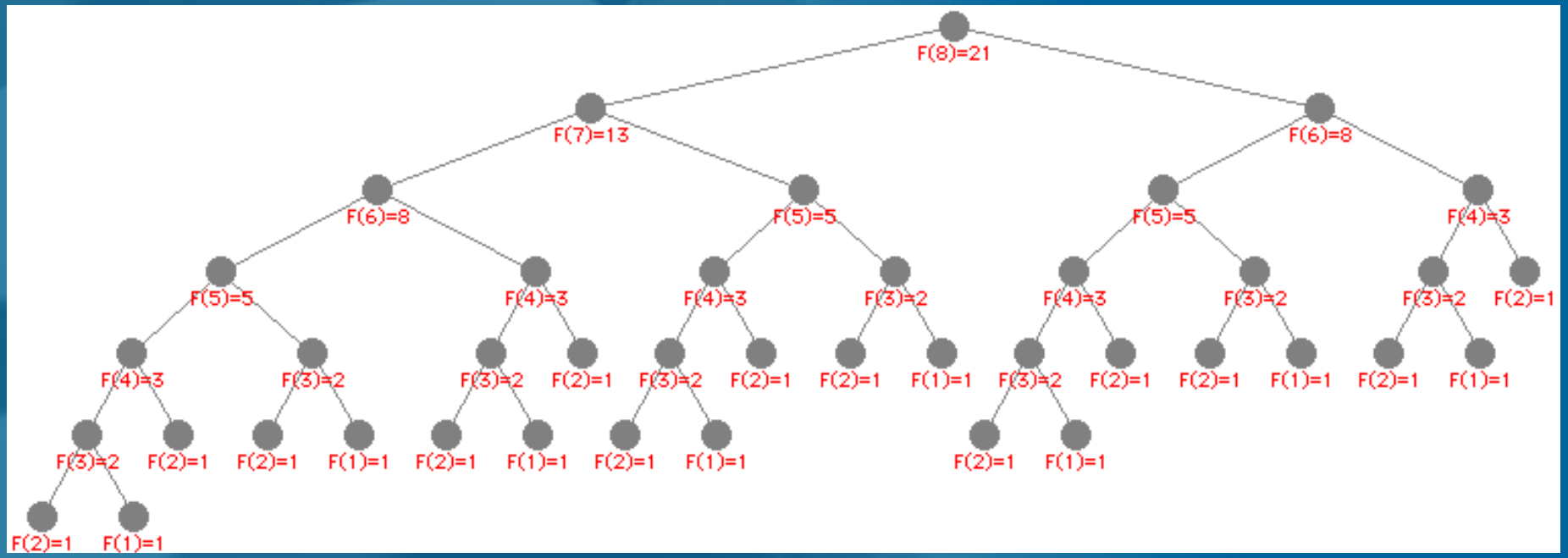
In ogni chiamata si eseguono due linee di codice, oltre a quelle eseguite nelle chiamate ricorsive

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

In generale, il tempo richiesto da un algoritmo ricorsivo è pari al tempo speso all'interno della chiamata più il tempo speso nelle chiamate ricorsive

Albero della ricorsione

- Utile per risolvere la relazione di ricorrenza
- Nodi corrispondenti alle chiamate ricorsive
- Figli di un nodo corrispondenti alle sottochiamate



Calcolare T(n)

- Etichettando i nodi dell'albero con il numero di linee di codice eseguite nella chiamata corrispondente:
  - I nodi interni hanno etichetta 2
  - Le foglie hanno etichietta 1
- Per calcolare T(n):
  - Contiamo il numero di foglie
  - Contiamo il numero di nodi interni

Calcolare T(n)

Lemma
Il numero di foglie dell'albero della ricorsione di `fibonacci2(n)` è pari a F_n
dim
(per induzione su *n*)

Lemma
Il numero di nodi interni di un albero in cui ogni nodo interno ha due figli è pari al numero di foglie -1
dim
(per induzione sul numero di nodi dell'albero *n*)

- In totale le linee di codice eseguite sono

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

Osservazioni

`fibonacci2` è un algoritmo lento:

$$T(n) \approx F_n \approx \Phi^n$$

linee di codice eseguite

$n = 8$ $T(n)=3 \cdot F_8 - 2 = 3 \cdot 21 - 2 = 61$

$n = 45$ $T(n) = 3 \cdot F_{45} - 2 = 3 \cdot 1.134903.170 - 2 = 3.404.709.508$

$n = 100\ldots$ con le attuali tecnologie, calcolare F_{100} richiederebbe circa 8000 anni!

Possiamo fare meglio?

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]*

tecnica della *programmazione dinamica*

Corretto? sì!

Calcolo del tempo di esecuzione

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

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

$$T(1) = 4$$

$$T(45) \leq 93$$

38 milioni di volte più veloce dell'algortimo `fibonacci2`!

Calcolo del tempo di esecuzione

- L'algoritmo `fibonacci3` impiega tempo proporzionale a *n* invece di esponenziale in *n* come `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!

Algoritmo `fibonacci4`

- `fibonacci3` usa un array di dimensione n prefissata
- 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)* $\rightarrow$ *intero*

```
a ← b ← 1
for i = 3 to n do
    c ← a+b
    b ← a
    a ← c
return a
```

$T(n) \leq 4n + 2$

più veloce o più lento di `fibonacci3`?

Notazione asintotica (1 di 4)

- esprimere T(n) in modo qualitativo
- perdere un po' in precisione (senza perdere l'essenziale) ma guadagnare in semplicità
- di T(n) vogliamo descrivere come cresce al crescere di n
- grossolanamente:
  - ignoro costanti moltiplicative
  - ignoro termini di ordine inferiore

- Esempio:
  - $T(n)=5n + 9 = O(n)$
  - $T(n)=6n^2-8n+13=O(n^2)$

Notazione asintotica (2 di 4)

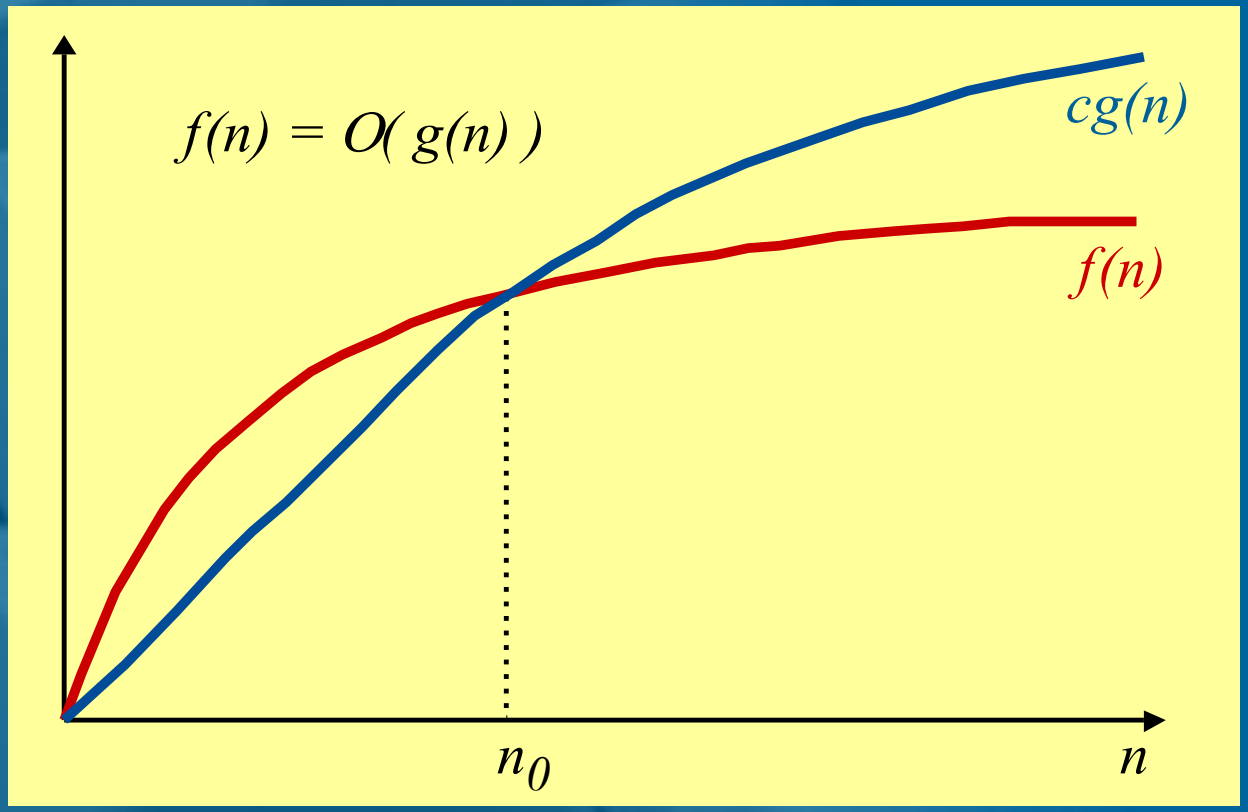
- Osservazione:
  - nel nostro caso sembra che l'approssimazione che facciamo sia "sensata"
- Infatti
  - Se andiamo a capo più spesso, aumenteranno le linee di codice sorgente, ma certo non il tempo richiesto dall'esecuzione del programma!
  - Per lo stesso programma impaginato diversamente potremmo concludere ad esempio che $T(n)=3n$ oppure $T(n)=5n$

Domanda: ma è sensato misurare la complessità di un algoritmo contando il numero di linee di codice eseguite?

…vedremo!

Notazione asintotica (3 di 4)

- Diremo che f(n) = O(g(n)) se f(n) ≤ c g(n) per qualche costante c, ed n abbastanza grande



Notazione asintotica (4 di 4)

- Ad esempio, possiamo rimpiazzare:
  - $T(n)=3F_n$ con $T(n)=O(F_n)$
  - $T(n)=2n$ e $T(n)=4n$ con $T(n)=O(n)$
  - $T(n)= F_n$ con $T(n)=O(\phi^n)$

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$$
$$\phi = 1.618...$$
$$\hat{\phi} = -0.618...$$

Un nuovo algoritmo

Possiamo sperare di calcolare F_n in tempo inferiore a O(n)?

Potenze ricorsive

- `fibonacci4` non è il miglior algoritmo possibile
- E' possibile dimostrare per induzione la seguente proprietà di matrici:

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

- Useremo questa proprietà per progettare un algoritmo più efficiente

…prodotto di matrici

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

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

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

Lemma:

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

dim

Per induzione su n

Fissiamo per convenzione $F_0=0$ (notare che $F_2=F_1+F_0$)

Caso base, n=1:

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

Lemma:

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

dim

Caso induttivo, $n > 1$:

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

per ipotesi induttiva

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

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** $M[0][0]$

- Il tempo di esecuzione è ancora O(n)
- Cosa abbiamo guadagnato?

Calcolo di potenze

- Possiampo calcolare la n-esima potenza elevando al quadrato la $(\lfloor n/2 \rfloor)$-esima potenza
- Se n è dispari eseguiamo una ulteriore moltiplicazione

- Esempio:

$3^2=9 \qquad 3^4=(9)^2=81 \qquad 3^8=(81)^2=6561$

$\Rightarrow$ Ho eseguito solo 3 prodotti invece di 7

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** $M[0][0]$

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

Tempo di esecuzione

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

$$T(n) \leq T(\lfloor n/2 \rfloor) + c \qquad \text{(c: costante)}$$

Metodo dell'iterazione

Risulta:

$$T(n) \leq c + T(\lfloor n/2 \rfloor)$$
$$\leq 2c + T(\lfloor n/4 \rfloor)$$
$$\leq 3c + T(\lfloor n/8 \rfloor)$$
$$\vdots$$
$$\leq i\,c + T(\lfloor n/2^i \rfloor)$$

Per $i = \lfloor \log_2 n \rfloor$ si ottiene

$$T(n) \leq c \lfloor \log_2 n \rfloor + T(1) = O(\log_2 n)$$

`fibonacci6` è quindi esponenzialmente più veloce di `fibonacci3`!

Due parole sulla complessità spaziale: quanta memoria usa un algoritmo?

Algoritmo non ricorsivo: dipende dalla memoria (ausiliaria) allocata; es. variabili, array, matrici, strutture dati, ecc.

Algoritmo ricorsivo: dipende dalla memoria (ausiliaria) allocata da ogni chiamata e dal numero di chiamate che sono contemporaneamente attive.

Nota: un chiamata usa sempre almeno memoria costante (anche se non utilizza variabili ausiliarie)

Analizzare l'albero della ricorsione aiuta a capire le chiamate che possono essere attive nello stesso momento.

Riepilogo

	Tempo di esecuzione	Occupazione di memoria
`fibonacci2`	$O(\phi^n)$	$O(n)$
`fibonacci3`	$O(n)$	$O(n)$
`fibonacci4`	$O(n)$	$O(1)$
`fibonacci5`	$O(n)$	$O(1)$
`fibonacci6`	$O(\log_2 n)$	$O(\log_2 n)$