



# Algoritmi e Strutture Dati

Informazioni sul corso

+

Un'introduzione informale  
agli algoritmi

Domenico Fabio Savo



# Informazioni Generali

## Domenico Fabio Savo

**Email:** [savo@dis.uniroma1.it](mailto:savo@dis.uniroma1.it)

**Web:** <http://www.dis.uniroma1.it/~savo>

**Libro di testo adottato:**

C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione, Gennaio 2008.

**Pagina web del corso:**

[http://www.dis.uniroma1.it/~savo/didattica/asd/asd1516/home\\_asd.html](http://www.dis.uniroma1.it/~savo/didattica/asd/asd1516/home_asd.html)

**Ricevimento:** guardare sulla pagina del corso.



# Obiettivi del corso

Il corso ha lo scopo di illustrare le tecniche algoritmiche e le strutture di dati fondamentali per risolvere in modo efficiente problemi computazionali.

Impareremo:

- a cosa serve la progettazione di algoritmi e delle strutture dati
- come si misura l'efficienza delle strutture dati e degli algoritmi
- come scegliere gli algoritmi e le strutture dati adatti a risolvere in modo efficiente un problema

# Definizione informale di algoritmo

Insieme di istruzioni, definite passo per passo, in modo da poter essere eseguite meccanicamente e tali da produrre un determinato risultato

- Esempio: algoritmo **preparaCaffè**

## **algoritmo preparaCaffè**

1. Svita la caffettiera.
2. Riempi d'acqua il serbatoio della caffettiera.
3. Inserisci il filtro.
4. Riempi il filtro con la polvere di caffè.
5. Avvita la parte superiore della caffettiera.
6. Metti la caffettiera, così predisposta, su un fornello acceso.
7. Spegni il fornello quando il caffè è pronto.
8. Versa il caffè nella tazzina.



# Perché studiare gli algoritmi?

- Gli algoritmi sono alla base dei programmi, nel senso che forniscono il procedimento per giungere alla soluzione di un dato problema di calcolo

# Pseudocodice

- Per mantenere il massimo grado di generalità, descriveremo gli algoritmi in pseudocodice:
  - ricorda linguaggi di programmazione reali come C, C++ o Java
  - può contenere alcune frasi in italiano

La traduzione in un particolare linguaggio di programmazione può essere fatta in modo quasi meccanico



# Correttezza ed efficienza

Vogliamo progettare algoritmi che:

- Producano correttamente il risultato desiderato
- Siano efficienti in termini di tempo di esecuzione ed occupazione di memoria



# Perché analizzare algoritmi?

- L'analisi teorica sembra essere più affidabile di quella sperimentale: vale su **tutte le possibili istanze di dati su cui l'algoritmo opera**
- Ci aiuta a scegliere tra diverse soluzioni allo stesso problema
- Permette di predire le prestazioni di un programma software, prima ancora di scriverne le prime linee di codice



# **Un esempio giocattolo: i numeri di Fibonacci**

**(un problema semplice, ma con molte soluzioni)**

# 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 genera due coniglietti ogni anno
- I conigli cominciano a riprodursi soltanto al secondo anno dopo la loro nascita
- I conigli sono immortali

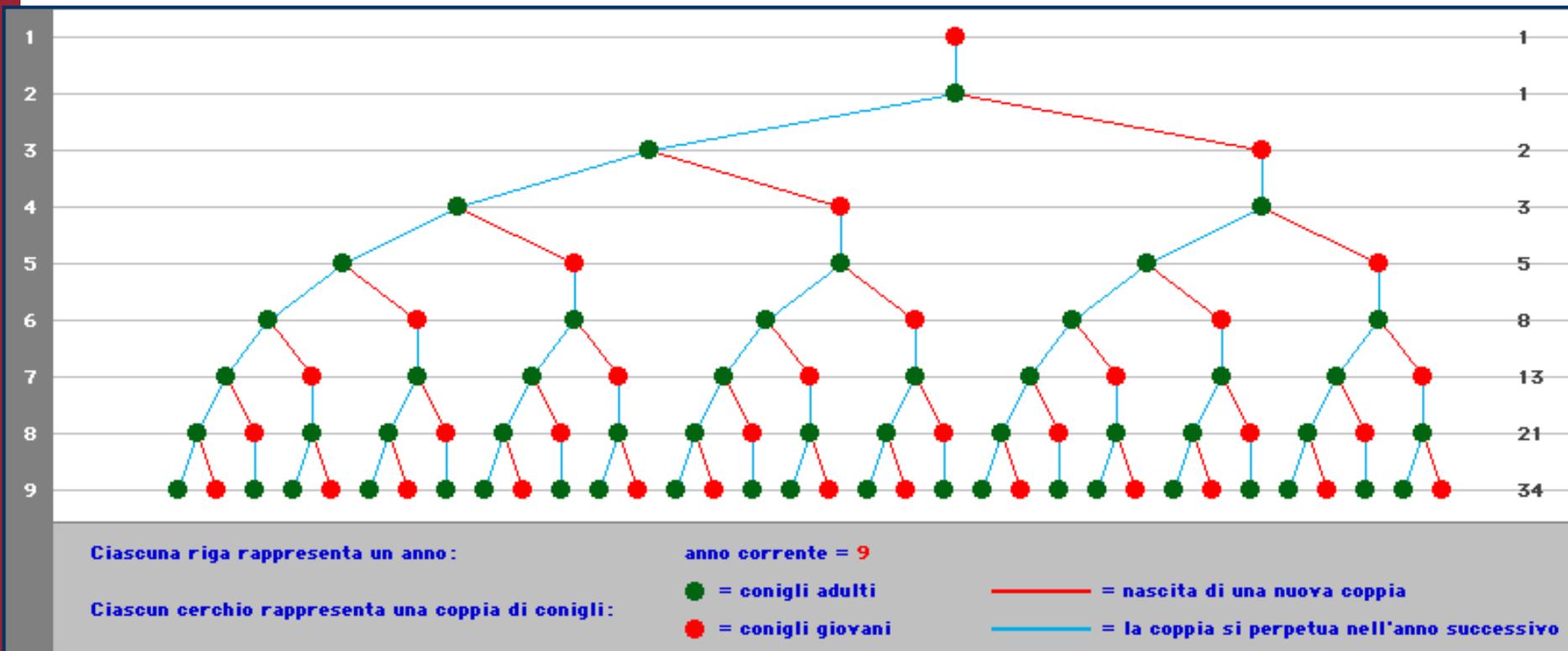
# Tasso di riproduzione dei conigli

- $F_n$  : numero di coppie di conigli presenti nell'anno  $n$
- $F_1 = 1$  (una sola coppia)
- $F_2 = 1$  (troppo giovani per riprodursi)
- $F_3 = 2$  (prima coppia di coniglietti)
- $F_4 = 3$  (seconda coppia di coniglietti)
- $F_5 = 5$  (prima coppia di nipotini)



# 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

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

# Correttezza?

- Qual è l'accuratezza su  $\Phi$  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 <sub>n</sub>
3	1.99992	2	2
16	986.698	987	987
18	2583.1	2583	2584

# Algoritmo fibonacci2

Poiché fibonacci1 non è corretto, un approccio alternativo consiste nell'utilizzare direttamente la definizione ricorsiva:

```
algoritmo fibonacci2(intero n) → intero  
  if (n ≤ 2) then return 1  
  else return fibonacci2(n-1) +  
                fibonacci2(n-2)
```

Opera solo con numeri interi



# Domande tipiche di questo corso

- Quanto tempo richiede `fibonacci2`?
- Come misuriamo il tempo?
  - In secondi (dipende dalla piattaforma)
  - In numero di istruzioni macchina (dipende dal compilatore,...)
- Prima approssimazione:
  - numero di linee di codice mandate in esecuzione (indipendente dalla piattaforma e compilatore)

# Tempo di esecuzione

- 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

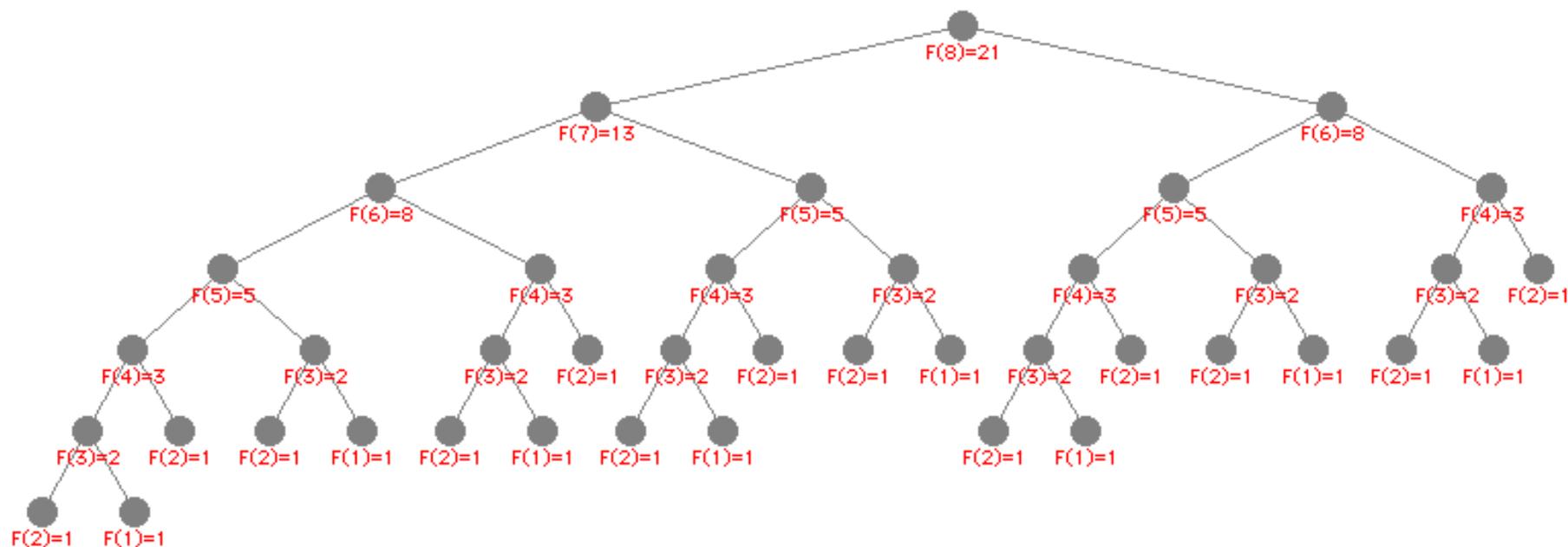
In ogni chiamata si eseguono due linee di codice, oltre a quelle eseguite nelle chiamate ricorsive

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

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





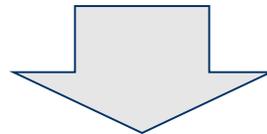
# Calcolare $T(n)$

- 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 etichetta 1
- Per calcolare  $T(n)$ :
  - Contiamo il numero di foglie
  - Contiamo il numero di nodi interni



# Calcolare $T(n)$

- Il numero di foglie dell'albero della ricorsione di `fibonacci2(n)` è pari a  $F(n)$
- Il numero di nodi interni di un albero in cui ogni nodo ha due figli è pari al numero di foglie - 1



- In totale le linee di codice eseguite sono

$$F(n) + 2 (F(n)-1) = 3F(n)-2$$



# Osservazioni

`fibonacci2` è un algoritmo lento:

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

Possiamo fare di 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; i ≤ n; i++} do  
    Fib[i] ← Fib[i-1] + Fib[i-2]  
  return Fib[n]
```

# 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:

	<code>fibonacci2(58)</code>	<code>fibonacci3(58)</code>
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  $n$
- 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; i ≤ n; i++} do  
    c ← a + b  
    a ← b  
    b ← c  
  return b
```



# Notazione asintotica (1 di 4)

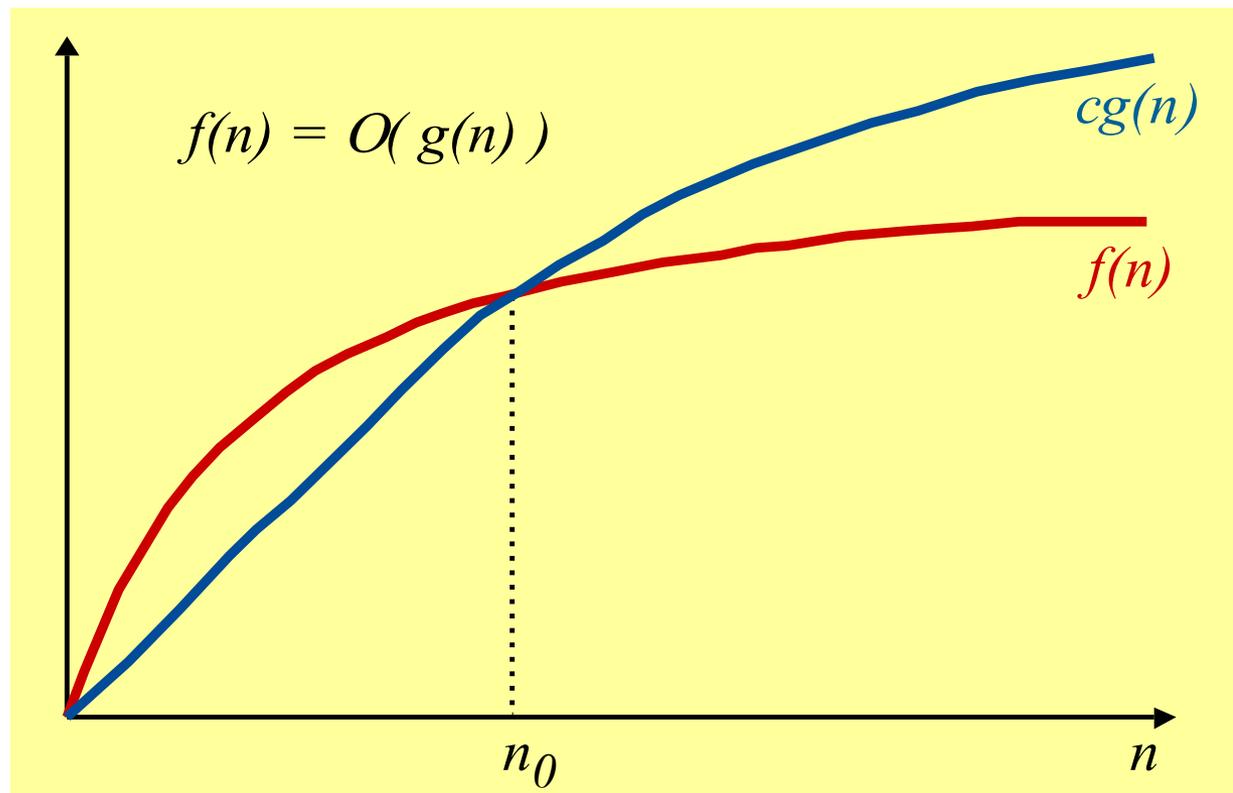
- Misurare  $T(n)$  come il numero di linee di codice mandate in esecuzione è una misura molto approssimativa del tempo di esecuzione
- Se andiamo a capo più spesso, aumenteranno le linee di codice sorgente, ma certo non il tempo richiesto dall'esecuzione del programma!

# Notazione asintotica (2 di 4)

- Per lo stesso programma impaginato diversamente potremmo concludere ad esempio che  $T(n)=3n$  oppure  $T(n)=5n$
- Vorremmo un modo per descrivere l'ordine di grandezza di  $T(n)$  ignorando dettagli inessenziali come le costanti moltiplicative...
- Useremo a questo scopo la notazione asintotica  $O$

# 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:

–  $T(n) = 3n + 4$  con  $T(n) = O(n)$

–  $T(n) = 100n + 50$  con  $T(n) = O(n)$

–  $T(n) = 2^n + 1$  con  $O(2^n)$

–  $T(n) = 2^n + n^4 + 3$  con  $O(2^n)$



# Un nuovo algoritmo

Domanda:

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

# 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.  $M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
4. **return**  $M[0][0]$

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



# Calcolo di potenze

- Possiamo calcolare la n-esima potenza di un numero moltiplicando questo numero per se stesso n volte.

$$3^8 = 3 \cdot 3 = 6561$$

- Oppure ricorrere a **quadrati ripetuti**:

$$3^2 = 9, 9^2 = 3^4 = 81, 81^2 = 3^8 = 6561$$

ovvero:  $3 \cdot 3$  poi  $9 \cdot 9$  poi  $81 \cdot 81$

Se n è dispari eseguo un'ulteriore moltiplicazione

# Algoritmo fibonacci6

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

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

**procedura** potenzaDiMatrice(*matrice*  $M$ , *intero*  $n$ )

4. **if** ( $n > 1$ ) **then**
5.     potenzaDiMatrice( $M, n/2$ )
6.      $M \leftarrow M \cdot M$
7.     **if** ( $n$  è dispari) **then**  $M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

# 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  $n/2$
- La relazione di ricorrenza è pertanto:

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

- Come risolverla?

# Metodo dell'iterazione

$$T(n) \leq c + T(n/2) \leq c + c + T(n/4) = 2c + T(n/2^2)$$

In generale:

$$T(n) \leq kc + T(n/2^k)$$

Per  $k = \log_2 n$  si ottiene

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

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



# Riepilogo

	Tempo di esecuzione	Occupazione di memoria
fibonacci2	$O(2^n)$	$O(n)$
fibonacci3	$O(n)$	$O(n)$
fibonacci4	$O(n)$	$O(1)$
fibonacci5	$O(n)$	$O(1)$
fibonacci6	$O(\log n)$	$O(\log n)$