

# *Il lato oscuro della ricorsione*

corso di laurea in **Matematica**

*Informatica Generale* Lezione **4(b)**

**Ivano Salvo**



**SAPIENZA**  
UNIVERSITÀ DI ROMA

# Numeri di Fibonacci

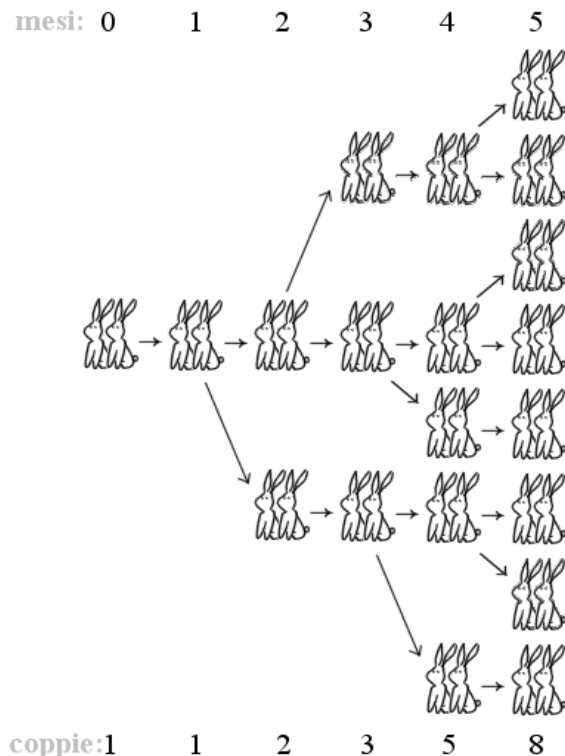
Abbiamo già visto la definizione induttiva dei **Numeri di Fibonacci**:

$$fib(n + 2) = fib(n + 1) + fib(n)$$

$$fib(1) = 1$$

$$fib(0) = 0$$

Che definisce la sequenza 0, 1, 1, 2, 3, 5, 8, 13, ...



Questa serie ha numerosissime proprietà e applicazioni.

Fu introdotta per studiare la **riproduzione dei conigli**: ogni coppia si riproduce dopo 1 mese, producendo 1 nuova coppia di conigli.

Al mese  $n+2$  appaiono quindi tante nuove coppie quante ne avevo al mese  $n-1$  più quelle che avevo al mese  $n-2$ .



# La funzione di Fibonacci

Al solito, è immediato derivare un programma ricorsivo ottenuto traducendo le equazioni ricorsive.

```
def fib(n):  
    # REQ:  $n \geq 0$   
    # ENS: return fib(n)  
    if n < 2: return n  
    return fib(n-1) + fib(n-2)
```

Ma qual è il costo computazionale di questa funzione?

$$T(n) = T(n - 1) + T(n - 2) + \theta(1) \quad \text{per } n \geq 2$$

$$T(1) = T(0) = \theta(1)$$

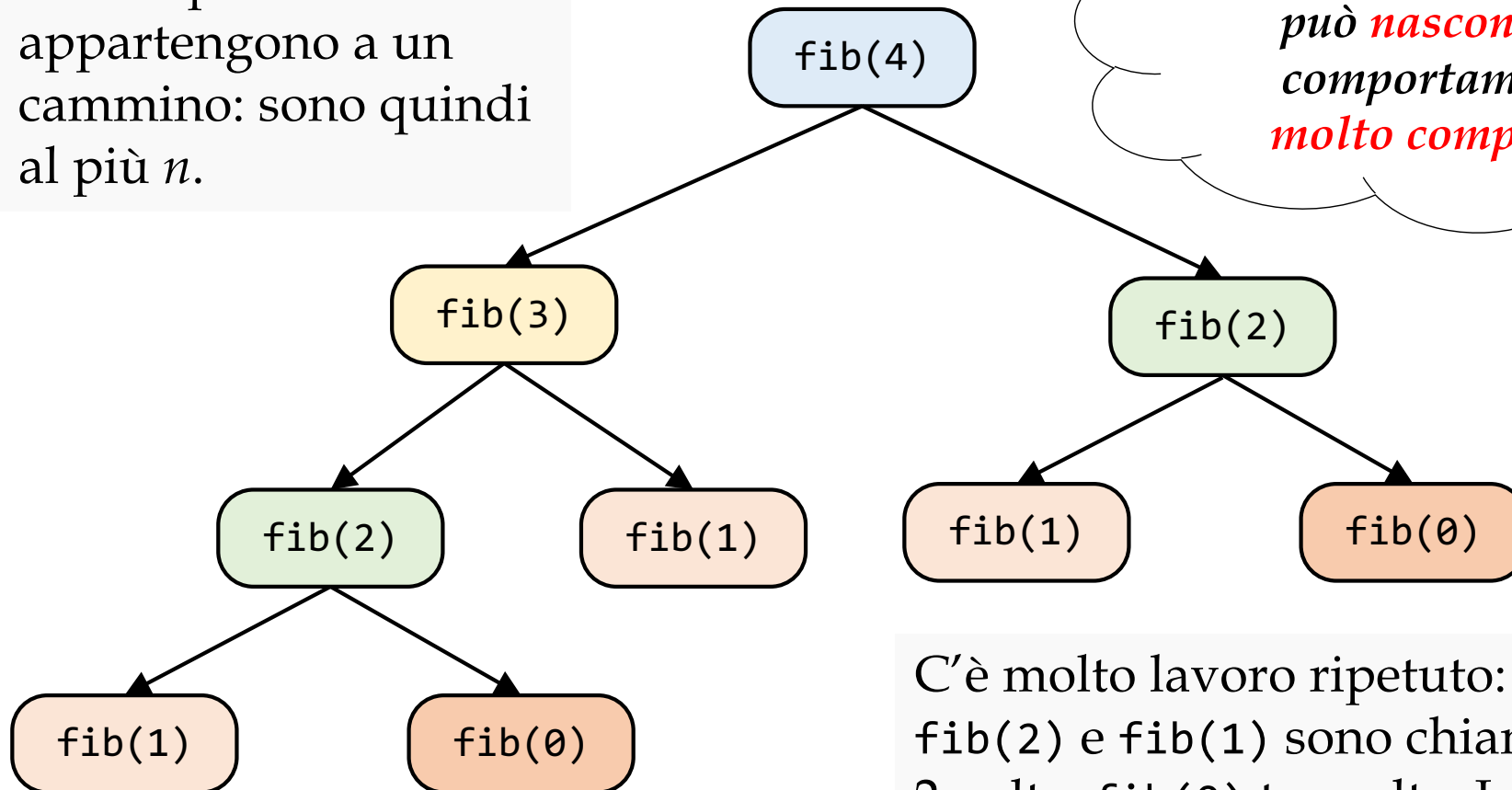
A ben vedere, l'equazione di ricorrenza ha la **stessa forma** della definizione della **serie di Fibonacci**.

Quindi, questa funzione fa  $\theta(\text{fib}(n))$  **somme** per calcolare  $\text{fib}(n)$ !

Questo si può capire anche dal fatto che la ricorsione deve scendere a  $\text{fib}(1)$  per aggiungere un 1. Quindi deve scendere  $\text{fib}(n)$  volte.

# Esecuzione di Fibonacci ricorsivo

Le chiamate “aperte” contemporaneamente appartengono a un cammino: sono quindi al più  $n$ .



*Morale: l'esecuzione di un algoritmo **ricorsivo** può **nascondere** comportamenti **molto complessi***

C'è molto lavoro ripetuto:  $\text{fib}(2)$  e  $\text{fib}(1)$  sono chiamate 2 volte,  $\text{fib}(0)$  tre volte. La cosa peggiora al crescere di  $n$ .



## *Ma quanto cresce fib(n)?*

Anche qui dobbiamo risolvere l'equazione di ricorrenza.

Maggioriamo e minoriamo (usiamo il fatto che sia **crescente**).

$$T(n) = T(n-1) + T(n-2) + \theta(1) \geq 2 T(n-2) + \theta(1) = \Omega(2^{n/2})$$

$$T(n) = T(n-1) + T(n-2) + \theta(1) \leq 2 T(n-1) + \theta(1) = \mathcal{O}(2^n)$$

Abbiamo che i numeri di Fibonacci crescono **esponenzialmente**, ma **meno di  $2^n$**

Alternativamente, assumiamo di conoscere il limite  $\phi = T(n+1)/T(n)$  [**metodo di sostituzione**]. Allora ho che:

$$\phi = \frac{T(n) + T(n-1)}{T(n)} = 1 + \frac{T(n-1)}{T(n)} = 1 + \frac{1}{\phi}$$

Da cui  $\phi > 0$  è la soluzione positiva di  $\phi^2 - \phi - 1 = 0$ . Cioè  $\phi = \frac{1+\sqrt{5}}{2}$ .

Abbiamo che i numeri di Fibonacci crescono **esponenzialmente** come  $\phi^n$ .

# *Fibonacci iterativo*

Dovrebbe essere chiaro che conoscendo i primi  $n$  numeri di Fibonacci, posso calcolare l' $n+1$ -esimo semplicemente **sommando gli ultimi due**. Quindi bastano  $n$  **somme** per calcolare  $fib(n)$ .

Infine, durante il calcolo **non è necessario** memorizzare **tutta la sequenza**, ma solo gli ultimi due numeri calcolati: quindi basta avere due variabili  $fib$  e  $fibP$  (=precedente) e mantenere l'invariante  $fib=fib(i)$  e  $fibP=fib(i-1)$  fino a che  $i$  non diventa  $n$ .

```
def fib(n):  
    # REQ:  $n \geq 0$ , ENS: return fib(n)  
    if n<2: return n  
    fib, fibP, i = 1, 0, 1 # fib(1)=1=fib, fib(0)=0=fib(i-1)  
    while i!=n:  
        # INV: fib = fib(i) & fibP = fib(i-1)  
        fib, fibP, i = fib+fibP, fib, i+1  
    return fib
```



# Ma è colpa della ricorsione?

**Ovviamente no!** È possibile scrivere un programma **ricorsivo** che ha **lo stesso comportamento** di quello **iterativo** appena visto.

Idea: **memorizzare** sui **parametri** lo **stato della computazione** durante il **while**. Nel nostro caso, le variabili **fib**, **fibP**, **i** ed **n**.

Dopodiché ogni chiamata ricorsiva avrà lo stesso comportamento di una iterazione del **while** nel programma precedente

È preferibile che la funzione **fibRec** abbia **l'interfaccia attesa** con un solo parametro e inneschi la funzione ausiliaria **fibRecAux**.

*manteniamo  
l'interfaccia  
originale*

```
def fibRec(n):  
    # REQ:  $n \geq 0$   
    # ENS: return fib(n)  
    if n < 2: return n  
    return fibRecAux(n, 2, 1, 1)
```

```
def fibRecAux(n, i, fib, fibP):  
    # REQ:  $i \leq n$  &  $fib = fib(i)$  &  $fibP = fib(i+1)$   
    # ENS: return fib(n)  
    if i == n: return fib  
    return fibRecAux(n, i+1, fib+fibP, fib)
```

*La prima chiamata  
soddisfa le precondizioni  
della funzione ausiliaria*



# *Un conto in sospeso: predecessore*

A questo punto dovrebbe essere chiaro come mimare **qualunque iterazione** con una **funzione ricorsiva**.

Siamo pronti quindi a scrivere la funzione **predecessore**, nel caso sfortunato in cui il nostro esecutore sia in grado di fare +1, ma non -1: è sufficiente mimare le versioni iterative, usando i parametri.

```
def pred(n):  
    # REQ: n > 0  
    # ENS: return n-1  
    return predAux(0, n)
```

```
def predAux(i, n):  
    # REQ: i < n  
    # ENS: return n-1  
    if i+1==n: return i  
    return predAux(i+1, n)
```

La complessità di predAux è data dall'equazione di ricorrenza:

$$T(n) = T(n-1) + \theta(1) \quad T(1) = \theta(1)$$

Srotoliamo col metodo **iterativo**:

$$\begin{aligned} T(n) &= T(n-1) + \theta(1) = T(n-2) + \theta(1) + \theta(1) = T(1) + \theta(1) + \dots + \theta(1) = \\ &= \sum_{i=1, \dots, n} \theta(1) = n \theta(1) = \theta(n) \end{aligned}$$

# *Iterazione vs Ricorsione reloaded*

A questo punto, dovrebbe essere chiaro che **ogni iterazione** si può facilmente **simulare** con una **ricorsione**.

È **meno evidente il contrario**, ma vedremo come sia possibile aiutandosi con opportune **strutture dati**.

Ad esempio **non è ovvio** scrivere un programma **iterativo** che risolve il problema della **Torre di Hanoi** o che **ha lo stesso comportamento** del calcolo ricorsivo **inefficiente di Fibonacci**.

In un mondo di **sole chiamate ricorsive** possiamo anche **fare a meno dell'assegnazione (!!)** e abbiamo il seguente risultato, analogo del Teorema di Böhm-Jacopini per il mondo della ricorsione.

**Teorema:** *Ogni funzione (parziale) calcolabile  $f: \mathbb{N}^k \rightarrow \mathbb{N}$  può essere calcolata usando una macchina capace di sommare 1, verificare se due espressioni siano uguali o diverse, e di eseguire funzioni definite per ricorsione.*

# Correttezza delle funzioni ricorsive

La **correttezza** di una **funzione ricorsiva** si stabilisce per **induzione**.

Si **assume induttivamente** che le **chiamate ricorsive** siano corrette (occorre **verificare** che le **chiamate ricorsive rispettino** le **precondizioni**) e se ne deduce la correttezza della funzione.

Nel caso di `fibRecAux`, la correttezza del **caso base** ( $i = n$ ) discende banalmente dalla precondizione  $fib = fib(i) = fib(n)$  perché  $i = n$ .

Il **passo induttivo** è banale perché discende dalla correttezza della chiamata ricorsiva.

Devo però controllare che  $fib + fibP = fib(i+1)$  e questo discende dalle precondizioni  $fib = fib(i)$  e  $fibP = fib(i-1)$  unita alla definizione di Fibonacci. Inoltre, se non esco per il caso base ( $i = n$ ), significa che  $i < n$  e quindi  $i \leq n$  rispetta la prima precondizione.

```
def fibRecAux(n, i, fib, fibP):  
    # REQ:  $i \leq n$  &  $fib = fib(i)$  &  $fibP = fib(i-1)$ , ENS: return fib(n)  
    if i==n: return fib  
    return fibRecAux(n, i+1, fib+fibP, fib)
```

# Usare i valori di ritorno

Prima abbiamo usato i parametri per far comunicare due attivazioni della funzione ricorsiva fibRecAux.

Nel nostro pseudoPython, possiamo usare i valori tornati, ritornando più di un valore, ad esempio **due numeri consecutivi di fibonacci**.

**Esercizio:** eseguire con carta e penna la funzione fibRecAux.

**Esercizio:** dimostrare la correttezza di questa versione.

```
def fibRec(n):  
    # REQ:  $n \geq 0$   
    # ENS: return fib(n)  
    if  $n < 2$ : return n  
    fib, fibP = fibRecAux(n)  
    return fib
```

```
def fibRecAux(n):  
    # REQ:  $2 \leq n$   
    # ENS: return fib(n), fib(n-1)  
    if  $n == 2$ : return 1, 1  
    fib, fibP = fibRecAux(n-1)  
    return fib+fibP, fib
```