

Esercizio con idea di soluzione (1)

(dal compito d'esame del 18/2/2021)

Sia dato un array A di interi e due valori a e b con $a \leq b$, vogliamo sapere quanti elementi di A sono compresi nell'intervallo chiuso $[a, b]$.

- Si progetti un algoritmo per risolvere tale problema su qualsiasi array A .
- Si progetti un algoritmo più efficiente del precedente assumendo che A sia già ordinato e che contenga solo valori distinti.

Per ciascun algoritmo si descriva a parole l'idea, si scriva lo pseudocodice e si analizzi il tempo di esecuzione asintotica

Esercizio con idea di soluzione (2)

Prima parte (array disordinato con possibili ripetizioni)

Scorro l'array e conto gli elementi inclusi in $[a,b]$:

```
def Conta_in_Intervallo (A, a,b)
    cont = 0;
    for i in range len(A) :
        if a ≤ A[i] ≤ b
            cont++;
    return cont
```

Costo: $\Theta(n)$

Esercizio con idea di soluzione (3)

Seconda parte (array ordinato senza ripetizioni)

Cerco a e b nell'array con la ricerca binaria. Serve una piccola modifica nel caso questi due valori non siano nell'array: se l'elemento cercato non c'è, invece di ritornare -1 ritorno l'indice m.

Esercizio con idea di soluzione (4)

(seconda parte - segue)

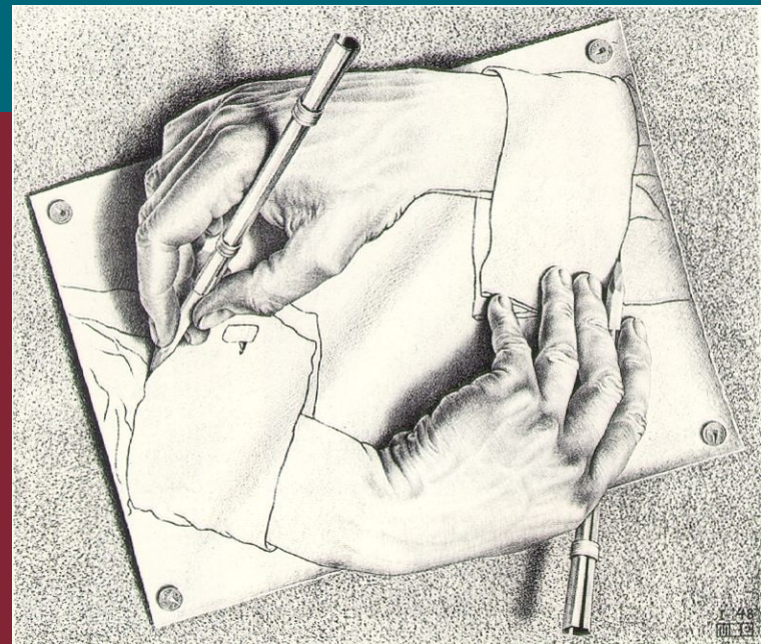
```
def Conta_in_Intervallo_Ordinato (A, a,b)
    ind_a = 0; ind_b = 0;
    ind_a = Ricerca_binaria_modificata(A,a);
    if A[ind_a]<a ind_a++;
    ind_b = Ricerca_binaria_modificata(A,b);
    if A[ind_b]>b ind_b--
    return ind_b-ind_a+1
```

Costo: $O(\log n)$

Corso di laurea in Informatica
Algoritmi 1
A.A. 2025/26

La ricorsione

Tiziana Calamoneri



SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Ricerca binaria ricorsiva (1)

Consideriamo una diversa formulazione dell'algoritmo di ricerca binaria, nella quale sono volutamente tralasciati i dettagli per catturarne l'essenza:

```
RicBin(A, v)
```

```
    se A è vuoto restituisci -1
```

```
    ispeziona l'elemento A[centrale]
```

```
    se è uguale a v: restituisci il suo indice
```

```
    se  $v < A[\text{centrale}]$ 
```

```
        esegui RicBin(metà sinistra di A, v)
```

```
    se  $v > A[\text{centrale}]$ 
```

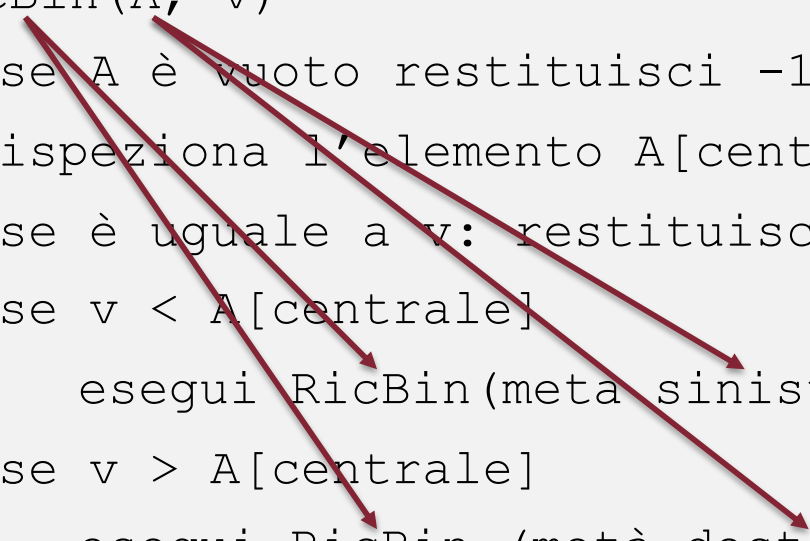
```
        esegui RicBin(metà destra di A, v)
```

Ricerca binaria ricorsiva (2)

L'aspetto cruciale di questa formulazione risiede nel fatto che l'algoritmo risolve il problema “riapplicando” se stesso su un sottoproblema (una delle due metà dell'array).

Questa tecnica si chiama **ricorsione**.

```
RicBin(A, v)
    se A è vuoto restituisci -1
    ispeziona l'elemento A[centrale]
    se è uguale a v: restituisci il suo indice
    se  $v < A[\text{centrale}]$ 
        esegui RicBin(metà sinistra di A, v)
    se  $v > A[\text{centrale}]$ 
        esegui RicBin (metà destra di A, v)
```

The diagram illustrates the recursive process of the binary search algorithm. It features four red arrows originating from the function name 'RicBin' in the first line of the code. One arrow points to the 'v' parameter in the first recursive call 'RicBin(metà sinistra di A, v)'. Another arrow points to the 'v' parameter in the second recursive call 'RicBin (metà destra di A, v)'. A third arrow points to the 'A' parameter in the first recursive call. A fourth arrow points to the 'A' parameter in the second recursive call. These arrows visually represent how the function calls itself with modified arguments to solve subproblems.

Funzioni matematiche ricorsive

Una funzione matematica è detta **ricorsiva** quando la sua definizione è espressa in termini di se stessa.

Esempio: $n! = n \cdot (n - 1)!$ Se $n > 0$

$$0! = 1$$

prima riga: meccanismo di calcolo ricorsivo

seconda riga: caso base



una funzione matematica ricorsiva deve sempre avere un caso base!

Algoritmi ricorsivi (1)

Nel campo degli algoritmi vi è un concetto del tutto analogo, quello degli algoritmi ricorsivi:

un algoritmo è detto ricorsivo quando è espresso in termini di se stesso.

Algoritmi ricorsivi (2)

Un algoritmo ricorsivo ha sempre queste proprietà:

- la soluzione del problema complessivo è costruita risolvendo (ricorsivamente) uno o più sottoproblemi di dimensione minore e combinando poi queste soluzioni;
- la successione dei sottoproblemi, che sono sempre più piccoli, deve sempre convergere ad un sottoproblema che costituisca un **caso base** (detto anche **condizione di terminazione**), in cui la ricorsione termina.

Algoritmi ricorsivi (3)

Esempio: Calcolo del fattoriale di un intero dato n.

```
def Fattoriale_Iter (n): // n: intero non neg.  
    fatt = 1  
    for i in range(1, n+1):  
        fatt = fatt*i  
    return fatt
```

```
def Fattoriale_Ric (n): // n : intero non neg.  
    if (n == 0) return 1  
    return n*Fattoriale_Ric (n - 1)
```

Algoritmi ricorsivi (4)

- Se una funzione ricorsiva non ha un caso base, la sua esecuzione genererà una catena illimitata di chiamate ricorsive che non terminerà mai



Accertiamoci che il caso base non possa essere mai “saltato”



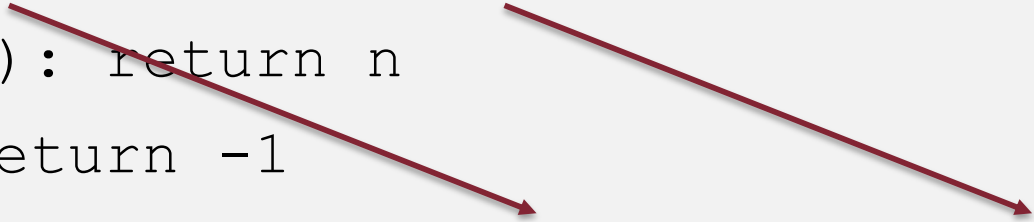
- Nulla vieta di prevedere più di un caso base, basterà assicurarsi che **uno tra i casi base sia sempre e comunque incontrato.**

Algoritmi ricorsivi (5)

Esempio: algoritmo ricorsivo per la ricerca sequenziale su n elementi:

- ispezioniamo l' n -esimo elemento;
- se non è l'elemento cercato risolviamo il problema ricorsivamente sui primi $(n - 1)$ elementi.

```
def Ricerca_seq_ric(A, v, n=len(A)-1):  
    if (A[n]==v): return n  
    if (n==0) return -1  
    else return Ricerca_seq_ric(A, v, n - 1)
```

Two red arrows originate from the recursive call 'Ricerca_seq_ric(A, v, n - 1)' in the code. One arrow points to the 'n' parameter in the 'if (A[n]==v): return n' line, and the other points to the 'n' parameter in the 'if (n==0) return -1' line, illustrating the recursive process.

Algoritmi ricorsivi (6)

Esempio: algoritmo ricorsivo per la ricerca binaria su n elementi:

- ispezioniamo l'elemento centrale;
- se non è l'elemento cercato risolviamo il problema ricorsivamente **su una sola** metà dell'array.

Algoritmi ricorsivi (7)

```
def Ricerca_bin_ric (A, v, i_min=0, i_max=len(A)):  
    if (i_min > i_max): return -1  
    m =(i_min + i_max)//2  
    if (A[m]==v): return m  
    elif (A[m] > v):  
        return Ricerca_bin_ric (A, v, i_min, m - 1)  
    else:  
        return Ricerca_bin_ric (A, v, m + 1, i_max)
```

Algoritmi ricorsivi (8)

Nella ric. bin. ogni nuova chiamata ricorsiva riceve un sotto-problema da risolvere la cui dimensione è circa la metà di quello originario quindi, come nella versione iterativa, si giunge molto rapidamente al caso base.

Da ciò deriva un costo computazionale di

$$O(\log n)$$

nel caso peggiore come per la ricerca binaria iterativa.

[Da dimostrare poi...]

La ricorsione (1)

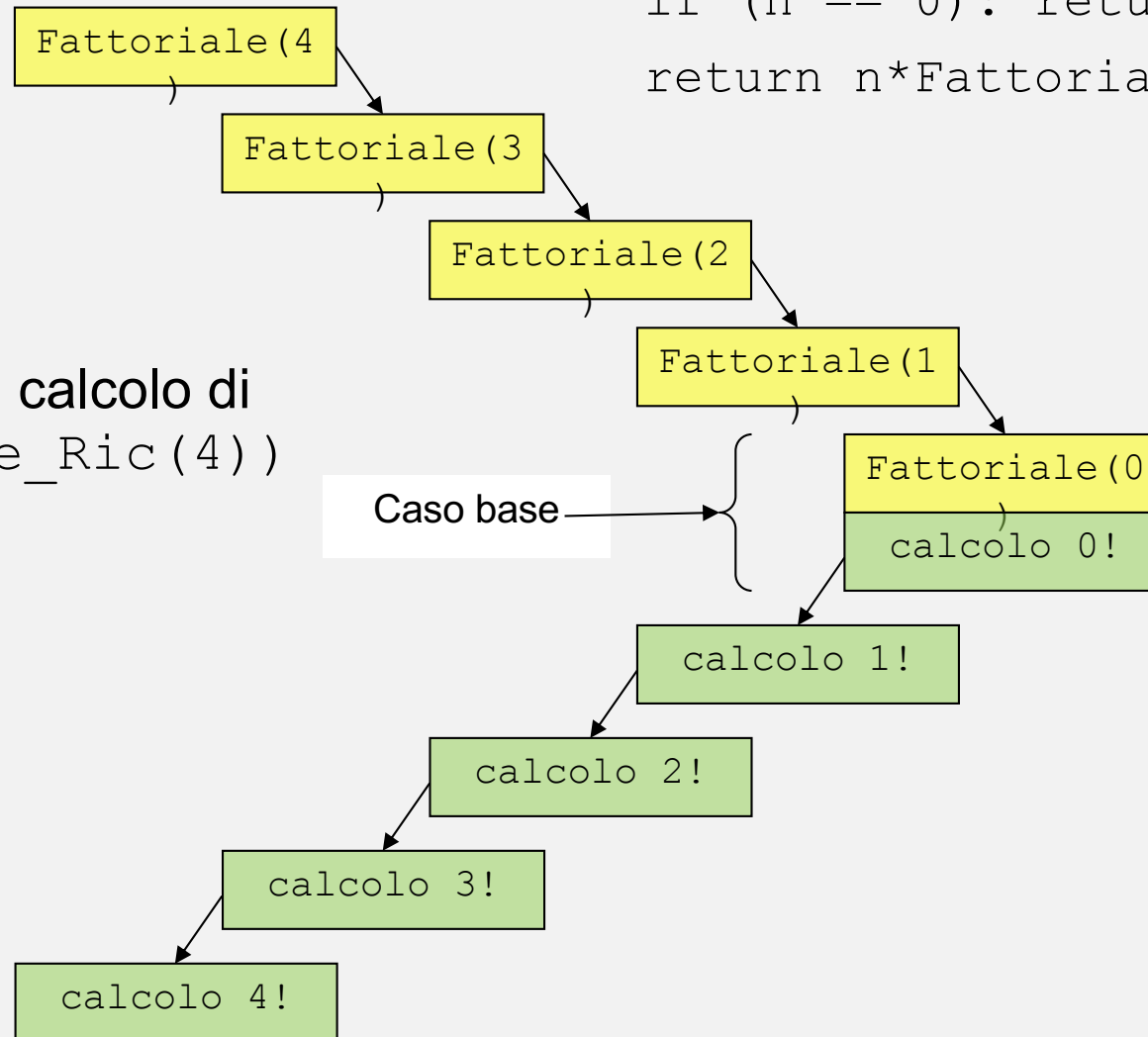
Ripassiamo come si sviluppa il procedimento di calcolo effettivo di una funzione ricorsiva sull'esempio del fattoriale:

```
def Fattoriale_Ric (n)
    if (n == 0): return 1
    return n*Fattoriale_Ric (n - 1)
```

La ricorsione (2)

```
def Fattoriale_Ric (n)
    if (n == 0): return 1
    return n * Fattoriale_Ric (n - 1)
```

(sviluppo del calcolo di
`Fattoriale_Ric(4)`)



La ricorsione (3)

💡 qualsiasi problema risolvibile con un algoritmo ricorsivo può essere risolto anche con un algoritmo iterativo.

meglio **ricorsivo** quando la formulazione del problema è inerentemente ricorsiva, mentre la soluzione iterativa è molto più complicata o addirittura non evidente

Cosa usare?



meglio **iterativo** quando:

- esiste una soluzione iterativa altrettanto semplice e chiara;
- l'efficienza è un requisito primario

La ricorsione (4)

Infatti:

ogni funzione, ricorsiva o no, richiede per la sua esecuzione una certa quantità di memoria, per:

- caricare in memoria il suo codice;
- passare i parametri;
- memorizzare le sue variabili locali.

Le funzioni ricorsive in generale hanno maggiori esigenze in termini di memoria, rispetto alle funzioni iterative.

La ricorsione (5)



Una funzione ricorsiva mal progettata, (ad es. senza la condizione di terminazione) esaurisce con estrema rapidità la memoria disponibile con la conseguente inevitabile terminazione forzata dell'esecuzione.

La ricorsione (6)

Esempio: calcolo dell'n-esimo numero di Fibonacci

Definizione dei numeri di Fibonacci:

- $F(0) = 0$
- $F(1) = 1$
- $F(i) = F(i - 1) + F(i - 2)$ se $i > 1$



Leonardo Fibonacci
(1170-1250)

Esiste una formula chiusa per il calcolo (formula di Binet), che però introduce errori numerici dovuti agli inevitabili arrotondamenti causati dai calcoli in virgola mobile:

$$F(n) = \frac{\Phi^n}{\sqrt{5}} - \frac{(1-\Phi)^n}{\sqrt{5}} \quad \text{con } \Phi = \frac{1+\sqrt{5}}{2}$$

La ricorsione (7)

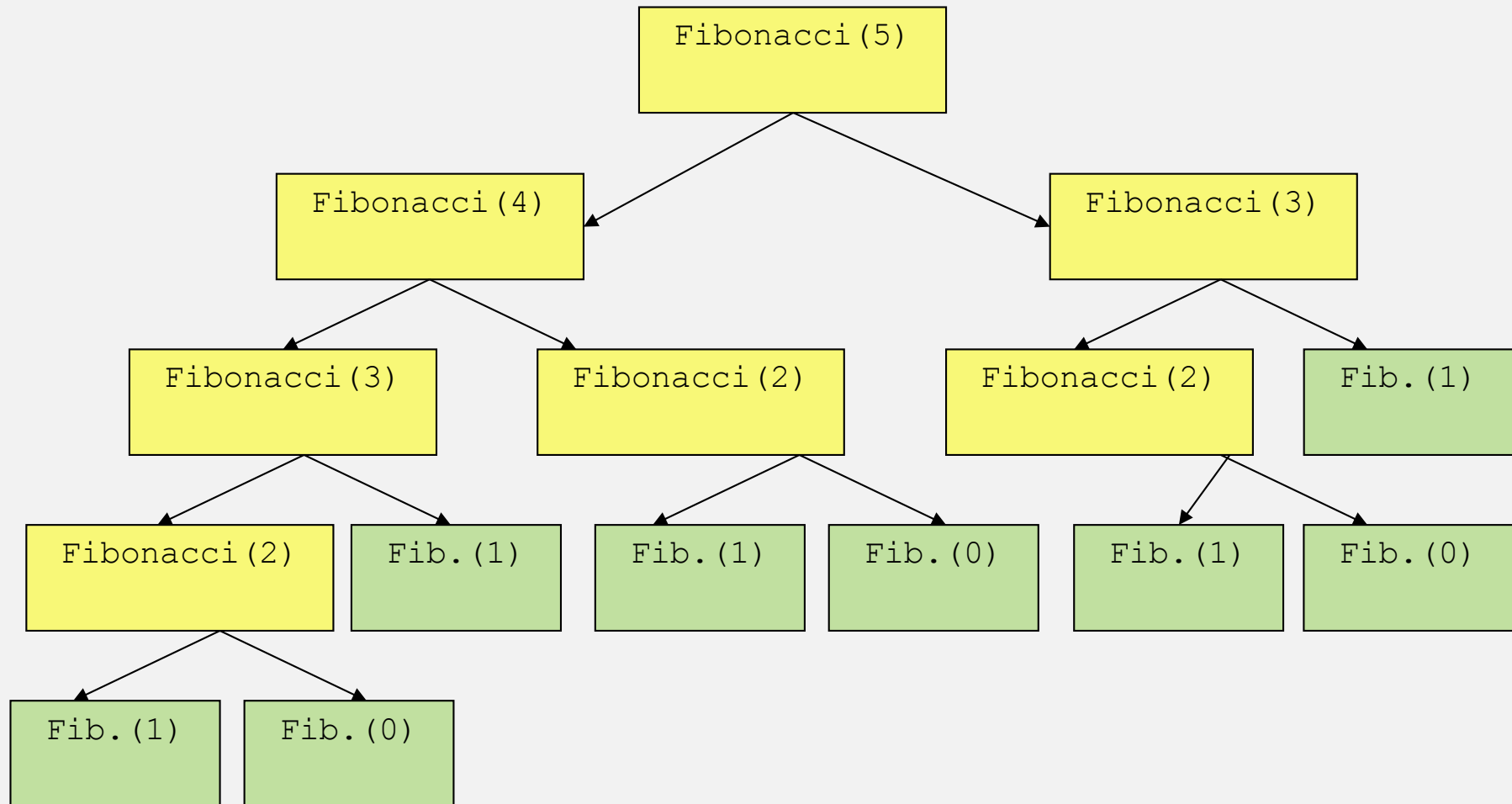
... quindi impostiamo il calcolo mediante la seguente funzione ricorsiva:

```
def Fibonacci (n):  
    if (n <=1): return n  
    return (Fibonacci(n - 1) + (Fibonacci(n - 2)))
```



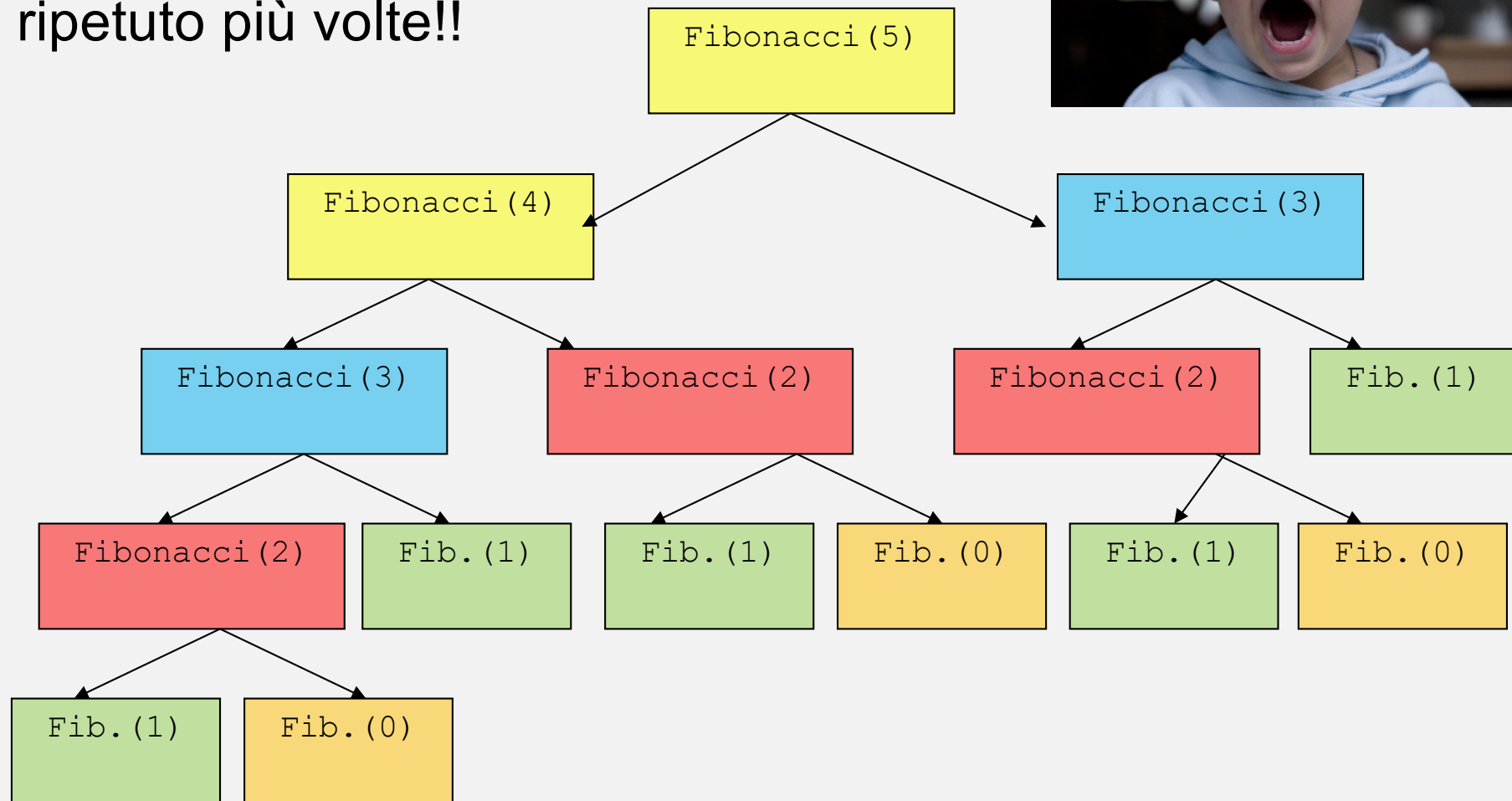
nel corpo della funzione, ci sono due chiamate ricorsive, non una sola come nel caso della ricerca binaria. Questo si riflette in una più complessa articolazione delle chiamate di funzione:

La ricorsione (8)



La ricorsione (9)

Oss. 1: uno stesso calcolo viene ripetuto più volte!!



La ricorsione (10)

Oss. 2: L'occupazione dello spazio di memoria, legata alle chiamate di funzione, è molto alta:

- Il numero totale delle chiamate effettuate cresce molto velocemente con n .
- Quando si arriva a ciascun caso base vi è una catena di chiamate “aperte” lungo il percorso che va dalla chiamata iniziale al caso base.

Oss. 3: Per risolvere un problema di dimensione n si devono risolvere due sottoproblemi di dimensione ben poco inferiore (rispettivamente, $n-1$ ed $n-2$).

La ricorsione (11)

La versione iterativa della funzione per il calcolo del numero di Fibonacci:

```
def Fibonacci_iter(n):  
    if (n <= 1): return n  
    fib_prec_prec = 0  
    fib_prec = 1  
    for i in range(2, n+1):  
        fib_prec_prec, fib_prec = fib_prec, fib_prec + fib_prec_prec  
    return fib_prec
```

ha costo in tempo pari a $\Theta(n)$ ed in spazio pari a $\Theta(1)$



La ricorsione (12)

Qual è il costo computazionale della versione ricorsiva?

Funzione Fib (n)

if (n<=1))	$\Theta(1)$
return n	$\Theta(1)$
return(Fib (n-1) + (Fib (n-2))	$T(n-1)+T(n-2)$

Otteniamo un costo $T(n)=\Theta(1)+T(n-1)+T(n-2)$ che è simile alla formula che definisce i numeri di Fibonacci, per cui prevediamo sia un costo esponenziale. Ma come lo dimostriamo?



Con le **equazioni di ricorrenza** (in seguito)

La ricorsione – esercizio 1

Dati in input n e k interi, calcolare la potenza k -esima di n



possiamo basarci sul fatto che $n^k = n \cdot n^{k-1}$ ed $n^0 = 1$.

```
def Enne_alla_Kappa (n, k)
    if (k=0): return 1
    return n*Enne_alla_Kappa (n, k-1)
```

Costo:

Se $k=0$: $T(n,k) = \Theta(1)$

Se $k>0$: $T(n,k) = \Theta(1) + T(n,k-1)$



n è un parametro inutile per il costo computaz.

La ricorsione – esercizio 2

Dato in input un array, calcolare la somma dei suoi elementi.



la somma è pari al valore di un elemento più la somma calcolata ricorsivamente sul resto dell'array

```
def SommaArray (A, ult_ind=len(A)-1)
    if (ult_ind==0):
        return A[ult_ind]
    return (A[ult_ind] + SommaArray(A, ult_ind-1))
```

OSSERVAZIONE:

Non è lo stesso eliminare l'elemento più a destra o quello più a sinistra...

Costo:

Se $n=1$: $T(n)=\Theta(1)$

Se $n>1$: $T(n)=\Theta(1)+T(n-1)$

La ricorsione – esercizio 3

Dato in input un array di n interi, trovare il minimo.



il minimo è pari al minore fra il valore di un elemento e il minimo trovato ricorsivamente sul resto dell'array

```
def MinArray (A, ult_ind=len(A)-1)
    if (ult_ind==0):
        return A[ult_ind]
    return min(A[ult_ind], MinArray(A, ult_ind-1))
```

Costo:

Se $n=1$: $T(n)=\Theta(1)$

Se $n>1$: $T(n)=\Theta(1)+T(n-1)$

La ricorsione – esercizio 4 (1)

Dato in input un array di n interi, verificare se è palindromo.

Esempio di array palindromo:

1	5	4	8	4	5	1
---	---	---	---	---	---	---

💡 un array è palindromo se i suoi estremi sono uguali e il resto dell'array è anch'esso palindromo.

Nota: se n è dispari c'è un elemento centrale, altrimenti no; questa differenza va gestita.

La ricorsione – esercizio 4 (2)

```
def Palin (A, i_min=0, i_max=len(A)-1)
    if (i_max <= i_min):                // Caso base
        return true
    if (A[i_min] != A[i_max]):
        return false
    return Palin(A, i_min+1, i_max-1)
```



dim dell'input $n = \text{len}(A) = i_{\text{max}} - i_{\text{min}} + 1$

Costo:

Se $n \leq 1$: $T(n) = \Theta(1)$

Se $n > 1$: $T(n) = \Theta(1) + T(n-2)$

La ricorsione – esercizio 5

Dato in input un array di n interi, stampare le chiavi dall'ultima alla prima, ossia nell'ordine: $A[n-1]$ $A[n-2]$ $A[n-3]$... $A[2]$ $A[1]$ $A[0]$

💡 stampiamo l'ultima chiave e chiamiamo ricorsivamente la funzione sul resto dell'array, gestendo opportunamente un secondo parametro.

```
def StampaArray (A, ind_ult: len(A)-1)
    stampa (A[ind_ult])
    if (ind_ult > 0)
        StampaArray (A, ind_ult-1)
```

Costo: Se $n=1$: $T(n)=\Theta(1)$

Se $n>1$: $T(n)=\Theta(1)+T(n-1)$

La ricorsione – esercizio 6

E se invece vogliamo stampare le chiavi dalla prima all'ultima?

💡 dobbiamo stampare durante la risalita dalle chiamate ricorsive, dopo aver incontrato il caso base per $n = 1$. Basta scambiare l'ordine della stampa rispetto alla chiamata ricorsiva:

```
def StampaArray_2 (A, ui: len(A)-1)
    if (ui > 0):
        StampaArray(A, ui-1)
    stampa (A[ui])
```

Costo: Se $n=1$: $T(n)=\Theta(1)$

Se $n>1$: $T(n)=\Theta(1)+T(n-1)$

OSSERVAZIONE:

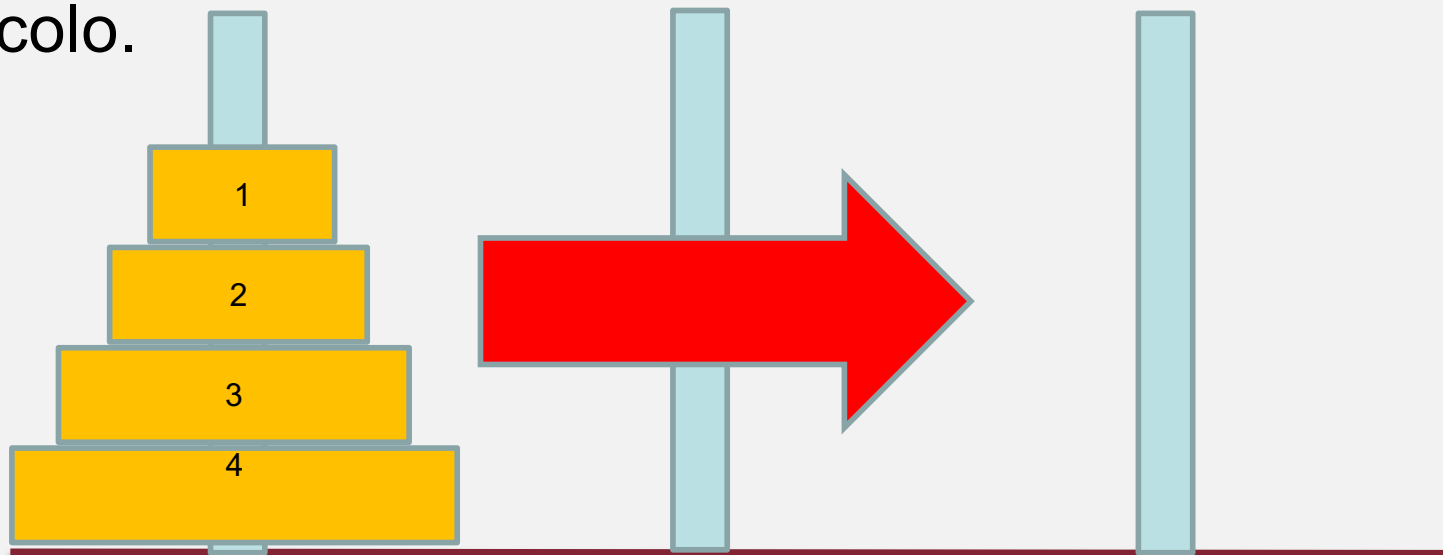
Non va bene stampare una chiave e poi richiamare sulla parte destra dell'array...

La ricorsione – esercizio 7 (1)

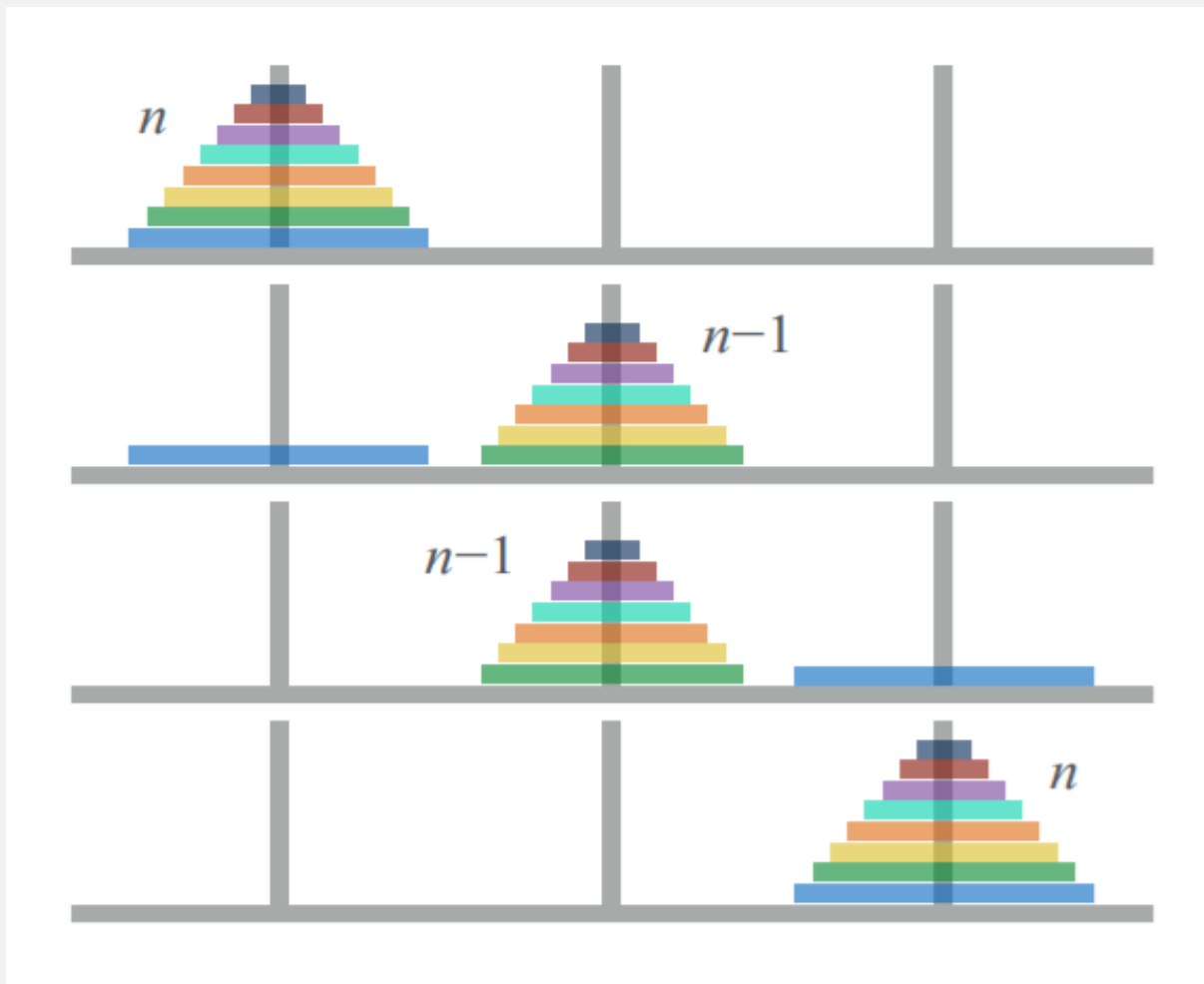
Torri di Hanoi

spostare una torre di dischi da un piolo ad un altro, con questi vincoli:

- In una mossa si può muovere un solo disco
- E' vietato spostare un disco più grande sopra un disco più piccolo.



La ricorsione – esercizio 7 (2)



[Disegno di Ozalp Babaoglu, Università di Bologna]

La ricorsione – esercizio 7 (3)

Strategia ricorsiva:

per spostare una torre di n dischi, dal piolo i al j :

1. Spostare gli $n-1$ dischi più piccoli da i a k ;
2. Spostare il disco n -esimo (il più grande) da i a j ;
3. Spostare gli $n-1$ dischi da k a j .

I passi 1. e 3. sono adatti all'uso della ricorsione, il passo 2 invece lo consideriamo un'operazione elementare.

La ricorsione – esercizio 7 (4)

```
def SpostaTorreH (n, i, j)
    if (n=0) return
    SpostaTorreH (n-1, i, 3-(i+j))
    SpostaDisco (i,j)
    SpostaTorreH (n-1, 3-(i+j), j)
```



1. I pioli sono numerati da 0 a 2;
2. I dischi sono numerati dal più piccolo (1) al più grande (n)
3. La funzione `SpostaTorreH` quindi agisce sui dischi dall'alto.

La ricorsione – esercizio 7 (5)

Costo computazionale:

💡 per spostare una torre di n dischi, bisogna spostare due torri di $n-1$ dischi, il che ci fa capire che il numero di mosse cresce esponenzialmente con n (simile ai numeri di Fibonacci).



Esercizi per casa

Esercizi per casa

- Progettare una funzione ricorsiva che, dati due interi k ed n , $0 \leq k \leq n$, calcoli il valore del coefficiente binomiale n su k utilizzando la seguente relazione:
$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \quad \text{con} \quad \binom{n}{0} = \binom{n}{n} = 1$$
- Progettare un algoritmo ricorsivo che, dati due interi x e y , $x > y > 0$, ne calcoli il massimo comun divisore utilizzando il seguente procedimento (di Euclide):
 - se $y=0$ allora $\text{MCD}(x,y)=x$
 - altrimenti $\text{MCD}(x,y)=\text{MCD}(y,x\%y)$dove $x\%y$ rappresenta il resto della divisione tra x ed y

Esercizio risolto

(guardare la sol. dopo aver provato a risolvere...)

- Dati due interi k ed n , $0 \leq k \leq n$, si calcoli il valore del coefficiente binomiale n su k utilizzando la seguente relazione: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ con $\binom{n}{0} = \binom{n}{n} = 1$

Sol. Basta usare la def. ricorsiva data:

```
def Choose_Ric(k,n)
if (k=0) or (k=n) return 1;
return Choose_Ric(k,n-1)+Choose_Ric(k-1,n-1);
```

Costo: $T(k,n) = \Theta(1) + T(k,n-1) + T(k-1,n-1)$

$T(0,n) = \Theta(1)$, $T(n,n) = \Theta(1)$