Ricorsione

Definizioni induttive

- Sono comuni in matematica
- Esempio: il fattoriale di un naturale N (N!)
  - se N=0 il fattoriale N! è 1
  - se N>0 il fattoriale N! è N * (N-1)!
- Esempio: numeri pari
  - **0 è** un numero pari
  - **se n** è un numero pari anche **n+2** è un numero pari

Dimostrazioni induttive

- Dimostriamo che $(2n)^2 = 4n^2$

  (distributività del quadrato rispetto alla moltiplicazione)

  1) se n=1 : vero (per verifica diretta)

  2) **suppongo** sia vero per n'=k (ip. di induz.) e lo **dimostro** per n=k+1:

  $(2n)^2 = (2(k+1))^2 = (2k+2)^2 = \mathbf{(2k)^2} + 8k + 4 =$

  (*per ipotesi di induzione*) $\mathbf{4k^2} + 8k + 4 =$

  $4(k^2 + 2k + 1) = 4(k+1)^2 = 4n^2$

1) è il caso base, 2) è il passo induttivo

Iterazione e ricorsione

- Sono i due concetti informatici che nascono dal concetto di induzione
- L'iterazione si realizza mediante la tecnica del ciclo
- Per il calcolo del fattoriale:
  - 0! = 1
  - n! = n (n - 1)(n - 2)…. 1 (realizzo un ciclo)

Fattoriale – versione iterativa

```
def fatt(n):
    f=1
    while n>0:
        f=f*n
        n=n-1
    return f

print(fatt(5))
```

Lo "spirito" del metodo ricorsivo

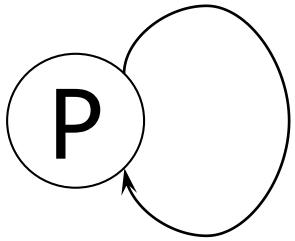
- Esiste un ***CASO BASE***, che rappresenta un sotto-problema facilmente risolvibile
  - Esempio: se **N=0**, so N! in modo "**immediato**" (vale 1)
- Esiste un ***PASSO INDUTTIVO*** che ci riconduce (prima o poi) al caso base
  - Consiste nell'esprimere la soluzione al problema (su dati di una "dimensione" generica) in termini di operazioni semplici e della soluzione allo **stesso** problema su dati "più piccoli" (che, per tali dati, **si suppone risolto *per ipotesi***)
    - Esempio: per **N *generico*** esprimo N! in termini di **N** (che è un dato direttamente accessibile) **moltiplicato per** (è una operazione semplice) il valore di **(N-1)!** (che so calcolare **per ipotesi induttiva**)

Fattoriale – versione ricorsiva

- 1) n! = 1 se n = 0
- 2) n! = n * (n - 1)! se n > 0
  - riduce il calcolo a un calcolo più semplice
  - ha senso perché si basa sempre sul fattoriale del numero più piccolo, che io conosco
  - ha senso perché si arriva a un punto in cui non è più necessario riusare la definizione 2) e invece si usa la 1)
  - 1) è il caso base, 2) è il passo induttivo

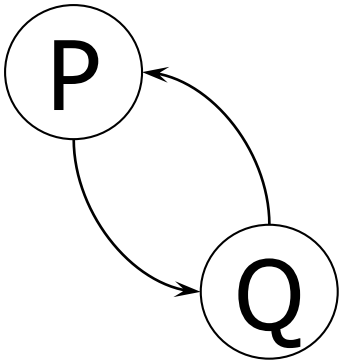
Ricorsione nei sottoprogrammi

- Dal latino re-currere
  - ricorrere, fare ripetutamente la stessa azione
- Un sottoprogramma P **invoca se stesso**
  - Direttamente
    - P invoca P



oppure

  - Indirettamente
    - P invoca Q che invoca P



Formulazione ricorsiva in C

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

print(fatt(5))
```

Simulazione del calcolo

Invocazione di: FattRic(3)

3 = 0? No $\Rightarrow$ calcola fattoriale di 2 e moltiplica per 3

2 = 0? No $\Rightarrow$ calcola fattoriale di 1 e moltiplica per 2

1 = 0? No $\Rightarrow$ calcola fattoriale di 0 e moltiplica per 1

0 = 0? Si $\Rightarrow$ fattoriale di 0 è 1

$\Rightarrow$ fattoriale di 1 è 1 per fattoriale di 0, cioè $1 \times 1 = 1$

$\Rightarrow$ fattoriale di 2 è 2 per fattoriale di 1, cioè $2 \times 1 = 2$

$\Rightarrow$ fattoriale di 3 è 3 per fattoriale di 2, cioè $3 \times 2 = 6$

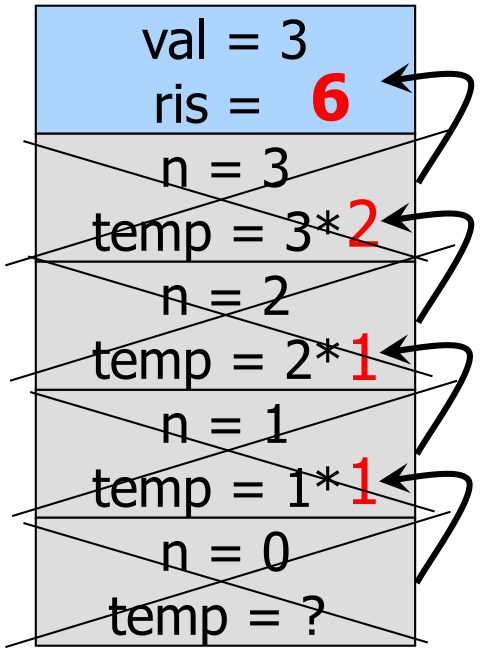
Esecuzione di funzioni ricorsive

- In un certo istante possono essere in corso ***diverse attivazioni*** dello **stesso** sottoprogramma
  - Ovviamente sono tutte sospese tranne una, l'ultima invocata, all'interno della quale si sta svolgendo il flusso di esecuzione
- Ogni attivazione esegue **lo stesso codice** ma opera su **copie distinte** dei parametri e delle variabili locali

Il modello a runtime: esempio

```
def fatt(n):
    if n==0:
        return 1
    else:
        temp=n*fatt(n-1)
        return temp


val=int(input("Inserire un numero: "))
ris=fatt(val)
print(ris)
```



temp: cella temporanea per memorizzare il risultato della funzione chiamata

assumiamo val = 3

Ma…

- … se ogni volta la funzione richiama se stessa… *perché la catena di invocazioni non continua* ***all'infinito***?
- Quando si può dire che una ricorsione è ben definita?
- Informalmente:
  - Se per ogni applicazione del passo induttivo ci si avvicina alla situazione riconosciuta come caso base, allora la definizione non è circolare e la catena di invocazioni termina

Un altro esempio: la serie di Fibonacci

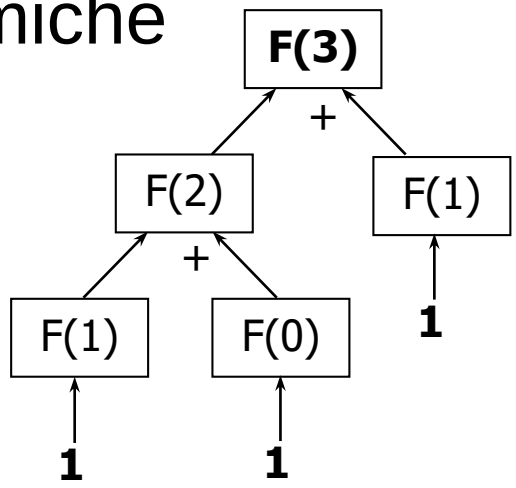
- Fibonacci (1202) partì dallo studio sullo sviluppo di una colonia di conigli in circostanze ideali

  - Partiamo da una coppia di conigli
  - I conigli possono riprodursi all'età di un mese
  - Supponiamo che dal secondo mese di vita in poi, ogni femmina produca una nuova coppia
  - e inoltre che i conigli non muoiano mai…

– Quante coppie ci sono dopo n mesi?

Definizione ricorsiva della serie

- I numeri di Fibonacci
  - Modello a base di molte dinamiche evolutive delle popolazioni



$F = \{f_0, \ldots, f_n\}$

- $f_0 = 1$
- $f_1 = 1$

} casi base (2!)

- Per $n > 1$, $f_n = f_{n-1} + f_{n-2}$

} 1 passo induttivo

Notazione "funzionale": $F(i) = f_i$

Numeri di Fibonacci in C

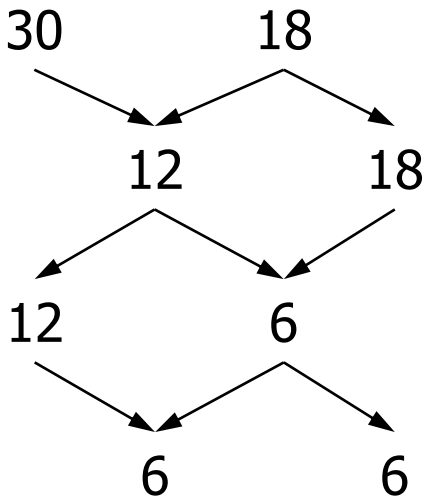
```
def Fibo(n):
    if n == 0 or n == 1:
        return 1
    else:
        return Fibo(n-1) + Fibo(n-2)
```

Ovviamente supponiamo che n>=0

Un altro esempio: MCD à-la-Euclide

- Il MCD tra M e N (M, N naturali positivi)
  - se M=N allora MCD è N *1 caso base*
  - se M>N allora esso è il MCD tra N e M-N
  - se N>M allora esso è il MCD tra M e N-M

2 passi induttivi



MCD – versione iterativa

```
def Euclide(m, n):
    while m != n:
        if m > n:
            m = m – n
        else:
            n = n – m
    return m
}
```

MCD – versione ricorsiva

```
def Euclide(m, n):
    if m == n:
        return n
    if m > n:
        return Euclide(m – n, n)
    else:
        return Euclide(m, n – m)
```

Funzione esponenziale (intera)

- Definizione iterativa:

  1) $x^y = 1$ se y = 0

  2) $x^y = x * x * \ldots x$ ***(y volte)*** se y > 0

- Definizione ricorsiva:

  1) $x^y = 1$ se y = 0

  2) $x^y = x * x^{(y-1)}$ se y > 0

Codice iterativo:

```
def esp(x, y):
    i=1
    e = 1
    while i <= y:
        e = e * x
        i=i+1
    return e
```

Codice ricorsivo:

```
def esp(x, y):
    if y == 0:
        return 1
    else:
        return x * esp(x, y-1)
```

Terminazione (ancora!)

- Attenzione al rischio di ***catene infinite*** di chiamate
- Occorre che le chiamate siano soggette a una condizione che prima o poi assicura che la catena termini
- Occorre anche che l'argomento sia "progressivamente ridotto" dal passo induttivo, in modo da tendere prima o poi al caso base

Le parole palindrome

- Una parola è un *palindromo* (dal greco: παλιν-, *ancora*, *indietro*, *di nuovo* e -δρομος, *corsa*, *percorso*) se la si può leggere indifferentemente da destra a sinistra e viceversa
- (ovviamente) tutte le parole di un solo carattere sono considerate palindrome
  - "a" -> sì
  - "db" -> no (anche se è graficamente simmetrica…)
  - "Anna" -> no (l'analisi è case sensitive)
  - "anno" -> no
  - "anilina" -> sì
  - "onorarono" -> sì
  - "saippuakivikauppias" -> sì (*venditore di liscivia*, in finlandese)

Esercizio

- Si scriva un programma che memorizza in un array di caratteri una parola letta da stdin e verifica se la parola è o non è palindroma
- **palindromo(parola)**
  - Restituisce True o False
- **Versione iterativa**: confronto tra tutte le coppie di lettere simmetriche rispetto al "centro"
  - (attenzione, la parola può avere un numero pari o dispari di caratteri)
- **Versione ricorsiva**: un palindromo è tale se...

Palindromi in versione ricorsiva

Un palindromo è tale se:

- la parola è di lunghezza 0 o 1; **Caso base**

  oppure

- il primo e l'ultimo carattere della parola sono uguali **e inoltre** la sotto-parola che si ottiene ignorando i caratteri estremi è a sua volta un palindromo

Passo induttivo

Il passo induttivo riduce la ***dimensione*** *del problema!*

Palindromi in versione ricorsiva

```
def palindromo(s):
  return palindromoAux(s,0,len(s)-1)

def palindromoAux(s,inizio,fine):
  if fine<=inizio:
    return True
  else:
    return s[inizio]==s[fine] and palindromoAux(s,inizio+1,fine-1)

print(palindromo("anilina"))
print(palindromo("anna"))
print(palindromo("Ciao"))
print(palindromo(""))
print(palindromo("Anna"))
```

Palindromi ricorsivi: variante

```
def palindromo(s):
  return palindromoAux(s,0,len(s)-1)

def palindromoAux(s,inizio,fine):
  if fine<=inizio:
    return True
  else:
    return palindromoAux(s,inizio+1,fine-1) and s[inizio]==s[fine]

print(palindromo("anilina"))
print(palindromo("anna"))
print(palindromo("Ciao"))
print(palindromo(""))
print(palindromo("Anna"))
```

Qual è la differenza?

Altri tipi di palindromi (1)

- Palindromi **a parola**:

  "Fall leaves after leaves fall"

- Palindromi **a frase** (ignorando spazi e punteggiatura):

  "I topi non avevano nipoti"

  "Avida di vita, desiai ogni amore vero, ma ingoiai sedativi, da diva"

  "Sun at noon, tan us!"

  – Molti esempi notevoli:
    - G. Perec, "9691" (> 5000 caratteri)
      – "Trace l'inégal palindrome. Neige [...] ne mord ni la plage ni l'écart."
    - G. Varaldo, "11 Luglio 1982" (> 4000 caratteri)
      – Ai lati, a esordir, dama [...] a Madrid, rosea Italia!
    - Batman, "Una storia italiana" (> 1000 caratteri)
      – O idolo, se vero, mal onori parole [...] rapirono l'amore, v'è sol odio.

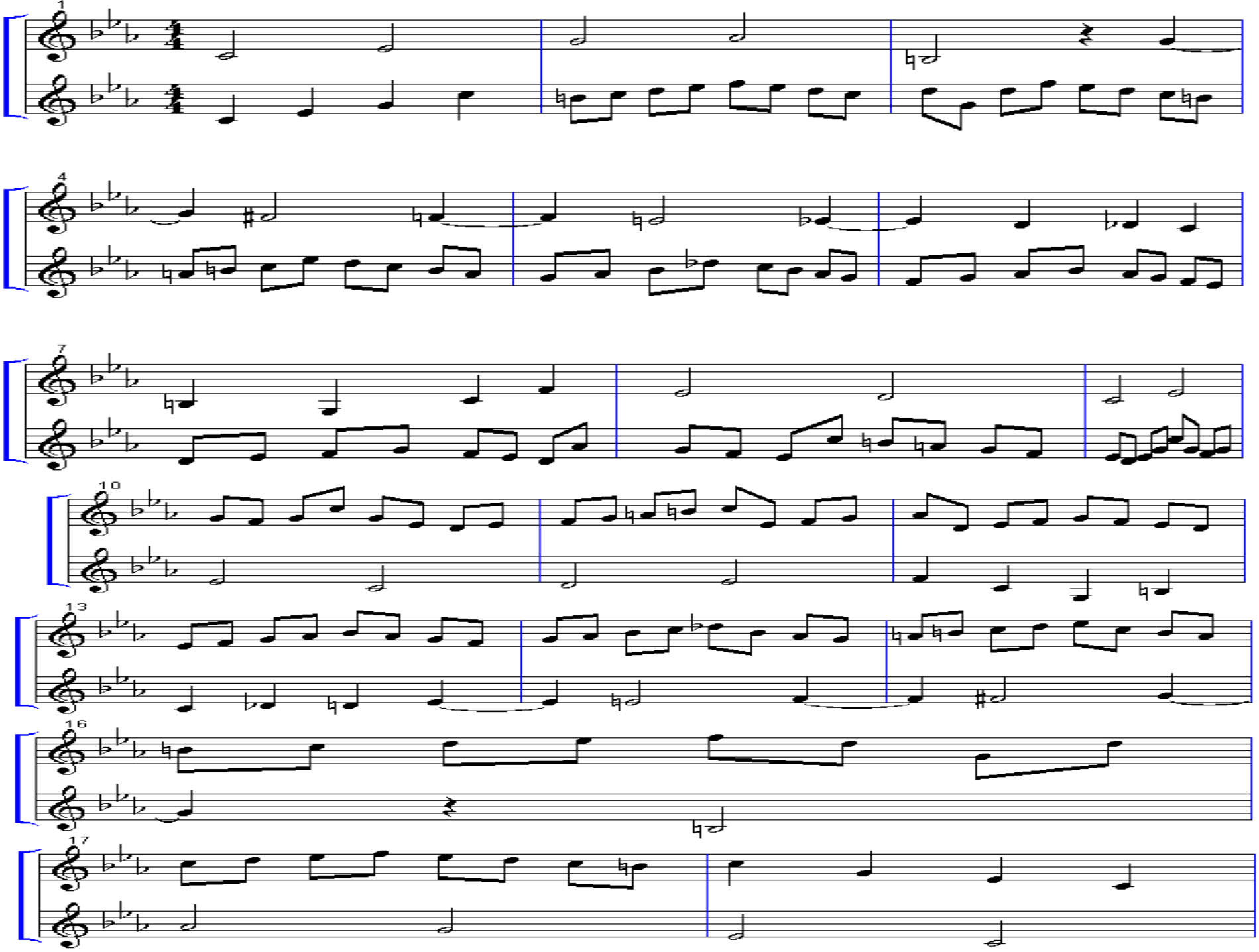
Altri tipi di palindromi (2)

- Palindromi **a riga**
  - J. A. Lindon, "Doppelganger"

    "Entering the lonely house with my wife,
    I saw him for the first time
    peering furtively from behind a bush
    […]
    Peering furtively from behind a bush,
    I saw him, for the first time
    entering the lonely house with my wife."
- Esercizio: implementare funzioni di verifica palindromi a riga, a frase, a parola

Digressione: palindromi ovunque

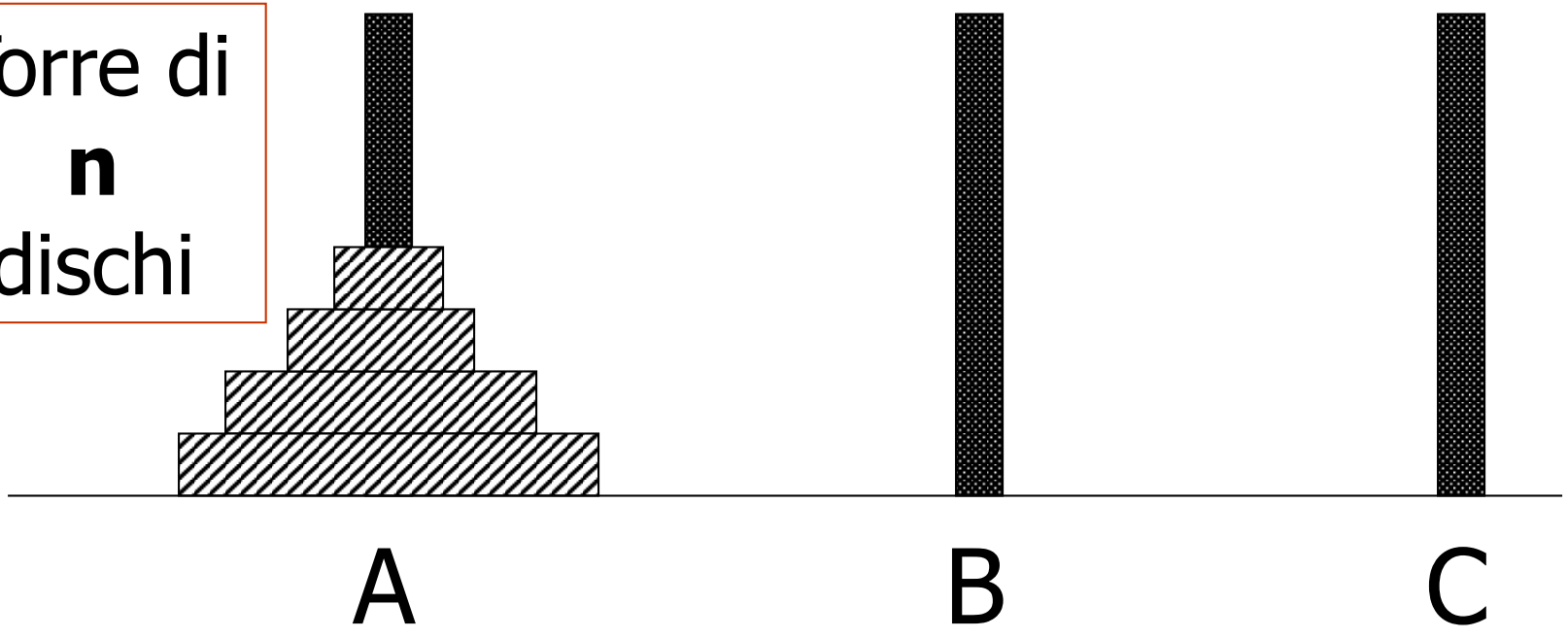
- I palindromi esistono anche in matematica (numeri palindromi)
- …in pittura…
- …in musica…
  - J. S. Bach ha scritto un "canone cancrizzante" a due voci
    - Scambiando la prima e la seconda voce, e leggendo la partitura da sinistra a destra, si ottiene ancora lo stesso brano



Le torri di Hanoi

Spostare tutta la torre da A a C **spostando un cerchio alla volta** e **senza mai mettere un cerchio più grosso su uno più piccolo**

Torre di **n** dischi

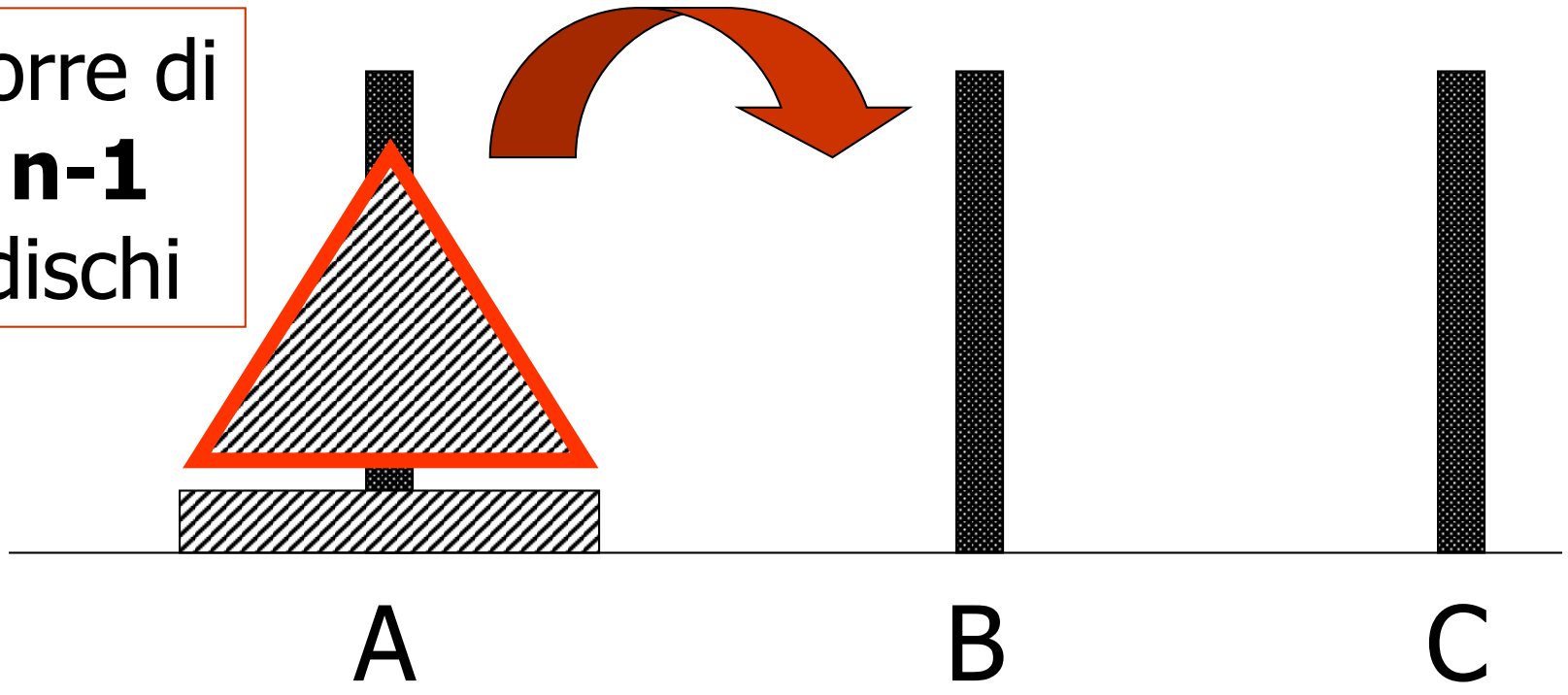


FORMULAZIONE RICORSIVA?

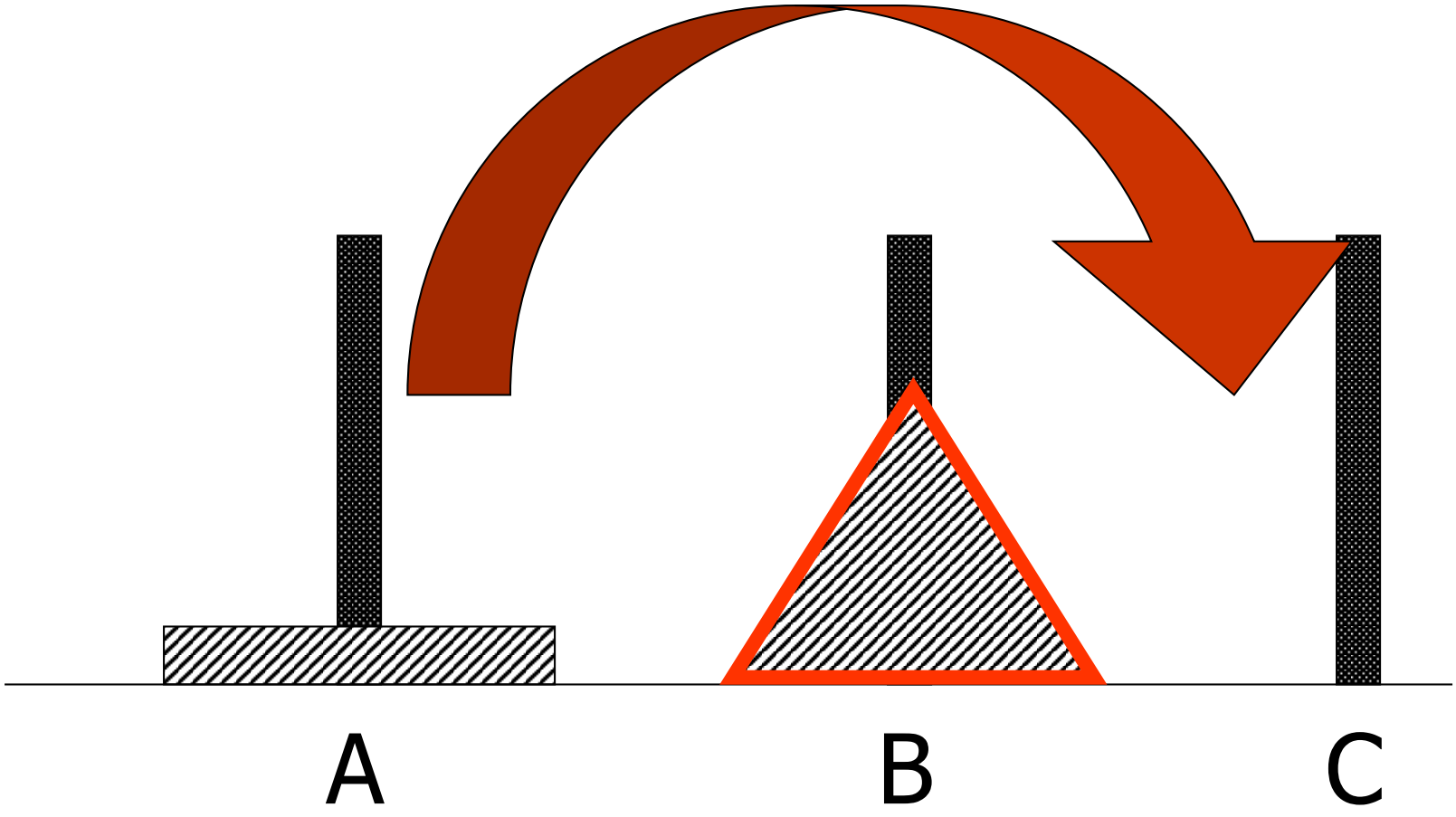
Le torri di Hanoi

FORMULAZIONE RICORSIVA

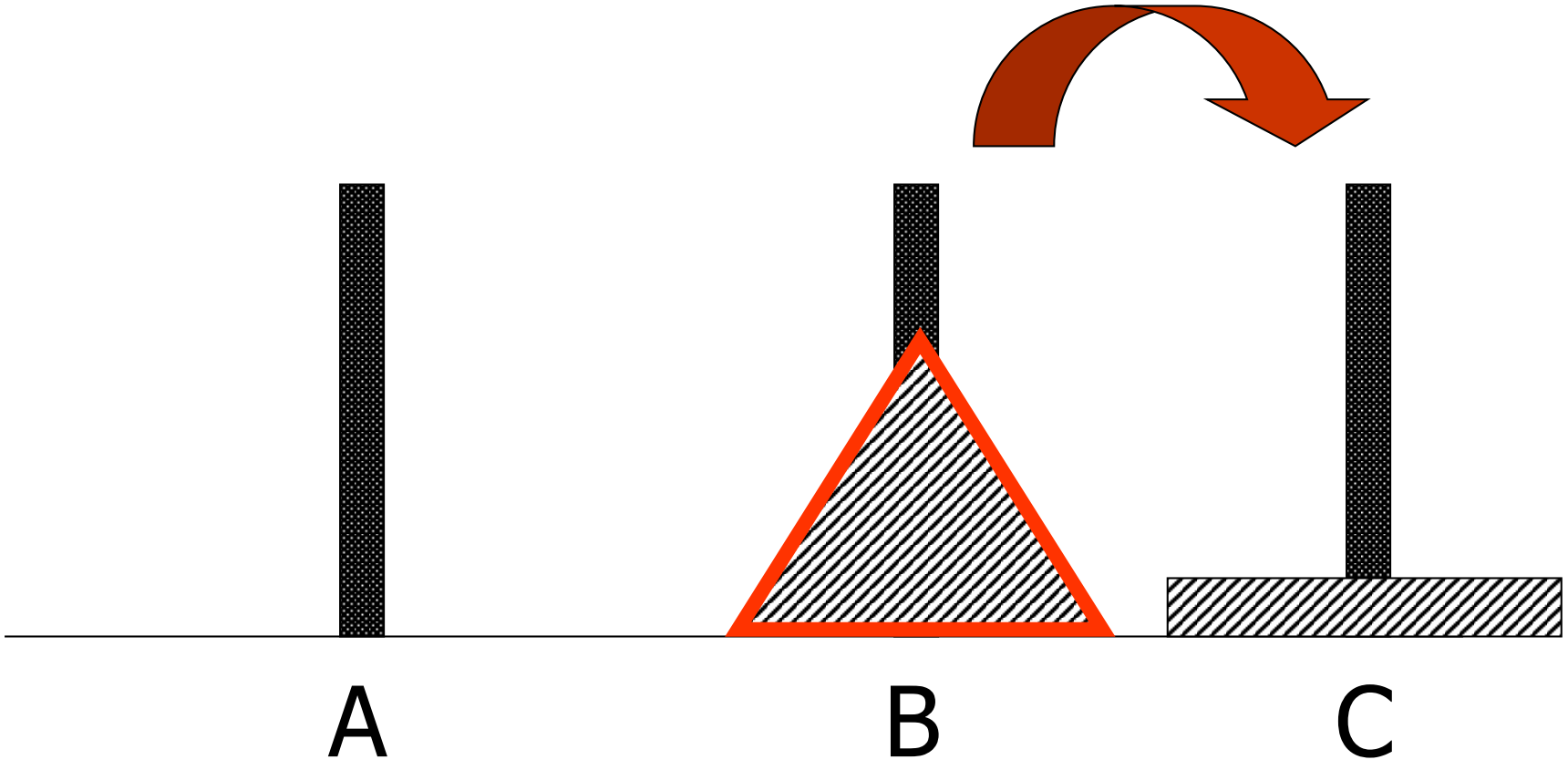
Torre di **n-1** dischi



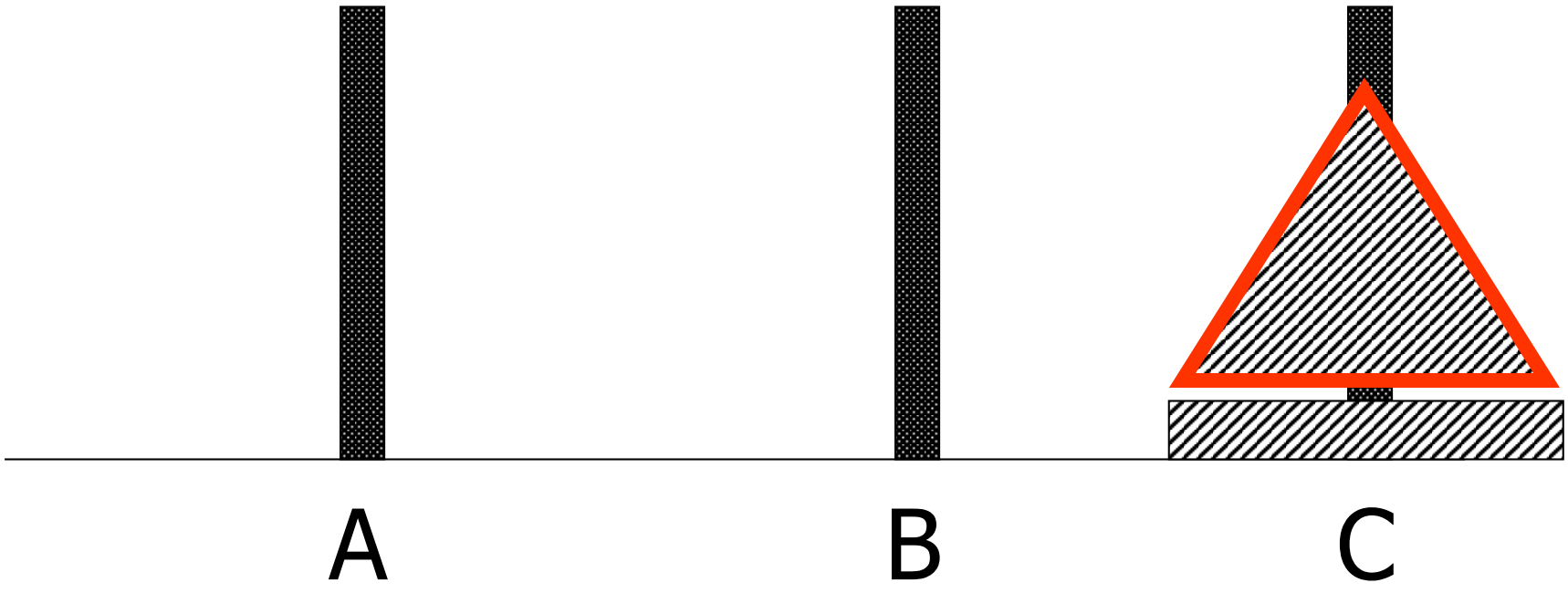
Le torri di Hanoi



Le torri di Hanoi



Le torri di Hanoi



Il metodo

- Il programma deve stampare una serie di comandi di spostamento
  - So spostare la torre di 1 elem. da A a C

    (caso di base)
  - Per spostare la torre di N elem. da A a C
    - sposto la torre di N-1 cerchi da A a B
    - sposto il cerchio restante in C
    - sposto la torre di N-1 elementi da B a C

Chiamata iniziale:

hanoi(12,"A","C","B")

Significa che abbiamo una torre di 12 cerchi da trasferire da A a C potendo usare B

```
def hanoi (n, a, c, b):
   if n != 0:
      hanoi (n-1, a, b, c)
      print("sposta cerchio da " + a + " " + c)
      hanoi (n-1, b, c, a)
```

Ricorsione o iterazione?

- Spesso le soluzioni ricorsive sono eleganti
- Sono vicine alla *definizione* del problema
- Però possono essere inefficienti
- Chiamare un sottoprogramma significa allocare memoria a run-time

N.B. è **sempre** possibile trovare un corrispondente iterativo di un programma ricorsivo

Numeri di Fibonacci e memorizzazione

- La prima volta che calcolo un dato numero di Fibonacci lo memorizzo in una lista
- Dalla seconda volta in poi, anziché ricalcolarlo, lo leggo direttamente dalla lista

Drastica riduzione della *complessità* (aumento di efficienza)
Questa soluzione richiede un tempo *lineare* in n
La soluzione precedente richiede un tempo *esponenziale* in n

Calcolo con memorizzazione

```
def fastFib(n, memo):
    if len(memo)<n:
        memo.append(fastFib(n-1, memo) + fastFib(n-2, memo))
    return memo[n-1]

def fibo(n):
    memo = [1,1]
    return fastFib(n, memo)

res = fibo(6)
print (res)
```

Esercizi Ricorsione

Esercizio

- Scrivere un programma che, dato un numero calcola la somma dei primi N numeri pari positivi in maniera ricorsiva.
- Specifica Liv 1: La somma dei primi N numeri pari è data dalla seguente,

  $S_N = 2*1 + 2*2 + 2*3 + \ldots + 2*i + \ldots + 2*(N-1) + 2*N.$
- Specifica Liv 2:
  - se N =1, $S_N = 2$, **(CASO BASE)**
  - se N >1, $S_N = 2 * N + S_{N-1}$ **(PASSO INDUTTIVO)**

    (somma dell'N-esimo numero pari + la sommatoria dei primi N-1 numeri pari.)

```
def sommaPari(n):
    if n==1:
        return 2
    else:
        return 2*n+sommaPari(n-1)

print(sommaPari(1))
print(sommaPari(2))
print(sommaPari(5))
print(sommaPari(10))
```

Esercizio

- Calcolo del massimo di una lista con procedimento ricorsivo.
- Si pensi di assegnare il primo elemento come massimo e confrontarlo con tutti gli altri cambiando il valore del massimo se questo è minore della cella corrente della lista.

```
def max(lis):
    return maxaux(lis,0)

def maxaux(lis,posizione):
    if posizione==len(lis)-1:
        return lis[posizione]
    maxdeglialtri=maxaux(lis,posizione+1)
    if lis[posizione]>maxdeglialtri:
        return lis[posizione]
    else:
        return maxdeglialtri

print(max([78,56,345,78,98]))
print(max([778,56,345,78,98]))
print(max([78,56,345,78,998]))
```

Esercizio

- Scrivere un programma che stampi sullo standard output tutti i valori del triangolo di Tartaglia per un certo ordine N, utilizzando una funzione ricorsiva

1	n = 0
1 1	n = 1
1 2 1	n = 2
1 3 3 1	n = 3
1 4 6 4 1	n = 4
1 5 10 10 5 1	n = 5
1 6 15 20 15 6 1	n = 6
.....	

Leggendo la figura del triangolo di Tartaglia riga per riga, è possibile dedurre come il calcolo di ognuna di esse sia funzione della riga precedente. Il calcolo dei coefficienti binomiali segue dunque le seguenti regole:
con k == 0 e k == n, cobin(n,k) = 1. **(caso base)**
ogni coefficiente è la somma del suo "soprastante" e del predecessore di quest'ultimo. **(passo induttivo)**

```
def cobin(n,k):
    if k==0 or n==k:
        return 1
    else:
        return cobin(n-1,k-1)+cobin(n-1,k)

print(cobin(2,0))
print(cobin(2,1))
print(cobin(2,2))
print(cobin(3,2))
print(cobin(4,0))
print(cobin(4,1))
print(cobin(4,2))
print(cobin(4,3))
print(cobin(4,4))
print(cobin(5,2))
print(cobin(1,1))
print(cobin(0,0))
```

Esercizio

- Scrivere un programma che stampi a video tutte le possibili N! permutazioni degli elementi di una lista di N interi.
- Il problema proposto si presta in modo naturale ad una formulazione ricorsiva, infatti:
- La lista è lunga “len” e inizialmente dobbiamo costruire le permutazioni di “n = len” elementi:
  - Per generare tutte le possibili permutazioni di n elementi, si può pensare di tenere fisso l’ elemento in prima posizione e stampare l’intera sequenza per ognuna delle permutazioni dei restanti n-1 elementi.
  - Scambiare il primo degli n elementi da permutare con il secondo
  - ripetere considerando n-1 elementi (tranne il primo)
  - scambiare nuovamente il primo degli n elementi con il secondo
  - Scambiare il primo degli n elementi elemento con il terzo
  - ripetere considerando n-1 elementi
  - scambiare nuovamente il primo degli n elementi con il secondo