Corso di laurea in Informatica
Introduzione agli Algoritmi

Algoritmi ricorsivi

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

FUNZIONI RICORSIVE

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

$$0! = 1$$
$$n! = n(n-1)! \text{ se n>0}$$

- la prima riga della definizione determina il **caso base**, in questo caso indica che il fattoriale di 0 è 1.

- la seconda riga determina il meccanismo di calcolo ricorsivo, ossia stabilisce come, conoscendo il valore della funzione per un certo numero intero si calcola il valore per l'intero successivo.

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.

- la soluzione del problema complessivo è costruita risolvendo (ricorsivamente) uno o più sottoproblemi di dimensione minore e successivamente producendo la soluzione complessiva mediante combinazione delle soluzioni dei sottoproblemi;

- 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***), incontrato il quale la ricorsione termina.

Algoritmi per il calcolo del fattoriale

```
def fattoriale(n):
    f = 1
    for i in range(1, n+1):
        f = f * i
    return f
```

$$T(n) = \Theta(n)$$

NOTA: Complessità di spazio $\Theta(1)$

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

$$T(n) =$$

$$\Theta(1)$$

$$T(n-1)$$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{altrimenti} \end{cases}$$

Algoritmi ricorsivi

• E' fondamentale assicurarsi che **le funzioni ricorsive abbiano almeno un caso base**, altrimenti la loro esecuzione genererebbe una catena illimitata di chiamate ricorsive che non terminerebbe mai (provocando ben presto la terminazione forzata dell'elaborazione a causa dell'esaurimento delle risorse di calcolo, per ragioni che vedremo fra breve).

• Bisogna quindi essere assolutamente certi che il caso base non possa essere mai "saltato" durante l'esecuzione.

• Ogni funzione, sia essa ricorsiva o no, richiede per la sua esecuzione una certa quantità di memoria, per:

- caricare in memoria il codice della funzione;
- passare i parametri alla funzione;
- memorizzare i valori delle variabili locali della funzione.

- Quindi le funzioni ricorsive in generale hanno maggiori esigenze, in termini di memoria, delle funzioni non ricorsive (dette anche funzioni iterative).

Per inciso, questa è la ragione per la quale una funzione ricorsiva mal progettata, ad es. nella quale la condizione di terminazione non si incontra mai, esaurisce con estrema rapidità la memoria disponibile con la conseguente inevitabile terminazione forzata dell'esecuzione.

Va detto che qualsiasi problema risolvibile con un algoritmo ricorsivo può essere risolto anche con un algoritmo iterativo.

In quali situazioni è consigliabile utilizzare un algoritmo ricorsivo anziché uno iterativo?

Queste considerazioni possono essere d'aiuto nella decisione:

- è conveniente utilizzare la ricorsione quando essa permette di formulare la soluzione del problema in un modo che è chiaro ed aderente alla natura del problema stesso, mentre la soluzione iterativa è molto più complicata o addirittura non evidente;

- non è conveniente utilizzare la ricorsione quando esiste una soluzione iterativa altrettanto semplice e chiara;

- non è conveniente utilizzare la ricorsione quando l'efficienza è un requisito primario.

ESEMPIO: Algoritmo ricorsivo per la ricerca sequenziale in lista di n elementi

- se la lista è vuota restituiamo None;
- ispezioniamo il primo elemento della lista;
- se non è l'elemento cercato risolviamo il problema ricorsivamente sugli altri $n-1$ elementi.

```
def RicercaSeqR(v, A):
    if A==[]:
        return False
    if A[0]==v:
        return True
    return RicercaSeqR(v, A[1:])
```

$T(n) =$

$\Theta(n)$

$T(n-1)$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(n) & \text{altrimenti} \end{cases}$$

NOTA: Complessità di spazio $\Theta(n^2)$

ESEMPIO: Algoritmo ricorsivo per la ricerca sequenziale in lista di n elementi

- se la lista è vuota restituiamo None;
- ispezioniamo il primo elemento della lista;
- se non è l'elemento cercato risolviamo il problema ricorsivamente sugli altri $n-1$ elementi.

```
def RicercaSeqR(v, A, i=0):
    if i==len(A):
        return False
    if A[i]==v:
        return True
    return RicercaSeqR(v, A, i+1)
```

$T(n) =$

$\Theta(1)$

$T(n-1)$

Chiamata iniziale: >>> Ricerca_sequenziale_ricorsiva(v, A)

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{altrimenti} \end{cases}$$

NOTA: Complessità di spazio $\Theta(n)$

Esempio: algoritmo ricorsivo per la ricerca binaria in lista ordinata di *n* elementi

```
def RicercaBinR(v, A, inizio, fine):
    if inizio > fine:
        return  False
    m = (inizio + fine)//2
    if A[m] == v:
        return True
    if A[m] < v:
        return RicercaBinR(v, A,  m + 1, fine)
    return RicercaBinR(v, A, inizio, m - 1)
```

$T(n) =$

$\Theta(1)$

$T\left(\frac{n}{2}\right)$

$T\left(\frac{n}{2}\right)$

Chiamata iniziale:

```
>>> RicercaBinR(v, A, 0, len(A)-1)
```

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(\frac{n}{2}) + \Theta(1) & \text{altrimenti} \end{cases}$$

ESEMPIO: Algoritmo ricorsivo per la ricerca binaria in lista di n elementi non ordinata:

```
def Ricerca (v,A, inizio, fine):
    if inizio > fine:
        return  False
    m = (inizio + fine)//2
    if A[m] == v:
        return True
    if Ricerca(v, A,  inizio, m-1):
        return True
    return Ricerca(v, A,  m + 1, fine)
```

$T(n) =$

$\Theta(1)$

$T\left(\frac{n}{2}\right)$

$T\left(\frac{n}{2}\right)$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ 2 \cdot T(\frac{n}{2}) + \Theta(1) & \text{altrimenti} \end{cases}$$

Esempio: calcolo dell'n-esimo numero di Fibonacci

La definizione dei numeri di Fibonacci è:

Leonardo Fibonacci
(1170-1250)

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

dalla formula otteniamo: 0, 1, 1, 2, 3, 5, 8, 13, 21 ……..

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

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

Segue esempio: calcolo dell'n-esimo numero di Fibonacci.

Viene naturale impostare il calcolo di questi numeri mediante la seguente funzione ricorsiva:

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

Segue esempio: calcolo dell'n-esimo numero di Fibonacci.

Qual è il costo computazionale della versione ricorsiva?

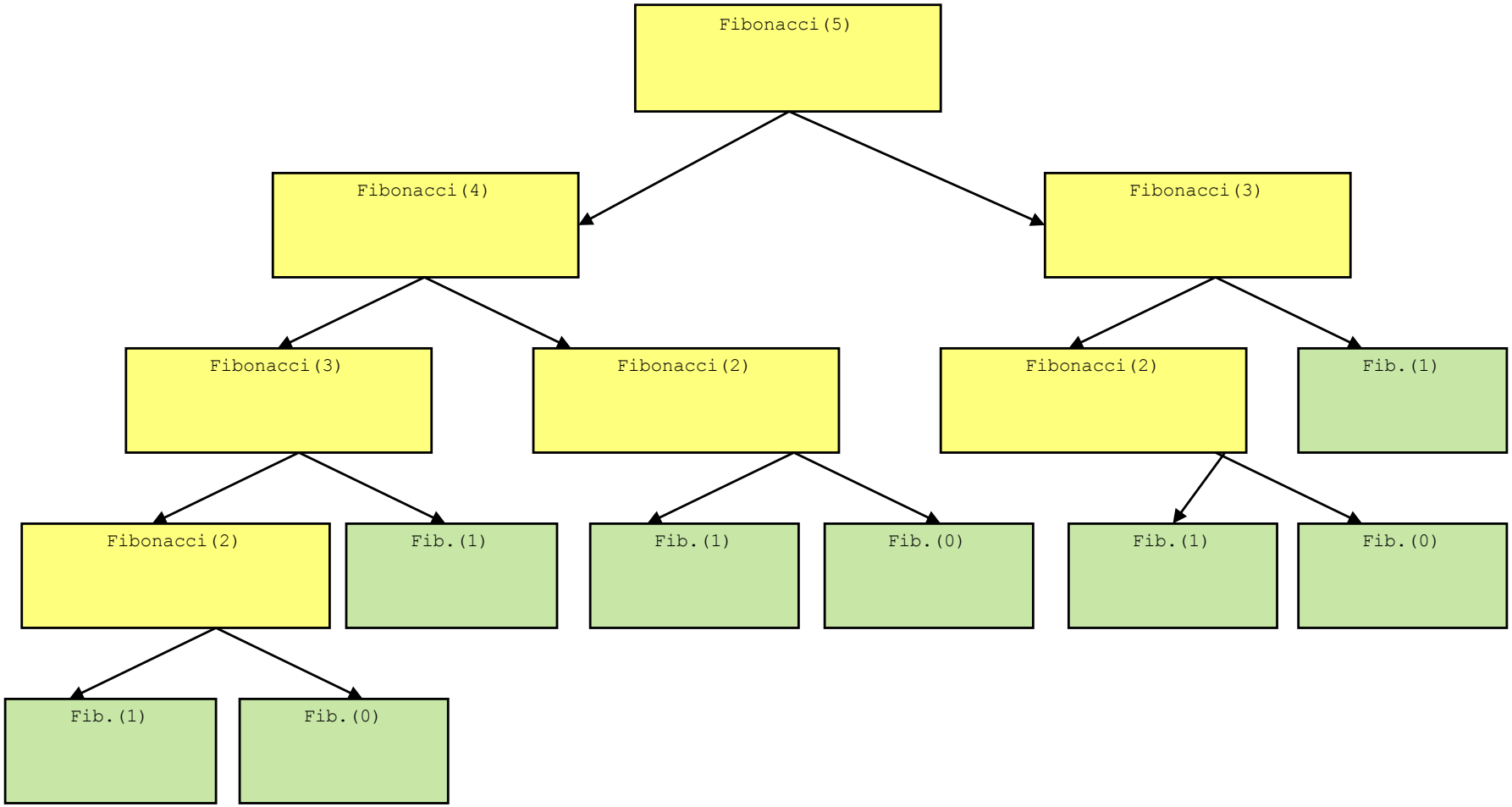
```
def  FibR(n):
  if n<=1:                <-   Θ(1)
      return n            <-   Θ(1)
  return FibR(n-1)+ FibR(n-2)   <-   T(n − 1) + T(n − 2) + Θ(1)
```

Il costo della funzione è dato dalla seguente equazione di ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & \text{altrimenti} \end{cases}$$

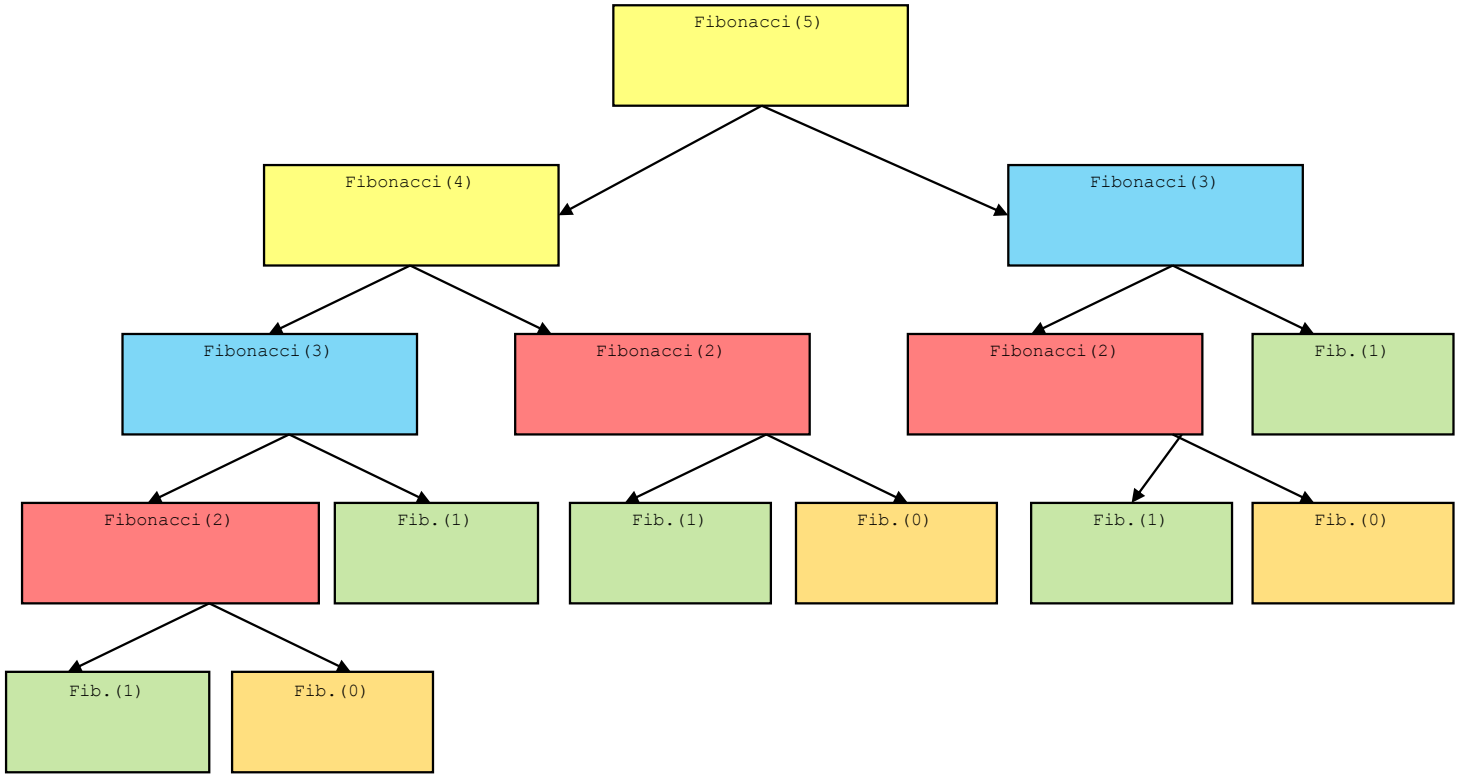
che è simile alla formula che definisce i numeri di Fibonacci, numeri che crescono molto velocemente. Vedremo nella prossima lezione che risolta quest'equazione da appunto un costo esponenziale.

Segue esempio: calcolo dell'n-esimo numero di Fibonacci.



Segue esempio: calcolo dell'n-esimo numero di Fibonacci.

Osservazione 1: Uno stesso calcolo viene ripetuto più e più volte !!!



Segue esempio: calcolo dell'n-esimo numero di Fibonacci.

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

- Il numero totale delle chiamate effettuate dall'inizio alla fine della elaborazione cresce molto velocemente con *n*.
- Inoltre, 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 in questione.

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

Segue esempio: calcolo dell'n-esimo numero di Fibonacci.

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

```
def  FibI(n):
    a,b=0,1
    for i in range(2,n+1):
        a, b = b, a+b
    return b
```

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

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

from time import time

def TempoFibR(n):
    a=time()
    FibR(n)
    return time()-a


>>> TempoFibR(20)
0.0042459964752197266
>>> TempoFibR(30)
0.5020561218261719
>>> TempoFibR(32)
1.225944995880127
>>> TempoFibR(35)
5.374284029006958
>>> TempoFibR(37)
14.031291723251343
>>> TempoFibR(40)
59.66544198989868
>>> TempoFibR(50)
?
```

```python
def  FibI(n):
    a,b=0,1
    for i in range(2,n+1):
        a, b = b, a+b
    return b

from time import time

def TempoFibI(n):
    a=time()
    FibI(n)
    return time()-a

>>> TempoFibI(50)
1.2636184692382812e-05
>>> TempoFibI(500)
7.796287536621094e-05
>>> TempoFibI(5000)
0.0007832050323486328
```

esercizio:

Data una lista di *n* interi, verificare se la lista è palindroma.

Esempio di lista palindroma:

1	5	4	8	4	5	1

💡: una lista è palindroma se i suoi estremi sono uguali e il resto del vettore è anch'esso palindromo.

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

```python
def palindroma(A):
    if len(A)<=1: return True
    if A[0]!=A[-1]: return False
    return palindroma(A[1:-1])
```

<—-Caso base $\Theta(1)$

$\Theta(1)$

$T(n-2)+\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n-2)+\Theta(n) & \text{altrimenti} \end{cases}$$

NOTA: Complessità di spazio $\Theta(n^2)$

seguito:

```
def Palindroma (A, inizio, fine):
    if inizio >= fine:
        return True
    if A[inizio] != A[fine]:
        return False
    return Palindroma(A, inizio+1, fine-1)
```

<-Caso base $\Theta(1)$

$\Theta(1)$

$T(n-2)+\Theta(1)$

la funzione viene invocata da

```
>>> Palindroma(A, 0, len(A)-1)
```

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n-2)+\Theta(1) & \text{altrimenti} \end{cases}$$

NOTA: Complessità di spazio $\Theta(n)$

esercizio:

Data una lista di *n* interi calcolare la somma dei suoi elementi.

💡: la somma della lista vuota è zero mentre la somma di una lista non vuota è pari al valore dell'ultimo elemento più la somma calcolata ricorsivamente sul resto della lista

```
def SommaLista(A, i = 0):
    if i == len(A):
        return 0
    return A[i] + SommaLista(A, i+1)
```

La funzione viene invocata con:

```
>>> SommaLista(A)
```

esercizio:

Data una lista A di interi non vuota trovare il minimo.

💡: se la lista contiene un unico elemento il minimo è quell'elemento altrimenti il minimo è pari al minore fra il valore dell'ultimo elemento e il minimo trovato ricorsivamente sul resto della lista

```
def MinimoLista(A, i=0):
    if i == len(A) - 1:
        return A[i]
    return min(A[i], MinimoLista(A, i+1))
```

La funzione viene invocata con:

```
>>> MinimoLista(A)
```

esercizio:

Data una lista di *n* interi, stampare le chiavi dall'ultima alla prima, ossia nell'ordine:

```
lista[n-1], lista[n-2], lista[n-3]    .....
```

💡: stampiamo l'ultima chiave e chiamiamo ricorsivamente la funzione sul resto della lista, gestendo opportunamente il secondo parametro.

```
def  StampaLista(lista, i):
        if i >= 0:
            print(lista[i])
            StampaLista(lista, i-1)
```

La funzione viene invocata con

```
>>> StampaLista(A, len(A)-1)
```

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

💡: dobbiamo fare in modo che le stampe avvengano solo durante la risalita dalle chiamate ricorsive, dopo aver incontrato il caso base per $n = 0$.

Basta scambiare l'ordine della stampa rispetto alla chiamata ricorsiva:

```
def StampaLista(lista, i):
    if i >= 0:
        StampaLista(lista, i-1)
        print(lista[i])
```

ESEMPIO: Algoritmo ricorsivo che, dati n ed m, calcola la potenza n^m

Sfruttiamo la definizione ricorsiva: $$n^m = \begin{cases} 1 & \text{se } m = 0 \\ n & \text{se } m = 1 \\ n \cdot n^{m-1} & \text{altrimenti} \end{cases}$$

```python
def Potenza(n,m):
    if m==0:return 1
    if m==1: return n
    return n*Potenza(n,m-1)
```

$T(n,m) =$

$\Theta(1)$

$\Theta(1)$

$T(n, m-1)$

$$T(n,m) = \begin{cases} \Theta(1) & \text{se } m = 1 \\ T(n, m-1) + \Theta(1) & \text{altrimenti} \end{cases}$$

Corso di laurea in Informatica
Introduzione agli algoritmi

Esercizi

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO 1

Progettare una funzione ricorsiva che, dati due interi k ed n, $0 \leq k \leq n$, calcoli il valore del coefficiente binomiale $\binom{n}{k}$ utilizzando la seguente definizione ricorsiva:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \text{ con } \binom{n}{0} = \binom{n}{n} = 1$$

ESERCIZIO 2

Progettare un algoritmo ricorsivo che, dati due interi x e y *con* $x > y > 0$, ne calcoli il *massimo comun divisore* utilizzando il seguente procedimento (di Euclide):

$$MCD(x, y) = \begin{cases} x & \text{se } y = 0 \\ MCD(y, x \,\%\, y) & \text{altrimenti} \end{cases}$$

dove $x \,\%\, y$ *rappresenta il resto della divisione tra* x *ed* y