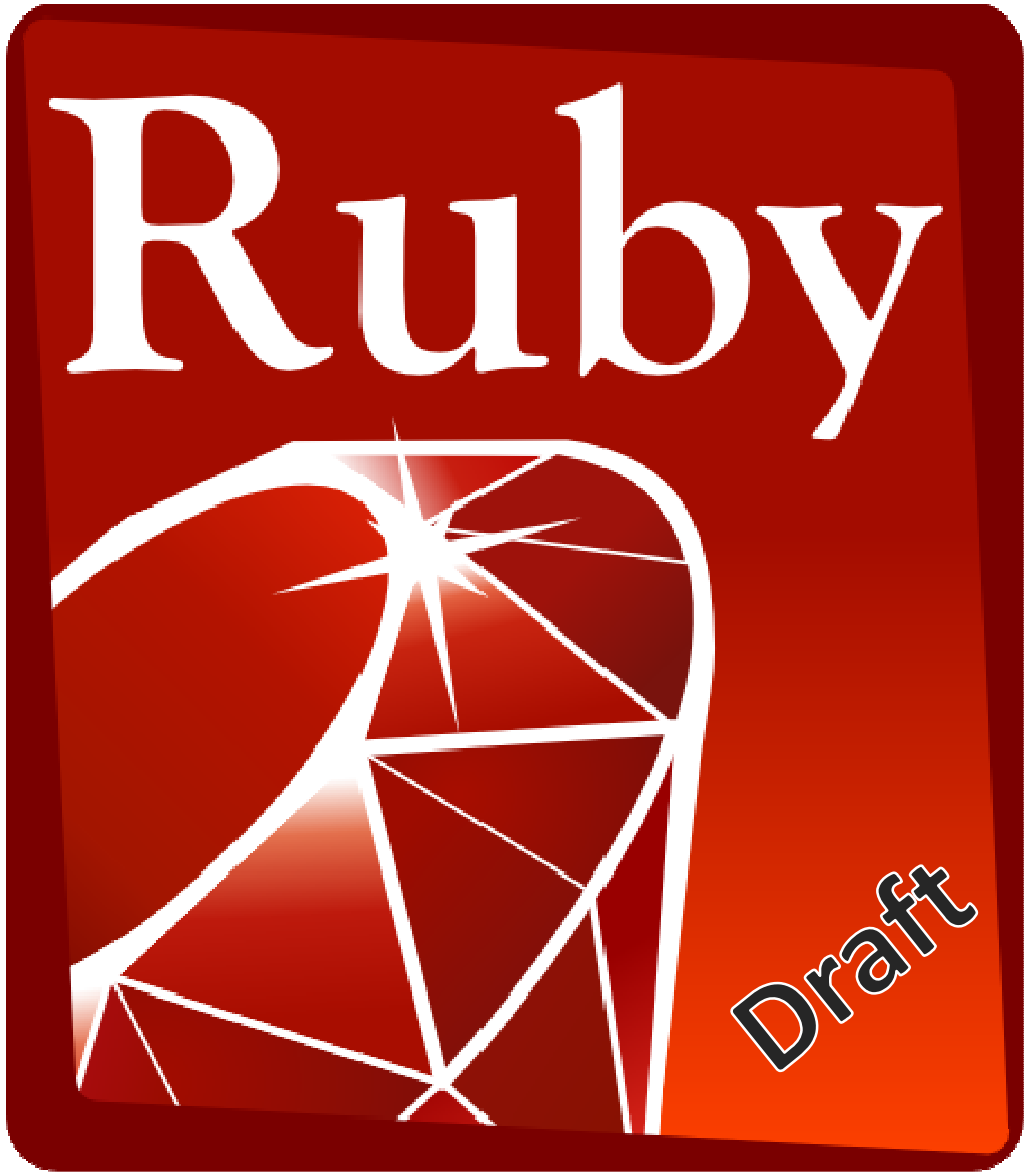
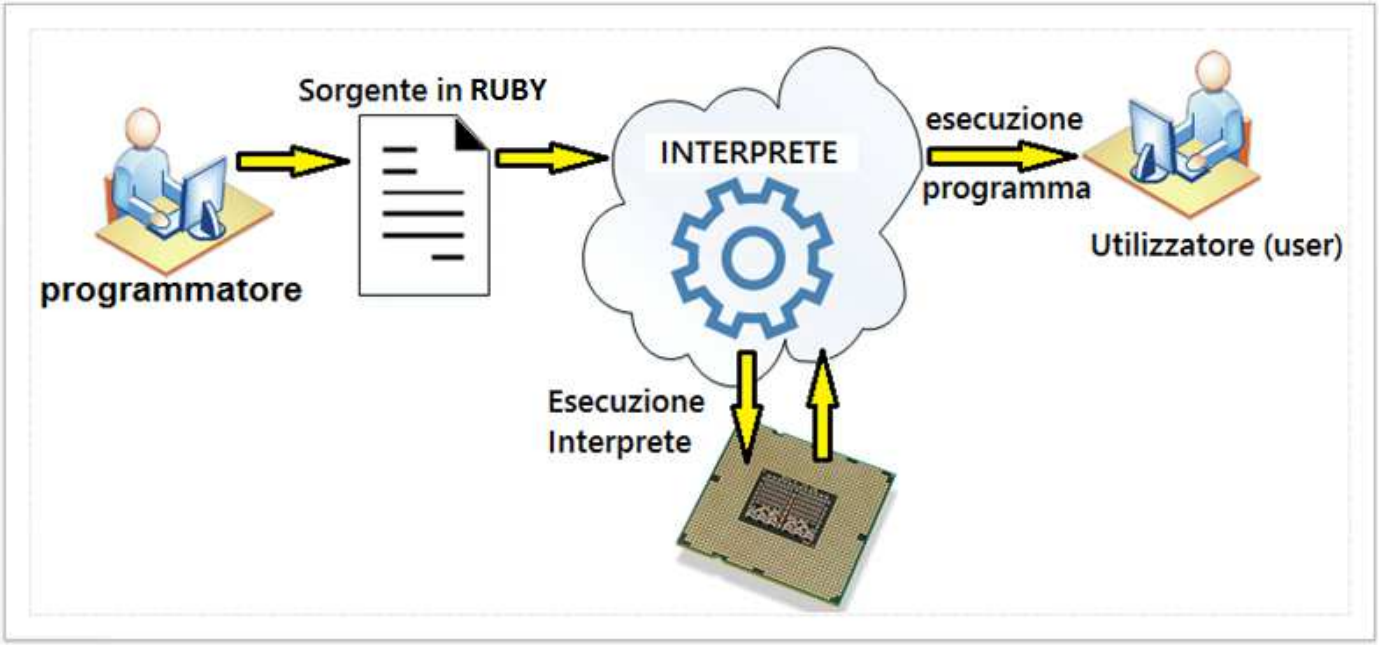


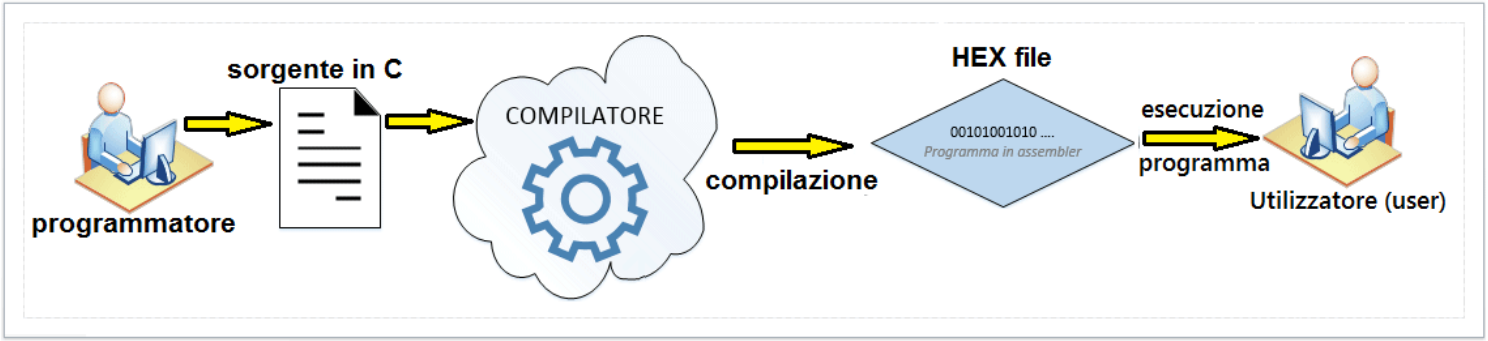
Vers. 07/07/2016

Linguaggio interpretato: un linguaggio interpretato necessità di un ulteriore programma (l'interprete appunto) per essere eseguito.

Un interprete è un programma in grado di eseguire altri programmi a partire dal loro codice sorgente (scritto con un linguaggio ad alto livello). Interpreta la sequenza di comandi, detta script, ed emula l'esecuzione associata ad ogni singola istruzione traducendola di volta in volta in istruzioni in linguaggio macchina.



Ai linguaggi interpretati si contrappongono i programmi compilati.



Un particolare programma detto **compilatore** trasforma il codice sorgente (scritto con un linguaggio ad alto livello) in **codice macchina** (assembler). Un programma compilato risulta mediamente più veloce rispetto al corrispondente interpretato poiché composto da istruzioni assembler. Ai primordi le CPU erano straordinariamente lente pertanto il vantaggio, in termini di velocità, di un programma compilato era assolutamente significativo e critico (secondi invece che minuti).

Variabili

Una **variabile** è una zona di memoria atta a contenere dei valori. Correttamente è un nome che Ruby associa ad un oggetto. Il nome di una **variabile inizia** (obbligatoriamente!) **con una lettera minuscola** seguita da altre lettere, numeri e _. Una variabile può essere manipolata esattamente allo stesso modo dell'oggetto che essa rappresenta.

Costanti

Le costanti sono come variabili nel senso che associano un nome ad un determinato oggetto. La differenza che nelle costanti il valore rimane fisso. Se provo a modificare il contenuto di una costante riceverò un avvertimento (warning).
Per definire una costante basta scrivere maiuscola solo la prima lettera del nome della variabile. E' convenzione mettere tutte le lettere maiuscole in modo da evidenziarlo.

Esercizi

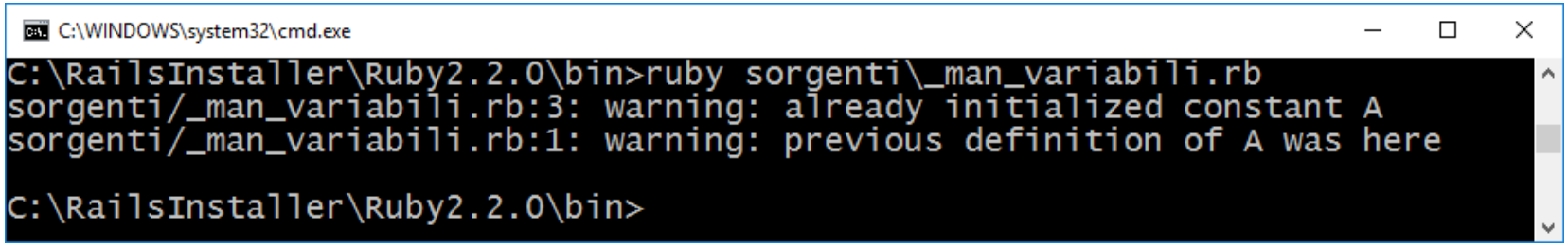
Quali di questi nomi è un nome di variabile legale ? Tra quelli validi quali identificano una costante ?

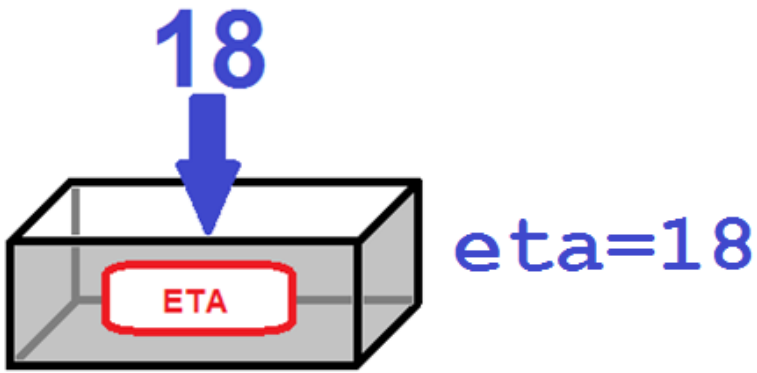
```
big_pay_day
my pal al
oh_noes!!!
Furlongs_per_fortnight
lucky7
76trombones
LEAP_YEAR_DAYS
theBigShow
```

Ruby e' un linguaggio di scripting interpretato, nel quale le variabili sono identificatori privi di tipo e e privi di dimensioni definita. Non occorre effettuare alcuna dichiarazione di tipo per le variabili utilizzate che vengono dichiarate automaticamente al loro primo utilizzo. Questo perché si tratta di un linguaggio interpretato

Esempi

```
A=1
b=2
A=b+A
b=b+A
```





Resta evidente che le variabili non sono tipizzate, ovvero non e' necessario definire a priori che tipo di valore contengono. Un altra cosa da tenere in considerazione e' che le variabili possono essere ridefinite in qualunque punto del codice, per cui possiamo prima definire inizialmente la variabile `a = 123` e successivamente ridefinirla con `a = "pippo"`.

Astraendo il concetto, immaginate una variabile come una scatole con una etichetta: la etichetta e' il nome della variabile (possibilmente univoco), lo spazio all'interno garantisce di conservare delle informazioni: il contenuto della variabile.

SCOPE DELLE VARIABILI

Argomento particolare e' lo **scope di una variabile**, detta anche visibilità di una variabile. Il *contesto* in cui una variabile e' inizializzata ne definisce la visibilità che tale variabile ha all'interno del codice. Una classe inizializzata in un determinato contesto potrebbe non essere visibile in un altro. Quando una variabile non e' visibile non e' possibile accedere al suo contenuto.

I **contesti** all'interno di uno script Ruby sono porzioni di codice racchiuse tra parole chiave begin ... end . Sono contesti di codice anche i blocchi (racchiusi tra do ... end o tra parentesi graffe) e eventuali cicli (for , while , etc.) e blocchi di istruzioni condizionali (if , case , etc.)

Esistono diverse tipologie di variabili, e il tipo e' riconosciuto sulla base della notazione usata per scrivere la "etichetta". Diamo uno sguardo alle tipologie di variabili:

- *variabili locali*
- *variabili globali*
- *variabili di istanza di una classe*
- *variabili di classe*
- *pseudo variabili e variabili predefinite*

Variabili locali

Le **variabili locali** sono le variabili il cui nome comincia per caratteri minuscoli **a-z** o con un underscore _ . La assegnazione di una variabile locale si effettua utilizzando l'operatore di assegnazione = . Le variabili locali possono contenere numeri, stringhe ed in generale istanze di oggetti.

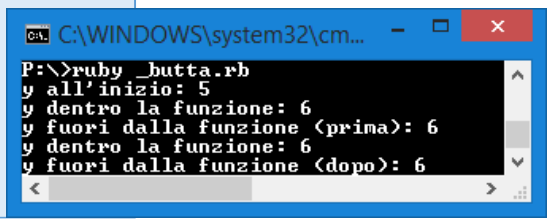
Variabili globali

Le **variabili globali** sono variabili che cominciano con il carattere **$** . Il termine globali e' in relazione alla loro visibilita': queste variabili sono visibili all'interno di tutto l'ambiente dell'interprete, indipendentemente dal contesto in cui ci si trova. Alcune di queste variabili sono predefinite in automatico dall'interprete all'avvio, come la variabile $0 che contiene il nome dello script in esecuzione (questa e' anche una **variabile predefinita**)

Un concetto importante, legato al comportamento delle variabili globali è quello di chiusura. Da un punto di vista informatico le chiusure permettono di passare le funzioni come parametri ad altre funzioni.

Si osservi un particolare nell'esecuzione di questo script: nonostante l'assegnamento il valore di **$y** resta immutato. Questo perchè **funz** risulta "***chiusa***" intorno al valore predefinito ovvero viene mantenuto il valore presente al momento della definizione della funzione. C'è una conservazione del contesto.

```
$y = 5
def funz
   $y=6 # se  omesso l'output sarà sempre 5
   puts("y dentro la funzione: #$y")
end
# Sy = 6 # mi da un warning (costante già iniz.)
puts("y all'inizio: #$y")
funz()
Sy = 7
puts("y fuori dalla funzione (prima): #$y")
funz()
puts("y fuori dalla funzione (dopo): #$y")
```

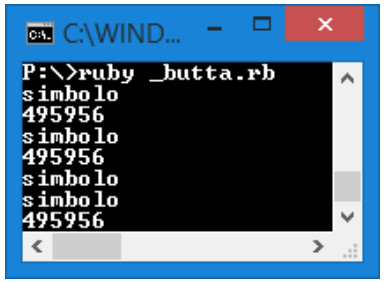


I simboli

Riconoscere un **simbolo** è abbastanza semplice. Solitamente i simboli sono definiti mediante una serie di caratteri preceduti da **`:`** . Quando la serie di caratteri, che identifica il simbolo, ha degli spazi, bisogna racchiuderli tra singolo apice oppure tra apici doppi.

A livello di codice un simbolo si definisce scrivendo **`:nomesimbolo`**. Quando si vuole definire un simbolo durante l'esecuzione di un programma (runtime) si può utilizzare il metodo **`str.to_sym`** definito nella classe **`String`**. Questo metodo trasforma direttamente la stringa, in un simbolo.

```ruby
pippo = :simbolo
puts pippo
puts pippo.__id__
# trasformo una stringa in un simbolo
pippo = "simbolo".to_sym
puts pippo
puts pippo.__id__
# si può usare anche direttamente
puts :simbolo
puts :simbolo.to_s
puts :simbolo.__id__
```



I simboli sono immutabili poiché, dopo essere stati invocati per la prima volta, l'interprete provvede ad assegnargli un numero intero che li identifica in modo univoco, come una specie di ID. I simboli sono quindi elementi del linguaggio di programmazione Ruby che hanno sia una *rappresentazione in stringa* che una *rappresentazione numerica*.

I simboli hanno un valore estremamente importante nella programmazione in Ruby. Possono essere utilizzati per rappresentare metodi di classe, variabili, etc. Noi li utilizzeremo come chiavi per accedere agli elementi degli **Hash**.

OPERATORI ARITMETICI

Suggerimento: provare alcune espressioni nell' "interactive Ruby Shell (comando irb)

+ -	addizione e sottrazione
*** /**	prodotto e divisione
%	resto divisione (operandi decimali arrotondati)
******	elevamento a potenza

OPERATORI DI ASSEGNAZIONE

=	assegnazione semplice
+=	aggiungi ed assegna
-=	sottrai ed assegna
***=**	moltiplica ed assegna
/=	dividi ed assegna
%=	resto ed assegna (**y%=x** equivale a **y=y%x**)
****=**	eleva ed assegna

OPERATORI DI RANGE

..	Costruisce un range con gli estremi inclusi [a,b]
...	Costruisce un range con estremo destro escluso [a,b)

OPERATORI DI CONFRONTO

==	uguale
!=	diverso
>	maggiore
<	minore
>=	maggiore o uguale
<=	minore o uguale
<=>	spaceship: non restituisce vero o falso come gli altri operatori ma:
	-1 se **a < b**
	0 se **a == b**
	1 se **a > b**
	nil se **a != b** e non esiste regola di comparazione per definire il maggiore o il minore

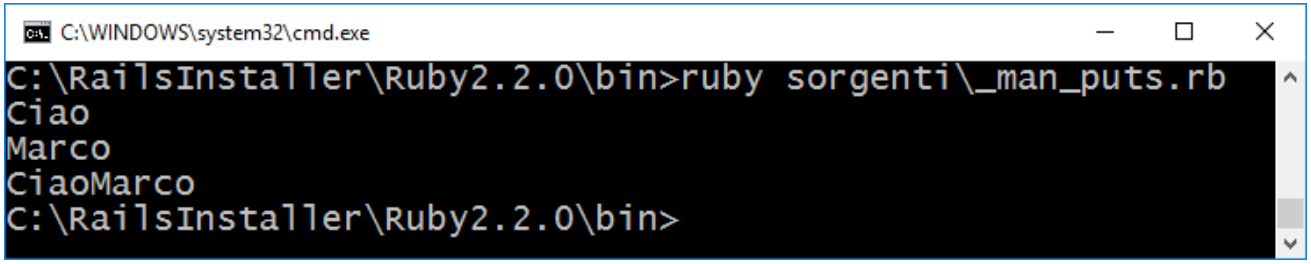
OPERATORI LOGICI

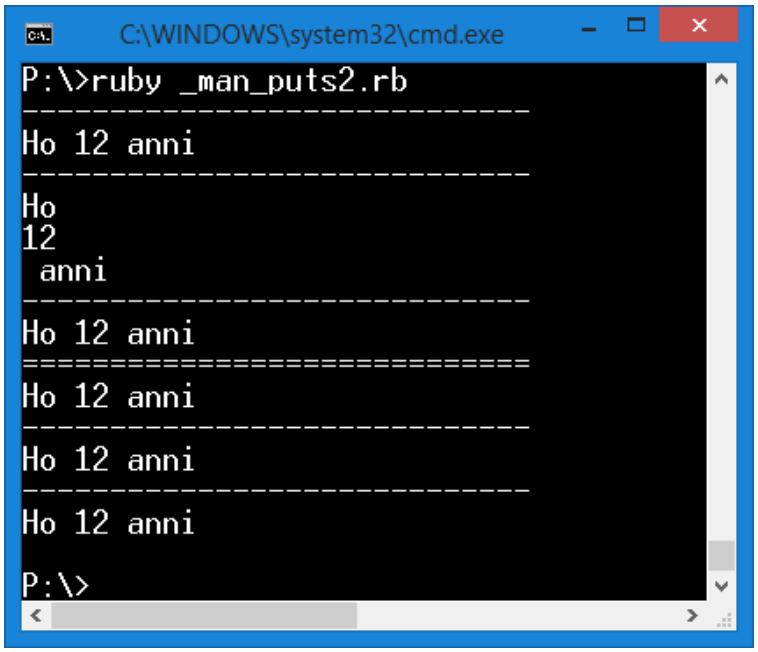
and , &&	**a && b** ritorna vero se sia **a** che **b** sono veri
or , ||	**a || b** ritorna vero se **a** oppure **b** sono veri
!	**!a** ritorna la sua negazione

PUTS / PRINT – Istruzione di Output

Serve per mostrare un risultato a video. La differenza è che **puts** aggiunge una CRLF (invio) mentre il **print** no.

```
puts "Ciao"
puts "Marco"
print "Ciao"
print "Marco"
```






```
anni=12
puts "----------"
puts "Ho "+anni.to_s+" anni"
puts "----------"
puts "Ho ", anni ," anni"
puts "----------"
puts "Ho #{anni} anni"
puts "=========="
print "Ho "+12.to_s+" anni\n"
puts "----------"
print "Ho ", 12 ," anni\n"
puts "----------"
print "Ho #{12} anni\n"
```

GETS – Istruzione di input

Serve per richiedere dei dati all'utente. E' una funzione che restituisce la stringa digitata compreso il carattere l'invio ("\n").

Esempio 1.gets

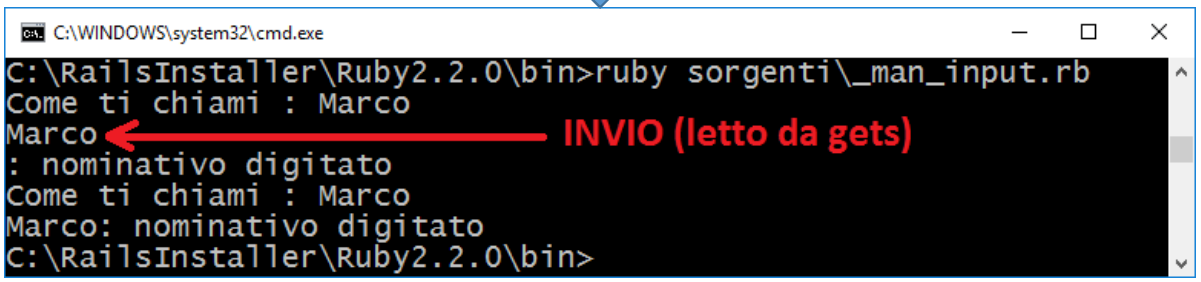
```
print "Come ti chiami : "
nominativo=gets
print nominativo + ": nominativo digitato\n"
print "Come ti chiami : "
nominativo=gets.chomp
print nominativo + ": nominativo digitato"
```

La sua sintassi è la seguente:

```
variabile=gets
```

Dove *variabile* contiene la stringa letta mediante **gets**. Per evitare di leggere invio possiamo usare:

```
variabile=gets.chomp
```



Esercizi I/O

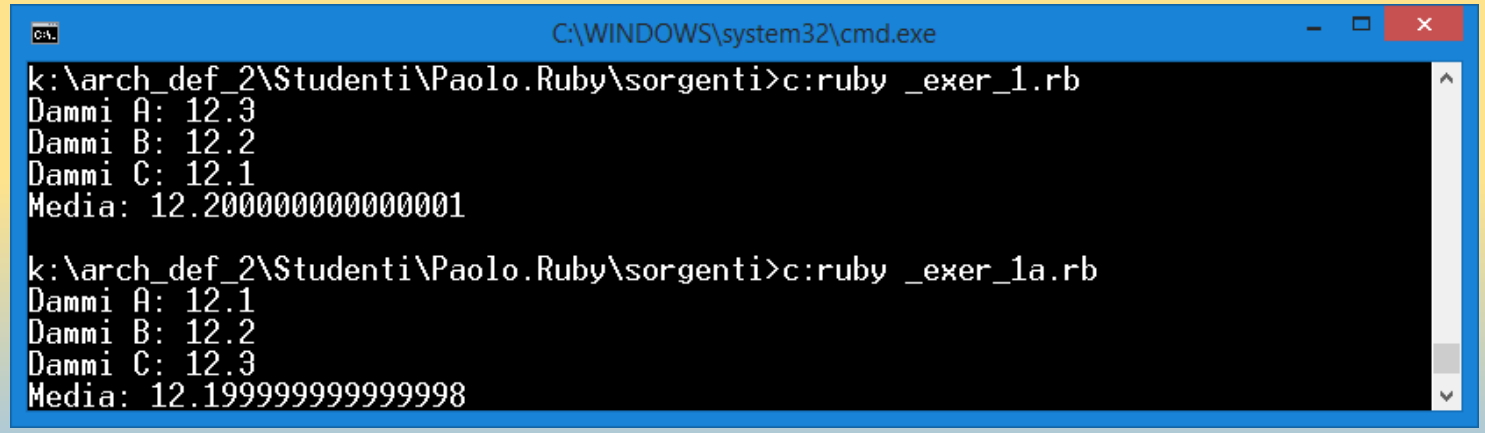
A. Scrivere un programma che chiede 3 numeri con decimali e ne calcola la media
B. Scrivere un programma che chiede un numero di minuti e lo converte in ore e minuti. Ad esempio se digito 375 l'output dovrebbe essere: "6 ore e 15 minuti"
C. Scrivere un programma che chiede il nome di un prodotto, il suo prezzo unitario, la quantità e calcola il costo totale mostrando una frase come la seguente "*3.5 Kg di Pere costano 27.3 euro*".

Soluzione A

```
print "Dammi A: "
a=gets
print "Dammi B: "
b=gets
print "Dammi C: "
c=gets
puts "Media: "+((a.to_f+b.to_f+c.to_f)/3.0).to_s
```

```
print "Dammi A: "
a=gets.to_f
print "Dammi B: "
b=gets.to_f
print "Dammi C: "
c=gets.to_f
puts "Media: #{((a+b+c)/3.0)}"
```

Si osservi che cambiando l'ordine dei valori della sequenza ottengo 2 risultati differenti dovuti ad errori di arrotondamento nella rappresentazione floating point.



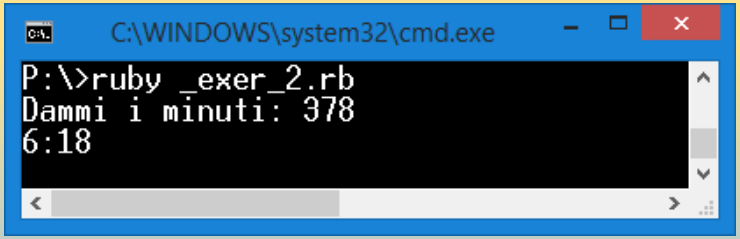
Soluzione B

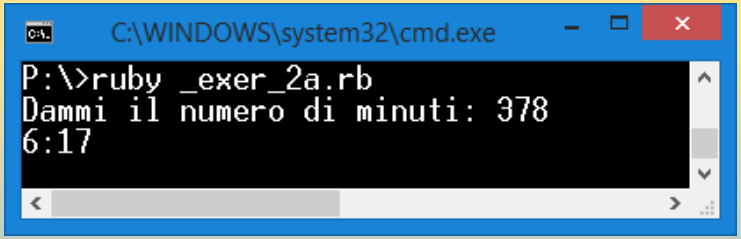
```
# INPUT
print "Dammi i minuti: "
minuti=gets.to_i
# ALGORITMO
ore = (minuti - (minuti % 60 ))/60
minuti = minuti % 60
# OUTPUT
puts "#{ore}:#{minuti}"
```

In alternativa:

```
# INPUT - prova 378
print "Dammi il numero di minuti: "
minuti=gets.to_f
# ALGORITMO
ore=minuti/60
decimale=ore-ore.to_i
minuti=(decimale*60).to_i
ore=ore.to_i
# OUTPUT
puts "#{ore}:#{minuti}"
```

Si osservi come l'uso del tipo float (soluzione alternativa) introduce la possibilità di errori dovuti alla granularità della rappresentazione floating point.

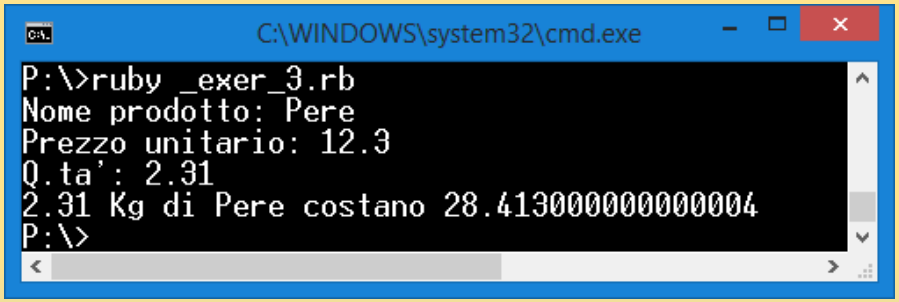




Soluzione C

```
print "Nome prodotto: "
nomeP=gets.chomp # Evito un a capo nel print finale
print "Prezzo unitario: "
prezzo=gets.to_f
print "Q.ta': "
qta=gets.to_f
print "#{qta} Kg di #{nomeP} costano #{qta*prezzo}"
```

Ecco un esempio di esecuzione

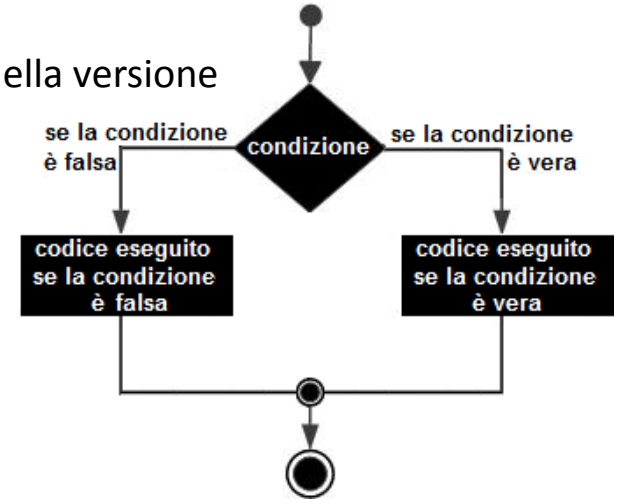


ISTRUZIONI CONDIZIONALI

Ogni linguaggio di programmazione possiede un'istruzione che permette di scegliere fra due strade in base ad una condizione.
L'istruzione Ruby è:

```
if condizione [then]
    istruzioni quando la condizione è vera
else
    istruzioni quando la condizione è falsa
end
```

Flow Chart della versione
IF ELSE END

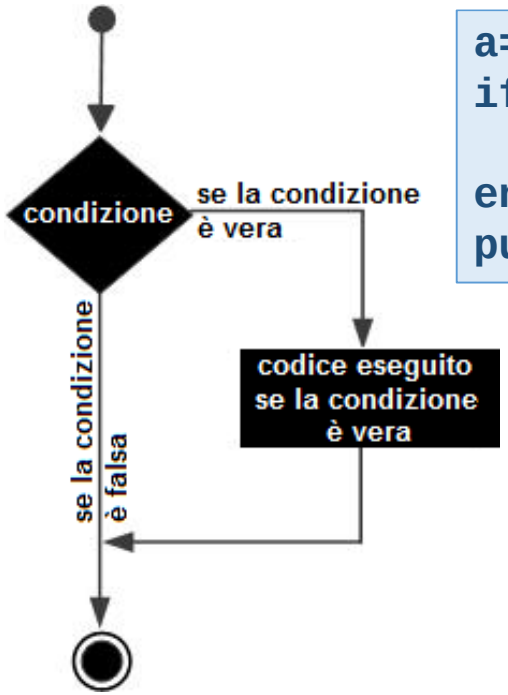



```
a=2
if (a % 2 == 1)
        puts "Dispari"
else
        puts "Pari"
end
```

La seconda parte (ramo **else**) è facoltativa, non è facoltativa l'istruzione **end**.

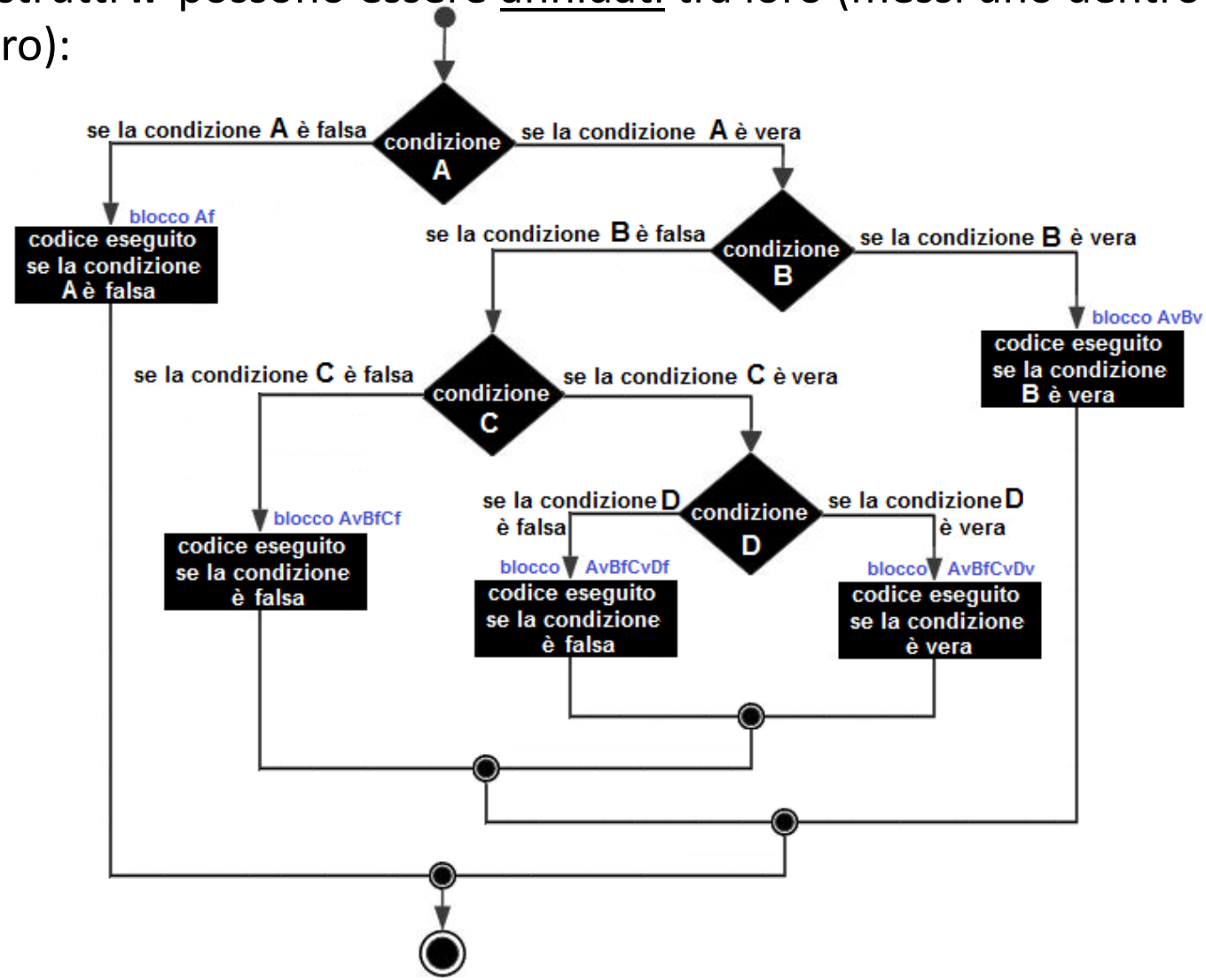
```
if condizione [then]
    istruzioni quando la condizione è vera
end
```

Flow Chart della versione ridotta **IF END**




```
a=-2
if (a < 0)
        a=-a
end
puts "Valore assoluto: "+a.to_s
```

I costrutti **IF** possono essere annidati tra loro (messi uno dentro l'altro):

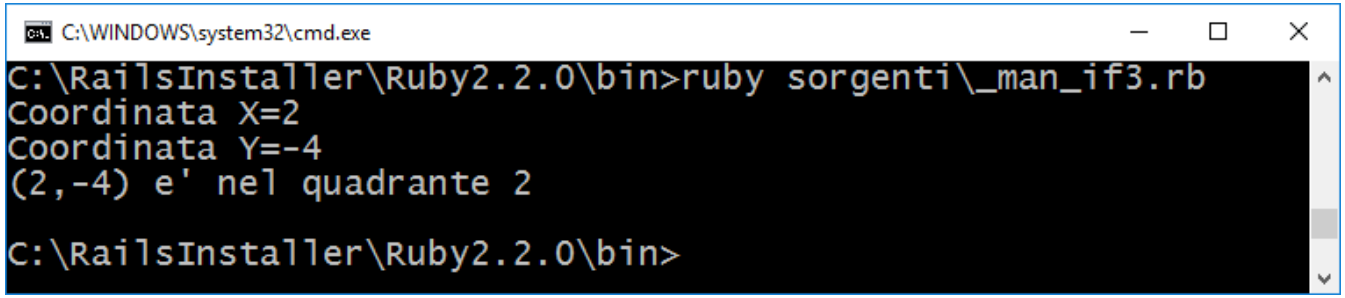


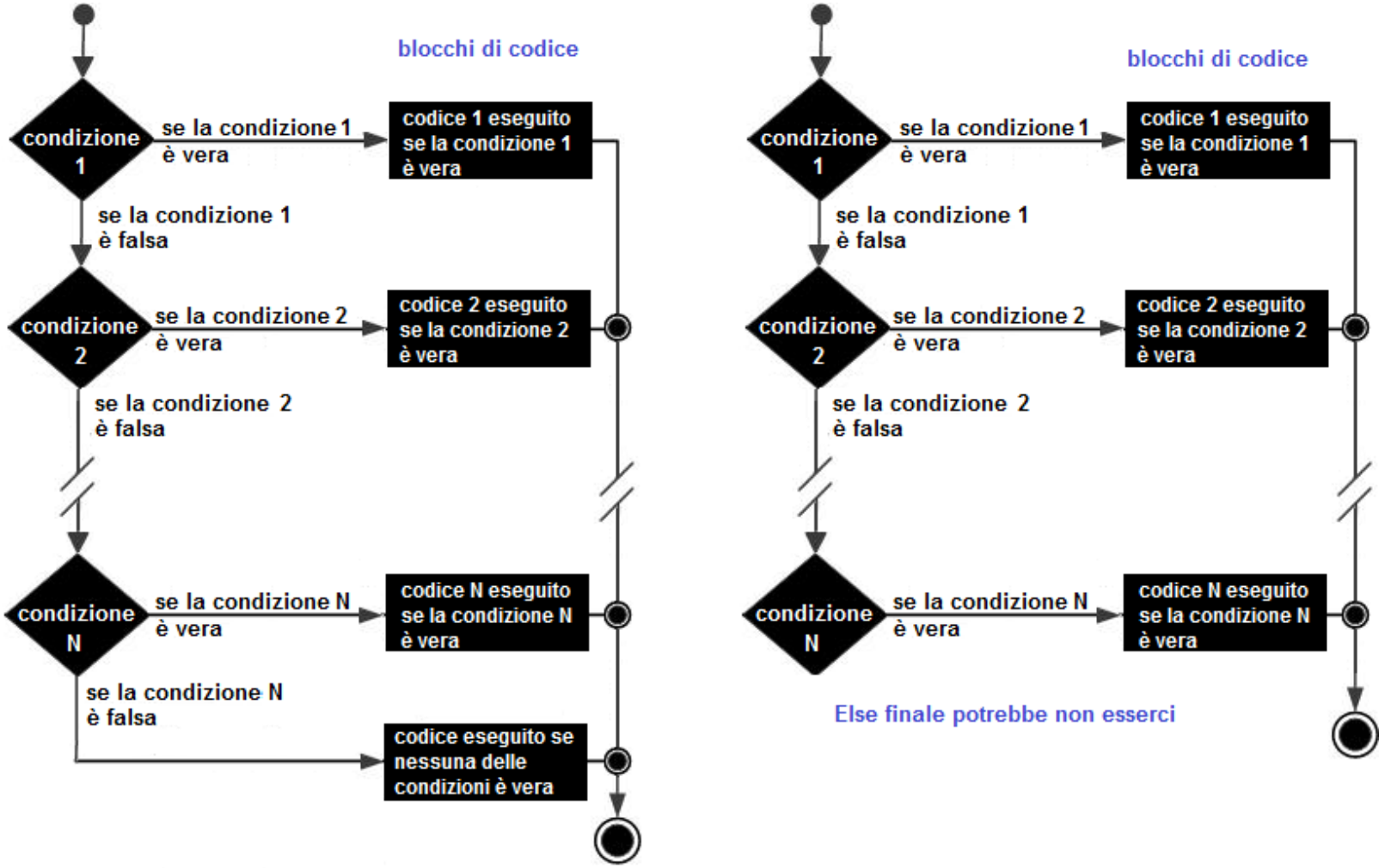
Nella forma più generale l'istruzione condizionale diventa:

```
if condizioneA [then]
     istruzioniA
elsif condizioneB [then]
     istruzioniB
elsif condizioneC [then]
     istruzioniC
. . .
else
     istruzioniElse
end
```

```
print "Coordinata X="
x=gets.to_i
print "Coordinata Y="
y=gets.to_i
# Valuto il quadrante di appartenenza
if (x > 0) && (y > 0)
        risposta="nel quadrante 1"
elsif (x > 0) && (y < 0)
        risposta="nel quadrante 2"
elsif (x < 0) && (y < 0)
        risposta="nel quadrante 3"
elsif (x < 0) && (y > 0)
        risposta="nel quadrante 4"
else
        risposta="sugli assi"
end
puts "("+x.to_s+","+y.to_s+") e' "+risposta
```

L'**else** finale potrebbe non esserci.





Flow Chart della versione completa **IF ELSIF ELSE END**

L’utilizzo di questa tipologia può semplificarci molto la vita, come nel caso di questa definizione a tratti:

```
x=gets.to_i
if x <= 1 then
   s=1
else
   if x <= 2 then
      s=2
   else
      if x <= 4 then
         s=3
      else
         if x <= 6 then
            s=4
         else
            if x <= 8 then
               s=5
            else
               s=6
            end
          end
       end
   end
end
puts s.to_s
```

```
x=gets.to_i
if x <= 1 then
   s=1
elsif x <= 2 then
   s=2
elsif x <= 4 then
   s=3
elsif x <= 6 then
   s=4
elsif x <= 8 then
   s=5
else
   s=6
end
puts s.to_s
```

La modalità di scrittura a destra è decisamente più semplice

```
x=gets.to_i
if x <= 1 then
   s=1
else
   if x <= 2 then
      s=2
   else
      if x <= 4 then
         s=3
      else
         if x <= 6 then
            s=4
         else
            if x <= 8 then
               s=5
            else
               s=6
            end
          end
       end
   end
end
puts s.to_s
```

Indentazione

I 2 programmi sono assolutamente identici. Nel primo le istruzioni risultano **indentate**. La maggior leggibilità del codice rende evidente la relazione tra la condizione e la sequenza di istruzioni sottesa.

```
x=gets.to_i
if x <= 1 then
s=1
else
if x <= 2 then
s=2
else
if x <= 4 then
s=3
else
if x <= 6 then
s=4
else
if x <= 8 then
s=5
else
s=6
end
end
end
end
end
end
puts s.to_s
```

Un'istruzione analoga all' **IF ELSIF ELSE END** è il **CASE.** Le parti **when** *successive* alla prima e **else** sono facoltative. Anche la parola chiave **then** può essere omessa. Ci possono essere più rami **when** ma di **else** ve ne può essere uno solo. L'istruzione **CASE** valuta una sola volta l'**espressione** e la confronta con il contenuto della **variabile** fino a quando non trova la prima sezione **when** per la quale è vera

```
case variabile
     when espressioneConfrontoA then
          IstruzioniA
     when espressioneConfrontoB then
          istruzioniB
     when espressioneConfrontoC then
          istruzioniC
     …
     else
         istruzioniElse
end
```

```
print "Digita un numero:"
v=gets.chomp.to_f
case v
   when 1..100
      risposta="numero reale da 1 a 100"
   when 1000, 10000
      risposta="1000 o 10000"
   when 0
      risposta "zero"
   else
      risposta="altro"
end
puts v.to_s+" e' "+risposta
```

```
print "Digita un numero:"
b=gets.chomp.to_f
case
   when b < 3 then
      puts "Minore di 3"
   when b == 3 then
      puts "Uguale a 3"
   when (1..10) === b then
      puts "nell'intervallo [1..10]"
   else
      puts "altro"
end
```

Il costrutto **case-when-else-end** (senza variabile dopo il case) può essere utilizzato come alternativa alla formulazione **if elsif else end** . I 2 codici successivi sono completamente equivalenti.

```
x=gets.to_i
if x <= 1 then
   s=1
elsif x <= 2 then
   s=2
elsif x <= 4 then
   s=3
elsif x <= 6 then
   s=4
elsif x <= 8 then
   s=5
else
   s=6
end
puts s.to_s
```

```
x=gets.to_i
case
   when x <= 1 then
      s=1
   when x <= 2 then
      s=2
   when x <= 4 then
      s=3
   when x <= 6 then
      s=4
   when x <= 8 then
      s=5
    else
      s=6
end
puts s.to_s
```

L' **if postfisso** è una istruzione condizionale posta alla fine di una riga di codice (post-fissa) in grado di inibirne la esecuzione.

```
Istruzione if condizione
```

È equivalente a:

```
if condizione [then]
    istruzione
end
```

In alternativa ad if mette a disposizione la keyword **unless** che effettua la valutazione di un codice solo se la condizione risulta essere falsa.

```
unless condizione [then]
    istruzione
end
```

È equivalente a:

```
if not condizione [then]
    istruzione
end
```

L'**operatore ternario**, comune a moltissimi linguaggi di programmazione, è una forma contratta per la espressione di un costrutto condizionale. La sintassi è la seguente:

```
risultato = (condizione) ? (valore se true) : (valore se false)
```

Esercizi IF

A. Costruire una breve applicazione che partendo da 5 numeri **a, b, c, d, e** (digitati da tastiera) calcola il valore minimo
B. Costruire una breve applicazione che partendo da 5 numeri **a, b, c, d, e** (digitati da tastiera) determina se il valore in **a** risulta compreso in uno degli intervalli [**b,c**] oppure [**d,e**] ? Si suppone che **b<c** e **d<e**.
C. Costruire una breve applicazione che legge 2 interi **a** e **b** e stabilisce se **a** è un multiplo di **b**
D. Costruire una breve applicazione che legge 3 numeri interi **a, b** e **c** e determina se **a** è un multiplo sia di **b** che di **c**
E. Costruire una breve applicazione che dato un carattere **a** e una stringa **s** determini se **a** è presente nella stringa **s**
F. Costruire una breve applicazione che descriva il risultato finale dello scrutinio per lo studente indicato. Nella stesura della soluzione si osservi che la risposta prodotta dovrà tener conto delle risposte fornite dall'utente in merito: 1) al *nominativo* , 2) al *sesso (pertanto si dovrà visualizzare la parola promosso/promossa a seconda),* 3) al *numero di debiti*. Uno studente che ha 3 o più debiti è bocciato. Uno studente con meno di 3 debiti è promosso
Alcuni esempi di output potrebbero essere i seguenti:

Rossi Maria** è stat**a promossa
Verdi Paolo** è stat**o promosso** con **due** debit**i
Gialli Giacomo** è stat**o bocciato

G. Costruire una breve applicazione che calcoli l'interesse maturato per un capitale **c** tenendo presente che il tasso **i** varia a seconda del capitale come indicato nella tabella sottostante:

Capitale<=-1000.00 € allora interessi passivi al 20.00%
Capitale >-1000.00 € e <0.00 € allora interessi passivi al 15.00%
Capitale >= 0.00 € e < 1000.00 € allora interessi attivi al 3.25%
Capitale >= 1000.00 € e < 5000.00 € allora interessi attivi al 5.25%
Capitale >= 5000.00 € allora interessi attivi al 8.75%

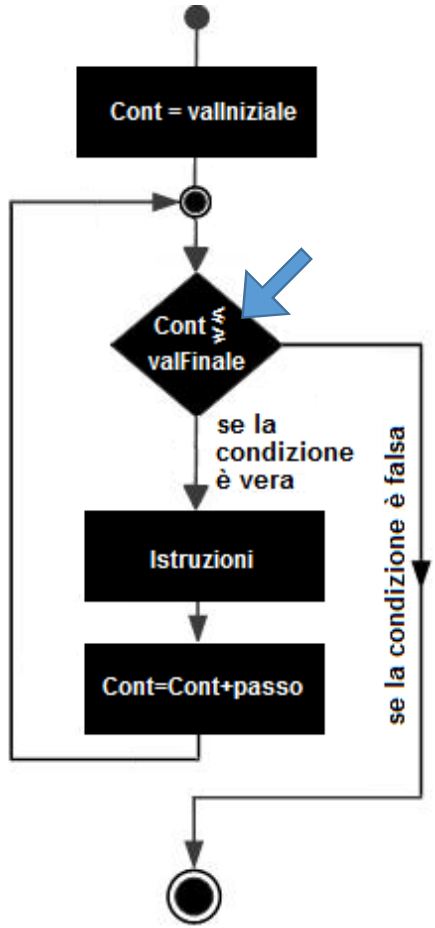
Soluzione A

...

```
. . .
```

ISTRUZIONI ITERATIVE - FOR

```
for cont in valIniz..valFine
   istruzioni
end
```



Il **FOR** viene eseguito in questo modo:

1) Esegue l'assegnamento iniziale **`cont=valIniz`**
2) Valuta la condizione **`cont<=valFine`**. Nel caso restituisca falso passa al punto 6)
3) Esegue le **`Istruzioni`** all'interno del **FOR**
4) Incrementa il contatore **`cont`** di 1.
5) Salta incondizionatamente al punto 2)
6) Termina il **FOR** e continua con le istruzioni successive.

```
song="Il carrozzone"
for i in 0...song.length
   print song[i]+","
end
```

```
C:\WINDOWS\system32\cmd.exe
C:\RailsInstaller\Ruby2.2.0\bin>ruby sorgenti\_man_loop2.rb
I,l, ,c,a,r,r,o,z,z,o,n,e,
C:\RailsInstaller\Ruby2.2.0\bin>_
```

Lo scorrimento a partire dall'ultimo elemento fino al primo può essere ottenuto in questo modo:

```
for cont in valMaggiore.downto(valMinore)
   istruzioni
end
```

```
for value in 10.downto(1)
  print "#{value} "
end
```

Il **FOR** , oltre agli oggetti range, può essere applicato agli **array**

```
for cont in [val1, val2, … , valn]
   istruzioni
end
```

```
for value in [1,2,3,4,5,6,7,8,9,10]
  print "#{value} "
end
```

Lo scorrimento a partire dall'ultimo elemento fino al primo può essere ottenuto in questo modo:

```
for cont in [val1, val2, … , valn].reverse
   istruzioni
end
```

```
for value in [1,2,3,4,5,6,7,8,9,10].reverse
  print "#{value} "
end
```

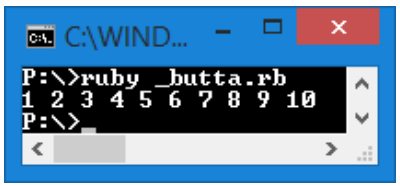
ITERATORI: EACH , TIMES, UPTO, DOWNTO e STEP, REVERSE_EACH

Gli iteratori sono metodi che eseguono un blocco di istruzioni tante volte quante sono le iterazioni richieste.

each

Il più semplice iteratore è **each**. Tutti gli oggetti iterabili (esempio **array, range** ed **hash**) hanno il metodo **each** che consente di applicare un blocco di istruzioni su ogni valore contenuto nell'oggetto contenitore. Vediamone un esempio:

```
[1,2,3,4,5,6,7,8,9,10].each { |i| print "#{i} " }
```



Per svolgere iterazioni più complesse si può utilizzare **do…end** per racchiudere un blocco di operazioni più complesse

```
[1,2,3,4,5,6,7,8,9,10].each do |valore|
     print "#{valore} "
end
```

L'operatore **each** può essere applicato anche all'oggetto **range**

```
(1..10).each  {|i| print "#{i} "}
```

Reverse_each

Simile all'iteratore **each** scorre l'intervallo da sinistra a destra. Risulta utile per lo scorrimento inverso dal valore più grande al più piccolo.

```
(1..10).reverse_each  {|i| print "#{i} "}
```

upto

L'iteratore **vInizio.upto(vFine)** esegue la sezione iterativa a partire dal numero **vInizio** fino al valore **vFine** (è necessario che **vFine** sia maggiore di **vInizio**). Quindi se vogliamo stampare valori da 1 a 10 dobbiamo scrivere la seguente istruzione:

```
1.upto(10)  {|i| print "#{i} "}
```

oppure

```
1.upto(10) do |i|
      print "#{i} "
end
```

downto

L'iteratore **vInizio.downto(vFine)** è simile all'iteratore è simile a **upto** solo che lo scorrimento risulta invertito dal valore più alto **vInizio** al valore più basso **vFine**. Quindi il seguente script stampa la sequenza da 10 a 1

```
10.downto(1)  {|i| print "#{i} "}
```

Equivalente a:

```
-10.upto(-1)  {|i| print "#{-i} "}
```

step

L'iteratore **vInizio.step(vFine,vStep)** è simile a **upto** solo che consente di definire anche il passo. Quindi il seguente script è equivalente a tutti i precedenti.

```
1.step(10, 1) { |i| print "#{i} "}
```

times

L'iteratore **times** è simile al classico ciclo **FOR** e permette di ripetere un blocco di istruzioni un numero **x** di volte (da 0 a n-1)

```
10.times {|i| print "#{i} "}
```

oppure

```
10.times do |i|
      print "#{i} "
end
```

Esercizi FOR

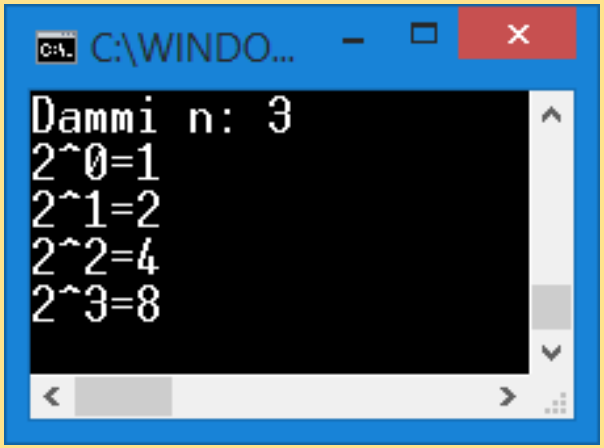
A. Costruire una breve applicazione che partendo da un numero **n** (digitato da tastiera) generi tutte le potenze di 2 fino a $\mathbf{2^n}$ quindi **1**, **2**, **4, 8, 16 … , $\mathbf{2^n}$**
B. Costruire un programma che genera **n** numeri interi compresi tra 1 e 100 e mostra la media di tali valori

```
rnd=Random.new(Random.new_seed) # inizializzo il generatore di numeri casuale
puts rnd.rand                   # genera un float in [0,1)
puts rnd.rand(2)                # genera un intero in [0,2)
puts rnd.rand(1.5)              # genera un float in [0,1.5)
puts rnd.rand(10..15)           # genera un intero in [10,15]
puts rnd.rand(10...15)          # genera un intero in [10,15)
puts rnd.rand(6.0..10.0)        # genera un float in [6.0,10.0]
puts rnd.rand(6.0...10.0)       # genera un float in [6.0,10.0]
# potevo anche scrivere
srand(Random.new_seed)
puts rand(n)
```

C. Leggere una sequenza di **n** numeri positivi e restituire il valore minimo e massimo
D. Costruire un programma che dato un intero **n** mostri i suoi divisori
E. Costruire un programma che legge due numeri razionali **a/b** e **c/d** (quindi dovrò leggere 4 variabili **a**, **b**, **c** e **d**) e successivamente mostra a video la frazione somma ridotta ai minimi termini.
F. Costruire una breve applicazione che legge una frase **S** e restituisce, per ogni vocale, il numero di occorrenze
G. Costruire un programma che visualizza il calendario del mese basandosi su due dati (richiesti all'utente!):
   - numero di giorni nel mese
   - giorno della settimana del primo del mese

Soluzione A

```
# INPUT
print "Dammi n: "
n=gets.to_i
# ALGORITMO+OUTPUT
r=1 # 2^0
for i in 0..n
   puts "2^#{i}=#{r}"
   # preparo il dato per il ciclo successivo
   r=r*2
end
```



Si poteva sfruttare, in alternativa, l'operatore potenza **

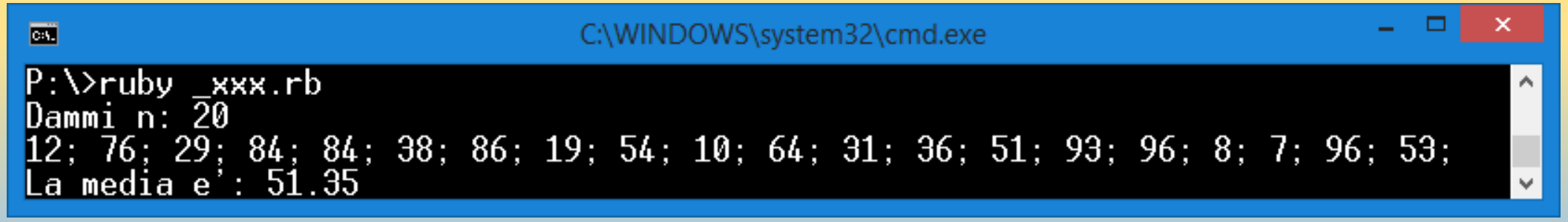
```
# INPUT
print "Dammi n: "
n=gets.to_i
# ALGORITMO+OUTPUT
for i in 0..n
   puts "2^#{i}=#{2**i}"
end
```

Oppure una delle forme iterative compatte

```
# INPUT
print "Dammi n: "
n=gets.to_i
# ALGORITMO+OUTPUT
0.step(n,1) { |i| puts "2^#{i}=#{2**i}" }
```

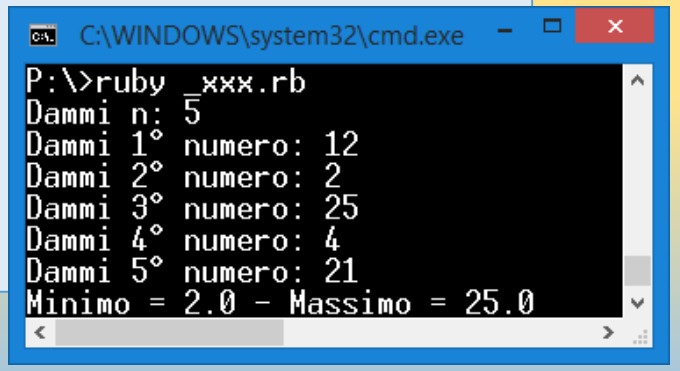
Soluzione B

```
print "Dammi n: "
n=gets.to_i
somma=0 # All'inizio non ho ancora valori per cui la somma e' zero
rnd=Random.new(Random.new_seed)
for i in 1..n # da 1 a n
   # genero in modo casuale il "valore corrente" x
   x =rnd.rand(1..100)
   print "#{x}; " # stampo per verifica
   # aggiungo il "valore corrente" x alla "somma totale"
   somma=somma+x
end # fine da
# stampo la media dividento la "somma totale" per n
puts "\nLa media e': #{ "%.2f" % (somma.to_f/n.to_f) }"
# Oppure
# puts "La media e': "+(somma.to_f/n.to_f).to_s
```



Soluzione C

```
print "Dammi n: "
n=gets.to_i
print "Dammi 1° numero: "
minimoattuale=gets.to_f  # il 1^ valore letto e' sia minimo che massimo
massimoattuale=minimoattuale
for i in 2..n # da 2 a n
    # chiedo l'i-esimo numero ("valore corrente")
    print "Dammi #{i}° numero: "
    x=gets.to_f
    # valuto se il valore corrente e' minore del "minimo corrente"
    # se si allora il valore corrente diventa il nuovo "minimo corrente"
    if (x<minimoattuale)
       minimoattuale=x
    end
    # valuto se il valore corrente e' maggiore del "massimo corrente"
    # se si allora il valore corrente diventa il nuovo "massimo corrente"
    if (x>massimoattuale)
       massimoattuale=x
    end
end # fine da
# stampo il minimo e il massimo
puts "Minimo = #{minimoattuale} - Massimo = #{massimoattuale}"
```



...

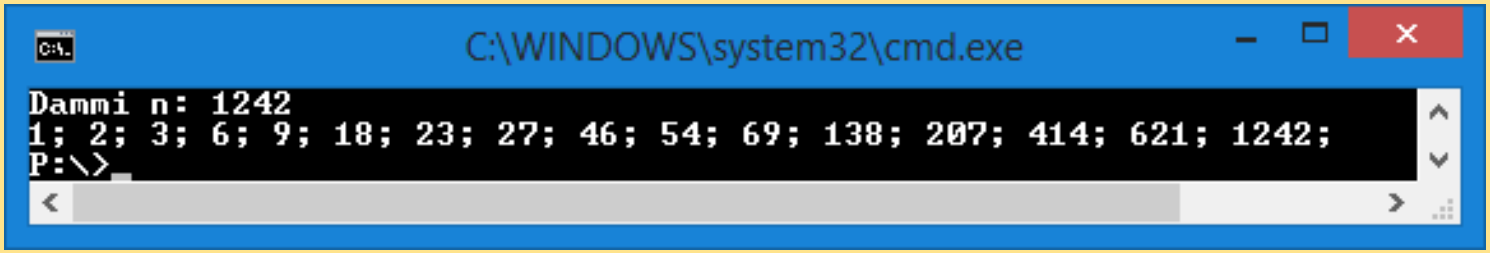
Si poteva, in alternativa, inizializzare **`minimoattuale`** e **`massimoattuale`** rispettivamente al massimo e al minimo valore che può assumere un **`Integer`**.

L'istruzione **`[42].pack('i')`** trasforma gli elementi dell'array in sequenze binarie corrispondenti alla direttiva '**`i`**' ovvero in un **`signed int`**

```
class Integer
  N_BYTES = [42].pack('i').size # dimensione di un signed int
  N_BITS = N_BYTES * 8
  MAX = 2 ** (N_BITS - 1) - 1
  MIN = -MAX - 1
end
print "Dammi n: "
n=gets.to_i
minimoattuale=Integer::MAX
massimoattuale=Integer::MIN
for i in 1..n # da 2 a n
    # chiedo l'i-esimo numero ("valore corrente")
    print "Dammi #{i}° numero: "
    x=gets.to_f
    # Aggiorno eventualmente il "minimo corrente"
    if (x<minimoattuale)
          minimoattuale=x
    end
    # Aggiorno eventualmente il "massimo corrente"
    if (x>massimoattuale)
          massimoattuale=x
    end
end # fine da
puts "Minimo = #{minimoattuale} - Massimo = #{massimoattuale}"
```

Soluzione D

```
print "Dammi n: "
n=gets.to_i
for d in 1..n
   if (n % d ==0)
      print "#{d}; "
   end
end
```



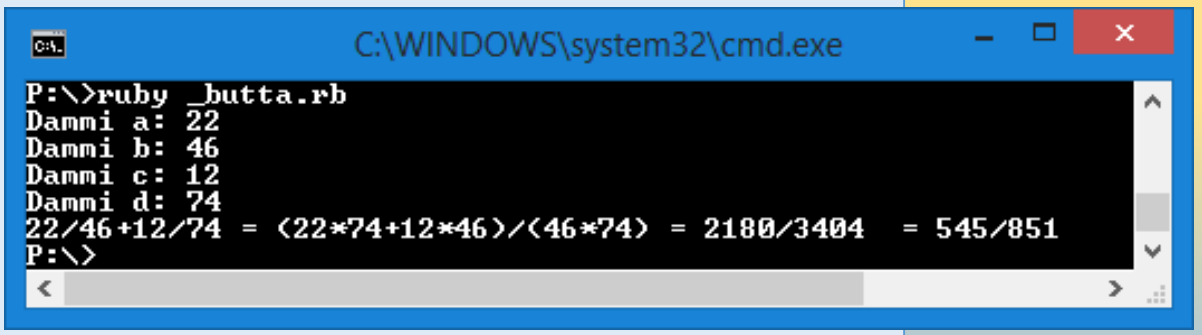
Oppure una delle forme compatte

```
print "Dammi n: "
n=gets.to_i
1.upto(n) { |d| print "#{d}; " if (n % d ==0) }
```

```
print "Dammi n: "
n=gets.to_i
for d in 1..n
   print "#{d}; " if (n % d ==0)
end
```

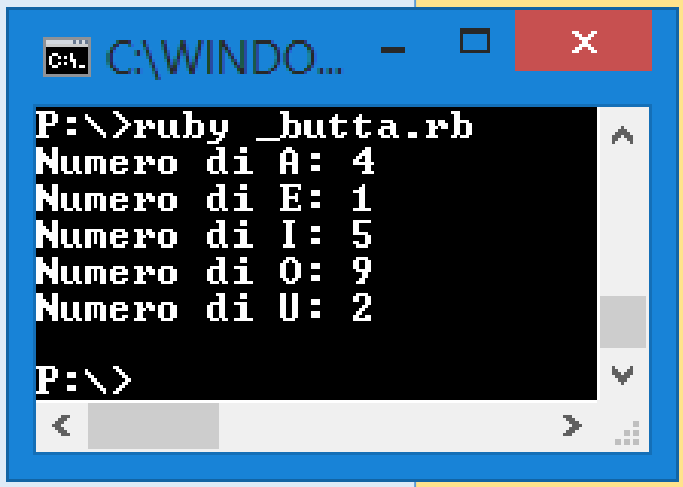
Soluzione E

```
def Leggo(c)
   print "Dammi #{c}: "
   return gets.to_i
end

# Leggo i 4 valori
a=Leggo("a")
b=Leggo("b")
c=Leggo("c")
d=Leggo("d")
# Calcolo la frazione somma (a/b+c/d) = (ad+cb)/(bd)
num = a*d+c*b # => numeratore:   parte sopra della frazione somma
den = b*d     # => denominatore: parte sotto della frazione somma
print "#{a}/#{b}+#{c}/#{d} = (#{a}*#{d}+#{c}*#{b})/(#{b}*#{d}) = #{num}/#{den} "
# Riduco ai minimi termini
minimo=( num < den ) ? num : den
minimo.downto(2) do |i|
   if (den % i == 0 ) && (num % i == 0 )
      den = den / i
      num = num / i
   end
end
# Stampo
print " = #{num}/#{den} "
```



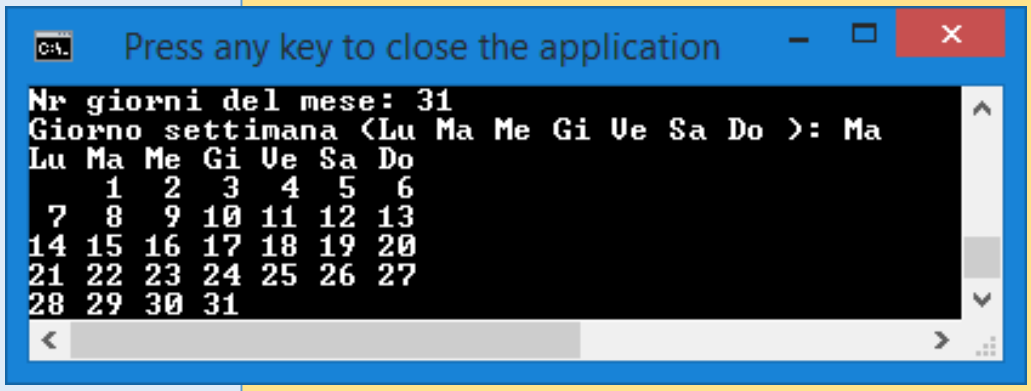
Soluzione F

```
# INIZIALIZZAZIONE
s="Ciao sono Giacomo e sto provando il linguaggio RUBY"
contaA, contaE, contaI, contaU = 0, 0, 0, 0
# ALGORITMO
for i in 0...s.length
      if (s[i]=="A") || (s[i]=="a")
         contaA=contaA+1
      end
      contaE=contaE+1 if s[i].upcase == "E"
      contaI=contaI.next if s[i].upcase == "I"
end
# scorro ogni singolo carattere
s.each_char { |c| contaU+=1 if (c.upcase=="U") }
# lunghezza stringa originale - quella priva del carattere O = nr di O
contaO=s.length-(s.upcase.gsub "O", "").length
# OUTPUT
print "Numero di A: #{contaA}\n"
print "Numero di E: #{contaE}\n"
print "Numero di I: #{contaI}\n"
print "Numero di O: #{contaO}\n"
print "Numero di U: #{contaU}\n"
```



Soluzione G

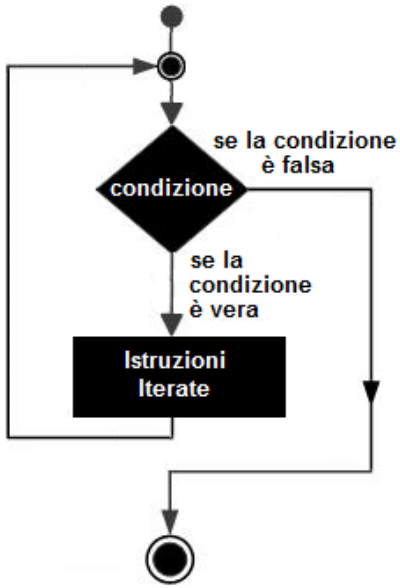
```
strW="Lu Ma Me Gi Ve Sa Do "
print "Nr giorni del mese: "
ggM=gets.to_i
print "Giorno settimana (#{strW}): "
ggW=gets.chomp
da=strW.index(ggW)/3
print strW+"\n"
print " "*strW.index(ggW)
for i in da...(ggM+da)
   print "#{"%2d " % (i-da+1) }"
   if (i % 7 == 6)
      print "\n"
   end
end
```



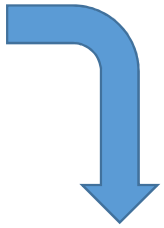
ISTRUZIONI ITERATIVE – WHILE … END

RUBY presenta diversi tipi di istruzioni cicliche. L'istruzione ciclica **`while`** continua il loop fino a quando la condizione resta vera

```
while condizione do
    istruzioni
end
```




```
n = 1
while n <= 10 do
    puts "n vale ora #{n}"
    n=n+1
end
```

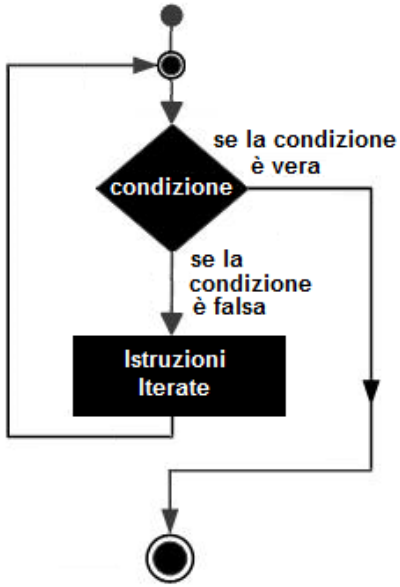


```
C:\WINDOWS\system32\cmd.exe
C:\RailsInstaller\Ruby2.2.0\bin>ruby sorgenti\_man_loop1.rb
n vale ora 1
n vale ora 2
n vale ora 3
n vale ora 4
n vale ora 5
n vale ora 6
n vale ora 7
n vale ora 8
n vale ora 9
n vale ora 10

C:\RailsInstaller\Ruby2.2.0\bin>
```

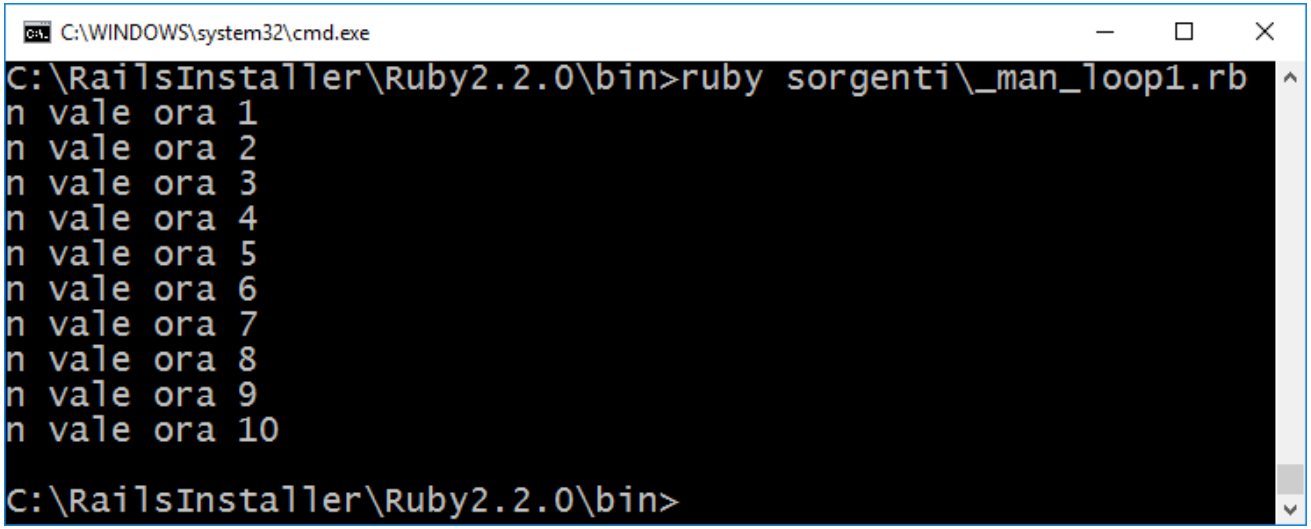
ISTRUZIONI ITERATIVE – UNTIL … END

L'istruzione ciclica **until** continua il loop fino a quando la condizione resta falsa

```
until condizione [do]
   istruzioni
end
```



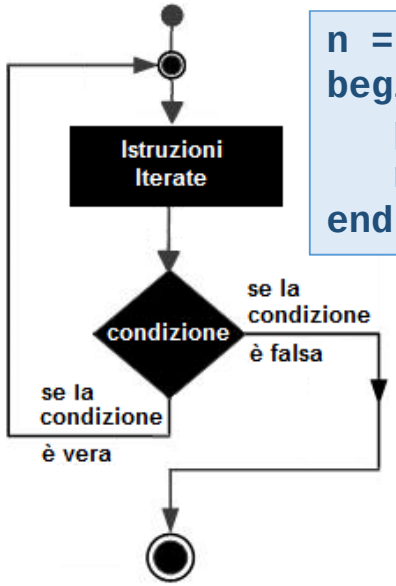

```
n = 1
until n > 10 do
   puts "n vale ora #{n}"
   n=n+1
end
```



ISTRUZIONI ITERATIVE – BEGIN … WHILE/UNTIL

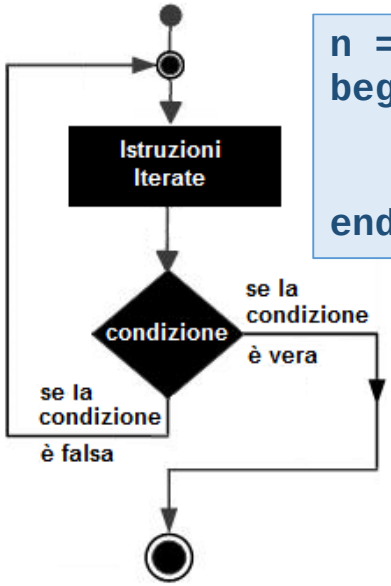
L'istruzione ciclica **until** e **while** possono essere scritte anche nella seguente forma (utile quando almeno una volta la parte iterata deve essere riscritta)

```
begin
   istruzioni
end while condizione
```




```
n = 1
begin
   puts "n vale ora #{n}"
   n+=1
end while n <= 10
```

```
begin
   istruzioni
end until condizione
```




```
n = 1
begin
   puts "n vale ora #{n}"
   n+=1
end until n > 10
```

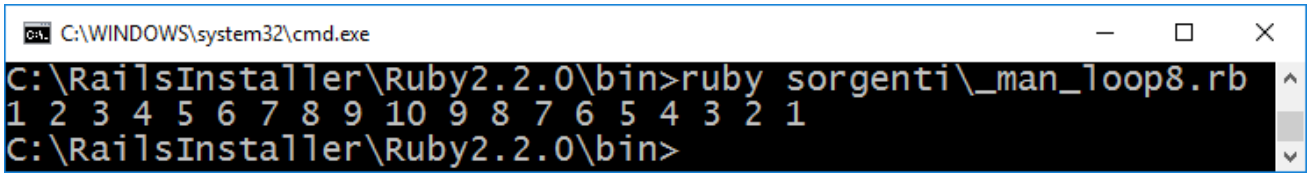
Per uscire da questi cicli in anticipo si usa **break**.

Ruby consente l'uso di **`while`** e **`until`** come modificatori il che significa che possono essere messi in fondo ad una istruzione (come avviene del resto anche per l' **`if`**) per ripeterla. Otteniamo quindi una versione più breve dell'istruzione di loop.

```
Istruzione while condizione
```

```
Istruzione until condizione
```

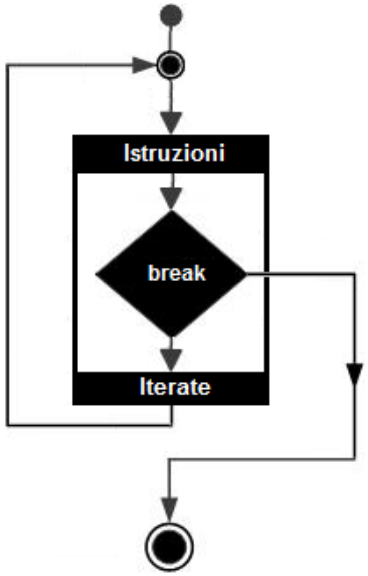
```
i=0
print "#{i+=1} " while i < 10 # 1..10
print "#{i-=1} " until i < 2 # 9..1
```



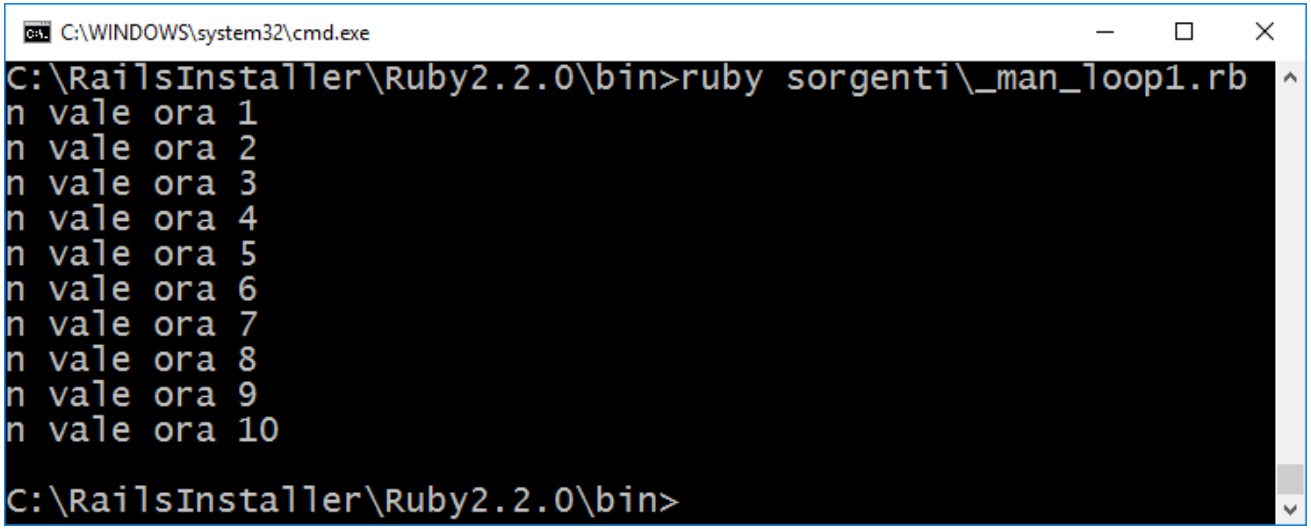
ISTRUZIONI ITERATIVE – LOOP … END

L'istruzione ciclica **loop** continua il loop all'infinito. In realtà utilizzando l'istruzione **break if** possiamo definire una condizione di uscita. La parola chiave **do** in questo caso è obbligatoria.

```
loop do
   istruzioni
end
```



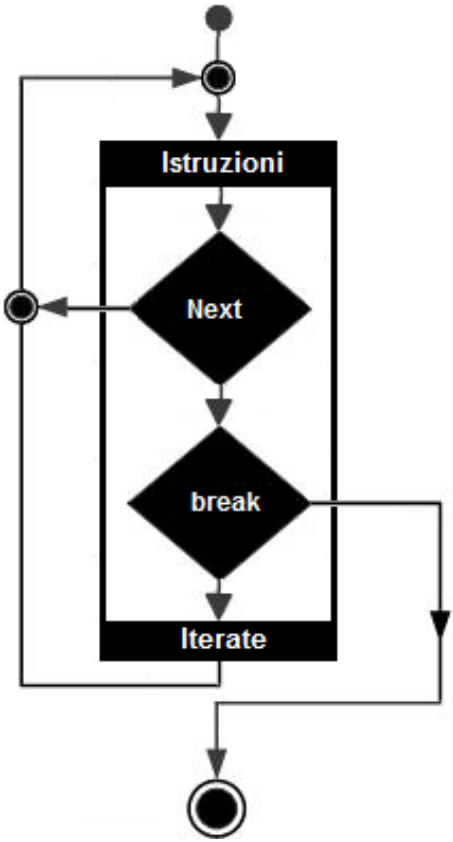

```
n = 1
loop do
   puts "n vale ora #{n}"
   n+=1
   break if n>10
end
```



L'istruzione **next** permette di saltare il ciclo corrente e passare al loop successivo

```
loop do
    ...
    next if condizione
    ...
    break if condizione
end
```

```
n=0
loop do
  n+=1
  next if n % 2 == 1
  print "#{n} "
  break if n>=10
end
```




```
C:\WINDOWS\system32\cmd.exe
C:\RailsInstaller\Ruby2.2.0\bin>ruby sorgenti\_man_loop6.rb
2 4 6 8 10
C:\RailsInstaller\Ruby2.2.0\bin>_
```

L'istruzione **break** ammette anche un argomento. In questo caso il **loop** restituisce il valore indicato in **break**

```
i=0
puts(loop do
    i=i+1
    print "#{i} "
    break ' - Ho finito!' if i==10
end)
```

```
C:\WINDOWS\system32\cmd.exe
C:\RailsInstaller\Ruby2.2.0\bin>ruby sorgenti\_man_loop7.rb
1 2 3 4 5 6 7 8 9 10  - Ho finito!
C:\RailsInstaller\Ruby2.2.0\bin>_
```

Esercizi WHILE

A. Costruire un programma che continua a richiedere la digitazione di un numero fino a quando non inserisco un valore compreso tra 10 e 20
B. Costruire un programma che legge una sequenza di numeri la cui lunghezza non è nota a priori (la sequenza si considera terminata quando l'utente digita 0).
C. Costruire un programma che legge una sequenza di numeri la cui lunghezza non è nota a priori (la sequenza si considera terminata quando l'utente digita 0). Al termine della lettura il programma visualizza:
   a) il minimo valore digitato
   b) il massimo valore digitato
   c) la somma dei valori digitati

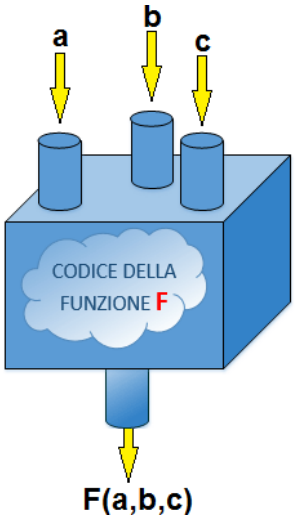
```
. . .
```

Soluzione A

...

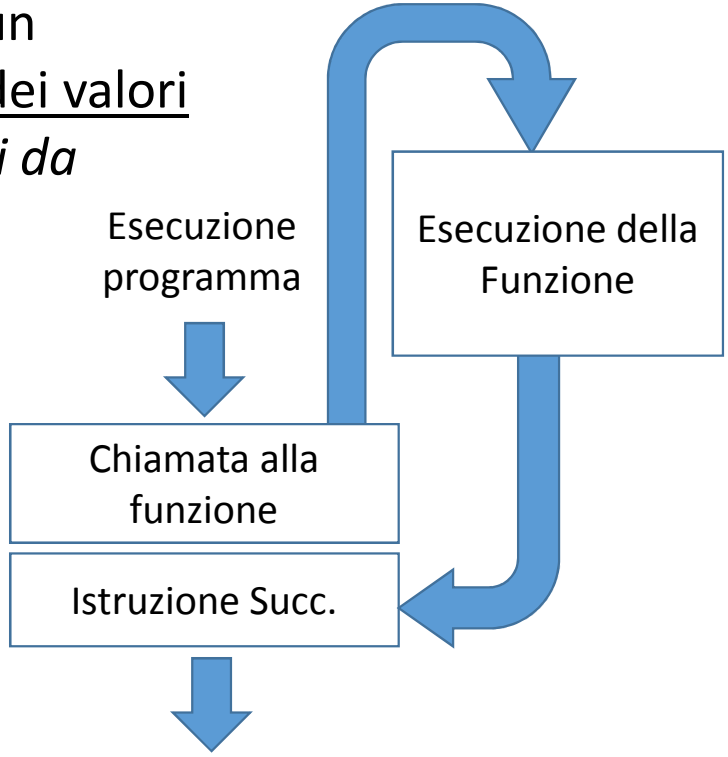
```
. . .
```

FUNZIONI



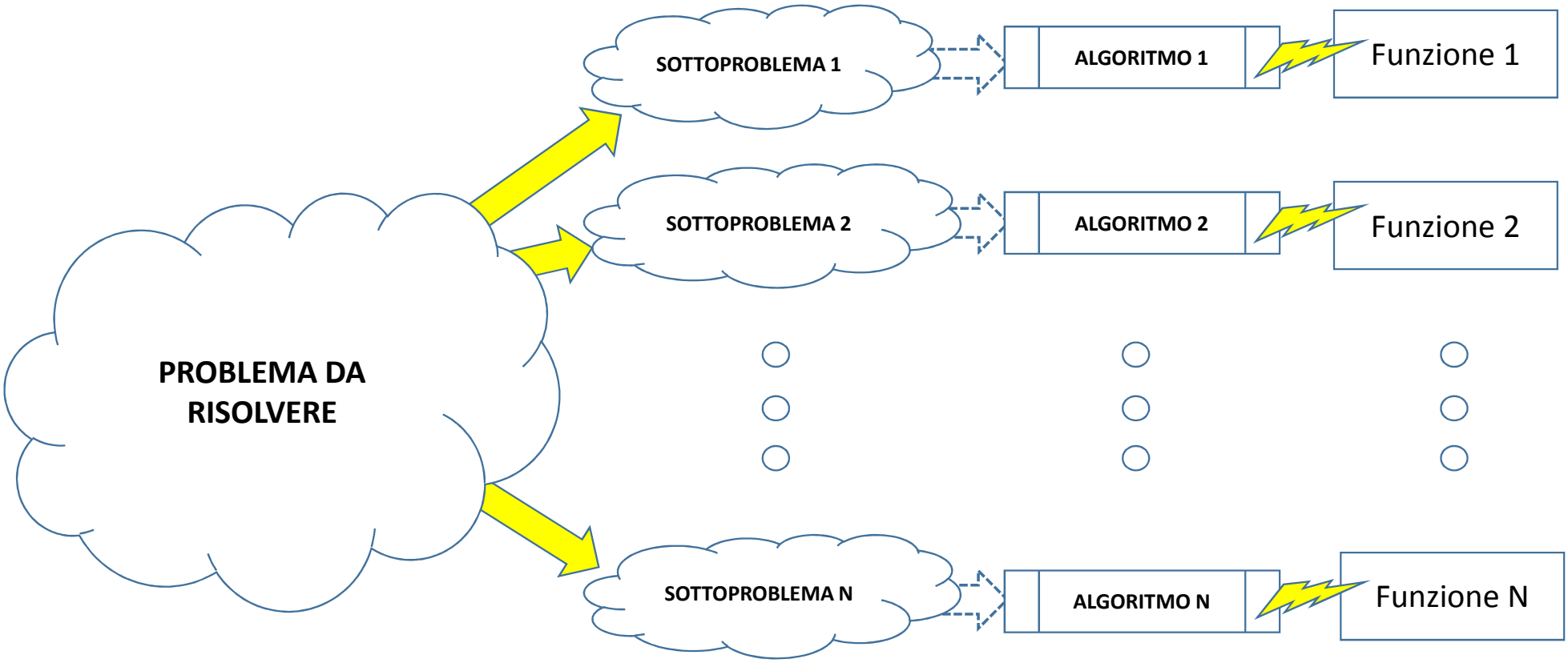
Una **funzione** è un particolare costrutto sintattico che permette di raggruppare diverse righe di codice in una struttura unica che può essere richiamata piu' volte attraverso il suo nome. Le funzioni permettono di racchiudere una sequenza codice che esegue uno specifico algoritmo.

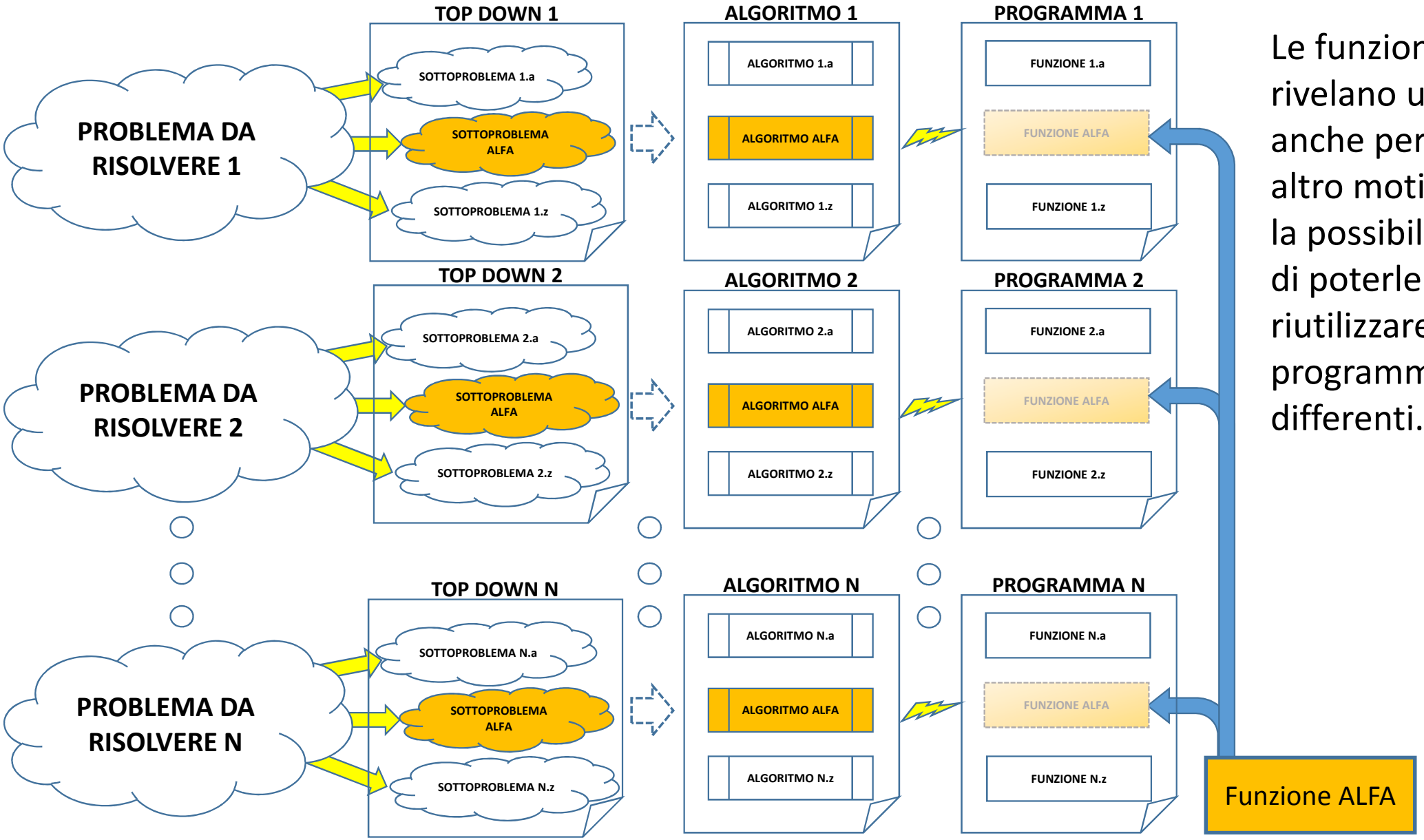
Una funzione (subroutine) è sostanzialmente un **sottoprogramma** al quale è possibile passare dei valori sotto forma di parametri (*rappresentano i dati da elaborare!*).



Inserendo nel programma principale il nome della funzione con i valori dei parametri (se presenti!) si richiama l'**esecuzione del sottoprogramma associato alla quella funzione.** L'esecuzione della funzione termina appena incontra un punto di uscita (`end` o `return`). Il flusso del programma principale prosegue dall'**istruzione successiva** alla chiamata.

Le funzioni si rivelano utili anche per un altro motivo: con esse, infatti, è possibile suddividere un programma in unità più piccole, più facili da scrivere e manutenere.





Le funzioni si rivelano utili anche per un altro motivo: la possibilità di poterle riutilizzare in programmi differenti.

I vantaggi nell'utilizzo di sottoprogrammi possono essere cosi riassunti:

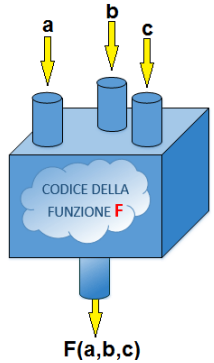
- **Modularità del codice**: la scomposizione in sottoprogrammi facilita la lettura e la comprensione dell'algoritmo;
- **Riutilizzo del codice**: Una subroutine può essere utilizzata
  - in più punti di uno stesso programma;
  - può essere utilizzata in altri programmi
- **Manutenzione del codice**: Correzioni di errori e miglioramenti dell'efficienza si concentrano su una sola porzione di codice

Ogni funzione deve svolgere un solo compito. Funzioni complesse (che svolgono più compiti) devono essere scomposte secondo la logica **top-down** (approccio illustrato successivamente in queste slide).

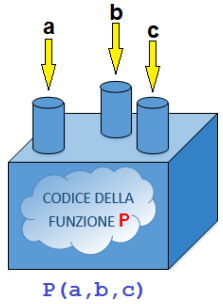
La distinzione **Programma principale** e **sottoprogramma** evidenzia il rapporto tra due funzioni. La prima utilizza la seconda per svolgere una parte del compito assegnato.

Concettualmente le funzioni possono essere ricondotte a due tipi:

- **Funzioni**: sottoprogrammi descrivibili come vere e proprie funzioni matematiche che possiedono un dominio ed un codominio. Pertanto restituiscono un risultato.



- **Procedure**: sottoprogrammi che con la loro esecuzione modificano lo stato di alcune variabili di programma e contrariamente alle funzioni non restituiscono alcun valore.

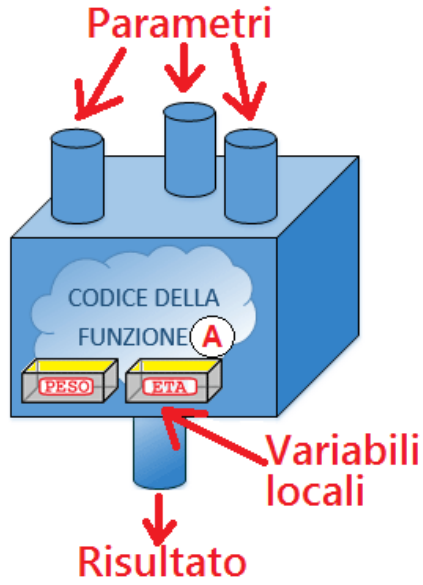


Funzioni e procedure hanno:

- ❑ un unico punto d'ingresso
- ❑ uno o più punti di uscita

Inoltre consentono:

- ❑ la possibilità di poter accettare in ingresso uno o più **parametri** passati dal **routine chiamante**;
- ❑ la definizione di variabili visibili solo all'interno della subroutine (**variabili locali**);
- ❑ (solo se abbiamo return) la possibilità di restituire un dato come risultato della loro esecuzione



La definizione di una funzione e' caratterizzata dai seguenti elementi:

```
# Commento
def nomeFunzione (arg1, … , argN)
   corpo della funzione ...
   [return valorerestituito]
end
```

- **def**: questa keyword indica all'interprete che sta iniziando la dichiarazione di una funzione
- **nomeFunzione**: Rappresenta il nome della funzione e la identifica in modo univoco. Specificare lo stesso nome per due funzioni porta alla sovrascrittura della funzione interpretata prima con quella interpretata successivamente. Come le variabili, anche le funzioni devono sottostare a delle regole di visibilità.
- **(arg1, ... , argN)**: gli argomenti della funzione. Si tratta di variabili che vengono inizializzate con l'invocazione della funzione e hanno visibilità solo all'interno della funzione. Se la funzione non accetta argomenti in ingresso le parentesi si omettono.
- **corpo della funzione**: sono le linee di codice, racchiuse tra le keyword **def** e **end**, e descrivono le operazioni che la funzione esegue.
- **return** : questa peculiare keyword determina quale sara' il valore restituito dalla funzione
- **# commento**: posto prima della funzione risulta utile per ricordare che cosa fa la funzione, quali siano i valori presi in input e quali siano quelli ritornati in output. Questa parte non necessaria ma buona norma inserirla.

Il passaggio dei parametri è generalmente a valore. Non è così per i vettori

```
def Incrementa(x)
   x=x+1
end

x=1
Incrementa(x)
puts "x=#{x}" # stampa 1
```

```
def IncrementaV(x)
   x[0]=x[0]+1
end

y=[1]
IncrementaV(y)
puts "y=#{y[0]}" # stampa 2
```

Nella creazione di un metodo è possibile definire valori di default (**defaulted parameters**):

```
def Incrementa(x,passo=1)
   x=x+passo
   print "#{x}\n"
end

Incrementa(2)   # mostra 3
Incrementa(2,4) # mostra 6
```

Una novità di Ruby 2.0 sono invece i cosiddetti **named parameters** o keyword arguments. Ecco un esempio

```
def Potenza(base: , esponente:1)
   return base**esponente
end

puts Potenza(esponente:2,base:3) # 9
puts Potenza(base:2,esponente:3) # 8
puts Potenza(base:2)             # 2
# non ammessa! Potenza(2,3)
```

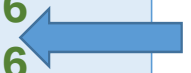
Si osservi che i parametri possono avere una Inizializzazione di default. Se vi sono dei parametri normali i named parameters devono devono sempre essere messi successivamente dopo di essi.

E' possibile non definire a priori i parametri da passare ad un metodo grazie all'utilizzo del simbolo "*" che prende il significato di "qualsiasi parametro".

```
def Somma(* param)
   s=0
   param.each { |x|  s=s+x }
   return s
end

puts Somma(1,2,3)      # mostra 6
puts Somma(1,11,2,34) # mostra 48
```

E' interessante notare che il nostro asterisco può essere usato anche nell'ambito del richiamo di un metodo per passare ad esso un array di modo che ogni elemento dello stesso sia visto come un parametro indipendente. L'esempio seguente chiarisce in modo semplice:

```
def f(a,b,c)
   return a*b*c
end
v=[1,2,3]
puts f(*v)    # 6
puts f *v     # 6
puts f 1,2,3  # 6
```



Si osservi che in Ruby le () possono essere omesse

Ogni metodo in Ruby restituisce un valore. A meno di altre indicazioni esso **è il valore dell'ultima espressione presente nel metodo stesso**. Quindi un metodo tipo restituisce 1

```
def func
   x = 0
   y = 1
end
```

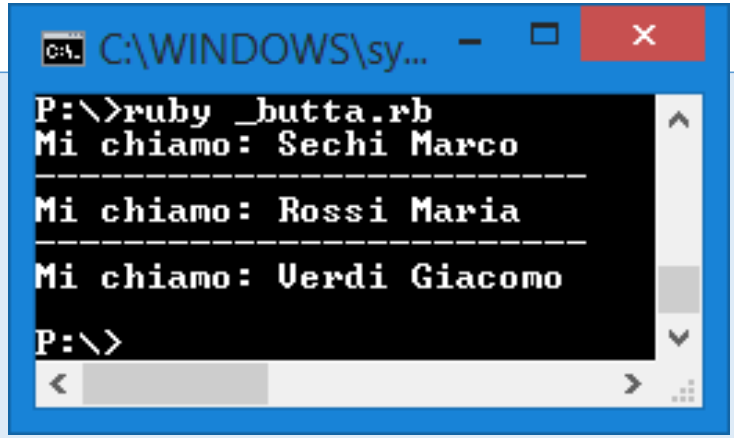
Blocchi di codice

Un **blocco di codice** non è altro che una porzione di codice racchiusa all'interno di una coppia di parentesi graffe:

$$\{|x_1, x_2, ...x_M| \ istruz_1 \ ; \ istruz_2 \ ; \ \ldots \ ; \ istruz_N\}$$

La parola chiave **`yield`** si occupa di richiamare il blocco di codice e di mandarlo in esecuzione. Terminata questa fase il controllo viene restituito all'istruzione successiva.

```
def funz (x, y)
   yield "Sechi","Marco"
   puts "-"*25
   yield "Rossi", "Maria"
   puts "-"*25
   yield x, y
end

funz("Verdi","Giacomo") {|x,y| print "Mi chiamo: "; puts(x+" "+y) }
```



I blocchi di codice non sono oggetti. Tuttavia è possibile creare degli oggetti che rappresentano dei blocchi e quindi operare su di essi con gli strumenti caratteristici proprio per gli oggetti. Esiste un preciso oggetto che rappresenta un blocco di istruzioni e questo oggetto è **Proc** (ovvia abbreviazione di Procedure).

Come da definizione gli oggetti **Proc** sono blocchi di istruzioni legati a variabili ed utilizzabili in varie parti del codice del programma. Come tutto gli oggetti è dotato del metodo **new** che consente di dar vita ad una nuova istanza. Vediamo subito un esempio:

```
f = Proc.new{|x,y| x * y}
z = f.call(2,3)
puts(z) # stampa 6
```

Esistono altri 2 modi per invocare il blocco di istruzioni associato all'oggetto di tipo proc.

```
z = f[2,3]
z = f.(2,3)
```

E' possibile anche lavorare senza parametri e con più istruzioni

```
f = Proc.new {
   x = 0
   y = 2
   puts(x + y)
   puts(x - y)
}
puts(f.call()) # 2\n-2
```

In alternativa a **proc** è possibile utilizzare il metodo **lambda.** Si tratta di un metodo del modulo kernel e si comporta quindi come una funzione globale. Tale modulo è infatti incluso in **Object** ed è quindi disponibile universalmente in ambito Ruby. Una differenza tra **proc.new** e **lambda** consiste nel fatto che il secondo costrutto, quindi **lambda**, controlla il numero di parametri in ingresso, diversamente da **proc**.

```
f = lambda{|x, y| x * y}
s = f.call(2,3)
puts(s)
```

La keyword **lambda** può essere sostituita dal simbolo **->** , con gli argomenti messi all'esterno delle parentesi graffe che continuano invece a contenere il corpo della funzione**Proc** (ovvia abbreviazione di Procedure).

```
f = lambda{|x, y| x * y}
s = f.call(2,3)
puts(s)
```

```
f = -> (x,y){x * y}
s = f.call(2,3)
puts(s)
```

Nel caso in cui non vi fosse apparentemente nessun valore di ritorno, come ad esempio nel caso di un metodo siffatto che in realtà restituisce il classico nil che ovviamente potrebbe essere attribuito senza problemi ad una variabile.

```
def func
   puts "Ciao"
end
puts func==nil # true
```

return permette di restituire più di un valore come si vede nel presente esempio. Il valore restituito è un array.

```
def funct(a, b, c)
   return a+b,a+c,b+c
end
print funct(1,2,3) # [3,4,5]
```

E' possibile integrare la chiamata ad un certo blocco attraverso il suffisso **&** ed eseguirlo mediante la keyword **call.**

Vediamo un esempio. Come si vede la definizione del metodo comprende un riferimento ad un blocco riconoscibile per il carattere **&**. L'esecuzione di questo riferimento viene attuata tramite il metodo **call**. I parametri di chiamata del metodo restano invariati rispetto a quelli definiti nel blocco. Nella chiamata al metodo il nostro blocco anonimo deve stare sulla stessa riga della funzione, come fosse un normale parametro. Dal punto di vista sintattico il richiamo tramite **&** nella definizione del metodo va messo rigorosamente in coda rispetto a tutti gli altri parametri.

```
def fn(a,b, &x)
   print "#{a}+#{b}="
   x.call(a,b)
end
fn(3,2){|x,y| puts (x+y)}
```

FUNZIONI RICORSIVE

Una funzione è detta ricorsiva quando è in grado di richiamare se stessa. Le funzioni ricorsive sono caratterizzate da questi tre elementi:

- una struttura condizionale che determini il flusso di esecuzione della funzione
- una istruzione che richiama la funzione stessa in uno dei rami della istruzione condizionale
- una istruzione che ritorna un valore ed esce dalla funzione ricorsiva nell'altro ramo della istruzione condizionale

```
def nomeFunzione(argomenti)
  if condizione
     return [caso base]
  end
  [istruzioni]
  nomeFunzione(altriargomenti)
end
```

Esempio 1: il fattoriale

0! = 1
1! = 1
2! = 2*1
3! = 3*2*1
. . .
n! = n*(n-1)*(n-2)*..... *3*2*1

$$Definizione\ iterativa: n! = \prod_{k=1}^{n} k$$

Definizione ricorsiva: $n! = \begin{cases} 1 & se\ \ n = 0; \\ n \cdot (n-1)! & se\ n > 0; \end{cases}$

```
# Definizione Iterativa
def fattoriale(n)
  n_fatt= 1
  1.upto(n) { |i| n_fatt*=i }
  return n_fatt
end

print "Digita n: "
n = gets.to_i
puts "#{n}! = #{fattoriale(n)}"
```

```
def fattoriale(n)
    if n > 1 then
       # n!=n*(n-1)!
       return n * fattoriale(n-1)
    else
       # 0! Caso base
       return 1
    end
end

print "Digita n: "
n = gets.to_i
puts "#{n}! = #{fattoriale(n)}"
```

Esempio 2: somma primi N interi

S(0)= 0
S(1) = 0+1
S(2) = 0+1+2
S(3) = 0+1+2+3
S(4) = 0+1+2+3+4
.....
S(n) = n+(n-1)+(n-2)+ ..... +3+2+1

$$Definizione\ iterativa: S(n) = \sum_{k=1}^{n} k$$

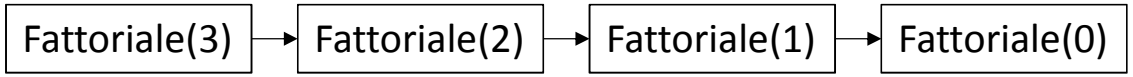
Definizione ricorsiva: $S(n) = \begin{cases} 0 & se\ \ n = 0; \\ n + S(n-1) & se\ n > 0; \end{cases}$

```
def Somma(n)
  n_somma= 0
  1.upto(n) { |i| n_somma+=i }
  return n_somma
end

print "Digita n: "
n = gets.to_i
puts "S(#{n}) = #{Somma(n)}"
```

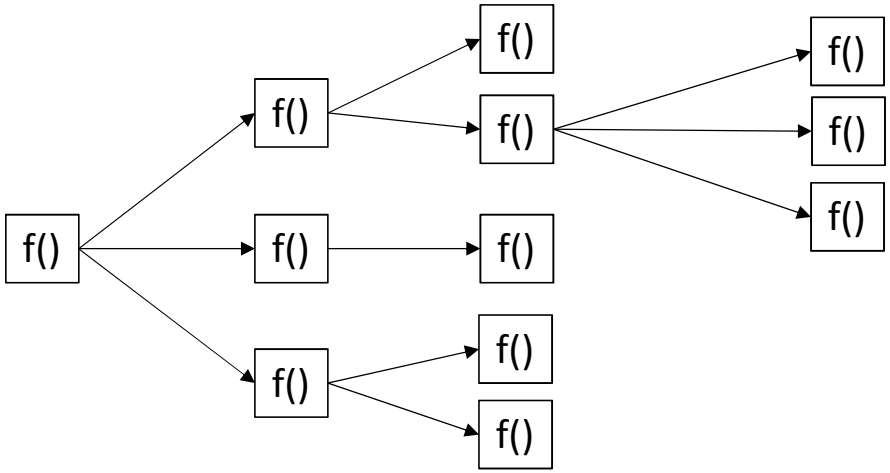
```
def Somma(n)
   if (n>0)
      # S(n)=n+S(n-1)
      return n+Somma(n-1)
   else
      # S(0)=0 caso base
      return 0
   end
end

print "Digita n: "
n = gets.to_i
puts "S(#{n}) = #{Somma(n)}"
```

Ricorsione lineare e non lineare

Si parla di **ricorsione lineare**, quando vi è solo una chiamata ricorsiva all'interno della funzione,



Si parla di **ricorsione non lineare**, quando all'interno della funzione abbiamo più chiamate ricorsive



Esempio: fibonacci

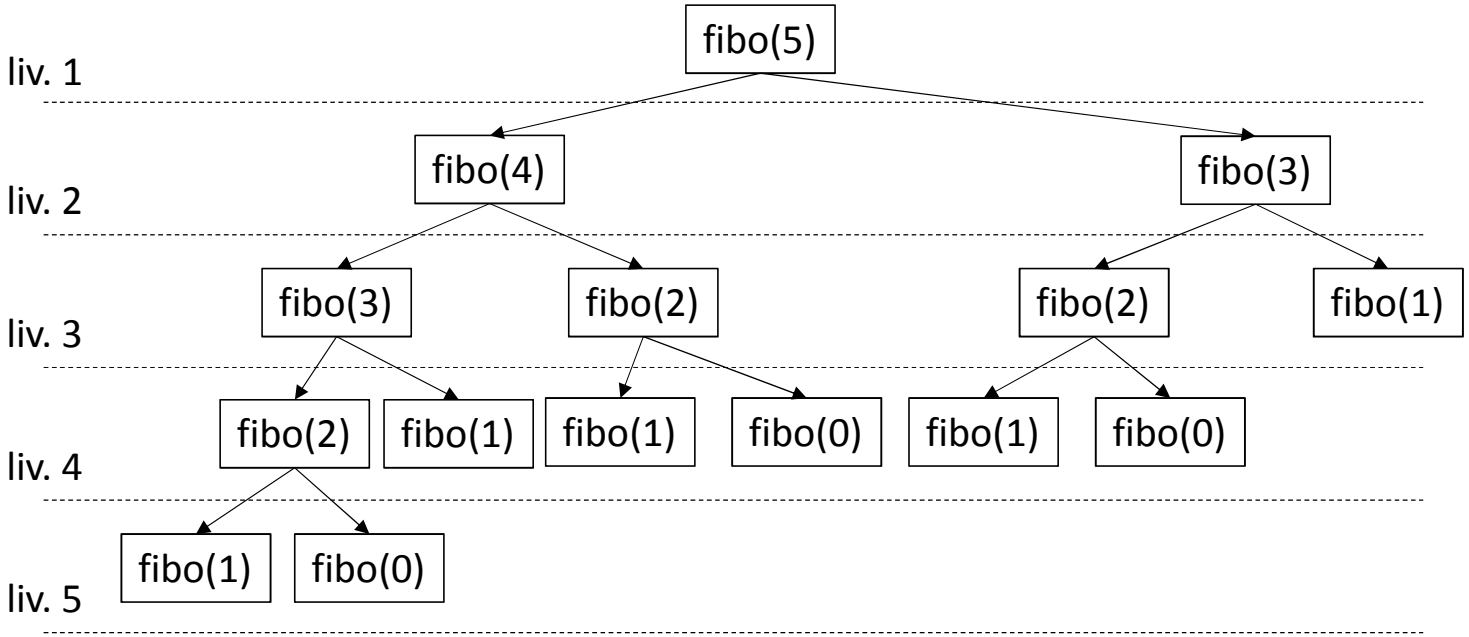
L'implementazione ricorsiva della **serie di Fibonacci** è un esempio di **ricorsione non lineare.** La sequenza è ottenuta considerando come valore dell'elemento corrente x_n la somma dei due termini precedenti $x_{n-1}+x_{n-1}$. Per definizione i primi 2 termini x_1=fibo(1) e x_2=fibo(0) della successione valgono 1

$$fibo(n) = \begin{cases} 1 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ fibo(n-1) + fibo(n-2) & \text{se } n > 1 \end{cases}$$

Quindi **fibo(4)** ricorsivamente viene così sviluppato:

$$\begin{aligned} fibo(4) &= fibo(3) + fibo(2) \\ &= \big(fibo(2) + fibo(1)\big) + \big(fibo(1) + fibo(0)\big) \\ &= \Big(\big(fibo(1) + fibo(0)\big) + 1\Big) + (1 + 1) \\ &= \big((1 + 1) + 1\big) + (1 + 1) = 5 \end{aligned}$$

L'esecuzione dell'algoritmo ricorsivo determina un livello di ricorsione pari a n.



Ad ogni livello abbiamo il raddoppio delle chiamate rispetto al livello precedente.
Il numero totale di chiamate **C** al livello n è quindi

$$C(n) \leq \sum_{i=0}^{n} 2^i = 2^n\text{-}1$$

Si tratta quindi di una crescita esponenziale!

Equivalenza ricorsione/iterazione

Iterazione e **Ricorsione** sono i due approcci algoritmici che derivano dal **concetto di induzione**. Il principio d'induzione è un enunciato sui numeri naturali che in matematica trova un ampio impiego nelle dimostrazioni. Esso asserisce che se una determinata proprietà **P** vale per un determinato valore iniziale $\mathbf{n_0}$ (**passo base**) e se la veridicità di **P(n)** implica la veridicità di **P(n+1)** (**passo induttivo**) allora P(n) vale per qualsiasi numero naturale n >=$\mathbf{n_0}$.

L'idea intuitiva che ci aiuta a comprendere il senso del principio di induzione è quella dell' "effetto domino":
Affinché delle tessere da domino, disposte lungo una fila, cadano tutte sono sufficienti due condizioni:

- che cada la prima tessera
- che le tessere siano posizionate in maniera tale che la caduta di ogni tessera provochi la caduta della tessera successiva

L'**iterazione** si realizza mediante la tecnica del ripetizione (ciclo). La **ricorsione** usa invece subroutine/funzioni che richiamano se stesse.

L'**algoritmo iterativo** è costituito da un *passo iniziale*, un *passo iterato* e una *condizione* che ferma il ciclo . L'**algoritmo ricorsivo** è definito in termini di se stesso: è composto da un passo ricorsivo e da un passo base a cui converge la ricorsione e che rappresenta la condizione di chiusura.

Nel *passo ricorsivo* si definisce una funzione per un generico valore **n** utilizzando la stessa funzione applicata al valore **n-1**. La condizione di chiusura definisce la funzione con n=$\mathbf{n_0}$. Chiamando la funzione per un valore **n** si possono calcolare in cascata tutti i risultati intermedi fino a n=n_0. Il passaggio che chiude la ricorsione si dice *passo base*. I passaggi intermedi passi di ricorsione.

Ad ogni formula iterativa corrisponde sempre una formulazione ricorsiva equivalente. Si consideri il generico ciclo **while**

```
PassoIniziale(condizione)
while condizione do
   passoIterato(condizione)
end
```

Corrisponde alla seguente formulazione ricorsiva

```
def CicloR(Condizione)
   if (Condizione) then
     PassoIterato(Condizione)
     CicloR(Condizione)
   end
end
PassoIniziale(Condizione)
CicloR(Condizione)
```

Esempio:

```
x="E vai ..."
while x!="Basta!"
   print "Digita qualcosa: "
   x=gets.chomp
end
```

=

```
def CicloR(condizione)
   if condizione
      print "Digita qualcosa: "
      x=gets.chomp
      CicloR(x!="Basta!")
   end
end
x="E vai ..."
CicloR(x!="Basta!")
```

Analogamente consideriamo la seguente formulazione ricorsiva

```
def Ricorsione(parametri)
   if (CondizioneDiStop(parametri))
      return PassoBase
   else
      return Operazione(PassoRicorsivo(parametri),Ricorsione(parametri))
   end
end
risultato=Ricorsione(ValoreIniziale)
```

Corrisponde alla seguente formulazione iterativa

```
Parametri=ValoreIniziale
Risultato=PassoBase
While Not CondizioneDiStop(parametri)
      Risultato=Operazione(Risultato,PassoRicorsivo(parametri))
Wend
```

Un definizione ricorsiva è ben posta se:

- **non-ambigua**: ogni argomento genera una sola risposta
- **non-circolare**: la funzione non usa mai se stessa con gli stessi argomenti
- **finita**: la successione delle chiamate termina sempre in un caso base

L'algoritmo ricorsivo richiama se stesso generando una sequenza di chiamate che ha termine al verificarsi di una condizione particolare che viene chiamata **condizione di terminazione**.

La tecnica ricorsiva permette di scrivere algoritmi eleganti e sintetici per molti tipi di problemi comuni, anche se non sempre le soluzioni ricorsive sono le più efficienti. Questo è dovuto al fatto che comunemente la ricorsione viene implementata utilizzando le funzioni, e l'invocazione di una funzione ha un costo rilevante in termini di tempo e questo rende meno efficienti gli algoritmi ricorsivi rispetto alla corrispondente soluzione iterativa.

Esercizi RICORSIONE

A. Leggere una sequenza di numeri terminata con 0 e stamparla in modo rovesciato
B. Scrivere una procedura ricorsiva che continua a chiedermi un numero fino a quando non digito zero

```
...
```

Soluzione A

...

```
. . .
```

Le classi

Ruby è un linguaggio di programmazione puramente orientato agli oggetti. Il paradigma di **programmazione orientato agli oggetti** è detto comunemente **OOP**.

La **OOP** prevede di raggruppare in una zona circoscritta del codice sorgente (chiamata **classe**) la dichiarazione delle strutture dati e delle procedure che operano su di esse. Le classi, quindi, costituiscono dei modelli astratti, che durante l'esecuzione vengono invocate per istanziare o creare **oggetti** software relativi alla classe invocata. Questi ultimi sono dotati di **attributi** (dati che definiscono le caratteristiche o proprietà degli oggetti instanziabili) e **metodi** (procedure che operano sugli attributi) secondo quanto definito/dichiarato nelle rispettive classi.

Un **oggetto** è una istanza di una classe. Esso è dotato di tutti gli attributi e i metodi definiti dalla classe, ed agisce come un fornitore di "*messaggi*" (i metodi) che il codice eseguibile del programma può attivare su richiesta. Inviare un messaggio ad un oggetto si dice, in gergo, *invocare un metodo* su quell'oggetto. Il metodo riceve come parametro (in modo implicito) l'oggetto stesso su cui è stato invocato e questo può essere referenziato tramite una parola-chiave (in Ruby si usa la parola-chiave `self`).

Dal punto di vista del calcolatore, ogni oggetto è identificato da una certa zona di memoria, nella quale sono memorizzati gli attributi. Il valore di questi ultimi determina lo *stato interno dell'oggetto*. Istanziare un oggetto vuol dire allocare la memoria necessaria ed eventualmente inizializzarla secondo le specifiche definite dalla classe. Molti linguaggi forniscono un supporto per l'inizializzazione automatica di un oggetto, con uno o più metodi speciali, detti **costruttori**. Analogamente, la fine della vita di un oggetto può essere gestita con un metodo detto **distruttore**.

Si definisce **interfaccia di una classe** l'insieme dei dati e dei metodi visibili all'esterno degli oggetti che sono istanze di quella classe. Secondo il principio noto come *information hiding*, l'accesso ai campi di un oggetto deve essere permesso solo tramite metodi invocati su quello stesso oggetto (definiti dall'interfaccia della classe). Il vantaggio principale è che il controllo completo sullo stato interno viene assegnato ad una zona ristretta del codice eseguibile del programma (la classe appunto) poiché il codice esterno non è autorizzato a modificarlo.

La parte del programma che fa uso di un oggetto si chiama *client.* Un linguaggio di programmazione è definito ad oggetti quando permette di implementare tre meccanismi usando la sintassi nativa del linguaggio:

- **Incapsulamento**: consiste nella *separazione* della cosiddetta *interfaccia* di una classe dalla corrispondente *implementazione*. I client di un oggetto di quella classe conoscono solo l'interfaccia ma non la sua implementazione (*separazione interfaccia e implementazione*). Con l'incapsulamento gli attributi che definiscono lo stato interno di un oggetto e i metodi che ne definiscono la logica sono accessibili solo ai metodi dell'oggetto stesso, mentre non sono visibili ai client. Per alterare lo stato interno dell'oggetto, è necessario invocare i metodi pubblici, ed è questo lo scopo principale dell'incapsulamento. L'incapsulamento permette di vedere l'oggetto come una black-box di cui, attraverso l'interfaccia, *è noto solo cosa fa, ma non come lo fa*.
- **Ereditarietà** : permette essenzialmente di definire delle classi a partire da altre già definite. Una classe derivata attraverso l'ereditarietà (*sottoclasse*), mantiene i metodi e gli attributi delle classi da cui deriva (classe base, *superclasse* o classe padre). Inoltre, può aggiungere dei propri metodi o attributi e ridefinire il codice di alcuni dei metodi ereditati tramite un meccanismo chiamato **overriding**. Quando una classe eredita da una sola superclasse si parla di **eredità singola**; viceversa, si parla di **eredità multipla**. L'ereditarietà può essere usata come meccanismo per ottenere l'estensibilità e il riuso del codice, e risulta particolarmente vantaggiosa quando viene usata per definire sottotipi, sfruttando le relazioni esistenti nella realtà di cui la struttura delle classi è una modellizzazione.

Esempio: *se nel programma esiste una classe MezzoDiTrasporto che ha come attributi i dati: posizione, velocità, destinazione e carico utile. Possiamo definire una nuova classe Aereo direttamente dall'oggetto MezzoDiTrasporto dichiarando che la classe di tipo Aereo è figlia di MezzoDiTrasporto. Possiamo aggiungere nuovi attributi che identificano ad esempio la quota di crociera. Il vantaggio è che la nuova classe acquisirà tutti i membri definiti in MezzoDiTrasporto per il fatto stesso di esserne sottoclasse.*

- **Polimorfismo**: permette ad un client di attivare comportamenti diversi senza conoscere a priori il tipo specifico dell'oggetto che gli viene passato. (classi diverse possono condividere la stessa interfaccia – Esempio **Fixnum** e **Bignum**).
  Esempio: *per ritornare all'esempio precedente, si suppone che un "Aereo" debba affrontare procedure per l'arrivo e la partenza molto più complesse di quelle di un normale camion, come in effetti è: allora le procedure arrivo() e partenza() devono essere cambiate rispetto a quelle della classe base "MezzoDiTrasporto". Si provvede quindi a ridefinirle nella classe "Aereo" in modo da ottenere il comportamento necessario (polimorfismo). A questo punto, dalla lista di mezzi è possibile prendere qualsiasi mezzo e chiamare arrivo() o partenza() senza più doversi preoccupare di che cos'è l'oggetto che si sta maneggiando: che sia un mezzo normale o un aereo, si comporterà rispondendo alla stessa chiamata sempre nel modo giusto.*

Ruby è un puramente OOP perchè tutte le rappresentazioni interne di dati sono oggetti.

La classe RADIO

Supponiamo ora di applicare i concetti della OOP all'oggetto generico RADIO. Tale astrazione d'ora in avanti la chiameremo ***classe delle Radio***. Pensando all'insieme di tutte le radio possiamo definire come ***attributi*** le seguenti caratteristiche:

- *il colore*
- *la dimensione*
- *il peso*
- *le bande di ricezione disponibili*
- *la lunghezza della antenna*
- *etc.*

Si considerino attributi anche lo stato attuale della radio generica come:

- *il valore del volume*
- *la frequenza di sintonia*
- *accesa/spenta*
- *etc.*

Tutte le operazioni che ci permettono di modificare lo stato di una radio, come ad esempio:

- alzare/abbassare il volume
- modificare la sintonia
- accendere/spegnere la radio
- etc.

possono essere definiti come **metodi** della nostra classe.

L'interfaccia è rappresentata dagli strumenti che ci permettono di agire effettivamente agire sui metodi (manopole, interruttori, etc.).

Pensiamo ad una radio in particolare con la sua dimensione, colore, etc. Questa radio è a tutti gli effetti *una istanza della classe Radio* (oggetto). A differenza della classe, che è una astrazione che definisce unicamente come sono fatte in generale le Radio, l'istanza ha stato ed attributi ben definiti, e quindi è unica.

Poche persone aprono una radio funzionante per vedere che cosa c'è dentro, anche se tutti si aspettano di trovarsi una qualche forma di circuito elettrico che definisce la sintonia, un demodulatore che elimina la portante per mantenere solo il segnale (detto intelligence), un amplificatore che amplifica il segnale di intelligence, etc.

Più o meno sappiamo della presenza di tutte queste cose in una radio, ma difficilmente conosciamo nei minimi particolari lo schema effettivo dei circuiti, che ci spiega come sono implementate le funzioni di:

- ricezione
- demodulazione
- amplificazione
- etc.

Anche questi sono metodi della classe, ma sono **metodi privati** ovvero metodi che sono all'interno ma non utilizzabili da fuori, se non facendo uso dei metodi definiti in precedenza e che permettono in qualche modo di accedere a tali metodi privati (o volendo anche attributi privati). Abbiamo appena spiegato la proprietà di incapsulamento. All'utente finale (detto client, chi ascolta la radio) non interessa la effettiva implementazione dei singoli metodi o conoscere eventuali attributi interni (circuiti, colore dei fili interni, etc.). Per il client sono sufficienti l'interfaccia (manopole, interruttori, etc.) e gli attributi esposti (valore del volume da 0 a 100, indicatore di frequenza di sintonia, etc.).

Immaginiamo ora una nuova classe, simile alla classe Radio (con la quale condivide gli attributi, i metodi e interfaccia) ma che implementa qualcosina in più, come la visualizzazione dell'ora e la funzionalità di sveglia. Oltre agli attributi precedenti dobbiamo quindi aggiungere:

- attributi del display
- ora attuale
- ora di sveglia
- stato della sveglia
- etc.

Anche per i metodi dobbiamo definirne dei nuovi:

- imposta ora
- imposta sveglia
- attiva/disattiva sveglia
- etc.

Tralasciando i metodi e gli attributi privati necessari alla implementazione effettiva della sveglia. Anche l'interfaccia differisce, avendo aumentato il numero di *metodi pubblici* messi a disposizione (bottoni per settare l'ora, interruttore per attivare/disattivare la sveglia, il pulsante di snooze).

Quindi la classe Radio-Sveglie implementa attributi e metodi della classe Radio. In questo caso, la OOP permette di fare uso della proprietà di ereditarietà, secondo la quale, piuttosto che re-implementare nuovamente da zero i metodi, questi possono essere ereditati dalla classe Radio. La nuova classe deve solo definire i metodi e gli attributi che non sono definiti nella classe dallla quale ha ereditato. In questo caso la classe Radio-Sveglie è detta sotto-classe, mentre la classe Radio è la super-classe.

Il **polimorfismo** per inclusione è legato alle relazioni di eredità tra classi e garantisce che tali oggetti, pur di tipo differente, abbiano una stessa interfaccia. Nei linguaggi ad oggetti tipizzati, le istanze di una sottoclasse possono essere utilizzate al posto di istanze della superclasse.

L'**overriding** dei metodi o delle proprietà permette che gli oggetti appartenenti alle sottoclassi di una stessa classe rispondano diversamente agli stessi utilizzi. Ad esempio, supponiamo di avere una gerarchia in cui le classi *Cane* e *Gatto* discendono dalla superclasse *Animale*. Quest'ultima definisce un metodo *cosaMangia(),* le cui specifiche sono: "*restituisce una stringa che identifica il nome dell'alimento tipico dell'animale*". I due metodi *cosaMangia()* definiti nelle classi Cane e Gatto si sostituiscono a quello che ereditano da Animale e, rispettivamente, restituiscono "*osso*" e "*pesce*".

Supponiamo di scrivere un client che lavora con oggetti di tipo Animale; ad esempio, una banale funzione che prende in ingresso un oggetto di tipo Animale, ne invoca il metodo *cosaMangia()* e stampa a video il risultato. Questa funzione otterrà due risultati diversi a seconda del tipo effettivo dell'oggetto che le viene passato come argomento. Il comportamento di un programma abbastanza complesso, quindi, può essere alterato considerevolmente in funzione delle sottoclassi che sono istanziate durante l'esecuzione e le cui istanze sono passate alle varie parti del codice.

I metodi che vengono ridefiniti in una sottoclasse sono detti polimorfi, in quanto lo stesso metodo si comporta diversamente a seconda del tipo di oggetto su cui è invocato.

Nei linguaggi in cui le variabili non hanno tipo, come Ruby, è possibile estendere le possibilità del polimorfismo oltre le relazioni di ereditarietà: nell'esempio di prima, non è necessario che le classi Cane e Gatto siano sottoclassi di Animale, perché ai client interessa solo che i tre tipi espongano uno stesso metodo con il nome cosaMangia con la stessa lista di argomenti.

Le classi in Ruby

Nel linguaggio Ruby le classi sono definibili come istanza della classe **`Class`**. Nel momento in cui si definisce una nuova classe, utilizzando il costrutto "**`class Nome_classe ... end`**", viene definito un oggetto di tipo "**`Class`**" al quale viene assegnato come valore la costante "**`Nome_classe`**".

```
class Amici
   ...
end
```

L'invocazione "**`Nome_classe.new`**" viene utilizzata per la creazione di un nuovo oggetto; quando ciò accade viene eseguito il metodo per la creazione della nuova istanza della classe che riserva una parte di memoria per l'oggetto appena creato e successivamente viene chiamato il metodo per l'inizializzazione dell'oggetto.

Le due fasi descritte ("**costruzione**" e "**inizializzazione**" dell'oggetto) avvengono separatamente e possono essere riscritte (**overridden**). Quella relativa all'inizializzazione è determinata dal metodo **`initialize`** che quindi non funge da costruttore, compito che spetta invece al metodo **`new`** della classe.

Le classi consentono di creare blocchi di istruzioni, anche molto lunghi, che possono essere salvati in un file. Tali classi possono essere richiamate quando serve tramite una semplice inclusione. Per effettuare l'inclusione di un file esterno in Ruby si utilizza l'istruzione **`require`** a cui viene passato come argomento il nome del file senza l'estensione ".rb".

Implementazione ed utilizzo della classe all'interno dello stesso file **".rb"**

```ruby
class Friends

   # inizializzazione
   def initialize(nome1, nome2)
      # variabili di istanza della classe
      @nome1 = nome1
      @nome2 = nome2
   end

   # definizione di un primo metodo
   def domanda
     puts "Ciao #{@nome1}, hai visto #{@nome2}?"
   end

   # definizione di un secondo metodo
   def risposta
     puts 'No!'
   end

end

# costruzione dell'oggetto
d = Friends.new('Carla', 'Luca')

# utilizzo dei metodi
d.domanda  # stampa "Ciao Carla, hai visto Luca?"
d.risposta # stampa "No!"
```

Implementazione della classe nel file **"_amici.rb"**

```ruby
class Friends
   # inizializzazione
   def initialize(nome1, nome2)
      # variabili di istanza della classe
      @nome1 = nome1
      @nome2 = nome2
   end
   # definizione di un primo metodo
   def domanda
     puts "Ciao #{@nome1}, hai visto #{@nome2}?"
   end
   # definizione di un secondo metodo
   def risposta
     puts 'No!'
   end
end
```

=

```ruby
require '.\_amici'

# costruzione dell'oggetto
d = Friends.new('Carla', 'Luca')

# utilizzo dei metodi
d.domanda  # stampa "Ciao Carla, hai visto Luca?"
d.risposta # stampa "No!"
```

Analizzando il codice della precedente classe evidenziamo:

```
class Friends
   # inizializzazione
   def initialize(nome1, nome2)
      # variabili di istanza della classe
      @nome1 = nome1
      @nome2 = nome2
   end
   # definizione di un primo metodo
   def domanda
     puts "Ciao #{@nome1}, hai visto #{@nome2}?"
   end
   # definizione di un secondo metodo
   def risposta
     puts 'No!'
   end
end
```

@nome1 e **@nome2** sono delle *variabili d'istanza* e sono visibili da tutti i metodi della classe. Sono detti anche **attributi**. Per poter accedere (in lettura) a tali campi occorre creare dei metodi accessori del tipo **get** (getter):

```
def nome1
   @nome1
end
```

```
def nome2
   @nome2
end
```

Per poter accedere agli attributi anche in scrittura occorre definire anche in questo caso un apposito metodo di tipo **set**:

```
def nome1=(nome)
   @nome1 = nome
end
```

```
def nome2=(nome)
   @nome2 = nome
end
```

Siamo in grado di leggere o impostare i valori degli attributi in questo modo:

```
puts d.nome1
d.nome2="Genoveffa"
```

Chiaramente se voglio leggere non posso usare il metodo di tipo set e pertanto devo definire lo specifico metodo di tipo get.

Le funzioni **setter** e **getter** si potevano scrivere anche nella seguente forma più compatta:

```
def nome; @nome; end
def cognome; @cognome; end
def nome=(value); @nome=value; end
def cognome=(value); @cognome=value; end
```

La gestione degli attributi è molto elementare e allo stesso tempo molto tediosa. Essendo però una pratica di uso comune, il Ruby fornisce delle comode scorciatoie per la gestione degli attributi. Utilizzando `attr`. Il nostro codice diventa:

```
class Friends
   attr :nome1, false # è a sola lettura
   attr :nome2, true  # è modificabile

   # inizializzazione
   def initialize(nome1, nome2)
      # variabili di istanza della classe
      @nome1, @nome2 = nome1, nome2
   end
end
```

```
require '.\_amici'

# costruzione dell'oggetto
d = Friends.new('Carla', 'Luca')

# da errore: non è setter
# d.nome1="Genoveffa"
d.nome2="Genoveffa"

puts d.nome1
puts d.nome2
```

Il valore **`true`** dopo il nome dell'attributo sta ad indicare che la variabile è anche scrivibile. in alternativa avremmo potuto usare **`attr_reader`** (getter) e **`attr_writer`** (setter) oppure **`attr_accessor`** (entrambe) che è sinonimo di "**`attr nome_simbolo, true`**"

Implementazione della classe nel file "**_amici.rb**"

```
class Friends
   attr_reader :nome1   # lettura
   attr_writer :nome2   # scrittura
   attr_accessor :nome3 # lettura/scrittura
   # inizializzazione
   def initialize(nome1, nome2, nome3)
      # variabili di istanza della classe
      @nome1 = nome1
      @nome2 = nome2
      @nome3 = nome3
   end
end
```

```
require '.\_amici'
# costruzione dell'oggetto
d = Friends.new('Carla', 'Luca', "Elena")
# da errore: non è setter
# d.nome1="Genoveffa"
d.nome2="Genoveffa"
d.nome3="Genoveffa"
puts d.nome1
# da errore: non è un getter
# puts d.nome2
puts d.nome3
```

Prima di proseguire dobbiamo dare uno sguardo ad un importante concetto espresso da una semplice keyword: **`self`**. Detto brevemente, **`self`** si riferisce all'oggetto stesso, un po' come una persona può riferirsi a se con "*me stesso*".

Implementazione della classe **Successione** nel file "**_successione.rb**"

```
class Successione
   def initialize(x=0)
      @x=x
   end
   def incr
      @x=@x+1
   end
   def stampa
      print "#@x\n"
   end
end
```

Implementazione della classe **Sequenza** nel file "**_sequenza.rb**"

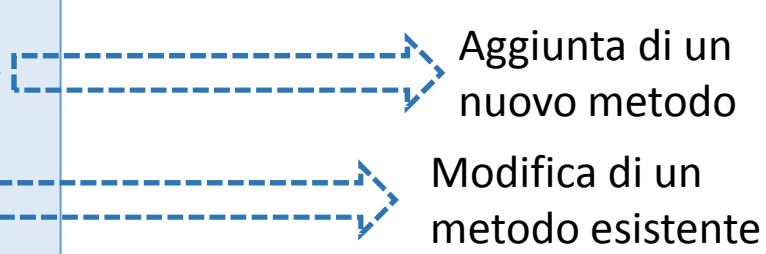
```
class Sequenza
   attr:x, true #devo dichiararla
   def initialize(x=0)
      self.x=x
   end
   def incr
      self.x=self.x+1
   end
   def stampa
      print "#{self.x}\n"
   end
end
```

Implementazione del main

```
require '.\_successione'
require '.\_sequenza'

# costruzione dell'oggetto
x=Successione.new
x.incr
x.stampa

# costruzione dell'oggetto
y=Sequenza.new(3)
y.incr
y.stampa
```

Una caratteristica interessante della classi in Ruby è che essere possono essere riaperte per aggiungere o modificare delle funzionalità.

```ruby
class Uno
   def one
      puts "one"
   end
end
u = Uno.new
u.one # one
class Uno
   def ein
      puts("ein")
   end
   def one
      puts("1")
   end
end
e = Uno.new
u.one # 1
e.one # 1
e.ein # ein
```



All'interno delle classi possiamo trovare delle costanti. Ricordiamoci l'uso della maiuscola con la quale deve iniziare il nome di una costante. Per accedere ad una costanti dall'esterno basta scrivere **`NomeClasse::NomeCostante`**.

Le **variabili di classe** consentono di conservare informazioni rilevanti per la classe e per gli oggetti da questa derivanti. In questo senso esse sono private (non sono quindi visibili all'esterno) ma disponibili per la classe e le sue istanze. A caratterizzare questa tipologia di variabili è il loro prefisso che è costituito da un coppia di @@. Quindi per definire la variabile di classe nristanze dovremo scrivere:

```
class Punto
   @@nristanze = 0
   def initialize
      @@nristanze += 1
   end
   def istanze # getter
      @@nristanze
   end
   def Punto.istanze=(valore) # setter
      @@istanze = valore
   end
end

p1 = Punto.new()
puts(p1.istanze) # 1
p2 = Punto.new()
puts(p2.istanze) # 2
puts(p1.istanze) # 2
```

Variabile di classe che conteggia il numero di oggetti di quella classe che sono stati istanziati

Incrementa la variabile di classe ad ogni inizializzazione

Torniamo ora parlare dei metodi e della loro visibilità. Esistono tre keywords in Ruby che modificano le possibilità di accesso ai metodi e quindi definiscono il perimetro del loro utilizzo; tali parole chiave sono **`public`**, **`private`** e **`protected`**. Chi ha già esperienza di programmazione object oriented troverà familiari questi modificatori.

La prima parola, **`public`**, è la più semplice; esso significa che il metodo può essere invocato liberamente senza alcuna restrizione. Ogni metodo di una classe *è pubblico di default*. Fa eccezione ovviamente **`initialize`** che *è private di default*. Un metodo **`private`** è invece riservato alla implementazione della classe stessa. Può essere richiamato da un altro metodo interno ma non può essere richiamato tramite un oggetto ***obj***. In breve se ***metodo1*** è un metodo privato questo può essere richiamato all'interno della classe come ***metodo1*** ma non come ***obj.metodo1*** e nemmeno nel formato ***self.metodo1***. Questo non vale per i **`public`** e nemmeno per i **`protected`**.
I metodi marcati come **`protected`** possono essere invocati dagli oggetti creati a partire dalla classe in cui sono definiti e dalle sue sottoclassi e dagli oggetti da queste derivati.

```
class Test
private
   ... # sezione privata
protected
   ... # sezione privata
public
   ... # sezione pubblica
end
```

Un altro punto da segnalare è un metodo alternativo alla creazione di "sezioni" dedicate ai metodi pubblici, privati e protetti. Tale sistema prevede l'etichettatura successiva dei metodi in questo modo

```
class Test
   ... # definizione metodi
End
public    :metodo1
protected :metodo4, :metodo2
private   :metodo2
```

Quando si parla di classi occorre parlare di concetti come ereditarietà, polimorfismo etc...

Ereditarietà: si tratta di un concetto che lega tra di loro due classi, una di esse sarà identificata come ***superclasse***, in soldoni si potrebbe dire quella creata per prima e l'altra è definita come ***sottoclasse*** in quanto deriva tutta o in parte le sue caratteristiche dalla superclasse (se volete potete chiamarle madre e figlia, o padre e figlio, o anche classe base e classe derivata). Dal punto di vista pratico, si dice anche una sottoclasse estende la superclasse o le superclassi da cui deriva. Vedremo come questo abbia effettivamente un senso. L'ereditarietà può essere singola o multipla; nel primo caso una sottoclasse può discendere da una e una sola superclasse, nel secondo caso può avere più progenitori. Si tratta di una differenza sostanziale ai fini pratici. Ruby, come la maggior parte dei linguaggi, adotta il primo approccio. Ruby lavora adottando l'ereditarietà singola e questo è quanto. Tramite l'ereditarietà è possibile creare complesse gerarchie con molti livelli di classi. In particolare, se è vero che una sottoclasse può avere una sola superclasse è però altrettanto vero che ogni classe può avere quante sottoclassi si vogliono da essa derivanti. Quindi le strutture gerarchizzate saranno per lo più simile ad alberi a testa in giù.

Ma cosa vuol dire in pratica "ereditare"? In breve, se una certa classe ha determinate proprietà una sottoclasse da essa derivante le possiede anch'essa senza bisogno di ridefinirle. Questo non impedisce che la sottoclasse possa avere proprietà sue che non sono presenti nella superclasse e che possa ridefinire ciò che eredita dalla superclasse. A questo punto è arrivato il momento di dare uno sguardo ad un po' di codice.

Si osservi quel **<** che altro non è che l'operatore adottato in Ruby per innescare l'ereditarietà. Il metodo **hi**, definito nella classe padre risulta disponibile per eredità nelle istanze della classe figlia **DueSaluti**.

```
class UnSaluto
   def hi
      puts "Hi"
   end
end

class DueSaluti < UnSaluto
   def ciao
      puts "ciao"
   end
end

s1 = UnSaluto.new
s2 = DueSaluti.new
s1.hi
s2.hi
s2.ciao
```

Nel caso non fosse noto quale sia la superclasse di una sottoclasse, Ruby mette a disposizione il metodo **superclass** che ci permette di risalire di un gradino nella gerarchia delle classi.

```
class UnSaluto
   def hi
      puts "Hi"
   end
end
class DueSaluti < UnSaluto
   def ciao
      puts "Ciao"
   end
end
class TreSaluti < DueSaluti
   def hola
      puts "Hola"
   end
end

puts(TreSaluti.superclass) # DueSaluti
puts(TreSaluti.superclass.superclass) # UnSaluto
puts(TreSaluti.superclass.superclass.superclass) # Object
puts(TreSaluti.superclass.superclass.superclass.superclass) # BasicObject
puts(TreSaluti.ancestors.to_s) # Array: [TreSaluti, DueSaluti, UnSaluto,
                                        Object, Kernel, BasicObject]
puts(TreSaluti.class.ancestors.to_s) # Array: [Class, Module, Object, Kernel
                                                BasicObject]
puts(UnSaluto.superclass) # Object
puts(UnSaluto.superclass.superclass) # BasicObject
puts(UnSaluto.superclass.superclass.superclass) # nil
```

L'utilizzo in cascata del metodo **superclass** consente di evidenziare la gerarchia di una classe. La classe **TreSaluti** è discendente da una generica classe **Object** che risulta sottoclasse di **BasicObject**: la root del sistema di classi di Ruby. Non esiste progenitore di **BasicObject**. Sopra questa radice c'è il nulla (**nil**). Il metodo **ancestors** fornisce un array con tutta la gerarchia delle classi. Si noti che **TreSaluti.class** restituisce **class** essendo **TreSaluti** un'istanza della classe **class**.

Questione diversa è quella dell'**overriding**. In altre parola, possiamo ereditare un metodo o una proprietà da una superclasse ma potremmo anche volerne modificare in qualche modo il funzionamento. Il procedimento è molto semplice e consiste nella semplice riscrittura del metodo con le modifiche che vogliamo applicare.
L'esempio che segue è inerente a questa tecnica:

```
class Uno
   def one; puts "1"; end
end
class UnoDue < Uno
   def two; puts "2"; end
end
class UnoDueTre < UnoDue
   alias due :two
   def two; puts "2Bis"; end
end
s1 = UnoDueTre.new
s1.due # 2
s1.two # 2Bis
```

In questo modo potete continuare ad utilizzare quanto fatto nel metodo **two** della classe **UnoDue**.

```
class UnSaluto
   def hi; puts "Hi"; end
end
class DueSaluti < UnSaluto
   def ciao; puts "Ciao"; end
end
class TreSaluti < DueSaluti
   def ciao; puts "Ciao - 2"; end
end
s2 = DueSaluti.new
s3 = TreSaluti.new
s2.ciao # Ciao
s3.ciao # Ciao - 2
```

Ruby dimostra una certa elasticità. Supponiamo ad esempio di dover effettuare l'**override** di un metodo di una superclasse ma vorremmo anche conservare il lavoro fatto in quest'ultimo per poterlo utilizzare senza doverlo riscrivere. Come si può fare? Certo, attraverso la keyword **`alias`**, che usa la sintassi:

```
alias nomenuovo :nomemetodosuper
```

La classe **Numeric**

http://ruby-doc.org/core-2.3.1/Numeric.html

La classe **Numeric** è una classe di base che definisce in forma generale i metodi e gli attributi che avranno le sottoclassi come **Fixnum**, **Bignum**, e **Float**. È all'interno di questa classe che sono definiti sia gli operatori di Comparazione che moltissimi altri metodi utili ad effettuare la manipolazione di oggetti numerici.
Per accedere ai metodi della classe di un oggetto, è necessario utilizzare la cosidetta *dot-notation*.
Un metodo molto importante, che non fa parte realmente della classe **Numeric**, ma della classe **Object**, è il metodo `num.to_s` , che trasforma il contenuto di `num` in una stringa (sequenza di caratteri).

Sottoclasse FIXNUM

http://ruby-doc.org/core-2.3.1/Fixnum.html

La sottoclasse principe di *Numeric* è la classe **Fixnum**, che definisce i numeri interi. In generale la definizione di un numero intero si ottiene mediante l'assegnazione ad una variabile di un numero intero (*dove non viene indicato il separatore decimale che rappresentato dal punto*).

```
num=100
```

Il range dei numeri **Fixnum** si estende fino ad una determinata dimensione, definita dalla configurazione dell'interprete. Quando si sfora questo range, automaticamente l'interprete trasforma il **Fixnum** in un **Bignum**, che dal punto di vista del tempo di esecuzione del codice è molto meno efficiente. A livello di interfaccia (polimorfismo) le due classi sono del tutto uguali.

Definita la variabile, `num` , questa ha la possibilità di utilizzare i metodi tipici definiti nella classe *Fixnum* e nella classe *Numeric*. Per poter accedere ai metodi, ancora una volta faremo uso della *dot-notation*.

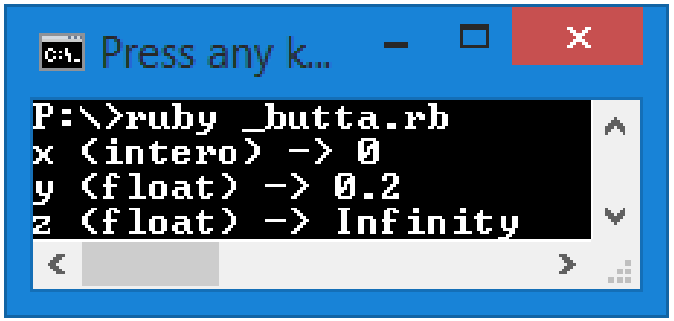
Sottoclasse FLOAT

Un numero reale si definisce per mezzo della assegnazione ad una variabile di un numero con valore decimale o in notazione scientifica. Quella variabile diventa una istanza della classe **Float**, e risponde a tutte le regole della aritmetica tra numeri reali.

```
num = 1234.56       # Numero reale
num_ns = 1.23456e3 # Notaz. Scientifica – e3 indica 10**3
```

Il range della classe **Float** si estende da **−∞** a **+∞**. Infatti nel caso di una divisione per 0, il risultato è ∞. Nella classe Fixnum invece una divisione per 0 genera un errore. Ruby rappresenta l'infinito mediante una costante: **Infinity**. Qualche volta può capitare che operazioni portino a risultati non rappresentabili o che possano essere considerati come non numeri. In questo caso il risultato viene rappresentato con un **NaN** (Not a Number). Vediamo nel pratico questo cosa significa, soprattutto nell'utilizzo della divisione:

```
# Aritmetica degli interi
a_int = 12
b_int = 60
x_int = a_int / b_int

# Aritmetica dei float
a_flt = 12.0
b_flt = 60.0
d_flt = 0.0
y_flt = a_flt / b_flt
z_flt = a_flt / d_flt

puts "x (intero) -> #{x_int}"
puts "y (float) -> #{y_flt}"
puts "z (float) -> #{z_flt}"
```



Nella divisione tra interi restituisce come risultato ancora un valore intero

Metodi della classe **Numeric**

num.ceil

Ritorna il numero intero più piccolo possibile che risulti maggiore o uguale del numero **num**.

```
puts 12.1.ceil  # ==> 13
puts 12.9.ceil  # ==> 13
puts 12.ceil    # ==> 12
puts -12.1.ceil # ==> -12
puts -12.9.ceil # ==> -12
puts -12.ceil   # ==> -12
```

num.floor

Ritorna il numero intero più grande possibile che risulti minore o uguale del numero **num**.

```
puts 12.1.floor  # ==> 12
puts 12.9.floor  # ==> 12
puts 12.floor    # ==> 12
puts -12.1.floor # ==> -13
puts -12.9.floor # ==> -13
puts -12.floor   # ==> -12
```

num.abs – num.magnitude

Restituisce il valore senza segno di **num**.

```
puts -12.3.abs        # ==> 12.3
puts -12.magnitude    # ==> 12
```

num.nonzero?

Restituisce **num** (se stesso **self**) se il valore è diverso da zero altrimenti **nil**.

```
puts 0.nonzero? ==nil  # ==> true (poiché vale nil!)
puts 10.nonzero?       # ==> 10
```

num1.divmod(num2) ➔ [num1.div(num2), num1.modulo(num2)]

Restituisce un vettore contenente il quoziente e il resto della divisione tra **num1** e **num2**. In altre parole restituisce **[num1/num2 , num1 % num2]**

```
num1, num2 = 17, 7
puts num1.divmod(num2).to_s         # ==> [2, 3]
puts [num1/num2, num1 % num2].to_s # ==> [2, 3] - num1/num2 è ancora un intero
puts [num1.div(num2), num1.modulo(num2)].to_s # ==> [2, 3]
```

num.integer?

Restituisce true se il valore **num** è un intero.

```
puts 1.integer?      # ==> true
puts (1.0).integer?  # ==> false
```

num.round(ndigit)

Restituisce il numero **num** arrotondato a **ndigit** cifre decimali.

```
puts 5191.25943.round(-4) # ==> 10000
puts 5191.25943.round(-3) # ==> 5000
puts 5191.25943.round(-2) # ==> 5200
puts 5191.25943.round(-1) # ==> 5190
puts 5191.25943.round(0)  # ==> 5191
puts 5191.25943.round(1) # ==> 5191.3
puts 5191.25943.round(2) # ==> 5191.26
puts 5191.25943.round(3) # ==> 5191.259
puts 5191.25943.round(4) # ==> 5191.2594
```

num.class

Restituisce il nome della classe di appartenenza.

```
puts "1 => #{1.class}"                    # ==> 1 => Fixnum
puts "100**100 => #{(100**100).class}" # ==> 100**100 => Bignum
puts "1.0 =>  #{1.0.class}"               # ==> 1.0 => Float
```

Metodi della classe **Fixnum**

int.succ – int.next

Ritorna il numero intero successivo a **int**.

```
puts 1.next       # ==> 2
puts 1.succ.succ  # ==> 3
```

int.even – int.odd

Ritornano **true** se **int** è rispettivamente pari/dispari.

```
puts 1.odd?   # ==> true
puts 1.even?  # ==> false
```

int.to_s - int.to_s(base)

Restituisce la stringa corrispondente all'intero **int** convertito nella **base** indicata. Se la **base** non è indicata usa come base 10.

```
puts 255.to_s(2)  # ==> 11111111
puts 255.to_s(8)  # ==> 377
puts 255.to_s(16) # ==> ff
```

int.to_f

Trasforma l'intero `int` in **Float**.

```
puts 1.to_f    # ==> 1.0
```

int.size

Restituisce la dimensione in byte del numero `int`.

```
puts 1.size    # ==> 4
```

int.bit_length

Restituisce il numero di bit della rappresentazione binaria di `int`.

```
puts 126.bit_length    # ==> 7
```

int+num – int-num – int*num – int/num – int**num

Operatori algebrici. Il tipo restituito dipende dal numerico `num`.

```
puts 126.bit_length    # ==> 7
```

int1 << n – int >> n

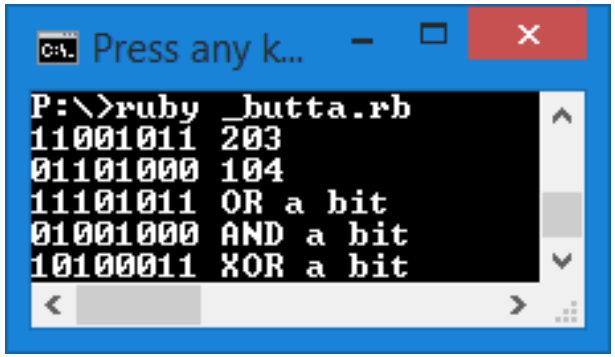
Shift a sinistra/destra di **n** bit della rappresentazione binaria di **int**.

```
puts 15.to_s(2).rjust(8,"0")+" << "+(15 << 2).to_s(2).rjust(8,"0")
# "00001111 << 00111100" - Shift a sinistra di 2 bit
puts 224.to_s(2).rjust(8,"0")+" >> "+(224 >> 3).to_s(2).rjust(8,"0")
# "11100000 >> 00011100" -  Shift a destra di 3 bit
```

int1 | int2 – int1 & int2 – int1 ^ int2

OR, **AND** e **XOR** a bit tra la rappresentazione binaria di **int1** e **int2**.

```
puts 203.to_s(2)+" 203"
puts 104.to_s(2).rjust(8,"0")+ " 104"
puts (104 | 203).to_s(2).rjust(8,"0")+" OR a bit "
puts (104 & 203).to_s(2).rjust(8,"0")+" AND a bit "
puts (104 ^ 203).to_s(2).rjust(8,"0")+" XOR a bit "
```



int[n]

Restituisce l'ennesimo bit della rappresentazione binaria di **int**.

```
puts 5[2].to_s+ 5[1].to_s+ 5[0].to_s # ==> "101"
```

Metodi della classe **FLOAT**

flt.infinite?

Restituisce 1 se `flt` è uguale alla costante `Infinity`, -1 se è pari a `-Infinity` mentre `nil` se si tratta di un numero finito.

```
puts (0.0).infinite?          # ==> nil
puts (-1.0/0.0).infinite?     # ==> -1
puts (+1.0/0.0).infinite?     # ==> 1
```

flt.nan?

Restituisce true se `flt` non è una valida rappresentazione di un Floating Point IEEE 754

```
puts (0.0/0.0).nan?           # ==> true
```

flt.next_float – flt.prev_float

Restituisce il successivo/precedente float rappresentapile dopo `flt`.

```
p 1.0.next_float  # ==> 1.0000000000000002
p 1.0.prev_float  #=> 0.9999999999999999
```

flt.truncate – flt.to_i

Restituisce il numero **num** arrotondato a **ndigit** cifre decimali.

```
puts 1.25943.truncate # ==> 1
puts 1.95943.truncate # ==> 1
```

flt.razionalize([eps]) – flt.to_r

Restituisce una semplice approssimazione del valore **flt** tale che valga la relazione:

flt-|eps! <= risultato <= flt+|eps|

Se **eps** non è fornito verrà assegnato in modo automatico. **flt.razionalize(0)** è equivalente a **flt.to_r**.

```
puts 1.33.rationalize          # ==> 133/100
puts 1.33.rationalize(1)       # ==> 1/1
puts 1.33.rationalize(0.1)     # ==> 4/3
puts 1.33.rationalize(0.001)   # ==> 105/79
puts 1.33.rationalize(0.0001) # ==> 133/100
puts 1.33.rationalize(0)       # ==> 7487234380503 45/562949953421312
puts 1.33.to_r                 # ==> 7487234380503 45/562949953421312
```

La classe **String**

http://ruby-doc.org/core-2.3.1/String.html

La classe delle stringhe è una classe molto simile alla classe **Array**. Entrambe rappresentano un vettore di elementi ordinato, ma nel caso della classe **String**, contiene unicamente caratteri. La manipolazione di stringhe e la loro gestione è molto utile a livello di generazione di output e interfaccia uomo macchina, ma anche per moltissime altre applicazioni, quali la codificazione dei dati o l'autenticazione.

Ruby mette a disposizione diverse modalità per creare delle stringhe (sequenze di simboli alfanumerici). Quando si utilizza il singolo apice le possibili sostituzioni sono ridotte al minimo. Ecco degli esempi

```
puts 'All\'orizzonte\nvedo il \\nulla\\' # ==> All'orizzonte\nvedo il \nulla\
puts String(12) # ==> "12"
```

Quando si utilizza il doppio apice **"** sono ammesse diverse sostituzioni. Ecco degli esempi

```
puts "All\'orizzonte\nvedo il nulla" # ==> All'orizzonte<a capo>vedo il nulla
puts "\103\151\141\157" # ==> "Ciao" (codici Ascii in ottale)
```

La differenza tra la stringa costruita con un *apice singolo* e quella costruita con l'*apice doppio* risiede principalmente nella **interpolazione**. L'interpolazione si effettua utilizzando la sintassi **`#{...}`** dove al posto dei puntini si inserisce una qualsiasi espressione in linguaggio Ruby. Il costrutto **`#{...}`** indica all'interprete che il contenuto deve essere valutato e sostituito con la corrispondente rappresentazione sotto forma di stringa.

```
puts '2 + 2 = #{2 + 2}' # ==> 2 + 2 = #{2 + 2}
puts "2 + 2 = #{2 + 2}" # ==> 2 + 2 = 4
```

Per formato si intende lo stile con il quale un numero/dato deve essere stampato a schermo. La sintassi utilizzata da Ruby è del tutto compatibile a quella del C. In generale il formato si esplicita nel modo seguente

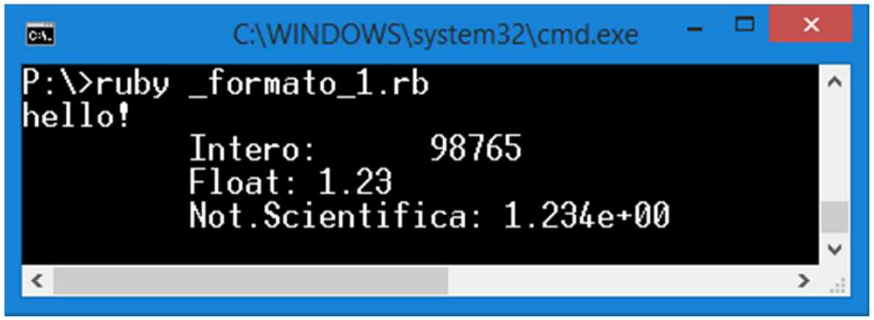
```
"Stringa formato" % [lista, delle, variabili]
```

La **stringa di formato**, tra apici singoli o doppi, è specificata a sinistra del carattere % mentre le variabili che dovranno essere sostituite devono essere separate da una virgola. Questa sintassi viene usata all'interno delle parentesi graffe della interpolazione.

```
intero = 98765
reale = 1.2345
stringa = "hello!"
s = ("%s \n\t Intero: %10d \
      \n\t Float: %2.2f \
      \n\t Not.Scientifica: %3.3e" %
       [stringa, intero, reale, reale])
puts s
```

Usare \ per scrivere una stringa su più linee

Per scrivere un'istruzione su più linee occorre scrivere l'operatore in fondo alla linea prima dell'invio




```
puts "#{"%.2f" % 89.12945 }" # ==> 89.13
puts "#{"%10s %03d" % ["ciao",1] }" # ==> ......ciao.001 // I punti indicano spazi
```

Per accodare una stringa **Str2** ad un'altra **Str1** (concatenazione) abbiamo 2 modi:

```
Str1=Str1+Str2
Str1 << Str2
```

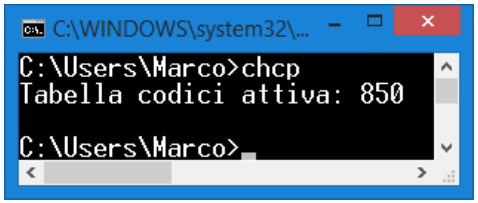
La codifica **US-ASCII** consente la *rappresentazione numerica* dei caratteri alfanumerici, simboli di punteggiatura e altri simboli. Il metodo collegato alla codifica ascii per le stringhe è:

Str.ord

Restituisce la codifica Ascii del primo carattere della stringa.

```
puts "Ave Students".ord ==> 65
```

Poiché il numero dei simboli usati nelle lingue naturali è molto più grande dei caratteri codificabili con **US-ASCII** è stato necessario espandere il set di codifica. Le varie estensioni utilizzano 128 caratteri aggiuntivi codificabili utilizzando l'ottavo bit disponibile in ogni byte.
L'IBM introdusse una codifica a 8 bit sui suoi IBM PC con varianti per i diversi paesi. Le codifiche IBM risultavano ASCII-compatibili, poiché i primi 128 caratteri del set mantenevano il valore originario (US-ASCII). Le varie codifiche vennero divise in pagine (**code page**).
Le diverse code page differivano nei successivi 128 caratteri aggiuntivi codificati utilizzando l'ottavo bit disponibile in ogni byte. I pc costruiti per il Nord America utilizzavano la code page 437, per la Grecia la code page 737, per l'Italia e la Francia la **code page 850**.
Per vedere in DOS la pagina attiva occorre usare il comando dos **chcp**. Nella slide successiva il set di caratteri (esclusi quelli US-ASCII) relativi alla **code page 850**



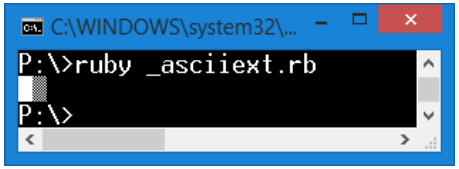
Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)
010 1010	052	42	2A	*
010 1011	053	43	2B	+
010 1100	054	44	2C	,
010 1101	055	45	2D	-
010 1110	056	46	2E	.
010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	;
011 1100	074	60	3C	<
011 1101	075	61	3D	=
011 1110	076	62	3E	>
011 1111	077	63	3F	?

Binary	Oct	Dec	Hex	Glyph
100 0000	100	64	40	@
100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M
100 1110	116	78	4E	N
100 1111	117	79	4F	O
101 0000	120	80	50	P
101 0001	121	81	51	Q
101 0010	122	82	52	R
101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]
101 1110	136	94	5E	^
101 1111	137	95	5F	_

Binary	Oct	Dec	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
110 0111	147	103	67	g
110 1000	150	104	68	h
110 1001	151	105	69	i
110 1010	152	106	6A	j
110 1011	153	107	6B	k
110 1100	154	108	6C	l
110 1101	155	109	6D	m
110 1110	156	110	6E	n
110 1111	157	111	6F	o
111 0000	160	112	70	p
111 0001	161	113	71	q
111 0010	162	114	72	r
111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	|
111 1101	175	125	7D	}
111 1110	176	126	7E	~

Per stampare un carattere appartenente alla all'ASCII esteso possiamo utilizzare le codifiche presenti in tabella:

```
print "\u2592"
print 178.chr(Encoding::CP850)
```



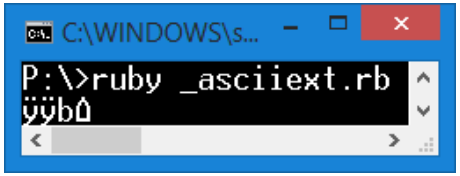
In seguito al proliferare di codifiche proprietarie, l'ISO rilasciò uno standard denominato ISO/IEC 8859 contenente un'estensione a 8 bit del set ASCII. Il più importante fu l'ISO/IEC 8859-1, detto anche **Latin1**, contenente i caratteri per i linguaggi dell'Europa Occidentale. Lo standard ISO/IEC 8859 rappresenta la base di partenza per le codifiche **ISO-8859-1** e **Windows-1252** (*utilizzato come standard di default per le versioni europee di Windows*). **ISO-8859-1** è la codifica di default dei documenti HTML distribuiti mediante il protocollo HTTP con MIME Type del tipo "*text*".

L'elenco delle codifiche disponibili può essere ottenuto in Ruby con il seguente comando:

```
print Encoding.list
```

	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F
0–	NUL 0000 *0*	☺ 263A *1*	☻ 263B *2*	♥ 2665 *3*	♦ 2666 *4*	♣ 2663 *5*	♠ 2660 *6*	• 2022 *7*	◘ 25D8 *8*	○ 25CB *9*	◙ 25D9 *10*	♂ 2642 *11*	♀ 2640 *12*	♪ 266A *13*	♫ 266B *14*	☼ 263C *15*
1–	► 25BA *16*	◄ 25C4 *17*	↕ 2195 *18*	‼ 203C *19*	¶ 00B6 *20*	§ 00A7 *21*	▬ 25AC *22*	↨ 21A8 *23*	↑ 2191 *24*	↓ 2193 *25*	→ 2192 *26*	← 2190 *27*	∟ 221F *28*	↔ 2194 *29*	▲ 25B2 *30*	▼ 25BC *31*
7–	p 0070 *112*	q 0071 *113*	r 0072 *114*	s 0073 *115*	t 0074 *116*	u 0075 *117*	v 0076 *118*	w 0077 *119*	x 0078 *120*	y 0079 *121*	z 007A *122*	{ 007B *123*	| 007C *124*	} 007D *125*	~ 007E *126*	⌂ 2302 *127*
8–	Ç 00C7 *128*	ü 00FC *129*	é 00E9 *130*	â 00E2 *131*	ä 00E4 *132*	à 00E0 *133*	å 00E5 *134*	ç 00E7 *135*	ê 00EA *136*	ë 00EB *137*	è 00E8 *138*	ï 00EF *139*	î 00EE *140*	ì 00EC *141*	Ä 00C4 *142*	Å 00C5 *143*
9–	É 00C9 *144*	æ 00E6 *145*	Æ 00C6 *146*	ô 00F4 *147*	ö 00F6 *148*	ò 00F2 *149*	û 00FB *150*	ù 00F9 *151*	ÿ 00FF *152*	Ö 00D6 *153*	Ü 00DC *154*	ø 00F8 *155*	£ 00A3 *156*	Ø 00D8 *157*	× 00D7 *158*	ƒ 0192 *159*
A–	á 00E1 *160*	í 00ED *161*	ó 00F3 *162*	ú 00FA *163*	ñ 00F1 *164*	Ñ 00D1 *165*	ª 00AA *166*	º 00BA *167*	¿ 00BF *168*	® 00AE *169*	¬ 00AC *170*	½ 00BD *171*	¼ 00BC *172*	¡ 00A1 *173*	« 00AB *174*	» 00BB *175*
B–	░ 2591 *176*	▒ 2592 *177*	▓ 2593 *178*	│ 2502 *179*	┤ 2524 *180*	Á 00C1 *181*	Â 00C2 *182*	À 00C0 *183*	© 00A9 *184*	╣ 2563 *185*	║ 2551 *186*	╗ 2557 *187*	╝ 255D *188*	¢ 00A2 *189*	¥ 00A5 *190*	┐ 2510 *191*
C–	└ 2514 *192*	┴ 2534 *193*	┬ 252C *194*	├ 251C *195*	─ 2500 *196*	┼ 253C *197*	ã 00E3 *198*	Ã 00C3 *199*	╚ 255A *200*	╔ 2554 *201*	╩ 2569 *202*	╦ 2566 *203*	╠ 2560 *204*	═ 2550 *205*	╬ 256C *206*	¤ 00A4 *207*
D–	ð 00F0 *208*	Ð 00D0 *209*	Ê 00CA *210*	Ë 00CB *211*	È 00C8 *212*	ı 0131 *213*	Í 00CD *214*	Î 00CE *215*	Ï 00CF *216*	┘ 2518 *217*	┌ 250C *218*	█ 2588 *219*	▄ 2584 *220*	¦ 00A6 *221*	Ì 00CC *222*	▀ 2580 *223*
E–	Ó 00D3 *224*	ß 00DF *225*	Ô 00D4 *226*	Ò 00D2 *227*	õ 00F5 *228*	Õ 00D5 *229*	µ 00B5 *230*	þ 00FE *231*	Þ 00DE *232*	Ú 00DA *233*	Û 00DB *234*	Ù 00D9 *235*	ý 00FD *236*	Ý 00DD *237*	¯ 00AF *238*	´ 00B4 *239*
F–	SHY 00AD *240*	± 00B1 *241*	‗ 2017 *242*	¾ 00BE *243*	¶ 00B6 *244*	§ 00A7 *245*	÷ 00F7 *246*	¸ 00B8 *247*	° 00B0 *248*	¨ 00A8 *249*	· 00B7 *250*	¹ 00B9 *251*	³ 00B3 *252*	² 00B2 *253*	■ 25A0 *254*	NBSP 00A0 *255*

Per stampare un carattere presente in altre codifiche ecco degli esempi:




```
print 255.chr(Encoding::ISO_8859_1)
print 255.chr(Encoding::Windows_1252)
print 384.chr(Encoding::UTF_8)
print 127.chr(Encoding::ASCII_8BIT)
```

Windows-1252 (CP1252)

	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F
0–	NUL 0000 *0*	SOH 0001 *1*	STX 0002 *2*	ETX 0003 *3*	EOT 0004 *4*	ENQ 0005 *5*	ACK 0006 *6*	BEL 0007 *7*	BS 0008 *8*	HT 0009 *9*	LF 000A *10*	VT 000B *11*	FF 000C *12*	CR 000D *13*	SO 000E *14*	SI 000F *15*
	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
7–	p 0070 *112*	q 0071 *113*	r 0072 *114*	s 0073 *115*	t 0074 *116*	u 0075 *117*	v 0076 *118*	w 0077 *119*	x 0078 *120*	y 0079 *121*	z 007A *122*	{ 007B *123*	| 007C *124*	} 007D *125*	~ 007E *126*	DEL 007F *127*
8–	€ 20AC *128*		‚ 201A *130*	ƒ 0192 *131*	„ 201E *132*	… 2026 *133*	† 2020 *134*	‡ 2021 *135*	ˆ 02C6 *136*	‰ 2030 *137*	Š 0160 *138*	‹ 2039 *139*	Œ 0152 *140*		Ž 017D *142*	
9–		‘ 2018 *145*	’ 2019 *146*	“ 201C *147*	” 201D *148*	• 2022 *149*	– 2013 *150*	— 2014 *151*	˜ 02DC *152*	™ 2122 *153*	š 0161 *154*	› 203A *155*	œ 0153 *156*		ž 017E *158*	Ÿ 0178 *159*
A–	NBSP 00A0 *160*	¡ 00A1 *161*	¢ 00A2 *162*	£ 00A3 *163*	¤ 00A4 *164*	¥ 00A5 *165*	¦ 00A6 *166*	§ 00A7 *167*	¨ 00A8 *168*	© 00A9 *169*	ª 00AA *170*	« 00AB *171*	¬ 00AC *172*	SHY 00AD *173*	® 00AE *174*	¯ 00AF *175*
B–	° 00B0 *176*	± 00B1 *177*	² 00B2 *178*	³ 00B3 *179*	´ 00B4 *180*	µ 00B5 *181*	¶ 00B6 *182*	· 00B7 *183*	¸ 00B8 *184*	¹ 00B9 *185*	º 00BA *186*	» 00BB *187*	¼ 00BC *188*	½ 00BD *189*	¾ 00BE *190*	¿ 00BF *191*
C–	À 00C0 *192*	Á 00C1 *193*	Â 00C2 *194*	Ã 00C3 *195*	Ä 00C4 *196*	Å 00C5 *197*	Æ 00C6 *198*	Ç 00C7 *199*	È 00C8 *200*	É 00C9 *201*	Ê 00CA *202*	Ë 00CB *203*	Ì 00CC *204*	Í 00CD *205*	Î 00CE *206*	Ï 00CF *207*
D–	Ð 00D0 *208*	Ñ 00D1 *209*	Ò 00D2 *210*	Ó 00D3 *211*	Ô 00D4 *212*	Õ 00D5 *213*	Ö 00D6 *214*	× 00D7 *215*	Ø 00D8 *216*	Ù 00D9 *217*	Ú 00DA *218*	Û 00DB *219*	Ü 00DC *220*	Ý 00DD *221*	Þ 00DE *222*	ß 00DF *223*
E–	à 00E0 *224*	á 00E1 *225*	â 00E2 *226*	ã 00E3 *227*	ä 00E4 *228*	å 00E5 *229*	æ 00E6 *230*	ç 00E7 *231*	è 00E8 *232*	é 00E9 *233*	ê 00EA *234*	ë 00EB *235*	ì 00EC *236*	í 00ED *237*	î 00EE *238*	ï 00EF *239*
F–	ð 00F0 *240*	ñ 00F1 *241*	ò 00F2 *242*	ó 00F3 *243*	ô 00F4 *244*	õ 00F5 *245*	ö 00F6 *246*	÷ 00F7 *247*	ø 00F8 *248*	ù 00F9 *249*	ú 00FA *250*	û 00FB *251*	ü 00FC *252*	ý 00FD *253*	þ 00FE *254*	ÿ 00FF *255*

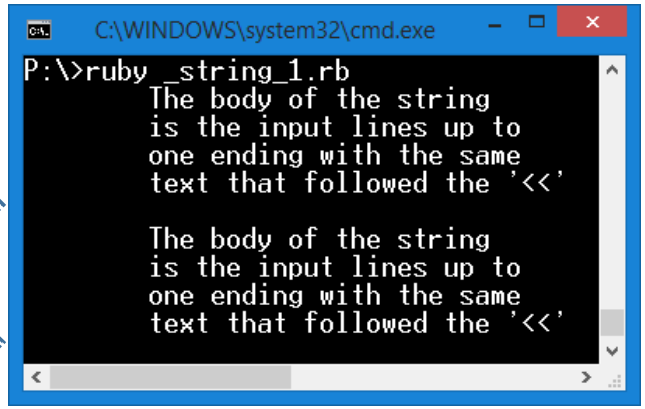
ISO-8859-1

	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F
0–	NUL 0000 *0*	SOH 0001 *1*	STX 0002 *2*	ETX 0003 *3*	EOT 0004 *4*	ENQ 0005 *5*	ACK 0006 *6*	BEL 0007 *7*	BS 0008 *8*	HT 0009 *9*	LF 000A *10*	VT 000B *11*	FF 000C *12*	CR 000D *13*	SO 000E *14*	SI 000F *15*
	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
7–	p 0070 *112*	q 0071 *113*	r 0072 *114*	s 0073 *115*	t 0074 *116*	u 0075 *117*	v 0076 *118*	w 0077 *119*	x 0078 *120*	y 0079 *121*	z 007A *122*	{ 007B *123*	| 007C *124*	} 007D *125*	~ 007E *126*	DEL 007F *127*
8–	PAD 0080 *128*	HOP 0081 *129*	BPH 0082 *130*	NBH 0083 *131*	IND 0084 *132*	NEL 0085 *133*	SSA 0086 *134*	ESA 0087 *135*	HTS 0088 *136*	HTJ 0089 *137*	VTS 008A *138*	PLD 008B *139*	PLU 008C *140*	RI 008D *141*	SS2 008E *142*	SS3 008F *143*
9–	DCS 0090 *144*	PU1 0091 *145*	PU2 0092 *146*	STS 0093 *147*	CCH 0094 *148*	MW 0095 *149*	SPA 0096 *150*	EPA 0097 *151*	SOS 0098 *152*	SGCI 0099 *153*	SCI 009A *154*	CSI 009B *155*	ST 009C *156*	OSC 009D *157*	PM 009E *158*	APC 009F *159*
A–	NBSP 00A0 *160*	¡ 00A1 *161*	¢ 00A2 *162*	£ 00A3 *163*	¤ 00A4 *164*	¥ 00A5 *165*	¦ 00A6 *166*	§ 00A7 *167*	¨ 00A8 *168*	© 00A9 *169*	ª 00AA *170*	« 00AB *171*	¬ 00AC *172*	SHY 00AD *173*	® 00AE *174*	¯ 00AF *175*
B–	° 00B0 *176*	± 00B1 *177*	² 00B2 *178*	³ 00B3 *179*	´ 00B4 *180*	µ 00B5 *181*	¶ 00B6 *182*	· 00B7 *183*	¸ 00B8 *184*	¹ 00B9 *185*	º 00BA *186*	» 00BB *187*	¼ 00BC *188*	½ 00BD *189*	¾ 00BE *190*	¿ 00BF *191*
C–	À 00C0 *192*	Á 00C1 *193*	Â 00C2 *194*	Ã 00C3 *195*	Ä 00C4 *196*	Å 00C5 *197*	Æ 00C6 *198*	Ç 00C7 *199*	È 00C8 *200*	É 00C9 *201*	Ê 00CA *202*	Ë 00CB *203*	Ì 00CC *204*	Í 00CD *205*	Î 00CE *206*	Ï 00CF *207*
D–	Ð 00D0 *208*	Ñ 00D1 *209*	Ò 00D2 *210*	Ó 00D3 *211*	Ô 00D4 *212*	Õ 00D5 *213*	Ö 00D6 *214*	× 00D7 *215*	Ø 00D8 *216*	Ù 00D9 *217*	Ú 00DA *218*	Û 00DB *219*	Ü 00DC *220*	Ý 00DD *221*	Þ 00DE *222*	ß 00DF *223*
E–	à 00E0 *224*	á 00E1 *225*	â 00E2 *226*	ã 00E3 *227*	ä 00E4 *228*	å 00E5 *229*	æ 00E6 *230*	ç 00E7 *231*	è 00E8 *232*	é 00E9 *233*	ê 00EA *234*	ë 00EB *235*	ì 00EC *236*	í 00ED *237*	î 00EE *238*	ï 00EF *239*
F–	ð 00F0 *240*	ñ 00F1 *241*	ò 00F2 *242*	ó 00F3 *243*	ô 00F4 *244*	õ 00F5 *245*	ö 00F6 *246*	÷ 00F7 *247*	ø 00F8 *248*	ù 00F9 *249*	ú 00FA *250*	û 00FB *251*	ü 00FC *252*	ý 00FD *253*	þ 00FE *254*	ÿ 00FF *255*
	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F

Per inserire una stringa distribuita su più righe possiamo utilizzare queste modalità:

```
testo  = <<END_OF_STRING
    The body of the string
    is the input lines up to
    one ending with the same
    text that followed the '<<'
END_OF_STRING
put testo
```

```
testo2 = "\tThe body of the string\n\
\tis the input lines up to\n\
\tone ending with the same\n\
\ttext that followed the '<<'"
puts testo2
```



Metodi della classe **String**

String(Oggetto)

Converte l'argomento in stringa utilizzando il suo metodo `.to_s`.

```
puts String(nil)     # ==> ""
puts String(12)      # ==> "12"
puts String([4,-2,3,-23,189,46,343,12]) # ==> "[4,-2,3,-23,189,46,343,12]"
puts "1.23456".to_f # ==> 1.23456
```

str.length

Restituisce la lunghezza della stringa `str` in caratteri.

```
puts "SEI ALTiSsImO!".length # ==> 13
```

str.empty?

Restituisce vero se `str` è la stringa vuota `""`.

```
puts "".empty?  # ==> true
puts " ".empty? # ==> false
```

str.to_f

Converte la stringa `str` in un numero in virgola mobile. Caratteri estranei dopo la fine di un numero valido sono ignorati. Se non abbiamo un numero valido all'inizio il metodo restituisce `0.0`. Questo metodo non genera mai un'eccezione (errore).

```
puts "euro 90".to_f   # ==> 0.0
puts "90 euro".to_f   # ==> 90.0
puts "9x12 euro".to_f # ==> 9.0
puts "1.23456".to_f   # ==> 1.23456
```

str.to_i

Converte la stringa `str` in un numero intero. Caratteri estranei dopo la fine di un numero valido sono ignorati. Se non abbiamo un numero valido all'inizio il metodo restituisce `0`. Questo metodo non genera mai un'eccezione (errore).

```
puts "euro 90".to_i  # ==> 0
puts "90 euro".to_i  # ==> 90
puts "9x0 euro".to_i # ==> 9
puts "1.23456".to_i  # ==> 1
```

str.upcase

Restituisce una copia di **str** con tutti i caratteri alfabetici sostituiti con il corrispondente maiuscolo. Solo i caratteri dalla '**a**' alla '**z**' vengono considerati.

```
puts "Sei più alto!".upcase # ==> "SEI PIù ALTO!"
```

str.downcase

Restituisce una copia di **str** con tutti i caratteri alfabetici sostituiti con il corrispondente minuscolo. Solo i caratteri dalla '**A**' alla '**Z**' vengono considerati.

```
puts "SEI ALTiSsImO!".downcase # ==> "sei altissimo!"
```

str1.chomp – Str1.chomp(str2)

Elimina da **str1** l'eventuale invio "**\n**" finale oppure la stringa **str2** se è alla fine di **str1**.

```
puts "Ciao\nCiao\n".chomp         # ==> "Ciao\nCiao"
print "Arrivederci".chomp("rci") # ==> "Arrivede"
```

str.chr

Restituisce il primo carattere all'inizio della stringa **str**. Equivale a **Str[0]**.

```
puts "Sei più alto!".chr # ==> "S"
```

str.ord

Restituisce la codifica Ascii del primo carattere di `str`.

```
puts "Ave Students".ord # ==> 65 (Codice ascii di "A")
```

str.empty?

Restituisce vero se `str` è la stringa vuota `""`.

```
puts "".empty?  # ==> true
puts " ".empty? # ==> false
```

str.byteslice(i,n)

Estrae, partendo dall'`i`-esimo carattere di `str`, `n` caratteri (si ricorda che la numerazione posizionale dei caratteri parte da zero!). Non modifica la stringa originale.

```
puts "SEI ALTiSsImO".byteslice(0)     # ==> "S"
puts "SEI ALTiSsImO".byteslice(-1)    # ==> "O"
puts "SEI ALTiSsImO".byteslice(4,5)   # ==> "ALTiS"
puts "SEI ALTiSsImO".byteslice(4..8)  # ==> "ALTiS"
puts "SEI ALTiSsImO".byteslice(4...8) # ==> "ALTi" (con … l'estremo è
                                                   escluso!)
```

str.include?(str2)

Restituisce **true** se **str2** è presente in **str** come sua sottostringa

```
print "Casotto".include?("a")  # ==> true
print "Casotto".include?("so") # ==> true
print "Casotto".include?("e")  # ==> false
print "Casotto".include?("O")  # ==> false
```

str.center(n,ch)

Se **n** è più grande della lunghezza di **str** il metodo restituisce una nuova stringa lunga **n** dove **str** è posta al centro tra caratteri **ch**. Se invece **n** è minore della lunghezza di **str** allora la funzione restituisce **str**.

```
print "|"+"*".center(11," ") +"|\n" # ==> "|     *     |"
print "|"+"*".center(12," ") +"|\n" # ==> "|     *      |"
print "|"+"*".center(13," ") +"|\n" # ==> "|      *      |"
```

str.rjust(n,ch) – str.ljust(n,ch)

Se **n** è maggiore della lunghezza della stringa **str**, il metodo restituisce una nuova stringa di lunghezza **n** giustificata a destra/sinistra e riempita con caratteri **ch**. In caso contrario restituisce **str**.

```
puts "11101".rjust(8,"0")  # ==> "00011101"
puts "ciao".ljust(8)       # ==> "ciao    " (se ch è omesso usa lo spazio)
```

str.split(sep=nil[,limit])

Restituisce un array contenente le sottostringhe di **str** delimitate dalla stringa **sep**. Se **sep** è omesso e la variabile **$;** (variabile predefinita $FIELD_SEPARATOR) è **nil** allora lo spazio verrà utilizzato come separatore (spazi multipli successivi al primo vengono ignorati).

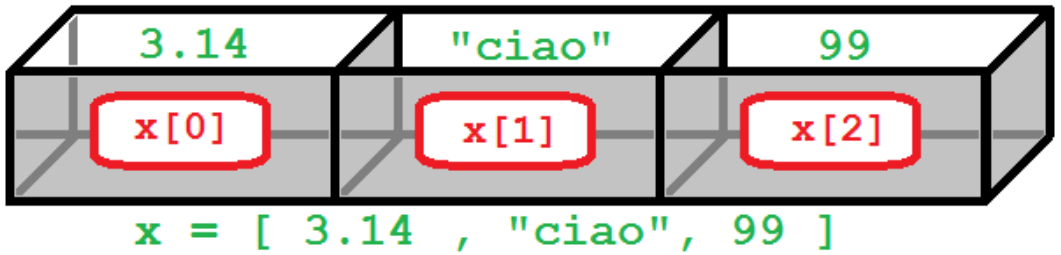
```
print "1,2,3,4,5".split(",")  # ==> ["1","2","3","4","5"]
```

La classe **ARRAY**

http://ruby-doc.org/core-2.3.1/Array.html

Talvolta succede che occorra svolgere ripetutamente la stessa operazione su un gruppo di variabili. Essendo variabili distinte non è possibile ripetere la stessa operazione sfruttando i *costrutti iterativi*

Possiamo ovviare ricorrendo agli **array**. La classe **array** contiene una collezione di riferimenti ad oggetti. Ogni riferimento occupa una posizione nell'array identificata da un indice intero non negativo.

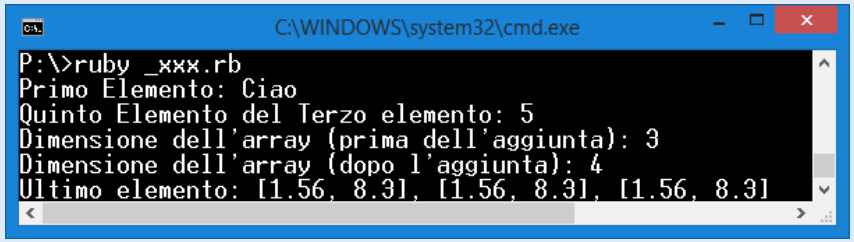


La dimensione dell'array (*numero di elementi*) in Ruby è dinamica (può essere *cambiata durante l'esecuzione del programma*) e dipende dal numero di elementi aggiunti. La numerazione delle posizioni parte da 0.

Gli Array hanno le seguenti caratteristiche:

- I dati contenute nelle singole caselline del vettore non devono essere necessariamente tutti della stessa classe.
- L'indice comincia da 0, e si specifica tra parentesi quadre: **`Elenco[index]`**
- Non è necessario definire a priori la dimensione dell'Array (ovvero il numero delle caselline dentro le quali possiamo inserire dati). La dimensione dell'Array può quindi variare dinamicamente.
- Gli array possono essere multidimensionali, ovvero un Array può contenere un altro Array. Spesso ci si riferisce a questo tipo di array con il nome di **Matrici**.

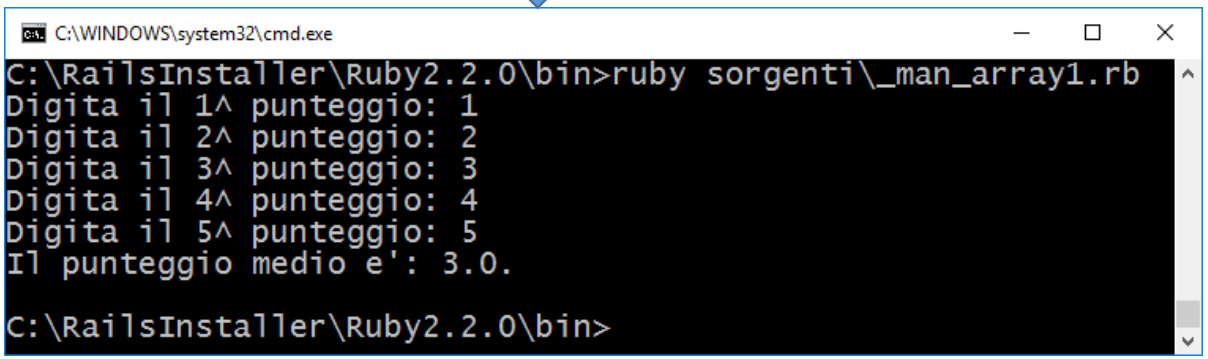
```ruby
# Definizione di un Array vuoto. Esistono 2 modi per definirlo:
elenco = Array.new # oppure
elenco = []
# Inizializzazione array con valori (popolato)
elenco = [ "Ciao", 1.123, [1, 2, 3, 4, 5] ]
puts "Primo Elemento: #{elenco[0]}"
puts "Quinto Elemento del Terzo elemento: #{elenco[2][4]}"
puts "Dimensione dell'array (prima dell'aggiunta): #{elenco.size}"
# Possiamo inserire un nuovo elemento alla fine
elenco << [1.56,8.3] #oppure elenco.push([1.56,8.3])
puts "Dimensione dell'array (dopo l'aggiunta): #{elenco.size}"
# Per accedere all'ultimo elemento ecco 3 modi:
puts "Ultimo elemento: #{elenco[-1]}, #{elenco.last}, #{elenco[elenco.size-1]}"
```



L'utilizzo dell'array permette di evitare di avere un numero di linee di codice proporzionale al numero di variabili necessarie a memorizzare una serie di dati (*si veda l'esempio sottostante*).

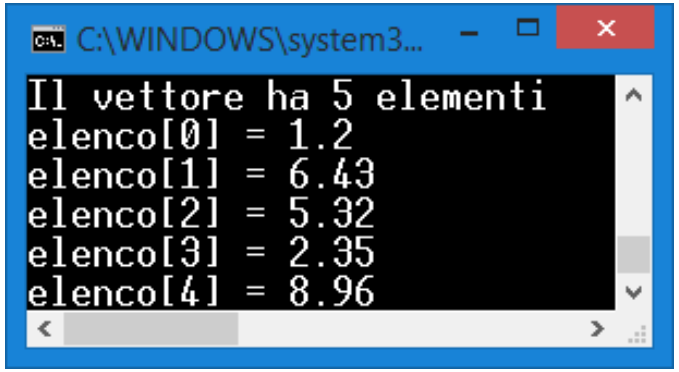
```
score1=0.0  # --- INIZIALIZZAZIONE
score2=0.0
score3=0.0
...
scoreN=0.0
average=0.0
# INPUT
print "Digita il 1^ punteggio: "
score1 = gets.to_f
print "Digita il 2^ punteggio: "
score2 = gets.to_f
print "Digita il 3^ punteggio: "
score3 = gets.to_f
...
print "Digita il N^ punteggio: "
scoreN = gets.to_f
average = score1
# ALGORITMO:
average = average+score1
average = average+score2
...
average = average+scoreN
average = average / N
# OUTPUT
puts "Il punteggio medio e': #{average}."
```

```
# INIZIALIZZAZIONE
score=[]
average=0.0
n=10
# INPUT
1.upto(n) do |i|
        print "Digita il #{i}^ punteggio: "
        score[i-1]=gets.to_f
end
# ALGORITMO: calcola la media
score.each { |x| average=average+x.to_f }
average=average/score.length #anche count
# OUTPUT
puts "Il punteggio medio e': #{average}."
```



L’attraversamento di un array consiste nello scorrere l'intero contenuto mediante un indice. Il costrutto for si adatta molto bene allo scopo

```
elenco = [1.2, 6.43, 5.32, 2.35, 8.96]
puts "Il vettore ha #{elenco.size} elementi"
for i in (0...elenco.size)
   puts "elenco[#{i}] = #{elenco[i]}"
end
```



Per stampare una sottosequenza possiamo indicare tra [] l'indice iniziale e quello finale.

```
elenco = [1.2, 6.43, 5.32, 2.35, 8.96]
puts elenco[1,3] # stampa 6.43, 5.32, 2.35
```

Esercizi ARRAY FOR

A. Generare una sequenza di 10 valori casuali nell'intervallo reale [1,10] e stamparla in modo rovesciato.
B. Generare una sequenza di 1000 interi casuali appartenenti all'intervallo [10,100]. Leggere successivamente un valore **x** e stabilire se tale valore è presente nella sequenza e in caso affermativo indicare la sua prima posizione.
   *Suggerimento: costruire una funzione che dato un numero **x** e un vettore **v[]** di valori restituisce: - la posizione di **x** nel vettore **v[]** se esiste - **nil** nel caso **x** non sia presente in **v[].***
C. Costruire un programma che determini, in una sequenza di interi casuali compresi nell'intervallo [-50, 50] quante volte appare il minimo valore
D. Si consideri un array contenente la seguente sequenza: **[1 2 3 4 … N]**. Si costruisca una prima funzione che ribalta il contenuto dell'array utilizzando un eventuale array di supporto. Successivamente si implementi un'altra funzione che, senza utilizzare ulteriori array di supporto, ribalta il contenuto ovvero restituisce **[N, N-1, … , 2,1]**.
E. Costruire un programma che genera un vettore di **n** numeri interi da 1000 a 99.999 e restituisca il loro **MCD** e **mcm**.
F. Generare 10 numeri float (range a piacere) e restituire il valore che più si avvicina alla media della sequenza
G. Generare una sequenza di lunghezza casuale (compresa tra 3 e 10) di 0 e 1. Interpretare la sequenza come un numero binario $\mathbf{b_n b_{n-1}..b_2 b_1 b_0}$ e convertirlo nel corrispondente numero in base 10 utilizzando la formula $\sum_{i=0}^{n} 2^i \cdot b_i$
H. Costruire una breve applicazione che genera 10 numeri nell'intervallo [10, 100) e restituisce la differenza minima tra una qualsiasi coppia di numeri della sequenza
I. Generare 2 sequenze di interi casuali di lunghezza compresa tra 4 e 10. Visualizzare successivamente gli elementi in comune (non è ammesso l'utilizzo dell'operatore **&** sugli array).
J. Generare una sequenza di numeri casuali compresi tra 1 e 100 e salvarli in un elenco di 100 numeri. Successivamente rimuovere tutti i duplicati (non è permesso l'uso del metodo **uniq**)

Per generare le sequenze numeriche utilizzare la seguente funzione

```
def GeneraSequenza(n,da,a)
   rnd=Random.new(Random.new_seed)
   x=Array.new()
   n.times { x << rnd.rand(da..a) }
   return x
end
# esempio d'uso: x=GeneraSequenza(10,3,8)
```

Soluzione A

Si potevano sfruttare anche forme alternative di iterazione

```
def GeneraSequenza(n,da,a)
   rnd=Random.new(Random.new_seed)
   x=Array.new()
   n.times { x << rnd.rand(da..a) }
   return x
end

v=GeneraSequenza(10,2,9)
print v.to_s + "\n"

for i in (v.size-1).downto(0)
   print "#{v[i]}; "
end
```

```
n=v.size-1
for i in 0..n
   print "#{v[n-i]}; "
end
```

```
for x in v.reverse
  print "#{x}; "
end
```

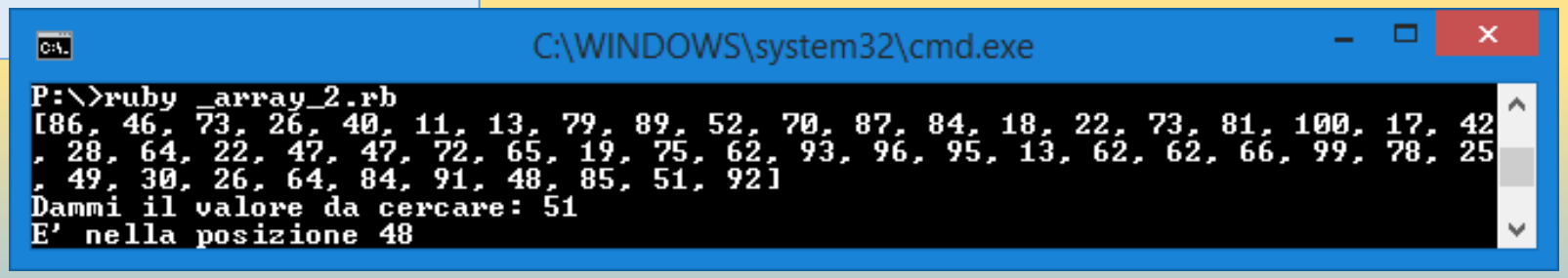
```
v.reverse_each  {|x| print "#{x}; "}
```

```
(v.size-1).downto(0)  {|i| print "#{v[i]}; "}
```

Soluzione B

```
def GeneraSequenza(n,da,a)
   rnd=Random.new(Random.new_seed)
   x=Array.new()
   n.times { x << rnd.rand(da..a) }
   return x
end

def Ricerca(x, v)
   n=v.size-1
   for i in 0..n
      return i if v[i]==x
   end
   return nil
End
...
```

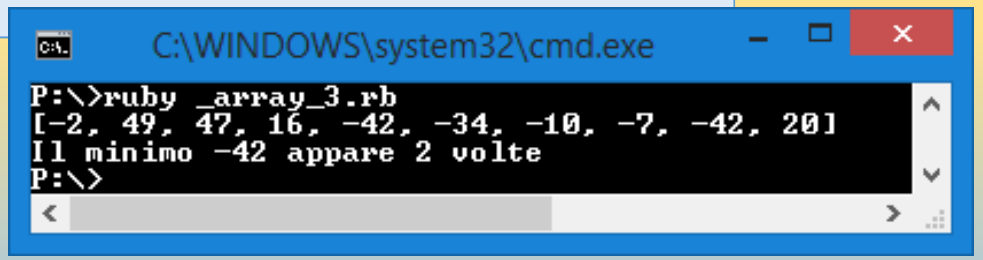
```
...
v=GeneraSequenza(50,10,100)
print v.to_s + "\n" # per verifica!
print "Dammi il valore da cercare: "
x=gets.to_i
p=Ricerca(x,v)

if p==nil
   print "Non trovato"
else
   print "E' nella posizione #{p}"
end
```



Soluzione C

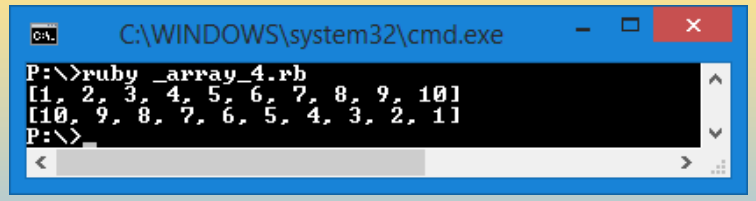
```
def GeneraSequenza(n,da,a)
   rnd=Random.new(Random.new_seed)
   x=Array.new()
   n.times { x << rnd.rand(da..a) }
   return x
end

def minimo(v)
   min=Float::INFINITY
   n=v.size-1
   for i in 0..n
      min = v[i] if (min > v[i])
   end
   return min
End
...
```

```
...
def conta(x,v)
   nr=0
   n=v.size-1
   for i in 0..n
      nr=nr+1 if (x == v[i])
   end
   return nr
end

v=GeneraSequenza(10,-50,50)
print v.to_s + "\n" # per verifica
m=minimo(v) # oppure m=v.min
nr=conta(m,v)
print "Il minimo #{m} appare #{nr} volte"
```



Soluzione D

```
def Ribalta(v)
   t=[]
   n=v.size-1
   for i in 0..n
      t[i]=v[n-i]
   end
   return t
end

v=[1,2,3,4,5,6,7,8,9,10]
puts v.to_s # per verifica
b=Ribalta(v)
print b.to_s # per verifica
```




```
def RibaltaINLINE(v)
   n=v.size-1
   for i in 0..n
      break if (i >= n-i)
      t=v[i]
      v[i]=v[n-i]
      v[n-i]=t
   end
end

v=[1,2,3,4,5,6,7,8,9,10]
puts v.to_s # per verifica
RibaltaINLINE(v)
print v.to_s # per verifica
```

Si osservi che se passo alla funzione un vettore la funzione è in grado di modificarne il contenuto.
Generalmente il passaggio dei parametri in Ruby è a valore.

Soluzione E

La soluzione con il for richiede la conoscenza del valore finale del ciclo. Un multiplo di tutti gli elementi di **v** è dato dalla produttoria di tutti gli elementi di **v** e pertanto il **mcm** sarà minore o uguale a tale intero. Valendo la relazione sottostante la produttoria può essere presa come valore di terminazione dell'iterazione

$$\max_{i:0\to n-1}(v[i]) \le mcm \le \prod_{i=0}^{n-1} v[i]$$

```
def GeneraSequenza(n,da,a)
    rnd=Random.new(Random.new_seed)
    x=Array.new()
    n.times { x << rnd.rand(da..a) }
    return x
end

puts v.to_s
print "Il MCD e' : #{MCD(v)}\n"
print "Il mcm e' : #{mcm(v)}\n"
```

La variabile **divisibile** diventa 0 se ho un elemento di **v** non divisibile per **d**. Pertanto **d** non è un divisore di tutti i termini di **v** e quindi non può essere il MCD.

```
def MCD(v)
   for d in (v.min).downto(1)
      divisibile=1
      for x in v
         divisibile=0 if (x % d !=0)
         break if (divisibile==0)
      end
      break if divisibile==1
   end
   return d
end
```

```
def mcm(v)
   f=100
   v.each {|x| f=f*x}
   for d in (v.max)..f
      divisibile=1
      for x in v
         divisibile=0 if (d % x !=0)
         break if (divisibile==0)
      end
      break if divisibile==1
   end
   return d
end
```

```
def mcm(v)
   d=v.max
   loop do
      divisibile=1
      for x in v
         divisibile=0 if (d % x !=0)
         break if (divisibile==0)
      end
      break if divisibile==1
      d=d+1
   end
   return d
end
```

Utilizzando il loop do – end non è necessario stabilire un valore finale del ciclo

```
def MCD(v)
 for d in (v.min).downto(1)
      divisibili=0
      for x in v
         break if (x % d !=0)
         divisibili=divisibili+1
      end
      break if divisibili==v.size
   end
   return d
end
```

La variabile **divisibili** conta quanti elementi di **v** sono divisibili per **d**. Pertanto se il numero dei divisibili per **d** non è pari al numero di elementi di **v** segue che ho qualche elemento di **v** non divisibile per **d** e quindi **d** non può essere il MCD.

Soluzione F

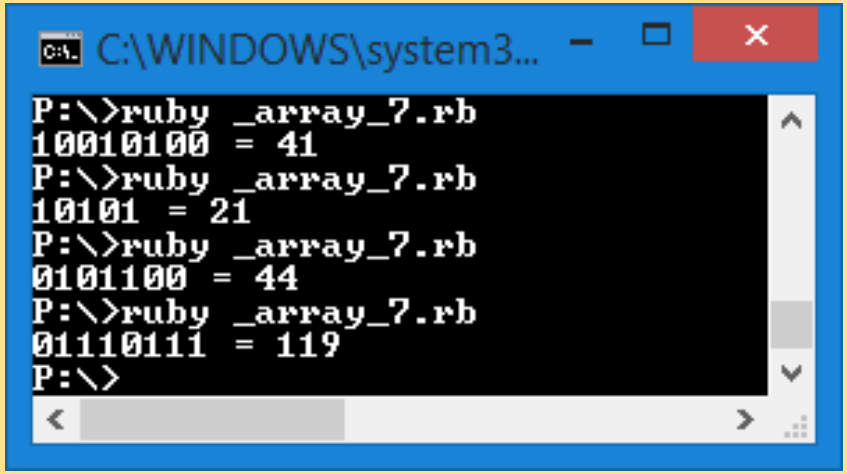
```
v=GeneraSequenza(10,-5.0,5.0,3)
s=0.0 # determino la media
v.each {|x| s=s+x}
m=s/v.size
p_vicino=0 # determino il valore + vicino alla media
distmin=(v[0]-m).abs
for i in 1..(v.size-1)
   if distmin > (v[i]-m).abs
      distmin=(v[i]-m).abs
      p_vicino=i
   end
end
puts "Il valore #{v[p_vicino]} e' quello che + si avvicina alla media #{m}"
```

La funzione di generazione della sequenza è stata modificata con l'aggiunta di un parametro opzionale (**cifredec**) che consente di ottenere sequenze casuali con un numero di decimali a piacere. L'uso del parametro opzionale, opportunamente gestito, consente di mantenere la compatibilità con le soluzioni dei precedenti esempi.

```
def GeneraSequenza(n,da,a, cifredec=0)
   rnd=Random.new(Random.new_seed)
   x=Array.new()
   n.times { x << rnd.rand(da..a) }
   if cifredec!=0
      (0..(x.size-1)).each { |i| [i]=x[i].round(cifredec) }
   end
   return x
end
```

Soluzione G

```
rnd=Random.new(Random.new_seed)
n=rnd.rand(3..10)
b=GeneraSequenza(n,0,1,0,rnd)

s=0
for i in 0..(b.size-1)
   s=s+b[i]*(2**(b.size-1-i))
end
print b.join("")+" = #{s}"
```



La funzione di generazione della sequenza è stata ulteriormente modificata con l'aggiunta del parametro opzionale (**rnd**) che consente di riutilizzare un generatore di sequenze casuali già attivato in precedenza. L'uso del parametro opzionale, opportunamente gestito, consente di mantenere la compatibilità con le soluzioni dei precedenti esempi.

```
def GeneraSequenza(n,da,a, cifredec=0, rnd=nil)
   rnd=Random.new(Random.new_seed) if rnd==nil
   x=Array.new()
   n.times { x << rnd.rand(da..a)}
   if cifredec!=0
      (0..(x.size-1)).each { |i| x[i]=x[i].round(cifredec) }
   end
   return x
end
```

Esercizi ARRAY WHILE

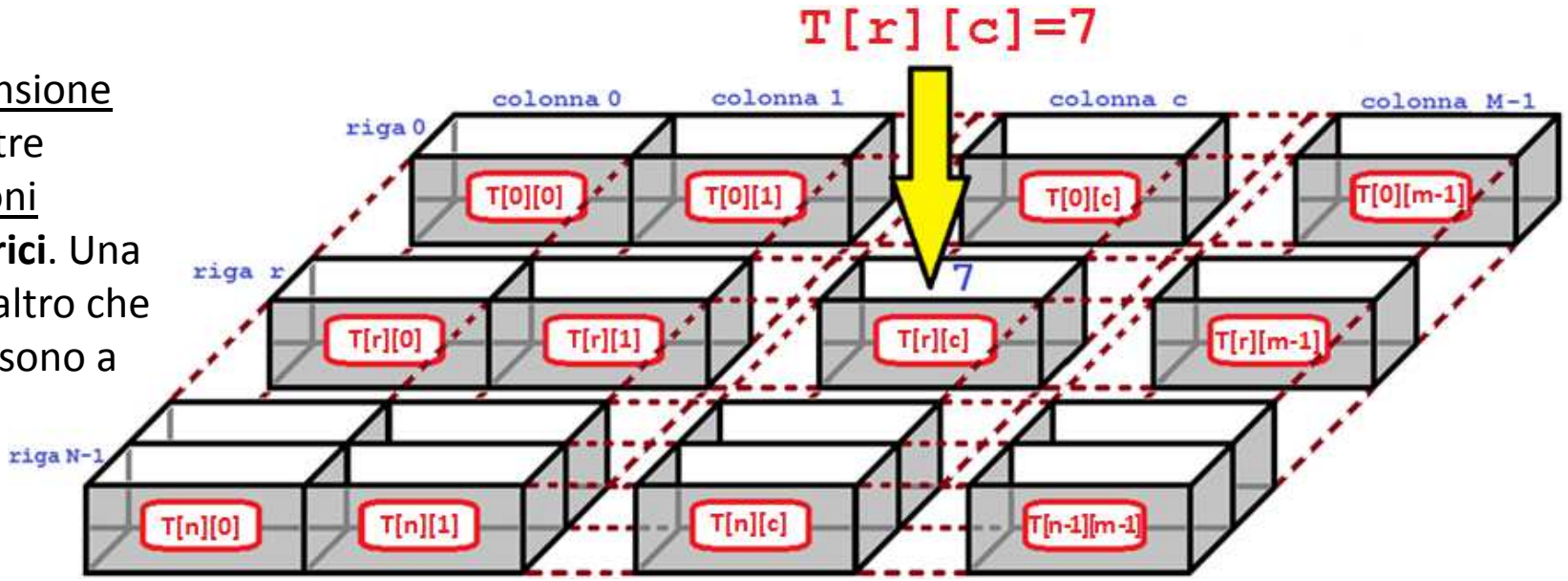
A. Costruire un programma che legge una sequenza di numeri la cui lunghezza non è nota a priori (la sequenza si considera terminata quando l'utente digita 0). Al termine della lettura viene stampata la sequenza invertendo l'ordine di digitazione (l'ultimo valore letto è il primo ad essere stampato).
B. Costruire programma che legge una sequenza di numeri la cui lunghezza non è nota a priori (la sequenza si considera terminata quando l'utente digita 0). Al termine della lettura l'elenco dei numeri deve già risultare ordinato in modo crescente. Occorre quindi inserire il nuovo elemento in modo opportuno all'interno dell'Array
C. Costruire un programma che legge una sequenza di cognomi (si supponga che la sequenza termini quando l'utente digita la stringa vuota ""). I cognomi, nell'elenco iniziale, possono essere anche ripetuti più volte nella sequenza. Al termine della lettura viene fornito l'elenco, privo di ripetizioni, con tutti i cognomi digitati (non è ammesso l'uso del metodo uniq).

Soluzione A

...

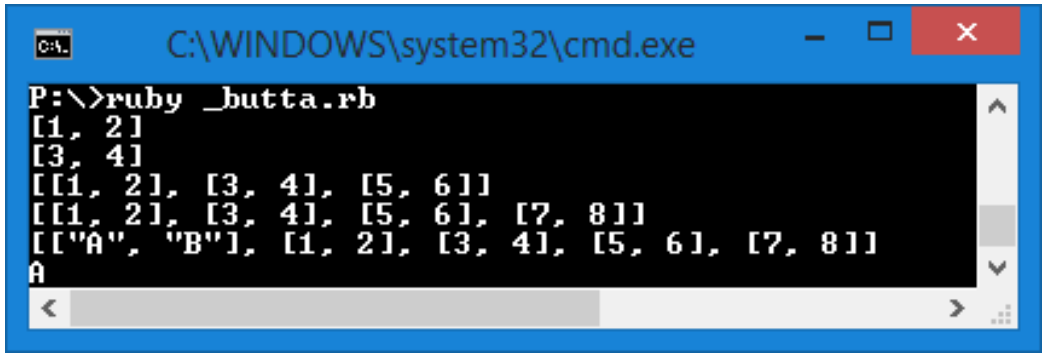
```
. . .
```

Gli array con una dimensione sono detti **vettori** mentre quelli con più dimensioni vengono chiamati **matrici**. Una matrice in Ruby non è altro che un array i cui elementi sono a loro volta degli array:

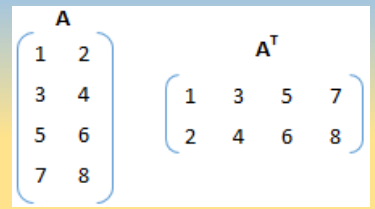


Per accedere ad un preciso elemento si utilizzano le sue coordinate

```
matrice = [[1, 2],[3, 4]]
matrice.each { |r| puts r.to_s }
matrice << [5, 6]
puts matrice.to_s
matrice.push([7,8])
puts matrice.to_s
matrice.unshift(["A","B"])
puts matrice.to_s
puts matrice[0][0] # [r,c] o [r][c]
```



Esercizi MATRICI



A. Costruire un programma che genera una matrice **2**x**4** e stampa la sua trasposta (non è ammesso utilizzare il metodo **traspose**)

B. Sia **A** una matrice NxM e **c** una costante reale. Costruire il programma che stampa il prodotto scalare **c·A**

C. Sia **A** e **B** due matrici **N**x**M**. Costruire il programma che stampa la somma tra le matrici **A+B**

Per generare una matrice numerica utilizzare la seguente funzione

```
def GeneraMatrice(nc,nr,da,a)
   rnd=Random.new(Random.new_seed)
   matr=Array.new()
   nr.times do
      c=Array.new()
      nc.times { c << rnd.rand(da..a) }
      matr << c
   end
   return matr
end
# x=GeneraMatrice(4,5,1,9) # esempio d'uso:
# puts x.to_s
# x.each { |r| puts r.to_s }
```

Soluzione A

...

```
. . .
```

Metodi della classe **Array**

Array.new() – [] – [val_1, .. . , val_n]

Permette di creare un nuovo array

```
elenco = []
elenco = Array.new           #==> []
elenco[3]="Ciao"             #==> [nil, nil, nil,"Ciao"]
elenco = Array.new(3)        #==> [nil, nil, nil]
elenco = Array.new(3, true) #==> [true, true, true]
elenco = [1, 1.3, "ciao"]
elenco = Array.new(5){ |index| index ** 2 } #==> [0, 1, 4, 9, 16]
```

Arr.empty

Ritorna **true** se l'array è vuoto.

```
print [].empty? # ==> true
print [nil].empty? # ==> false
print ["ciao", 12, 11.01].empty? # ==> false
```

Arr.length – Arr.size – Arr.count

Restituisce il numero di elementi in **Arr**.

```
puts [1.2, 16.43, 5.32, 0.35, 8.96].size # ==> 5
puts [1.2, 16.43, 5.32, 0.35, 8.96].length #==> 5
puts [1.2, 16.43, 5.32, 0.35, 8.96].count #==> 5
```

Arr.min

Restituisce il più piccolo elemento in **Arr**.

```
puts [1.2, 16.43, 5.32, 0.35, 8.96].min # ==> 0.35
puts ["Marco", "Paola", "Anna", "Walter", "Michele"].min # ==> "Anna"
```

Arr.max

Restituisce il massimo elemento in **Arr**.

```
puts [1.2, 16.43, 5.32, 0.35, 8.96].max # ==> 16.43
puts ["Marco", "Paola", "Anna", "Walter", "Michele"].max # ==> "Walter"
```

Arr.clear

Svuota l'array.

```
elencoA = [1, 2, 3, 4]
elencoA.clear #  ==> []
```

Arr.first

Restituisce il primo elemento di **Arr**.

```
[1.2, 6.43, 5.32, 2.35, 8.96].first # ==> 1.2
```

Arr.last

Restituisce l'ultimo elemento di **Arr**.

```
[1.2, 6.43, 5.32, 2.35, 8.96].last # ==> 8.96
```

Arr.compact

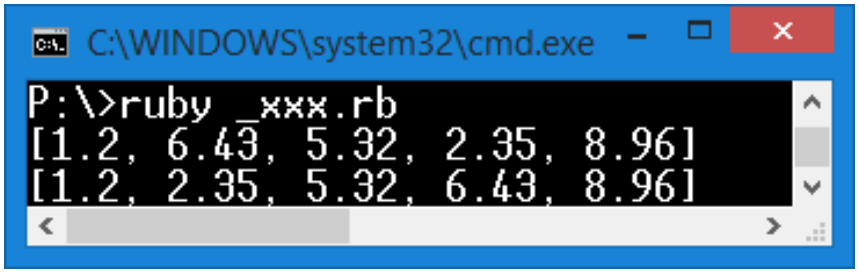
Ritorna un array identico ad **Arr** senza gli elementi **nil**.

```
[1.2, nil, 16.43, nil, 8.96].compact # ==> [1.2, 16.43, 8.96]
```

Arr.sort

Restituisce un array ordinato contenente gli elementi di **Arr**.

```
elenco = [1.2, 6.43, 5.32, 2.35, 8.96].sort
puts elenco.to_s
elencoO=elenco.sort
puts elencoO.to_s
```



Arr.delete(x)

Elimina da **Arr** tutti gli elementi uguali a **x**.

```
elenco=[1, 2, 2, 4, 2, 6]
elenco.delete(2)
print elenco # ==> [1, 4, 6]
```

Arr.delete_at(p)

Restituisce l'elemento in posizione **p** e lo elimina dall'array **Arr** (quindi modifica l'array!).

```
elenco=[1, 2, 2, 4, 2, 6]
x=elenco.delete_at(3)
puts x # ==> 4
print elenco # ==> [1, 2, 2, 2, 6]
```

Arr.drop(n)

Restituisce un array contenente gli elementi di **Arr** posti a partire dalla posizione **n**.

```
elenco=[1.2, 16.43, 5.32, 0.35, 8.96]
print elenco.drop(3) # ==> [0.35, 8.96]
```

Arr.pop

Estrae l'ultimo elemento dell'array **Arr** (la cui dimensione quindi diminuisce di una unità)

```
elenco=[1.2, 16.43, 5.32, 0.35, 8.96]
x=elenco.pop
print elenco # ==> [1.2, 16.43, 5.32, 0.35]
```

Arr.push(x) – Arr << x

Aggiunge alla fine dell'array **Arr** l'elemento **x**.

```
elenco=[1.2, 16.43, 5.32, 0.35, 8.96]
elenco << "ciao" # ==> [1.2, 16.43, 5.32, 0.35, 8.96, "ciao"]
```

Arr.unshift(x)

Inserisce **x** all'inizio di **Arr**.

```
elenco=[1.2, 16.43, 5.32, 0.35, 8.96]
elenco.unshift(7)
puts elenco # ==> [7, 1.2, 16.43, 5.32, 0.35, 8.96]
```

Arr.shift

Estrae il primo elemento (quindi lo elimina!) all'inizio di **Arr**.

```
elenco=[1.2, 16.43, 5.32, 0.35, 8.96]
x=elenco.shift
puts x # 1.2
print elenco # ==> [16.43, 5.32, 0.35, 8.96]
```

Arr.insert(p, x_1[,x_2,...x_n])

Aggiunge in **Arr**, a partire dalla posizione **p**, l'elenco dei valori x_1…x_n.

```
elenco=[1.2, 16.43, 5.32, 0.35, 8.96]
print elenco.insert(2,"A","B") # ==> [1.2, 16.43, "A", "B", 5.32, 0.35, 8.96]
```

Arr.rindex [{ |e| block}]

Ritorna la posizione dell'ultimo elemento di **Arr** uguale a **x**. Nel caso vi sia una sezione **block** quello che per ultimo soddisfa la condizione indicata nel blocco. Se non trovato restituisce **nil**

```
a = [ "a", "b", "b", "b", "c" ]
puts a.rindex("b")              # ==> 3
puts a.rindex("z")              # ==> nil
print a.rindex { |x| x == "b" } # ==> 3
```

Arr.index(x)

Ritorna la posizione del primo elemento di **Arr** uguale a **x**. Nel caso vi sia una sezione **block** quello che per primo soddisfa la condizione indicata nel blocco. Se non trovato restituisce **nil**

```
a = [ "a", "b", "b", "b", "c" ]
puts a.index("b")               # ==> 1
puts a.index("z")               # ==> nil
print a.index { |x| x == "b" } # ==> 1
```

Arr[p,n]

Restituisce un array utilizzando gli **n** elementi di **Arr** posti a partire dalla posizione **p**.

```
elenco=[1.2, 16.43, 5.32, 0.35, 8.96]
print elenco[1,3] # ==> [16.43, 5.32, 0.35]
```

Arr[p..n]

Restituisce un array utilizzando gli elementi di **Arr** posti a partire dalla posizione **p** fino alla posizione **n**.

```
elenco=["A", "B", "C", "D", "E", "F", "G", "H", "I"]
print elenco[3..4] # ==> ["D","E"]
```

Arr[i] – Arr.at(i) – Arr.fetch(i[,messaggio])

Restituisce l'elemento **i**-esimo di **Arr**. **fetch** permette di mostrare un avviso se la richiesta è "*out of range*".

```
elenco=[1.2, 16.43, 5.32, 0.35, 8.96]
puts elenco[1]        # ==> 16.43
puts elenco.at(1)     # ==> 16.43
puts elenco.fetch(1) # ==> 16.43
puts elenco.fetch(100,"Sei fuori dall'array!") # ==> "Sei fuori dall'array!"
```

Arr.include?(x)

Ritorna **true** se l'array contiene un elemento uguale a **x**.

```
puts [1, "ciao", 2.12].include?("Ciao") # ==> false
puts [1, "ciao", 2.12].include?("ciao") # ==> true
puts [1, "ciao", 2.12].include?("ia")   # ==> false
puts [1, "ciao", 2.12].include?("2.12") # ==> false
```

Array.slice(p) - Array.slice(p,n) – Arr.slice(n..m)

Consente di leggere uno o più elementi dell'array **Arr**. Sono equivalenti a: **Arr[p]**, **Arr[p,n]** e **Arr[n..m]**

```
elenco = ["a", "b", "c", "d", "e"]
print elenco.slice(1)    # ==> "b"
print elenco.slice(2..3) # ==> ["c", "d"]
print elenco.slice(1,2)  # ==> ["b", "c"]
print elenco.slice(-1)   # ==> "e"
print elenco.slice(6)    # ==> nil
```

Arr.uniq

Restituisce un array identico ad **Arr** ma senza duplicati (l'array **Arr** resta inalterato!).

```
elenco=[1, 2, 2, 4, 2, 6, 1]
print elenco.uniq # ==> [1, 2, 4, 6]
```

Arr1 & Arr2

Restituisce un nuovo array contenente gli elementi in comune tra i 2 array **Arr1** e **Arr2** senza duplicati. L'ordine rispetta quello del primo array.

```
elencoA = [1, 5, 2, 2, 9, 6, 3, 7]
elencoB = [7, 2, 4, 6, 1, 5]
print elencoA & elencoB # ==> [1, 5, 2, 6, 7]
```

Arr1 + Arr2

Restituisce un nuovo array ottenuto concatenando tutti gli elementi di **Arr1** con quelli di **Arr2**.

```
elencoA = [1, 2, 3, 4]
elencoB = [4, 5, 6]
print elencoA + elencoB # ==> [1, 2, 3, 4, 4, 5, 6]
```

Arr * Str

Restituisce una stringa contenente tutti gli elementi di **Arr** separati da **Str**.

```
elencoA = [1, 5, 2, 2, 9]
print elencoA * " - " # ==> "1 - 5 - 2 - 6 - 7"
```

Arr1 - Arr2

Restituisce un nuovo array contenente gli elementi di **Arr1** che non sono in **Arr2** rimuovendo tutti i duplicati. L'ordine rispetta quello del primo array.

```
elencoA = [1, 9, 2, 2, 9, 6, 3, 7]
elencoB = [7, 2, 4, 6, 1, 5]
print elencoA - elencoB # ==> [9, 9, 3]
```

Arr.take(n)

Restituisce un array contenente i primi elementi di **Arr** (l'array originale resta inalterato).

```
elenco=[1.2, 16.43, 5.32, 0.35, 8.96]
print elenco.take(3) # ==> [1.2, 16.43, 5.32]
```

Arr.select {|x| condizione}

Estrae gli elementi di **Arr** che soddisfano la condizione. L'array originale **Arr** non viene modificato

```
elenco=[1, 2, 3, 4, 5, 6]
estraz=elenco.select { |x| x % 2 == 0 }
print estraz # ==> [2, 4, 6]
```

Arr.reject {|x| condizione}

Estrae gli elementi di **Arr** che non soddisfano la condizione. L'array originale **Arr** non viene modificato

```
elenco=[1, 2, 3, 4, 5, 6]
estraz=elenco.reject { |x| x % 2 == 0 }
print estraz # ==> [2, 4, 6]
```

Arr.delete_if {|x| condizione}

Rimuove dall'array tutti gli elementi che soddisfano la condizione (l'array originale **Arr** viene quindi modificato).

```
elenco = [1, 2, 5, 2, 9, 6, 3, 7]
elenco.delete_if { |x| x <=3 }
print elenco # ==> [5, 9, 6, 7]
```

Arr.keep_if {|x| condizione}

Rimuove dall'array tutti gli elementi che non soddisfano la condizione (l'array originale **Arr** viene quindi modificato).

```
elenco = [1, 2, 5, 2, 9, 6, 3, 7]
elenco.keep_if { |x| x <=3 }
print elenco # ==> [1, 2, 2, 3]
```

Arr.join(sep)

Trasforma l'array **Arr** in una stringa separando ogni elemento con **sep**.

```
elenco = [1, 2, 3, 4]
elenco.join(" // ") # ==> "1 // 2 // 3 // 4"
[].join(" - ")      # ==> ""
```

Arr.pack(direttiva)

Compatta il contenuto dell'array **Arr** in sequenze binarie corrispondenti alla **direttiva** passata come argomento

```
print [65,66,65,66].pack('IiiI').to_s+".\n" # ==> "A   B   A   B   ."
print [65,66,65,66].pack('CccC').to_s+".\n" # ==> "ABAB."
print [65,66,65,66].pack('Qqq').to_s+".\n"  # ==> "A       B       A."
print [65,66,65,66].pack('Q').to_s+".\n"    # ==> "A."
print ["A","B"].pack("A3A4").to_s+".\n"     # ==> "A  B   ."
```

Il metodo **pack** accetta come argomento una stringa di direttive (un char per ogni elemento dell'array). Se la lunghezza dell'argomento non coincide con quella dell'array gli elementi del vettore in eccesso vengono ignorati

Ecco un elenco dei possibili valori della **direttiva**

Integer Directive	Array Element	Meaning
C	Integer	8-bit unsigned (unsigned char)
S	Integer	16-bit unsigned, native endian (uint16_t)
L	Integer	32-bit unsigned, native endian (uint32_t)
Q	Integer	64-bit unsigned, native endian (uint64_t)
c	Integer	8-bit signed (signed char)
s	Integer	16-bit signed, native endian (int16_t)
l	Integer	32-bit signed, native endian (int32_t)
q	Integer	64-bit signed, native endian (int64_t)
S_, S!	Integer	unsigned short, native endian
I, I_, I!	Integer	unsigned int, native endian
L_, L!	Integer	unsigned long, native endian
s_, s!	Integer	signed short, native endian
i, i_, i!	Integer	signed int, native endian
l_, l!	Integer	signed long, native endian

Float Directive		Meaning
D, d	Float	double-precision, native format
F, f	Float	single-precision, native format
E	Float	double-precision, little-endian byte order
e	Float	single-precision, little-endian byte order
G	Float	double-precision, network (big-endian) byte order
g	Float	single-precision, network (big-endian) byte order

String Directive		Meaning
A	String	arbitrary binary string (space padded, count is width)
a	String	arbitrary binary string (null padded, count is width)
Z	String	same as ``a'', except that null is added with *
B	String	bit string (MSB first)
b	String	bit string (LSB first)
H	String	hex string (high nibble first)
h	String	hex string (low nibble first)
u	String	UU-encoded string
M	String	quoted printable, MIME encoding (see RFC2045)
m	String	base64 encoded string (see RFC 2045, count is width)
		(if count is 0, no line feed are added, see RFC 4648)
P	String	pointer to a structure (fixed-length string)
p	String	pointer to a null-terminated string

Arr.combination(n) [{ |c| block }]

Quando viene invocato produce tutte le combinazioni di lunghezza **n** di elementi presi dall'array **Arr**. Se non viene dato alcun blocco viene restituito un **enumeratore**.

```
a=[1, 2, 3]
print a.combination(0).to_a          # ==> [[]]
print a.combination(1).to_a          # ==> [[1],[2],[3]]
print a.combination(2).to_a          # ==> [[1,2],[1,3],[2,3]]
print a.combination(3).to_a          # ==> [[1,2,3]]
print a.combination(4).to_a          # ==> []
```

Arr.permutation(n) [{|p| block}]

Quando viene invocato produce tutte le permutazioni di lunghezza **n** di elementi presi dall'array **Arr**. Se non viene dato alcun blocco viene restituito un **enumeratore**.

```
a=[1, 2, 3]
print a.permutation(1).to_a # ==> [[1],[2],[3]]
print a.permutation(2).to_a # ==> [[1, 2],[1, 3],[2, 1],[2, 3],[3, 1],[3,2]]
print a.permutation(3).to_a # ==> [[1, 2, 3],[1, 3, 2],[2, 1, 3],[2, 3, 1],
                                  [3, 1, 2], [3, 2, 1]]
print a.permutation(0).to_a # ==> [[]] # 1 permutazione di lunghezza 0
print a.permutation(4).to_a # ==> []   # nessuna permutazione di lunghezza 4
```

Arr.collect [{|c| block }] - Arr.map [{|c| block }]

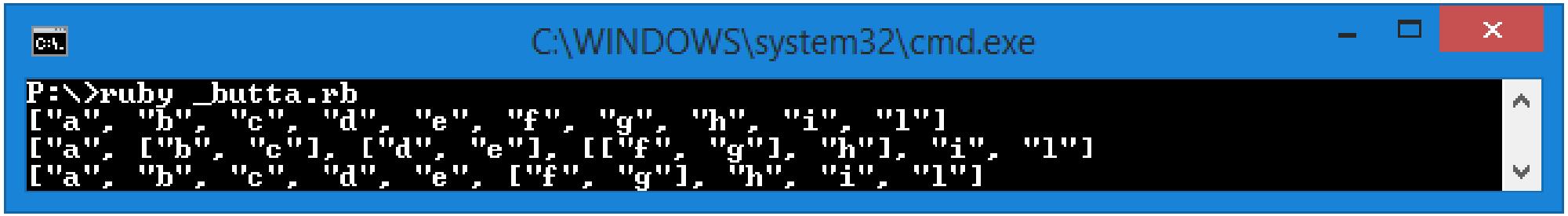
Crea un nuovo array contenenti i valori ritornati dal blocco.

```
a = [ "a", "b", "c", "d" ]
print a.collect { |x| x + "!" }      #=> ["a!", "b!", "c!", "d!"]
```

Arr.flatten([livello])

Ritorna un nuovo array monodimensionale i cui elementi sono ottenuti estraendo tutti gli elementi presenti in un qualsiasi sottoarray contenuto in **arr**. L'argomento opzionale livello determina il livello di recursione per l'appiattimento.

```
a = [ ["a", ["b" , "c"]] , [["d", "e"] , [["f" , "g"] , "h"] , "i"] , "l" ]
puts a.flatten.to_s
puts a.flatten(1).to_s
puts a.flatten(2).to_s
```



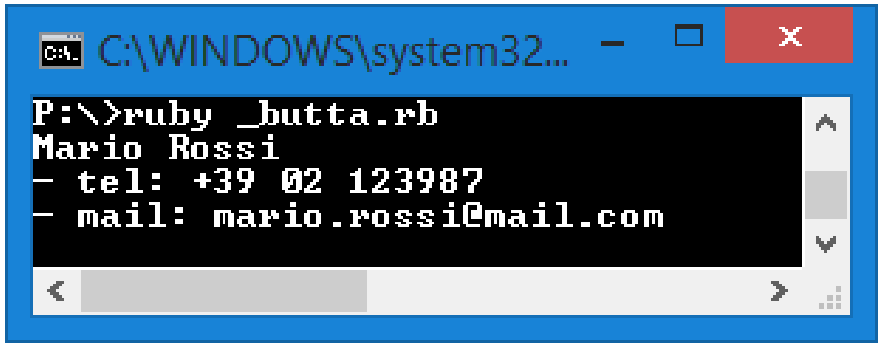
La classe **HASH**

http://ruby-doc.org/core-2.3.1/Hash.html

Gli **Hash**, a volte chiamati anche *Dizionari* sono dei contenitori non-ordinati di dati. Come gli Array, possono contenere diversi elementi al loro interno. A differenza degli Array, gli elementi contenuti all'interno di un Hash non sono ordinati ne identificati per mezzo di un indice a valore crescente ma vengono individuate per mezzo di chiavi (keys) univoche. Solitamente le chiavi utilizzate negli Hash, per motivi puramente prestazionali, sono dei **simboli** ma potrebbero essere anche stringhe!.

Proviamo a costruire un Hash che descrive il "contatto" di una persona: nome, cognome, numero di telefono e indirizzo mail:

```
persona = {
              :nome => "Mario",
              :cognome => "Rossi",
              :telefono => "+39 02 123987",
              :mail => "mario.rossi@mail.com"
          }
puts "#{persona[:nome]} #{persona[:cognome]}"
puts "- tel: #{persona[:telefono]}"
puts "- mail: #{persona[:mail]}"
```



Per accedere ad un elemento dello hash usiamo le [ ] con al loro interno la chiave. Diventa molto più semplice definire strutture complesse, nelle quali accedere senza dover ricordarsi a memoria la posizione di ogni elemento.

Uno **Hash** è chiamato anche *Array associativo*. Un Hash può essere creato utilizzando la forma implicita

```
eta = { "Rossi" => "10", "Verdi" => "6" }
puts "Rossi"+eta["Rossi"]
```

Si può utilizzare la seguente sintassi alternativa quando le chiavi sono simboli. Quindi al posto di scrivere

```
eta = { :Rossi => "10", :Verdi => "6" }
```

possiamo scrivere:

```
eta = { Rossi: "10", Verdi: "6" }
```

Ogni chiave definita mediante un **simbolo** può essere letta in questo modo

```
eta = { Rossi: "10", Verdi: "6" }
puts "Rossi: "+eta[:Rossi]
```

Uno Hash può essere creato anche mediante il suo metodo **new**. Hash ha un **default** che è ciò che viene restituito quando la chiave di accesso utilizzata non esiste. Se il **default** non è stato impostato hash restituisce **nil**. Il **default** può essere definito inserendo il valore come argomento del metodo **new** oppure utilizzando la proprietà **default** :

```
eta = Hash.new("Non esiste")
eta[:Rossi], eta[:Verdi] = "10", "6"
puts "rossi: "+eta[:rossi] # ==> "Non esiste"
puts "Rossi: " +eta[:Rossi] # ==> 10
```

Oppure

```
eta = Hash.new()
eta.default="Non esiste"
...
```

Gli Hash sono utilizzati anche per fornire un nome ai parametri delle funzioni. Si osservi che nessuna parentesi è utilizzata in quel caso. Se l'ultimo argomento è un Hash non è necessario nell'istruzione di chiamata utilizzare le parentesi.

```
def StampaNominativo(parametri)
   puts parametri[:cognome]+" "+parametri[:nome]
end

def StampaElenco(parametri)
   parametri.each { |key, value| puts ":#{key}=#{value}"}
end

StampaNominativo :cognome => "Sechi", :nome => "Marco"
StampaNominativo cognome: "Peroni", nome: "Elena"
StampaElenco :cognome => "Sechi", :nome => "Marco"
```

Esercizi HASH

A. Costruire una rubrica telefonica (inserite in un array **db** i cui elementi sono degli HASH le cui voci sono: Cognome, Nome e Telefono) . Implementare successivamente le seguenti funzioni:
   - **`stampaElenco`** ➔ accetta come argomenti il **db** e stampa per ogni elemento tutte le singole voci
   - **`ricercaPer`** ➔ accetta come argomenti **dove** (*campo dove cercare*) e **cosa** (*valore cercato*) e restituisce l'HASH che soddisfa i requisiti della ricerca).
   - **`ordinaPer`** ➔ accetta come argomento **dove** (campo da utilizzare per la ricerca) e riordina array **db**

Soluzione A

...

```
. . .
```

Metodi della classe **Hash**

Hash(Oggetto) – Hash(key_1,$value_1$,…) – Hash([key_1,$value_1$],…)

Crea uno hash utilizzando gli argomenti passati

```
Hash["a", 100, "b", 200]              # ==> {"a"=>100, "b"=>200}
Hash[ [ ["a", 100], ["b", 200] ] ]    # ==> {"a"=>100, "b"=>200}
Hash["a" => 100, "b" => 200]          # ==> {"a"=>100, "b"=>200}
Hash[:a => 100, :b => 200]            # ==> {:a=>100, :b=>200}
Hash[a:100, b:200]                    # ==> {:a=>100, :b=>200}
```

Hash.new – Hash.new(default) – Hash.new {|hash, key| block}

Crea un nuovo hash. Se viene specificato un blocco questo verrà eseguito usando come parametri l'hash stesso e la chiave.

```
h = Hash.new { |hash, key| hash[key] = "Chiave assegnata: #{key}" }
puts h["k"]           # ==> "Chiave assegnata: k"
```

hsh1 < hsh2

Restituisce true se **hsh2** è un sottoinsieme proprio di **hsh1**.

```
h1 = {a:1, b:2}
h2 = {a:1, b:2, c:3}
h1 < h2    # ==> true
h2 < h1    # ==> false
h1 < h1    # ==> false
```

hsh1 <= hsh2

Restituisce true se **hsh2** è uguale o un sottoinsieme di **hsh1**.

```
h1 = {a:1, b:2}
h2 = {a:1, b:2, c:3}
h1 <= h2   # ==> true
h2 <= h1   # ==> false
h1 <= h1   # ==> true
```

hsh.key(value)

Restituisce la **chiave** del primo elemento che ha come valore **value**. Se non esiste il valore **value** il metodo restituisce **nil**.

```
print ({ "a" => 100, "b" => 200, "c" => 300, "d" => 300 }).key(200) # ==> "b"
```

hsh1 == hsh2

Due hash sono uguali se ognuno contiene le stesse chiavi e se per ogni chiave corrisponde lo stesso valore

```
h1 = { "a" => 1, "c" => 2 }
h2 = { "a" => 1, "b" => 2, "c" => 3 }
h3 = { "b" => 2, "c" => 3, "a" => 1 }
h4 = { "a" => 1, "d" => 2, "f" => 3 }
puts h1 == h2   # ==> false
puts h2 == h3   # ==> true (stessi elementi ma in posizione diversa)
puts h3 == h4   # ==> false
# non posso dichiarare chiavi duplicate nello stesso hash!
# h = { "a" => 1, "c" => 2, "a" => 1 } mi da un warning (duplicated key)!
```

hsh[key]=value – hsh.store(key,value)

Abbina alla chiave **key** il valore **value**.

```
h={"a"=>9, "c"=>4}
h.store("b", 42) # ==> 42
puts h   #=> {"a"=>9, "c"=>4, "b"=>42}
h.store("a", 12) # ==> 42
puts h   #=> {"a"=>12, "c"=>4, "b"=>42}
```

hsh.clear

Svuota l'hash

```
h = { "a" => 1, "c" => 2 }
h.clear # ==> {}
```

h.assoc(key)

Restituisce un array di 2 elementi contenente la coppia `[key, value]` dove `value` è il valore associato alla chiave `key`. Se `key` non è presente in `hsh` verrà restituito `nil` (il `default` value viene ignorato in ogni caso).

```
h={ "a"=>1, "b"=>2 }
h.default="Non trovato"
print h.assoc("b") # ==> ["b",2]
print h.assoc("x") # ==> nil
```

h.rassoc(value)

Restituisce un array di 2 elementi contenente la coppia `[key, value]` dove `key` è la prima chiave associata al valore `value`. Se `value` non è presente in `hsh` verrà restituito `nil` (il `default` value viene ignorato).

```
h={ "a"=>1, "b"=>2, "c"=>1, "d"=>2, "e"=>1 }
print h.rassoc(2) # ==> ["b",2]
print h.rassoc(5) # ==> nil
```

hsh.delete_if {|key,value| block}

Elimina fisicamente gli elementi che soddisfano la condizione indicata nel blocco.

```
h = { "a" => 100, "b" => 200, "c" => 300 }
h.delete_if {|key, value| key >= "b" }
print h # ==> {"a"=>100}
```

hsh.each {|key,value| block}

Esegue il blocco per ogni elemento in **hsh** passando come parametro la coppia **key-value**.

```
h = { "a" => 100, "b" => 200 }
h.each {|key, value| puts "#{key} = #{value}" }
```

hsh.flatten(level)

Restituisce un array mono-dimensionale ottenuto appiattendo l'hash. Ogni chiave **key** e valore **value** vengono inseriti come elementi dell'array restituito. Il metodo non appiattisce ricorsivamente ma il livello è determinato dall'argomento opzionale **level**.

```
a = {1=> "one", 2 => [2,"two"], 3 => "three"}
a.flatten    # => [1, "one", 2, [2, "two"], 3, "three"] # ==> come flatten(1)
a.flatten(2) # => [1, "one", 2, 2, "two", 3, "three"]
```

hsh.has_key?(x) - hsh.include?(x) - hsh.key?(x) - hsh.member?(x)

Restituisce true se **x** è uno delle chiavi di **hsh**.

```
h = { "a" => 100, "b" => 200 }
h.has_key?("a")   # ==> true
h.has_key?("z")   # ==> false
```

hsh.has_value?(x)

Restituisce true se **x** è uno dei valori di **hsh**.

```
h = { "a" => 100, "b" => 200 }
h.has_value?(100)   # ==> true
h.has_value?(999)   # ==> false
```

hsh.to_s

Restituisce il contenuto dell'hash sotto forma di stringa.

```
print ({ "a" => 100, "b" => 200 }).to_s # ==> {"a"=>100,"b"=>200}
```

hsh.invert

Restituisce un nuovo Hash ottenuto utilizzando come valori le chiavi e come chiavi i valori di **hsh** . Se una chiave ha lo stesso valore di un'altra chiave allora l'ultima che viene definita verrà utilizzata come valore mentre il precedente verrà scartato.

```
h = { "n" => 100, "m" => 100, "y" => 300, "d" => 200, "a" => 0 }
print h.invert   # ==> { 100=>"m", 300=>"y", 200=>"d", 0=>"a" }
```

hsh1.merge(hsh2)

Restituisce un nuovo hash (non modifica quindi **hsh1**!) composto dagli elementi di **hsh1** e **hsh2**. Il valore assegnato alle chiavi presenti sia in **hsh1** e **hsh2** è quello presente in **hsh2**. In presenza di una sezione block il valore verrà determinato dall'esecuzione del blocco che ha come parametri la chiave **key**, il suo valore in **hsh1** (**oldval)** e in **hsh2** (**newval**).

```
h1 = { "a" => 100, "b" => 200 }
h2 = { "b" => 254, "c" => 300 }
h1.merge(h2)   # ==> {"a"=>100, "b"=>254, "c"=>300}
h1.merge(h2) {|key,oldval,newval| oldval} # ==> {"a"=>100, "b"=>200, "c"=>300}
h1.merge(h2) {|key,oldval,newval| newval} # ==> {"a"=>100, "b"=>254, "c"=>300}
h1.merge(h2) {|key,oldval,newval| newval-oldval}
# ==> {"a"=>100, "b"=>54, "c"=>300}
```

hsh.keys

Restituisce un array dove gli elementi sono le chiavi di **hsh**

```
print ({ "a" => 100, "b" => 200 }).keys # ==> ["a" ,"b"]
```

hsh.delete(key) [{|elem| block}]

Elimina, se esiste, l'elemento con chiave **key**. Se **key** non è presente in **hsh** verrà restituito **nil**.

```
h={ "a"=>1, "b"=>2 }
h.delete("b")  # ==> {"a"=>1}
h.delete("c")  { |el| print "#{el} non trovato!" } # ==> c non trovato!
```

hsh.length – hsh.size

Restituisce il numero di elementi che compone **hsh**.

```
print ({ "a" => 100, "b" => 200 }).length # ==> 2
```

hsh.to_a

Converte l'Hash in un array i cui elementi sono array composti dalle coppie **[key,value]** di **hsh**.

```
print ({"a"=>1, "b"=>2, "c"=>3 }).to_a # ==> [["a", 1], ["b", 2], ["c", 3]]
```

hsh.select {|x| blocco}

Restituisce un nuovo hash composto dagli elementi di **hsh** per i quali il blocco risulta vero.

```
h = { "a" => 100, "b" => 200, "c" => 300 }
h.select {|k,v| k > "a"}  # ==> {"b" => 200, "c" => 300}
h.select {|k,v| v < 200}  # ==> {"a" => 100}
```

hsh.reject {|x| blocco}

Restituisce un nuovo hash composto dagli elementi di **hsh** per i quali il blocco risulta falso.

```
h = { "a" => 100, "b" => 200, "c" => 300 }
h.reject {|k,v| k < "b"}  # ==> {"b" => 200, "c" => 300}
h.reject {|k,v| v > 100}  # ==> {"a" => 100}
```

hsh.shift

Restituisce un array contenente i 2 elementi (**key** e **value**) del primo elemento che viene poi rimosso da **hsh**.

```
h = { 1 => "a", 2 => "b", 3 => "c" }
x=h.shift     # ==> [1, "a"]
puts x.to_s   # ==> [1, "a"]
puts h.to_s   # ==> {2=>"b", 3=>"c"}]
```

hsh.values

Restituisce un array dove gli elementi sono i valori di **hsh**

```
print ({ "a" => 100, "b" => 100 }).values # ==> [100,100]
```

Metodi della modulo **Math**

Il modulo Math contiene le funzioni di base trigonometriche e trascendentali

Math.sqrt

Restituisce la radice quadrata del numero passato come argomento. Segnala l'errore **ArgError** se l'argomento è negativo.

```
puts Math.sqrt(16) # ==> 4.0
```

Math::E – Math::PI

Restituisce le costanti float corrispondenti al numero di eulero e al pi greco.

```
puts Math::E , Math::PI # ==> 2.718281828459045\n3.141592653589793
```

Math.cbrt

Restituisce la radice cubica del numero passato come argomento.

```
puts Math.cbrt(64) # ==> 4.0
```

Math.hypot(x,y)

Restituisce l'ipotenusa ($\sqrt[2]{x^2 + y^2}$) di un triangolo rettangolo con lati x e y.

```
puts Math.hypot(3,4) # ==> 5.0
```

ALGORITMI

Testare se si tratta di un numero - funzioni

Possiamo utilizzare la seguente funzione:

```
def is_number?(object)
  true if Float(object) rescue false
end
```

`Float` converte l'oggetto passato come argomento in float. Il comando `rescue` postfisso intercetta l'eventuale impossibilità di conversione (quando non è un numero!) e restituisce false

```
print "Dammi a: "
a=gets.chomp
if is_number?(a)
        print "E' un numero!"
else
        print "Non e' un numero!"
end
```

Questa funzione non fornisce una risposta corretta quando ho degli zeri superflui in fondo alla parte decimale. Infatti `"10.00".to_f =10.0` e `10.0.to_s="10.0".` Chiaramente se passo alla funzione una stringa ottengo sempre come risposta false. Infatti `"ciao".to_f = 0.0` e `0.0.to_s="0.0"` che è diverso da `"ciao".to_s ="ciao".` Nel caso di un intero la funzione restituisce `true` se non ha gli zeri nella parte decimale: infatti `"12".to_i=12` che trasformato in stringa mediante il metodo `.to_s` diventa `"12"` che coincide con `"12".to_s`

```
def is_number?(object)
   object.to_f.to_s == object.to_s || object.to_i.to_s == object.to_s
end
```

Testare se si tratta di un numero - metodo

Possiamo definire le precedenti funzioni come nuovi metodi della classe **Object** o **String**:

```
class String
  def is_integer?
    self.to_i.to_s == self
  end
  def is_number?
    self.to_f.to_s == self || is_integer?
  end
end
```

```
class Object
  def is_number?
    true if Float(self) rescue false
  end
end
```

```
print "Dammi a: "
a=gets.chomp
if a.is_number?
       print "E' un numero!"
else
       print "Non e' un numero!"
end
```

Determinazione del minimo

L'algoritmo per determinare il valore minimo e massimo contenuto all'interno di un array è di tipo iterativo. Un procedimento potrebbe essere quello di partire considerando gli elementi iniziali come **minimo** e **massimo attuali** della nostra sequenza. Successivamente scorriamo l'array elemento per elemento. Se all'i-esima iterazione l'elemento i-esimo `v[i]` risulta minore del `minimoattuale` allora `v[i]` diventa il nuovo `minimoattuale`. Analogamente se l'i-esimo `v[i]` risulta maggiore del `massimoattuale` allora `v[i]` diventa il nuovo `massimoattuale`. Al termine in `minimoattuale` e `massimoattuale` avremo il valore minimo e massimo della nostra sequenza `v`. Per quanto riguarda l'inizializzazione del minimo e del massimo attuale possiamo ricorrere alla costante `Float::INFINITY` (porrò come minimo attuale iniziale il massimo valore possibile (`Float::INFINITY`) mentre come massimo attuale iniziale il minimo valore possibile (`-Float::INFINITY`). In alternativa si poteva utilizzare i medodi `min` e `max` dei vettori.

Esempio di programma che utilizza le funzioni proposte

```
a=[23, 5, 35, 47, 254, 59, "gianni", "berti", "gianniisa", 214]
v=EliminaElementiNonNumerici(a)
print MinMax(v).to_s+"\n"
print MinMax_1(v).to_s+"\n"
print MinMax_2(v).to_s+"\n"
```

Procedura per eliminare gli elementi non numerici

```
def EliminaElementiNonNumerici(v)
   b=[]
   for i in 0..(v.size-1) do
      if v[i].is_a?(Numeric) then
         b.push(v[i]) # b << v[i]
      end
   end
   return b
end
```

```
def MinMax_1(v)
   min = Float::INFINITY
   max = -Float::INFINITY
   for i in 0..(v.size-1) do
      min = v[i] if (v[i] < min)
      max = v[i] if (v[i] > max)
   end
   return [min, max]
end
```

Ecco diverse funzioni che implementano il calcolo del minimo e del massimo (la funzione MinMax_2 non è applicabile se non abbiamo degli array)

```
def MinMax_2(a)
   return [a.max,a.min]
end
```

```
def MinMax(v)
   min = v[0]
   max = v[0]
   for i in 1..(v.size-1) do
      min = v[i] if (v[i] < min)
      max = v[i] if (v[i] > max)
   end
   return [min, max]
end
```

Bubble SORT

L'algoritmo di ordinamento (sorting) **bubble sort** è un metodo iterativo per l'ordinamento di una sequenza di dati. Fondamentalmente ci spostiamo, ad ogni iterazione, lungo il vettore comparando due a due gli elementi, procedendo in una direzione stabilita. Se il numero a sinistra è maggiore di quella a destra, i due elementi vengono scambiati di posizione.

Dopo la prima iterazione il valore massimo si trova sempre in ultima posizione, quindi nell'iterazione successiva si riducono i confronti di una unità. Effettuate le iterazioni necessarie a coprire tutto l'Array, il vettore finale risulterà ordinato.

Il Bubble sort non è un algoritmo efficiente: ha una complessità computazionale dell'ordine di $\theta(n^2)$ confronti, con **n** elementi da ordinare; si usa solamente a scopo didattico in virtù della sua semplicità, e per introdurre i futuri programmatori al ragionamento algoritmico e alle misure di complessità.

```
elenco=[3, 7, 4, 9, 5, 2, 6, 1]
for j in 1...elenco.length do # Scorro tutto l'array
   # scorro tutte le coppie da sinistra a destra
   for i in (0...(elenco.length - j))
      if elenco[i] > elenco[i.next] #swap
         elenco[i], elenco[i.next] = elenco[i.next], elenco[i]
      end
   end
end
```

Script che implementa l'algoritmo del bubble sort senza alcuna ottimizzazione.

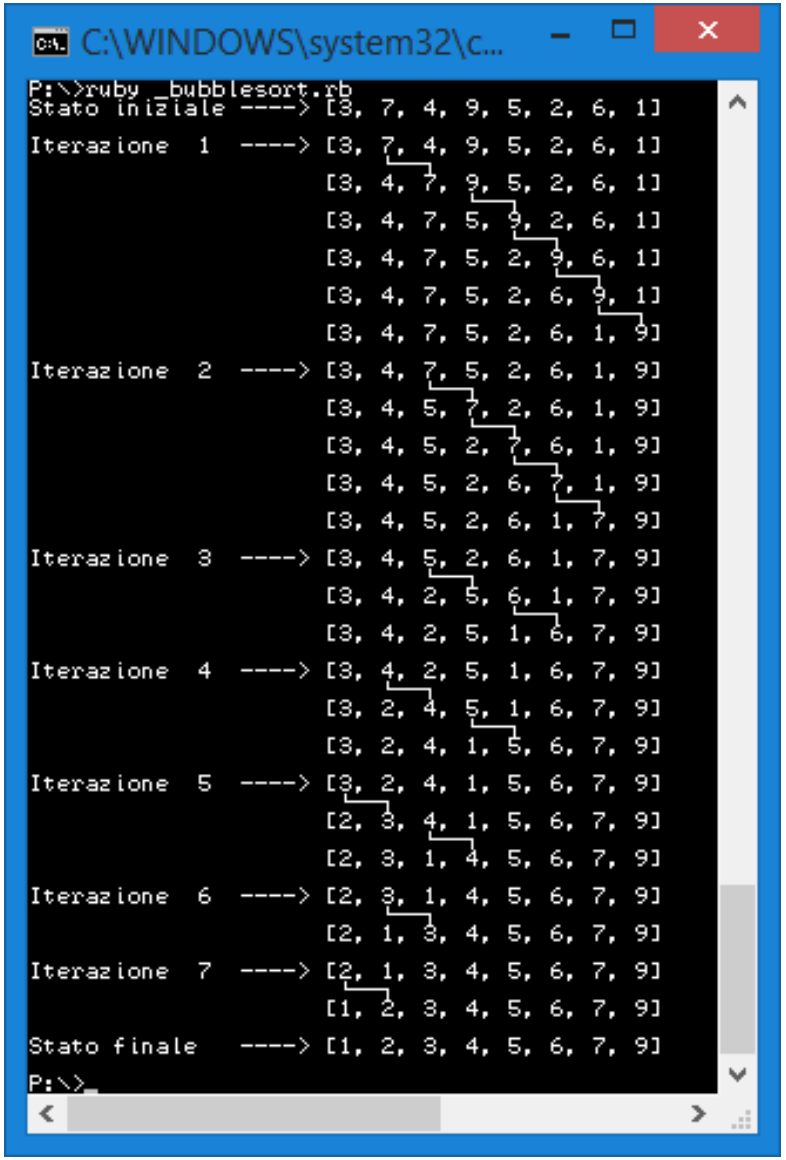
Nella slide successiva un piccolo miglioramento: se non ho avuto swap segue che l'array è ordinato per cui è inutile proseguire e esco con un break dal ciclo.

```ruby
elenco=[3, 7, 4, 9, 5, 2, 6, 1]
puts "Stato iniziale ----> #{elenco}"
symbSwap="\u2514\u2500\u2500\u2510"
# Scorro tutto l'array
for j in 1...elenco.length do
   puts "\nIterazione #{ "%2d " % j} ----> #{elenco}"
   hoscambiato=false
   # scorro tutte le coppie da sinistra a destra
   for i in (0...(elenco.length - j))
      if elenco[i] > elenco[i.next] #swap
         t=elenco[i]
         elenco[i]=elenco[i.next]
         elenco[i.next]=t
         puts (" "*22)+("   "*i)+ symbSwap
         puts (" "*21)+"#{elenco}"
         hoscambiato=true
      end
   end
   #termino in anticipo se non ho fatto scambi
   break if (!hoscambiato)
end
puts "\nStato finale   ----> #{elenco}"
```

Potevo usare in alternativa:

```ruby
elenco[i], elenco[i.next] = elenco[i.next], elenco[i]
```

Evito con questa variabile di continuare poiché senza scambi l'array è ordinato.



In realtà tutti gli elementi che si trovano a valle dell'ultimo scambio effettuato risultano ordinati e si può evitare di trattarli ancora nell'iterazione successiva.

```
elenco=[3, 7, 4, 9, 5, 2, 6, 1]
puts "Stato iniziale ----> #{elenco}"
# Scorro tutto l'array
fine=elenco.length-1
for j in 1...elenco.length do
   puts "\nIterazione #{ "%2d " % j} ----> #{elenco}"
   ultimoscambio=0
   # scorro tutte le coppie da sinistra a destra
   for i in (0...fine)
      if elenco[i] > elenco[i.next] #swap
         t=elenco[i]
         elenco[i]=elenco[i.next]
         elenco[i.next]=t
         puts (" "*22)+("   "*i)+"\u2514\u2500\u2500\u2510"
         puts (" "*21)+"#{elenco}"
         ultimoscambio=i
      end
   end
   fine=ultimoscambio
end
puts "\nStato finale   ----> #{elenco}"
```

Evito con questo assegnamento di di continuare oltre l'ultimo scambio poiché oltre l'array risulta ordinato.

I Files

I file, la loro lettura e la loro scrittura, sono solitamente un concetto ostico per i neofiti della programmazione. L'utilizzo dei file da parte di un programma è solitamente molto diverso rispetto a quello di una persona. Pensate ad un file di testo, lo aprite con il vostro editor e usate il mouse per portarvi esattamente nella posizione che volete.

Per un programma non è così. Il file è un flusso (stream) di dati che si attraversa. Non esiste una posizione assoluta, ma solo una posizione relativa rispetto all'inizio del file. Ruby mette a disposizione delle funzioni di alto livello per la gestione di files, che per nostra comodità sono visti come sequenza di righe (quindi stringhe interrotte dal carattere nuova linea **\n**).

Nei File dobbiamo tenere in considerazione 3 cose:

- attributi di accesso al file (in lettura, scritture, o lettura e scrittura?)
- encoding esterno del file
- encoding interno del file

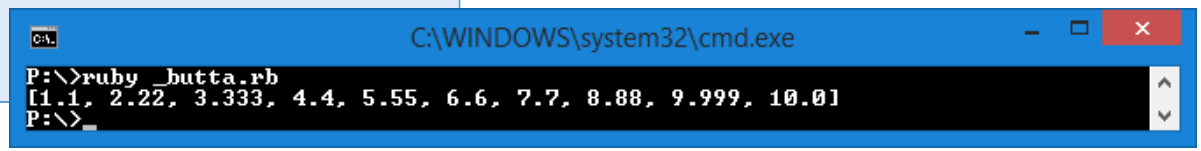
Durante la apertura di un file in Ruby, dobbiamo specificare in che modo vogliamo accedere a tale file. Per quanto riguarda l'encoding, ci basti sapere che tutti i caratteri diversi dai 256 caratteri fondamentali della tabella ASCII, sono espressi per mezzo di codici. La codifica di tale codice è chiamata **encoding**. Se un vostro collega russo vi invia un file creato sul suo computer che ha un determinato encoding, probabilmente è una buona idea specificare come *encoding esterno* quello relativo al sistema del vostro collega, mentre come *coding interno* quello relativo al nostro sistema, in modo tale da leggere correttamente il file. Se leggete file creati sullo stesso sistema, potrete essere abbastanza sicuri di non dover specificare l'encoding.

Ipotizziamo di avere un file "*file.txt*" nella stessa directory del nostro script. Ogni volta che si apre un file, bisogna ricordarsi di chiuderlo prima di terminare l'esecuzione di uno script, oppure utilizzare una funzione che lo apra e lo chiuda al posto nostro (un blocco!). Una volta aperto il file in lettura, vogliamo leggere i dati (Float) che contiene, separati da una tabulazione. Lo possiamo fare con pochissime righe di codice:.

```
# Apro in lettura.
file = File.open("file.txt", "r")
data = []
file.each_line do |line| # scorro linea per linea
   # genero un array con le sottostringhe della
   # linea che sono separati dal carattere tab
   line_data = line.chomp.split("\t")
   # convertiamo gli elementi in Float
   line_data=line_data.map {|e| e.to_f }
   # accodiamo i valori all'array data
   data << line_data
end
# dobbiamo ricordarci di chiudere il file!
file.close
# appiattisco l'array
print data.flatten.to_s
```

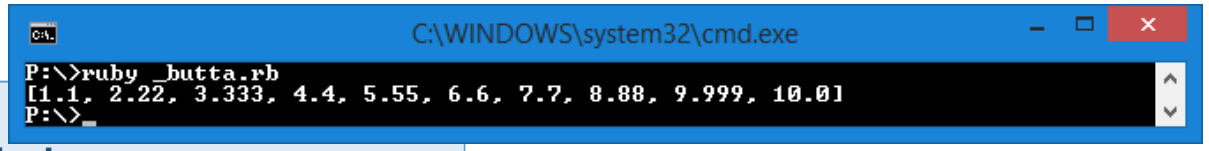
Contenuto del file "file.txt"

```
1.1    2.22   3.333
4.4    5.55
6.6
7.7    8.88   9.999  10
```



Possiamo anche evitare di aprire e chiudere il file, e lasciarlo fare alla libreria, utilizzando open come se fosse un blocco (questo metodo è da preferirsi in quanto più robusto, al fine di proteggere i dati contenuti nei File)

```
data = []
File.open("file.txt", "r") { |file|
   file.each_line { |line|
      line_data = line.chomp.split("\t")
      line_data = line_data.map {|e| e.to_f }
      data << line_data
   }
} # il file viene automaticamente chiuso
print data.flatten.to_s # appiattisco l'array
```



Contenuto del file "file.txt"

```
1.1    2.22   3.333
4.4    5.55
6.6
7.7    8.88   9.999  10
```

Un metodo di livello ancora superiore permette di racchiudere apertura e iterazione tra le linee del file in un colpo solo:

```
data=[]
File.foreach("file.txt", "r") { |allLines|
   line_data = allLines.gsub("\n","\t").split("\t")
   line_data = line_data.map {|e| e.to_f }
   data << line_data
} # ha letto l'intero files
print data.flatten.to_s # appiattisco l'array
```

Per aprire un file in scrittura, utilizziamo l'attributo di accesso "**w**" al posto di "**r**" . La classe mette a disposizione un paio di metodi molto comodi di una istanza di oggetto File:

- **`puts`** : inserisce una riga di testo e la termina con il carattere di nuova linea
- **`print`** : inserisce dei dati nel file senza aggiungere un carattere di fine linea

Per vedere come funzionano, immaginiamo di avere ancora la nostra variabile **`data`**, costituita da un Array di Float, che vogliamo scrivere in un file.

```
data=[[1.1, 2.22, 3.333], [4.4, 5.55], [6.6], [7.7, 8.88, 9.999, 10.0]]
File.open("nuovo_file.txt", "w") { |file|
   data.each { |el|
      el.each { |x|
         file.print "#{x}\t"
      }
      file.puts # inserisce un invio
   }
} #chiusura automatica del file
```

Contenuto del file "nuovo_file.txt"

```
1.1    2.22   3.333
4.4    5.55
6.6
7.7    8.88   9.999  10
```

Gestione errori

Può accadere che durante la esecuzione di un programma qualcosa vada storto! Dobbiamo scrivere il nostro codice in modo tale che sia robusto, ovvero sia in grado di gestire la sua esecuzione anche in occasione di errori.

raise

Quando l'interprete identifica un errore, esegue un **`raise`** di un errore, specificando la natura dell'errore e nel limite del possibile un messaggio che possa aiutare il programmatore a capire quale è stato l'errore.

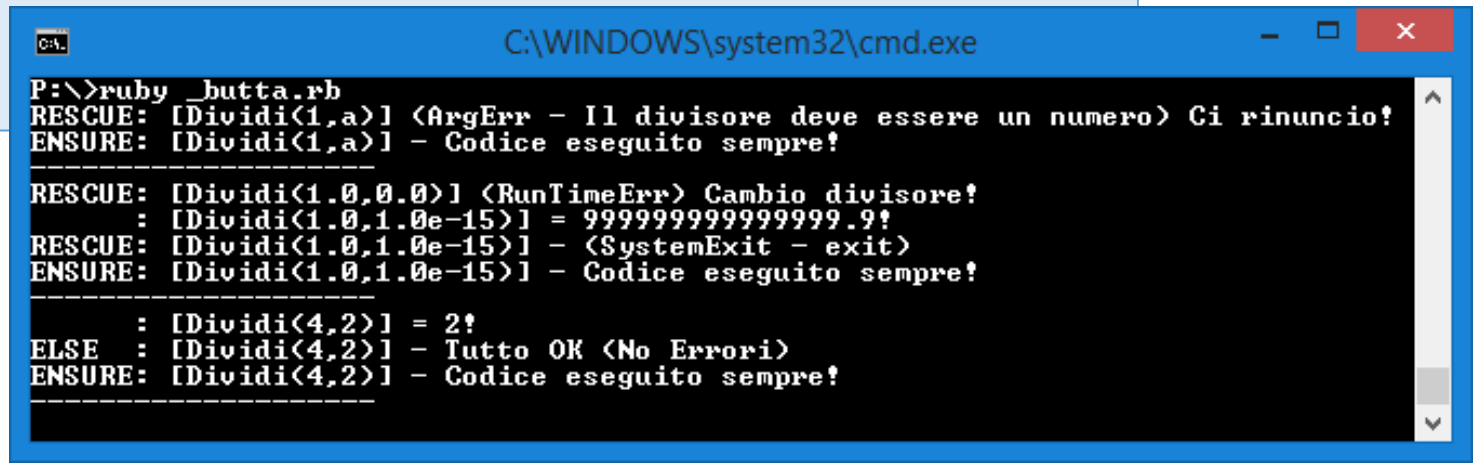
rescue

In molti casi è necessario gestire tale errore, cercando di recuperare il recuperabile: questo è il caso della keyword **`rescue`** . Il codice che segue **`rescue`** è eseguito quando un errore viene sollevato dall'interprete.

ensure

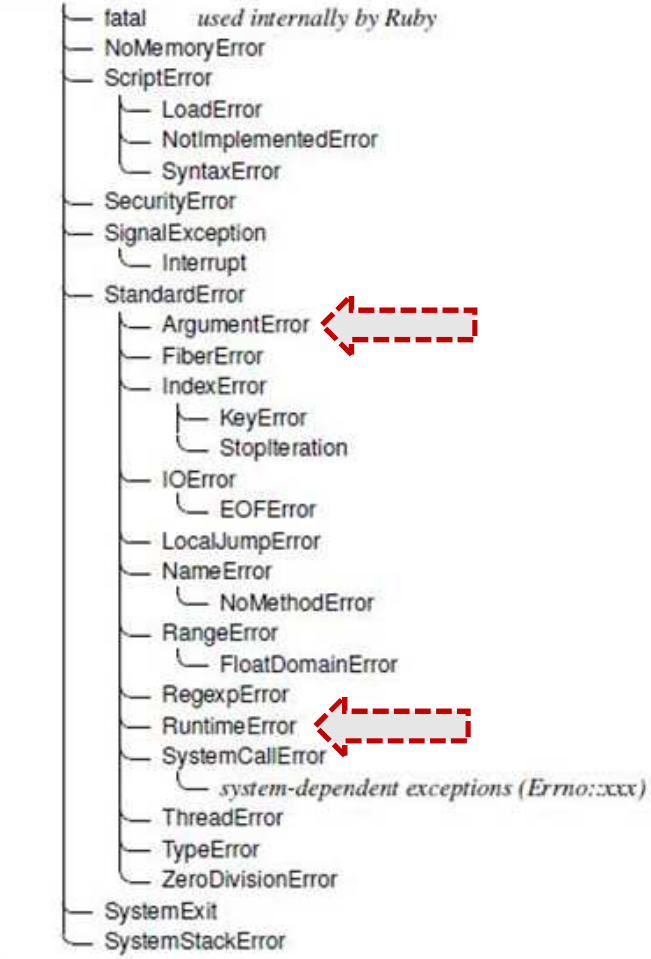
In alcuni casi, nostante i vari tentativi di recupero, non è possibile recuperare tali errori. Ancora non è tutto perduto. Dopo la keyword **`ensure`** possiamo inserire del codice che verrà eseguito nonostante sia stato generato un errore non recuperabile. Queste linee sono fondamentali per tentare di salvare i risultati dei nostri calcoli fino al punto in cui è stato generato l'errore, e devono essere utilizzati per routine estremamente robuste che sicuramente andranno a buon fine (come ad esempio la chiusura di un file che avevamo aperto, al fine di salvare i dati e non perderli).

```ruby
def dividi(dividendo, divisore)
   # Controllo sugli argomenti in ingresso: definisco una frase di errore predefinita
   raise ArgumentError, "Il dividendo deve essere un numero" if not dividendo.is_a?(Numeric)
   raise ArgumentError, "Il divisore deve essere un numero" unless divisore.is_a? Numeric

   # Controllo che il divisore sia diverso da zero
   raise RuntimeError, "Il divisore deve essere diverso da 0" if divisore == 0

   puts "      : [Dividi(#{dividendo},#{divisore})] = #{dividendo/divisore}!"
   exit if  divisore <= 10**(-15)  # in questo modo genero l'eccezione SystemExit
   # Gestione dell'errore
   rescue  RuntimeError
      puts "RESCUE: [Dividi(#{dividendo},#{divisore})] (RunTimeErr) Cambio divisore!"
      if divisore == 0 then
         divisore += 10**(-15) # il nostro epsilon
         retry # riprova ad eseguire la funzione dall'inizio, con  divisore modificato
      end
   rescue  ArgumentError=>e
      puts "RESCUE: [Dividi(#{dividendo},#{divisore})] (ArgErr - #{e.message}) Ci rinuncio!"
   rescue Exception => e # Tutti gli errori derivano dalla classe "Exception" pertanto con
                         # questa scrittura tutti gli errori verranno gestiti
      puts "RESCUE: [Dividi(#{dividendo},#{divisore})] - (#{e.class} - #{e.message})"
   else
      puts "ELSE  : [Dividi(#{dividendo},#{divisore})] - Tutto OK (No Errori)"
   ensure
      puts "ENSURE: [Dividi(#{dividendo},#{divisore})] - Codice eseguito sempre!\n"+"-"*20
end
dividi(1,"a")
dividi(1.0,0.0)
dividi(4,2)
```



Vediamo ora un esempio pratico. Supponiamo di scrivere una funzione che esegue una divisione. La funzione deve generare un errore se i due valori inseriti non sono numeri o se il divisore è nullo:

Gerarchia delle eccezioni in Ruby.



Da notare l'introduzione di due tipi diversi di errore:

- **`ArgumentError`** : l'errore è sul tipo di argomento passato alla funzione
- **`RuntimeError`** : gli argomenti sono corretti, ma quello che contiene uno degli argomenti andrà a generare un errore

Gli errori sono innalzati dall'interprete se e solo se l' **if postfisso** ritorna true . Se nessun errore viene generato, la funzione continua con la sua esecuzione classica. In generale lo schema di gestione degli errori è il seguente:

```
begin
   #.. Sequenza di istruzioni
   #.. raise exception
rescue
   # .. Gestione dell'errore
else
   #.. Sezione eseguita se non si sono
   #   verificate eccezioni tra quelle
   #   previste in raise
ensure
   #.. Questa parte verrà sempre eseguita.
end
```

La classe **Class**

…

```
class Amici
   ...
end
```

funzione

…

```
Esempio # ==> Note
```

http://www.evc-cit.info/cit020/beginning-programming/chp_02/user_input.html