Docenti: **Matteo Re** **(INFORMATICA)**
Alessandro di Domizio **(STATISTICA)**

UNIVERSITÀ DEGLI STUDI DI MILANO

C.d.l. Biotecnologia

A.A. 2016-2017 semestre II

Informatica e Statistica

3 Linguaggi programmazione

Modulo: **INFORMATICA**

Definizione intuitiva di algoritmo

Un **algoritmo** si può definire come un *procedimento* che consente di *ottenere* un *risultato* eseguendo, in un determinato ordine, un insieme di *passi semplici* corrispondenti ad azioni scelte solitamente da un insieme finito.

Esempi:

- Una procedura per il calcolo del minimo comune multiplo fra due numeri naturali
- La ricetta di una torta
- Una procedura per ordinare un insieme di oggetti in base ad un criterio.

Il termine algoritmo deriva dall'arabo

Abū Ja-far Muhammad ibn Mūsā Khwārizmī - in arabo:
محمد خوارزمی - Baghdad, IX secolo matematico, astronomo, e geografo persiano.

Una pagina dall' *Algebra* di al- Khwārizmī

Caratteristiche degli algoritmi

- La sequenza di istruzioni deve essere finita (*finitezza*);
- La procedura deve portare ad un risultato (*effettività*);
- Le istruzioni devono essere eseguibili materialmente (*realizzabilità*);
- Le istruzioni devono essere espresse in modo non ambiguo (*non ambiguità*).

Problemi e algoritmi

Un algoritmo risolve un *problema* (come funzione di corrispondenza fra spazio delle istanze e delle soluzioni)

Esempio:

Problema: Allineamento di due sequenze di DNA

Istanze: Coppie di sequenze

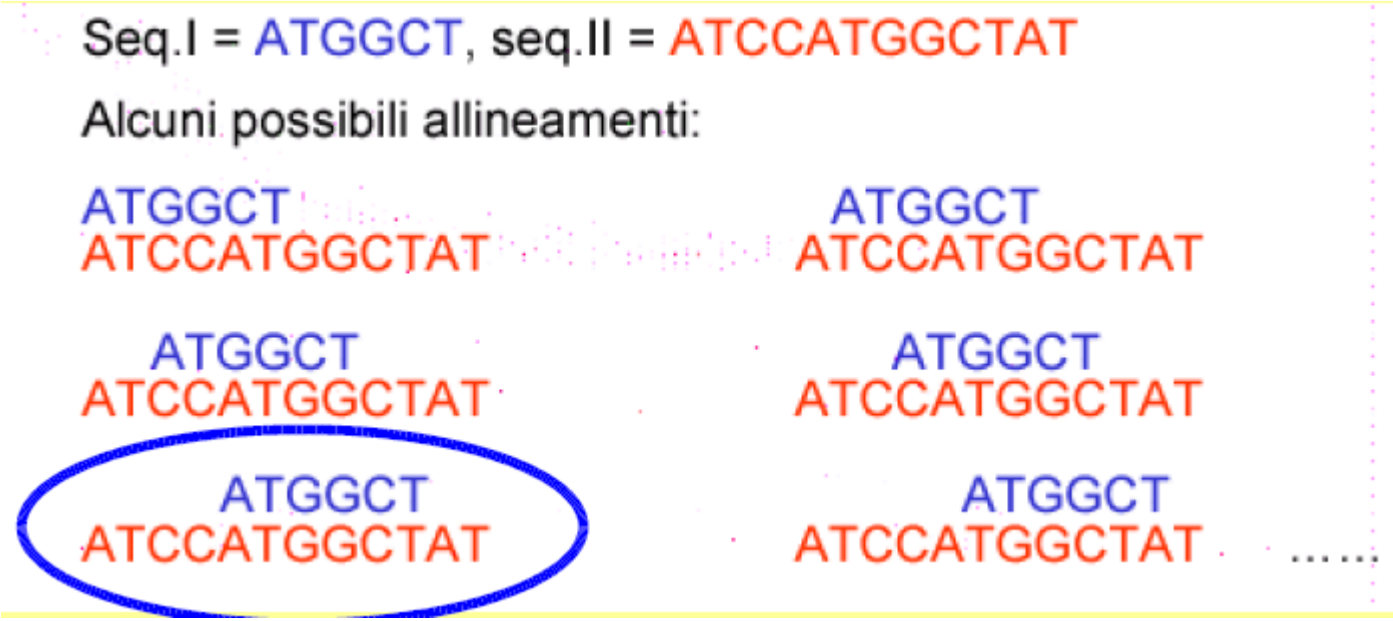
Soluzioni: Giustapposizioni delle due sequenze (che massimizzino la similarita')

Es: algoritmo di allineamento fra due sequenze

Input: 2 sequenze Seq.I e Seq.II

1. Confrontare le sequenze (partendo dal I nucleotide di ogni sequenza)
2. Assegnare un punteggio all'allineamento sulla base di criteri fissati per la similarità
3. Ripetere l'operazione, "facendo scorrere" in tutti i modi possibili una sequenza rispetto all'altra

Output: allineameno con il punteggio massimo



Complessità dell'algoritmo elementare di allineamento

- L' algoritmo fa scorrere ciclicamente una sequenza sull'altra, spostandosi ad ogni ciclo di una posizione e verificando quante posizioni hanno un'identità.
- Per ogni ciclo si devono verificare tutte le posizioni, quindi alla fine dovremo fare un numero di verifiche pari al numero di cicli per numero di posizioni: *più o meno dell'ordine del prodotto delle lunghezze delle due sequenze*.
- Se la lunghezza dele due sequenze e' di n nucleotidi, si dovranno quindi effettuare n^2 verifiche (operazioni elementari): *la complessità dell'algoritmo è quindi quadratica* rispetto alla lunghezza delle sequenze di ingresso.

Ogni algoritmo e' caratterizzato da una complessità temporale e spaziale rispetto alle dimensioni dei dati di ingresso.

Linguaggi di programmazione: strumenti per comunicare ad una macchina come risolvere un problema.

- Strumenti per la comunicazione uomo-macchina
- Permettono di esprimere e rappresentare *programmi = algoritmi + strutture dati* comprensibili ed eseguibili da una macchina

Caratteristiche dei linguaggi di programmazione

- Sono analoghi ai linguaggi naturali, con la differenza che vengono usati per comunicare con una macchina

Come i linguaggi naturali sono caratterizzati dalle seguenti componenti:

• Insieme di simboli (*alfabeto*) e di parole (*dizionario*) che possono essere usati per formare le frasi del linguaggio

• Insieme delle regole grammaticali (*sintassi*) per definire le frasi corrette composte dalle parole del linguaggio

• Significato (*semantica*) delle frasi del linguaggio

• Per utilizzare correttamente un linguaggio è necessario conoscerne la *pragmatica* (ad es: quali frasi è opportuno usare a seconda del contesto).

• *I linguaggi di programmazione, a differenza di linguaggi naturali, non devono essere ambigui e devono essere formalizzati* (definiti in maniera non equivocabile).

Linguaggi macchina

- Linguaggi immediatamente comprensibili per una macchina:
  - Istruzioni e dati sono sequenze di numeri binari
  - Le istruzioni operano direttamente sull' hardware (registri, locazioni di memoria, unità fisiche di I/O del calcolatore)
  - Sono specifici per un determinato processore o famiglia di processori
  - Assumono il modello computazionale di Von Neumann

Esempio:

Calcolo della somma S di due numeri A e B

Linguaggio macchina	Linguaggio assembly
00000010101111001010	LOAD A
00000010111111001000	ADD B
00000011001110101000	STORE S

Un *linguaggio assembly* è la forma simbolica di un linguaggio macchina: si usano nomi al posto dei codici binari per le operazioni e locazioni di memoria delle macchine.

Linguaggi a basso ed alto livello

- Linguaggi assembly e macchina sono *linguaggi a basso livello*
- *I linguaggi ad alto livello* permettono di scrivere programmi con un linguaggio piu' vicino a quello naturale

Esempio:

"Stampa sullo schermo la somma fra C ed il prodotto di A e B":

Linguaggio ad alto livello (C++):

```
cout << A * B + C;
```

Linguaggio Assembly:

```
mov eax,A
mul B
add eax,C
call WriteInt
```

Linguaggio macchina:

```
A1     00000000
F7 25 00000004
03 05 00000008
E8 00500000
```

Esempio: funzione per il calcolo della media in C ed in linguaggio assembly

Linguaggio C (alto livello)

```
double mean (double* x, unsigned n)
{
  double m = 0;
  int i;
  for (i=0; i<n; i++)
     m += x[i];
  m /= n;
  return m;
}
```

Linguaggio Assembly

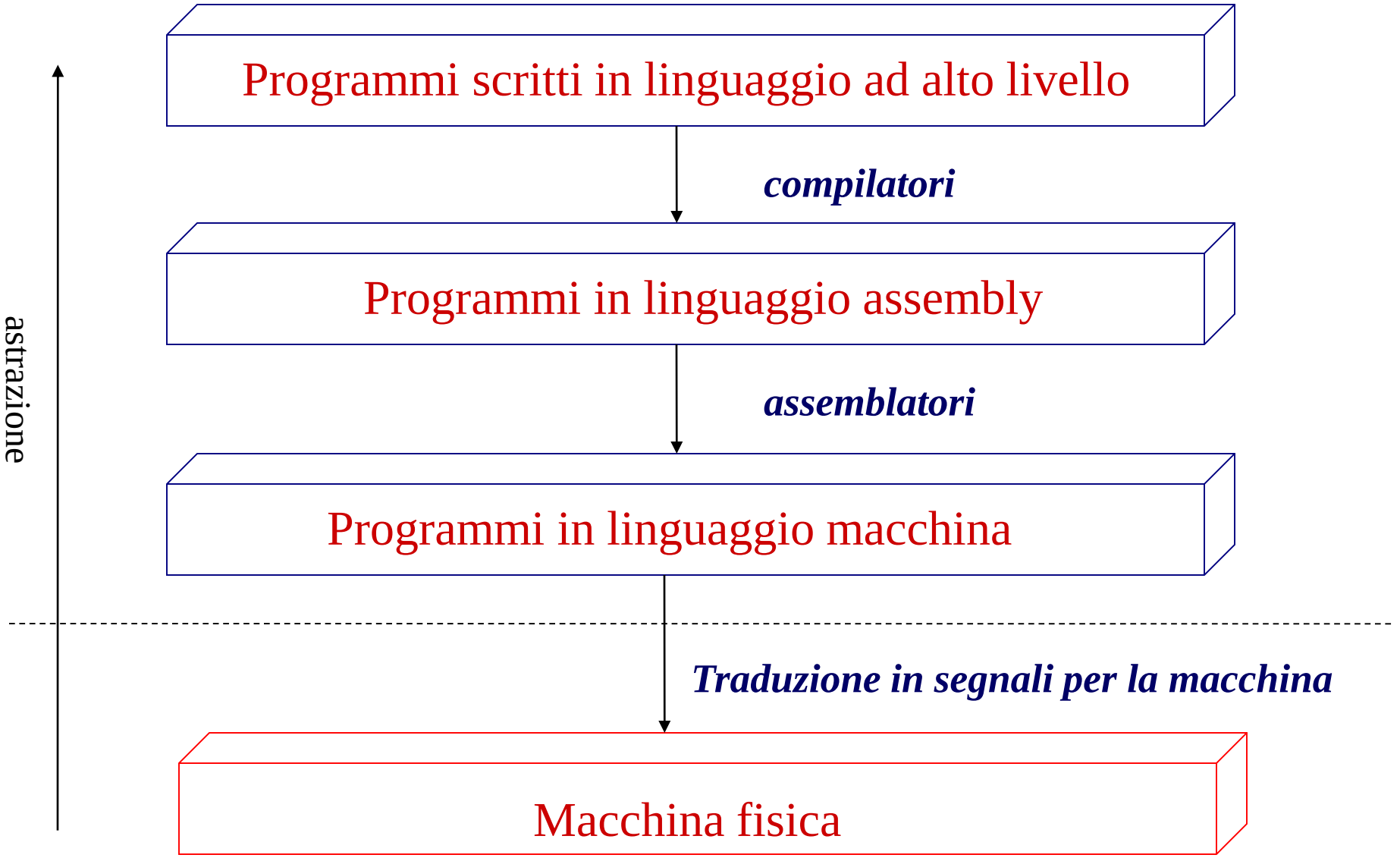
```
.file       "qq.c"
.text
.globl mean
.type       mean,@function
mean:
pushl       %ebp
movl        %esp, %ebp
subl        $24, %esp
movl        $0, -8(%ebp)
movl        $0, -4(%ebp)
movl        $0, -12(%ebp)
.L2:
movl        -12(%ebp), %eax
cmpl        12(%ebp), %eax
jb          .L5
jmp         .L3
.L5:
movl  -12(%ebp), %eax
leal  0(,%eax,8), %edx
movl  8(%ebp), %eax
fldl  -8(%ebp)
faddl (%eax,%edx)
fstpl -8(%ebp)
leal  -12(%ebp), %eax
incl  (%eax)
jmp   .L2
.L3:
movl  12(%ebp), %eax
movl  $0, %edx
pushl %edx
pushl %eax
fildll      (%esp)
leal  8(%esp), %esp
fldl  -8(%ebp)
fdivp %st, %st(1)
fstpl -8(%ebp)
movl  -8(%ebp), %eax
movl  -4(%ebp), %edx
movl  %eax, -24(%ebp)
movl  %edx, -20(%ebp)
fldl  -24(%ebp)
leave
ret
.Lfe1:
.size mean,.Lfe1-mean
.ident      "GCC: (GNU) 3.2.2
```

Linguaggi ad alto livello

- Sono definiti *astraendo* rispetto alla macchina fisica
- Realizzano una *macchina virtuale* soprastante alla macchina fisica e visibile al programmatore
- Richiedono di essere implementati su un particolare sistema di calcolo tramite strumenti opportuni (*compilatori* o *interpreti*)

Esempi: *fortran*, *C*, *lisp*, *java*, *R*

Livelli di rappresentazione e macchine astratte



Linguaggi compilati e linguaggi interpretati

- I *programmi compilati* vengono tradotti completamente dalla prima all' ultima istruzione nel linguaggio macchina del sistema sottostante (netta distinzione fra compile-time e run-time).
  Es: programmi in *C*, *fortran*, *C++*
- I *programmi interpretati* vengono tradotti ed eseguiti immediatamente riga per riga (l' interprete simula una macchina astratta, no distinzione netta fra compile-time e run-time).
  Es: programmi in R
- Esistono casi "intermedi": es: *java*.
- I *compilatori* e gli *interpreti* sono i programmi che effettuano la traduzione dal linguaggio ad alto livello al linguaggio macchina

Linguaggi di programmazione e produzione del software

Modello tradizionale “a cascata” per la produzione del sw:

- Analisi e specificazione dei requisiti
- Progetto (design) del sistema
- Implementazione: Produzione del codice nel linguaggio prescelto
- Verifica e validazione
- Manutenzione

In realtà il processo di produzione è ciclico.

Linguaggi di programmazione e ambienti di sviluppo

Ogni fase dello sviluppo del sw può essere supportato da *ambienti di sviluppo*.

Ambienti di sviluppo per l'implementazione del sw:

- Text editor
- Compilatori
- Linker
- Librerie
- Debugger
- ...

Linguaggi di programmazione per la bioinformatica

- In linea di principio qualsiasi linguaggio ad alto livello può essere utilizzato.
- Esistono comunque *linguaggi con librerie e package specifici* specializzati per la bioinformatica:

Progetti Open Source

  - *Perl* e *BioPerl*: http://bioperl.org
  - *Python* e *Biopython*: http://biopython.org
  - *Java* e *BioJava*: http://biojava.org
  - *R* e *Bioconductor*: http://www.bioconductor.org

  - Matlab e toolbox per la bioinformatica

Docenti: **Matteo Re** **(INFORMATICA)**
Alessandro di Domizio **(STATISTICA)**

UNIVERSITÀ DEGLI STUDI DI MILANO

C.d.l. Biotecnologia

A.A. 2016-2017 semestre II

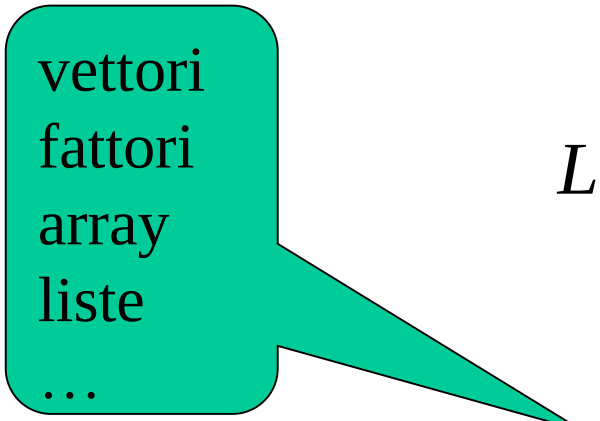
Informatica e Statistica

4 Programmi e funzioni in R

Modulo: **INFORMATICA**

Programmi in R

Strutture dati + Algoritmi = Programmi



Linguaggio R

- Strutture + Funzioni
- Classi + metodi

Modo di esecuzione dei programmi in R

I programmi (sequenze di espressioni) possono essere eseguiti :

Interattivamente: ogni istruzione viene eseguita direttamente al prompt dei comandi

Non interattivamente: le espressioni sono lette da un file (tramite la funzione *source*) ed eseguite dall' interprete una ad una in sequenza.

Usare un text editor (ad esempio *Notepad++*, scaricabile gratuitamente dalla rete)

Esempio di script R

```
# Functional classification of yeast genes using gene expression data
library(yeastCC);
library(e1071);
data(spYCCES);
source("yeastGO.R");

# data preparation
Yeast.specific.TAS <- Get.yeast.GO.specific.classes(evidence="TAS");
Yeast.general.TAS <- Get.yeast.GO.all.classes(Yeast.specific.TAS);
load("Yeast.general.classes.TAS.object");
l <- Count.examples.per.class(Yeast.general.classes.TAS);
cl1.genes <- Extract.class(Yeast.general.classes.TAS,"GO:0000902"); cl2.genes <- Extract.class(Yeast.general.classes.TAS,"GO:0006092");
paste("GO:0000902", ":", get.Term.Definition("GO:0000902")); paste("GO:0006092", ":", get.Term.Definition("GO:0006092"));
exprs.cl1 <- exprs(spYCCES)[as.character(cl1.genes$gene.names),]; exprs.cl2 <- exprs(spYCCES)[as.character(cl2.genes$gene.names),];
exprs.cl1[is.na(exprs.cl1)] <- 0; exprs.cl2[is.na(exprs.cl2)] <- 0;

# classification of yeast genes
n1.test<-round(nrow(exprs.cl1)/3); n1.train<-nrow(exprs.cl1)-n1.test;
n2.test<-round(nrow(exprs.cl2)/3); n2.train<-nrow(exprs.cl2)-n2.test;
Xtrain<-rbind(exprs.cl1[1:n1.train,],exprs.cl2[1:n2.train,]);
Xtest<-rbind(exprs.cl1[(n1.train+1):nrow(exprs.cl1),],exprs.cl2[(n2.train+1):nrow(exprs.cl2),]);
ytrain<-as.factor(c(rep(1,n1.train),rep(2,n2.train)));
ytest<-as.factor(c(rep(1,n1.test),rep(2,n2.test)));
model <- svm(as.matrix(Xtrain),ytrain,type="C-classification", kernel="linear", cost=1, gamma=1, degree=2, coef0=1);
prediction <- predict(model,Xtest);
conf.matrix <- table(prediction,ytest);
sensitivity <- conf.matrix[1,1]/(conf.matrix[1,1]+conf.matrix[2,1]);
specificity <- conf.matrix[2,2]/(conf.matrix[2,2]+conf.matrix[1,2]);
accuracy <- (conf.matrix[1,1] + conf.matrix[2,2])/length(ytest);
```

Funzioni

Abbiamo già visto molti esempi di funzioni disponibili in R

Le funzioni in R possono anche definite dagli utenti

I programmi in R possono essere realizzati tramite funzioni

Funzioni: sintassi

La sintassi per scrivere una funzione è:

`function (argomenti)  corpo_della_funzione`

- `function` è una parola chiave di R
- `Argomenti` è una lista eventualmente vuota di *argomenti formali* separati da virgole:
  `(arg1, arg2, ..., argN)`

Un *argomento formale* può essere un simbolo o un'istruzione del tipo '`simbolo=espressione`'

Il `corpo` può essere qualsiasi espressione valida in R. Spesso è costituito da un gruppo di espressioni racchiuso fra parentesi graffe

Funzioni: esempi (1)

```
# Funzione per il calcolo della statistica di Golub
# x,y : vettori di cui si vuole calcolare la statistica di golub
# La funzione ritorna il valore della statistica di Golub
```

argomenti

```
golub <- function(x,y)   {
  mx <- mean(x);
  my <- mean(y);
  vx <- sd(x);
  vy <- sd(y);
  g <- (mx-my)/(vx+vy);
  return(g);
}
```

La sequenza di istruzioni del corpo della funzione deve essere racchiusa fra parentesi quadre

Funzioni: esempi (2)

Utilizzo della funzione di Golub:

La funzione golub è memorizzata nel file “golub.R” (ma potrebbe essere memorizzata in un file con nome diverso)

Caricamento in memoria della funzione. Due possibilità:

1. `> source("golub.R")`
2. Dal menu File/Source R code ...

Chiamata della funzione:

```
> x<-runif(5) # primo argomento della funzione
> x
[1] 0.6826218 0.9587295 0.4718516 0.8284525 0.2080131
> y<-runif(5) # secondo argomento della funzione
> y
[1] 0.6966353 0.0964740 0.4310154 0.1467449 0.2801970
> golub(x,y) # chiamata della funzione
[1] 0.5553528
```

Argomenti formali e attuali

x e y sono *argomenti formali*:

```
> golub <- function(x,y)  { ... }
```

Tali valori vengono sostituiti dagli *argomenti attuali* quando la funzione è chiamata:

```
> d1 <- runif(5)
> d2 <- runif(5)
```

`d1` e `d2` sono gli argomenti attuali che sostituiscono i formali e vengono effettivamente utilizzati all' interno della funzione:

```
> golub(d1,d2)
[1] 0.2218095
> d3 <- 1:5
> golub(d1,d3)
[1] -1.325527
```

Gli argomenti sono passati per valore

Le modifiche agli argomenti effettuate nel corpo delle funzioni non hanno effetto all' esterno delle funzioni stesse:

```
> fun1 <- function(x) { x <- x*2 }
> y<-4
> fun1(y)
> y
[1] 4
```

In altre parole i valori degli argomenti attuali sono modificabili all' interno della funzione stessa, ma non hanno alcun effetto sulla variabile dell' ambiente chiamante.

Nell' esempio precedente la copia di `x` locale alla funzione viene modificata, ma non viene modificato il valore della variabile `y` passata come argomento attuale alla funzione `fun1`

Modalità di assegnamento degli argomenti: assegnamento posizionale

Tramite questa modalità gli argomenti sono assegnati in base alla loro posizione nella lista degli argomenti:

```
> fun1 <- function (x, y, z, w) {}
> fun1(1,2,3,4)
```

L' argomento attuale 1 viene assegnato a *x*, 2 a *y*, 3 a *z* e 4 a *w*.

Altro esempio:

```
>  sub <- function (x, y) {x-y}
> sub(3,2) # x<-3 e y<-2
[1] 1
> sub(2,3) # x<-2 e y<-3
[1] -1
```

Modalità di assegnamento degli argomenti: assegnamento per nome

Tramite questa modalità gli argomenti sono assegnati in base alla loro nome nella lista degli argomenti:

```
> fun1 <- function (x, y, z, w) {}
> fun1(x=1, y=2, z=3, w=4)
```

L' argomento attuale 1 viene assegnato a *x*, 2 a *y*, 3 a *z* e 4 a *w*.

Quando gli argomenti sono assegnati per nome non è necessario rispettare l' ordine degli argomenti:

```
fun1(y=2, w=4, z=3, x=1) ≡ fun1(x=1, y=2, z=3, w=4)
```

Ad esempio:

```
>  sub <- function (x, y) {x-y}
> sub(x=3,y=2) # x<-3 e y<-2
[1] 1
> sub(y=2,x=3) # x<-3 e y<-2
[1] 1
```

Valori di default per gli argomenti

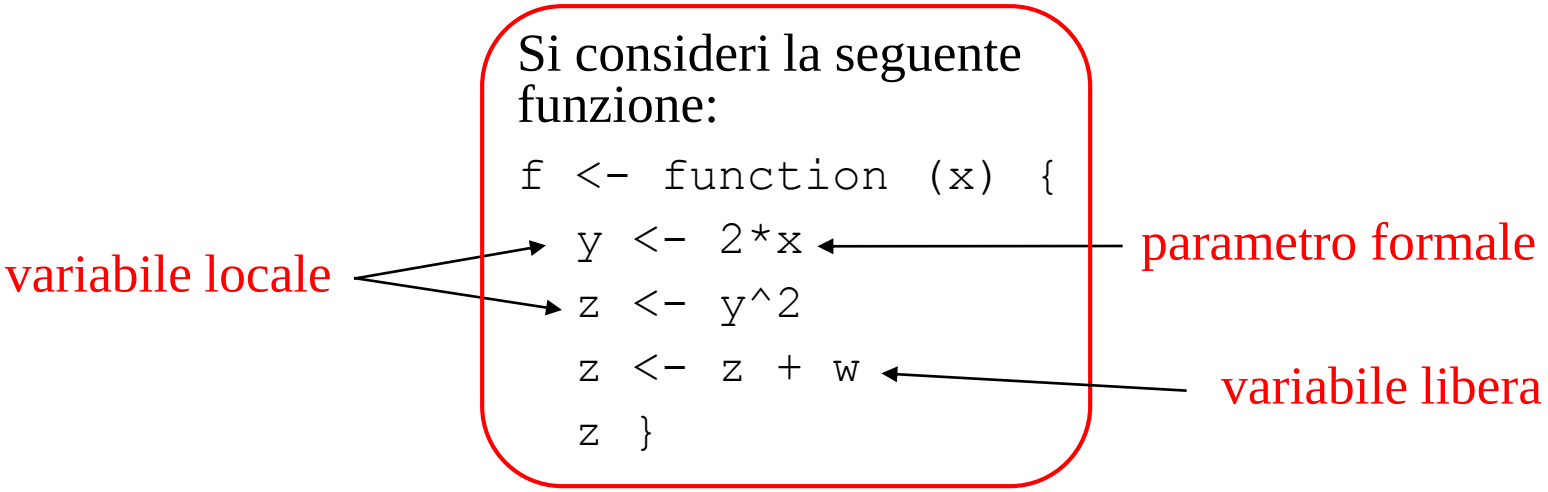
E' possibile stabilire valori predefiniti per tutti o per parte degli argomenti: tali valori vengono assunti dalle variabili a meno che non vengano esplicitamente modificati nella chiamata della funzione.

Esempio:


```
> fun4 <- function (x, y, z=2, w=1) {x+y+z+w}
> fun4(1,2) # x<-1, y<-2, z<-2, w<-1
[1] 6
> fun4(1,2,5) # x<-1, y<-2, z<-5, w<-1
[1] 9
> fun4(1) # y non ha valore di default !
Error in fun4(1) : Argument "y" is missing, with no
   default
```

Parametri formali, variabli locali e variabili libere

Le variabili che non sono nè parametri formali e nè variabili locali sono chiamate variabili libere.
Il binding delle variabili libere viene risolto cercando la variabile nell' ambiente in cui la funzione è stata creata:




```
> f(3)
Error in f(3) : Object "w" not found
> w <- 3
> f(3)
[1] 39
```

L'operatore di "superassegnamento"

Il passaggio dei parametri alle funzioni avviene per valore.

Tramite l' operatore di superassegnamento '<<-' è però possibile modificare il valore della variabile nell' ambiente di livello superiore

```
f <- function (x)
{
  y <- x/2;
  z <- y^2;
  x <<- z-1;
}
```

superassegnamento →

Quando la funzione `f` viene chiamata il valore della variabile `x` viene modificato:

```
> x=1; f(x)
> x
[1] -0.75
```

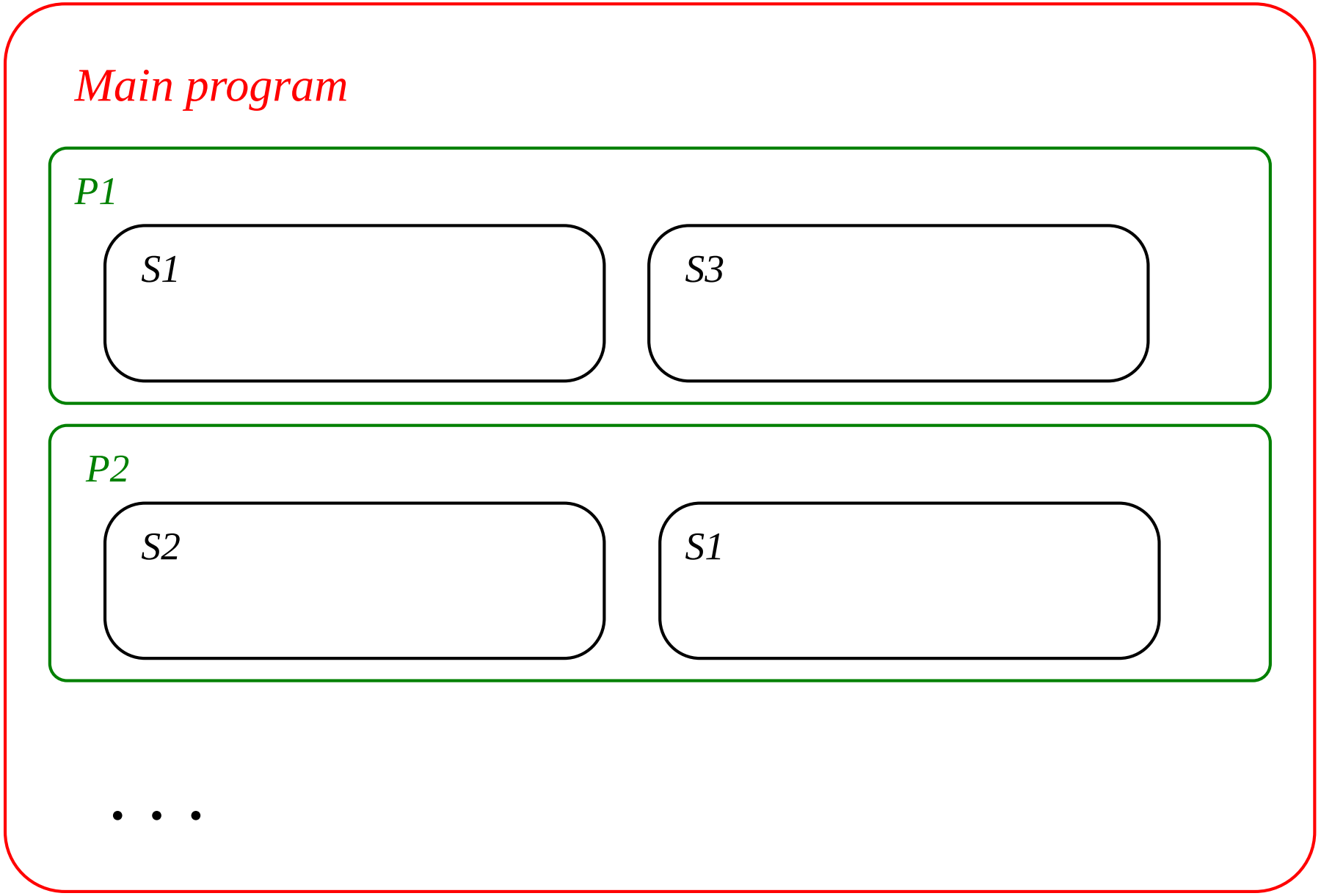
Se la variabile `x` non viene trovata nell' ambiente top-level, `x` viene creata e le viene assegnato il valore calcolato dalla funzione:

```
> rm(x); f(1)
> x
[1] -0.75
```

Programmazione modulare

Le funzioni R possono richiamare altre funzioni, permettendo in tal modo di strutturare i programmi in modo “gerarchico”:

```
# funzioni di "secondo livello" chiamate dalla funzioni
# P1 e P2

S1 <- function (x) {... }

S2 <- function () {... }

S3 <- function () {... }


# funzioni di primo livello" chiamate dalla funzione
# principale

P1 <- function (x) { S1(x); S3(); }

P2 <- function (x) { S2(); S1(x); ... }


# funzione principale del programma R

MainProgram <- function(x,y,z) {P1(x); P2(y); P1(z) ... }
```


Main program
P1
S1
S3
P2
S2
S1
. . .

Programmazione top-down

La programmazione modulare consente di affrontare i problemi “dall’ alto al basso” (approccio *top-down*), cercando cioè di partire dal problema principale definito come una funzione (MainProgram nell’ esempio precedente) con determinati ingressi (dati del problema che si vuole risolvere) ed uscite (risposte/soluzioni al problema)

Dal problema principale si cerca poi di individuare un insieme di sottoproblemi tramite cui sia possibile risolvere il problema principale; i sottoproblemi sono risolti tramite le funzioni P1 e P2.

A loro volta i sottoproblemi P1 e P2 si possono essere scomposti in sotto-sottoproblemi (implementati tramite le funzioni S1, S2, S3)

Il processo di scompozione dei problemi “dall’ alto al basso” può proseguire ancora o arrestarsi a seconda della tipologia del problema.

In generale tale approccio non è lineare, ma richiede raffinamenti successivi

R consente anche altri tipi di approcci al design del software (ad es: approccio bottom-up, object-oriented)

Esercizi (I)

1. Scrivere una funzione *compute.mean.var* che, avendo come argomento una lista di vettori numerici, calcoli la media e la varianza per ciascun elemento della lista.
2. Scrivere una funzione find.stop.codon che, ricevuto come argomento un vettore di “triplette” del codice genetico ritorni un messaggio “codon di stop trovato” o “codon di stop non trovato” a seconda che una delle triplette “UAA”, “UAG” o “UGA” sia presente o meno nel vettore di ingresso.
3. Scrivere una funzione find.codon che, ricevuto in ingresso un vettore di “triplette” del codice genetico ed una tripletta codon arbitraria, stampi sullo schermo un messaggio di codon trovato e la sua posizione, o un messaggio di codon non trovato.
   In caso pero’ incontri prima un codon di stop deve stampare un messaggio di stop codon trovato, la sua posizione e terminare.

Esercizi (II)

4. Scrivere una funzione *analyze_string* che ricevuto in ingresso una stringa arbitraria calcoli la frequenza dei simboli componenti la stringa stessa
5. Scrivere una funzione che calcoli i numeri di Fibonacci
6. Scrivere un programma *analyze* che calcoli alcune semplici statistiche relative a 5 diverse tipologie di analisi. In particolare analyze deve:
    1. Leggere da un file una matrice con un numero arbitrario di righe (ogni riga rappresenta un campione) e con 5 colonne che rappresentano dati numerici relativi a 5 diverse analisi.
    2. Trasformi la matrice in un data frame con variabili var1,var2,...,var5.
    3. Memorizzi il data frame in un file
    4. Per ogni variabile calcoli media, deviazione standard.
    5. Stampi sullo schermo i valori relativi a media e deviazione standard per ogni variabile
7. Scrivere una funzione *CalcCovCor* che calcoli le matrici di covarianza e di correlazione fra *n* variabili i cui valori siano generati casualmente. La funzione deve permettere di specificare il numero delle realizzazioni (campioni) generati casualmente ed il tipo di generazione (secondo la distribuzione uniforme o gaussiana). Le matrici vanno poi memorizzate in 2 diversi file (i cui nomi devono essere specificati dall' utente).

Docenti: **Matteo Re** **(INFORMATICA)**
Alessandro di Domizio **(STATISTICA)**

UNIVERSITÀ DEGLI STUDI DI MILANO

C.d.l. Biotecnologia

A.A. 2016-2017 semestre II

Informatica e Statistica

5 Leggere/scrivere dati da/su file

Modulo: **INFORMATICA**

Lettura e scrittura di dati da file

I dati utilizzati in bioinformatica sono usualmente di *grandi dimensioni* (ad es: file PDB che memorizzano la struttura tridimensionale delle proteine, file per la memorizzazione di dati di espressione genica, etc)

Oggetti di grandi dimensioni sono usualmente memorizzati in *file esterni su memoria di massa*

In R esistono diverse *funzioni di I/O* per la lettura e scrittura di file

Esistono anche funzioni e facility per importare/esportare dati verso altri ambienti/linguaggi di programmazione

Per maggiori dettagli si consulti il manuale *R Data Import/Export* disponibile on-line

Caricare e salvare oggetti in formato binario

Caricare e salvare oggetti arbitrari in formato binario:

Salvare oggetti in formato binario:

```
> x <- runif(20);
> y <- list(a = 1, b = TRUE, c = "oops");
> save(x, y, file = "xy.Rdata");
> rm(x,y)
> x
Errore: oggetto "x" non trovato
```

Caricare oggetti in formato binario:

```
> load("xy.Rdata");
> ls()
[1] "x"    "y"
```

Caricare e salvare oggetti relativi ad un'intera sessione di lavoro:

```
> save.image();
> load(".RData");
```

Scrittura su file di data frame

La funzione write.table *memorizza un data frame in un file.*

Sintassi: write.table (*x*, file="*data*")

data è il nome del file su cui verrà scritto il data frame *x*.

La funzione write.table possiede molti altri argomenti che permettono di modularne opportunamente la semantica.

```
Esempio:
> m1 <-matrix(1:12,nrow=2); v <- c("A","C")
> daf3<-data.frame(m1,v); daf3
  X1 X2 X3 X4 X5 X6 v
1  1  3  5  7  9 11 A
2  2  4  6  8 10 12 C
> write.table(daf3,file="data.df") # memorizza nel file
# "data.df" il data frame daf3
```

Lettura di data frame da file

La funzione read.table *legge un file memorizzato su disco, inserendo i dati direttamente in un data frame*.

Il file esterno deve essere memorizzato nel modo seguente:

- La prima riga del file deve avere un nome per ciascuna variabile del data frame
- Le righe successive del file memorizzano le osservazioni che saranno memorizzate nel data frame
- Ciascuna di queste righe può avere come primo valore l' etichetta di riga (che sarà memorizzata nel' attributo row.names del data frame)
- Ciascun valore sulla riga è separato da un blank (spazio, tabulazione, etc)
- Possono essere selezionati altri separatori
- read.table dispone di molti altri parametri che si possono settare per esigenze particolari (vedi help).

Lettura di data frame da file: esempi

Il seguente data frame è memorizzato sul file “data.df”:

```
  X1 X2 X3 X4 X5 X6 v
1  1  3  5  7  9 11 A
2  2  4  6  8 10 12 C
```

La lettura viene effettuta tramite la funzione read.table:

```
daf4<-read.table("data.df")
> daf4
  X1 X2 X3 X4 X5 X6 v
1  1  3  5  7  9 11 A
2  2  4  6  8 10 12 C
```

Il file può naturalmente essere generato da altri programmi (purchè in ASCII), ad es: tramite un qualsiasi text editor, ed essere letto tramite read.table.

Lettura e scrittura di data frame : esempi

Sia read.table, sia write table possono avere altri argomenti opzionali:

```
> m1 <-matrix(1:12,nrow=2); v <- c("A","C")
> daf3<-data.frame(m1,v)
> write.table(daf3,file="data.df",col.names=paste("col",1:7,sep=""))
> read.table("data.df")
     col1 col2 col3 col4 col5 col6 col7
1    1    3    5    7    9   11    A
2    2    4    6    8   10   12    C
>  write.table(daf3,file="data.df",sep = ",") # file memorizzato
# utilizzando la virgola come separatore: controllare con un editor
>  read.table("data.df",sep=",")
  X1 X2 X3 X4 X5 X6 v
1  1  3  5  7  9 11 A
2  2  4  6  8 10 12 C
```

Funzioni generali per lettura/scrittura di file

In R sono presenti diverse funzioni generali per lettura e scrittura di file in formato ASCII o binario.

Ad es: la funzione file può aprire, creare o chiudere file e più in generale *connessioni*: ad es: file in scrittura e/o lettura, connessioni di rete tramite socket o descritte da URL.

Ci occuperemo solo dell' insieme di funzioni per la scrittura/lettura di file.

Scrittura di file: esempio

```
>  ff <- file("ex.data", "w")  # apertura di un file in
   scrittura
>      cat("TITLE extra line", "2 3 5 7", "", "11 13 17",
   file = ff, sep = "\n") # scrittura d 4 linee di testo
>      cat("One more line\n", file = ff)
>      close(ff) # chiude la connessione al file
>      readLines("ex.data") # lettura delle righe dal file
[1] "TITLE extra line" "2 3 5 7"           ""
   "11 13 17"          "One more line"
>      unlink("ex.data")  # cancella il file dal disco
```

Per scrivere dati su file si può usare anche la funzione write (utilizzata usualmente per scrivere matrici)

Le funzioni di I/O si possono usare anche per il download/upload di file in rete:

```
x <- readLines("http://homes.dsi.unimi.it/~valenti/DATA/MICROARRAY-
DATA/Leukemia/Readme.Leukemia");
```

Lettura di file: esempio

```
> ff <- file("ex.data", "r") # apertura file in lettura
>  readLines(ff) # lettura di tutto il file
[1] "TITLE extra line" "2 3 5 7"          ""                  "11 13
   17"          "One more line"
> seek(ff,0) # "rewind" del file
[1] 54
>  readLines(ff,n=1) # lettura d una riga alla volta
[1] "TITLE extra line"
>  readLines(ff,n=1)
[1] "2 3 5 7"
>  readLines(ff,n=1)
[1] ""
>  readLines(ff,n=1)
[1] "11 13 17"
>  readLines(ff,n=1)
[1] "One more line"
>  readLines(ff,n=1)  # esaurite le righe del file
character(0)
```

La funzione scan

La funzione scan legge un file di input e memorizza i dati in un vettore o una lista.

Esempi:

A. Memorizzazione dati in un vettore

```
> x <- matrix(1:10, nrow=2)
> write (x, "data")
   # scrittura della matrice
   # su file
> xread <- scan ("data",0)
Read 10 items
> xread
 [1]  1  2  3  4  5  6  7  8
   9 10
```

B. Memorizzazione dati in una lista

Si supponga di avere un file "data" composto dalle seguenti linee:

```
A    0.1  0.2  Q
B    0.5  0.4  M
A    1.1  1.2  Q
Q    0.3  0.9  P
> inp <-
   scan("data",list("",0,0,""))
   # lettura file e memorizzazione
   # in una lista: si noti la
   # lettura "per colonne"
Read 4 records
> inp
[[1]] "A" "B" "A" "Q"
[[2]]  0.1 0.5 1.1 0.3
[[3]]  0.2 0.4 1.2 0.9
[[4]] "Q" "M" "Q" "P"
```

Accesso a data set built-in

Molti data set sono disponibili con R (data set built-in) ed altri sono contenuti nei package.

Per listare i data set built-in si utilizza la funzione data().

Per caricare un data set built-in la sintassi è:
> data (nome-data-built-in)

Esempio:

```
> data(iris)
> iris
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
3            4.7         3.2          1.3         0.2     setosa
.....
```

Editing dei dati

E’ possibile utilizzare la funzione edit per effettuare cambiamenti “manuali” su matrici e data frame

E’ possibile anche utilizzare la funzione edit per costruire ex novo nuove matrici e data frame

La funzione edit fornisce un ambiente di editing simile a quello di un foglio elettronico

Esempi:

```
> edit(iris) # editing di un data frame
# esistente

> new.data.frame <- edit (data.frame())
# creazione di un nuovo data frame
```

Importare, esportare file in Excel

A. Usare write.table e read.table e le funzioni di conversione di Excel:

```
> data(iris)
> write.table(iris, file="iris.txt",
row.names=F)
```

Aprire “iris.txt” con Excel ed utilizzare le conversioni formato.
Per salvare un file Excel usare “Salva formato testo con tabulazioni”.
Per aprire il file in R:

```
> iris2<- read.table("iris2.txt", header=T)
```

B. Leggere e scrivere direttamente file Excel: il package xlsReadWrite:

```
> library(xlsReadWrite)
> data(iris)
> write.xls( iris, file="iris.xls",
colNames=TRUE);
> iris2 <- read.xls("iris.xls")
```

Esercizi

1. Costruire un data frame df1 di 5 righe con 6 variabili di cui 4 numeriche e 2 a caratteri. Memorizzare su file il data frame e quindi leggerlo, assegnandolo alla variabile df2.
2. Costruire una matrice numerica utilizzando la funzione *edit*. Scriverla su file tramite la funzione *write*. Ricaricare quindi la matrice in memoria. Si potrebbero utilizzare altre funzioni per memorizzare la matrice?
3. Scrivere su file il data frame df1 dell' es. 1 separando però gli elementi con virgole, ed omettendo il nome delle variabili.
4. Effettuare tramite R il download del file "colon128.train" dal sito http://homes.dsi.unimi.it/~valenti/DATA/Colon. Il file e' un data set con 31 campioni (righe), ognuno costituito da 129 feature separate da virgole.
5. Carica dal package *Biobase* il data set *aaMap*. A cosa si riferisce? Tramite quale struttura dati è rappresentato?

Docenti: **Matteo Re** **(INFORMATICA)**
Alessandro di Domizio **(STATISTICA)**

UNIVERSITÀ DEGLI STUDI DI MILANO

C.d.l. Biotecnologia

A.A. 2016-2017 semestre II

Informatica e Statistica

6 Controllo del flusso di esecuzione

Modulo: **INFORMATICA**

Controllo del flusso di esecuzione di un programma

I programmi sono eseguiti sequenzialmente, istruzione dopo istruzione, ma in alcuni casi il *flusso di esecuzione* può scegliere vie alternative o ripetersi ciclicamente.

In R esistono strutture di controllo specifiche per regolare il flusso di esecuzione di un programma:

Blocchi di istruzioni

Istruzioni condizionali

Istruzioni di looping

Sequenze e blocchi di istruzioni

Le istruzioni possono essere raggruppate insieme utilizzando le parentesi graffe. Una sequenza di istruzioni fra parentesi graffe costituisce un blocco.

```
Esempio:
{
  x <- runif(10);
  y <- runif(10);
  mx <- mean(x);
  my <- mean(y);
  vx <- sd(x);
  vy <- sd(y);
  g <- (mx-my)/(vx+vy);
  g;
}
```

- Si noti che i blocchi vengono valutati solo dopo la chiusura delle parentesi graffe.
- Si può pensare ad un blocco come ad un' unica macro istruzione costitutita da una sequenza di istruzioni

Istruzioni condizionali: l' istruzione if ... else

L' istruzione if ... else permette *flussi alternativi di esecuzione* dipendenti dalla valutazione di una *condizione logica*.

Sintassi:

```
if (condizione)
      blocco1
else
      blocco2
```

Semantica:

se la condizione è vera viene eseguito il `blocco1` altrimenti viene eseguito il `blocco2`.

If..else: esempi

Es.1:

```
if (x>=0)
  print("x è positivo")
else
  print("x è negativo")
```

Es.2:

```
if (x<=0) {
  y <- x^2;
  z <- log2(1+y);
}
else
    z <-log2(x);
```

Es.3:

Il ramo else può anche essere assente:

```
if (x<0)
  x <- -x;
sqrt(x)
```

L' istruzione `sqrt(x)` viene sempre eseguita, mentre `x<--x` viene eseguita solo se `x` è negativo.

Es.4:

Cosa accade se viene valutata un varabile non di "mode" logical?

```
if (x)
  print("x è diverso da 0")
else
  print("x è uguale a 0")
```

Istruzione if..else innestate

Le istruzioni if..else possono essere innestate:

```
if (condizione1)
  blocco1
else if (condizione2)
  blocco2
...
else if (condizioneN)
  bloccoN
else
  bloccoN+1
```

La funzione switch

La *funzione* switch consente di scegliere fra opzioni multiple.

La sua semantica è simile a quella dell' omonima struttura di controllo di altri linguaggi di programmazione.

Sintassi:

```
switch (istruzione, lista)
```

Semantica:

Viene valutata `istruzione` e viene ritornato un `valore`. Se `valore` è un numero compreso fra 1 e lunghezza della lista, allora viene valutato il corrispondente elemento della lista e viene ritornato un risultato. Se `valore` è troppo grande o troppo piccolo viene ritornato NULL.

La funzione switch: esempi

```
> x <- 3
> switch(x, 2+2, mean(1:100), rnorm(3))
[1] -0.3393166  0.1595591 -0.2016252
> x <- 2
> switch(x, 2+2, mean(1:100), rnorm(3))
[1] 50.5
> x <- 5
> switch(x, 2+2, mean(1:100), rnorm(3))
NULL
```

Se in `switch (espressione, lista con nomi)`

la valutazione di `espressione` ritorna un vettore di caratteri che corrisponde al nome associato ad un elemento della lista, tale elemento viene valutato.

Esempio:

```
> y <- "frutto"
> switch(y, frutto="pera", ortaggio="cavolo",
   legume="fagiolo")
[1] "pera"
```

Istruzioni di loop

Permettono di ripetere ciclicamente blocchi di istruzioni per un numero prefissato di volte o fino a che una determinata condizione logica viene soddisfatta

Sono istruzioni la cui struttura sintattica è del tipo:
loop `{ blocco di istruzioni}`

Esistono diverse forme di istruzioni di loop:

1. `for`
2. `while`
3. `repeat`

Istruzione for

Sintassi:

for (*nome* in *v*)
    blocco di istruzioni

v può essere un vettore o una lista

Semantica:

Gli elementi di *v* sono assegnati ad uno ad uno alla variabile *nome* ed il *blocco di istruzioni* viene valutato ciclicamente fino a che non sono stati esauriti tutti gli elementi di *v*.

Istruzione for: esempi

```
> v = round(runif(50)*5)
> for (i in 1:5) cat(v[i],"  ")
4   4   5   2   3
> for( i in (1: 10)* 5)  cat(v[i],"  ")
3   5   2   5   2   1   1   3   4   0
> for( j in c( 3,1,4,1,5,9,2,7))  cat(v[j],"  ")
5   4   2   4   3   3   4   1
```

L' istruzione *for* può ciclare su qualsiasi tipo di sequenza:

Es: accedere in sequenza alle componenti di un data frame

```
> for( var in names(data)) {... ; comp<- data$var; ... }
```

Es: accedere in sequenza a funzioni diverse:

```
> x <- c(pi, pi/2, pi/4) # pi corrisponde a π
> for( f in c(sin, cos, tan)) print(f(x))
[1] 1.224606e-16 1.000000e+00 7.071068e-01
[1] -1.000000e+00  6.123032e-17  7.071068e-01
[1] -1.224606e-16  1.633178e+16  1.000000e+00
```

Istruzione while

Sintassi:

while (*condizione*)
blocco di istruzioni

condizione è un' espressione logica

Semantica:

condizione viene valutata: se il suo valore è TRUE allora viene eseguito il *blocco di istruzioni*.
Il blocco di istruzioni continua ad essere eseguito ciclicamente se *condizione* rimane TRUE.
Quando *condizione* diventa FALSE allora si

Istruzione while - esempi

```
f <-function(y) {
  i <- 0;
  while (y > 1) {
    y <- y/2;
    i <- i + 1;
  }
  i
}
```

```
> f(1)
[1] 0
> f(2)
[1] 1
```

```
> f(10)
[1] 4
> f(1000)
[1] 10
```

```
> i<-1; while (a[i] < 0) i <- i+1;
```

Ciclo infinito:

```
while (TRUE) {... }
```

Ricerca della prima occorrenza di “UAG” nel vettore di caratteri d:

```
> i<-1; while (d[i] != "UAG" & i <=length(d)) i <- i+1;
```

Istruzione repeat

Sintassi:

repeat

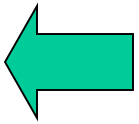
blocco di istruzioni

Semantica:

blocco di istruzioni viene eseguito ciclicamente all' infinito a meno che non venga incontrata una istruzione break che forzi l' uscita dal loop

Istruzione repeat - esempi

```
f1 <-function(y) {
  i <- 0;
  repeat {
    if (y<=1)
      break;
    y <- y/2;
    i <- i + 1;
  }
  i
}
```



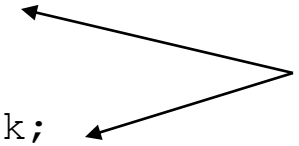
La funzione `f1` è semanticamente equivalente alla funzione `f` precedentemente vista negli esempi per l' istruzione *while*

Ciclo infinito:

```
repeat {... }
```

Si possono prevedere anche più punti di uscita da un repeat:

```
repeat {
   if (a[i]> 0.1) break;
   i<-i+1;
   ...
   if (i>length(a)) break;
   ...
}
```



punti di uscita dal loop

Iterazioni e operazioni/funzioni vettorizzate -1

Molte operazioni e funzioni in R sono *vettorizzate* ed operano elemento per elemento su interi oggetti.

Utilizzare direttamente operazioni o funzioni vettorizzate *è più efficiente* che effettuare le medesime operazioni utilizzando cicli for.

Esempio: prodotto scalare di due vettori:

```
> x<-runif(1000000); y <-runif(1000000);
```

A. Calcolo con cicli for:

```
z <- 0;
> system.time(for (i in 1:1000000) z <- z + x[i]*y[i])
   user  system elapsed
   2.88    0.00    2.88
```

B. Calcolo con funzioni vettorizzate:

```
> system.time(sum(x*y))
   user  system elapsed
   0.01    0.00    0.01
```

Iterazioni e operazioni/funzioni vettorizzate -2

Esistono almeno due buone ragioni per rimpiazzare (dove sia possibile) i cicli for con funzioni/operazioni vettorizzate:

1. *La velocità*: il loop `for` è molto più lento perchè deve essere valutato ogni volta dall' interprete
2. *La chiarezza*: è molto più semplice e sintetica l' espressione `sum(a*b)` piuttosto di una serie di cicli `for`.

Le funzioni vettorizzate includono:

1. Gli operatori & , | , ! , + , -, * , / , ^ , %%
2. Funzioni matematiche. Ad es: sin, cos, log, pnorm, choose
3. Generatori di numeri casuali: rnorm, runif, rpois,

I comandi “ciclici” della famiglia apply

I *comandi della famiglia apply* iterano una funzione specificata su insiemi di oggetti.

La loro sintassi generale è del tipo:

`comando_apply (insieme_di_oggetti, f)`

La funzione `f` viene applicata ciclicamente a ciascun oggetto contenuto nell’ `insieme_di_oggetti.`

Sono semanticamente equivalenti ad un ciclo for del tipo:

```
for (i in insieme_di_oggetti)
    f(insieme_di_oggetti[i])
```

In generale la loro esecuzione è più efficiente del corrispondente ciclo for.

Ne esistono diverse varianti (si veda l’ help in linea):

lapply ed sapply si applicano a liste; apply si applica ad array; tapply si usa con fattori.