Programmazione

- **Introduzione**

Come potete vedere e avete visto finora, *Mathematica* dispone di potentissimi comandi che permettono di eseguire operazioni e calcoli che richiederebbero un tempo notevole per poter essere eseguiti in qualsiasi altro modo (a me conosciuto). Soprattutto i comandi per il calcolo simbolico sono molto potenti, e permettono di poter effettuare cose che per esempio, in Fortran non si possono neanche pensare.

Sembra naturale ed ovvio, quindi, disporre di strumenti per poter mettere assieme questi comandi, in modo da poter eseguire una serie di operazioni in maniera automatizzata, esattaemente come si tratta per i programmi veri e propri; anzi, *Mathematica* consiste proprio di in un linguaggi di programmazione ad alto livello con comandi molto potenti, come avete anche visto con i vostri occhi e le vostre agili dita sulla tastiera.

Vedremo adesso i concetti base di programmazione in *Mathematica*, anche se probabilmente sarà una lettura molto veloce per molti di voi, dato che i principi base sono gli stessi per qualsiasi linguaggio di programmazione.

Quello che permette *Mathematica*, invece, è un controllo più avanzato per quanto riguarda le interfacce visuali, usando entrambi i sistemi Java e .NET. Tuttavia io non so ancora programmare ad oggetti, per cui le interfacce grafiche in *Mathematica*, come in C++, mi sono oscure. Tuttavia cercherò almeno di introdurvi nel discorso, in modo da poter cercare da soli la vostra strada per raggiungere la meta (miiii, che poetaaaa!!!! da leggere alla Aldo...).

- **Comandi base**

Cominciamo direttamente con un esempio...

Inizio

```
In[1]:= Print["Ciao bello!"]

        Ciao bello!
```

Quello che abbiamo ottenuto è l'equivalente in *Mathematica* del famoso programma "Hello World" che qualsiasi professore di qualsiasi corso di qualsiasi università vi fa il primo giorno... Solo che qua le cose sono decisamente più semplici, dato che non esistono main ed altre cavolatine varie (qualcuno ha detto Basic?).

Il comando Print, effettivamente, non fa altro che rappresentare il suo argomento sullo schermo; in questo caso, una stringa. Ebbene sì, miei cari allievi pendenti dalle mie labbra: *Mathematica* permette anche di gestire le stringhe, fra l'altro con la stessa potenza (quasi, va') del calcolo. D'altronde, essendo un programma di calcolo simbolico, che difficoltà ha nel gestire stringhe che altro non sono se una serie di simboli?

Abbiamo anche visto che le funzioni che possiamo definire in *Mathematica* possono fare ben più che dare come risultato un mero numero: in effetti, possiamo anche pensarle come le funzioni dei linguaggi di programmazione, che riescono a fare un compito assegnato qualsiasi, ammesso che si riesca a programmarlo... Ma questa è un altra storia. Consideriamo, per esempio, la seguente funzione:

```
In[2]:= esprod[x_, y_, t_, n_Integer] := FullSimplify[Series[x y, {t, 0, n}]]

In[3]:= esprod[x^2 + Sqrt[x], Gamma[x], x, 3]

Out[3]= 1/Sqrt[x] - EulerGamma Sqrt[x] + x + 1/12 (6 EulerGamma^2 + π^2) x^(3/2) -
        EulerGamma x^2 + 1/12 (-2 EulerGamma^3 - EulerGamma π^2 - 4 Zeta[3]) x^(5/2) +
        1/12 (6 EulerGamma^2 + π^2) x^3 + O[x]^(7/2)
```

La nostra funzione, in questo caso, non restituisce un numeri, ma un'espressione, come potete ben vedere. Possiamo considerarla come un sottoprogramma, come una macro, come quello che volete, insomma. Le funzioni che si definiscono possono contenere un numero arbitrario di funzioni e comandi, diventando dei veri e propri programmi che girano all'interno di *Mathematica*. Il concetto è proprio questo, ed in questo risiede buona parte della potenza che un utente avanzato riesce a spremere dal programma. Non per niente in giro esiste una guida in inglese di 3400 pagine su come programmare in *Mathematica*... Brrrrrr.... mi vengono i brividi solo a pensare di aprire un simile libro...

Tranquilli, le cose che faremo qua sono molto, mooolto più semplici (e con questo mi sono dato dell'imbecille).

Cominciamo a vedere i comandi, per scrivere e ricevere dati.

Input ed output

Abbiamo visto, prima, come importare ed esportare dati da files, in modo semplice e diretto. Abibamo anche visto, appena sopra, che è possibile dare in pasto ad un programmino opportuni argomenti, e inoltre abbiamo visto come stampare i risultati mediante il comando Print. Tuttavia, in maniera analoga ad altri casi, possiamo anche introdurre dei dati mediante il comando Input:

`Input["String"]`	richiede un valore, esplicitando la richiesta mediante string

Possiamo eseguire, per esempio, questo comando:

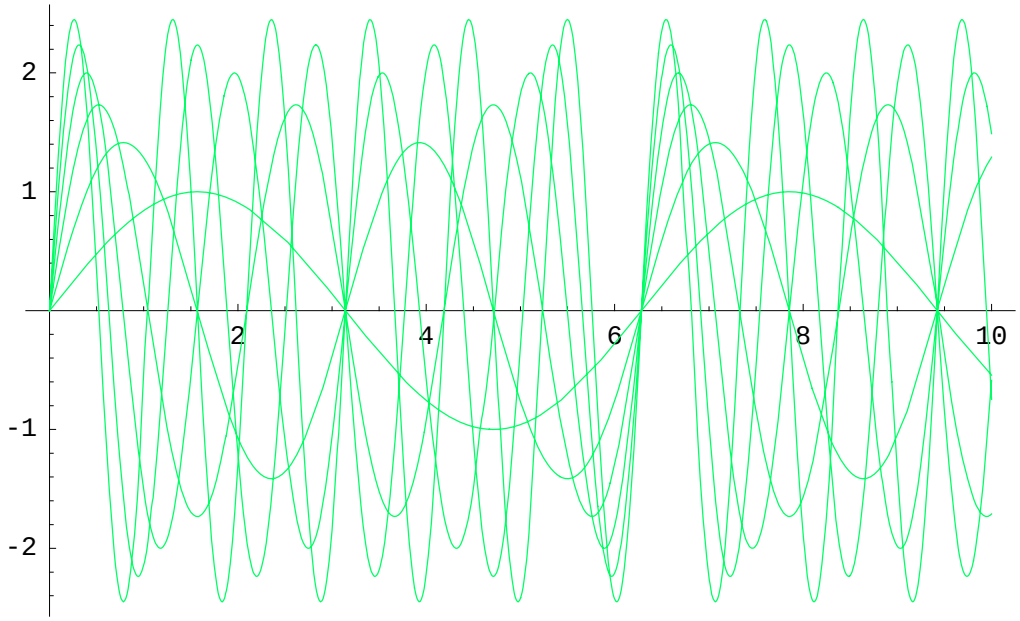
```
In[4]:= a = Input["Inserisci il numero"]
Out[4]= 555

In[5]:= a
Out[5]= 555
```

Qua non posso mostrarlo, ma appena eseguiamo il comando, appare una finestra del kernel del programma, dove compare sopra la stringa che abbiamo scritto, e sotto un riquadro dove scrivere il valore desiderato. In questo modo ho introdotto il valore, che è stato assegnato, come avete visto, alla variabile a. Questo è utile quando dobbiamo scrivere funzioni con molti parametri, e magari utilizziamo alcuni solo in situazioni particolari. Tuttavia si tratta sempre di valori che dobbiamo introdurre noi, per cui non è possibile effettuare Input se la funzione deve essere automatizzata, cioè deve essere effettuata senza il nostro intervento, In questo caso bisogna passare tutti gli argomenti necessari, oltre a magari definire prima delle variabili opportune.

Come abbiamo visto, per l'output su schermo possiamo usare il comando Print, oltre che i vari comandi di disegno nel caso ci servano.

In[6]:=
```
disegno[] := Sequence[
  n = Input["Inserisci il numero di seni:"],
  Plot[
   Evaluate[
    Table[
     Sqrt[i] Sin[i x], {i, n}
    ]
   ],
   {x, 0, 10},
   PlotStyle → Hue[0.4]
  ],
  Print["Hai disegnato " <> ToString[n] <> " seni."]
   Clear[n]
 ]
```

In[7]:=
```
disegno[];
```




```
Hai disegnato 6 seni.
```

In questo programmino abbiamo visto tutti e tre i metodi: il modo di avere un input al programma, e due output: uno grafico, che si ottiene semplicemente con il comando Plot, e l'altro ottenuto mediante Print.

Possiamo anche scrivere e leggere file che contengano espressioni di *Mathematica*, se non abbiamo bisogno di importare ed esportare, semplificando la sintassi. Supponiamo che dobbiamo scrivere un'espressione, e di volerla salvare su un file perchè, per esempio, dobbiamo uscire con la nostra ragazza, ed al ritorno, se saremo tristi e soli perchè ci ha lasciati, possiamo aprire un nuovo file senza dover ricaricare e valutare quello vecchio, continuando da dove ci siamo interrotti:

`Get: <<`	legge un file contenente espressioni di *Mathematica*, restituendo il risultato dell' ultima
`Put: >>`	scrive *l'* espressione in un file. Possiamo scriverne più di una nello stesso file con la forma Put[ex1, ex2,...., " file "]
`PutAppend: >>>`	accoda *l'* espressione nel file

Dicevamo, che avevamo il seguente output:

```
In[8]:= Expand[(x - 6)^7]
```

$$\text{Out[8]= } -279936 + 326592\,x - 163296\,x^2 + 45360\,x^3 - 7560\,x^4 + 756\,x^5 - 42\,x^6 + x^7$$

Adesso, vogliamo memorizzarlo nel file

```
In[9]:= % >> "espanso"
```

Questo comando memorizza l'espressione nel file. Vediamo se lo contiene:

```
In[10]:= !! espanso

-279936 + 326592*x - 163296*x^2 + 45360*x^3 - 7560*x^4 + 756*x^5 -
42*x^6 +
 x^7
```

Abbiamo visto il contenuto del file. Se vogliamo ricaricarlo in memoria, a questo punto basta fare

```
In[11]:= espr = << espanso
```

$$\text{Out[11]= } -279936 + 326592\,x - 163296\,x^2 + 45360\,x^3 - 7560\,x^4 + 756\,x^5 - 42\,x^6 + x^7$$

```
In[12]:= espr
```

$$\text{Out[12]= } -279936 + 326592\,x - 163296\,x^2 + 45360\,x^3 - 7560\,x^4 + 756\,x^5 - 42\,x^6 + x^7$$

Come potete vedere, adesso espr contiene l'espressione che avevamo salvato in precedenza. Notate come !! non restituisce l'espressione, ma solo il contenuto del file. Abbiamo comunque bisogno del comando Get per poter leggere il file, interpretarlo e memorizzarlo nella variabile.

A volte, invece, mentre lavoriamo, può capitare di non dover memorizzare tutte le espressioni in una volta, ma a poco a poco. In questo ci viene in aiuto il comando PutAppend:

In[13]:=
```
ExpandAll[(x - 4)^5 / (x - 2)^7] // TraditionalForm
```

Out[13]//TraditionalForm=

$$\frac{x^5}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}-\frac{20x^4}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}+\frac{160x^3}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}-\frac{640x^2}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}+\frac{1280x}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}-\frac{1024}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}$$

In[14]:=
```
% >>> espanso
```

In[15]:=
```
!! espanso
```

```
-279936 + 326592*x - 163296*x^2 + 45360*x^3 - 7560*x^4 + 756*x^5 -
42*x^6 +
 x^7
-1024/(-128 + 448*x - 672*x^2 + 560*x^3 - 280*x^4 + 84*x^5 - 14*x^6
+ x^7) +
 (1280*x)/(-128 + 448*x - 672*x^2 + 560*x^3 - 280*x^4 + 84*x^5 -
14*x^6 +
   x^7) - (640*x^2)/(-128 + 448*x - 672*x^2 + 560*x^3 - 280*x^4 +
84*x^5 -
   14*x^6 + x^7) + (160*x^3)/(-128 + 448*x - 672*x^2 + 560*x^3 -
280*x^4 +
   84*x^5 - 14*x^6 + x^7) - (20*x^4)/(-128 + 448*x - 672*x^2 +
560*x^3 -
   280*x^4 + 84*x^5 - 14*x^6 + x^7) +
 x^5/(-128 + 448*x - 672*x^2 + 560*x^3 - 280*x^4 + 84*x^5 - 14*x^6
+ x^7)
```

Come potete vedere adesso, il contenuto del file è cambiato, aggiungendo l'ultima espressione a quella già memorizzata in precedenza. In questo caso, però, dobbiamo fare attenzione. Abbiamo detto che il comando valuta tutte le espressioni, ma restituisce solamente l'ultima: se andiamo a memorizzare il contenuto del file abbiamo:

In[16]:=
```
espr2 = << espanso // TraditionalForm
```

Out[16]//TraditionalForm=

$$\frac{x^5}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}-\frac{20x^4}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}+\frac{160x^3}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}-\frac{640x^2}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}+\frac{1280x}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}-\frac{1024}{x^7-14x^6+84x^5-280x^4+560x^3-672x^2+448x-128}$$

In[17]:=
```
Simplify[%]
```

Out[17]=
$$\frac{(-4+x)^5}{(-2+x)^7}$$

Possiamo vedere come solamente l'ultimo risultato memorizzato nel file venga poi effettivamente memorizzato. Se vogliamo leggerli tutti dobbiamo utilizzare il comando ReadList:

In[18]:=
```
espr3 = ReadList["espanso"];
```

In[19]:=
```
Simplify[espr3]
```

Out[19]=
$$\left\{(-6+x)^7,\ \frac{(-4+x)^5}{(-2+x)^7}\right\}$$

Come possiamo vedere, stavolta abbiamo ottemuto una lista contente tutte le espressioni contenute nel file.

A questo punto, ci si può chiedere perchè Get restituisce solamente l'ultima espressione valutata. Questo dipende dal fatto che quando inseriamo una sequenza di operazioni un unico comando, *Mathematica* restituisce solo l'ultimo, anche se li calcola tutti. Può sembrare uno svantaggio, ma l'uso è invece specifico. Possiamo, infatti, memorizzare delle definizioni di funzioni o di programmi personali nel file. Una volta che si legge, valuta e calcola tutte le funzioni, che quindi saranno direttamente disponibili senza doverle riscrivere oppure senza dover aprire e valutare il notebook dove sono contenute: in questo caso, infatti, non appena avremo eseguito Get ed il programma avrà valutato ogni espressione, avrà tutto quanto in pancia, semplificandoci il lavoro. In effetti, Get viene utilizzato spesso per caricare Packages esterni, ovvero file dove sono memorizzate definizioni, costanti e funzioni che permettono di snellire il lavoro per un compito specifico, come quello standard in *Mathematica* dedicato al calcolo vettoriale.

Flusso di programma

Come certamente saprete, ci sono casi in cui il programma non deve svolgersi esattamente riga per riga dall'inizio alla fine, ma bisogna effettuare delle scelte, ed eseguire determinate operazioni a seconda del tipo di risposta ottenuta. Naturalmente cominceremo con il comando più conosciuto:

`If[`*p*`,` *then*`,` *else*`]`	restituisce *then* se *p* è `True`, ed *else* se *p* è `False`

In altre parole, esegue l'operazione logica *p*, e decide cosa fare a seconda del risultato. Per scrivere codice, userò l'annidamento, dato che ormai sapere che si possono annidare comandi senza problemi (mica ve l'avevo fatto vedere per niente... tutto ha un fine!):

In[20]:=
```
test[x_] :=
 If[x^2 > 5,
  Print["Il test è corretto, e risulta ", x^2],
  Print["Hai sbagliato tutto!!!"]
 ]
```

Non credo che il programmino (assimilabile ad una funzione personalizzata, ma abbiamo capito che sono la stessa cosa), abbia bisogno di spiegazioni, vero?

In[21]:=
```
test[2]
```

```
Hai sbagliato tutto!!!
```

In[22]:=
```
test[4]
```

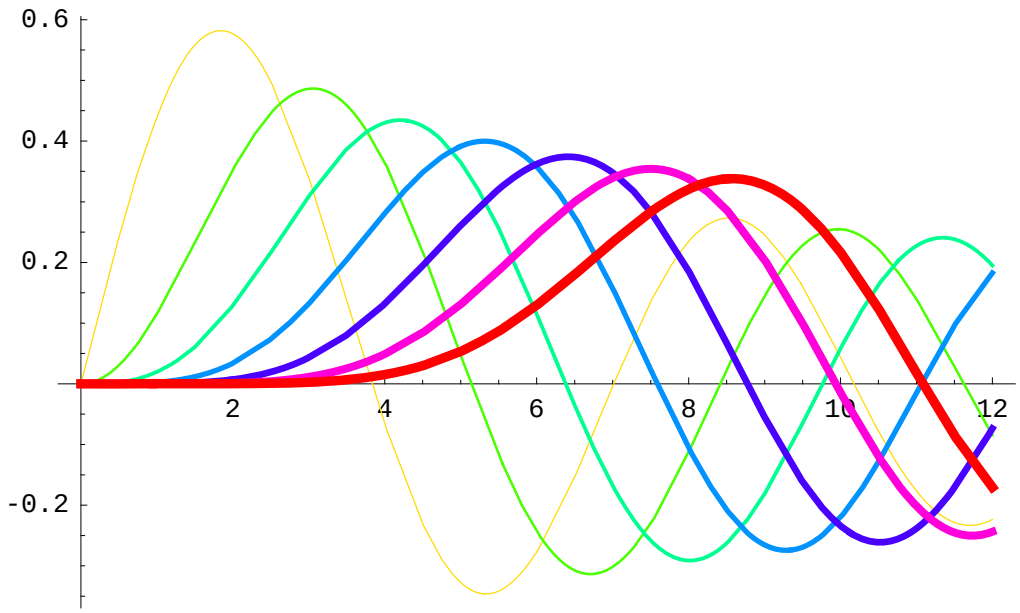
```
Il test è corretto, e risulta 16
```

Già da qua potete vedere come spuntano cosucce carine e notevoli, come il fatto che si possono mischiare stringe e valori, e notando come non sia necessario, come il C, l'uso di caratteri e stringhe come %d o %f. Risulta più semplice scrivere questo semplice programma in *Mathematica* che in C o in Basic, figurarsi quelli complicati...

Per saggiare la flessibilità e la potenza dell'ambiente di programmazione, consideriamo quest'altro programma:

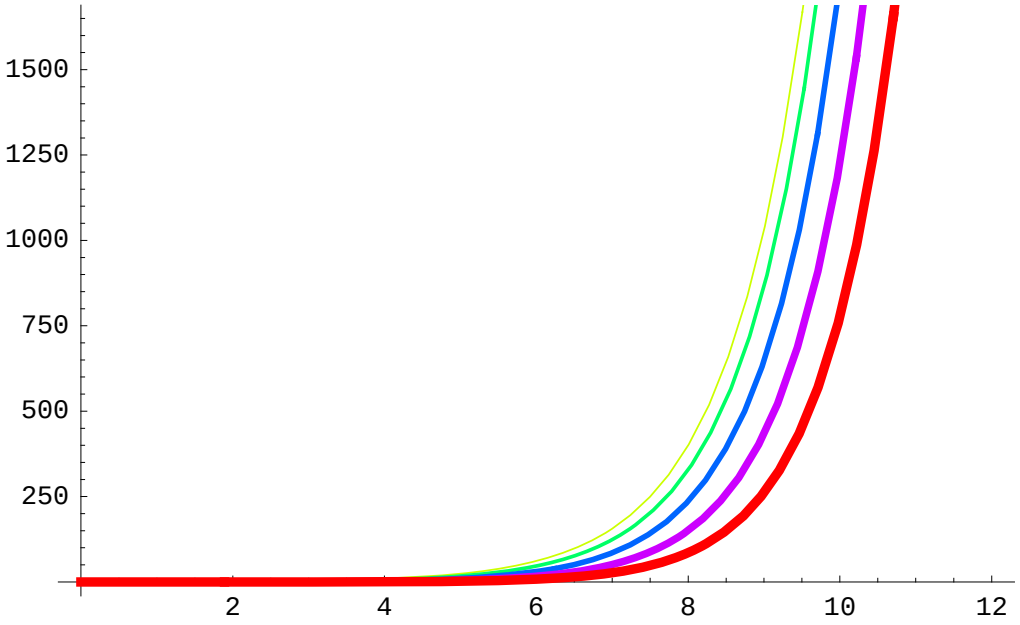
In[23]:=
```
disegno[tipo_Integer, n_Integer] :=
  If[tipo == 0,
   Plot[
    Evaluate[
     Table[
      BesselJ[q, x], {q, n}
     ], {x, 0, 12},
     PlotStyle →
      Table[{Hue[r / n], Thickness[0.01 r / n]}, {r, n}]
    ]
   ],
   Plot[
    Evaluate[
     Table[
      BesselI[q, x], {q, n}
     ], {x, 0, 12},
     PlotStyle →
      Table[{Hue[r / n], Thickness[0.01 r / n]}, {r, n}]
    ]
   ]
  ];
```

In[24]:=
```
disegno[0, 7]
```



Out[24]= - Graphics -

In[25]:=
```
disegno[2, 5]
```



Out[25]=
```
- Graphics -
```

In[26]:=

Possiamo vedere come si possano effettuare e sintetizzare in un unico comando cose anche più complicate. Provate a farlo in C...

Un piccolo limite (ma neanche tanto) della programmazione in *Mathematica* riguarda la concatenazione di più comandi assieme: considerate, per esempio, il seguente programmino:

In[27]:=
```
prova[c_] := Print["Il sole è bello"]
a = c
Print["a"]
Clear[a];
```

Out[28]=
```
c

a
```

In[31]:=
```
prova[4]

Il sole è bello
```

Come potete vedere, c'è qualcosa che non va. Prima di tutto, appena definita la funzione ho un risultato, cosa che non dovrebbe succedere quando si dediniscono funzioni oppure programmi, mentre una volta richiamato il programmino da l'output incompleto, restituendo solamente il primo

comando. Per *Mathematica*, in effetti, abbiamo definito prima vari comandi tutti distinti fra di loro, e quindi il secondo comando, quello di assegnamento, è disconnesso da tutto il resto. Possiamo provare a visualizzare il contenuto del programma:

In[32]:=
```
?? prova
```

```
Global`prova

prova[c_] := Print[Il sole è bello]
```

Come potete vedere, la funzione è formato solo dal primo comando. Quindi non ci sono comandi concatenati. Effettivamente, per poterli fare, opporre usare un altro comando, dove all'interno sono contenuti tutti i comandi del programma:

`Sequence[...]`	esegue in sequenza tutti i comandi racchiusi all'interno delle sue parentesi.

In[33]:=
```
prova[c_] := Sequence[
   Print["Il sole è bello"],
   a = c,
   Print[a],
   Clear[a]
  ];
```

Come potete vedere, abbiamo definito il programma come un'unica funzione. Vediamo adesso se funziona:

In[34]:=
```
prova[8]
```

```
Il sole è bello

8
```

Out[34]=
```
Sequence[Null, 8, Null, Null]
```

In[35]:=
```
prova[2]
```

```
Il sole è bello

2
```

Out[35]=
```
Sequence[Null, 2, Null, Null]
```

Come potete vedere, questa volta il programma funziona, avendo eseguito tutti i comandi in sequenza. Inoltre, si può vedere come ritorna anche Sequence con il risultato di ogni comando

contenuto in esso, cioè tre Null dovuti ad un Print ed al Clear, ed l'8 dell'assegnazione: Print, restituisce Null perchè stampa qualcosa sullo schermo, ma non memorizza niente all'interno del programma come variabili od altro... Quindi se a noi restituisce un risultato, questo non è altrettanto vero per *Mathematica*, e quindi, restituisce Null.

Ora sappiamo usare l'If e sappiamo anche come si realizza una sequenza di programmi, per cui possiamo anche vedere gli altri comandi di controllo, che sono simili agli altri linguaggi di programmazione:

lhs `:=` *rhs* `/;` *test*	usa la definizione solo se *test* risulta `True`
`If[`*test*`,` *then*`,` *else*`]`	valuta *then* se *test* risulta `True`, ed *else* se risulta `False`
`Which[`$test_1$`,` $value_1$`,` $test_2$`, … ]`	valuta $test_i$ e quando trova il primo che risulta True, restituisce il corrispondente valore
`Switch[`*expr*`,` $form_1$`,` $value_1$`,` $form_2$`, … ]`	compara *expr* con le forme in $form_i$, restituendo il risultato della prima corrispondenza che trova
`Switch[`*expr*`,` $form_1$`,` $value_1$`,` $form_2$`, … , _,` *def*`]`	usa *def* come valore di default
`Piecewise[{{`$value_1$`,` $test_1$`}, … },` *def*`]`	restituisce il primo valore $test_i$ che corrisponde a `True`

Sono tutte istruzioni condizionali, che permettono di scrivere in diverse maniere un algoritmo di scelta, restituendo quella più opportuna. Si decide quale usare a seconda di quella che restituisce la sintassi più breve e coincisa. In teoria, infatti, niente ci vieta di sostituire ogni comando con una serie opportunamente annidata di If...

In[36]:=
```
prova[x_] := Sequence[
  Print["Inizio"],
  Which[
   x > 0, Print["Il numero è positivo"],
   x < 0, Print["Il numero è negativo" ],
   x == 0, Print["Hai beccato lo zero!!!"]
  ],
  Print["Il quadrato risulta ", x^2],
  Print["Fine programma. Ciao da Daniele"]
 ]
```

In[37]:=
```
prova[4];
```

```
Inizio

Il numero è positivo

Il quadrato risulta 16

Fine programma. Ciao da Daniele
```

In questo esempio abbiamo visto come si usa Which, indicando ogni condizione ed ogni comando da eseguire per la condizione giusta; notate anche come abbia potuto eseguire una formattazione della scrittura tipica dei linguaggi di programmazione, dato che andando a capo non si interrompe la sintassi del comando, esattamente come in C.

Non è che il programma sia complicatissimo... Tuttavia, serve solo per farvi capire come funzionano i comando, dato che presumo che almeno qualcosina di programmazione la conoscete, e sapete cosa fare, e come farlo.

In[38]:=
```
altraprova[x_] := Sequence[
  Print["Analisi del numero"],
  Switch[PrimeQ[x],
   True, Print["Il numero è primo"],
   False,
   Print["Il numero non è primo, e i suoi fattori sono:\n",
    FactorInteger[x]]
  ]
 ]
```

In[39]:=
```
altraprova[7365576585];
```

```
Analisi del numero

Il numero non è primo, e i suoi fattori sono:
{{3, 1}, {5, 1}, {877, 1}, {559907, 1}}
```

In[40]:=
```
altraprova[508969];
```

```
Analisi del numero

Il numero è primo
```

Abbiamo visto un altro esempio, questa volta a proposito del comando Switch; prima di tutto abbiamo testato se l'argomento era un numero primo, indicando, in caso affermativo, che era tale ed, in caso alternativo, eseguiamo la fattorizzazione del numero primo, mostrandola attraverso il comando Print.

Naturalmente, non sono queste le uniche istruzioni di flusso, come ben sapete; ci sono anche istruzioni che determinano la ripetizione di un blocco di codice per un determinato numero di volte. Uno dei comandi più utili ed importanti di questo tipo in *Mathematica* è il seguente:

`Do[`*expr*`, {`*i*`, `*imax*`}]`	valuta *expr* per il numeri specificato di volte, con *i*

	che varia da `1` a *imax* con passo `1`
`Do[`*expr*`, {`*i*`, `*imin*`, `*imax*`, `*di*`}]`	valuta *expr* con *i* che varia da *imin* a *imax* con passo *di*
`Do[`*expr*`, {`*n*`}]`	valuta *expr n* volte

Il comando è l'equivalente del For in C:

```
In[41]:= Do[Print["Numero ", i], {i, 6}]
```

```
Numero 1

Numero 2

Numero 3

Numero 4

Numero 5

Numero 6
```

Possiamo vedere come si comporta esattamente come vogliamo, della maniera preferita. Un aspetto interessante è che possiamo fare dei cicli annidati senza usare istanze multiple del comando, ma semplicemente indicando tutte le variabili, come facciamo con il comando Table:

```
In[42]:= Do[Print[i, " x ", j], {i, 3}, {j, 3}]
```

```
1 x 1

1 x 2

1 x 3

2 x 1

2 x 2

2 x 3

3 x 1

3 x 2

3 x 3
```

Possiamo vedere come il ciclo più interno sia quello corrispondente alla variabile che scriviamo per ultima: con *i* pari ad 1, *j* varia da 1 a 3: poi *i* viene incrementato l'indice *i* e si ripete il ciclo *j*. Un

altro comando simile, ma estremamente utile in alcuni casi, è il seguente:

`FixedPoint[`*f*`,` *expr*`]`	comincia con *expr*, ed applica *f* iterativamente fino a quando il risultato non varia più

Come potete vedere, questo comando permette di avere una terminazione che non dipende dal ciclo, ma dobbiamo stare attenti che effettivamente si arrivi ad un punto in cui la funzione non cambia più: In pratica, effettua questi passaggi:

$x,\ f[x],\ f[f[x]],\ f[f[f[x]]]$ ...

Questo fino a quando il risultato non varia più. Di solito si usa per condizioni di convergenza:

```
In[43]:= FixedPoint[Function[t, Print[t]; Floor[Sqrt[t / 2]]], 2350]

2350

34

4

1

0

Out[43]= 0
```

Come potete vedere, abbiamo applicato la funzione Floor[Sqrt[*x*]] ripetitivamente. Ho usato le funzioni pure, cosa che vedremo più avanti, ma per un utente avanzato questo può veramente risolvere problemi difficili da studiare in altro modo. Un modo più semplice di usarlo è con il comando ReplaceAll (//.), che esegue quasi esattamente la stessa cosa, in forma più sintetica

```
In[44]:= x^2 + y^6 //. {x → 2 + a, a → 3}

Out[44]= 25 + y^6
```

In questo caso, le regole di sostituzione non vengono effettuate una volta sola, ma fino a quando il risultato non varia più. Tuttavia si può notare come mostri soltanto il risultato finale, oltre ad avere qualche problema di inizializzazione che invece risulta più semplice con FixedPoint.

Possiamo anche applicare la funzione in maniera iterativa per un numero determinato di volte, mediante il seguente comando:

`Nest[`*f*`,` *expr*`,` *n*`]`	applica *f* iterativamente su *expr n* volte

Questo può essere utile per lavori iterativi:

In[45]:=
```
f[x_] := 1 / (1 + x^2)
```

In[46]:=
```
Nest[f, q, 5]
```

Out[46]=

$$\frac{1}{1+\frac{1}{\left(1+\frac{1}{\left(1+\frac{1}{\left(1+\frac{1}{(1+q^2)^2}\right)^2}\right)^2}\right)^2}}$$

Come possiamo vedere, come primo argomento abbiamo bisogno soltanto del nome della funzione, non delle parentesi quadre. Se nella funzione sono presenti più argomenti, di cui tutti gli altri specificati, conviene creare una nuova funzione che abbia come argomento un solo parametro:

In[47]:=
```
g[x_] := fun[var1, x, var2]
```

ed usare questa nel comando Nest. Ci sarebbero funzioni più eleganti dal punto di vista di programmazione avanzata, come le funzioni pure che abbiamo visto poco fa, ma al nostro livello questa soluzione è sicuramente quella più facile da eseguire. Un comando simile è:

`NestWhile[`*f*`,` *expr*`,` *test*`]`	cominica con *expr*, ed applica iterativamente *f* fino a quando *test* applicato al risultato non è più `True`

Quindi, cambia la condizione di terminazione, che diventa da determinata a condizionale; cioè non viene applicata iterativamente *f* per un certo numero di volte predefinito, ma viene applicata fino a quando non si raggiunge una condizione di stabilità. Ovviamente, anche in questo caso occorre che la condizione di stabilità venga raggiunta, cosa che si deduce dal tipo di problema che si sta elaborando.

Altri costrutti per poter eseguire ripetitivamente blocchi di codice, oltre al Do, sono:

`While[`*test*`,` *body*`]`	valuta test e, se risulta True, esegue *expr*: dopo rivaluta test, e se risulta ancora True riesegue body, e così via fino a quando test risulta False
`For[`*start*`,` *test*`,` *incr*`,` *body*`]`	valuta dapprima *start* e poi, ciclicamente, *body* e *incr* fino a quando *test* risulta True, e dopo si interrompe.

Come potete vedere, sono equivalenti ai corrispettivi comandi nei vari linguaggi di programmazione; per esempio potremmo scrivere qualcosa come:

In[48]:=
```
For[n = 1; a = {}, n <= 10, n++, a = Append[a, n]; Print[a]]
```

```
{1}

{1, 2}

{1, 2, 3}

{1, 2, 3, 4}

{1, 2, 3, 4, 5}

{1, 2, 3, 4, 5, 6}

{1, 2, 3, 4, 5, 6, 7}

{1, 2, 3, 4, 5, 6, 7, 8}

{1, 2, 3, 4, 5, 6, 7, 8, 9}

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Come abbiamo visto, abbiamo ottenuto una lista di dieci elementi, eseguendo il semplice comando.

Inoltre, in questi comandi possiamo anche introdurre i comandi Break[] e Continue[], che conoscete tutti; il primo serve per interrompere il ciclo più interno in cui si trova il comando, mentre il secondo serve per saltare il restante ciclo e ricominciare il successivo. Io personalmente li uso poco, perchè, anche se in maniera estremamente ridotta rispetto a Goto (che è presente ma che non menziono), crea dei salti di flusso, cosa che a me non è mai piaciuta. Sappiate comunque che esistono e che, se volete, potete anche usarli tranquillamente.

Stringhe

Abbiamo visto che *Mathematica* è anche in grado di gestire le stringhe. In effeti, i suoi strimenti sono abbastanza potenti, e questo perchè si deve poter eseguire anche un determinato numero di operazioni che possono essere importanti dal punto di vista matematico e del lavoro che si sta svolgendo: si pensi, ad esempio, all'importazione di una stringa di 1000000 caratteri, dove ogni carattere corrisponde allo stato di un sistema e vedere, ad esempio, quando e quante volte il sistema, nella sua storia, passa dallo stato T allo stato F, per effettuare delle necessarie statistiche. Le stringhe possono quindi anche essere utili per il calcolo, e vengono trattate come tali. Le funzioni forse più elementari per la gestione delle stringhe sono le seguenti:

s_1 <> s_2 <> … oppure `String Join[{`s_1, s_2, … `}]`	unisce assieme più stringhe per formarne una sola
`StringLength[`*s*`]`	restituisce il numero di caratteri della stringa
`StringReverse[`*s*`]`	inverte l'ordine dei caratteri della stringa

Definiamo, per capirci meglio, le seguenti stringhe:

```
In[49]:= stringa1 = "Io sono";
```

```
In[50]:= stringa2 = "un";
```

```
In[51]:= stringa3 = "gran figo...";
```

Se adesso andiamo ad unirle assieme, otteniamo:

```
In[52]:= stringa1 <> stringa2 <> stringa3
Out[52]= Io sonoungran figo...
```

Naturalmente, non appare nella forma voluta perchè non abbiamo considerato gli spazi, dato che lo spazio è considerato come carattere equivalente agli altri, e *Mathematica* NON li aggiunge se non lo specigichiamo noi:

```
In[53]:= stringa1 <> " " <> stringa2 <> " " <> stringa3
Out[53]= Io sono un gran figo...
```

Ora le cose sono più vicine alla realtà...

Possiamo anche contare i caratteri dell'ultima stringa:

In[54]:=
```
StringLength[%]
```

Out[54]=
```
23
```

Come potete vedere, 23 caratteri bastano per dire una grande verità. Vediamo adesso anche l'ultimo comando dei tre:

In[55]:=
```
StringReverse["Alice attraverso lo specchio"]
```

Out[55]=
```
oihcceps ol osrevartta ecilA
```

Naturalmente possiamo giocare anche con la lingua italiana... Consideriamo la frase "I topi non avevano nipoti":

In[56]:=
```
StringReverse["itopinonavevanonipoti"]
```

Out[56]=
```
itopinonavevanonipoti
```

Ah, potenza dell'italiano, che HA una lingua beLa aSai...

Operazioni un poco più avanzate (ma neanche di tanto, in fondo) che si possono fare nelle stringhe sono le seguenti:

`StringTake[s, n]`	crea una stringa formata dai primi *n* caratteri di *s*

`StringTake[s, {n}]`	estreae l' n^{th} carattere da *s*
`StringTake[s, {n1, n2}]`	crea una stringa estratta da *s* dal carattere n_1 al carattere n_2
`StringDrop[s, n]`	crea una stringa ottenuta cancellando i primo *n* ccaratteri da *s*
`StringDrop[s, {n1, n2}]`	cancella i caratteri dalla posizione n_1 fino a n_2
`StringInsert[s, snew, n]`	inserisce la stringa *snew* nella posizione *n* in *s*
`StringInsert[s, snew, {n1, n2, … }]`	inserisce più copie di *snew* into *s* nelle posizioni n_1, n_2, …
`StringReplacePart[s, snew, {m, n}]`	sostituisce i caratteri da *m* a *n* nella stringa *s* con la stringa *snew*
`StringReplacePart[s, snew, {{m1, n1}, {m2, n2}, …`	sostituzioni multiple in *s* con *snew*
`StringReplacePart[s, {snew1, snew2, … }, {{m1, n1}, {m2, n2}, … }]`	sostituisce più sottostringhe in *s* dalle corrispondenti nuove sottostringhe $snew_i$
`StringPosition[s, sub]`	restituisce una lista di coppie di valori iniziali e finali delle posizioni in cui *sub* compare come sottostringa in *s*
`StringPosition[s, sub, k]`	mostra solo le prime *k* occorenze di *sub* in *s*
`StringPosition[s, {sub1, sub2, … }]`	mostra le posizioni di più sottostringhe sub_i
`StringCount[s, sub]`	conta quante volte *sub* compare in *s*
`StringCount[s, {sub1, sub2, … }]`	conta tutte le volte che compare una delle stringhe sub_i
`StringFreeQ[s, sub]`	testa se *s* non contiene *sub*
`StringFreeQ[s, {sub1, sub2, … }]`	testa *s* non contiene nessuna delle stringhe sub_i
`StringReplace[s, sb -> sbnew]`	sostituisce *sb* con *sbnew* ovunque compaia in *s*
`StringReplace[s, {sb1 -> sbnew1, sb2 -> sbnew2, …`	sostituisce le sottostringhe sb_i dalle corrispondenti nuove sottostringhe $sbnew_i$
`StringReplace[s, rules, n]`	esegua al massimo *n* sostituzioni
`StringReplaceList[s, rules]`	crea una lista delle stringhe ottenute effettuando ogni singola sostituzione
`StringReplaceList[s, rules, n]`	limita la lista a *n* risultati
`StringSplit[s]`	divide *s* nelle sottostringhe delimitate dagli spazi
`StringSplit[s, del]`	divide la stringa usando il delimitatore *del*
`StringSplit[s, {del1, del2, … }]`	usa più delimitatori del_i
`StringSplit[s, del, n]`	divide al massimo *n* sottostringhe
`StringSplit[s, del -> rhs]`	inserisce *rhs* nella posizione di ogni delimitatore

Come potete vedere, solo con questi comandi base possiamo fare veramente molte cose: praticamente tutto quello che ci serve per gestire le stringhe ad un livello poco più avanzato di quello base:

```
In[57]:= stringa = "Il mio nome è Daniele, e sono
             talmente felice che tu abbia letto fino a qua che per
             festeggiare me ne andrò al mare a fare un bel bagno";
```

Sapete com'è, fuori il tempo è bello, e ne devo approfittare prima di studiare per gli esami di giugno-luglio...

Comunque, una volta creata la stringa, possiamo gestircela come più ci piace. Mettiiamo caso che non mi piaccia più andare al mare (cosa impossibile):

```
In[58]:= StringReplace[stringa, "al mare" -> "in montagna"]

Out[58]= Il mio nome è Daniele, e sono talmente
           felice che tu abbia letto fino a qua che per
           festeggiare me ne andrò in montagna a fare un bel bagno
```

Ovviamente, il bagno me lo farò nel lago...

Vediamo di ordinare alfabeticamente le parole. In questo caso ho bisogno prima di separare le parole, e dopo ordinare le singole stringhe ottenute. Indico come delimitatore solo lo spazio (considerando, quindi, i segni di punteggiatura, come le virgole, caratteri). Dopo averlo fatto, ordino le stringhe così ottenute.

```
In[59]:= StringSplit[stringa, " "]

Out[59]= {Il, mio, nome, è, Daniele,, e, sono, talmente,
           felice, che, tu, abbia, letto, fino, a, qua, che, per,
           festeggiare, me, ne, andrò, al, mare, a, fare, un, bel, bagno}

In[60]:= Sort[%]

Out[60]= {a, a, abbia, al, andrò, bagno, bel, che, che, Daniele,,
           e, è, fare, felice, festeggiare, fino, Il, letto, mare,
           me, mio, ne, nome, per, qua, sono, talmente, tu, un}
```

Se invece, voglio ordinarli in ordine inverso, basta considerare il fatto che abbiamo a che fare con una lista, quindi basta usare il comando per invertire gli elementi di una lista, cosa che abbiamo visto a suo tempo:

```
In[61]:= Reverse[%]

Out[61]= {un, tu, talmente, sono, qua, per, nome, ne, mio, me,
          mare, letto, Il, fino, festeggiare, felice, fare, è, e,
          Daniele,, che, che, bel, bagno, andrò, al, abbia, a, a}
```

Ovviamente potevamo ottenere tutto quanto in un unico comando con le funzioni annidate:

```
In[62]:= Reverse[Sort[StringSplit[stringa]]]

Out[62]= {un, tu, talmente, sono, qua, per, nome, ne, mio, me,
          mare, letto, Il, fino, festeggiare, felice, fare, è, e,
          Daniele,, che, che, bel, bagno, andrò, al, abbia, a, a}
```

Nidificare le funzioni è una cosa che farete abbastanza spesso, una volta presa confidenza con il programma. Anzi, sono sicuro che già l'avrete fatto molte volte, vero? Fate bene, perchè vi permette di risparmiare tempo e spazio a meno che, naturalmente, non vi servano anche i risultati intermedi.

Supponiamo, adesso, di dover censurare delle parole: per esempio il mio nome non deve essere reso pubblico. Non sapendo a priori la posizione in cui compaia il mio nome, basta fare:

```
In[63]:= StringPosition[stringa, "Daniele"]

Out[63]= {{15, 21}}
```

Ed, a questo punto:

```
In[64]:= StringReplacePart[stringa, "XXX", %[[1]]]

Out[64]= Il mio nome è XXX, e sono talmente felice che tu abbia letto fino a
           qua che per festeggiare me ne andrò al mare a fare un bel bagno
```

Notate come ho dovuto indicare l'indice, dato che il risultato di StringPosition è una lista doppia. Naturalmente, potevo anche farlo direttamente:

```
In[65]:= StringReplace[stringa, "Daniele" -> "XXX"]

Out[65]= Il mio nome è XXX, e sono talmente felice che tu abbia letto fino a
           qua che per festeggiare me ne andrò al mare a fare un bel bagno
```

Queste sono solo alcuni esempi banali. I comandi esistono per fare enalisi più serie; per esempio, in una stringa può essere memorizzata una sequenza di DNA, e vedere quando compare una specfica sequenza, e dove. Possiamo analizzare direttamente il primer, senza dover per forza visualizzare tutta la stringa, che può essere veramente grossa. Naturalmente dovrete essere voi a deciderne l'uso da fare.

Mathematica dispone anche di alcune funzioni per lavorare direttamente con i caratteri di una stringa:

`Characters["`*string*`"]`	converte una stringa in una lista di caratteri
`StringJoin[{"`c_1`", "`c_2`", ... }]`	converte una lista di caratteri in una stringa
`DigitQ[`*string*`]`	verifica se gli elementi della stringa sono tutte cifre
`LetterQ[`*string*`]`	verifica se *i* caratteri della stringa sono tutte lettere
`UpperCaseQ[`*string*`]`	testa se i caratteri sono tutti maiuscoli
`LowerCaseQ[`*string*`]`	verifica se i caratteri sono tutti minuscoli
`ToUpperCase[`*string*`]`	converte la stringa in una con le lettere tutte maiuscole
`ToLowerCase[`*string*`]`	genera la stringa con tutte le lettere minuscole
`CharacterRange["`c_1`", "`c_2`"]`	genera una lista con i caratteri da c_1 a c_2

Supponiamo, per esempio, di dover convertire in lista di caratteri la seguente stringa:

```
In[66]:= stringa = "Qualche Lettera Scelta A Caso";

In[67]:= Characters[stringa]

Out[67]= {Q, u, a, l, c, h, e,  , L, e, t, t, e,
          r, a,  , S, c, e, l, t, a,  , A,  , C, a, s, o}
```

Notate come anche gli spazi siano considerati caratteri:

```
In[68]:= Sort[%]

Out[68]= {  ,  ,  ,  , a, a, a, a, A, c, c, C, e,
           e, e, e, h, l, l, L, o, Q, r, s, S, t, t, t, u}
```

Supponiamo di fregarcene della netiquette di un newsgroup e di voler gridare questa frase in un thread:

```
In[69]:= ToUpperCase[stringa]

Out[69]= QUALCHE LETTERA SCELTA A CASO
```

Al contrario, adesso siamo stati tirati per l'orecchio, e vogliamo fare i timidoni:

```
In[70]:= ToLowerCase[stringa]

Out[70]= qualche lettera scelta a caso
```

Non credo che servano ulteriori commenti a questi comandi, molto semplici ma flessili, ammesso ovviamente che ci dobbiate fare qualcosa!

Possiamo anche usare all'interno delle stringhe i caratteri speciali di *Mathematica*. Se avete sperimentato un poco, avrete notato una Palette contenente le lettere greche e altri simboli matematici, come le frecce. Avrete forse anche notato che potete scriverle direttamente da tastiera. Per esempio, potete scrivere π sia con [Esc]pi[Esc], sia con \[pi]. Sono entrambe forme per scrivere notaziona matematica. Quella ufficiale di *Mathematica* (se aprite un notebook con un editor di testo, ve ne accorgerete), è quella estesa, cioè la seconda. Possiamo usarla anche per scrivere i caratteri speciali all'interno delle stringhe:

```
In[71]:= stringa = "Questa contiene i caratteri α e β"
Out[71]= Questa contiene i caratteri α e β

In[72]:= FullForm[%]
Out[72]//FullForm= "Questa contiene i caratteri \[Alpha] e \[Beta]"
```

Come potete vedere, se scrivete la stringa qua sopra ottenete le lettere greche. Non posso farvelo vedere direttamente perchè *Mathematica* sostituisce automaticamente le lettere non appena sono scritti i giusti codici, ma il comando FullForm fa vedere a cosa corrispondono effettivamente le lettere greche. Se vogliamo includere nella stringa caratteri come \ oppure come ", dobbiamo farli precedere da \:

```
In[73]:= stringa1 = "Questo \ e questo " non si vedono
Out[73]= Questo  e questo  non si vedono
```

Invece, scritto in questa maniera:

```
In[74]:= stringa2 = "Questo \" e questo \\ si vedono"
Out[74]= Questo " e questo \ si vedono
```

Vedete come occorra in questi casi usare un piccolo accorgimento, ma non è niente che non abbiate anche visto in C. E, come in C, possiamo definire i caratteri speciali, come \n:

```
In[75]:= stringa = "E' un'unica stringa. \nMa vado a capo"
Out[75]= E' un'unica stringa.
         Ma vado a capo
```

Vedete come sia stato inserito il carattere speciale di ritorno a capo. Possiamo anche includere la tabulazione:

```
In[76]:= stringa = "Primo\tSecondo\tTerzo"

Out[76]= Primo    Secondo    Terzo
```

E con questo, credo di aver detto abbastanza sulle stringhe... Anche voi ne avete abbastanza? Ma sono cose tanto facili e carucce....

Comunque, una volta imparati questi comandi, se riuscite ad inventare qualcosa che non si possa risolvere con questi (e ovviamente usando le solite funzioni di *Mathematica*), esistono altri comandi avanzati per la programmazione e la gestione di file (come comandi per criptare i vostri files). Andate a vedere quello che vi servirà, d'accordo? Troverete sempre di tutto....