Questo mese lascio la parola ad un collega ed amico, buon esperto di Mathematica. *Questa esposizione sarà indubbiamente utile a quanti, esperti e non, desiderano navigare nel* mare magnum *della programmazione in* Mathematica

Francesco Romani

Programmare *Mathematica*

di Giuseppe Fiorentino

L'iniziale stupore suscitato da *Mathematica* come "supercalcolatrice interattiva", si muta spesso in sgomento di fronte all'arduo compito di programmare un sistema così complesso. Uno studio attento del sistema però rivela, dietro l'apparente complessità, la potenza di poche e semplici idee portate all'estremo della generalità. Con il pretesto di calcolare i valori della ben nota successione di Fibonacci:

F(0)=F(1)=1, F(n)=F(n-1)+F(n-2),

analizzeremo vari aspetti legati alla programmazione.

Fibonacci... forever!

Un indiscutibile pregio di *Mathematica* è la possibilità di definire funzioni e procedure utilizzando di fatto tutti gli stili di programmazione; questo, risparmiando all'utente l'imposizione di una forma mentis predefinita, favorisce la migrazione indolore verso lo stile funzionale, che meglio realizza lo spirito di *Mathematica*.

Lo stile imperativo

Chi usa i linguaggi imperativi come **C** o **Pascal**, abituato a pensare in termini di variabili, blocchi e iterazioni risolverebbe il problema più o meno così:

```
function Fib(n:Integer);
  var uno,due,temp,i:Integer;
  begin
     uno:=1;  due:=1;
     for i:=1 to n-1 do begin
        temp:=uno;
        uno :=uno+due;
        due :=temp
     end; (* for *)
     Fib:=uno
  end; (* Fib *)
```

Con *Mathematica* è possibile riprodurre tale armamentario in maniera familiare:

In[1]:=

```
fib1[n_] :=
 Module[{uno, due, temp, i},
        uno=1; due=1;
        For[i=1, i<n, i++,
            temp=uno;
            uno =temp+due;
            due =temp];
        uno] /; IntegerQ[n]
```

Il costrutto **/; condizione** permette di introdurre dei vincoli per l'applicabilità della definizione; la funzione booleana **IntegerQ[.]**, in questo caso, restringe **fib1** ai soli argomenti interi:

In[2]:=

```
{fib1[5], fib1[x]}
```

Out[2]=

```
{8, fib1[x]}
```

Per confrontarle, cronometreremo tutte le definizioni con un argomento campione:

In[3]:=

```
Timing[fib1[20]]
```

Out[3]=

```
{0.05 Second, 10946}
```

Uno stile procedurale migliore

Un uso migliore di *Mathematica* consente di riscrivere **fib1** in modo più elegante, inizializzando le variabili locali e utilizzando il costrutto iterativo **Do[.]** con l'assegnamento simultaneo:

In[4]:=

```
fib2[n_] := Module[{uno=1, due=1},
     Do[{uno, due}={uno+due, uno}, {n-1}];
     uno]     /; IntegerQ[n]
```

si eliminano sia la variabile di appoggio **temp** che quella di iterazione.

In[5]:=

```
Timing[fib2[20]]
```

Out[5]=

```
{0.0333333 Second, 10946}
```

Uno stile funzionale "naïf"

Chi allo stile imperativo preferisce il paradigma funzionale alla **LISP**:

```
(defun fib (n)
   (cond
    ((lessp n 2) 1)
    (t (+ (fib (- n 1)) (fib (- n 2))))))
```

trova in *Mathematica* una limpida implementazione, sgravata dal fardello delle tante parentesi. Ne sia dimostrazione l'espressione condizionale per definire **Fib(n)**:

In[6]:=

```
fib3[n_ /; IntegerQ[n]] =
  If[ n<2, 1, fib3[n-1]+fib3[n-2]];
```

La condizione di applicabilità, posta nella testa della funzione, consente l'uso dell'assegnamento immediato: ogni tentativo di riapplicare immediatamente la regola viene arrestato dalla presenza di un argomento simbolico.

Prendendo i tempi...

In[7]:=

```
Timing[fib3[20]]
```

Out[7]=
```
{50.9333 Second, 10946}
```
...appare chiaro che il tutto è, a dir poco, inefficiente.
Il motivo è da ricercarsi nelle innumerevoli chiamate fatte per ottenere dei valori già calcolati; la chiamata di **fib3[n-2]** procede ciecamente senza utilizzare quanto ottenuto nel calcolo di **fib3[n-1]**. Il difetto, ovviamente, non è nello stile impiegato ma nell'uso ingenuo che ne è stato fatto; a questo porremo adeguato rimedio nel seguito...

Lo stile dichiarativo

Anche gli amanti dei linguaggi dichiarativi si troveranno a casa la possibilità di fornire al sistema dei *fatti* relativi a valori particolari e delle *regole* applicabili in generale richiama l'uso delle clausole tipiche del **Prolog**:

```
fib(0,1).
fib(1,1).
fib(n,f):-n>1,fib(n-1,x),fib(n-2,y),f is x+y.
```
Così, i "fatti" per i valori particolari 0 e 1
In[8]:=
```
fib4[0] = 1;
fib4[1] = 1;
```
e la "regola" di calcolo per **n** generico
In[9]:=
```
fib4[n_Integer]=fib4[n-1]+fib4[n-2];  (*1*)
```
danno una buona definizione per **Fib(n)**; la definizione opportuna viene selezionata di volta in volta in funzione del parametro (*Pattern Matching*). Il pattern matching esteso di *Mathematica* consente poi definizioni come la (*1*) dove il vincolo è incluso nel pattern. È possibile conoscere l'effettivo ordine di applicazione chiedendo "delucidazioni" al sistema con **?argomento**:
In[10]:=
```
?fib4
Global`fib4
fib4[0] = 1
fib4[1] = 1
fib4[n_Integer] =
  fib4[-2 + n] + fib4[-1 + n]
```
In generale, la definizione da applicare è individuata come segue: vengono tentate in ordine di generalità crescente utilizzando la prima compatibile con il valore dell'argomento.
Anche **fib4**, messa alla prova,
In[11]:=
```
Timing[fib4[20]]
```
Out[11]=
```
{24.1 Second, 10946}
```
manifesta l'inefficienza dovuta alle chiamate inutili.

Definizioni che ricordano i valori

Il paradigma dichiarativo, come si è visto, prevede la definizione di fatti, registrati nel database,
In[12]:=
```
fib5[0] = fib5[1] = 1;   (*2*)
```
e regole usate come procedimenti di calcolo per gli altri valori; le regole però non aggiungono *conoscenza* al database.
Per evitare le chiamate superflue si dovrebbero memorizzare i valori calcolati; questo non è realizzabile né con un assegnamento immediato (=), che aggiungerebbe un ulteriore fatto o regola, né con un semplice assegnamento ritardato (:=), il cui effetto sarebbe il dilazionare la definizione; un uso "furbo" di entrambi tuttavia...
La (*2*) dimostra che l'assegnamento è a sua volta un'espressione che restituisce il valore assegnato per cui, continuando lo scioglilingua, se alla definizione generale **fib5[n_Integer]** si assegna (in maniera ritardata) l'assegnamento (immediato questa volta):

fib5[n]=fib5[n-1]+fib5[n-2],

otteniamo una funzione
In[13]:=
```
fib5[n_Integer] :=
 fib5[n] = fib5[n-1]+fib5[n-2]
```
il cui effetto collaterale, ad ogni chiamata, è l'aggiunta di una esplicita definizione per **fib5[n]** (valore particolare, senza il pattern _) utilizzando il valore calcolato!
La ricorsione si occupa del resto, definendo **fib5** per tutti i valori minori di **n**.
Ad esempio, al database inizialmente minimo
In[14]:=
```
?fib5
Global`fib5
fib5[0] = 1
fib5[1] = 1
fib5[n_Integer] :=
  fib5[n] = fib5[n - 1] + fib5[n - 2]
```
la chiamata
In[15]:=
```
Timing[fib5[20]]
```
Out[15]=
```
{0.133333 Second, 10946}
```
aggiunge tutti i valori trovati durante il calcolo
In[16]:=
```
?fib5
Global`fib5
fib5[0] = 1
fib5[1] = 1
...
fib5[20] = 10946
fib5[n_Integer] :=
  fib5[n] = fib5[n - 1] + fib5[n - 2]
```
consentendo la buona performance misurata e risposte fulminee per valori già calcolati o vicini:
In[17]:=
```
Timing[fib5[25]]
```
Out[17]=
```
{0.0166667 Second, 121393}
```
Un tempo esponenziale, dovuto alle chiamate inutili, è stato barattato con un'occupazione lineare di memoria, per le definizioni aggiuntive. Bisogna dire comunque che non è stato un buon affare, visto che il problema, come evidenziato dalla formulazione imperativa, è risolubile senza impegnare grandi risorse; vi sono però dei casi in cui la complessità della funzione da calcolare rende prezioso ogni valore già trovato.

Verso un vero stile funzionale

L'idea vincente dell'approccio imperativo è quella di procedere *in avanti* nel calcolo anziché a ritroso, evitando così le chiamate inutili; si può riprendere tale strada notando che per avanzare basta ricordare solo gli ultimi due termini.

Con la funzione ausiliaria
In[18]:=
```
fibStep[{a_, b_}] = {a+b, a};
```
per passare dalla coppia `{Fib(n),Fib(n-1)}` alla successiva `{Fib(n+1),Fib(n)}` e l'operatore predefinito di composizione `Nest[funzione, valore iniziale, numero di applicazioni]`:
In[19]:=
```
Nest[fun, x0, 3]
```
Out[19]=
```
fun[fun[fun[x0]]]
```
è possibile calcolare **Fib(n)** partendo da `{1,0}`, componendo `fibStep` **n** volte e prendendo la prima componente del risultato con l'operatore `[[.]]`.
In[20]:=
```
fib6[0] = 1;
fib6[n_Integer?Positive] :=
     Nest[fibStep, {1,0}, n] [[1]]
```
Come si vede, il pattern matching esteso consente di testare il valore dell'argomento oltre che il tipo; combinando opportunamente le definizioni si controlla in modo raffinato il dominio di definizione restringendolo, come in questo caso, ai naturali.
Prendendo i tempi
In[21]:=
```
Timing[fib6[20]]
```
Out[21]=
```
{0.05 Second, 5.21614}
```
si conferma il ritorno all'efficienza.

Le funzioni pure

La possibilità, presente in ogni ambiente funzionale che si rispetti, di definire delle funzioni pure (lambda espressioni) rende inutili le funzioni ausiliarie; la funzione "quadrato" si esprime in *Mathematica* come `Function[x, x^2]` e si usa come una funzione esplicita a tutti gli effetti:
In[22]:=
```
Function[x, x^2][10]
```
Out[22]=
```
100
```
Nelle funzioni pure il pattern matching non è consentito per cui, volendo riscrivere `fibStep` come una funzione pura, la lista `{Fib(k),Fib(k-1)}` va passata come un unico argomento; le componenti rimangono accessibili tramite l'operatore di selezione `[[.]]` visto sopra:
```
     Function[x, {x[[1]]+x[[2]], x[[1]]}]
```
Utilizzando le funzioni pure, si ha:
In[23]:=
```
fib7[0] = 1;
fib7[n_Integer] :=
  Nest[Function[x, {x[[1]]+x[[2]], x[[1]]}],
       {1,0}, n] [[1]]
```
con prestazioni
In[24]:=
```
Timing[fib7[20]]
```
Out[24]=
```
{0.05 Second, 10946}
```
più o meno equivalenti a `fib6`.

Programmazione funzionale avanzata

È possibile definire funzioni pure con più argomenti

```
Function[{x1,...,xn}, <body>][a1,...,an],
```

ma il non poter catturare, via pattern matching, le componenti di una lista porta a definizioni non proprio eleganti come `fib7`; l'argomento `[{Fib(k),Fib(k-1)}]` è infatti ben diverso dalla coppia `[Fib(k),Fib(k-1)]` dove si hanno due argomenti distinti.
Vi è un modo, offerto dall'operatore `Apply[head,espr]`, per cambiare la testa di `espr` con `head`; si può ad esempio trasformare `{x,y}` (`List[x,y]`) in x^y (`Power[x,y]`) con
In[25]:=
```
Apply[Power, {x, y}]
```
Out[25]=
x^y
Per cui, definita la funzione pura
In[26]:=
```
Function[{x,y},{x+y,x}]    (*3*)
     [Fib[k],Fib[k-1]]
```
Out[26]=
```
{Fib[-1 + k] + Fib[k], Fib[k]}
```
che restituisce la lista `{Fib(k+1),Fib(k)}` dagli argomenti `[Fib(k),Fib(k-1)]`, utilizzando una versione modificata di `Nest` che "applica" la funzione ad ogni composizione:
In[27]:=
```
NestApply[f_,a_,n_]:=
    Nest[Function[x, Apply[f, x]], a, n]
```
si calcola **Fib(n)** applicando la (*3*) `n-1` volte con `NestApply` partendo da `{1,1}`:
In[28]:=
```
fib8[0] = 1;
fib8[n_Integer?Positive] :=
   NestApply[Function[{x, y}, {x+y, x}],
             {1,1},n-1] [[1]]
```
Si recupera il controllo del dominio risparmiando anche un'iterazione.
In[29]:=
```
Timing[fib8[20]]
```
Out[29]=
```
{0.0833333 Second, 10946}
```

Programmazione funzionale criptica!

Volendo fornire delle "scorciatoie sintattiche" spesso vengono introdotti nei linguaggi di programmazione dei veri e propri boomerang semantici tali, se adoperati male, da rendere incomprensibile (perfino all'autore) anche il programma più semplice (i programmatori in **C** ne sanno qualcosa...).
In un contesto funzionale-simbolico il risultato può essere anche più funesto.

Funzioni anonime

In *Mathematica* una scorciatoia sintattica consente di fare a meno dei parametri formali; i parametri attuali nelle funzioni sono individuati posizionalmente con `#1`, `#2...`
Il solito quadrato si esprime in tale notazione con `(#^2)&` :
In[30]:=
```
(#^2)& [10]
```
Out[30]=

```
100
```

dove # sta per **#1**, il primo parametro, e **&** marca la fine della definizione. Riscrivendo la **Function** in **fib7** come **{#[[1]]+#[[2]],#[[1]]}&** si ottiene la definizione:

In[31]:=
```
fib7bis[0] = 1;
fib7bis[n_Integer?Positive]:=
   Nest[{#[[1]]+#[[2]], #[[1]]}&,
        {1,1}, n-1] [[1]]
```
non proprio cristallina, ma... funzionante (!) infatti:

In[32]:=
```
Timing[fib7bis[20]]
```
Out[32]=
```
{0.0166667 Second, 10946}
```

Programmazione geroglifica

Le cose peggiorano quando si affiancano delle forme infisse (con una sintassi non sempre felice) agli operatori funzionali prefissi; uno di questi è **Apply**, la forma **Apply[fun, arg]** è equivalente a quella infissa **fun @@ arg**.

È chiaro che componendo funzioni pure con operatori infissi si può finire con l'ottenere una definizione

In[33]:=
```
fib8bis[n_Integer]:=
   Nest[{#1+#2,#1}& @@ # &,{1,0},n] [[1]]
```
equivalente a **fib8** ma meno comprensibile di una stele di Luxor! Sbirciando come il sistema ha registrato la funzione:

In[34]:=
```
?fib8bis
Global`fib8bis
fib8bis[n_Integer] :=
  Nest[Apply[{#1 + #2, #1} & , #1] & ,
    {1, 0}, n][[1]]
```
ritorna comprensibile la composizione di funzioni.

In[35]:=
```
Timing[fib8bis[20]]
```
Out[35]=
```
{0.0333333 Second, 10946}
```

Follia funzionale

L'apoteosi della programmazione criptica si tocca utilizzando le funzioni pure per le definizioni ricorsive; infatti, se **#1**, **#2**,...**#n** rappresentano il primo, secondo, ... ennesimo parametro, è "chiaro" (???!) che **#0** sta per la funzione stessa...

Per cui,

In[36]:=
```
CrazyFib := If[#<2, 1, #0[#-1]+#0[#-2]]&
```
calcola ancora Fibonacci, sia pure in modo criptico...

In[37]:=
```
Timing[CrazyFib[20]]
```
Out[37]=
```
{36.0167 Second, 10946}
```
... ed inefficiente!

Le regole di riscrittura

Riflettendo sugli esempi precedenti, emerge l'abilità di *Mathematica* nel riconoscere e manipolare espressioni; tale capacità è utilizzabile in maniera "nativa" usando *regole di trasformazione* del tipo: **pattern -> expr**

L'insieme di regole di trasformazione:

rules = {pattern$_1$->expr$_1$,..., pattern$_n$->expr$_n$}

applicato ad un'espressione con **espr /. rules** trasforma **espr** rimpiazzando simultaneamente con **expr$_k$** tutte le sottoespressioni compatibili con **pattern$_k$**:

In[38]:=
```
{x^y, y^x} /. {x->y, y->x}
```
Out[38]=

$\{y^x, x^y\}$

La trasformazione **{a_,b_}->{a+b,a}** è quindi equivalente alla **fibStep** del punto 2.6 inoltre, poiché con **//.** la trasformazione si può ripetere fino a quando il risultato non cambia, la regola "trucco":

In[39]:=
```
fibRule = {a_,b_,n_?Positive}->{a+b,a,n-1};
```
combinata con la definizione "furba":

In[40]:=
```
fib9[n_Integer] :=
  ({1,1,n-1} //. fibRule)[[1]]
```
definiscono, ancora una volta, un effettivo (ed efficiente) procedimento di calcolo per **Fib(n)**:

In[41]:=
```
Timing[fib9[20]]
```
Out[41]=
```
{0.05 Second, 10946}
```
È degna di nota anche la restrizione del dominio realizzata dalla combinazione delle definizioni.

Approccio Mat(h)ematico

Il modo naturale per calcolare i numeri di Fibonacci utilizzando le capacità algebriche e numeriche di *Mathematica* passa per la soluzione dell'equazione alle differenze:

F(n+2)-F(n+1)-F(n)=0

F(0)=F(1)=1

che fornisce l'espressione esplicita per il termine generico

$$\mathrm{Fib}(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}. \qquad (*4*)$$

Da qui si dipartono due strade: la prima, algebrica, espande brutalmente e poi semplifica (*4*) fino ad ottenere il risultato

In[42]:=
```
fibAlgebrico[n_Integer]:=
 With[{r5 = Sqrt[5], n1=n+1},
   Expand[((1+r5)^n1-(1-r5)^n1)/(2^n1 r5)]]
```
In[43]:=
```
Timing[fibAlgebrico[20]]
```
Out[43]=
```
{0.283333 Second, 10946}
```
La seconda strada è quella numerica: osservando che il secondo termine in (*4*) è minore di 1/2, basta arrotondare il valore dato dal primo addendo. Dato che (*4*) dimostra la crescita esponenziale di **Fib(n)**, i numeri coinvolti devono essere sufficientemente precisi per cui bisogna utilizzare delle approssimazioni con un numero crescente di cifre:

In[44]:=
```
fibNumerico[n_Integer] :=
  With[{r5 = Sqrt[5]},
```

```
    Round[N[((1+r5)/2)^(n+1)/r5, n]]]
```

In[45]:=

```
Timing[fibNumerico[20]]
```

Out[45]=

```
{0.0333333 Second, 10946}
```

La superiorità del secondo metodo emerge al crescere della dimensione:

In[46]:=

```
Timing[fibAlgebrico[300]]
```

Out[46]=

```
{2.96667 Second,
  3595793252065835609617656651721890990 52\
   3672143092672322555898 01}
```

In[47]:=

```
Timing[fibNumerico[300]]
```

Out[47]=

```
{0.2 Second,
  3595793252065835609617656651721890990 52\
   3672143092672322555898 01}
```

Il primo approccio, calcolando le potenze elevate di un binomio, giunge presto a espressioni ciclopiche tutt'altro che semplici da manipolare.

Usando il cervello!

Chiudiamo l'estenuante carrellata mostrando come le soluzioni migliori a volte arrivano da dove meno le si aspetta.

La moltiplicazione russa

Si dice che i contadini di alcune zone della Russia, ancora oggi, effettuino la moltiplicazione con il seguente algoritmo:

$$a \times b = \begin{cases} (a+a) \times (b/2) & b \text{ pari} \\ a + a \times (b-1) & b \text{ dispari} \end{cases}$$

assumendo, come al solito, ax0=0.

Lo schema, a ben pensarci, è generalizzabile a tutti quei casi in cui dato un operatore binario ***op***, dotato di un elemento neutro ***nul***, bisogna calcolare *a op a op...op a* (*n* volte).

Tale generalizzazione si può formulare ricorsivamente:

In[48]:=

```
russa[op_, a_, 0 , nul_] = nul;
russa[op_, a_, n_, nul_] :=
  If[EvenQ[n],
     russa[op, op[a, a], n/2, nul],
     op[a,russa[op, a, n-1, nul]] ]
```

realizzando, ad esempio, moltiplicazione e elevazione a potenza utilizzando somme e prodotti:

In[49]:=

```
{russa[Plus,a,0,10], russa[Times,a,1,10]}
```

Out[49]=

$$\{10\ a,\ a^{10}\}$$

Le "matrici di Fibonacci"

Una delle tante curiosità legate ai numeri di Fibonacci è la seguente: disponendo tre termini consecutivi nella matrice:

In[50]:=

```
{{Fib[n],Fib[n-1]},{Fib[n-1],Fib[n-2]}};
```

e moltiplicandola per `{{1,1},{1,0}}`,

In[51]:=

```
%.{{1,1},{1,0}} // MatrixForm
```

Out[51]//MatrixForm=

```
Fib[-1+n]+Fib[n]       Fib[n]
Fib[-2+n]+Fib[-1+n]    Fib[-1+n]
```

si ottiene la matrice "successiva" nella sequenza. Pensandoci bene vuol dire che la matrice di partenza altro non era che l'ennesima potenza di `{{1,1},{1,0}}`.

Fibonacci alla Russa

Mettendo insieme le cose, si può calcolare **Fib(n)** elevando `{{1,1},{1,0}}` all'ennesima potenza prelevando la componente in posizione 1,1:

In[52]:=

```
fibRussa[n_] := russa[Dot,
  {{1,1},{1,0}},n,{{1,0},{0,1}}][[1,1]]
```

Il numero di passi effettuato risulta logaritmico in **n**:

In[53]:=

```
Timing[fibRussa[3000]]
```

Out[53]=

```
{0.25 Second, 6643904603669600722802178478\
   6602838424416351245278325940557976554 26\
   2121416121925739644981098299982039113 22\
   2680280946513244634933199440943492601 90\
   4534272374918853031699467847355132063 51\
   0109961938297318162258568733693978437 35\
   2789755548948684172613173381434012917 56\
   2245042160510102589717323599066277020 37\
   5643878651753054710112374884914025268 61\
   2010403264702514559895667590213501056 69\
   0978312495943646982555831428970135422 71\
   5178460286571078062467510705656982282 05\
   4284666032181383889627581975328137149 18\
   0900441221912485637512169481172872421 36\
   6781457732661852147835766185901896731 33\
   5484017840319755996905651079170985914 41\
   73304364898001}
```

e porta a performance di tutto rispetto; specialmente se confrontata con il modo naturale di effettuare il calcolo usando l'operatore predefinito `MatrixPower`

In[54]:=

```
fibFast[n_] :=
  MatrixPower[{{1,1},{1,0}}, n] [[1,1]]
```

In[55]:=

```
Timing[fibFast[3000]]
```

Out[55]=

```
{0.15 Second, ...}
```

Conclusioni

Dicevamo all'inizio che la vera forza di *Mathematica* consiste nel portare all'estrema generalità poche idee, semplici ma potenti; tale approccio risulta spesso vincente anche nella risoluzione di problemi concreti. Nonostante le potenzialità offerte dal programma, questo va comunque considerato uno strumento; la capacità di analizzare i problemi ed individuare delle soluzioni creative è ancora una prerogativa umana, per il momento...

MC

Francesco Romani è raggiungibile tramite Internet all'indirizzo romani@di.unipi.it
Giuseppe Fiorentino, Dottorando in Informatica presso l'Università di Pisa, è raggiungibile all'indirizzo di posta elettronica: fiorent@di.unipi.it