LA POTENZA DEL PATTERN MATCHING

Giuseppe Fiorentino

Uno degli aspetti di *Mathematica* che più facilmente sfugge agli utilizzatori di linguaggi tradizionali è il pattern matching. Si perdono così la capacità di riconoscere intere strutture assegnando dei nomi alle componenti e la possibilità di definire le funzioni per casi, lasciando al sistema la selezione della regola "giusta".
Un esempio, ormai trito, è dato dal fattoriale definito dalla regola particolare

In[1]:=
```
fatt[0] = 1;
```
e da quella generale

In[2]:=
```
fatt[n_Integer?Positive] := n fatt[n-1]
```
Il tutto funziona grazie al fatto che *Mathematica* ordina, e tenta di applicare, le regole per generalità crescente del pattern: dal caso particolare alla regola generale.
Le possibilità, come si può intuire, sono enormi; le scopriremo esplorando lo strano mondo degli algoritmi di ordinamento, campo ideale per l'impiego di ogni tipo di *diavoleria.*
Per favorire un uso "massiccio" del pattern matching eviteremo, nei limiti del possibile, le funzioni predefinite e le ottimizzazioni per non appesantire la trattazione. Protagonista sarà il pattern matching che, come vedremo, se la caverà egregiamente.

Definizioni utili

Prima di partire è opportuno attrezzarci con alcune definizioni utili; l'armamentario comprende delle liste per testare gli algoritmi proposti e delle funzioni per verificare l'ordinamento di una lista e visualizzare graficamente i risultati.

Le funzioni test

Il parametro principe nella valutazione dell'efficienza di un algoritmo di ordinamento è senz'altro la complessità, intesa come una misura del tempo di esecuzione richiesto al crescere della dimensione della lista da ordinare. Tuttavia, dato che spesso l'efficienza dipende sensibilmente dal modo con cui si presentano i dati, utilizzeremo quattro liste diverse per testare gli algoritmi proposti.
La prima, paradossalmente critica per alcuni algoritmi, contiene i valori da 1 a n già ordinati ed è definita da:

In[3]:=
```
test[1, n_] := Range[n]
```
la seconda presenta gli stessi elementi in ordine inverso,

In[4]:=
```
test[2, n_] := Reverse[Range[n]]
```
La terza si ottiene "*effettuando una visita anticipata di un albero binario di ricerca perfettamente bilanciato contenente i valori 1..n*" ... in poche parole, è il caso ottimo per gli algoritmi di ordinamento alla quicksort che usano il primo elemento della lista come *pivot*!
A dispetto dell'apparente complicazione, la lista si ottiene facilmente innescando con

In[5]:=
```
test[3, n_] := Flatten[mixer[1,n]]
```
il "frullatore ricorsivo" seguente che genera la lista usando `i` e `j` come valori estremi della sequenza da generare

In[6]:=
```
mixer[i_,i_]:={i}
mixer[i_,j_]:={}    /; i>j
mixer[i_,j_]:=
  {#,mixer[i,#-1],mixer[#+1,j]}&
    [Floor[(i+j+1)/2]]
```
Come si vede, la definizione usa un vincolo per riconoscere il caso `i>j` e il pattern matching per il caso `i=j`. Si noti anche il calcolo del punto medio fatto *una tantum* grazie alla notazione funzionale
Infine, la quarta ed ultima delle liste di test

In[7]:=
```
test[4, n_] := pingPong[1,n,ping,{}]
```
attiva un ping-pong a quattro parametri

In[8]:=
```
pingPong[i_,i_,_,{s___}]    :={i,s}
pingPong[i_,j_,_,l_]        :=l /;i>j
pingPong[i_,j_,ping,{s___}]:=
  pingPong[i+1,j-1,pong,{i,s,j}]
pingPong[i_,j_,pong,{s___}]:=
  pingPong[i+1,j-1,ping,{j,s,i}]
```
che, rimbalzando tra la terza e la quarta definizione, accumula valori grandi e piccoli nella lista in quarta posizione; la "partita" finisce nella prima e seconda definizione quando i valori di `i` e `j` si eguagliano o si invertono, indipendentemente dal valore del terzo parametro (il segnaposto _ rappresenta un qualsiasi valore singolo).

Verifica di ordinamento

Per verificare che gli "ordinatori" che andremo definendo ordinano davvero, controlleremo i risultati con la funzione booleana `inOrdine` che definiremo per casi, sistemando innanzitutto quelli semplici come la lista vuota:

In[9]:=
```
inOrdine[{}] = True;
```
che è chiaramente ordinata, così come la lista che contiene un solo elemento

In[10]:=
```
inOrdine[{_}] := True
```
Il caso generale infine viene catturato dalla regola

In[11]:=
```
inOrdine[{x_,y_,z___}]:=
 (x<=y) && inOrdine[{y,z}] (*1*)
```
stabilendo che la lista `{x,y,z}`, formata dagli elementi `x` e `y`

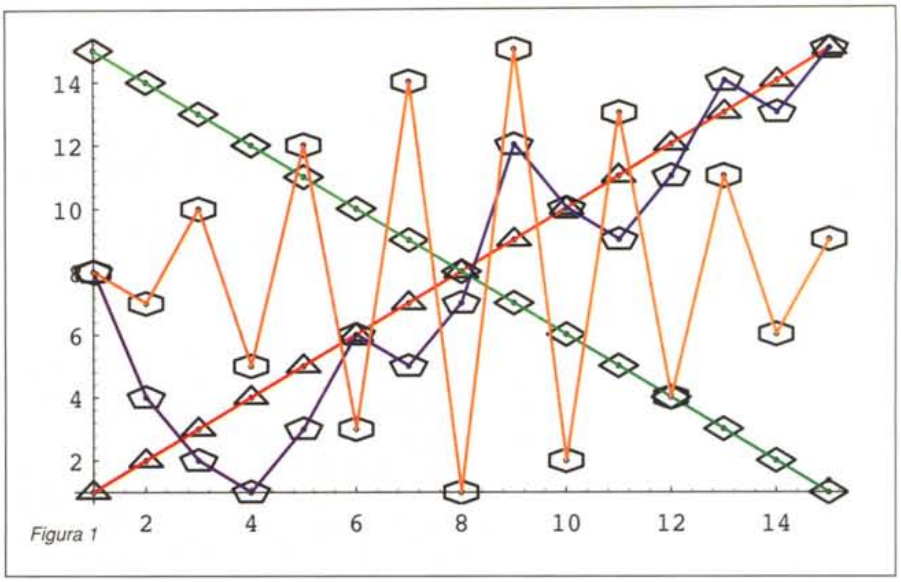


Figura 1

e dalla sequenza **z** (i tre underscore ___ denotano zero o più elementi), è ordinata se lo è la coppia (**x**, **y**) e, ricorsivamente, la lista **{y,z}**.
La correttezza del programma è garantita dal fatto che la sequenza **z** in **(*1*)** può essere vuota; usando due underscore invece, notazione per le liste proprie, il programma non è più in grado di catturare le liste con due elementi.
Dato che *Mathematica* limita, per default, il numero di chiamate ricorsive a 255 bisogna aumentare questo limite
In[12]:=

```
$RecursionLimit = 100000;
$IterationLimit = 100000;
```

per non avere problemi con la funzione **inOrdine** che, grosso modo, effettua tante chiamate ricorsive quanti sono gli elementi nella lista da controllare.

L'output grafico

Se è vera la massima che un disegno vale quanto mille parole, figuriamoci quando le "chiacchiere" sono dei numeri! Valuteremo allora la complessità qualitativamente mostrando i grafici dei tempi di elaborazione al variare della dimensione e della distribuzione dei dati.
Per la visualizzazione di più funzioni discrete sullo stesso grafico utilizzeremo **MultipleListPlot** definita nel package standard
In[13]:=

```
Needs["Graphics`MultipleListPlot`"]
```

e la funzione **mlp**
In[14]:=

```
mlp[{l__List}, opt___] :=
 MultipleListPlot[l,opt,
   PlotJoined->True,
   PlotRange->All]
```

i cui argomenti sono una lista di liste (almeno una) e delle opzioni.
Utilizzeremo anche il package dei colori per distinguere le liste test con colori:
In[15]:=

```
Needs["Graphics`Colors`"]
 $LineStyles=
   {{Red},{Green},{Blue},{Orange}};
```

e simboli diversi:
In[16]:=

```
$DotShapes= Table[MakeSymbol[
   RegularPolygon[i,0.025]],{i,3,6}];
```

(poligoni regolari con numero di lati da tre a sei).
Visualizziamo, ad esempio, le funzioni test di lunghezza 15
In[17]:=

```
mlp[Table[test[i,15],{i,4}],
   AxesOrigin->{1,1}];
```

(Vedi figura 1)
Finalmente, con la funzione
In[18]:=

```
contr[f_, l_] := Module[{t,s},
                 {t,s}=Timing[f[l]];
                 {t/Second, inOrdine[s]}]
```

che applica **f** alla lista **l** restituendo il tempo di esecuzione e l'ordinamento del risultato, definiamo la funzione di valutazione principale che visualizza i tempi di esecuzione, controllando l'avvenuto ordinamento di tutte le liste:
In[19]:=

```
valuta[f_, n_:10, p_:10] :=
  Module[{t,b},
   {t,b}=Transpose[
    Table[contr[f,test[i, j p]],
      {j, 0, n}, {i, 4}],3,2,1}];
    Print["Ordinate:",If[And@@
      Flatten[b],"Si","No"]];
    mlp[t,Ticks->
     {Table[{i+1,i p},{i,0,n}], Automatic},
      AxesOrigin->{1,0}  ] ]
```

I parametri sono la funzione da valutare il numero di campioni e il passo (inizializzati a 10).

Il bubblesort

Il più semplice degli algoritmi di ordinamento, il bubblesort, applica ripetutamente la *regola aurea* **(BS)**

se ci sono elementi adiacenti in ordine errato scambiali

Ordinamento con regola

Mathematica cattura facilmente la regola **(BS)** con il pattern per le sequenze di lunghezza arbitraria **(___)**: gli elementi adiacenti sono quelli preceduti e seguiti da sequenze.
In[20]:=

```
ruleSort[{a___,c_,b_,d___}] :=
  ruleSort[{a,b,c,d}]    /; c>b
```

Nell'applicare la regola, vengono tentate tutte le lunghezze per la sequenza **a___** fino a trovare, se ci sono, i primi due elementi successivi **b** e **c** in ordine "errato"; a questo punto il pattern {sequenza1, elemento1, elemento2, sequenza2 } viene trasformato in {sequenza1, elemento2, elemento1, sequenza2} in cui la coppia (**b**, **c**) è stata *sistemata*.
Per uscire vivi dalla ricorsione, si definisce
In[21]:=

```
ruleSort[l_List] := l
```

per restituire l'argomento quando la prima regola non è più applicabile (cioè quando la lista è ordinata).
Purtroppo, a dispetto della semplicità, le prestazioni sono a dir poco disastrose...
In[22]:=

```
valuta[ruleSort];
Ordinate: Si
```

(Vedi figura 2)
Infatti, l'ordine quadratico tipico del bubblesort nel caso pessimo, viene peggiorato dal fatto che, dopo gli scambi, l'algoritmo riparte alla ricerca del nuovo punto di applicazione ponendo sequenza1={}; questo *regala* un altro ordine alla complessità che arriva a circa `n^3` passi per ordinare una lista lunga `n`.

Il bubblesort vero

L'alternativa è quella di effettuare davvero le spazzate del bubblesort utilizzando una tecnica molto utile: il *travaso*.

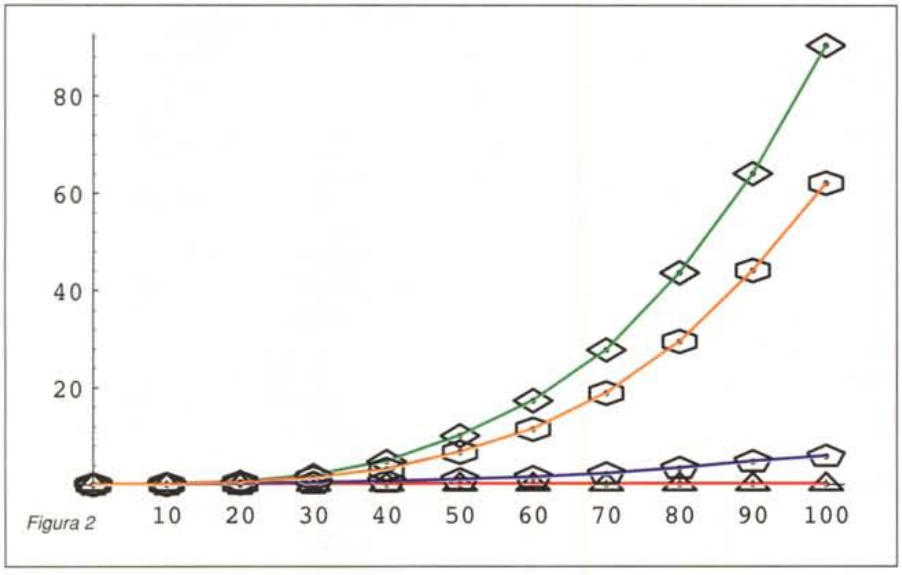


Figura 2

Vediamone un esempio: la funzione
In[23]:=

```
bubbleSort[l_List] := bubble-
Sort[l,{},0]
```

pone la lista `l` nel primo argomento, che chiameremo *sorgente*, da dove, effettuando gli opportuni scambi, verrà riversata nel secondo, la *destinazione*. Il terzo parametro conterà il numero di scambi effettuati in una spazzata.
Alla base della c'è la sorgente completamente riversata nella destinazione; a questo punto, se non ci sono stati scambi, la lista è ordinata e viene restituita:
In[24]:=

```
bubbleSort[{},l_,0 ] := l
```

altrimenti, si travasa la destinazione nella sorgente e si riparte:
In[25]:=

```
bubbleSort[{},l_,n_] := bub-
bleSort[l,{},0]
```

Seguono quindi le definizioni per il caso generale dove, a parte il caso in cui la sorgente è ridotta ad un singoletto,
In[26]:=

```
bubbleSort[{x_},{l___},n_] :=
  bubbleSort[{},{l,x}, n]
```

si confrontano i primi due elementi della sorgente muovendone il minore nella destinazione:
In[27]:=

```
bubbleSort[{x_,y_,z___},{l___},n_]:=
 bubbleSort[{y,z},{l,x},n] /; x<y
```

aggiornando gli scambi quando serve
In[28]:=

```
bubbleSort[{x_,y_,z___},{l___},n_]:=
  bubbleSort[{x,z},{l,y},n+1]
```

Il tutto, sebbene sia meno elegante, offre prestazioni decisamente migliori, anche se permangono notevoli differenze dovute alle diverse distribuzioni.
In[29]:=

```
valuta[bubbleSort];
Ordinate: Si
```

(Vedi figura 3)

L'insertion sort

Un altro algoritmo di ordinamento prevede di estrarre man mano elementi dalla *sorgente* inserendoli in posizione corretta nella *destinazione* (inizialmente vuota).
Cominciamo con l'inserimento di un elemento `x` in una lista ordinata; si effettua un *travaso* estraendo dalla sorgente un elemento alla volta, per scaricarlo nella destinazione se è minore di `x`
In[30]:=

```
ins[x_,{t_,c___},{s___}]:=
 ins[x,{c},{s,t}]     /;x>t
```

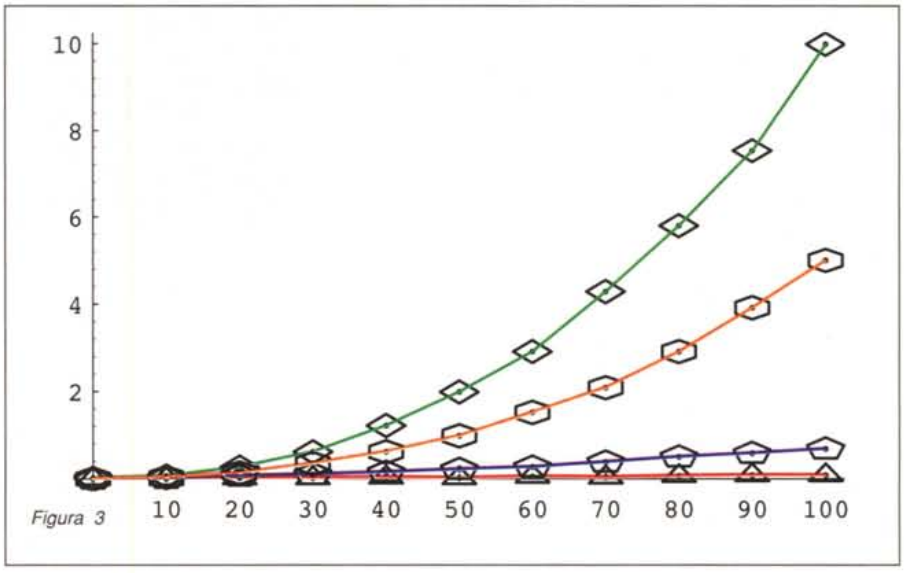


Figura 3

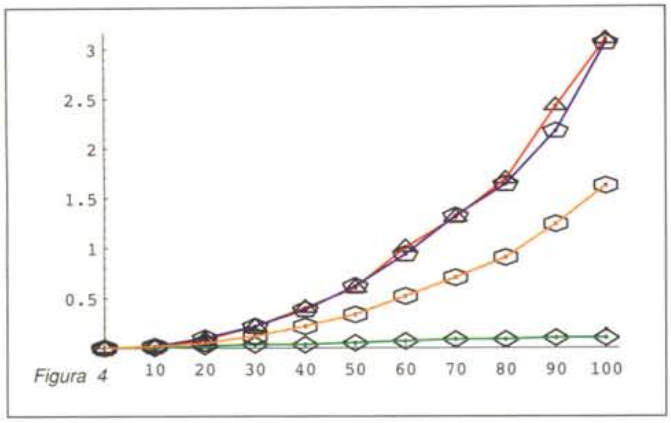


Figura 4

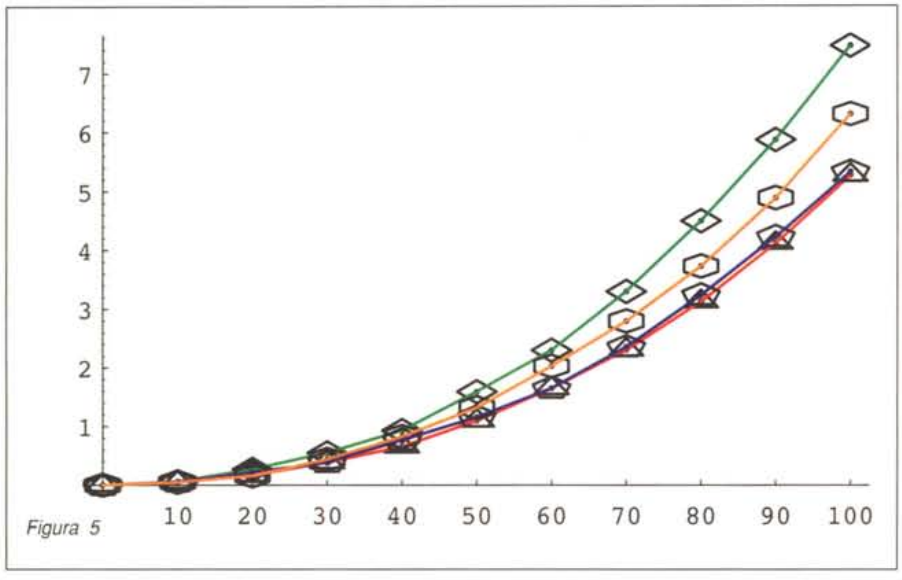


Figura 5

o impacchettando tutto quando arriva la posizione giusta per l'inserimento

In[31]:=

```
ins[x_,{t_,c___},{s___}]:={s,x,t,c}
```

oppure ponendo **x** in fondo quando la sorgente è esaurita

In[32]:=

```
ins[x_,{},{s___}]:={s,x}
```

Disponendo di **insert**, l'algoritmo di ordinamento è presto fatto; basta la funzione di innesco

In[33]:=

```
insSort[l_List] := insSort[l,{}]
```

che inizializza la destinazione chiamando la funzione ricorsiva **insSort**:

In[34]:=

```
insSort[{}, l_] := l
insSort[{t_,c___},l_]:=
      insSort[{c},ins[t,l,{}]]
```

che stacca ripetutamente la testa dalla sorgente inserendola nella destinazione (alle liste la testa ricresce...).

insSort all'opera rivela che il caso pessimo è, guarda caso, rappresentato dalle liste (parzialmente) ordinate,

In[35]:=

```
valuta[insSort];
```

```
Ordinate: Si
```

(Vedi figura 4)

gli elementi infatti finiscono sempre in fondo alla destinazione; l'opposto accade per la lista ordinata al contrario dove gli elementi si fermano sempre nella prima posizione.

Il selection sort

Il selection sort costruisce la lista ordinata estraendo ripetutamente il minimo dalla lista originaria. Cominciamo con la funzione **estrMin** che trattiene il valore minimo della lista man mano che la travasa dalla seconda posizione alla terza:

In[36]:=

```
estrMin[x_, {t_, c___}, {s___}] :=
 estrMin[x,{c},{s,t}] /; t>=x
estrMin[x_, {t_, c___}, {s___}] :=
 estrMin[t,{c},{s,x}]
```

restituendo la coppia {minimo, lista residua} a travaso ultimato

In[37]:=

```
estrMin[x_, {}, l_] := {x,l}
```

La funzione principale infine, dopo aver inizializzato la destinazione con la lista vuota,

In[38]:=

```
selSort[l_List] := selSort[l,
{}]
```

ed aver trattato il caso terminale,

In[39]:=

```
selSort[{}, l_] := l
```

per utilizzare le componenti restituite da **estrMin**

In[40]:=

```
selSort[{t_, c___}, l_] :=
   lHelp[estrMin[t,{c},{}],l]
```

si fa aiutare dalla funzione ausiliaria **selHelp**

In[41]:=

```
selHelp[{x_,l_},{s___}]:=sel-
Sort[l,{s,x}]
```

che aggiorna la destinazione e fa ripartire l'algoritmo con quanto rimane della lista originaria

Dalle prestazioni:

In[42]:=

```
valuta[selSort];
```

```
Ordinate: Si
```

(Vedi figura 5)

si nota che la distribuzione dei dati non altera granché il risultato il che è ovvio dato che per estrarre il minimo la lista va sempre guardata per intero.

Si poteva fare a meno della funzione ausiliaria? **SI**, ma con un risultato non altrettanto lineare ...

In[43]:=

```
selSort2[{}, {}, m_, {s___}] := {m,s}
selSort2[{}, {t_,c___}, m_, {s___}] :=
 selSort2[{c},{},t,{m,s}]
selSort2[{t_,c___}, {s___}, m_, l_] :=
 selSort2[{c},{m,s},t,l] /; t>m
selSort2[{t_,c___}, {s___}, m_, l_] :=
 selSort2[{c},{t,s},m,l]
```

... ma funzionante. *Provare per credere!*

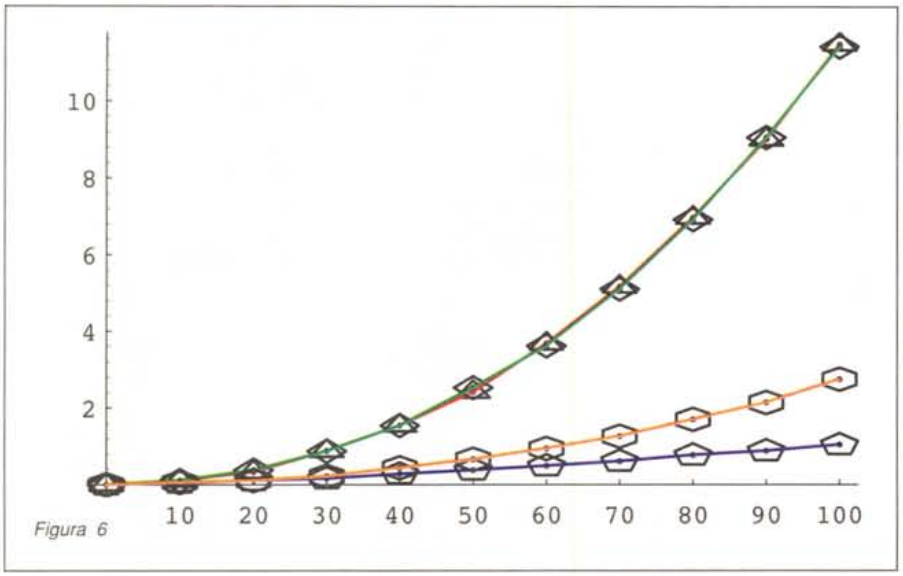


Figura 6

sultato:
In[47]:=
```
valuta[quickSort];
```
```
Ordinate: Si
```
(Vedi figura 6)

Modo più furbo

Un approccio più *vispo* usa una sola funzione:
In[48]:=
```
minMag[x_, {}, m_, p_] :={m,p}
minMag[x_, {t_,c___}, {m___},
p_ ] :=
      minMag[x,{c},{m,t},p]
/; t<=x
minMag[x_, {t_,c___}, m_,
{p___}] :=
 minMag[x,{c},m,{p,t}]
```
per dividere la lista originaria, mentre la funzione principale
In[49]:=
```
quick[l:({}|{_})] := l;      (*3*)
quick[{h_,t___}]:=
  quickHelp[h,minMag[h,{t},{},{}]]
```
si serve di una di appoggio per le chiamate ricorsive:
In[50]:=
```
quickHelp[x_,{m_,p_}]:=
  Flatten[{quick[m],x,quick[p]}]
```
La `(*3*)` rivela una nuova diavoleria: il *pattern matching con alternative* che assegna uno stesso nome a pattern diversi; si riconosce simultaneamente sia il caso della lista vuota che quello in cui c'è un solo elemento.
Le prestazioni, le migliori viste finora, mettono però in evidenza la grande pecca del quicksort:
In[51]:=
```
valuta[quick];
```
```
Ordinate: Si
```
(Vedi figura 7)
accanto ai casi ottimi (la terza lista è stata costruita apposta) e medi (come la quarta che sembrava difficile per gli algoritmi

Il quicksort

Il quicksort è sicuramente l'algoritmo di ordinamento più usato in assoluto; utilizza una tecnica *divide et impera* effettuando le chiamate ricorsive sulle sottoliste ottenute separando i valori che risultano maggiori dell'elemento di riferimento scelto dalla lista (il *pivot*) da quelli che invece sono minori o uguali.

Modo "naïf"

Un approccio semplice usa due funzioni che, con la solita tecnica del travaso, selezionano gli elementi minori (o uguali) dell'elemento di riferimento `x`:
In[44]:=
```
minEq[x_, {}, l_] = l;
minEq[x_, {t_, c___}, {s___}] :=
 minEq[x,{c},{s,t}] /; t<=x
minEq[x_, {_, c___},l_] :=
 minEq[x,{c},l]                (*2*)
```
e quelli maggiori:
In[45]:=
```
mag[x_, {}, l_] := l;
mag[x_, {t_, c___}, {s___}] :=
   mag[x,{c},{s,t}] /; t>x
mag[x_, {_, c___}, l_    ] :=
    mag[x,{c},l] (*2*)
```
Si noti che le definizioni `(*2*)`, praticamente il ramo "else" delle definizioni che le precedono, non si degnano nemmeno di dare un nome all'elemento che viene "scartato".
La funzione principale fornisce il *macchinario ricorsivo*:
In[46]:=
```
quickSort[{}]    = {};
quickSort[{x_}] := {x};
quickSort[{x_, y__}] :=
 Flatten[{quickSort[minEq[x,
{y}, {}]], x,
  quickSort[ mag[x, {y},{}]]}]
```
effettuando le chiamate di ordinamento sulle sottoliste e assemblando i risultati. Si ottiene in questo modo un discreto ri-

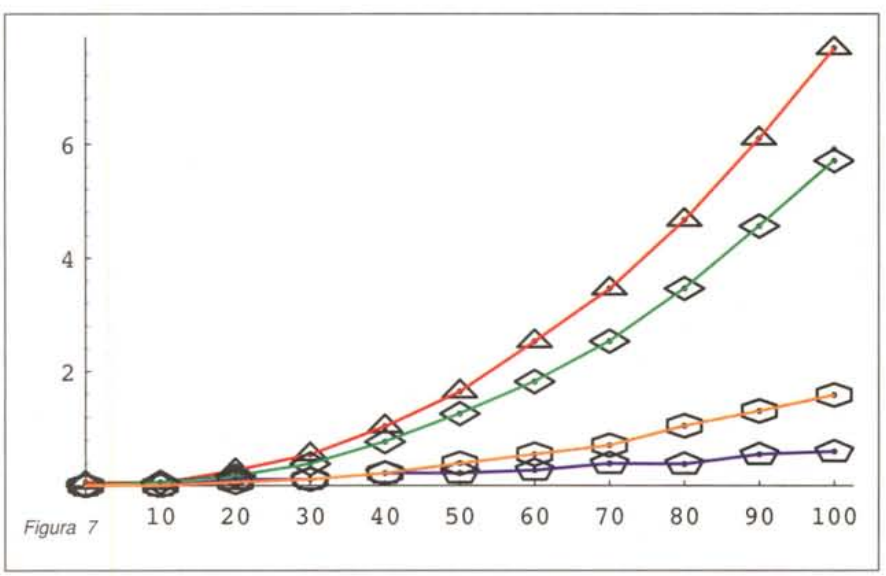


Figura 7

precedenti) in cui l'algoritmo si comporta molto bene ci sono i casi particolarmente *sfigati* in cui la complessità è quadratica. Il fatto che il caso pessimo sia rappresentato proprio dalle liste test 1 e 2 è dovuto all'uso il primo elemento come pivot (infatti una delle sottoliste è sempre vuota). Ma non ci si illuda, comunque si sceglie il pivot, è possibile modificare il mixer per ottenere il corrispondente frullato indigesto!

Il mergesort

La cucina del mergesort utilizza due ingredienti essenziali: la divisione della lista originaria in due parti di uguale lunghezza e la fusione di due liste ordinate in una.
Lo splitting si scrive simpaticamente con la funzione:
In[52]:=

```
split[{t_, c___}, {s___}, l_] :=
 split[{c},l,{s,t}];
```

che distribuisce gli elementi della prima lista *facendo la conta*, ponendo un elemento della sorgente nella prima destinazione e scambiando poi le destinazioni nella chiamata successiva. Le destinazioni vengono restituite insieme quando la sorgente si è prosciugata:
In[53]:=

```
split[{}, a_, b_] := {a, b};
```

Con la funzione per fondere due liste ordinate,
In[54]:=

```
merge[{}, {s1__}, {s2___}] := {s2, s1};
merge[{s1__}, {}, {s2___}] := {s2, s1};
merge[{t_, c___}, l:{x_, ___}, {s___}]:=
 merge[{c}, l, {s, t}]          /; t<=x   (*4*)
merge[l_, {t_, c___}, {s___}]:=
 merge[l, {c}, {s, t}]
```

facciamo la conoscenza di un'altra sottigliezza del pattern matching: in `(*4*)` si dà un nome sia a una struttura che a una sua parte, utilizzando poi la parte nel confronto con `t`, e il tutto nella chiamata ricorsiva.
La funzione **mergeSort**, infine, effettua le opportune chiamate ricorsive
In[55]:=

```
mergeSort[l:{(x_)...}] := l;  (*5*)
mergeSort[l_List] :=
  msHelp[split[l,{},{}]]
```

con l'ausilio della funzione di appoggio **msHelp**
In[56]:=

```
msHelp[{l1_, l2_}]:=
 merge[mergeSort[l1],mergeSort[l2],{}]
```

In `(*5*)` incontriamo l'ultima diavoleria: il *pattern ripetuto* (zero o più volte denotato con tre punti "`...`", in stretta analogia con la notazione per le sequenze). In questo caso si riconosce, e si assegna il nome `l`, ad una lista composta da zero o più elementi tutti uguali tra loro. Chiaramente, una lista siffatta è ordinata, come la `(*5*)` tiene a precisare.
Per finire, le prestazioni del **mergeSort**
In[57]:=

```
valuta[mergeSort];
Ordinate: Si
```

(Vedi figura 8)

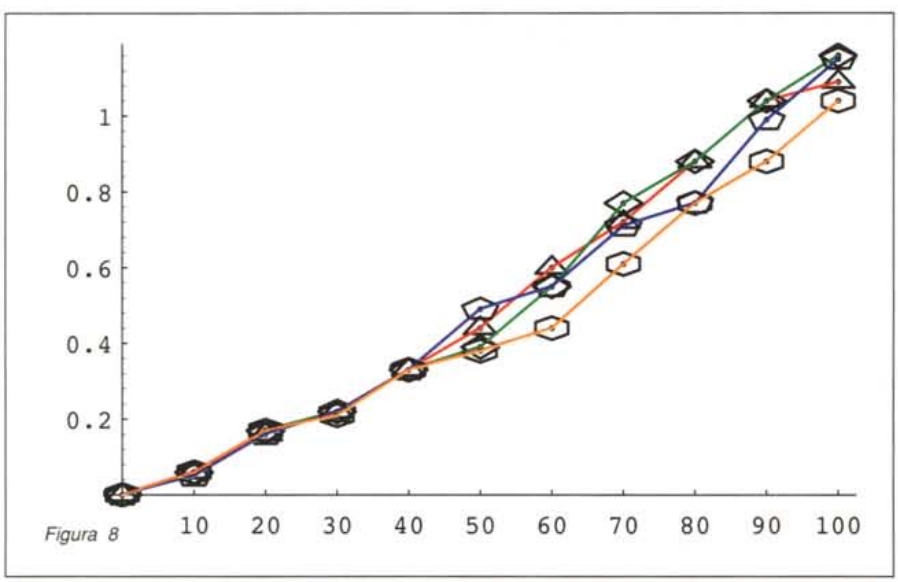


Figura 8

rivelano che è un metodo poco sensibile alla distribuzione dei dati ed efficiente quanto il **quickSort** nel caso medio. Non ci sono i disastrosi casi pessimi del **quickSort** ma nemmeno i suoi favolosi casi ottimi; d'altronde, si sa, chi non risica non rosica!

Conclusioni

Chi è sopravvissuto al viaggio, spero che abbia notato come la grande assente nelle funzioni presentate è proprio la programmazione tradizionale con le sue variabili, strutture di controllo e via dicendo. Questa è stata ridotta ad un sistema di riscrittura, dove le strutture riconosciute sono state opportunamente manipolare. Eppure la potenza di calcolo è rimasta intatta; abbiamo riscritto gli algoritmi di ordinamento più noti con "programmi" brevissimi utilizzando il solo pattern matching e il confronto tra due elementi,
Beh, a dire il vero qualche funzione predefinita è stata usata... Ma si potevano tutte ottenere con il pattern matching, come **Flatten**:
In[58]:=

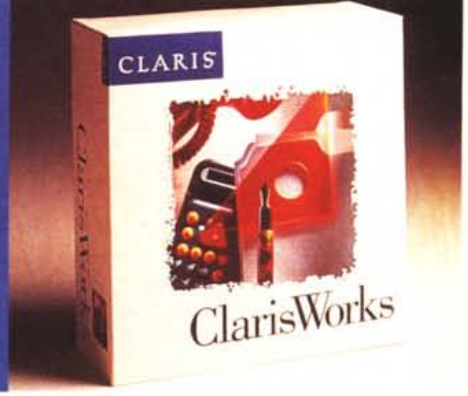
```
myFlat[l_List]:= flat[l,{}]
myFlat[{},l_] := l
myFlat[{{s1___}, s2___}, l_] :=
 myFlat[{s1,s2},l]
myFlat[{x_, s1___}, {s2___}] :=
 myFlat[{s1},{s2,x}]
```

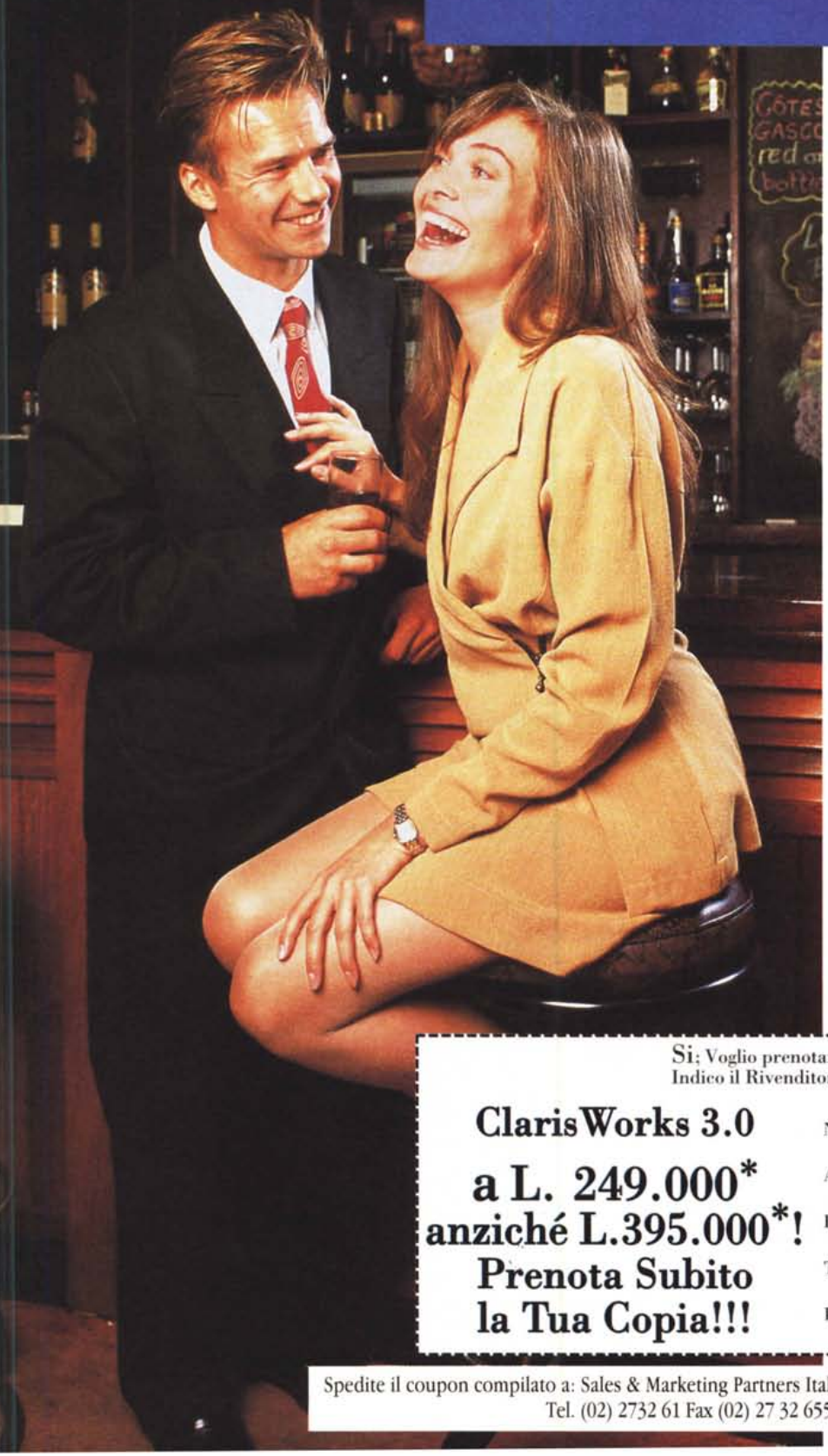
o **Range**, **Reverse** e **Table** che vi lascio come esercizi istruttivi... Arrivederci.

MC

Giuseppe Fiorentino è raggiungibile tramite Internet all'indirizzo fiorent@di.unipi.it

Io & ClarisWorks





"Io amo molto il mio lavoro - ma anche il mio tempo libero è importante. Ecco perché utilizzo ClarisWorks. Si apprende facilmente e mi permette di eseguire i miei lavori velocemente, lasciandomi più tempo per me stessa."

E voi, cosa state aspettando?

Scoprite anche voi ciò che più di due milioni di utenti già sanno: ClarisWorks è il solo software di cui avete bisogno per creare lettere, resoconti, presentazioni, depliant informativi, elenchi, newsletter ed altro ancora.

Grazie alla sua flessibilità, ClarisWorks permette di realizzare testi, progetti grafici e diagrammi nel modo in cui preferite, facendovi risparmiare tempo, denaro ed anche memoria, essendo molto compatto, occupando poco spazio nel vostro sistema.

Provate ClarisWorks 3.0, l'ultima versione di ClarisWorks. Caratterizzata dalla nuova "Assistant Technology", consente di ottenere risultati davvero professionali. Le nuove sorprendenti caratteristiche che distinguono questa versione, mettono a vostra disposizione vere e proprie guide per creare documenti con estrema facilità.

Perché non usare ClarisWorks 3.0? Si tratta pur sempre del software integrato per business, education e home use più venduto al mondo.

Pensateci...

CLARIS

Simply powerful software."

Sì; Voglio prenotare subito la mia copia di ClarisWorks 3.0 Windows.
Indico il Rivenditore di Fiducia presso il quale acquisterò il prodotto.

ClarisWorks 3.0
a L. 249.000*
anziché L.395.000*!
Prenota Subito
la Tua Copia!!!

Nome..... Cognome .....

Azienda .....

Indirizzo.....

Tel..... Fax .....

Rivenditore di fiducia .....

Spedite il coupon compilato a: Sales & Marketing Partners Italy Srl - Via Milano 150 - 20093 Cologno Monzese (MI)
Tel. (02) 2732 61 Fax (02) 27 32 6553 - (*) Prezzi I.V.A. Esclusa.