1.000.000!

Stavolta trattiamo un problema apparentemente banale, ma, sotto sotto, estremamente complicato, già trattato anche da Corrado Giustozzi nella rubrica Intelligiochi (MCmicrocomputer n. 104 e n.107). Si deve calcolare la somma delle cifre della rappresentazione decimale di n! per n grande (1.000, 10.000, 100.000, 1.000.000, ...). Anche se il risultato del calcolo non ha nessuna utilità, né pratica né teorica, la sua risoluzione mette a dura prova sia le capacità di programmazione che le risorse di calcolo disponibili. La strada che ho seguito per vincere la sfida è stata di usare un linguaggio di programmazione a basso livello (il Pascal) utilizzando Mathematica per i calcoli accessori e gli studi di fattibilità

di Francesco Romani

Introduzione

Il fattoriale di un numero intero n si indica con n! e consiste nel prodotto dei numeri naturali da 1 fino ad n; per definizione si pone 0!=1. Il programma per il calcolo del fattoriale (sia nella forma ricorsiva che in quella iterativa) è uno dei più semplici ma la funzione fattoriale cresce molto rapidamente e presto nei comuni linguaggi di programmazione si ottiene un triste *overflow* e il calcolo termina.

Mathematica contiene al suo interno un (efficiente?) programma C per il calcolo del fattoriale con un numero illimitato di cifre.

In[1]:=

```
Factorial[100]
```

Out[1]=

```
933262154439441526816992388562667004\
  907159682643816214685929638952175 9\
  999322991560894146397615651828625 3\
  697920827223758251185210916864000 0\
  000000000000000000
```

Ma quanto è lungo 1.000.000! e quanto costa calcolarlo? conviene aprire una prima parentesi.

Una limitazione superore banale al valore di n! è chiaramente n^n. Con questa stima si vede che

$1.000.000! < 1.000.000^{1.000.000} = 10^{6.000.000}$.

Una formula più accurata per n! è l'approssimazione di Stirling

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$ + termini di ordine inferiore.

Verifichiamo la bontà della approssimazione per n=10.000

In[2]:=

```
Timing[f=Factorial[10000];][[1]]
N[f]
```

Out[2]=

```
6.66 Second
4.0238726 10^2567
```

In[3]:=

```
facS[n_]:=N[Sqrt[2 Pi n] (n/E)^n]
facS[10000]
```

Out[3]=

```
4.0235372 10^2567
```

Se vogliamo quindi stimare rapidamente il numero di cifre decimali di n! basta fare il logaritmo in base 10 della formula di Strirling

In[4]:=

```
lfac[n_]:=N[Log[2 n Pi]/2 +
     n (Log[n]-1)]/Log[10.0]
lfac[10000000]
```

Out[4]=

```
5565708
```

Il numero che vorremmo calcolare ha più di cinque milioni di cifre decimali! Poiché numeri così grandi sono impossibili da stampare definiamo il problema in modo da ridurre drasticamente la lunghezza dell'output

Sfida: calcolare in un tempo accettabile (una notte) la somma delle cifre di 1.000.000!

Proviamo prima una implementazione brutale con *Mathematica*: usiamo la funzione **IntegerDigits** per spaccare il fattoriale e sommiamo il tutto. (In questa e in tutte le prove che seguono, i tempi sono stati misurati su un Macintosh PowerPc a 80 MHz)

In[4]:=

```
IntegerDigits[Factorial[100]]
```

Out[4]=

```
{9, 3, 3, 2, 6, 2, 1, 5, 4, 4, 3, 9,
  4, 4, 1, 5, 2, 6, 8, 1, 6, 9, 9,
  2, 3, 8, 8, 5, 6, 2, 6, 6, 7, 0,
  0, 4, 9, 0, 7, 1, 5, 9, 6, 8, 2,
  6, 4, 3, 8, 1, 6, 2, 1, 4, 6, 8,
  5, 9, 2, 9, 6, 3, 8, 9, 5, 2, 1,
  7, 5, 9, 9, 9, 9, 3, 2, 2, 9, 9,
  1, 5, 6, 0, 8, 9, 4, 1, 4, 6, 3,
  9, 7, 6, 1, 5, 6, 5, 1, 8, 2, 8,
  6, 2, 5, 3, 6, 9, 7, 9, 2, 0, 8,
```

```
    2, 7, 2, 2, 3, 7, 5, 8, 2, 5, 1,
    1, 8, 5, 2, 1, 0, 9, 1, 6, 8, 6,
    4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0}
In[5]:=
  Plus@@%
Out[5]=
  648
```

Proviamo a fare questo per numeri più grandi, fino a 10.000.

```
In[6]:=
  Do[Print[i," ",Timing[
      Plus@@IntegerDigits[Factorial[i]]]],
    {i,1000,10000,1000}]
Out[6]=
   1000 {  1.13 Second,  10539}
   2000 {  5.88 Second,  23382}
   3000 { 16.88 Second,  37602}
   4000 { 31.78 Second,  52830}
   5000 { 51.7  Second,  67698}
   6000 { 76.21 Second,  83619}
   7000 {105.06 Second,  99171}
   8000 {139.83 Second, 115974}
   9000 {178.75 Second, 132777}
  10000 {226.33 Second, 149346}
```

La faccenda promette male, è curioso il fatto che il tempo per la trasformazione di un numero a molte cifre in una lista richiede molto più tempo del calcolo del numero stesso (dei 226 secondi impiegati per il caso 10.000 solo 7 sono serviti per calcolare il fattoriale).

Il costo dell'algoritmo banale

Studiamo ora il problema di quanto costa il calcolo di n! con l'algoritmo più semplice: si fa dapprima il prodotto 2*3 poi moltiplicando il risultato per 4 e così via fino ad n. Si vede facilmente che la somma di due numeri di lunghezza h costa in modo proporzionale ad h mentre il prodotto di un numero di h cifre per un numero di k cifre costa k somme di h cifre (o viceversa) per un ordine di h k operazioni.

Nel nostro caso dobbiamo fare n moltiplicazioni la più difficile delle quali è n* ((n-1)!). Il numero n ha circa $\log_{10}$ n cifre mentre (n-1)! ha circa n log n cifre. Una limitazione superiore al costo totale è quindi dell'ordine di $n^2 \log^2 n$.

Si potrebbe obiettare che contare n volte la moltiplicazione più grossa è un po' eccessivo. Un calcolo più raffinato si può fare approssimando la sommatoria

$$\sum_{i=1}^{n} \log i \log((i-1)!)$$

con l'integrale

$$\int_1^n \log x \,(x \log x)\, dx$$

Tanto per avere un'idea di come cresce la stima vediamo anche quanto vale per 1.000.000.

```
In[1]:=
  Expand[
   Integrate[Log[x] x Log[x],{x,1,n+1}]]
Out[1]=
     1    n²     n² Log[n]    n² Log[n]²
   - - + - - ------------ + -------------
     4    4         2             2
In[2]:=
  Floor[N[%/.n->1000000]]
Out[2]=
  88776410709628
```

La stima migliora di un fattore 1/2 ma l'ordine resta $n^2 \log^2 n$. I lettori più esperti possono ottenere stime piu precise utilizzando la formula di Stirling e trattando con più cura i limiti dell'intervallo di integrazione.

Mettiamo alla prova *Mathematica* con il semplice calcolo del fattoriale senza la stampa del risultato o la somma delle cifre. Il calcolo è stato fatto reinizializzando il Kernel prima di ogni elaborazione per evitare che la memorizzazione di risultati intermedi falsasse le stime dei tempi.

```
  10000     6.66  Second
  20000    14.61  Second
  50000    59.41  Second
 100000   297.53  Second
 200000  1313.91  Second
```

Facciamo un fitting con la complessità stimata $n^2 \log^2 n$ ed estrapoliamo per n=100.000

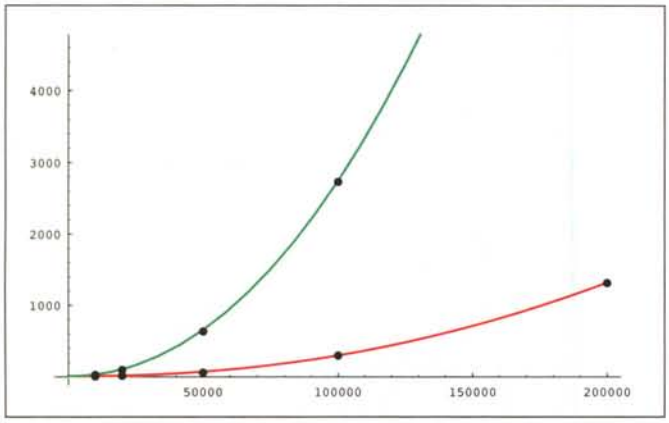
```
In[3]:=
  mm=Transpose[
   {{10000,20000,50000,100000,200000},
    {6.66, 14.61, 59.41, 297.53, 1313.91}}];
  fm=Fit[mm,{n^2 Log[n]^2},n]
  N[fm/.n->1000000] Second
  % /3600 Second
Out[3]=
  2.64 10^-10 n² Log[n]²
  42110.79 Second
  11.69 Hour
```

Lo stesso calcolo è stato effettuato con un programma che implementa in Pascal l'aritmetica tradizionale (quella studiata alle elementari) su numeri di lunghezza qualsiasi. In questo caso si può calcolare anche la somma delle cifre con costo trascurabile (in Pascal c'è l'accesso diretto ai dati e non vengono create complicate strutture dati). I tempi sono stati i seguenti:

```
  10000    26.55  Second
  20000    96.48  Second
  50000   632.90  Second
 100000  2729.88  Second
```

In questo caso, finché n è rappresentabile in un longint la complessità è (solo) n^2 log n. Facciamo un altro fitting e plottiamo il tutto *(vedi figura 1)*

```
In[4]:=
  pp=Transpose[
```




```
  {{10000,20000,50000,100000},
   {26.55, 96.48, 632.90, 2729.88}}];
fp=Fit[pp,{n^2 Log[n]},n]
N[fp/.n->1000000] Second
% /3600 Hour
```

Out[4]=

```
2.36 10^-8 n^2 Log[n]
327373.5 Second
90.93 Hour
```

In[5]:=

```
Needs["Graphics`Colors`"];
ListPlot[pp,PlotStyle->{PointSize[0.015]}];
ListPlot[mm,PlotStyle->{PointSize[0.015]}];
Plot[{fm,fp},{n,1,200000},
  PlotStyle->{Red,Green}];
Show[%,%%,%%%];
```

L'accordo è perfetto. Ci troviamo quindi nella condizione di calcolare 1.000.000! in 12 ore con *Mathematica* senza poterci fare nulla o di calcolarne la somma delle cifre in Pascal in 90 ore; cerchiamo una strada alternativa che permetta di fare le cose presto e bene.

Secondo approccio: divide et impera

Un principio fondamentale dell'informatica dice che vale molto di più un algoritmo più furbo che un calcolatore più potente. Un algoritmo più efficiente di moltiplicazione può essere derivato con la tecnica del *divide et impera*. Si assuma di lavorare in base d e che h sia una potenza di 2; due numeri a e b di h cifre possono essere scritti come

$a = a_1\, d^{h/2} + a_2, \quad b = b_1\, d^{h/2} + b_2;$

il prodotto a b può essere scritto nel seguente modo:

$$a\,b = (a_1\, d^{h/2} + a_2)(b_1\, d^{h/2} + b_2) =$$

$$a_1\, b_1\, d^{h} + (a_1 b_2 + a_2 b_1)\, d^{h/2} + a_2\, b_2.$$

Poiché vale l'identità

$$(a_1 b_2 + b_1 a_2) = (a_1 + a_2)(b_1 + b_2) - a_1\, b_1 - a_2\, b_2$$

Il prodotto a b può essere calcolato nel modo seguente:

a) calcola $a_1\, b_1$; (costo 1 moltiplicazione di h/2 cifre);

b) calcola $a_2\, b_2$; (costo 1 moltiplicazione di h/2 cifre);

c) calcola $(a_1 + a_2)(b_1 + b_2)$; (costo 1 moltiplicazione di n/2 cifre e due somme);

d) calcola $(a_1 b_2 + b_1 a_2)$ (costo 2 sottrazioni);

e) forma il risultato (costo due spostamenti e due somme).

In totale sono richieste tre moltiplicazioni con operandi di h/2 cifre più somme e spostamenti il cui costo è lineare in h. Si ottiene quindi per la complessità M(h) della moltiplicazione di operandi di h bit la relazione :

$$M(n) \le 3\, M(h/2) + \text{costante} * h$$

che si risolve ottenendo una limitazione superiore dell'ordine di $h^{\log 3/\log 2} = h^{1.59\cdots}$

Purtroppo questo algoritmo si applica bene solo se i due operandi sono della stessa lunghezza, per poterlo utilizzare per il fattoriale bisogna riorganizzare completamente il calcolo.

L'idea base è di trasformare il calcolo del fattoriale in una moltiplicazione di molti operandi, tutti più o meno della stessa lunghezza e della successiva applicazione dell'algoritmo di moltiplicazione veloce per fare i prodotti a due a due dei risultati parziali.

Se moltiplichiamo a due a due k numeri di h cifre, otteniamo **`Ceiling[k/2]`** numeri di 2h cifre, insistendo si raddoppia sempre il numero delle cifre mentre i moltiplicandi diminuiscono.

Non è però consigliabile usare come operandi di partenza i numeri 1,2,..., n. Vediamo allora come raggruppare i fattori n! nel prodotto di k numeri di h cifre abbastanza equilibrati tra loro.

Generazione degli operandi

Si decide di lavorare Pascal, simulando la base 10^7 e rappresentando le cifre con numeri di tipo `real` di 4 byte (ci stanno senza errori numerici). Per fare le 4 operazioni si trasformano i `real` in `double` di 8 byte e non si commettono errori: l'aritmetica reale del computer sostituisce le tabelline in base 10.000.000.

Per suddividere gli n fattori nel minor numero possibile di numeri di 32 cifre si moltiplicano tra loro dapprima i numeri grandi a partire da n e poi quelli piccoli (a partire da 2) fino a che non si è raggiunto il limite di 10^{224} (ovvero 32 cifre in base 10.000.000), il processo continua fino a che non ci sono più fattori da moltiplicare. Vediamo con *Mathematica* quello che succede per 1.000!

In[1]:=

```
BASE=10^7;
LIMITE=BASE^32;
N[%]
```

Out[1]=

```
1. 10^224
```

In[2]:=

```
accumula[j_]:=Module[{x,i1=j,i2=1},
  While[i1>=i2,
    x=1;
    While[(x*i2<LIMITE)&&(i1>=i2),
        x=x*i2;i2=i2+1];
    While[(x*i1<LIMITE)&&(i1>=i2),
```

```
        x=x*i1;i1=i1-1];
      Print[{i1,i2,N[x]}]]];
  accumula[1000];
```

Out[2]=

```
{925, 2,  5.80413 10^223}
{850, 7,  9.51963 10^223}
{774, 10, 6.87018 10^223}
{696, 10, 3.78063 10^223}
{617, 11, 3.68957 10^223}
{536, 11, 4.22977 10^223}
{453, 11, 4.07306 10^223}
{368, 12, 1.41892 10^223}
{279, 12, 2.0716  10^223}
{185, 13, 1.74363 10^223}
{ 79, 13, 4.60813 10^223}
{ 12, 13, 1.86767 10^223}
```

Si vede come si ottiene un buon equilibrio dei risultati. Nel caso 10.000 si ottengono 161 fattori molto vicino alla limitazione inferiore teorica

numero di cifre del fattoriale / numero di cifre per elemento.

In[3]:=

```
lfac[10000]/Log[10,LIMITE]
```

Out[3]=

```
159.19
```

Per la cronaca, nel programma Pascal si è introdotto un ulteriore accorgimento, si riducono i fattori divisibili per 10 eliminando alcuni degli zeri che formano la coda di n! e che non contribuiscono alla somma delle cifre. Nel caso 1.000.000 ci sono 111.111 zeri che vengono in tal modo eliminati.

Riduzione degli operandi e calcolo della complessità

Definiamo dapprima la funzione che trasforma **{k,h}** in **{Ceiling[k/2],2h}**.

In[1]:=

```
Tr[{a_,b_}]:={Ceiling[a/2],2b}
Tr[{1,b_}]  :={1,b}
```

Quindi la applichiamo fino a che non resta un solo numero e vediamo cosa succede per 2.000 operandi di 32 cifre.

In[2]:=

```
Drop[FixedPointList[Tr,{2000,32}],-1]
```

Out[2]=

```
{{2000,32},{1000,64},{500,128},{250,256},
 {125,512},{63,1024},{32,2048},{16,4096},
 {8,8192},{4,16384},{2,32768},{1,65536}}
```

La prima volta bi sogna fare 1.000 moltiplicazioni di 32 cifre, la seconda 500 di 64 cifre... infine una di 32.768 cifre. Per determinare il costo si considera il fatto che una moltiplicazione di h cifre si può fare con tre moltiplicazioni di h/2 cifre e un ulteriore costo proporzionale ad h.

In[3]:=

```
m[h_]:=3 m[h/2] + a[h/2];
a[h_]:=2 a[h/2];
a[1]  = aa;
m[1]  = mm;
```

Le costanti **aa** e **mm** dipendono da come è scritto il programma e dalla velocità della macchina, **mm** può essere interpretato come il tempo in secondi per eseguire una moltiplicazione elementare. Adesso se **start[n]** è il numero di operandi di 32 cifre a cui si riduce il fattoriale di n, la complessità in termini di **aa** e **mm** si può calcolare come segue.

In[4]:=

```
cmp[{a_,b_}]:=Floor[a/2] m[b]
story[n_]:=Drop[FixedPointList[Tr,
           {start[n],32}],-1]
cx[x_]:=Expand[Plus@@Map[cmp,story[x]]]
```

Per poter stimare davvero i tempi del nostro programma Pascal bisogna dare in input il valore di **start[n]** (facilmente calcolabili sia in *Mathematica* che in Pascal)

In[5]:=

```
start[100000]=2000;
start[200000]=4272;
start[500000]=11581;
start[600000]=14115;
start[700000]=16676;
start[800000]=19274;
start[1000000]=24524;
start[1300000]=32259;
```

I tempi effettivi di esecuzione del programma Pascal sulla macchina in uso sono stati i seguenti

.In[6]:=

```
time[100000]=616;
time[200000]=3807;
time[500000]=12931;
time[600000]=15402;
time[700000]=34321;
time[1000000]=39308;
time[1300000]=50293;
```

e sono serviti circa 25 megabyte di memoria RAM (fisica, non virtuale).

Risolvendo un sistema lineare si ricavano le stime di **aa** e **mm** e quindi i tempi in secondi stimati per il calcolo di n!

In[7]:=

```
solve[x_,y_]:=
  Solve[{cx[x]==time[x],cx[y]==time[y]}]
rul=N[solve[200000,500000][[1]]]
```

In[8]:=

```
{cx[700000]/.N[rul],
 cx[1000000]/.N[rul],
 cx[1300000]/.N[rul]}
```

Out[2]=

```
{32960.5, 38667.5, 49715.1}
```

Verifica dei risultati e conclusioni

I tempi veri per 700.000!, 1.000.000! e 1.300.000! sono stati rispettivamente 34321, 39308 e 50293 secondi, con un errore abbastanza piccolo rispetto alla stima. La moltiplicazione più grossa effettuata in questi tre casi è stata tra due numeri di 1.048.576 ($=2^{20}$) cifre in base 10^7; per andare oltre servirebbe una moltiplicazione tra numeri di 2^{21} cifre e tanta memoria in più. I valori calcolati per la somma delle cifre di n! sono stati i seguenti:

In[1]:=

```
ris[1000]=10539;
ris[2000]=23382;
```

```
ris[5000]=67698;
ris[10000]=149346;
ris[50000]=903555;
ris[100000]=1938780;
ris[200000]=4154076;
ris[500000]=11286711;
ris[600000]=13761612;
ris[700000]=16250679;
ris[1000000]=23903442;
ris[1000000]=31772529;
```

I lettori più maliziosi si chiederanno come si fa ad essere sicuri di questi risultati, se il programma Pascal ha un errore sistematico potremmo avere ottenuto dei numeri del lotto dopo ore ed ore di laboriosi calcoli. Fino a 10.000 si può fare il confronto diretto con i calcoli di *Mathematica*, e dopo?
Per fortuna è possibile ottenere una buona approssimazione statistica del risultato in tempi rapidissimi. Un fattoriale possiede una coda di zeri dovuti ai numerosi fattori 2 e 5. La lunghezza della coda di zeri è pari al numero di fattori 5 che a sua volta è pari a n/5 + n/25 + n/125+ ... Supponendo n abbastanza grande si può considerare la serie infinita (i termini che si aggiungono pesano molto poco) e stimarla con n/4
In[2]:=

```
Needs["Algebra`SymbolicSum`"]
SymbolicSum[n/5^k,{k,1,Infinity}]
```

Out[2]=

$$\frac{n}{4}$$

Se le cifre di testa di n! possono essere ritenute equidistribuite la loro somma vale circa 4.5 volte il loro numero ovvero
In[3]:=

```
stima[n_]:=Floor[4.5(lfac[n]-n/4)]
```

Per finire possiamo fare una tabella che riporta n, il numero delle cifre di n!, la loro somma vera, quella stimata e il tempo in secondi richiesto per effettuare il calcolo.
In[4]:=

```
Map[{#,Floor[lfac[#]],ris[#],stima[#],
     Time[#]}&,
    {100000,200000,500000,600000,
     700000,1000000,1300000}]//TableForm
```

Out[4]=

100000	456573	1938780	1942080	616
200000	973350	4154076	4155075	3807
500000	2632341	11286711	11283034	12931
600000	3206317	13761612	13753428	15402
700000	3787565	16250679	16256546	34321
1000000	5565708	23903442	23920690	39308
1300000	7383546	31772529	31763461	50293

Mi congedo dai lettori sfidando qualche abile programmatore a fare di meglio, magari in C su di un Pentium a 90 MHz.

MC

Francesco Romani è raggiungibile tramite Internet all'indirizzo romani@di.unipi.it