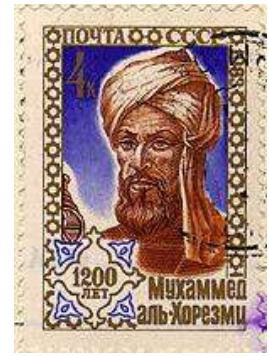


L'IMPORTANZA DI SCEGLIERE IL GIUSTO ALGORITMO

© 2005, Roberto A. Foglietta <me@roberto.foglietta.name>



Il termine Algoritmo deriva dal nome del matematico persiano *Abu Ja'far Mohammed ibn Mâsâ al-Khowârizmî* (825 d.C.). La sua opera venne tradotta in latino da un monaco europeo, con il titolo *Liber algarismi* (Il libro di al-Khwarizmi) e diede il nome all'algebra.

“Algoritmo” è una parola abbastanza comune ma questo non significa che la sua importanza sia universalmente nota. Anzi alcuni ne sottovalutano l'utilità per aumentare considerevolmente le prestazioni di un sistema informatico più di quanto non possa fare la migliore delle ottimizzazioni sul codice.

Supponiamo che il nostro problema sia quello di trovare il massimo comune divisore di due numeri interi positivi, siano N e M. Un modo per giungere al risultato è usare la forza bruta:

```
$ tail -n11 algo1.c
```

```
static
long trova_mcd(long M, long N) {

    long mcd = 1, i;
    if(M > 0 && N > 0)
        for(i = 1; i < max(N,M); i++)
            if(M == (M/i)*i
               && N == (N/i)*i)
                mcd = max(mcd,i);

    return mcd;
}
```

Listato n.1

Per il momento preferisco ignorare completamente tutte le problematiche che i *benchmark* comportano e limitarmi ad una stima grossolana e poi in seconda battuta rifletteremo sulla correttezza di questo approccio. Anche nelle tabelle successive i tempi sono riferiti ad un milione di interazioni.

Presentazione del sistema utilizzato:

```
$ uname -srpm
Linux 2.6.11-8mdksmp i686 Intel(R) Pentium(R) 4 CPU 2.80GHz
```

Per poter utilizzare il comando `time` ho sviluppato una parte comune che si interfaccia tramite la linea di comando con l'utente e dentro la quale i vari pezzi di codice vengono chiamati come funzione. L'aggiungo di seguito in modo da consentire al lettore di

ALGORITMO “FORZA BRUTA” – n.1

compilazione senza ottimizzazione

```
$ gcc -Wall -O0 algo1.c -o algo1
```

stima del tempo impiegato per milione di interazioni

```
$ time ./algo1 12 34 1000000
```

risultati per 12 34, 12 340, 120 1700 e 12 3400

```
1.02user 0.00system 0:01.02elapsed 99%CPU
8.43user 0.00system 0:08.44elapsed 99%CPU
41.49user 0.00system 0:41.52elapsed 99%CPU
82.60user 0.00system 1:22.69elapsed 99%CPU
```

	tempo impiegato	inclinazione retta
12, 34 x1	-> 1 sec	
12, 34 x10	-> 9 sec	(8.43-1.02)/9 = 0.82
120,34 x50	-> 42 sec	(41.49-1.02)/49 = 0.83
12, 34 x100	-> 83 sec	(82.60-1.02)/99 = 0.82

compilazione con ottimizzazione massima

```
$ gcc -Wall -O9 algo1.c -o algo1
```

```
0.65user 0.00system 0:00.65elapsed 99%CPU
5.16user 0.00system 0:05.17elapsed 99%CPU
26.07user 0.00system 0:26.07elapsed 100%CPU
50.11user 0.02system 0:50.13elapsed 100%CPU
```

fattore di ottimizzazione: 50-83/83 = -40% tempo
83-50/50 = +66% prestazioni

Tabella n.1

confrontare i suoi risultati con quelli pubblicati in questo articolo:

```
$ head -n36 algo1.c

#include <stdio.h>
#include <stdlib.h>

static long trova_mcd(long M, long N);

#define max(a,b) (((a)>(b))? (a):(b))
#define abs(a) ((a<0)?-a:a)

enum {
    CMDNAME=0,
    PRINUM,
    SECNUM,
    TIMES,
    N_PARAM
};

int main(int argc, char **argv) {
    long i, n[N_PARAM], mcd = 1;

    //Controlla se il numero dei parametri e' corretto
    if(argc != N_PARAM) {
        fprintf(stderr, "USAGE: algo1 <N> <M> <times>");
        return 1;
    }

    //Trasforma i parametri in interi positivi
    for(i = PRINUM; i < N_PARAM; i++)
        n[i] = abs(atol(argv[i]));

    for(i = 0; i < n[TIMES]; i++)
        mcd = trova_mcd(n[PRINUM], n[SECNUM]);

    fprintf(stdout, "mcd = %ld\n", mcd);
    return 0;
}
```

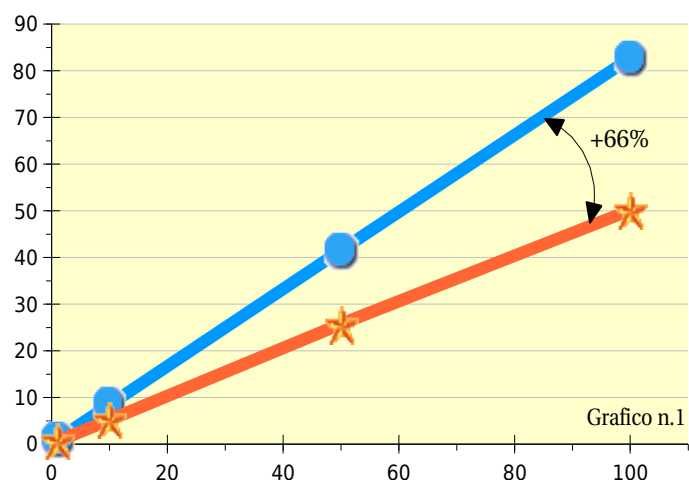
Listato n.2

Inizialmente compilo questo codice senza alcuna ottimizzazione, senza informazioni di *debug* e *profiling* ma con tutti i *warning* attivi per accertarmi della correttezza almeno a livello sintattico. Dalla tabella n.1 si evince che è richiesto un tempo che cresce linearmente con il massimo dei due numeri (cfr. 1^a parte di tabella n.2).

Successivamente ripeto le misure con una versione dell'eseguibile ottenuta compilando con la massima ottimizzazione possibile (cfr. 2^a parte di tabella n.2).

Si noti che, sebbene l'esecuzione sia più veloce del 66%, il carattere dell'algoritmo non è mutato. Questo accade in generale altrimenti ci troveremmo ad usare un compilatore che interpreta il nostro codice invece di limitarsi a tradurlo nel linguaggio macchina più appropriato per l'architettura hardware scelta.

carattere lineare



Insomma se si vuole ottenere qualcosa di considerevolmente diverso occorre metter mano al codice. Provo ad ottimizzare l'implementazione dell'algoritmo "forza bruta"

con le seguenti considerazioni:

- MCD è più piccolo del minore dei due (valore in top)
- $mcd=1$ se uno dei due numeri è minore uguale a 1
che sono cognizioni di algebra e definizione MCD
- il calcolo del valore top al di fuori del ciclo for
- uso dell'operatore resto intero (%) del linguaggio C
che sono cognizioni di programmazione
- imposizione diretta $mcd=i$ perché accade $i \geq mcd$ sempre
che è una cognizione dell'implementazione dell'algoritmo

cioè in questo modo:

```
$ tail -n11 algo2.c

#define min(a,b) (((a)<(b))?(a):(b))

static
long trova_mcd(long M, long N) {
    long mcd = 1, top = min(M,N), i;
    for(i = 1; i < top; i++)
        if(!(M%i || N%i))
            mcd = i;
    return mcd;
}
```

Listato n.3

Confrontando le tabelle 1 e 2 si nota che:

- i tempi sono scesi da 150 a 25 secondi, cioè esattamente di un fattore 6 ma...
- la linearità si è trasferita sul minore dei due numeri quindi il vantaggio sopra indicato c'è solo quando un numero è molto più piccolo dell'altro e...
- il codice scritto "per essere più veloce" è meno ottimizzabile dal compilatore ed anche meno comprensibile.

ALGORITMO "FORZA BRUTA" – n.2

compilazione senza ottimizzazione

```
$ gcc -Wall -O0 algo1.c -o algo1
```

risultati per 12,34 poi 12,340 e 12,3400

```
0.27user 0.00system 0:00.28elapsed 99%CPU
0.27user 0.00system 0:00.27elapsed 99%CPU
0.27user 0.00system 0:00.27elapsed 99%CPU
```

risultati per 1200,34 poi 1200,340 e 1200,3400

```
0.86user 0.00system 0:00.86elapsed 99%CPU
6.42user 0.00system 0:06.43elapsed 100%CPU
20.40user 0.00system 0:20.40elapsed 99%CPU
```

andamento lineare legato al minore dei due numeri

	tempo impiegato		inclinazione retta
12 impiegato	0.28		$0.28/12 = 0.023$
34 impiegato	0.86		$0.86/34 = 0.025$
340 impiegato	6.43		$6.43/340 = 0.019$
1200 impiegato	20.40		$20.40/1200 = 0.017$

compilazione con ottimizzazione massima

```
$ gcc -Wall -O9 algo2.c -o algo2
```

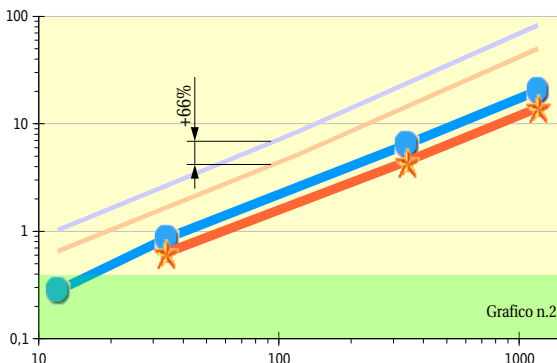
risultati per 1200,34 poi 1200,340 e 1200,3400

```
0.63user 0.00system 0:00.63elapsed 99%CPU
4.47user 0.00system 0:04.47elapsed 100%CPU
14.08user 0.00system 0:14.08elapsed 99%CPU
```

fattore di ottimizzazione: $14-21/21 = -33\%$ tempo
 $21-14/14 = +50\%$ prest.

Tabella n.2

Carattere lineare su bilogaritmica



Nel grafico bilogaritmico una differenza sull'asse verticale corrisponde ad un fattore moltiplicativo e poi due serie di dati che in un grafico cartesiano stanno su rette con diversa inclinazione qui stanno su quasi parallele.

La tabella 2 ci mostra che più sforzi si fanno per ottimizzare il codice prima si arriva al limite concettuale dell'algoritmo sottostante ed in particolare che l'uso della "forza bruta" non paga in termini di "carattere".

Perché preoccuparsi tanto del carattere di un algoritmo? In fondo con l'ottimizzazione del codice e del compilatore si può ottenere fin anche un fattore 10 in prestazioni. Nel caso ad esempio di un *webserver* sarebbe un risultato più che ottimo.

Vi faccio un esempio concreto: le chiavi crittografiche, quelle dei *browser* che supportano il protocollo *https* ma anche le stesse che sono usate anche da *ssh* e *gpg* sono tutte basate sui numeri primi e se una chiave fosse lunga 10 bit le combinazioni sarebbero 1024 se invece fosse lunga 10+6=16 bit allora le combinazioni sarebbero 64 volte di più. Solitamente sono di lunghezza compresa fra 128 e 2048 bit.

Senza scendere nei dettagli si fa presto a capire che con una coppia chiavi di questo genere non si può rompere con un algoritmo lineare sul numero di combinazioni per quanto possa essere veloce il computer su cui gira l'esponenziale $N_{comb} = 2^{nbit}$ cresce più rapidamente. Si tratta di spostare la linearità su una quantità che cresce "di meno".

Un altro caso concreto: il tempo di query su un *database* è un'altra di quelle attività in cui perdere tempo in modo lineare rispetto al numero di voci e/o referenze contenute in esso risulterebbe essere una pesante ipoteca sulla capacità del *database* di crescere.

Rompiamo ogni indugio ed affrontiamo di nuovo lo steso problema con una versione modernizzata del l'algoritmo di Euclide in due implementazioni quasi identiche:

```
$ tail -n18 algo3.c

static
long trova_mcd(long M, long N) {
    long t;
    if(M < N) {
        t = M; M = N; N = t;
    }
    printf("%ld\t%ld", M, N);
    if(N > 1) {
        while((t = M%N) > 0) {
            M = N;
            N = t;
            printf("\t%ld", N);
        }
    }
    printf("\n");
    return max(N,1);
}
```

Listato n.4

La differenza fra le misure indicate nelle due sezioni della tabella n.3 è solo nell'uso di binari generati con o senza la presenza delle tre righe di codice evidenziate nel listato n.4. Queste righe contengono delle stampe a video, una delle quali è contenuta nel ciclo for perciò eseguita ad ogni divisione.

ALGORITMO "EUCLIDE" – n.3	
compilazione senza ottimizzazione	
\$ gcc -Wall -O0 algo3.c -o algo3	
risultati per le coppie 12,34 poi 120,340 e 1223,3445 raccolti con le printf redirette su /dev/null	
1.03	user 0.00system 0:01.03elapsed 99%CPU
1.08	user 0.00system 0:01.09elapsed 99%CPU
2.98	user 0.00system 0:02.99elapsed 99%CPU
serie numeriche stampate per nei casi sopra	
34	12 10 2
340	120 100 20
3445	1223 999 224 103 18 13 5 3 2 1
ALGORITMO "EUCLIDE" – n.4	
senza le printf e con l'ottimizzazione massima	
\$ gcc -Wall -O9 algo4.c -o algo4	
risultati per le coppie casuali \$RANDOM,\$RANDOM	
0.08	user 0.00system 0:00.08elapsed 98%CPU
0.16	user 0.00system 0:00.17elapsed 98%CPU
0.23	user 0.00system 0:00.23elapsed 99%CPU
0.22	user 0.00system 0:00.22elapsed 99%CPU
0.29	user 0.00system 0:00.29elapsed 99%CPU
in particolare per 1200,3400 a ripetizione	
0.07	user 0.00system 0:00.07elapsed 98%CPU
0.06	user 0.00system 0:00.06elapsed 99%CPU
0.07	user 0.00system 0:00.07elapsed 96%CPU

Tabella n.3

Le ultime tre misure della tabella n.3 mostrano che si è guadagnato circa un fattore 200 in termini di prestazioni ma soprattutto le prime misure e quelle fatte con su coppie casuali di numeri mostrano che non vi è linearità fra il tempo di esecuzione e i numeri.

Se osserviamo ancora il grafico n.2 ci accorgiamo che le misure della 2^a parte della tabella n.3 cadono nella striscia in verde. Le fluttuazioni dei tempi di esecuzione non sono riconducibili ad eventi casuali in quanto le prime 3 misure della tabella n.2 e le ultime tre della tabella n.3 mostrano che time produce risultati piuttosto affidabili per i quali la stima dell'errore $< 5\% \pm 1$ digit mentre le fluttuazioni sono 10 volte più grandi.

Analizziamo l'ultima sequenza numerica nella tabella n.3 che viene generata in 10 passi:

1 -	3445	%	1223	=	999		
2 -			1223	%	999	=	224
3 -			999	%	224	=	103
4 -			224	%	103	=	18
5 -			103	%	18	=	13
6 -			18	%	13	=	5
7 -			13	%	5	=	3
8 -			5	%	3	=	2
9 -			3	%	2	=	1
10 -			2	%	1	=	0

Per definizione di MCD la sequenza così costruita è monotona strettamente decrescente¹ e converge a zero in un numero finito di passi². Questo numero è minore o uguale al più piccolo degli interi per cui accade $2^k > N$ infatti basta notare che la serie dei resti $\{n_i\}$ è sovrastata dalla $\{2^{k+1-i}\}$:

$$\begin{array}{l}
 M \% N = n_1 < N < 2^k \\
 N \% n_1 = n_2 < \min(N - n_1, n_1) < \min(\frac{1}{2}N, n_1) < \frac{1}{2}N < 2^{k-1} \\
 n_i \% n_{i+1} = n_{i+2} < 2^{k-i}
 \end{array}$$

Si noti che se $A \ B \ C \ \theta$ sono gli ultimi 4 termini di una sequenza come quella sopra allora $B=nC$ ma anche $A-mB=C \rightarrow A-mnC=C \rightarrow A=(mn+1)C$ quindi per induzione i primi due numeri della sequenza sono multipli dell'ultimo numero non nullo. Tralascio la dimostrazione che questo procedimento conduca realmente al MCD piuttosto che ad un divisore qualsiasi per invece esplorare la dipendenza dal tempo.

tempo	passi	rapporto
1.03	/ 3	= 0.34
1.08	/ 3	= 0.36
2.98	/ 10	= 0.30
tempo	printf	rapporto
1.03	/ (3+2)	= 0.21
1.08	/ (3+2)	= 0.22
2.98	/ (10+2)	= 0.25
tempo	termini	rapporto
1.03	/ 4	= 0.26
1.08	/ 4	= 0.27
2.98	/ 11	= 0.27

Tabella n.4

Da quanto detto sopra si evince che il tempo impiegato è linearmente proporzionale al numero di passi compiuti poiché questi sono comunque pochi, anche a fronte di numeri molto grandi, l'unico modo per rilevare questa dipendenza (tabella n.4) era inserire un'istruzione molto lenta, come ad esempio la printf, nel ciclo interno. Invece le altre due stampe sono state aggiunte per ottenere un output completo dal punto di vista della serie.

Se si conosce bene l'algoritmo fare delle misure potrebbe essere ritenuto superfluo per individuare la relazione ma l'implementazione può nascondere delle piccole sorprese: in questo caso la dipendenza sembra essere finita sul numero di termini stampati piuttosto che sul numero di printf o dei passi.

1 $a \% b = c$ significa che esiste k intero non nullo per cui $kb \leq a < (k+1)b$ quindi $a - kb = c$ dove $c < b$

2 la serie è positiva e strettamente decrescente (cfr. nota n.1) quindi converge a 0