

BIOGRAFIA

Rosario Turco è un ingegnere elettronico, laureato all'Università Federico II di Napoli, che lavora dal 1990 in società del gruppo Telecom Italia.

Le sue competenze professionali sono in ambito delle architetture hw/sw Object Oriented (OOA/OOP, AOP, SOA, Virtualization) e in generale "Java 2 Enterprise Edition". Ha lavorato molti anni nella progettazione e sviluppo di sistemi informatici su piattaforme Windows/ Unix/ Linux e con linguaggi Java, C, C++.

Negli ultimi anni si è particolarmente interessato alla crittografia e alla Teoria dei Numeri.

ANALISI COMPUTAZIONALE

In determinati settori civili e militari è di enorme importanza che gli algoritmi progettati e implementati, siano molto efficienti in termini di tempo di elaborazione e/ o di spazio (di memoria, di memoria di massa etc) che richiedono per l'elaborazione stessa. Normalmente la dimensione considerata è la quantità N di dati di input che il programma deve elaborare. L'analisi computazionale ha lo scopo di determinare la quantità di risorse impiegate (la maggior parte delle volte ci si riferisce al tempo) in funzione di N . L'indagine ricade solitamente sul *caso medio*, in corrispondenza della classe di dati di ingresso tipico e che normalmente si verifica, e sul caso peggiore, corrispondente alla peggiore configurazione di dati di ingresso. L'ideale sarebbe poter misurare tutti i tempi a fronte di ogni possibile configurazione di dati di ingresso. Tuttavia se questo è possibile per semplici programmi, nel caso di programmi complessi la strada non è percorribile.

L'analisi teorica, quindi, è tipicamente basata su vari passi:

- tentativo di legare il tempo di esecuzione entro un "limite superiore" indipendentemente dalle caratteristiche dei dati di ingresso;
- individuazione delle principali operazioni astratte dell'algoritmo, in modo da astrarre la logica dell'algoritmo dalla implementazione e dalla macchina;
- osservazione del comportamento dell'algoritmo, per trovare grandezze che descrivano il comportamento nel *caso medio* e nel *caso peggiore*

E' ovvio, ad esempio, che uno stesso algoritmo può essere eseguito più velocemente o meno velocemente su due PC con caratteristiche diverse; ma a parità di potenza elaborativa quello che fa la differenza è l'algoritmo se è ottimizzato o meno ai fini del tempo di esecuzione. L'analisi computazionale, quindi, si interessa solo dei tempi di esecuzione che dipendono dalle proprietà dell'algoritmo teorico e non della macchina a disposizione o del compilatore che può produrre del linguaggio macchina più o meno ottimizzato. L'individuazione delle operazioni astratte coinvolge di solito un numero ristretto di operazioni dell'algoritmo, in genere quelle di *potenza maggiore (di vasta portata)*, come ad esempio il numero di confronti effettuati da un algoritmo di ordinamento, il numero di cicli in gioco o di cicli annidati durante le elaborazioni, le tecniche di ricerca e di ordinamento usate etc.

In generale raramente vale la pena di fare un'analisi accurata; ma si è già abbondantemente contenti di una *stima delle prestazioni* che possano condurre ad una classificazione del tipo di algoritmo usato.

CLASSIFICAZIONE DEGLI ALGORITMI

Praticamente tutti i programmi conosciuti, anche quelli di questo libro, hanno tempi di esecuzione proporzionale a una delle seguenti funzioni:

1	La maggior parte delle istruzioni dell'algoritmo sono eseguite una sola volta. Il suo tempo di esecuzione è quindi costante e sempre prevedibile
Log N	Il tempo di esecuzione è <i>logaritmico</i> . Il programma rallenta leggermente al crescere di N. L'algoritmo tipico di questo genere è quello che riduce un grosso problema in N problemi più piccoli e in questi casi il tempo di esecuzione è una grossa costante. In generale se $N=10^3$ Log N=3, se $N=10^6$ Log N=6; quindi quando N raddoppia, Log N si incrementa di una costante e raddoppia solo quando N passa a N^2 .
N	Il tempo di esecuzione è <i>lineare</i> . Di solito accade quando il programma esegue poche operazioni sull'input. Se N raddoppia, raddoppia anche il tempo di esecuzione.
N Log N	Il tempo di esecuzione è <i>linearlogaritmico</i> . Solitamente si tratta di un programma che suddivide un grande problema in problemi minori, risolti indipendentemente e poi combina i risultati per ottenere la soluzione generale.
N^2	Il tempo di esecuzione è <i>quadratico</i> . Di solito si tratta di un algoritmo che deve risolvere problemi piccoli. Tempi del genere si hanno in algoritmi che elaborano dati a coppie, ad esempio in due cicli nidificati.
N^3	Il tempo di esecuzione è <i>cubico</i> . Si tratta di algoritmi che lavorano su terne di input (tre cicli nidificati ad esempio).
2^N	Il tempo di esecuzione è esponenziale. Pochi algoritmi di questo genere sono applicabili a problemi pratici.

In generale il tempo di esecuzione di un algoritmo è il valore di una costante per uno delle funzioni precedenti.

DIPENDENZA FUNZIONALE:TEMPO DI ESECUZIONE E NUMERO DI INGRESSI

Per poter trovare tale dipendenza si introduce la notazione O (si legge "o grande"): *Una funzione $h(N)$ è detta essere $O(f(N))$ se esistono due costanti c_0 e N_0 tali che $h(N) < c_0 f(N)$, $\forall N > N_0$.*

Tale definizione consente all'analista di liberarsi dal considerare macchina e compilatori; d'altra parte O è un utile strumento che individua l'estremo superiore del tempo di esecuzione (non si va oltre quindi). Lo scopo dell'analisi computazionale di un algoritmo è di dimostrare che il tempo di esecuzione è $O(f(N))$ per una qualche funzione f e che non esiste nessun algoritmo con un tempo di esecuzione $O(h(N))$, con h minore;

In altri termini tale che $\lim_{n \rightarrow \infty} \frac{h(N)}{f(N)} = 0$.

Senza volersi addentrare in dettagli matematici precisi, è giusto affermare che, in generale, f(N) presente nella definizione $O(f(N))$ è una delle funzioni di prima che danno luogo alla classificazione degli algoritmi. Non mostreremo però come si giunge a tali funzioni. Nel seguito, nell'analisi dei nostri algoritmi, faremo riferimento alla notazione O e comunque ad una classificazione degli algoritmi.

STRUMENTI E PROBLEMI

Per poter lavorare con numeri primi di valore molto elevato (10^{50} ad esempio) si va incontro ai seguenti problemi:

- Velocità degli algoritmi di elaborazione
- Scelta dei tipi da utilizzare negli algoritmi
- Architettura del computer a disposizione

Il primo tipo di problema comporta una attenta analisi del problema e una ottimizzazione dell'algoritmo utilizzato. Inoltre, per lavorare con la Teoria dei numeri,

si utilizzano spesso computer potenti con molta RAM e , ad esempio si usa il calcolo distribuito su computer a più CPU (multi-cpu), sfruttando ad esempio compilatori C come cilk adatti a lanciare parti di programma e calcoli ognuno su una CPU. Tale tipi di computer con accorgimenti algoritmici (programmazione parallela) consentono di parallelizzare i calcoli, suddividendoli in più parti eseguite contemporaneamente e sincronizzando alla fine i risultati dei vari task di calcolo.

Il secondo tipo di problema è legato alla scelta negli algoritmi dei tipi unsigned long int e double. Un unsigned long int tratta interi a 32 bit da 0 a $2^{32} - 1$ (escludendo lo zero) ovvero fino a 4294967296 circa; il double consente di trattare numeri a 64 bit per cui consente di trattare numeri fino a $2^{64} - 1$, ovvero a 18446744073709551616 circa e in forma scientifica (a virgola mobile) del tipo 10E50. Solo che la notazione scientifica complica gli algoritmi specie nei confronti tra numeri. Infatti il classico "Bug Alert" segnalato nella parte Operatori relazionali di un qualsiasi valido testo di linguaggio C è che la comparazione (<, >, ==) tra espressioni contenenti valori floating-point è che esse possono fallire (non essere vere). Ad esempio l'espressione:

$$(1.0/3.0 + 1.0/3.0 + 1.0/3.0) == 1.0$$

in matematica è vera ma per il computer no . Questo perché la divisione è affetta da una quantità periodica di numeri dopo la virgola (0.33333) che dipende anche dalla precisione e dagli arrotondamenti del computer e, quindi, alla fine la somma dei tre termini nella parte sinistra dell'espressione non sarà uguale a 1.

Il terzo problema è legato al tipo di architettura hardware a disposizione. Spesso per non ricorrere alla notazione scientifica è meglio disporre di computer non a 32 bit ma a 64 bit (ad esempio una SUN Solaris con più CPU a 64 bit). In tal caso si lavora più comodamente con gli unsigned long int. In tal caso occorre stare attenti a compilare e linkare con le librerie a 64 bit messe a disposizione della macchina e dal compilatore.

ALGORITMI PER SUPERARE I LIMITI DELL' ARCHITETTURA

Quando si deve lavorare con numeri infinitamente grandi, uno dei problemi è proprio quello delle operazioni con numeri con una quantità di cifre enorme, ben superiore di quelle disponibili con unsigned long int. Se si esegue un calcolo del genere il comportamento del programma è imprevedibile a fronte di un overflow. Spesso il programma su PC dà una risposta che non è corretta.

In generale con grandi numeri non è possibile lavorare nemmeno tanto con i double che hanno pur un limite ed inoltre fanno perdere di vista la fisicità del numero.

Per superare i limiti architetturali di un computer a 32 bit o a 64 bit, la soluzione è solo software; occorre, cioè, crearsi delle librerie matematiche che siano in grado di lavorare con numeri con grande quantità di cifre e di avere dei tempi di elaborazione accettabili. Come si può affrontare un progetto software del genere? Supponiamo di dover fare la moltiplicazione di due numeri X e Y con molte cifre, come ad esempio avviene nell'algoritmo di crittografia RSA per le chiavi pubbliche e private, ma il nostro hardware è limitato (non stiamo parlando solo di un PC).

METODO

E' possibile scomporre il numero di partenza in due altri numeri il cui numero di cifre significative è la metà della massima lunghezza rappresentabile dal dispositivo: es: 4 (8/ 2=4). Se il limite è minore o il numero da trattare è molto grande (centinaia di cifre) la tecnica si può iterare, suddividendo il numero di partenza in n numeri. I numeri

vedremo che andranno trattati come stringhe di caratteri e questo permette di conservare la notazione decimale indicando il numero con tutte le sue cifre.

prodotto X*Y

$$X = (x_1 + x_2) = (1234\ 0000 + 9999)$$

$$Y = (y_1 + y_2) = (9876\ 0000 + 9999)$$

$$X * Y = (x_1*y_1) + (x_1*y_2) + (x_2*y_1) + (x_2*y_2)$$

Vediamo cosa succede ai singoli prodotti

$$x_1*y_1 = 1234\ 0000 * 9876\ 0000 = 1234 * 9876 * 10^8 = 1218\ 6984\ 0000\ 0000$$

$$x_1*y_2 = 1234\ 0000 * 9999 = 1234 * 9999 * 10^4 = 1233\ 8766\ 0000$$

$$x_2*y_1 = 9999 * 9876\ 0000 = 9875\ 0124\ 0000$$

$$x_2*y_2 = 9999 * 9999 = 9998\ 0001$$

Se facessimo le somme con tali cifre supereremmo gli otto caratteri limite: il trucco allora è di non considerare gli zeri, rientrando negli otto caratteri. La somma logicamente viene fatta come in tabella.

Prodotto	C4	C3	C2	C1
x_1*y_1	1218	6984		
x_1*y_2		1233	8766	
x_2*y_1		9875	0124	
x_2*y_2			9998	0001
Somma	1219	8093	8888	0001

L'algoritmo deve sommare ogni colonna, badando di ottenere dalla somma solo 4 cifre; se le cifre sono maggiori di 4 la colonna ha un riporto a sinistra che va nella colonna successiva a sinistra.

Nell'esempio:

- C1 dà somma 0001
- C2 dà una somma 18888 che viene spezzato in C2d=8888 e C2s (riporto)=0001
- C3 dà una somma, che tenendo conto del riporto, è 18093 che si spezza in C3d=8093 e C3s(il riporto)=0001
- C4 dà una somma, che tenendo conto del riporto, è 1219

Il numero finale della somma è: 1219 8093 8888 0001 che è il risultato esatto. Qualche calcolo. Quanti Ci dobbiamo usare? Dipende dal numero più grande (Massimo numero di cifre) ottenuto dal prodotto (nell'esempio x_1*y_1) e dal massimo numero di cifre rappresentabili diviso 2.

$$\text{Massimo numero cifre } x_1*y_1 = 16$$

$$\text{Max cifre rappresentabili} = 8/2 = 4$$

$$\text{Num Ci} = 16/4 = 4 \text{ (si arrotonda per eccesso in generale).}$$

Somma X+Y

Il metodo ricade in quanto visto precedentemente.

Differenza X-Y

Esistono due strade alternative:

- si adegua l'algoritmo precedente (si aggiunge la differenza ma se ne può fare a meno)
- si adotta la tecnica del complemento a due.

Complemento a 2 su base 10

Dati A=54321 e B=12345 trovarne la differenza si può fare in questo modo:

Il complemento a 2 di B è: $\sim B = 10^5 - 12345 = 87655$.

Tale risultato era anche il complemento a 9 di ogni cifra.

Ora sommiamo $A + \sim B = 141976$. Trascuriamo l'ultima cifra e il risultato corretto della differenza $A - B = 41976$.

Divisione X/Y

Il metodo può essere affrontato iterativamente con la differenza.

Modulo X%Y

Si può basare sulle differenze. Ad esempio $12 \bmod 10 = 2$ e comunque sull'algoritmo della divisione.

Osservazione

In realtà l'unica operazione necessaria è la differenza e da essa si possono derivare tutte le altre operazioni.

Metodo alternativo adottato

Nel seguito mostriamo algoritmi, che si discostano da quanto visto precedentemente. Ad esempio la differenza non viene fatta come complemento a 2 potendo sfruttare la differenza normale tra interi del PC stesso.

Il concetto base su cui si possono basare delle librerie è che se siamo su un PC non è necessario porsi il problemi del numero di byte che con esso si può rappresentare, se possiamo usare una rappresentazione del tipo:

```
struct rappresentazione {
    char *store;
    int size;
};
```

```
typedef struct rappresentazione verylong;
```

E' da notare che per comodità di implementazione, i numeri possono essere conservati nella rappresentazione al rovescio, il che rende facile allinearli nei calcoli. Es: 1243 è rappresentato come 3421; cioè il MSB (Most Significant Byte) è l'ultimo, mentre il LSB (Least Significant Byte) è il primo.

Gli algoritmi di base sinteticamente si possono descrivere con uno pseudolinguaggio nel seguente modo:

```
v1_sum_1(n1, n2)
1. //sia ris un numero verylong
```

```

2. r ← 0
3. if n1.size > n2.size
4.   then max ← n1.size
5.   else max ← n2.size
6. for i ← 1 to min do // cifre comuni ad entrambi i numeri.
7.   t ← n1.store[i] + n2.store[i] + r
8.   ris.store[i] ← t mod 10
9.   r ← t div 10
10. // cifre definite per uno solo dei due numeri
11. if n1.size > n2.size
12.   then for i ← min+1 to max do
13.     t ← n1.store[i] + r
14.     ris.store[i] ← t mod 10
15.     r ← t div 10
16. if n2.size > n1.size
17.   then for i ← min+1 to max do
18.     t ← n2.store[i] + r
19.     ris.store[i] ← t mod 10
20.     r ← t div 10
21. // sistemazione del risultato
22. if r != 0
23.   then ris.store[max+1] ← r
24.   max ← max+1
25.   ris.size ← max
26. return ris

```

É facile vedere che questa somma è $O(n)$ nel numero di cifre del più grande dei due numeri.

```

vl_prod(n1, n2)
1. //acc e acc1 siano due numeri verylong
2. acc ← 0 //vedi commenti
3. for i ← 1 to n1.size do
4.   acc1 ← vl_prod_s(n2, n1.store[i-1])
5.   acc1 ← vl_shift(acc1, i)
6.   acc ← vl_sum(acc, acc1)
7. return acc

```

Se N è il numero di cifre del più grande dei due numeri l'algoritmo è $O(N^2)$.

Se entrambi i numeri hanno lo stesso N , allora ogni somma, shift e prodotto sono $O(N)$ ma a causa del ciclo ripetuto n volte è $N \cdot O(N)$ per cui $O(N^2)$.

```

vl_div(x, y) // si assume y != 0
1. r ← x // r é un numero verylong
2. q ← 0 // q é un numero verylong
3. w ← y // w é un numero verylong
4. while w ≤ x do w ← w*10
   // C1: w = 10^n * y > x and w/10 ≤ x per qualche n ∈ 0

```

```

5. while w != y // C2: q . w + r = x and r < w
6.   do q=q*10
7.     w←w/10 // C3: 10 w > r
8.     while w ≤ r
9.       do r←r-w
10.      q←q+1
11. /* C4: q . y + r =x and r<y */
12. return q

```

La divisione viene fatta per differenze successive, mettendo nel Resto la X e moltiplicando Y per dieci (shiftando con zeri) fino a raggiungere il numero di cifre di X. Ad ogni giro il quoziente deve avere il peso giusto e quindi deve essere moltiplicato per 10 per un numero di volte pari alla differenza di cifre tra il Resto e Y.

Facendo le differenze prima o poi X si accorcia di 1 cifra, per cui Y si shifta di tanti zeri quanto è lungo X. Alla fine il resto R è minore di Y e la sottrazione si arresta.

Esempio

X=R=7936; Y=12

7936 : 12 -> Q=661 con R=4

Come si procede per differenza? Il 12 viene shiftato (moltiplicato) per 2 volte; perché la differenza di cifre tra R e Y è 2. Lo shift, come si può vedere dal sorgente vlop.c, visto che i numeri sono rappresentati al rovescio aggiunge uno zero all'inizio. Es: 12 rovesciato è 21 e lo shift di 1 dà 021.

7936-1200=6736-1200=5536-1200=4336-1200=3136-1200=1936-1200=0736

Si sono dovute fare 6 sottrazioni affinché R perdesse una cifra. Per cui il quoto parziale è $6 * 10 * 10 = 600$ (proprio perché il quoto deve essere moltiplicato per 10 per due volte per la stessa ragione di Y).

Nel giro successivo R=736 e Y=120. Anche Q subirà una sola moltiplicazione. Per cui è:

736-120=616-120=496-120=376-120=256-120=136-120=16

Ci sono voluti altre 6 differenze, per cui il quoto di adesso è $6*10$ che sommato al precedente dà 660

Nel giro finale R=16 Y=12

16-12=4 per cui una sola differenza e il quoto parziale vale 1. Il quoto totale vale 661.

Anche questo algoritmo si può verificare che è $O(N^2)$. Somme, shift e differenze sono $O(n)$; mentre il ciclo in 5 viene fatto al più N volte, per cui $O(N^2)$.

vl_exp (a,b) \\ calcola a^b

```

1. if b=0
2.   then return 1
3.   if (b mod 2) = 0
4.     then d ← vl_exp(a,b div 2) \\ d è un numero verylong
5.     d ← d . d
6.     return d
7.   else d ← vl_exp(a,b-1)

```

8. $d \leftarrow d \cdot a$
 9. **return** d

Questo algoritmo si basa sul concetto “divide et impera”. Infatti è:

$$a^{2b} = (a^b)^2 \text{ e } a^{2b+1} = a \cdot (a^b)^2$$

`vl_expmod(a,b,m) \ \text{calcola } a^b \text{ mod } n`
 10. **if** b=0
 11. **then return** 1
 12. **if** (b **mod** 2) = 0
 13. **then** d \leftarrow vl_expmod(a,b **div** 2,m)
 14. $d \leftarrow (d \cdot d) \bmod m$
 15. **return** d
 16. **else** d \leftarrow vl_expmod (a,b-1,m)
 17. $d \leftarrow a \cdot d \bmod m$

Quest'altro algoritmo si basa sul concetto che:

$$a^{2b} \bmod n = (a^b \bmod n)^2 \bmod n$$

$$a^{2b+1} \bmod n = (a \cdot (a^b \bmod n)^2 \bmod n) \bmod n$$

Il numero di chiamate ricorsive che esegue l'algoritmo è dato da circa il numero di volte che si può dividere b per due fino a trovare 1, cioè circa $\log_2 b$.

Supponendo che a, b e m abbiano tutti circa n cifre decimali $\log_2 b$ sarà circa eguale a $3n$ (notare che $\log_2 10 \cong 3$). In ogni esecuzione del corpo della funzione c'è almeno una moltiplicazione, che abbiamo visto che è $O(N^2)$ e quindi l'intera funzione di esponenziale modulare è $O(N^3)$. Questa risulterà l'operazione più lenta di tutto il software.

This document was created with Win2PDF available at <http://www.win2pdf.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.