4.5 Tempo di calcolo: introduzione.

Perché un programma o sottoprogramma sia una soluzione accettabile di un problema, non basta che sia corretto rispetto alla specifica del problema; occorre anche che la risposta sia fornita in un tempo ragionevole, cioè che l'esecuzione del programma abbia una durata "non troppo lunga". Infatti, pur escludendo per ora i programmi con vincoli di "tempo reale", un programma che calcolasse una semplice funzione matematica (ad esempio la sequenza di Fibonacci) in modo corretto ma in tempi di secoli o millenni sarebbe di ben scarsa utilità (non si pensi ad esagerazioni: si vedrà nella seconda parte del corso che proprio una delle realizzazioni più "facili" della sequenza di Fibonacci ha questa spiacevole proprietà).

Naturalmente, il tempo di esecuzione di un programma dipende da molte cose, fra cui il tipo di calcolatore (cioè la velocità della sua unità centrale e dei suoi accessi alla memoria) e il valore dei dati in ingresso. Non possiamo quindi dare la misura della velocità di un programma fornendo un numero di millisecondi o di secondi; dovremmo invece fornire, per ciascun tipo di calcolatore, il tempo per ognuno degl'infiniti possibili valori di input. Dovremmo fornire, cioè, una funzione: se chiamiamo n il valore del parametro di input (che per ora, per semplicità, supponiamo unico e di tipo numerico), dovremmo fornire una funzione $t = T(n)$.

Come si è detto, la funzione $T(n)$ sarà diversa, per quanto riguarda i suoi valori numerici, da macchina a macchina; tuttavia, poiché ciò che distingue una macchina dall'altra è solo la velocità delle operazioni elementari, ma non la loro natura di base, l'andamento di $T(n)$ al variare (cioè al crescere) di n resterà lo stesso, nel senso che se ad esempio è $T(n) = an^2 + bn$, con a e b costanti, tale formula varrà per qualunque macchina, e muteranno soltanto i valori delle costanti a e b; in ogni caso il tempo crescerà sempre in modo quadratico rispetto ad n, e il suo grafico sarà costituito da una parabola.

In corsi successivi sarà definita rigorosamente la nozione di *complessità temporale* di un programma; intanto possiamo esaminare da questo nuovo punto di vista i programmi presentati fin qui.

Quelli per il calcolo dell'esponenziale impiegano, per calcolare l'n-esima potenza di x, un tempo lineare in n (ossia proporzionale a n), poiché (dopo le istruzioni di inizializzazione) ripetono n volte una sequenza di due istruzioni semplici; se chiamiamo a il tempo costante necessario per eseguire il test e il corpo del ciclo, e b il tempo necessario per le istruzioni di inizializzazione, il tempo di calcolo totale è (a parte qualche costante in piú o in meno):

$$T(n) = an + b$$

Si osservi che spesso la soluzione "piú naturale" di un problema è quella piú inefficiente: per scoprirne di migliori, oppure per scoprire che non possono esistere soluzioni migliori, sono di solito necessari ragionamenti di natura matematica.

Un esempio banale è il problema del calcolo della somma dei naturali da 1 a n; l'algoritmo naturale, consistente nel sommare ad uno ad uno tutti gli n numeri, impiega un tempo evidentemente proporzionale ad n; l'algoritmo scoperto da Gauss bambino, consistente nel calcolare l'espressione $n(n+1)/2$, richiede l'esecuzione di solo tre operazioni, che (almeno per n non troppo grande) l'uomo e a maggior ragione la macchina compiono in tempo costante. Si è passati così da un andamento lineare ad un andamento costante: ma per questo è stato necessario dimostrare un piccolo teorema.

4.6 L'esponenziale veloce.

4.6.1 Costruzione del programma.

Cerchiamo di costruire un algoritmo per il calcolo di X^N più veloce rispetto all'algoritmo ovvio, sfruttando una nota proprietà delle potenze:
se N è pari allora $X^N = (X^2)^{N \text{ div } 2}$

Usando una volta tale formula per calcolare una potenza con esponente pari N, invece di N moltiplicazioni ci basta fare una moltiplicazione per ottenere il quadrato, e poi N/2 moltiplicazioni: abbiamo quindi un risparmio di circa metà tempo.

Se ora per semplicità chiamiamo W il quadrato di X, e M la metà di N, se per caso M è ancora pari possiamo applicare la stessa formula anche nel calcolo di W^M; se invece M è dispari, allora M-1 è pari, e possiamo quindi applicare lo stesso metodo per calcolare W^{M-1}, e cosí via ... induttivamente, ottenendo - come vedremo - ben piú di un semplice risparmio di tempo di un fattore costante indipendente da N.

Come si scrive un programma che realizzi correttamente un tale metodo di calcolo? Più in generale, come si passa dall'idea di base di un algoritmo alla realizzazione di un programma corretto? Cerchiamo, per una volta, di esporre in dettaglio un percorso di ragionamento che conduce alla scrittura del programma.

Partiamo dall'analisi, su un paio di esempi, del modo in cui eseguiremmo il calcolo a mano. Prendiamo dapprima con un esponente che sia una potenza di due, ad esempio 16:

$3^{16} = 9^8 = 81^4 = 6561^2 = 43046721^1 = 43046721.$

Come si vede, il risultato si ottiene con sole quattro moltiplicazioni invece di sedici:

$3 \cdot 3 = 9;\ \ 9 \cdot 9 = 81;\ \ 81 \cdot 81 = 6561;\ \ 6561 \cdot 6561 = 43046721.$

È chiaro che per gli esponenti potenze di due il programma potrebbe essere costituito da un semplice ciclo *while* che ad ogni iterazione moltiplichi per se stesso il risultato precedente e divida per due l'esponente (e termini quando l'esponente arriva ad 1); ma si tratta di un caso troppo particolare.

Proviamo allora con un esponente che non sia una potenza di due, ad esempio 13:

$3^{13} = \mathbf{3 \cdot 3^{12}} = \mathbf{3 \cdot 9^6} = \mathbf{3 \cdot 81^3} = 3 \cdot 81 \cdot 81^2 = \mathbf{243 \cdot 81^2} = \mathbf{243 \cdot 6561^1} = 1594323$

Come si vede, la traccia del calcolo si può pensare costituita da una sequenza di espressioni numeriche (quelle evidenziate in grassetto): ognuna di esse è un prodotto di due fattori, dove il fattore di sinistra è sempre un valore già calcolato (come risultato dei passi precedenti), mentre il fattore di destra è una potenza ancora da calcolare, data da una coppia base-esponente. Il fattore di sinistra è quindi un risultato parziale; il fattore di destra esprime un calcolo ancora da fare, dove però si noti che la base non è piú in generale quella di partenza, ma è anch'essa il risultato di passi di calcolo precedenti.

Possiamo allora tenere il fattore di sinistra in una variabile `ris`, e tenere base ed esponente del fattore di destra in due variabili rispettivamente `x` ed `n`; l'invariante è quindi lo stesso della versione ingenua:

INVARIANTE: $x^n \cdot ris = X^N \quad \wedge \quad n \geq 0$

Ne consegue che, per avere il risultato finale in `ris`, devono essere le stesse anche l'inizializzazione e la condizione di uscita (quindi il test del *while*):

```
ris = 1;
while(n > 0) { ...
```

INVARIANTE: $x^n \cdot ris = X^N \quad \wedge \quad n \geq 0$

Ciò che è diverso rispetto alla versione precedente è il corpo del ciclo: esso dovrà infatti eseguire un nuovo tipo di passo di calcolo, pur mantenendo lo stesso invariante, cioè lasciando immutato il valore dell'espressione $x^n \cdot ris$. Poniamoci mentalmente all'inizio di una generica iterazione, assumendo che un risultato parziale si trovi in `ris`, e che x^n sia la quantità (non calcolata) "ancora da moltiplicare per `ris`".

Ci sono due casi, a seconda che il contenuto di `n` sia pari o dispari. Se `n` è pari si ha l'uguaglianza $x^n \cdot ris = (x^2)^{n/2} \cdot ris$. Se allora al posto del numero contenuto in `x` mettiamo il suo quadrato e simultaneamente dimezziamo il contenuto di `n`, il valore di x^n non cambia e quindi, lasciando invariato `ris`, il valore di $x^n \cdot ris$ non cambia:

```
if (n è pari) {
  x = x*x; n = n/2          // l'ordine non importa
}
```

Così si mantiene anche la seconda parte dell'invariante: se infatti `n` è (per la condizione del *while*) un intero > 0, `n`/2 è un intero ≥ 0.

Nel caso in cui `n` è dispari, la sopra ricordata proprietà delle potenze non si può applicare; poiché però l'invariante è lo stesso della soluzione ingenua, si può comunque eseguire in tal caso la stessa coppia di istruzioni che si eseguiva là:

```
else {
  ris = x*ris; n--;  (* l'ordine non importa *)
}
```

Rispetto alla soluzione ingenua è diversa solo la situazione in cui le due assegnazioni sono eseguite: come avevamo già notato nell'esempio numerico, x in generale non è piú uguale al valore iniziale della base. Scriviamo in conclusione una versione completa della funzione:

```
static double expVeloce(double x, int n) {
  double ris = 1;
  while(n > 0) {
    if(n%2 == 0) {  // n pari
      x = x*x;
      n = n/2;
    }
    else {              // n dispari
      ris = x*ris;
      n--;
    }
  }
  return ris;
}
```

4.6.2 Terminazione e correttezza totale.

Chi veda per la prima volta, senza le spiegazioni sopra riportate, il (sotto)programma appena scritto, potrebbe chiedersi a che cosa serva un passo di calcolo come quello del caso pari, che non porta alcun contributo all'accumulazione del risultato in `ris`.

Naturalmente la risposta è che un calcolo viene comunque fatto, in `x` invece che in `ris`. Il valore eventualmente accumulato in `x` attraverso una sequenza di passi "di nuovo tipo" verrà "moltiplicato in" `ris` al prossimo passo "di vecchio tipo", cioè non appena a forza di dividere per 2 si ottiene un esponente dispari. Ma non potrebbe succedere che non si ottenga mai un esponente dispari e più in generale che il programma non termini? Si vede subito che no.

Infatti, per l'invariante si ha sempre $n \geq 0$; d'altra parte ogni volta che il corpo del ciclo viene eseguito si deve avere prima dell'esecuzione $n > 0$ (per il test del *while*), ma:

se $n > 0$ allora $n/2 < n$ e $n-1 < n$

I valori successivamente assunti da `n` costituiscono perciò una successione strettamente decrescente di interi non negativi, che deve quindi necessariamente terminare con 0 (in parole povere: ad ogni iterazione `n` diventa strettamente piú piccolo, pur rimanendo un intero non negativo; allora prima o poi diventerà zero). Con ciò è dimostrato che la condizione di uscita dal *while* viene sempre raggiunta, e che quindi per qualunque input corretto si ha in tempo finito la risposta corretta.

Ai fini della comprensione di come il programma funziona, si può notare che se il valore iniziale di `n` è maggiore di zero il penultimo valore della successione è sempre il numero dispari 1: infatti ciò è ovvio se il valore iniziale di `n` è 1; d'altra parte, se è $n > 1$ si ha $n/2 > 1$ e $n-1 \geq 1$, e quindi prima o poi si ottiene $n=1$.

La proprietà di terminazione è particolarmente evidente se si pensa alla rappresentazione binaria. In essa i numeri pari e i numeri dispari sono semplicemente quelli rispettivamente con zero e con uno nel bit più a destra (il meno significativo); il decremento di 1 di un numero dispari equivale a sostituire uno con zero nel suo bit piú a destra, la divisione intera per 2 corrisponde a eliminare il bit meno significativo, cioè ad un'operazione di scorrimento (*shift*) verso destra (con immissione di 0 nel bit piú a sinistra, cioè il più significativo). La successione di divisioni per 2 e decrementi di 1 equivale perciò ad una successione di *shift* verso destra, che termina necessariamente con tutti i bit a zero, cioè con la rappresentazione binaria del numero 0.

La dimostrazione che quando si raggiunge la condizione di uscita si realizza la condizione finale richiesta è detta di correttezza parziale; la successiva o contemporanea dimostrazione che la condizione di uscita viene (sempre) raggiunta, cioè che il programma termina (sempre), permette di stabilire la cosiddetta correttezza totale del programma.

Di un programma del quale si possa dire che, se terminasse, calcolerebbe il risultato correttamente, ma in realtà non termina mai, non sapremmo che farcene! Nè sarebbe molto utile un programma che calcolasse il risultato corretto per certi valori di input, ma non terminasse per altri.

4.6.3 Complessità.

Analizziamo per il nuovo algoritmo l'andamento del tempo di calcolo in funzione del valore iniziale di n (che indichiamo semplicemente con *n*).

Consideriamo dapprima l'andamento di *T(n)* al crescere di *n* soltanto per quei valori di *n* che sono potenze di due (come 2, 4, 8, 16, 32, ...), cioè per i *casi migliori*. Sia dunque $n = 2^k$. Dividendo per due si ottiene di nuovo un numero pari 2^{k-1}, e cosí via fino a 1. Il ciclo *while* viene quindi ripetuto *k+1* volte: *k* volte per ridurre il valore di n a 1, piú una volta per passare da 1 a 0.

Quindi il tempo di calcolo è proporzionale a *k+1*; ma $k = \log_2 n$, dunque:

$T(n) @ \log_2 n + 1$

Per un numero pari generico naturalmente non è vero che la sua metà sia necessariamente pari; osserviamo tuttavia che se un numero è dispari, decrementandolo di 1 si ottiene necessariamente un numero pari. Allora a un passo di decremento segue sempre immediatamente un passo di divisione, e alla peggio si avranno per tutta l'esecuzione un passo di decremento e uno di divisione alternati.

Consideriamo l'andamento di *T(n)* al crescere di *n* soltanto per quei valori di *n* costituenti tali *casi peggiori*: cioè quei valori che divisi per due danno un numero dispari che a sua volta decrementato di 1 e diviso per due dà di nuovo un dispari, e cosí via.

Se si pensa di nuovo alla rappresentazione binaria, si vede subito che tali numeri sono quelli in cui dopo ogni *shift* verso destra si ritrova 1 nel bit piú a destra (il meno significativo); sono cioè i numeri formati da tutti 1, ossia quegli *n* tali che $n = 2^k - 1$ (come 3, 7, 15, 31, ...). Per passare da $2^k - 1$ a 0 sono quindi necessari *k* decrementi e *k-1* divisioni, in totale *2k-1* passi (cioè *2k-1* iterazioni del ciclo). Ma è $k = \log_2(n+1)$, quindi:

$T(n) @ 2\log_2(n+1) - 1 @ 2\log_2 n$

Per valori generici di *n* abbiamo evidentemente risultati intermedi fra il caso migliore e il caso peggiore; in generale si ha pertanto:

$$\log_2 n + 1 £ T(n) £ 2\log_2(n+1) - 1$$

(ossia, in modo piú grossolano: $\log_2 n £ T(n) £ 2\log_2 n$)

Una funzione di questo genere che, pur essendo rappresentata da una curva "irregolare", è però delimitata inferiormente e superiormente da due funzioni logaritmiche, cioè ha il grafico compreso fra due curve logaritmiche, si dice comunque che ha *andamento (strettamente) logaritmico*; simbolicamente si scrive

$T(n) = Q(n)$ (si legge: *T(n) è "theta grande" di n*)

L'esponenziale veloce ha dunque, rispetto all'esponenziale ingenuo, complessità temporale logaritmica invece che lineare; quindi, per *n* sufficientemente grande, il tempo di esecuzione dell'algoritmo veloce sarà inferiore a quello dell'algoritmo ingenuo.