Ideologia divide et impera

corso di laurea in **Matematica**

Informatica Generale, Lezione **11(b)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Moltiplicazione intera

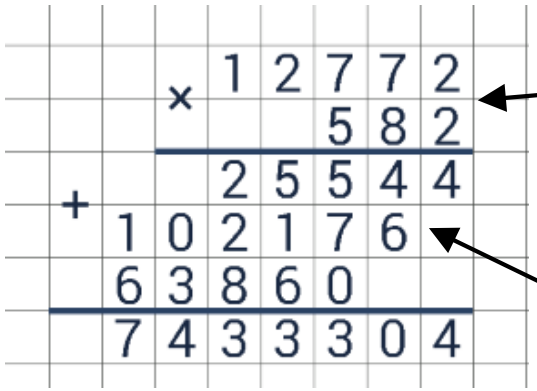
Complessità di somma e prodotto

Abbiamo rivisto nella Lezione **1**, gli usuali algoritmi di somma e prodotto cifra a cifra imparati fin dalle elementari.

Ma **qual è la loro vera complessità**?

Assumiamo i due numeri da sommare/moltiplicare abbiano la **stessa lunghezza** k, che significa $k = \log_b n$, dove b è la base con cui rappresento i numeri.

Quindi l'algoritmo ha costo $\Theta(k^2) = \Theta(\log_b{}^2 n)$. Si può fare meglio?



k prodotti per una cifra $\Theta(k)$ per un totale di $\Theta(k^2)$ operazioni elementari

k somme lunghe $2k$, ciascuna di costo $\Theta(k)$, per un totale di $\Theta(k^2)$

Idea Divide et Impera per il prodotto

Consideriamo due numeri x e y e scriviamoli scritti come sequenze di cifre (non è rilevante la base) e **dividiamoli in due** (al solito non è rilevante il caso dispari/pari):

$$x = x_0\, b^{k/2} + x_1 \;\text{ e }\; y = y_0\, b^{k/2} + y_1$$

A questo punto avrò che posso scomporre il prodotto come segue:

$$x \cdot y = x_0 \cdot y_0\, b^{k} + (x_0 \cdot y_1 + x_1 \cdot y_0)\, b^{k/2} + x_1 \cdot y_1$$

siamo riusciti a scrivere il **prodotto di due numeri di k cifre** in termini di **4 prodotti di numeri di $k/2$ cifre**, ma c'è guadagno?

Questo algoritmo porta alla relazione di ricorrenza:

$$T(k) = 4\, T(k/2) + \Theta(k)$$

e siccome $\log_2 4 = 2$ e $k^2 = \mathbf{\Omega}(k)$ per il Teorema Principale ho che la soluzione dell'equazione di ricorrenza è $\boldsymbol{\Theta(k^2)}$, quindi ho solo trovato un **modo più difficile** (**ricorsivo**) di definire **l'usuale metodo** noto fin dall'infanzia (**iterativo**).

Idea Divide et Impera per il prodotto

Tuttavia, ci sono altri modi di ragionare.

Consideriamo il seguente prodotto: Qui ci sono i **3 personaggi che voglio determinare**, e cioè $x_0\, y_0$, $x_0\, y_1 + x_1\, y_0$ e $x_1\, y_1$.

$$p = (x_0 + x_1)\,(y_0 + y_1) = x_0\, y_0 + (x_0\, y_1 + x_1\, y_0) + x_1\, y_1$$

Rassegnandoci a calcolare $x_0 \cdot y_0$ e $x_1\, y_1$, ho che

$$x_0\, y_1 + x_1\, y_0 = p - (x_0\, y_0 + x_1\, y_1)$$

e quindi posso ottenere le **3 sequenze cercate** a costo di **3 prodotti** e **qualche operazione lineare** in k (somme e differenze)

da cui otteniamo un algoritmo di complessità descritta dalla seguente relazione di ricorrenza:

$$T(k) = \mathbf{3}\; T(k/2) + \Theta(k)$$

e siccome $\log_2 3 = 1.59\ldots$ e $k^{1.59} = \mathbf{\Omega}(k)$ per il Teorema Principale ho che la soluzione dell'equazione di ricorrenza è $\Theta(k^{1.59})$, quindi ho che il prodotto di due numeri di k cifre può essere fatto in $\Theta(k^{1.59})$.

Caso base: sfruttare la macchina!

Ma qual è il **caso base**? Per il nostro bambino interiore è sempre il caso di numeri a 1 sola cifra. I risultati possono avere 2 cifre ma poco male, li devo solo sommare/sottrarre con l'usuale algoritmo di somma cifra a cifra.

Ma per il nostro esecutore d'elezione, la **macchina di von Neumann** che organizza le informazioni su **parole di memoria di l bit**? Conviene sfruttare il parallelismo offerto dalla macchina e considerare come caso base **sequenze di cifre rappresentabili su $l/2$ bit**, che rappresentano valori fino a $2^{l/2}-1$.

Infatti moltiplicando due numeri $x, y \leq 2^{l/2}-1$ ho che $xy < 2^l$ che è rappresentabile in una parola di memoria. Ragionando su sequenze di bit, quindi, **il caso base saranno le sequenze lunghe al massimo $l/2$** che posso moltiplicare con **un'unica operazione** della **macchina**.

Esempio: se avete una macchina a 64 bit, potete moltiplicare due numeri fino a $4{,}294 \cdot 10^9$ senza generare overflow. Ovviamente, dovendo produrre stampe in decimale… c'è del lavoro da fare… (oppure ragionare su sequenze di cifre decimali di al più 9 cifre…)

Pseudocodice

La scelta del caso base **non influenza la complessità asintotica**: ma pensare in termini di numeri di 32 bit **accelera la computazione di un fattore costante circa 9**, rispetto a pensare in termini di prodotti di singole cifre decimali! Alla faccia della complessità asintotica…

(Possiamo considerare numeri di lunghezza sempre potenze di 2, riempiti di zeri non significativi a sinistra)

Osservazione: riempire a sinistra di 0 o estrarre metà numero sono operazioni "facili".

```
def multiplyD&I(x, y, k):
    if k < maxDigit: return x*y
    Siano x_1, x_0 tali che x = x_0 2^(k/2) + x_1
    Siano y_1, y_0 tali che y = y_0 2^(k/2) + y_1
    p = multiplyD&I(somma(x_1, x_0, k/2),
            somma(y_1, y_0, k/2), k/2)
    a = multiplyD&I(x_1, y_1, k/2)
    c = multiplyD&I(x_0, y_0, k/2)
    b = diff(p, somma(a, b, k), k)
    return somma(b*2^k, somma(b*2^(k/2), c, k), 2*k)
```

3 chiamate ricorsive su numeri lunghi k/2

somma e diff lineari in 2k, k e k/2

Fibonacci veloce

Trasformazione di Fib. come matrice

La **trasformazione** di **un'iterazione** del calcolo dei numeri di Fibonacci può essere descritta da una **matrice** (a dire il vero ogni trasformazione se è semplicemente una sequenza di assegnamenti o, meglio ancora, un unico assegnamento parallelo).

Nel nostro caso abbiamo:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$

Questo però significa anche (per **associatività**) che:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

È interessante anche osservare che a ben vedere le potenze di questa matrice contengono sempre numeri di Fibonacci. Lo vediamo per induzione:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} \text{ e inoltre: } \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_{n+3} & F_{n+2} \\ F_{n+2} & F_{n+1} \end{bmatrix}$$

Esponenziale di una matrice

D'altra parte, le proprietà algebriche che valgono per i numeri, valgono senz'altro per le **matrici quadrate**.

Quindi: l'esponenziale si può calcolare **parentesizzando ad albero bilanciato** i prodotti, esattamente come fanno l'**esponenziale** e la **moltiplicazione egiziana**.

Risultato: posso calcolare l'esponenziale A^n di una matrice quadrata con $\theta(\log n)$ prodotti tra matrici.

Generalizzazione: questo metodo si può applicare a tutte le operazioni **associative**, in quanto posso scegliere diversi ordini in cui di eseguire i calcoli (che a volte permette di evitare di rifare conti uguali). Infatti, se $*$ è associativa:

$$((a * a) * a) * a = (a * a) * (a * a)$$

A destra ho 3 occorrenze di $*$ ma due si applicano agli stessi valori: **basta calcolarne 2**. La moltiplicazione egiziana (**Programmazione Dinamica**) accelera il calcolo evitando **sottocasi uguali**.

Complessità di Fibonacci

Trattandosi di matrici quadrate di lato 2, ogni passo elementare necessita di **8** prodotti e **4** somme (numero costante!).

I **numeri** però **crescono** fino a $\boldsymbol{\phi^n}$ ($\phi = \frac{1+\sqrt{5}}{2}$) e quindi una moltiplicazione costa $n \log^{1.59} \phi = \Theta(n)$. Complessivamente quindi **calcolare** l'n-esimo numero di **Fibonacci** si può fare in tempo $\boldsymbol{\Theta(n \log n)}$.

L'algoritmo **iterativo classico**, fa $\Theta(n)$ somme, ciascuna di costo $\Theta(n \log \phi)$, quindi complessivamente in tempo $\boldsymbol{\Theta(n^2)}$.

L'algoritmo **ricorsivo inefficiente**, fa ϕ^n somme di costo alla peggio $\log \phi^n = n \log \phi$. Per cui ha una complessità $\boldsymbol{\Theta(n\, \phi^n)}$.

Per quanto detto sulla **moltiplicazione**, va ricordato che queste operazioni risultano ragionevolmente efficienti per n ragionevolmente grandi, perché le **costanti moltiplicative** delle operazioni aritmetiche sono **molto basse**.