Moltiplicazioni di interi
Relazioni di ricorrenza e
Teoremi generali

23 marzo 2022

Applicazione di D&I alla Moltiplicazione di Interi

Come si moltiplicano due numeri? In seconda elementare abbiamo appreso che si fà così:

$$\begin{array}{r} 27\times \\ \underline{32\ =} \\ 54 \\ \underline{81\quad\ } \\ 864 \end{array}$$

e vale anche per numeri in binario!

$$
\begin{array}{r}
011011\times \\
\underline{110110 =} \\
000000 \\
011011 \\
011011 \\
000000 \\
011011 \\
\underline{011011} \\
10110110010
\end{array}
$$

Qual è la complessità dell'algoritmo che moltiplica, nel modo "elementare", due numeri di n bits ciascuno?

Vi sono n moltiplicazioni parziali, in ciascuna delle quali si moltiplicano n bits, in totale $O(n^2)$ moltiplicazioni di bit

Contando anche le addizioni, altre $O(n^2)$, otteniamo che l'algoritmo elementare effettua $O(n^2)$ operazioni per moltiplicare due numeri di n bit

Possiamo fare meglio?

Applicazione di D&I alla Moltiplicazione di Numeri

Ricordiamo il paradigma centrale della tecnica D&I: *"dividi il problema in sottoproblemi, risolvi i sottoproblemi, e combina le relative soluzioni in una soluzione per il problema globale".*

In ossequio a tale paradigma, dato un numero di n bit x iniziamo a dividerlo in bit di "ordine alto" e bit di "ordine basso", ovvero

$$x = x_1 \cdot 2^{n/2} + x_0$$

. Esempio:

$$x = \underbrace{11001110}_{x_1}\underbrace{00111001}_{x_0} = x_1\, 2^8 + x_0$$

$$x = 11001110\ 00000000 +$$
$$00111001 =$$
$$\text{---------------------}$$
$$11001110\ 00111001$$

Facciamo lo stesso con l'altro numero y da moltiplicare, scrivendo

$$y = y_1 \cdot 2^{n/2} + y_0$$

Esempio

$x = 101\ 110 = 46_{10}$

$x_1 = 101 = 5_{10}$

$x_2 = 110 = 6_{10}$

$x = 101\ 000 + 110 = 5 \cdot 2^3 + 6 = 46_{10}$

avendo scritto $x = x_1 \cdot 2^{n/2} + x_0$ e $y = y_1 \cdot 2^{n/2} + y_0$ otteniamo

$$\begin{aligned} x \cdot y &= (x_1 \cdot 2^{n/2} + x_0) \cdot (y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \end{aligned} \quad (1)$$

L' uguaglianza (1) riduce il problema di calcolare **un** singolo prodotto $x \cdot y$ di due numeri di n bit ciascuno, nel problema di calcolare **quattro** prodotti $(x_1 \cdot y_1, x_1 \cdot y_0, x_0 \cdot y_1, x_0 \cdot y_0)$ di numeri di $n/2$ bit ciascuno.

avendo scritto $x = x_1 \cdot 2^{n/2} + x_0$ e $y = y_1 \cdot 2^{n/2} + y_0$ otteniamo

$$\begin{aligned} x \cdot y &= (x_1 \cdot 2^{n/2} + x_0) \cdot (y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \end{aligned} \quad (1)$$

L' uguaglianza (1) riduce il problema di calcolare **un** singolo prodotto $x \cdot y$ di due numeri di n bit ciascuno, nel problema di calcolare **quattro** prodotti $(x_1 \cdot y_1, x_1 \cdot y_0, x_0 \cdot y_1, x_0 \cdot y_0)$ di numeri di $n/2$ bit ciascuno.

Abbiamo quindi un abbozzo di algoritmo D&I: "*calcola ricorsivamente ciascuno dei 4 sottoproblemi, di dimensione* $n/2$ *ciascuno, e combina i risultati ottenuti usando l'equazione* (1)".

Domanda: usando la tecnica D&I, di quanto abbiamo "stracciato" l'agoritmo appreso in seconda elementare?

Vediamo, Dobbiamo calcolare...

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

Detto $T(n)$ il numero di operazioni che il nostro algoritmo impiega per calcolare il prodotto di due numeri di n bit, avremo che

$$T(n) = 4T(n/2) + dn$$

Dove abbiamo fallito?

Abbiamo tentato di risolvere un problema di taglia n mediante 4 chiamate ricorsive a risoluzioni di problemi di taglia $n/2$

Possiamo diminuire il numero di chiamate ricorsive per calcolare

$$x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 ?$$

Vediamo...

Occorre calcolare $x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0$

Partiamo da $x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0 = (x_1 + x_0) \cdot (y_1 + y_0)$

che ci possiamo calcolare con una singola chiamata ricorsiva al prodotto del numero $(x_1 + x_0)$ per $(y_1 + y_0)$.

Occorre calcolare $x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0$

Partiamo da $x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0 = (x_1 + x_0) \cdot (y_1 + y_0)$ che ci possiamo calcolare con una singola chiamata ricorsiva al prodotto del numero $(x_1 + x_0)$ per $(y_1 + y_0)$.

Se ora ci calcoliamo x_1y_1 (con **una** chiamata ricorsiva) e x_0y_0 (con **un'altra** chiamata ricorsiva), ci possiamo calcolare

$$x_1y_0 + x_0y_1 = (x_1 + x_0) \cdot (y_1 + y_0) - x_1y_1 - x_0y_0$$

Occorre calcolare $x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0$

Partiamo da $x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0 = (x_1 + x_0) \cdot (y_1 + y_0)$ che ci possiamo calcolare con una singola chiamata ricorsiva al prodotto del numero $(x_1 + x_0)$ per $(y_1 + y_0)$.

Se ora ci calcoliamo x_1y_1 (con **una** chiamata ricorsiva) e x_0y_0 (con **un'altra** chiamata ricorsiva), ci possiamo calcolare

$$x_1y_0 + x_0y_1 = (x_1 + x_0) \cdot (y_1 + y_0) - x_1y_1 - x_0y_0$$

semplicemente sottraendo x_1y_1 e x_0y_0 da $(x_1 + x_0) \cdot (y_1 + y_0)$.

Quindi, con solo 3 chiamate ricorsive ci siamo calcolati tutti i termini che compaiono nel prodotto

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

e possiamo calcolarci tranquillamente $x \cdot y$

Algoritmo D&I per il calcolo di prodotti:

`Recursive-Multiply`(x, y)

  `Scrivi` $x = x_1 \cdot 2^{n/2} + x_0$

  $y = y_1 \cdot 2^{n/2} + y_0$

  `Calcola` $x_1 + x_0$ `e` $y_1 + y_0$

  $p \leftarrow$ `Recursive-Multiply`$(x_1 + x_0, y_1 + y_0)$

  $x_1 y_1 \leftarrow$ `Recursive-Multiply`(x_1, y_1)

  $x_0 y_0 \leftarrow$ `Recursive-Multiply`(x_0, y_0)

  `return` $x_1 y_1 \cdot 2^n + (p - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$

Detta $T(n)$ la complessità di `Recursive-Multiply`(x, y) per numeri x e y n bits, avremo

$$T(n) = 3T(n/2) + dn$$

Risolviamo $T(n) = 3T(n/2) + dn$

Ricordiamo che equazioni del tipo

$$T(n) = aT(n/c) + dn, \quad \text{con } a > c$$

hanno come soluzione $T(n) = O(n^{\log_c a})$, il che, nel nostro caso ci dice

$$\boxed{T(n) = O(n^{\log_c a}) = O(n^{\log_2 3}) = O(n^{1.59})}$$

migliorando, finalmente, un algoritmo da seconda elementare...

Per avere una idea del miglioramento, osserviamo che $n^2 \approx 17 \times n^{\log_2 3}$ per $n = 1000$, $n^2 \approx 309 \times n^{\log_2 3}$ per $n = 1000000$, e $n^2 \approx 5436 \times n^{\log_2 3}$ per $n = 1000000000$.

più seriamente...

Idee simili possono essere usate per ottenere algoritmi efficienti per calcolare (rapidamente) la *convoluzione* di due vettori.

Tale operazione é cruciale in molti campi dell'Analisi dei Segnali ed ha applicazioni pratiche importantissime (TAC, riconoscimento vocale, compressione di immagini JPEG, riconoscimento automatico di immagini, trattamento di segnali musicali, astronomia, ...)

Un algoritmo per il calcolo della convoluzione di vettori basato sulle idee che abbiamo presentato ha rivoluzionato il campo dell'Analisi dei Segnali.

Tale algoritmo, chiamato **Fast Fourier Transform** (FFT), fu scoperto da Cooley e Tukey nel 1965 (sebbene Gauss nel 1805 ne avesse pensato uno simile).

L'algoritmo FFT é entrato nella lista "Top Ten Algorithms of the Century", compilata dalla rivista *Computing in Science and Engineering* nel Gennaio 2000.

Relazioni di ricorrenza per vari algoritmi Divide-et-Impera

- Dividi il problema di taglia n in a sotto-problemi di taglia n/b
- Ricorsione sui sottoproblemi
- Combinazione delle soluzioni

$T(n)$= tempo di esecuzione su input di taglia n

$T(n) = D(n) + a\,T(n/b) + C(n)$

Alcune relazioni di ricorrenza

Abbiamo considerato una sotto-famiglia

- $T(n) = qT(n/2) + cn$ con $T(2)=c$

  per $q=1$ allora $T(n)= O(n)$

  $q=2$ allora $T(n)= O(n \log_2 n)$

  $q>2$ allora $T(n)= O(n^{\log_2 q})$

Più in generale.....

- $T(n)=2T(n/2) +cn^2$

  $T(n) = O(n^2)$

Un teorema generale

Teorema: Se n é potenza di c, la soluzione alla ricorrenza

$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ aT(n/c) + bn & \text{altrimenti} \end{cases}$$

é

$$T(n) = \begin{cases} O(n) & \text{se } a < c \\ O(n \log n) & \text{se } a = c \\ O(n^{\log_c a}) & \text{se } a > c \end{cases}$$

Esempi:

- Se $T(n) = 2T(n/3) + dn$, allora $T(n) = O(n)$
- Se $T(n) = 2T(n/2) + dn$, allora $T(n) = O(n \log n)$
- Se $T(n) = 4T(n/2) + dn$, allora $T(n) = O(n^2)$

Nota: Slide precedente per c=2; a=q (libro [KT]) .

Esempi

Sia $T(1) = 1$. Valutiamo

- $T(n) = 2T(n/2) + 6n$ $\quad T(n) = O(n \log n)$
- $T(n) = 3T(n/3) + 6n - 9$ $\quad T(n) = O(n \log n)$
- $T(n) = 2T(n/3) + 5n$ $\quad T(n) = O(n)$
- $T(n) = 2T(n/3) + 12n + 16$ $\quad T(n) = O(n)$
- $T(n) = 4T(n/2) + n$ $\quad T(n) = O(n^{\log_2 4}) = O(n^2)$
- $T(n) = 3T(n/2) + 9n$ $\quad T(n) = O(n^{\log_2 3}) = O(n^{1.584\ldots})$

Un teorema generale

Teorema: Se n é potenza di c, la soluzione alla ricorrenza

$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ aT(n/c) + bn & \text{altrimenti} \end{cases}$$

é

$$T(n) = \begin{cases} O(n) & \text{se } a < c \\ O(n \log n) & \text{se } a = c \\ O(n^{\log_c a}) & \text{se } a > c \end{cases}$$

Esempi:

- Se $T(n) = 2T(n/3) + dn$, allora $T(n) = O(n)$
- Se $T(n) = 2T(n/2) + dn$, allora $T(n) = O(n \log n)$
- Se $T(n) = 4T(n/2) + dn$, allora $T(n) = O(n^2)$

Più in generale.....

Master Theorem

facoltativo

Per forme ancora piú generali del Teorema, che permettono la risoluzione di equazioni di ricorrenza del tipo generale

$$T(n) = aT(n/b) + f(n)$$

sussiste il seguente risultato

1.Se $f(n) = O(n^{\log_b a - \epsilon})$, per qualche $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$

2. Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \log n)$

3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$, per qualche $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e n sufficientemente grande, allora $T(n) = \Theta(f(n))$

Nota: Vale anche con parte intera inferiore o superiore di n/b
La soluzione T(n) è in notazione Θ

Applicazioni del Master Theorem

$$T(n) = aT(n/b) + f(n)$$

1.Se $f(n) = O(n^{\log_b a - \epsilon})$, per qualche $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a}$

2. Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \log n)$

3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$, per qualche $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e n sufficientemente grande, allora $T(n) = \Theta(f(n))$

Confrontare f(n) con $n^{\log_b a}$: qual è «più grande»?
Il più grande (asintoticamente e *polinomialmente*) vince!

Esempio 1: T(n)= 2 T(n/2)+ log n: a = b = 2, f(n)= log n vs $n^{\log_b a} = n^{\log_2 2} = n^1$
f(n)=O($n^{1-\varepsilon}$) per ε=1/2, quindi T(n)= Θ(n)

Esempio 2: T(n)= 2 T(n/2)+ n: a=b=2, f(n)= n vs $n^{\log_b a} = n^{\log_2 2} = n^1$
f(n)= Θ(n), quindi T(n)= Θ(n log n)

Esempio 3: T(n)= 2 T(n/2)+ n^3: a=b=2, f(n)= n^3 vs $n^{\log_b a} = n^{\log_2 2} = n^1$
f(n)= Ω($n^{1+\varepsilon}$) e inoltre 2(n/2) $^3 \leq$ cn ^{3} per c= ¼<1
quindi T(n)= Θ(n^3)

Caso di non applicabilità

$T(n) = 2T(n/2) + n \log n$

$a = b = 2$,

$f(n) = n \log n$ vs $n^{\log_b a} = n^{\log_2 2} = n^1$

$f(n) = n \log n = \Omega(n^1)$, ma non esiste nessun ε per cui $f(n) = \Omega(n^{1+\varepsilon})$

Il Master Theorem non si applica a questa relazione di ricorrenza; bisogna applicare gli altri metodi

Esercizio: ricerca ternaria

- Progettare un algoritmo per la ricerca di un elemento key in un array ordinato A[1..n], basato sulla tecnica Divide-et-impera che nella prima fase divide l'array in 3 parti «uguali» (le 3 parti differiranno di al più 1 elemento).
- Scrivere la relazione di ricorrenza per il tempo di esecuzione dell'algoritmo proposto.
- *Risolvere la relazione di ricorrenza.*
- *Confrontare il tempo di esecuzione con quello della ricerca binaria.*

Svolto in aula, a meno della dimostrazione che l'algoritmo raggiunge sempre il caso base.
Come esercizio, potete completarlo e inserire la soluzione completa e corretta sulla piattaforma

Occorrenze consecutive di 2 (D&I) (dalla piattaforma)

Si scriva lo pseudo-codice di un algoritmo ricorsivo basato sulla tecnica Divide et Impera che prende in input un array di interi positivi e restituisce il massimo numero di occorrenze **consecutive** del numero ‘2’.

Ad esempio, se l'array contiene la sequenza <2 2 3 6 2 2 2 2 3 3> allora l'algoritmo restituisce 4.
Occorre specificare l'input e l'output dell’algoritmo.

Altri esempi

Sia $T(1) = 1$. Valutate

- $T(n) = 2T(n/2) + n^3$
- $T(n) = T(9n/10) + n$
- $T(n) = 16T(n/4) + n^2$
- $T(n) = 7T(n/3) + n^2$
- $T(n) = 7T(n/2) + n^2$
- $T(n) = 2T(n/3) + \sqrt{n}$
- $T(n) = T(n-1) + n$
- $T(n) = T(\sqrt{n}) + 1$

Alcuni esercizi sulla tecnica del Divide et Impera.

1. a) Descrivere gli aspetti essenziali della tecnica Divide et Impera, utilizzando lo spazio designato.

   b) Descrivere ed analizzare un algoritmo **basato sulla tecnica Divide et Impera** che dato un array $A[1, ..., n]$ di interi ne restituisca il massimo.

2. Sia $V[1..n]$ un vettore ordinato di 0 e 1.

   Descrivere ed analizzare un algoritmo per determinare il numero di 0 presenti in $V[1..n]$ in tempo $O(\log n)$.

5. a) Fornire lo pseudocodice di un algoritmo ricorsivo che ordini un array $A[1..n]$ nel seguente modo: prima ordina ricorsivamente $A[1..n-1]$ e poi inserisce $A[n]$ nell'array ordinato $A[1..n-1]$.

   b) Analizzare la complessita' di tempo dell'algoritmo proposto al punto b).

6. Sia dato un vettore binario ordinato $A[1..n]$.

   (a) Progettare un algoritmo di complessita' $\Theta(n)$ nel caso peggiore, che conti il numero di occorrenze di 1 nel vettore A.

   (b) Progettare un algoritmo di complessita' $O(\log n)$, che conti il numero di occorrenze di 1 nel vettore A.

7. Sia dato un vettore ordinato $A[1..n]$ di interi distinti. Progettare un algoritmo che determini, in tempo $O(\log n)$, se esiste o meno un intero i tale che $A[i] = i$.

8. Descrivere ed analizzare un algoritmo basato sul paradigma *divide et impera* che dato un vettore *ordinato* $A[1..n]$ di interi strettamente positivi (cioe' per ogni $1 \leq i \leq n$, $A[i] \geq 1$), restituisca il numero di occorrenze di 1 nel vettore A. L'algoritmo deve avere complessita' di tempo $O(\log n)$.