INFORMATICA GENERALE I–Z

PRIMO COMPITO DI ESONERO – 14 APRILE 2011

C. MALVENUTO, D.A. GEWURZ

Si consideri il seguente problema: dato in input un vettore $A[1..n]$ di n interi, calcolare il numero di valori distinti che si ripetono almeno due volte nel vettore.

Per esempio, se $A = [2, 4, 6, 5, 4, 2, 7, 4, 4, 2, 3, 6, 1, 8]$, il numero di doppioni distinti è 3 (che sono gli interi 2, 4 e 6).

Sia dato il seguente algoritmo per risolvere ricorsivamente il problema dato con un approccio *divide et impera*:

Algoritmo 1 Doppioni (array A, interi i, f)

```
1: if i ≥ f then
2:     return 0
3: m ← ⌊(i + f)/2⌋
4: count ← Doppioni(A, i, m) + Doppioni(A, m + 1, f)
5: for k = i to m do
6:     c1 ← 0
7:     for j = i to m do
8:         if A[k] = A[j] then
9:             c1 ← c1 + 1
10:    c2 ← 0
11:    for j = m + 1 to f do
12:        if A[k] = A[j] then
13:            c2 ← c2 + 1
14:    if (c1 = 1 and c2 = 1) then
15:        count ← count + 1
16: return count
```

Esercizio 1. (2 punti) Con quali parametri i e f andrà data la chiamata iniziale dell'algoritmo per risolvere il problema iniziale?

La generica chiamata con i parametri i e f prende in esame solo il sottovettore composto dagli elementi $A[i]$, $A[i + 1]$, ..., $A[f]$. Quindi la chiamata iniziale, che deve esaminare l'intero vettore dato, deve essere data con i parametri $i = 1$ e $f = n$.

Esercizio 2. (6 punti) Scrivere una ricorrenza soddisfatta da $T(n)$, il tempo di esecuzione dell'algoritmo dato su un input di dimensione n, dove $n = f - i + 1$, specificando anche il caso base $n = 1$.

Nella riga 4., oltre ad alcune operazioni in tempo costante, si hanno due chiamate ricorsive all'algoritmo stesso: la prima è relativa a un parametro $\lfloor (i + f)/2 \rfloor - i + 1 \approx n/2$, e analogamente per la seconda. Quindi questa riga dà un contributo $2T(n/2)$.

Le righe dalla 5. alla 15. sono costituite da un ciclo "for" con $m - i + 1$ ($\approx n/2$) iterazioni. In ogni iterazione vengono eseguiti altri due cicli "for", ognuno costituito da circa $n/2$ iterazioni. Le altre operazioni contenute in queste righe richiedono ognuna un tempo costante. Quindi queste righe richiedono un tempo dell'ordine di $\Theta(n^2)$.

Infine, le righe 1.-3. e la 16. richiedono un tempo costante.

Quindi il tempo di esecuzione complessivo $T(n)$ soddisfa la ricorrenza

$$T(n) = 2T(n/2) + \Theta(n^2).$$

Nel caso base $n = 1$, quando cioè $i = f$, vengono eseguite solo le prime due righe, in un tempo $\Theta(1)$.

Esercizio 3. (4 punti) Valutare la complessità asintotica dell'algoritmo, risolvendo la ricorrenza con il teorema fondamentale.

Vediamo se la ricorrenza trovata, $T(n) = 2T(n/2) + \Theta(n^2)$, rientra in uno dei tre casi contemplati dal teorema fondamentale. Usando la notazione del teorema abbiamo $a = b = 2$, e quindi dobbiamo confrontare $n^{\log_b a} = n$ con $f(n) = n^2$. Qui si ha $f(n) = \Omega(n^{1+\varepsilon})$, per esempio prendendo $\varepsilon = 1/2$, e quindi per vedere se rientriamo nel caso 3 del teorema dobbiamo ancora verificare se vale la condizione di linearità. Deve cioè esistere una costante $c < 1$ tale che si abbia, definitivamente, $af(n/b) \leq cf(n)$ cioè, nel nostro caso, $2n^2/4 \leq cn^2$. Questa disuguaglianza è soddisfatta per qualsiasi c tale che $1/2 \leq c < 1$, e quindi possiamo applicare il teorema.

Ne segue che $T(n) = \Theta(f(n)) = \Theta(n^2)$.

Esercizio 4. (6 punti) Valutare la complessità asintotica dell'algoritmo, risolvendo la ricorrenza col metodo iterativo (si supponga per semplicità che n sia una potenza esatta).

Consideriamo di nuovo la ricorrenza trovata, che scriviamo nella forma $T(n) = 2T(n/2)+n^2$ (ignorando per comodità le costanti contenute in $\Theta(n^2)$) e inseriamo, al posto di $T(n/2)$, l'espressione data dalla ricorrenza stessa, cioè $2T(n/2^2) + (n/2)^2$. Otteniamo quindi:

$$\begin{aligned} T(n) &= 2\left(2T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2\right) + n^2 \\ &= 2^2 T\left(\frac{n}{2^2}\right) + \left(1 + \frac{1}{2}\right) n^2. \end{aligned}$$

Iterando nuovamente otteniamo

$$\begin{aligned} T(n) &= 2^2\left(2T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right) + \left(1 + \frac{1}{2}\right) n^2 \\ &= 2^3 T\left(\frac{n}{2^3}\right) + \left(1 + \frac{1}{2} + \frac{1}{2^2}\right) n^2. \end{aligned}$$

La k-esima iterazione darà quindi:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \left(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{k-1}}\right) n^2.$$

Dato che il caso base si ha per $n = 1$, ci interessa il valore di k per il quale si ottiene $n/2^k = 1$, cioè $k = \log_2 n$. Si ha quindi:

$$\begin{aligned} T(n) &= 2^{\log_2 n} T(1) + \left(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{\log_2 n - 1}}\right) n^2 \\ &= n + 2\left(1 - \frac{1}{n}\right) n^2 = \Theta(n^2), \end{aligned}$$

ricordando che si ha $\sum_{i=0}^{m} \alpha^i = (1 - \alpha^{m+1})/(1 - \alpha)$.

Volendo, è possibile studiare la somma $\sum_{i=0}^{\log_2 n-1} 2^{-i}$ maggiorandola con la serie geometrica $\sum_{i=0}^{\infty} 2^{-i} = 2$ e osservando che, nonostante questa maggiorazione sia stretta, ciò non cambia le nostre stime asintotiche perché $1 \leq \sum_{i=0}^{\log_2 n-1} 2^{-i} < \sum_{i=0}^{\infty} 2^{-i} = 2$ e quindi la somma originaria è in ogni caso $\Theta(1)$.

Esercizio 5. Commentare le linee dello pseudocodice dell'algoritmo Doppioni, dando una spiegazione del suo funzionamento. In particolare:

- (2 punti) Che cosa fanno le righe da 1. e 2.?
- (2 punti) Che cosa fanno le righe da 3. e 4.?
- (6 punti) Che cosa fanno le righe da 5. a 15.?
- (2 punti) L'algoritmo risolve il problema? (Motivare la risposta)

- Le righe 1. e 2. identificano il caso base per la ricorsione che costituisce il seguito dell'algoritmo: se $i \geq f$, cioè se il sottovettore passato all'algoritmo ha lunghezza minore o uguale a uno, il valore restituito dall'algoritmo è 0.

  Attenzione: non è del tutto corretto dire che queste righe servono a prevenire "errori" nell'inserimento dei dati (non c'è niente di sbagliato in un vettore di lunghezza 1), e tantomeno dire che se è verificata la condizione dell'*if*, l'algoritmo non parte (l'algoritmo è già partito, e probabilmente quando è arrivato a questo punto ha già svolto numerosi cicli di ricorsione), o simili.
- La riga 3. calcola la media aritmetica m tra i e f, che servirà per identificare l'indice dell'elemento (circa) a metà del sottovettore $A[i..f]$.

  La riga 4. imposta la suddivisione del problema originario in due sottoproblemi di dimensioni inferiori, richiamando ricorsivamente l'algoritmo su due sottovettori del vettore dato, e sommando gli output relativi. Nell'analisi classica di questi tipi di algoritmi si considera una fase "divide", una fase "impera" e una fase "combina": queste righe si occupano del "divide" e del "combina".
- Queste righe (insieme alle prime due per quanto riguarda il caso base) svolgono la parte "impera" dell'algoritmo e quindi buona parte dell'elaborazione vera e propria dei sottoproblemi.

  In particolare, vengono passati in rassegna tutti gli elementi della prima metà del sottovettore (grazie al ciclo *for* della riga 5.). Ognuno di essi viene dapprima confrontato con ogni elemento della stessa prima metà (con il ciclo definito nella riga 7.) e si contano i casi in cui i due elementi coincidono (con il confronto della riga 8., usando il contatore inizializzato nella riga 6. e incrementato nella riga 9.). Dopodiché, lo stesso elemento della prima metà viene confrontato con ogni elemento della seconda metà del sottovettore (con il ciclo della riga 11.), di nuovo contando le coincidenze (righe 10., 12. e 13.).

  A questo punto, se un elemento compare esattamente una volta nella prima metà ed esattamente una volta nella seconda (condizione verificata nella riga 14.), allora viene incrementata la varibile $count$ (riga 15.). L'idea è che così si contano i doppioni non contati nella riga 4.: infatti (ammesso che l'algoritmo, complessivamente, funzioni) la riga 4. conta i numeri ripetuti all'interno di una delle metà del sottovettore, mentre questi cicli identificano i doppioni "a cavallo" delle due metà (se un elemento compare più di una volta nella stessa metà era già stato contato, e quindi ora interessano solo quelli che compaiono esattamente una volta in ogni metà).

- No, l'algoritmo non risolve il problema.

  Il modo più semplice per mostrarlo è trovare un controesempio. Se applichiamo l'algoritmo al vettore $[3, 3, 3, 3]$, il risultato è 2, ma la risposta corretta è ovviamente 1. (Anche il vettore A dato nel testo del problema funziona come controesempio.)

  Il problema è nel fatto che, per come è impostato il meccanismo della suddivisione del problema in sottoproblemi, se uno stesso elemento del vettore è un doppione sia all'interno della prima metà del vettore sia all'interno della seconda, viene contato separatamente entrambe le volte, e quindi il risultato finale è troppo alto. Nel caso di $[3, 3, 3, 3]$, l'algoritmo identifica sia i primi due "3" come un doppione, sia gli ultimi due.

Esercizio 6. (Esercizio facoltativo: 5 punti) Progettare un algoritmo iterativo che risolva il problema della ricerca dei doppioni in un vettore. Descrivere il procedimento a parole (o con uno pseudocodice).

Ci sono ovviamente innumerevoli modi possibili.

Una possibilità consiste nell'impostare due cicli annidati per considerare tutte le coppie di elementi di indici distinti (cioè un ciclo per i che va da 1 a $n - 1$, e uno per j che va da $i + 1$ a n, considerando quindi, a ogni iterazione, la coppia $(A[i], A[j])$). Se contassimo solo quante di queste coppie sono costituite da elementi uguali, staremmo contando tutte le coincidenze, e non solo il numero di elementi distinti che hanno un doppione. Quindi servirà una struttura ausiliare per tenere conto dei doppioni già contati. Si può costruire via via una lista dei doppioni identificati, e a ogni nuovo doppione, verificare se è già presente nella lista. Oppure, una volta identificato un doppione, si possono sostituire tutte le sue successive occorrenze nel vettore A con un simbolo non presente nel vettore (così non verrà contato di nuovo).

Un metodo di questo tipo ha un tempo di esecuzione (nel caso peggiore) tra n^2 e n^3, essendo costituito da due cicli di lunghezza circa n annidati, a cui bisogna aggiungere le verifiche per non contare più volte ogni doppione.

In alternativa, è possibile iniziare riordinando il vettore dato (per esempio con un algoritmo di ordinamento con un tempo dell'ordine di $n \log n$). Nel vettore ordinato gli eventuali doppioni saranno consecutivi, e quindi basta leggerlo una volta consecutivamente, aggiornando via via un contatore che tiene conto di quante volte di seguito è apparso uno stesso elemento; quando si passa a un elemento diverso, se il numero di occorrenze consecutive è maggiore di 1, si aggiorna un'altra variabile che tiene conto del numero di doppioni distinti. Ancor più semplicemente, basta cercare nel vettore riordinato i punti in cui due elementi consecutivi uguali sono seguiti da uno differente. La lettura consecutiva richiede un tempo dell'ordine di n, e quindi un algoritmo di questo tipo, se ben implementato, dovrebbe avere un tempo complessivo dell'ordine di $n \log n$.

Ecco una versione in pseudocodice di quest'ultimo algoritmo:

Algoritmo 2 DoppioniOrd (array A)

```
count ← 0
Riordina(A)                          ▷ Un qualsiasi algoritmo che ordina A in tempo n log n
for i = 2 to n − 1 do
    if A[i] = A[i − 1] e A[i] ≠ A[i + 1] then
        count ← count + 1
if A[n] = A[n − 1] then
    count ← count + 1
return count
```