Lettura di Fine Corso: Una perla di Programmazione

Ivano Salvo
Sapienza Università di Roma

Anno Accademico 2011-12

Ora vi chiedo di obliarmi e di cercar voi;
e soltanto allorchè m'avrete rinnegato io tornerò fra di voi.
In verità miei fratelli, con altri occhi cercherò
quelli che ho smarrito; vi amerò d'altro amore.
(Zarathustra, *Della Virtù Donatrice*)

Presentazione

La presente lettura mostra *una perla di programmazione*: alcune riviste scientifiche di informatica a volte pubblicano, accanto ai risultati di ricerca scientifica, una rubrica in cui vengono presentate le cosidette "programming pearls", cioè soluzioni brillanti, spesso a semplici problemi, che brillano per efficienza o eleganza.

Alcune di queste perle sono state per esempio raccolte da John Bentley in un paio di celebri libri *Programming Pearls* e *More Programming Pearls* (pubblicati da Addison Wesley). Ci sono anche libri più specifici come *Pearls of Functional Algorithms Design* di Richard Bird (Cambridge University Press) che presenta perle di programmazione funzionale.

All'inizio del corso, pensavo che ci sarebbe stato il tempo per dedicare un paio di lezioni all'illustrazione di 2 o 3 di queste perle di programmazione. Purtroppo, benchè qua e là dell'ottimo codice sia stato mostrato a lezione, la dura realtà è stata che c'era a malapena il tempo di illustrarvi il programma "istituzionale".

Quindi, per gli studenti interessati, ecco un'unica perla di programmazione, a cui sono personalmente legato, in quanto era il contenuto di una lezione del prof. Edsgar W. Dijkstra a cui ebbi la fortuna di assistere da studente di dottorato. Con C. A. R. Hoare, Dijkstra sviluppò la teoria delle asserzioni logiche e numerosissimi sono i suoi contributi alla teoria e alla pratica dei linguaggi di programmazione e degli algoritmi.

Il prof. Dijkstra prendeva un lucido bianco, lo poneva sul proiettore, scriveva vicino al lembo inferiore l'asserzione finale che il programma avrebbe dovuto soddisfare e procedeva a ritroso scrivendo comandi, fintantochè sul lucido non sarebbe apparsa l'asserzione iniziale (cioè le precondizioni) del programma. Il prof. Dijkstra

non amava particolarmente i calcolatori (era solito ripetere: "il mio lavoro riguarda i calcolatori, come il lavoro dell'astronomo riguarda i telescopi") e nell'epoca in cui i più ormai erano già dediti agli effetti speciali delle presentazioni in PowerPoint, continuava ad essere un instancabile ammanuense. Ovviamente, non riuscirò a riprodurre in queste brevi note la magia di cotanta maestria.

Chiudo con una citazione di "seconda mano": John Bentley, in un'altra raccolta di perle di programmazione, *Beatiful Code* (O'Reilly, 2007), apre il suo pezzo dedicato a varie implementazioni di *quickSort*, con una citazione da Antoine de Saint-Exupery (l'ingegnere-aviatore autore del *Piccolo Principe*):

> *"Chi progetta sa di aver raggiunto la perfezione non quando non ha più nulla da aggiungere ma quando non gli resta più niente da togliere".*

Augurandovi che questa massima vi guidi tanto nel ragionamento matematico, quanto nella pratica della programmazione, vi auguro un fruttuoso proseguimento degli studi.

⋆ Uguaglianza di due vettori a meno di shift

Consideriamo un problema che generalizza il problema dell'uguaglianza di due vettori. Immaginiamo i due vettori come circolari (cioé l'elemento successivo a $A[n-1]$ è $A[0]$). Per semplificare le notazioni immaginiamo associata ad un array A una funzione *periodica* omonima A definita da:

$$A(i) = A[i \bmod n]$$

dove mod è il resto della divisione intera. Il nostro obiettivo sarà verificare se per due vettori è verificata la seguente proprietà:

$$eq \equiv \exists i. \forall k. 0 \leq k < n. A(i+k) = B(k)$$

Osserviamo che questa proprietà può essere riscritta nella seguente:

$$eq \equiv \exists i, j. \forall k. 0 \leq k < n. A(i+k) = B(j+k)$$

che ha il vantaggio di far giocare un ruolo simmetrico ai due vettori A e B. Indichiamo, sempre per comodità di notazione, con SA_i (rispettivamente SB_i) la sequenza ottenuta leggendo circolarmente il vettore A (risp. B) a partire dall'indice i, cioè la sequenza:

$$A[i]\ A[i+1] \ldots A[n-1]\ A[0]\ A[1] \ldots\ A[i-1]$$

In questo modo il nostro problema può essere riformulato nello stabilire la verità o meno dell'asserzione:

$$\exists i, j. SA_i = SB_j \tag{1}$$

Ragionando esattamente come nel caso di uguaglianza dei due vettori, tenendo presente solo la natura "circolare" del problema in questione, abbiamo che il seguente ciclo, se termina con $h = n$, stabilisce che i due vettori sono uguali, a meno di shift, e che i, j sono gli indici necessari a provare la proprietà esistenziale espressa da (1):

```
while (h<n && A[(i+h) % n] == B[(j+h) % n]) h++;
/* INV: forall k. 0<=k<h. A(i+k) = B(j+k) */
/* se h=n allora SA.i=SB.j
```

Tuttavia, se da questo ciclo si esce perché, per un qualche $h < n$, $A(i+h) \neq B(j+h)$, sorge la questione se sia il caso di cercare una nuova coppia i, j o se abbiamo finito, potendo concludere che i due vettori sono diversi. L'altra questione è legata alla domanda se sia possibile recuperare in parte il lavoro svolto, evitando di ricominciare daccapo con una nuova coppia i, j ed eventualmente riispezionare inutilmente sottovettori già analizzati. Ora è chiaro che fissando uno dei due indici, ad esempio i ad un valore arbitrario (ad esempio 0) e testando tutte le uguaglianze:

$$SA_0 = SB_j \quad (j = 0, \ldots n-1)$$

otteniamo un algoritmo che risolve questo problema. La complessità peggiore sarà nell'ordine di n^2 (provate ad esempio a vedere quanti confronti sono necessari nel caso in cui $SA_0 = \underbrace{000\ldots0}_{n-1}1$ e $SB_0 = \underbrace{000\ldots0}_{n-2}11$). La domanda quindi è se, scoprendo che $SA_i \neq SB_j$, sia possibile recuperare parte del lavoro svolto e scegliere in modo più intelligente una nuova coppia di indici i, j. La cosa è possibile solo se stabiliamo una certo criterio con cui esplorare l'insieme delle coppie di sequenze SA_i e SB_j. L'osservazione chiave della soluzione che sarà proposta è che l'insieme delle sequenze può essere ordinato, ad esempio usando l'ordine *lessicografico*[1] (quello del dizionario per intenderci). Inoltre se esiste una coppia di indici i, j tale che $SA_i = SB_j$ anche gli insiemi $\{SA_i \mid i = 0, \ldots n-1\}$ e $\{SB_j \mid i = 0, \ldots n-1\}$ sono uguali ed inoltre hanno lo stesso elemento *massimo* nell'ordine lessicografico. Chiamiamo:

$$\begin{aligned} AA &= max_{0 \leq i \leq n-1}\ SA_i \\ BB &= max_{0 \leq i \leq n-1}\ SB_i \end{aligned}$$

Quindi l'idea diventa quella di esplorare l'insieme di sequenze sempre cercando tra quelle *maggiori* nell'ordine lessicografico. Questo è possibile scomponendo la condizione di uscita dal ciclo $A(i+h) \neq B(j+h)$ nei suoi due sottocasi, e cioè $A(i+h) < B(j+h)$ e $A(i+h) > B(j+h)$. Nel primo caso avrò che $SA_i < SB_j$, mentre nel secondo che $SA_i > SB_j$. Concentriamoci sul primo caso (l'altro evidentemente è simmetrico). Ho che, per definizione di massimo, $SB_j \leq BB$. Ma avendo trovato h uguaglianze del tipo $A(i+k) = B(j+k)$, per $0 \leq k < h$, possiamo dire che tutte le sequenze $SA_{i+k} < SB_{j+k} \leq BB$. E a poco servirebbe quindi aggiornare i incrementandolo di 1. Possiamo direttamente occuparci di cominciare la nuova ricerca dall'elemento seguente a $i+h$, cioè $i+h+1$. L'idea in buona sostanza è quella di cercare tra tutte le stringhe SA_i (e simmetricamente SB_j) finché sappiamo

[1] l'ordine lessicografico tra due sequenze dipende dall'ordine del primo elemento su cui differiscono.

che $SA_k < BB$ $(0 \leq k < i)$ (e simmetricamente $SB_k < AA$ $(0 \leq k < j)$). Quindi gli invarianti della ricerca saranno:

$$\begin{aligned} P[i,j,h] &\equiv \forall k.0 \leq k < h.A(i+k) = B(j+k) \\ QA[i] &\equiv \forall k.0 \leq k < i.SA_k < BB \\ QB[j] &\equiv \forall k.0 \leq k < j.SB_k < AA \end{aligned}$$

Se ho verificato la proprietà $P[i,j,n]$, allora significa che ho trovato due indici i,j tale che $SA_i = SB_j$. Viceversa se sono arrivato a verificare la proprietà $QA[n]$ significa che tutte le sequenze SA_i sono strettamente minori di BB e quindi i due vettori sono diversi. Analoga conclusione posso trarre dalla verifica di $QB[n]$.

```
int equalShift(int A[], int B[], int n) {
/* PREC: n>=0, n lunghezza di A e B
 * POST: ritorno 1 se esistono 0<=i,j<n tale che vale
 * P[i,j,n] = forall k, 0<=k<n, A(i+k)=B(j+k)
 */
   int i = 0; int j = 0; int h = 0;
   while (i<n && j<n && h<n) {
   /* INV: P[i,j,h] & QA[i] & QB[j]
    * TERM: 3n - (i+j+h)
    */
      if (A[(i+h) % n] == B[(j+h) % n]) h++
         /* continuo a vedere se SA.i = SB.j */
      else if (A[(i+h) % n] < B[(j+h) % n]) {i=i+h+1; h=0}
         /* scopro che SA.i, SA.i+1, ..., SA.i+h sono minori di BB
          * (quindi vale QA[i+h])
          */
      else {j=j+h+1; h=0}
         /* scopro che SB.j, SB.j+1, ..., SB.j+h sono minori di AA
          * (quindi vale QB[i+h])
          */
   }
   eq = (n<h);
   return eq;
}
```

Siccome la guardia implica che tutte e tre le variabili `i, j, h` assumeranno valori minori di n durante il ciclo, osserviamo che la funzione $t = 3n - (i + j + h)$ è positiva durante l'esecuzione del ciclo. La terminazione si dimostra semplicemente osservando che la somma $i + j + h$ aumenta *almeno* di uno ad ogni iterazione. In questo caso, la funzione di terminazione ci dà anche informazione sulla complessità del programma, in quanto ci dice che il ciclo verrà eseguito al più $3n$ volte. Un eccellente progresso rispetto alla complessità n^2 del programma ingenuo!