Algoritmi ricorsivi in Pascal

per il corso di

Informatica Generale — Scienze Ambientali

Prof. Giancarlo Mauri
Appunti scritti dal Dott. Alberto Leporati

Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)
Università degli Studi di Milano - Bicocca
Via Bicocca degli Arcimboldi 8, 20126 Milano

Attenzione: nell'esercizio riguardante il conteggio di 1 in un vettore di interi, mi sono dimenticato di gestire correttamente il caso in cui l'indice n passato alla funzione `ContaUni` è maggiore del numero N di elementi contenuti nel vettore. Ripeto quindi l'esercizio, con una breve spiegazione del funzionamento dell'algoritmo.

Conteggio del numero di 1 in un vettore di interi

Problema: dato un vettore A, contenente N numeri interi, scrivere una funzione ricorsiva che restituisca al chiamante il numero di 1 presenti nel vettore.

Un possibile approccio (sicuramente non l'unico) consiste nel considerare il vettore A come l'implementazione di una sequenza finita di interi. Allora, il numero di 1 contenuti nella sequenza è dato da:

- 1 più il numero di 1 contenuti nella sottosequenza ottenuta scartando il primo elemento della sequenza di partenza, se tale primo elemento è un 1, oppure semplicemente
- il numero di 1 contenuti nella sottosequenza ottenuta scartando il primo elemento della sequenza di partenza, se questo è diverso da 1.

In altre parole, detto $u_{i,N}$ il numero di 1 contenuti nel vettore A tra gli elementi $A[i]$, $A[i+1], \ldots, A[N]$ (dove naturalmente $i \leq N$), abbiamo che vale la seguente formula:

$$u_{1,N} = \begin{cases} 1 + u_{2,N} & \text{se } A[1] = 1 \\ u_{2,N} & \text{se } A[1] \neq 1 \end{cases} \tag{1}$$

Una volta che abbiamo messo in relazione il numero di 1 in una sequenza di N elementi con il numero di 1 contenuti in una sua sottosequenza di $N-1$ elementi, possiamo applicare

lo stesso metodo (ricorsivamente, per l'appunto) per calcolare quest'ultimo valore, in modo da sostituirlo al posto di $u_{2,N}$ in (1) e calcolare quindi il valore di $u_{1,N}$. Ad ogni applicazione ricorsiva del metodo appena descritto, questo viene utilizzato su una sequenza che contiene un elemento in meno rispetto a quella considerata al passo precedente; per definire un criterio che ci consenta di portare a termine il calcolo (altrimenti il metodo verrebbe applicato ricorsivamente un numero infinito di volte), possiamo osservare che siamo in grado di calcolare immediatamente il numero di 1 in una sequenza costituita da un singolo elemento, nel modo seguente:

$$u_{N,N} = \begin{cases} 1 & \text{se } A[N] = 1 \\ 0 & \text{se } A[N] \neq 1 \end{cases}$$

Volendo, possiamo anche adottare un atteggiamento più "estremo" (ma perfettamente corretto dal punto di vista matematico), dicendo che il numero di 1 presenti in una sequenza vuota (cioè costituita da 0 elementi) è 0:

$$u_{N+1,N} = 0$$

Dato che in Pascal non è possibile scrivere una funzione che accetti come argomento una sequenza di interi di lunghezza variabile (che venga cioè invocata una volta passandole una sequenza di 5 interi, un'altra volta con 4 interi, e così via)[1], possiamo passare alla funzione l'intero vettore A, insieme a un valore intero che indichi da quale indice in poi vanno considerati gli elementi del vettore. Così, se passiamo alla funzione il vettore A e il valore 5, essa saprà che deve considerare solamente gli elementi $A[5], A[6], \ldots, A[N]$ (supponendo, ovviamente, che sia $N \geq 5$).

Osserviamo poi che il passaggio di tutto il vettore A alla funzione come argomento per valore comporta uno spreco ingiustificato nell'utilizzo della memoria a pila che contiene le istanze delle variabili nel corso dell'esecuzione del programma; possiamo pertanto passare il vettore per indirizzo, oppure non passarlo affatto alla funzione e utilizzare una sola istanza (globale), del vettore. Per semplicità adottiamo quest'ultima soluzione; abbiamo pertanto la seguente codifica in Pascal:

```
Program ConteggioUni (input, output);
Const N = 100;
Var A: array [1..N] of integer;
    i: integer;

Function ContaUni (n: integer): integer;
begin
   if n <= N
      then if A[n] = 1
```

[1] In realtà, una situazione simile sarebbe implementabile in Pascal facendo uso di una *lista concatenata* di interi, la cui definizione, utilizzo e gestione esulano dai contenuti di questo corso; inoltre, l'onere derivante dalla gestione di una lista concatenata per un'applicazione così semplice risulterebbe in ogni caso eccessivo.

```
                then ContaUni := 1 + ContaUni(n + 1)
                else ContaUni := ContaUni(n + 1)
        else ContaUni := 0
end;

begin {corpo del programma}
   write('Inserisci', N, ' elementi interi: ');
   for i := 1 to N do
      readln(A[i]);
   writeln('Il numero di 1 inseriti e'': ', ContaUni(1))
end.
```

La parte che mancava a lezione è l'ultima clausola `else` della funzione (`else ContaUni := 0`); senza di questa, è vero che la funzione termina comunque, ma il valore da essa restituito al programma principale è casuale (e quindi sbagliato), in quanto la chiamata `ContaUni(N+1)` restituisce un valore assolutamente imprevedibile (anziché 0, come dovrebbe), e questo valore viene poi utilizzato per calcolare il valore da restituire per le chiamate `ContaUni(N)`, `ContaUni(N-1)`, ..., `ContaUni(1)`.

Torre di Hanoi

Problema: Scrivere un programma che, letto da tastiera un numero intero positivo n, scriva sullo schermo la sequenza di mosse necessarie per risolvere il problema della Torre di Hanoi con n dischi.

Ricordiamo che il problema della Torre di Hanoi è il seguente: data una tavoletta sulla quale sono infissi tre pioli (che indichiamo con le lettere maiuscole A, B e C), e dati n dischi, tutti di diametro diverso, forati al centro, impilati nel piolo A in ordine di diametro, con il disco più grande in basso e il disco più piccolo in alto, si vuole spostare gli n dischi dal piolo A al piolo C rispettando le seguenti regole:

- ad ogni mossa è possibile spostare un unico disco, da un piolo all'altro; in particolare, questo significa che non è possibile appoggiare i dischi sul tavolo, e non è possibile spostare un disco che abbia altri dischi sopra di esso;
- non è possibile mettere un disco sopra a un altro disco di diametro minore.

Il problema originale riguarda 64 dischi; la leggenda dice che i monaci (buddhisti) siano impegnati da secoli a risolvere il problema, e che quando finiranno il mondo terminerà. Se così fosse, potremmo stare tranquilli: è stato calcolato che, se anche un computer potesse calcolare un milione di mosse al secondo, sarebbero comunque necessari diversi miliardi di anni per elencare tutte le mosse che risolvono il gioco.

Se si prova a risolvere il problema a mano utilizzando un numero relativamente basso di dischi (ad esempio 8) ci si rende presto conto che è facile confondersi e fare delle mosse che non portano a nulla. In questo caso, una formulazione ricorsiva è proprio quello che

ci serve, in quanto è molto semplice da formulare: per risolvere il gioco, basterà allora far girare il programma su un computer, ed eseguire la sequenza di mosse da esso generata.

La formulazione ricorsiva del problema è la seguente: per spostare n dischi dal piolo A al piolo C, posso anzitutto spostare $n-1$ dischi dal piolo A al piolo B, poi sposto in C il disco che è rimasto su A, e quindi sposto in C gli $n-1$ dischi che ho temporaneamente messo in B.

Per quanto possa sembrare incredibile, la formulazione di cui sopra funziona perfettamente, e mostra in modo molto chiaro la potenza della ricorsione.

Dall'analisi ricorsiva del problema osserviamo che, per spostare gli n dischi dal piolo A al piolo C abbiamo utilizzato il piolo B come "base d'appoggio". Più precisamente, quindi, le operazioni da compiere sono:

- spostare $n-1$ dischi dal piolo di partenza (A) al piolo d'appoggio (B);
- spostare il disco rimanente dal piolo di partenza (A) al piolo d'arrivo (C);
- spostare gli $n-1$ dischi dal piolo d'appoggio (B) al piolo d'arrivo (C).

Se ora osserviamo che la prima operazione non è altro che il problema della Torre di Hanoi per $n-1$ dischi, con i ruoli dei pioli B e C scambiati (si tratta, cioè, di spostare $n-1$ dischi dal piolo A al piolo B utilizzando il piolo C come base d'appoggio), mentre la terza operazione è ancora il problema della Torre di Hanoi per $n-1$ dischi, dove questa volta sono i pioli A e B ad essersi scambiati i ruoli (ovvero, si tratta di spostare $n-1$ dischi dal piolo B al piolo C, utilizzando il piolo A come base d'appoggio), siamo in grado di scrivere la procedura ricorsiva che risolve il problema. È sufficiente dichiarare la procedura con i seguenti parametri formali:

```
Procedure Sposta (n, da, a, appoggio:  integer);
```

dove `n` è il numero di dischi da spostare, `da` è il piolo di partenza (e assume i valori 1, 2 o 3 a seconda che il piolo di partenza sia il piolo A, il piolo B o il piolo C[2]), `a` è il piolo d'arrivo e `appoggio` è il piolo da usare come base d'appoggio. Lo scopo della procedura è quello di stampare sullo schermo la sequenza di mosse da effettuare per spostare n dischi dal piolo indicato nel parametro `da` al piolo indicato nel parametro `a`, utilizzando il piolo indicato nella variabile `appoggio` come base d'appoggio.

Osserviamo che, nel caso in cui si tratti di spostare un unico disco, possiamo risolvere il problema utilizzando la seguente procedura, che stampa semplicemente la mossa sullo schermo:

```
Procedure SpostaDisco (da, a: integer);
begin
   writeln('Mossa: ', da, ' -> ', a)
end;
```

[2] È anche possibile dichiarare i parametri formali `da`, `a` e `appoggio` di tipo `char`, e assegnare loro i valori `'A'`, `'B'`, e `'C'`; è solo questione di gusti.

Abbiamo quindi tutti gli elementi necessari a scrivere la procedura `Sposta`:

```
Procedure Sposta (n, da, a, appoggio: integer);
begin
   if n >= 1
      then begin
              Sposta (n - 1, da, appoggio, a);
              SpostaDisco (da, a);
              Sposta (n - 1, appoggio, a, da)
           end
end;
```

Per risolvere il problema con 8 dischi, sarà sufficiente richiamare — nel corpo del programma principale — la procedura nel modo seguente:

```
Sposta (8, 1, 3, 2);
```

Questa chiamata genererà le seguenti:

```
Sposta (7, 1, 2, 3);
SpostaDisco (1, 3);
Sposta (7, 2, 3, 1)
```

che, rispettivamente, spostano 7 dischi dal piolo A al piolo B (usando C come appoggio), stampano la mossa `1 -> 3` (corrispondente allo spostamento del disco più grosso dal piolo A al piolo C), e infine spostano 7 dischi dal piolo B al piolo C (usando A come appoggio). Ciascuna chiamata ricorsiva alla funzione `Sposta` viene risolta come se fosse un problema indipendente, più semplice di quello di partenza (in quanto riguarda un disco in meno), dove il ruolo dei pioli cambia secondo la necessità.

Concludiamo osservando che, poiché la procedura `Sposta` termina senza far nulla se $n \leq 0$, la chiamata:

```
Sposta (1, x, y, z);
```

è equivalente a:

```
SpostaDisco (x, y);
```

per qualsiasi valore di `x`, `y` e `z` (intere).

Sequenze Palindrome

Data una sequenza finita di simboli (che possono essere numeri, lettere dell'alfabeto o altro), si dice che tale sequenza è *palindroma* se rimane invariata leggendola al contrario. Ad esempio, la sequenza `abbcbba` è palindroma, mentre la sequenza `abcbba` non lo è. Ci poniamo allora il seguente:

Problema: scrivere una funzione ricorsiva che, dato un vettore A di N caratteri, restituisca il valore booleano `true` se tale vettore contiene una sequenza palindroma, e `false` altrimenti.

Possiamo risolvere il problema osservando che una sequenza di caratteri è palindroma se e solo se il primo carattere è uguale all'ultimo, il secondo è uguale al penultimo, e così via. Ragionando in maniera ricorsiva, questo significa che una sequenza di caratteri è palindroma se e solo se il primo carattere è uguale all'ultimo, e inoltre la sottosequenza ottenuta scartando il primo e l'ultimo carattere è essa stessa palindroma. Tale sottosequenza ha due caratteri in meno di quella di partenza; procedendo con le chiamate ricorsive, ci ritroveremo ad operare su sequenze formate da 0 o da 1 carattere (a seconda che la sequenza di partenza sia di lunghezza pari o dispari. Una sequenza costituita da un solo carattere è sempre palindroma, mentre possiamo imporre che una sequenza costituita da 0 caratteri (sequenza vuota) sia palindroma per definizione. Possiamo allora scrivere la funzione ricorsiva `Palindroma` come segue:

```
Function Palindroma (n, m: integer): boolean;
begin
   if m - n < 1
      then Palindroma := true
      else Palindroma := (A[n] = A[m]) AND Palindroma(n + 1, m - 1)
end;
```

Osserviamo che, come per l'esercizio che chiedeva di contare il numero di 1 in una sequenza di interi, anche in questo caso la funzione opera su un unico vettore (`A`) globale, anziché passare (per valore o per indirizzo) il vettore come parametro. I parametri formali `n` ed `m` contengono rispettivamente l'indice minimo e l'indice massimo del vettore da considerare; in altre parole, la funzione restituisce il valore `true` se la sottosequenza $A[n]$, $A[n+1]$, ..., $A[m]$ è palindroma, e restituisce il valore `false` in caso contrario. Questo significa che, dichiarato il vettore globale (`A`) come segue:

```
Program SequenzePalindrome (input, output);
Const N = 100;
Var A: array [1..N] of char;
...
```

la funzione `Palindroma` andrà invocata nel corpo del programma principale come segue:

```
Palindroma (1, N);
```

(chiaramente, il valore restituito dalla funzione andrà memorizzato in una variabile, oppure stampato sullo schermo, oppure utilizzato all'interno di una espressione booleana). Se N è pari, l'ultima chiamata ricorsiva sarà tale che $m = n - 1$ (e quindi $m - n = -1$); se N è dispari, l'ultima chiamata ricorsiva sarà tale che $m = n$ (e quindi $m - n = 0$).

Per concludere, osserviamo che l'uso di due parametri formali (`n` ed `m`) nella funzione ci consente di rendere semplice ed intuitiva la funzione stessa, ma non è affatto necessario: lo stesso compito può essere svolto utilizzando un unico parametro formale. Chiamando n tale parametro, ad ogni chiamata ricorsiva consideriamo la sottosequenza $A[n]$, $A[n+1]$, ..., $A[N-n+1]$, per valori crescenti di n. La funzione diventa allora:

```
Function Palindroma (n: integer): boolean;
begin
   if N + 1 < 2*n
      then Palindroma := true
      else Palindroma := (A[n] = A[N - n + 1]) AND Palindroma (n + 1)
end;
```