5.3 Ordinamenti

Un altro, fondamentale problema dell'informatica spesso collegato con la ricerca è quello che consiste nell'ordinare un vettore disponendo i suoi elementi in ordine crescente o decrescente. Esistono numerosi algoritmi che consentono di ordinare un array; uno dei più famosi è quello comunemente detto *bubblesort*. Osserviamo una sua versione in C che ordina un array in modo crescente. L'array è identificato da `vet` e ha *n* elementi:

```
for(j=0;j<n-1; j++)
       for(i=0; i<n-1; i++)
         if(vet[i]>vet[i+1])
           {aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux;}
```

Nel ciclo più interno gli elementi adiacenti vengono confrontati: se `vet[i]` risulta maggiore di `vet[i+1]` si effettua lo scambio tra i loro valori. Per quest'ultima operazione si ha la necessità di una variabile di appoggio, che nell'esempio è `aux`. Il ciclo si ripete finché tutti gli elementi sono stati confrontati, quindi fino a quando `i` è minore di `n-1`, perché il confronto viene fatto tra `vet[i]` e `vet[i+1]`.
Questa serie di confronti non è in generale sufficiente a ordinare l'array. La sicurezza dell'ordinamento è data dalla sua ripetizione per $n-1$ volte; nell'esempio ciò si ottiene con un `for` più esterno controllato dalla variabile `j` che varia da 0 a $n-1$ (Figura 5.1).

vet[0] = 9
vet[1] = 18
vet[2] = 7
vet[3] = 15
vet[4] = 21
vet[5] = 11

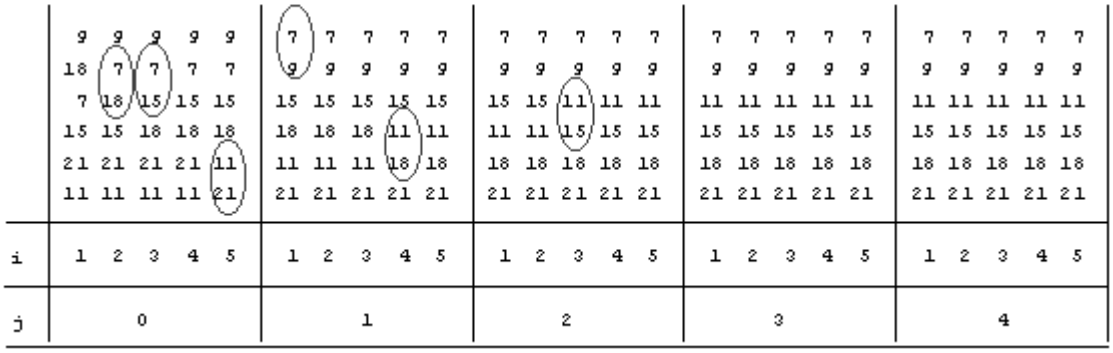

Figura 5.1 Esempio di ordinamento con l'algoritmo di bubblesort

In realtà il numero di volte per cui il ciclo interno va ripetuto dipende da quanto è disordinata la sequenza di valori iniziali. Per esempio, l'ordinamento di un array di partenza con valori 10, 12, 100, 50, 200, 315 ha bisogno di un solo scambio, che viene effettuato per `i=2` e `j=0`; dunque tutti i cicli successivi sono inutili. A questo proposito si provi a ricostruire i passaggi della Figura 5.1 con questi valori di partenza.
Si può dedurre che l'array è ordinato e cessare l'esecuzione delle iterazioni quando un intero ciclo interno non ha dato luogo ad alcuno scambio di valori tra `vet[i]` e `vet[i+1]`:

```
do {
       k = 0;
       for(i=0; i<n-1; i++)
         if(vet[i]>vet[i+1])
           {aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux; k=1;}
     }
     while(k==1);
```

Una prima ottimizzazione dell'algoritmo si ottiene interrompendo il ciclo esterno la prima volta che per un'intera iterazione del ciclo interno la clausola `if` non ha dato esito positivo.
Nel ciclo esterno la variabile `k` viene inizializzata a zero: se almeno un confronto del ciclo piccolo dà esito positivo, a `k` viene assegnato il valore uno. In pratica la variabile `k` è utilizzata come *flag* (bandiera): se il suo valore è 1 il ciclo deve essere ripetuto, altrimenti no.
Nel caso dell'array di partenza di Figura 5.1, l'adozione dell'ultimo algoritmo fa risparmiare un ciclo esterno (cinque cicli interni) rispetto al precedente. La prima volta che l'esecuzione del ciclo esterno non dà esito a scambi corrisponde al valore di `j` uguale a 3, `k` rimane a valore zero e le iterazioni hanno termine. Si provi con valori iniziali meno disordinati per verificare l'ulteriore guadagno in tempo d'esecuzione.

Osservando ancora una volta la Figura 5.1 si nota che a ogni incremento di `j`, variabile che controlla il ciclo esterno, almeno gli ultimi `j+1` elementi sono ordinati. Il fatto è valido in generale poiché il ciclo interno sposta di volta in volta l'elemento più pesante verso il basso. Dall'ultima osservazione possiamo ricavare un'ulteriore ottimizzazione dell'algoritmo:

```
do {
       k = 0;
       for(i=0; i<n-1; i++)
         if(vet[i]>vet[i+1])
           {aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux; k=1;}
       --n;
     }
```

```
    while(k==1);
```

In tale ottimizzazione, a ogni nuova ripetizione del ciclo esterno viene decrementato il valore limite del ciclo interno, in modo da diminuire di uno, di volta in volta, il numero di confronti effettuati. Ma è ancora possibile un'altra ottimizzazione:

```
p = n;
    do {
      k = 0;
      for(i=0; i<n-1; i++)
        if(vet[i]>vet[i+1]) {
          aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux;
           k = 1; p = i+1;
        }
    n = p;
    }
    while(k==1);
```

Il numero dei confronti effettuati dal ciclo interno si interrompe lì dove la volta precedente si è avuto l'ultimo scambio, come si osserva dal confronto tra le Figure 5.1 e 5.2.

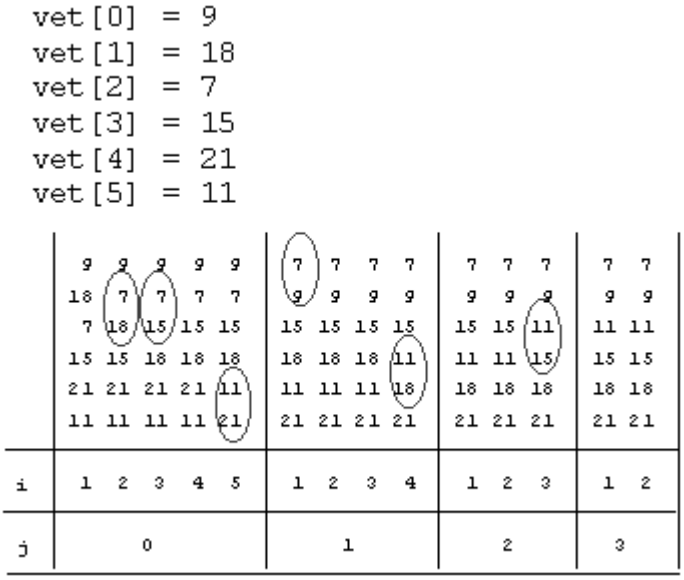

Figura 5.2 Esempio di ordinamento con l'algoritmo di *bubblesort* ottimizzato