

Ricerche, ordinamenti e fusioni

5.1 Introduzione

Questo capitolo ci permette di fare pratica di programmazione utilizzando gli strumenti del linguaggio introdotti finora. A una prima lettura possono essere saltati senza che ciò pregiudichi l'apprendimento degli argomenti seguenti, nel caso il lettore voglia continuare ad appropriarsi rapidamente delle possibilità offerte dal linguaggio.

È esperienza comune che nella gestione dei più svariati insiemi di dati (vettori o matrici, ma più in generale anche archivi cartacei, listini prezzi, voci di un'enciclopedia o addirittura semplici carte da gioco) sia spesso necessario: stabilire se un elemento è o no presente nell'insieme, ordinare l'insieme in un determinato modo (in genere in maniera crescente o decrescente), unire (fondere) due o più insiemi in un unico insieme evitando possibili duplicazioni. Queste tre attività, che in informatica vengono indicate rispettivamente con i termini di *ricerca*, *ordinamento* e *fusione* oppure con i loro equivalenti inglesi *search*, *sort* e *merge*, sono estremamente frequenti e svolgono un ruolo della massima importanza in tutti i possibili impieghi degli elaboratori. È stato per esempio stimato che l'esecuzione dei soli programmi di ordinamento rappresenti oltre il 30% del lavoro svolto dai computer. È quindi ovvio come sia della massima importanza disporre di programmi che svolgano questi compiti nel minor tempo possibile.

5.2 Ricerca completa

Un primo algoritmo per determinare se un valore è presente all'interno di un array, applicabile anche a sequenze non ordinate, è quello comunemente detto di *ricerca completa*, che opera una scansione sequenziale degli elementi del vettore confrontandoli con il valore ricercato. Nel momento in cui tale verifica dà esito positivo la scansione ha termine e viene restituito l'indice dell'elemento all'interno dell'array stesso.

Per determinare che il valore non è presente, il procedimento (Listato 5.1) deve controllare uno a uno tutti gli elementi fino all'ultimo, prima di poter sentenziare il fallimento della ricerca. L'array che conterrà la sequenza è `vet` formato da `MAX_ELE` elementi.

```
/* Ricerca sequenziale di un valore nel vettore */

#include <stdio.h>
#define MAX_ELE 1000 /* massimo numero di elementi */

main()
{
char vet[MAX_ELE];
int i, n;
char c;

/* Immissione lunghezza della sequenza */
do {
printf("\nNumero elementi: ");
scanf("%d", &n);
}
while(n<1 || n>MAX_ELE);

/* Immissione elementi della sequenza */
for(i=0; i<n; i++) {
printf("\nImmettere carattere n.%d: ",i);
scanf("%1s", &vet[i]);
}

printf("Elemento da ricercare: ");
scanf("%1s", &c);
```

```

/* Ricerca sequenziale */
i = 0;
while(c!=vet[i] && i<n-1) ++i;
if(c==vet[i])
    printf("\nElemento %c presente in posizione %d\n",c,i);
else
    printf("\nElemento non presente!\n");
}

```

Listato 5.1 Ricerca completa

Il programma presenta le solite fasi di richiesta e relativa immissione del numero degli elementi della sequenza e dei valori che la compongono. Successivamente l'utente inserisce il carattere da ricercare, che viene memorizzato nella variabile `c`. La ricerca parte dal primo elemento dell'array (quello con indice zero) e prosegue fintantoché il confronto fra `c` e `vet[i]` dà esito negativo e contemporaneamente `i` è minore di `n-1`:

```
while(c!=vet[i] && i<n-1) ++i;
```

Il corpo del ciclo è costituito dal semplice incremento di `i`. L'iterazione termina in tre casi:

1. `c` è uguale a `vet[i]` e `i` è minore di `n-1`;
2. `c` è diverso da `vet[i]` e `i` è uguale a `n-1`;
3. `c` è uguale a `vet[i]` e `i` è uguale a `n-1`.

In ogni caso `i` ha un valore minore o uguale a `n-1`, è dunque all'interno dei limiti di esistenza dell'array. L'`if` successivo determinerà se è terminato perché `c` è risultato essere uguale a `vet[i]`:

```
if(c==vet[i])
```

Esistono molte altre soluzioni al problema. Per esempio si potrebbe adottare un costrutto `while` ancora più sintetico, come il seguente:

```
while(c!=vet[i] && i++<n-1)
    ;
```

dove il corpo del ciclo non è esplicitato in quanto l'incremento di `i` avviene all'interno dell'espressione di controllo. Si noti però che, in questo caso, al termine delle iterazioni `i` ha un valore maggiorato di uno rispetto alla condizione che ha bloccato il ciclo, e di questo bisognerà tener conto nel prosieguo del programma.

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

```

	9 9 9 9 9	7 7 7 7 7	7 7 7 7 7	7 7 7 7 7	7 7 7 7 7
	18 7 7 7 7	9 9 9 9 9	9 9 9 9 9	9 9 9 9 9	9 9 9 9 9
	7 18 15 15 15	15 15 15 15 15	15 15 11 11 11	11 11 11 11 11	11 11 11 11 11
	15 15 18 18 18	18 18 18 11 11	11 11 15 15 15	15 15 15 15 15	15 15 15 15 15
	21 21 21 21 11	11 11 11 18 18	18 18 18 18 18	18 18 18 18 18	18 18 18 18 18
	11 11 11 11 21	21 21 21 21 21	21 21 21 21 21	21 21 21 21 21	21 21 21 21 21
i	1 2 3 4 5	1 2 3 4 5	1 2 3 4 5	1 2 3 4 5	1 2 3 4 5
j	0	1	2	3	4

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.

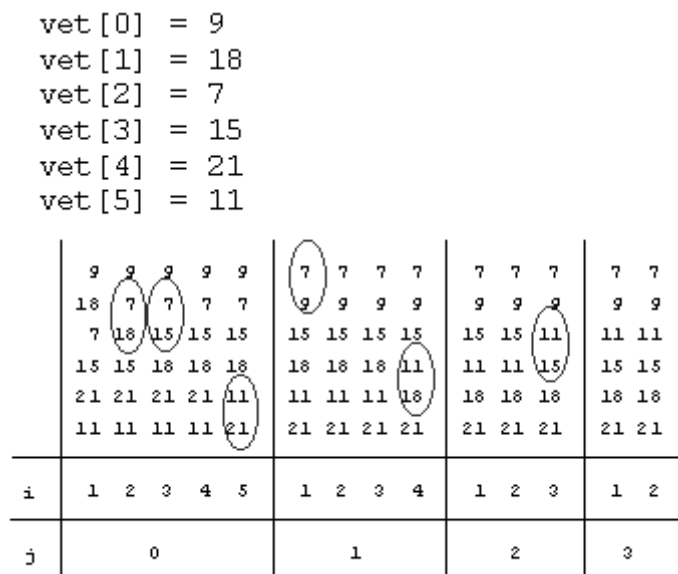


Figura 5.2 Esempio di ordinamento con l'algoritmo di *bubblesort* ottimizzato

5.4 Ricerca binaria

Quando l'array risulta ordinato la ricerca di un valore al suo interno può avvenire mediante criteri particolari, uno dei quali è la ricerca detta *binaria* o *dicotomica*.

```
/* Ricerca binaria */
#include <stdio.h>

main()
{
    char vet[6];          /* array contenente i caratteri immessi */
    int i,n,k,p;
    char aux;            /* variabile di appoggio per lo scambio */
    char ele;           /* elemento da ricercare */
    int basso, alto, pos; /* usati per la ricerca binaria */
```

```

/* Immissione caratteri */
n = 6;
for(i=0;i<=n-1; i++) {
    printf("vet %d° elemento: ", i+1);
    scanf("%1s", &vet[i]);
}

/* ordinamento ottimizzato */
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);

printf("\nElemento da ricercare: ");
scanf("%1s", &ele);

/* ricerca binaria */
n = 6;
alto = 0; basso = n-1; pos = -1;
do {
    i = (alto+basso)/2;
    if(vet[i]==ele) pos = i;
    else
        if(vet[i]<ele)
            alto = i+1;
        else
            basso = i-1;
}
while(alto<=basso && pos==-1);

if(pos != -1)
    printf("\nElemento %c presente in posizione %d\n",ele,pos);
else
    printf("\nElemento non presente! %d\n", pos);
}

```

Listato 5.2 Programma completo di immissione, ordinamento e ricerca

Nel Listato 5.2, dopo l'immissione dei valori del vettore, il loro ordinamento con bubblesort e l'accettazione dell'elemento da cercare, abbiamo i comandi della ricerca binaria vera e propria:

```

/* ricerca binaria */
alto = 0; basso = n-1; pos = -1;
do {
    i = (alto+basso)/2;
    if(vet[i]==ele) pos=i;
    else
        if(vet[i]<ele)
            alto = i+1;
        else

```

```

        basso = i-1;
    }
while(alto<=basso && pos==-1);

```

Si confronta il valore da ricercare, che è memorizzato nella variabile `ele`, con l'elemento intermedio dell'array. L'indice `i` di tale elemento lo si calcola sommando l'indice inferiore dell'array (0), memorizzato nella variabile `alto`, con quello superiore (`n-1`), memorizzato nella variabile `basso`, e dividendolo per due. Essendo l'array ordinato si possono presentare tre casi:

1. 1. `vet[i]` è uguale a `ele`, la ricerca è finita positivamente, si memorizza l'indice dell'elemento in `pos` e il ciclo di ricerca ha termine;
2. 2. `vet[i]` è minore di `ele`, la ricerca continua tra i valori maggiori di `vet[i]` che sono memorizzati negli elementi con indice compreso tra `i+1` e `basso`, per cui si assegna ad `alto` il valore `i+1`. Se non si sono già esaminati tutti gli elementi del vettore (`alto` non è minore o uguale a `basso`) la ricerca continua assegnando ancora una volta a `i` il valore $(basso+alto)/2$;
3. 3. `vet[i]` è maggiore di `ele`, la ricerca continua tra i valori minori di `vet[i]` che sono memorizzati negli elementi con indice compreso tra `alto` e `i-1`, per cui si assegna a `basso` il valore `i-1`. Se non si sono già esaminati tutti gli elementi del vettore (`alto` non è minore di `basso`) la ricerca continua assegnando ancora una volta a `i` il valore $(basso+alto)/2$.

Valore cercato `o(ele='o')`

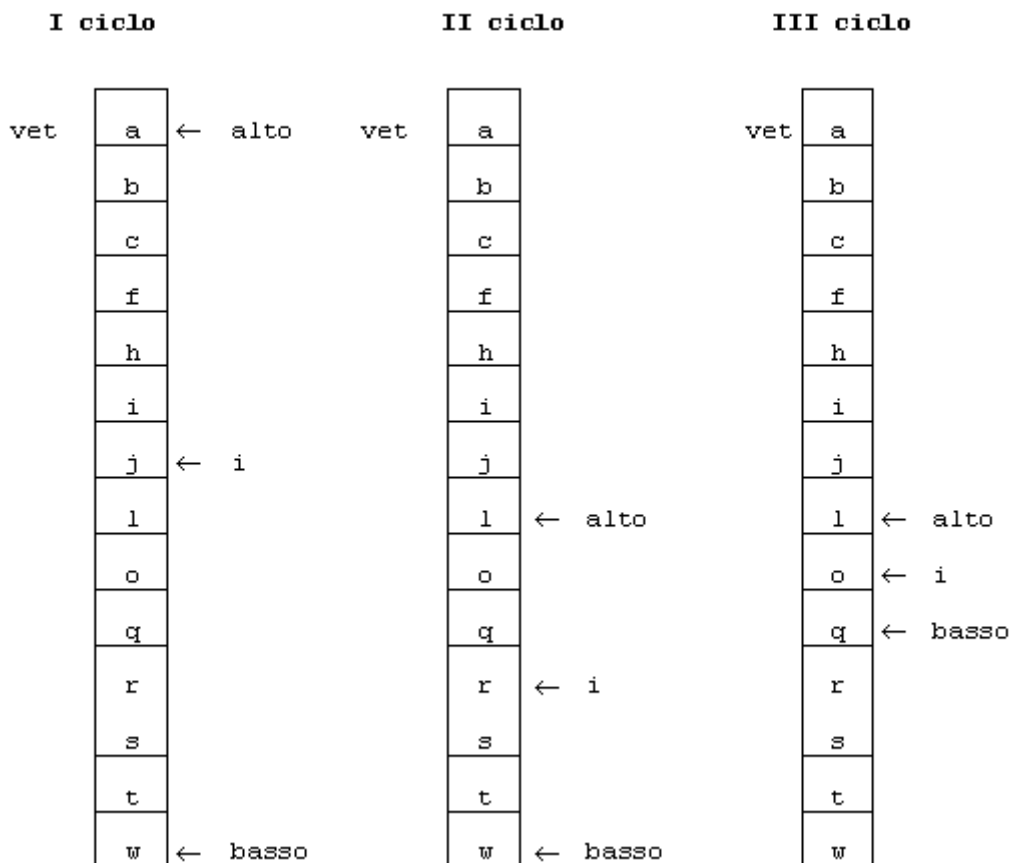


Figura 5.3 Esempio di ricerca binaria di `ele='o'`.

Nella Figura 5.3 si osserva il mutare dei valori di `alto`, `basso` e `i` fino al reperimento del valore desiderato ("o"). Il numero di cicli e corrispondenti confronti effettuati è risultato uguale a tre, mentre se avessimo utilizzato la ricerca sequenziale avremmo avuto nove iterazioni. La ricerca sequenziale esegue nel caso più fortunato – quello in cui l'elemento cercato è proprio il primo – un unico confronto; nel caso più sfortunato – quello in cui l'elemento cercato è invece l'ultimo – esegue n confronti. Si ha quindi che la ricerca sequenziale effettua in media $(n+1)/2$ confronti.

La ricerca binaria offre delle prestazioni indubbiamente migliori: al massimo esegue un numero di confronti pari al logaritmo in base due di n . Questo implica che nel caso in cui n sia uguale a 1000 per la ricerca sequenziale si hanno in media 500 confronti, per quella binaria al massimo 10. Poiché, come per l'ordinamento, il tempo impiegato per eseguire il programma è direttamente proporzionale al numero dei confronti effettuati, è chiaro come la ricerca binaria abbia tempi di risposta mediamente molto migliori della ricerca sequenziale.

Osserviamo, tuttavia, che mentre si può effettuare la ricerca sequenziale su qualsiasi vettore, per la ricerca binaria è necessario disporre di un vettore ordinato, così che non sempre risulta possibile applicare tale algoritmo.

5.5 Fusione

Un altro algoritmo interessante è quello che partendo da due array monodimensionali ordinati ne ricava un terzo, anch'esso ordinato. I due array possono essere di lunghezza qualsiasi e in generale non uguale. Il programma del Listato 5.3 richiede all'utente l'immissione della lunghezza di ognuna delle due sequenze e gli elementi che le compongono. Successivamente ordina le sequenze ed effettua la fusione (*merge*) di una nell'altra, memorizzando il risultato in un array a parte.

```
/* Fusione di due sequenze ordinate */
#include <stdio.h>
#define MAX_ELE 1000

main()
{
char vet1[MAX_ELE];      /* prima sequenza */
char vet2[MAX_ELE];      /* seconda sequenza */
char vet3[MAX_ELE*2];    /* merge */

int n;                   /* lunghezza prima sequenza */
int m;                   /* lunghezza seconda sequenza */

char aux;                /* variabile di appoggio per lo scambio */

int i, j, k, p, n1, m1;

do {
printf("Lunghezza prima sequenza: ");
scanf("%d", &n);
}
while(n<1 || n>MAX_ELE);

/* caricamento prima sequenza */
for(i = 0; i <= n-1; i++) {
printf("vet1 %d° elemento: ", i+1);
scanf("%ls", &vet1[i]);
}

do {
printf("Lunghezza seconda sequenza: ");
scanf("%d", &m);
}
while(m<1 || m>MAX_ELE);

/* caricamento seconda sequenza */
for(i=0; i<=m-1; i++) {
printf("vet2 %d° elemento: ", i+1);
scanf("%ls", &vet2[i]);
}
```

```

/* ordinamento prima sequenza */
p = n; n1 = n;
do {
    k = 0;
    for(i = 0; i < n1-1; i++) {
        if(vet1[i] > vet1[i+1]) {
            aux = vet1[i]; vet1[i] = vet1[i+1]; vet1[i+1] = aux;
            k = 1; p = i+1;
        }
    }
}
n1 = p;
}
while(k==1);

/* ordinamento seconda sequenza */
p = m; m1 = m;
do {
    k = 0;
    for(i=0; i<m1 - 1; i++) {
        if(vet2[i] > vet2[i+1]) {
            aux = vet2[i]; vet2[i] = vet2[i+1]; vet2[i+1] = aux;
            k = 1; p = i+1;
        }
    }
}
m1 = p;
}
while(k==1);

/* fusione delle due sequenze (merge) */
i = 0; j = 0; k = 0;
do {
    if(vet1[i] <= vet2[j])
        vet3[k++] = vet1[i++];
    else
        vet3[k++] = vet2[j++];
}
while(i < n && j < m);

if(i < n)
    for(; i < n; vet3[k++] = vet1[i++])
        ;
else
    for(; j < m; vet3[k++] = vet2[j++])
        ;

/* visualizzazione della fusione */
for(i=0; i < k; i++)
    printf("\n%c", vet3[i]);
}

```

Listato 5.3 Fusione di due array

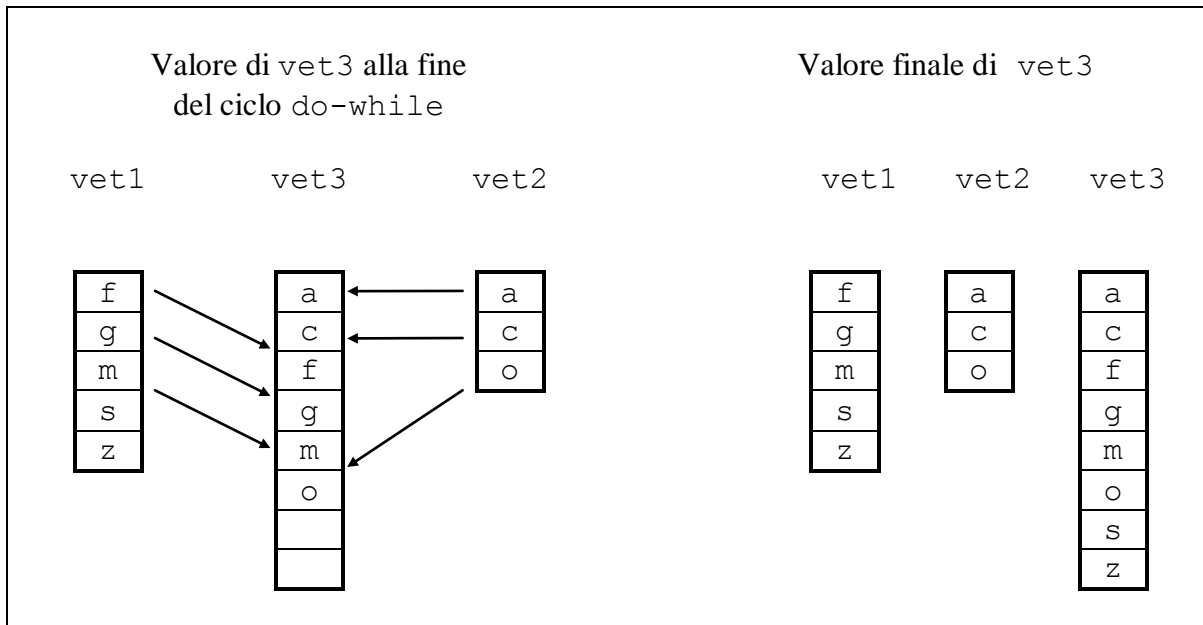


Figura 5.4 Risultato parziale e finale della fusione tra due vettori

In Figura 5.4 osserviamo il merge tra gli array ordinati `vet1` e `vet2` ordinati. L'operazione viene effettuata in due parti. La prima è data da:

```
i = 0; j = 0; k = 0;
do {
  if(vet1[i]<=vet2[j])
    vet3[k++] = vet1[i++];
  else
    vet3[k++] = vet2[j++];
}
while(i<n && j<m);
```

Si controlla se l' i -esimo elemento di `vet1` è minore o uguale al j -esimo elemento di `vet2`, nel qual caso si aggiunge `vet1[i]` a `vet3` e si incrementa i . Nel caso contrario si aggiunge a `vet3` l'array `vet2[j]` e si incrementa j . In ogni caso si incrementa k , la variabile che indicizza `vet3`, perché si è aggiunto un elemento a `vet3`. Dal ciclo si esce quando i ha valore $n-1$ o j ha valore $m-1$.

Si devono ancora aggiungere a `vet3` gli elementi di `vet1` ($j=m-1$) o di `vet2` ($i=n-1$) che non sono stati considerati. Nell'esempio precedente in `vet3` non ci sarebbero `s` e `z`. La seconda parte del merge ha proprio questo compito:

```
if(i<n)
  for(; i<n; vet3[k++] = vet1[i++])
    ;
else
  for(; j<m; vet3[k++] = vet2[j++])
    ;
```

□

5.6 Esercizi

* 1. Scrivere un programma di ordinamento in senso decrescente .

* 2. Scrivere un programma che carichi una matrice bidimensionale di caratteri e successivamente ricerchi al suo interno un valore passato in ingresso dall'utente. Il programma restituisce quindi il numero di linea e di colonna relativo all'elemento cercato se questo è presente nella matrice, il messaggio `Elemento non presente` altrimenti.

3. Modificare il programma per la ricerca binaria in modo che visualizzi i singoli passi effettuati (cioè mostri i dati di Figura 5.3). Sperimentare il comportamento del programma con la ricerca dell'elemento 45 nel seguente vettore:

vet

21	33	40	41	45	50	60	66	72	81	88	89	91	93	99
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

4. Verificare, analogamente a quanto fatto in Figura 5.1, il comportamento della prima versione di bubblesort applicata al seguente vettore:

vet

3	31	1	23	41	5	0	66	2	8	88	9	91	19	99
---	----	---	----	----	---	---	----	---	---	----	---	----	----	----

5. Verificare il comportamento della versione ottimizzata di bubblesort applicata al vettore del precedente esercizio. Quanti cicli interni si sono risparmiati rispetto alla prima versione?

6. Calcolare il numero di confronti effettuati dall'algorithm di ordinamento ingenuo applicato al vettore dell'Esercizio 4 e confrontarlo con quello di bubblesort.

7. Scrivere un programma che, richiedi i valori di un vettore ordinato in modo crescente, li inverta ottenendo un vettore decrescente. Si chiede di risolvere il problema utilizzando un solo ciclo.

8. Verificare il comportamento del programma di fusione applicato ai seguenti vettori:

vet1

3	31	41	43	44	45	80
---	----	----	----	----	----	----

vet2

5	8	21	23	46	51	60	66
---	---	----	----	----	----	----	----

9. Modificare l'algorithm di ricerca binaria nel caso il vettore sia ordinato in modo decrescente invece che crescente.

10. Se il vettore è ordinato la ricerca completa può essere migliorata in modo da diminuire in media il numero di confronti da effettuare: come? Modificare in questo senso il programma esaminato nel presente capitolo.

11. Scrivere un programma che, richiedi all'utente i primi $n-1$ elementi già ordinati di un vettore di dimensione n e un ulteriore elemento finale, inserisca quest'ultimo nella posizione corretta facendo *scivolare* verso il basso tutti gli elementi più grandi.

12. [*Insertion-sort*] Utilizzare l'algorithm del precedente esercizio per scrivere un programma che ordini il vettore contemporaneamente all'inserimento dei dati da parte dell'utente.

13. Scrivere un programma che, richiedi all'utente i valori di una matrice, ne ordini tutte le colonne in senso crescente.

15. Scrivere un programma che, richiedi all'utente i valori di una matrice, ne ordini le righe in modo che il vettore i cui elementi corrispondono alla somma delle righe risulti ordinato in senso crescente.