

## 6.4 Segmenti di codice e loro proprietà

Quanto detto fino ad ora mostra da un lato che i programmi sono logicamente composti da “pezzi”, denominati fino ad ora informalmente *segmenti* (cfr. esempio 6.1), e dall’altro che, facendo uso degli schemi algoritmici, è possibile derivare, per similitudine, per fusione, o per particolarizzazione da casi generali o astratti, una grande varietà di segmenti che, se opportunamente composti, permettono di risolvere problemi nuovi o comunque diversi da quelli già noti.

Questo procedere per decomposizione da problemi complessi a problemi più semplici e, viceversa, per fusione da problemi semplici a problemi più complessi, è molto tipico dell’attività progettuale nei diversi campi dell’ingegneria ed è universalmente considerata la chiave fondamentale per affrontare la complessità dei problemi del mondo reale.

Naturalmente, per individuare i componenti di un programma, per analizzarli, per sintetizzarli e per comporli in modo da ottenere componenti nuovi e più complessi, è necessario comprendere bene la natura di questi componenti e le regole che guidano il loro uso.

Più specificamente, si in fase di analisi dei programmi, che in fase di sintesi, occorre comprendere come:

- suddividere il programma in segmenti;
- individuare le relazioni tra i segmenti;
- individuare le proprietà dei singoli segmenti.

Naturalmente, sia nell’analisi che nella sintesi di un singolo segmento, se esso non è sufficientemente semplice, si può ripetere la decomposizione dando luogo a un procedimento *a scatole cinesi* tipico dei sistemi complessi.

Anche se, come abbiamo detto, in molti casi non vi sono regole precise e complete per affrontare i vari aspetti del problema, e non è possibile in ogni caso fare a meno dell’intuizione e dell’esperienza, pur tuttavia è possibile identificare alcune regole base che possono guidare nell’analisi e nella sintesi dei segmenti. Tali regole, anche quando non riescono a fornire un procedimento risolutivo chiaro, permettono almeno di evitare errori banali e rappresentano in ogni caso uno strumento per valutare a posteriori il programma sintetizzato. Nei prossimi paragrafi cercheremo di illustrare queste regole base e di dimostrarne l’applicazione e l’utilità attraverso alcuni ulteriori esempi.

### 6.4.1 Definizione

Il primo punto da chiarire è la nozione di segmento di codice introdotta informalmente nella discussione fin qui sviluppata.

**Definizione 6.2** *Un segmento di codice è una qualsiasi porzione di codice che contiene esclusivamente istruzioni semplici o istruzioni composte integre, complete cioè di tutte le loro parti componenti.*

Ad esempio (cfr. esempio 6.12), il gruppo di istruzioni:

```

if ( x == 0 ) {
    n_seq++;
    while ( x == 0 ) cin >> x;
}
else
    cin >> x;

```

è un segmento di codice, in quanto costituito da un’unica istruzione composta (istruzione **if**), interamente contenuta nel segmento. È anche un segmento di codice il gruppo di istruzioni (contenuto nel precedente):

```

n_seq++;
while ( x == 0 ) cin >> x;

```

in quanto composto da due istruzioni, la prima semplice e la seconda composta e interamente contenuta nel segmento (ciclo **while**).

Con riferimento al primo esempio, non è invece un segmento la porzione di codice:

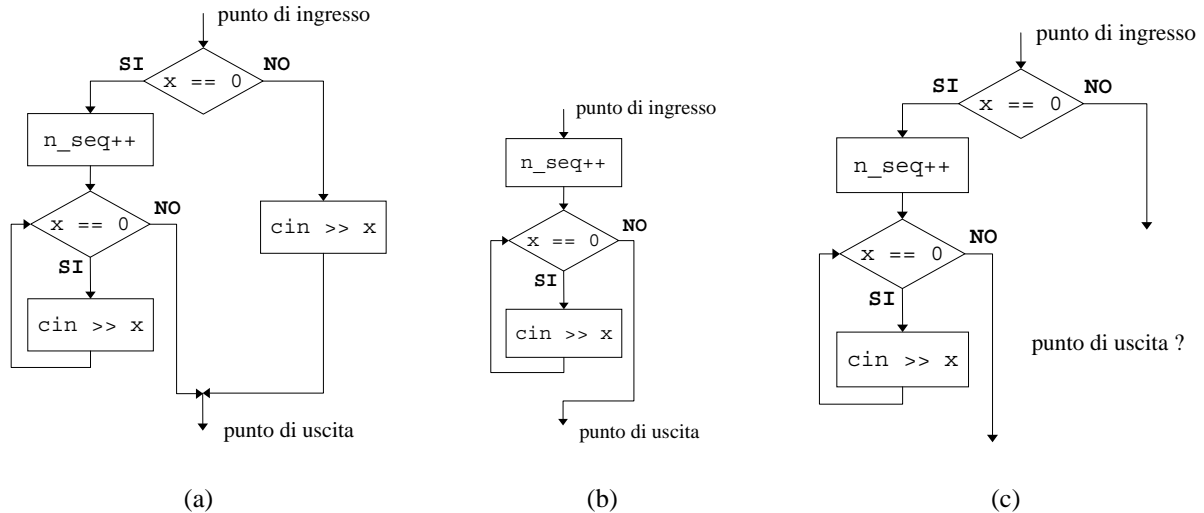


Figura 6.3: Flussi di informazioni in un segmento di codice.

```

if ( x == 0 ) {
    n_seq++;
    while ( x == 0 ) cin >> x;
}

```

perchè l'istruzione **if** è solo parzialmente presente (manca la parte *else*).

### 6.4.2 Composizione di segmenti di codice

Nell'analisi e nella sintesi dei programmi, quando si vogliono individuare o definire i componenti del programma, vanno presi in considerazione esclusivamente i segmenti di codice. Essi infatti, grazie al fatto di comprendere istruzioni complete, sono caratterizzati da un unico *punto di ingresso* e da un unico *punto di uscita*. In altre parole, per essi risultano ben determinate le istruzioni iniziale e finale di *ogni* sequenza dinamica che può essere generata dalla loro esecuzione.

Ad esempio, nel primo esempio del paragrafo precedente, trattandosi di un segmento costituito da un'unica istruzione composta (istruzione **if**), tutte le possibili sequenze dinamiche iniziano con la valutazione dell'espressione ( $x == 0$ ) e terminano dopo l'esecuzione della parte *then* o della parte *else* (entrambe contenute nel segmento). Nella figura 6.3-(a) è riportata la rappresentazione grafica della struttura del segmento da cui appare evidente l'unicità del punto di ingresso e del punto di uscita.

È facile verificare che questa stessa proprietà è posseduta dal segmento usato nel secondo esempio del paragrafo precedente (figura 6.3-(b)), mentre nel terzo esempio il punto di uscita non è univocamente determinato (figura 6.3-(c)).

I segmenti di codice, grazie all'unicità dei punti di ingresso e di uscita, hanno la proprietà di poter essere facilmente composti attraverso i tre meccanismi di composizione fondamentale (sequenza, selezione e ciclo) senza richiedere alcuna modifica.

Si consideri ad esempio la seconda soluzione dell'esempio 6.12. Una prima divisione in componenti è riportata in figura 6.4, dove il commento iniziale ha lo scopo di descrivere sinteticamente la funzione di ciascun segmento.

Come è facile verificare si tratta in entrambi i casi di segmenti di codice (il secondo consiste di una sola istruzione semplice), composti in sequenza. Questo evidenzia la logica del programma che può essere espressa scrivendo uno dopo l'altro i due commenti:

```

input dati di ingresso e calcolo del risultato
output risultato

```

```

// input dati di ingresso e calcolo del risultato
n_seq = 0;
cin >> x;
while ( x >= 0 )
  if ( x == 0 ) {
    n_seq++;
    while ( x != 0 ) cin >> x;
  }
  else
    cin >> x;

```

(a)

```

// output risultato
cout << n_seq;

```

(b)

Figura 6.4: Componenti principali dell'algoritmo.

Analizzando il primo componente, possiamo individuare al suo interno i seguenti tre sottocomponenti (tutti segmenti):

```

n_seq = 0;
cin >> x;

n_seq++;
while ( x != 0 ) cin >> x;

cin >> x;

```

Il primo segmento comprende l'inizializzazione di un contatore (`n_seq`) e l'acquisizione anticipata di un valore da `stdin`, il secondo l'incremento del contatore e l'avanzamento sullo `stdin` fino al termine di una sottosequenza di zeri consecutivi, il terzo effettua l'acquisizione di un valore da `stdin`.

I tre segmenti vengono composti secondo la logica di due schemi algoritmici (acquisizione di una lista terminata da un'informazione tappo e accumulazione del risultato nell'accumulatore `n_seq`) già evidenziati negli esempi precedenti. Tali schemi prevedono un'inizializzazione fuori ciclo (primo segmento) seguita da un ciclo **while** in cui si aggiorna l'accumulatore (prima istruzione del secondo segmento) e si avvanza (ciclo nel secondo segmento o, in alternativa, terzo segmento). La presenza dell'istruzione di selezione all'interno del ciclo rappresenta la naturale generalizzazione dello schema dell'accumulatore al caso in cui l'aggiornamento debba essere fatto solo se si verifica una particolare condizione (cfr. l'esempio 2 del paragrafo 6.3.5).

Per quanto detto, l'analisi del segmento di figura 6.4-(a) porta alla seguente descrizione dove, ai tre segmenti sono stati sostituite delle frasi in linguaggio naturale che ne riassumono la funzione:

```

inizializzazione del contatore e lettura anticipata del primo valore dallo stdin
while ( x >= 0 )
  if ( x == 0 ) {
    incrementa il contatore e
    avvanza fino al termine della sottosequenza di zeri consecutivi
  }
  else
    acquisisci un nuovo valore dallo stdin

```

Si noti che nella descrizione appena presentata, la condizione `x >= 0` del ciclo significa *la lista non è terminata*, e la condizione `x == 0` dell'istruzione di selezione significa *siamo all'inizio di una sottosequenza di zeri consecutivi*.

Con queste osservazioni, l'algoritmo può essere descritto dal seguente pseudo-codice:

```

inicializzazione del contatore e lettura anticipata del primo valore dallo stdin
while ( la lista non è terminata )
  if ( siamo all'inizio di una sottosequenza di zeri consecutivi ) {
    incrementa il contatore e
    avanza fino al termine della sottosequenza di zeri consecutivi
  }
else
  acquisisci un nuovo valore dallo stdin

```

che non contiene più alcun riferimento ai dettagli dell'algoritmo (variabili di appoggio usate, espressioni delle condizioni, ecc.).

L'ultima formulazione dell'algoritmo mette in evidenza la sua organizzazione logica e il significato che in questa organizzazione hanno le sue parti componenti. Pervenire a una formulazione dell'algoritmo con queste caratteristiche è particolarmente utile per comprendere il ragionamento fatto dal programmatore, per convincersi che non vi sono errori e per effettuare modifiche. Il procedimento seguito è basato sui seguenti passi fondamentali:

- la decomposizione in segmenti;
- l'analisi di tali segmenti attraverso l'impiego degli schemi algoritmici;
- l'identificazione del ruolo che tali segmenti svolgono nell'organizzazione complessiva dell'algoritmo ricorrendo ancora agli schemi algoritmici.

Sebbene tale procedimento non sia sempre lineare e dunque non sia riducibile a regole di automatica applicazione, l'esempio illustra il tipo di ragionamento da seguire e può essere usato come riferimento per effettuare l'analisi di altri semplici programmi (si veda a tal proposito gli esercizi proposti e i programmi mostrati nei paragrafi successivi).

### 6.4.3 Variabili di ingresso e di uscita

All'inizio del paragrafo 6.4 si è accennato alla necessità di individuare le relazioni tra i componenti di un programma. È infatti evidente che un programma non può ridursi ai suoi componenti presi isolatamente, ma che sono importanti anche le relazioni tra di essi, o, con altre parole, il modo con cui essi sono composti e cooperano al risultato finale.

Poiché i componenti sono segmenti di codice, occorre comprendere in che modo possono essere descritte le relazioni tra segmenti di codice. Tali relazioni sono di due tipi. Il primo tipo fa riferimento al meccanismo usato per comporre tra loro i segmenti. In altre parole se essi sono in sequenza, in alternativa o se vengono inclusi in un ciclo. Le relazioni di questo tipo riguardano il *controllo di flusso* del segmento risultante e determinano le possibili sequenze dinamiche che esso può generare. Come abbiamo visto informalmente nei paragrafi precedenti, in molti casi il meccanismo di composizione si basa sull'impiego di uno schema algoritmico. I segmenti vengono cioè composti mediante la loro sostituzione a quelle parti dello schema che sono descritte da frasi in linguaggio naturale. Naturalmente, la sostituzione non è puramente meccanica, ma può richiedere delle modifiche ai segmenti da comporre.

Il secondo tipo di relazioni è legato alle variabili e al fatto che esse permettono il trasferimento di informazioni da un segmento all'altro. Prima di discutere in dettaglio la questione consideriamo il seguente segmento di codice:

```

// input dati di ingresso
cin >> n;
for ( i=0; i<n; i++ )
  cin >> lista[i];
// calcolo media
somma = 0;
for ( i=0; i<n; i++ )
  somma = somma + lista[i];
media = somma/n;
// output risultato
cout << media;

```

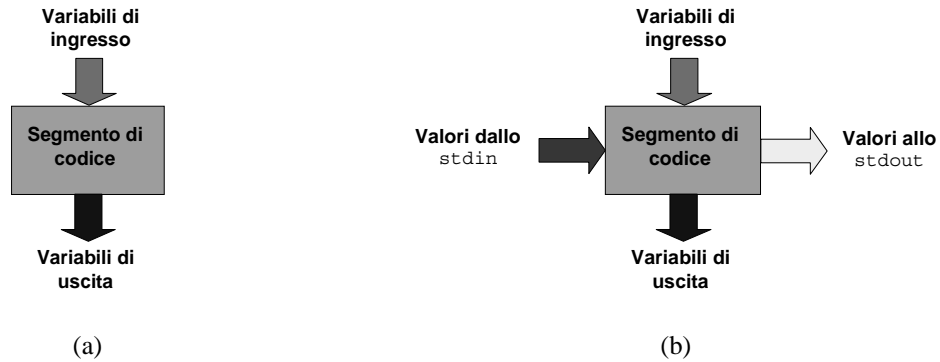


Figura 6.5: Flussi di informazioni in un segmento di codice.

Il segmento è a sua volta composto da tre segmenti (individuati dai commenti). È abbastanza evidente che il primo segmento, prendendo i valori dall'esterno, definisce le variabili `n` e `lista` che inizialmente possono essere indefinite (non contengono cioè valori utili). Il secondo segmento utilizza i valori memorizzati dal primo segmento in `n` e `lista` per calcolare la somma e definire così la variabile `somma`. Infine il terzo segmento utilizza il valore memorizzato dal secondo segmento in `somma` per effettuare la stampa e non definisce alcuna variabile.

Considerazioni simili a quelle appena fatte possono essere ripetute su algoritmo decomponibile in più segmenti. Per chiarire meglio cosa vogliamo dire, è opportuno però prima introdurre le nozioni di *variabili di ingresso* e *variabili di uscita* di un segmento di codice.

A tal fine forniamo inizialmente una definizione semplificata che applicheremo ad alcuni esempi per chiarirne il significato e metterne in luce i limiti. Sulla base di questa discussione forniremo successivamente una definizione completa delle due nozioni.

Consideriamo l'insieme  $var(S)$  delle variabili il cui identificatore compare in un segmento di codice  $S$ . Le variabili appartenenti a  $var(S)$  si dicono le variabili *referite* da  $S$ . Ad esempio, nel segmento di figura 6.4-(a), l'insieme  $var(S)$  comprende le variabili `n_seq`, `x`.

Ricordando che una variabile si dice *usata* quando viene usato il valore che essa contiene per il calcolo di un'espressione, e che si dice *definita* quando ad essa viene assegnato un valore, consideriamo l'insieme  $in(S) \subseteq var(S)$  di variabili che vengono usate prima di essere definite indichiamo poi con il termine *variabili di ingresso* del segmento  $S$  le variabili appartenenti all'insieme  $in(S)$ . Ad esempio, nel segmento di figura 6.4-(a) tale insieme è vuoto perché sia `n_seq`, sia `x` vengono subito definite. Al contrario, nel segmento di figura 6.4-(b), l'unica variabile in  $var(S)$ , cioè `x`, è una variabile di ingresso.

Consideriamo poi l'insieme  $out(S) \subseteq var(S)$  di variabili che sono definite (indipendentemente se sono anche usate) in un'istruzione del segmento e indichiamole con il termine *variabili potenzialmente<sup>4</sup> di uscita* di  $S$ . Ad esempio, nel segmento della figura 6.4-(a) le variabili potenzialmente di uscita sono, secondo questa definizione, `n_seq` e `x`, mentre nel segmento di figura 6.4-(b), non vi sono variabili di uscita.

Prima di procedere nel definire meglio sul piano formale i due concetti, conviene soffermarsi sul loro significato dal punto della logica che è alla base di un segmento di codice. Un segmento di codice descrive un algoritmo, eventualmente parziale, e pertanto descrive un procedimento che *produce un risultato a partire da certe informazioni di partenza*. Le variabili di ingresso rappresentano le informazioni di partenza dell'algoritmo descritto dal segmento e una parte di quelle di potenzialmente uscita rappresentano i risultati prodotti dall'esecuzione dell'algoritmo. In altri termini il segmento di codice, a partire dalle informazioni contenute nelle variabili di ingresso, produce dei risultati che vengono posti in un sottoinsieme delle variabili potenzialmente di uscita, che indicheremo con il termine *variabili di uscita*, senza più alcuna specificazione. La situazione può essere rappresentata graficamente con lo schema in figura 6.5-(a), dove il segmento è rappresentato da un rettangolo, le variabili di ingresso sono rappresentate mediante una freccia entrante nel lato superiore del rettangolo e le variabili di uscita sono rappresentate mediante una freccia uscente dal lato inferiore del rettangolo.

#### Osservazione 6.4

<sup>4</sup>La ragione della presenza dell'avverbio sarà chiara tra poco

Si noti la somiglianza della rappresentazione grafica usata per i segmenti del paragrafo 6.4.2 con quella usata nella figura 6.5-(a). Nel primo caso le frecce rappresentano il *flusso di controllo*, cioè la sequenza con cui i segmenti vengono eseguiti o, in altre parole, i vincoli sulle possibili sequenze dinamiche. Nel secondo caso le frecce rappresentano il flusso dei dati, cioè le variabili che consentono un trasferimento di informazioni da un segmento di codice all'altro.

Si noti anche come, dal punto di vista grafico, quando si compongono in sequenza due segmenti di codice le variabili di uscita del primo segmento vanno a coincidere con le variabili di ingresso del secondo. Sebbene sul piano della logica dell'algoritmo complessivo non sempre si dia tale esatta coincidenza<sup>5</sup>, pur tuttavia l'osservazione evidenzia anche graficamente il fatto che in un algoritmo ciascun segmento ha il compito di produrre le informazioni necessarie ai segmenti successivi e che tali informazioni vengono *trasferite* da un segmento all'altro attraverso le variabili definite (di uscita rispetto al segmento che precede) e usate (di ingresso rispetto al segmento che segue).

### Osservazione 6.5

È importante osservare che non bisogna confondere le nozioni di *variabili di ingresso e di uscita* di un segmento di codice con i *dati di ingresso e di uscita*. Nel primo caso la nozione si applica a delle *variabili* che svolgono il ruolo di trasferire informazioni *già presenti all'interno dell'esecutore* da un segmento di codice ad un altro. Nel secondo caso la nozione si applica a *informazioni* (in concreto ai *valori* di tali informazioni) che devono essere trasferite dall'esterno dell'esecutore al suo interno (dati di ingresso) o viceversa dall'interno dell'esecutore all'esterno (dati di uscita). Nel primo caso si tratta quindi di flussi interni all'algoritmo, mentre nel secondo caso si tratta di flussi tra il mondo esterno e l'algoritmo.

La situazione può essere efficacemente rappresentata in modo grafico come nella figura 6.5-(b).

Oltre alle frecce che rappresentano le variabili di ingresso e di uscita che seguono una direzione verticale dall'alto verso il basso, sono riportate una freccia entrante e una uscente che seguono una direzione orizzontale da sinistra verso destra. Queste ultime frecce corrispondono, rispettivamente, a eventuali istruzioni di input e di output presenti nel segmento e rappresentano l'ingresso o l'uscita di *valori*. Naturalmente, se il segmento non contiene istruzioni di input o di output, la corrispondente freccia rappresenterebbe un insieme vuoto di valori e potrebbe essere omessa.

Vale infine la pena notare come, ai fini della comprensione della logica che governa un algoritmo, il ruolo svolto dalle variabili di ingresso e uscita è di gran lunga più importante di quello svolto dai dati di ingresso che, sotto il profilo concettuale, non comportano normalmente grandi difficoltà<sup>6</sup>. Per questo motivo nell'analisi di un algoritmo ci concentreremo nel seguito principalmente sulle variabili di ingresso e di uscita dei segmenti che lo compongono, piuttosto che sulle istruzioni di input e di output presenti nel segmento stesso.

### Esempio 6.13

Indicando con  $S_1$  il segmento di figura 6.4-(a), abbiamo che  $in(S_1)$  è vuoto e che  $out(S_1) = \{n\_seq, x\}$ . Questo corrisponde alla logica con cui è stato scritto il segmento che ha il compito di acquisire i dati dall'esterno (non vi sono cioè informazioni di partenza già presenti in memoria) e di produrre il numero di sottosequenze di zeri consecutivi contenute nella lista fornita sullo `stdio` nell'unica variabile di uscita `n\_seq`.

### Osservazione 6.6

Si noti come  $out(S_1)$  contenga anche la variabile  $x$  che non è realmente di uscita rispetto a  $S_1$ . Se infatti si esamina la logica con cui è stato scritto l'algoritmo, si può osservare che solo `n\_seq` rappresenta un'informazione significativa al termine di  $S_1$ , mentre la variabile  $x$  ha il compito di memorizzare i valori della lista man mano che vengono acquisiti dallo standard input per consentire il conteggio delle sottosequenze di zeri consecutivi. Il suo valore finale però non ha alcun significato particolare. In altre parole la variabile  $x$  ha un ruolo all'interno di  $S_1$ , ma non ha alcun ruolo all'esterno di  $S_1$ .

Dal punto di vista della logica dell'algoritmo descritto da un segmento, questo comportamento è diverso sia da quello delle variabili di ingresso, che hanno il compito di legare un segmento con il precedente, sia da quello delle variabili di uscita come `n\_seq`, che hanno il compito di legare un segmento con il successivo.

<sup>5</sup>Non è sempre detto che *tutte* le variabili di uscita di un segmento siano di ingresso per il segmento immediatamente successivo. Una parte di esse, infatti, possono essere di ingresso per segmenti che seguono nella sequenza dinamica, ma che non sono contigui al segmento in esame.

<sup>6</sup>Non a caso le nozioni di dati di ingresso e dati di uscita sono stati introdotti subito nel capitolo 1, mentre le nozioni di variabili di ingresso e uscita hanno richiesto una trattazione molto più elaborata.

Questa osservazione porta a introdurre il termine di variabili *di algoritmo* o *ausiliarie*, termini usati per indicare le variabili solo potenzialmente di uscita, cioè quelle appartenenti ad  $out(S)$  che però non sono poi effettivamente di uscita.

Purtroppo la distinzione tra variabili di algoritmo e variabili di uscita non può essere fatta impiegando esclusivamente regole meccaniche, ma richiede la conoscenza della logica con cui è organizzato l'algoritmo descritto dal segmento. In altre parole, applicando regole meccaniche è possibile individuare solo l'insieme delle variabili potenzialmente di uscita. In tale insieme, con considerazioni di altra natura, è poi possibile distinguere le variabili di algoritmo dalle variabili di uscita vere e proprie.

#### Esempio 6.14

Alla luce di queste osservazioni, proviamo ad applicare le definizioni date ad alcuni esempi. Indicando con  $Q$  il segmento di codice:

```

m = (p+u)/2;
if ( x < a[m] )
    u = m-1;
else if ( x > a[m] )
    p = m+1;
else
    trovato = 1;

```

abbiamo:

$$in(Q) = \{p, u, x, a[m]\},$$

$$out(Q) = \{m, u, p, trovato\}.$$

Naturalmente, non avendo informazioni sul ruolo che il segmento deve svolgere non è possibile distinguere in  $out(Q)$  le variabili di algoritmo dalle variabili di uscita. Tuttavia, usando argomenti che non è il caso ora di illustrare, si può concludere che, nel caso specifico, tutte le variabili in  $out(Q)$  sono di uscita.

#### Osservazione 6.7

Notiamo che le variabili  $p$  e  $u$  sono contemporaneamente di ingresso e di uscita. È naturale definire tali variabili di *ingresso/uscita*. Nel seguito tuttavia parleremo solo di variabili di ingresso e di uscita, distinguendo quelle che appartengono contemporaneamente ai due insiemi solo quando è necessario.

#### Osservazione 6.8

Notiamo anche che possono essere variabili di ingresso, di uscita o di ingresso/uscita anche gli elementi di un array. Nell'esempio è di ingresso un elemento generico dell'array, in quanto non è in generale noto il valore di  $m$ . In queste situazioni devono essere considerate variabili di ingresso, uscita o ingresso/uscita, rispettivamente, tutti gli elementi dell'array a cui il codice fa potenziale riferimento (nell'esempio specifico tutti gli elementi, non avendo informazioni sul valore dell'indice).

#### Esempio 6.15

Consideriamo un secondo esempio. Dato il segmento di codice  $R$ :

```

for ( i=0; i<n; i++ )
    if ( a[i] < min )
        min = a[i];
    else if ( a[i] > max )
        max = a[i];

```

abbiamo:

$$\begin{aligned} in(R) &= \{n, a[0], a[1], \dots, a[n-1], \min, \max\}, \\ out(R) &= \{i, \min, \max\}. \end{aligned}$$

L'osservazione che  $i$  è la variabile di conteggio del ciclo **for** permette di concludere che  $i$  è una variabile di algoritmo e che pertanto le variabili di uscita sono solo  $\min$  e  $\max$ .

Si noti  $\min$  e  $\max$  sono sia di ingresso che di uscita perché entrambe sono usate nelle espressioni delle istruzioni di selezione prima di essere definite nella parte *then* delle stesse istruzioni.

### Osservazione 6.9

L'esempio appena mostrato mette in luce alcune ambiguità delle definizioni date in precedenza. Infatti, se il valore di  $n$  fosse minore o uguale a zero, il ciclo **for** non verrebbe eseguito, e gli elementi dell'array  $a$  e le variabili  $\min$  e  $\max$  non verrebbero né usate, né definite. Sorge allora il dubbio se tali variabili debbano essere considerate di ingresso o no. Anche per quanto riguarda le variabili di potenzialmente uscita sorge un problema analogo. Le variabili  $\min$  e  $\max$  vengono definite solo se si verificano le seguenti condizioni: ( $n > 0$ ) ed esistono elementi di  $a$  minori e maggiori, rispettivamente, dei valori iniziali di  $\min$  e  $\max$ , in caso contrario non vengono definite. Sorge allora il dubbio se le due variabili debbano essere ritenute di uscita o no.

Per rispondere al primo dubbio diamo una definizione più precisa di variabili di ingresso.

**Definizione 6.3** *Una variabile riferita in un segmento di codice è di ingresso rispetto al segmento se essa viene usata prima di essere definita in almeno una sequenza dinamica possibile per le istruzioni del segmento.*

La novità rispetto alla nozione informale usata fino ad ora sta nelle parole *in almeno una sequenza dinamica possibile per le istruzioni del segmento*. Tale precisazione permette di scogliere l'ambiguità rilevata in precedenza e concludere che gli elementi dell'array  $a$  e le variabili  $\min$  e  $\max$  sono variabili di ingresso del segmento di codice in esame, perché esistono sequenze dinamiche possibili in cui tali variabili vengono usate prima di essere definite.

Il secondo dubbio è analogo a quello rilevato per le variabili di ingresso e si risolve precisando anche in questo caso la definizione di variabile di uscita.

**Definizione 6.4** *Una variabile riferita in un segmento di codice è potenzialmente di uscita rispetto al segmento se si verifica una delle situazioni:*

1. *la variabile è definita in tutte le sequenze dinamiche possibili per le istruzioni del segmento;*
2. *la variabile è definita in almeno una sequenza dinamica possibile per le istruzioni del segmento, ed è anche di ingresso rispetto al segmento;*
3. *la variabile è definita in almeno una sequenza dinamica possibile per le istruzioni del segmento, non è di ingresso, ed è possibile stabilire se essa è stata effettivamente definita tramite altre variabili di ingresso o di uscita del segmento di codice.*

*In quest'ultimo caso la variabile si dice condizionata a tali variabili.*

Naturalmente, come già abbiamo osservato, tra le variabili potenzialmente di uscita così determinate andranno individuate quelle realmente di uscita con considerazioni che attengono alla logica dell'algoritmo descritto dal segmento.

Come si può notare la definizione appena data è molto più articolata di quella relativa alle variabili di ingresso e richiede qualche spiegazione.

Per quanto riguarda le diverse condizioni che deve verificare una variabile per poter essere considerata effettivamente di uscita, ciascuna di esse è motivata dal fatto che le variabili di uscita di un segmento devono in ogni caso assumere un valore definito (prime due condizioni della definizione), o in alternativa, deve essere possibile distinguere quando esse hanno un valore definito e quando invece sono indefinite e pertanto non possono essere usate (terza condizione della definizione).

Con riferimento al segmento dell'esempio, le variabili  $\min$  e  $\max$  verificano la definizione anche se vengono definite solo in alcune delle sequenze dinamiche possibili, poiché sono anche di ingresso (caso 2). Esse pertanto possono essere variabili di uscita del segmento, e di fatto lo sono in quanto il loro valore al termine del segmento è evidentemente significativo (rappresentano il valore minimo e massimo contenuti nella lista rappresentata dall'array  $a$  e dal riempimento  $n$  nel caso in cui  $n > 0$ ).

## 6.5 Regole per la costruzione, la composizione e la modifica di segmenti di codice

Nel paragrafo 6.4.2 abbiamo mostrato come i segmenti di codice possano essere composti mediante i meccanismi base della programmazione per ottenere segmenti più complessi. È tuttavia evidente che non basta comporre in un modo qualunque due segmenti di codice, ciascuno dei quali svolge una precisa funzione, per ottenere un segmento di codice che svolga correttamente la funzione desiderata, ma che la composizione va effettuata scegliendo accuratamente i componenti, scegliendo la modalità di composizione più opportuna e aggiungendo, in qualche caso, ulteriore codice “di collegamento”.

In generale, come abbiamo già precisato più volte, non è assolutamente possibile riassumere in regole precise come effettuare questa operazione che è per l'appunto uno dei compiti fondamentali nella sintesi dei programmi. È possibile però fornire alcune semplici regole che devono essere rispettate nella composizione di segmenti di codice perché il risultato della composizione possa essere potenzialmente corretto. Si tratta di condizioni *necessarie* per la correttezza, che sebbene non siano da sole sufficienti a garantirla, sono tuttavia utili per due ragioni. La prima è che esse offrono un procedimento per rilevare errori e incongruenze, sempre in agguato in ogni attività progettuale e, in particolare, nella sintesi dei programmi. La seconda ragione è che tali regole tendono a orientare l'attenzione del programmatore su aspetti essenziali, e pertanto rappresentano un solido fondamento su cui sviluppare, con l'aiuto dell'intuizione e dell'esperienza, la soluzione cercata.

Le regole che vogliamo illustrare sono tutte basate sulla nozione di variabili di ingresso e variabili di uscita di un segmento di codice e derivano dalla regola fondamentale che le variabili non possono mai essere usate senza prima essere state definite. Sebbene tale regola sia stata già illustrata parlando delle espressioni e dell'istruzione di assegnazione, la ripetiamo per maggiore chiarezza dell'esposizione.

**Regola 6.1** *Condizione necessaria per la correttezza di un programma è che, per qualunque sequenza dinamica possibile per le sue istruzioni, tutte le variabili usate siano state precedentemente definite.*

Applicando questa regola alla costruzione dei programmi mediante composizione e modifica di segmenti di codice, risultano come immediata conseguenza le seguenti condizioni necessarie per la correttezza della composizione.

**Regola 6.2** *Se un segmento di codice  $S$  ha una variabile di ingresso  $x$ , esso deve essere composto con altri segmenti di codice in modo tale che, per tutte le sequenze dinamiche possibili, la sua esecuzione sia preceduta dall'esecuzione di un altro segmento  $Q$  che ha  $x$  come variabile di uscita non condizionata.*

La regola precedente può essere estesa al caso in cui  $x$  è di uscita ripetuto a  $Q$ , ma condizionata a un'altra variabile  $y$ .

**Regola 6.3** *Nel caso la variabile  $x$  di uscita rispetto al segmento  $Q$  che precede  $S$ , sia condizionata a una o più variabili, allora la composizione è corretta solo se l'uso di  $x$  in  $S$  è condizionato al valore di tali variabili.*

Come corollario alla regola 6.1 vale la seguente.

**Regola 6.4** *Il segmento di codice con cui inizia un programma non deve avere variabili di ingresso.*

Si noti infatti che, in caso contrario, esisterebbero sequenze dinamiche possibili nelle quali tali variabili di ingresso verrebbero usate prima di essere state definite.

Un'altra regola di costruzione dei programmi è la seguente.

**Regola 6.5** *Un segmento di codice che non sia quello con cui termina il programma deve avere variabili di uscita, a meno che non contenga istruzioni di output.*

Il motivo di tale regola è che, in caso contrario (se cioè il segmento non ha né variabili di uscita, né contiene istruzioni di output), il segmento è inutile in quanto potrebbe essere tranquillamente tolto dal programma senza produrre alcun effetto. Sebbene una tale situazione non sia in sé necessariamente un errore, è quantomeno un indice che il programma contiene istruzioni non logicamente collegate alle altre.

Terminiamo con un esempio di applicazione delle nozioni introdotte nei paragrafi precedenti. Per snellire la discussione, nel seguito numereremo le linee di codice e useremo la convenzione di usare il simbolo  $S_{x-y}$  per indicare il segmento di codice che inizia alla linea  $x$  e termina alla linea  $y$  incluse. Nel caso un segmento sia costituito da una sola riga  $x$  useremo per indicarlo il simbolo  $S_x$ .

### Esempio 6.16

Consideriamo la specifica:

*Scrivere un programma che riceve in ingresso un numero naturale  $N$  seguito da una lista di  $N$  numeri e un ulteriore numero  $M$ , e che stampa in uscita la lista letta concatenata a se stessa  $M$  volte*

Il problema viene risolto dal seguente programma:

```

1.  #include <iostream.h>
2.  #include <stdlib.h>
3.
4.  int main()
5.  {
6.      int N, M;
7.      const int MAX = 100;
8.      int a[MAX];
9.      int i, j;
10.
11.     cin >> N;
12.     for ( i=0; i<N; i++ )
13.         cin >> a[i];
14.
15.     cin >> M;
16.
17.     for ( j=0; j<M; j++ )
18.         for ( i=0; i<N; i++ ) {
19.             cout << a[i];
20.             cout << ' ';
21.         }
22.     cout << endl;
23.
24.     return 0;
25. }
```

dove la numerazione delle linee di codice è stata riportata per semplificare la successiva discussione.

Cerchiamo ora di analizzare la soluzione proposta facendo uso delle nozioni di schema algoritmico e di segmento di codice e delle regole per la loro composizione.

A parte le dichiarazioni delle variabili, che possono essere trascurate perché non influenzano la logica dell'algoritmo, con riferimento agli schemi noti si può notare che il segmento:

```

11.     cin >> N;
12.     for ( i=0; i<N; i++ )
13.         cin >> a[i];
```

è un'acquisizione di lista di lunghezza nota.

Tale segmento non ha variabili di ingresso e ha  $N$ ,  $i$ ,  $a[0]$ ,  $a[1]$ ,  $\dots$ ,  $a[N-1]$  come variabili potenzialmente di uscita. Di queste,  $i$  è di algoritmo e le altre sono realmente di uscita. Il segmento è il primo del programma e verifica pertanto la regola 6.4.

Con riferimento alla logica con cui è organizzato il programma, il segmento appena considerato *acquisisce dallo standard input una lista preceduta dalla sua lunghezza*. Questo compito è coerente con quanto indicato nella traccia e con le variabili di uscita che coincidono con la rappresentazione della lista letta mediante array e riempimento.

Sempre con riferimento agli schemi introdotti, il segmento:

```

18.     for ( i=0; i<N; i++ ) {
19.         cout << a[i];
20.         cout << ' ';
21.     }

```

è una scansione semplice della lista  $(a,N)$  in cui l'elaborazione del generico elemento della lista coincide con un'istruzione di output che aggiunge il valore dell'elemento allo standard output, seguita da un'istruzione di output che aggiunge allo standard output un carattere bianco.

Le variabili di ingresso del segmento sono:  $N$ ,  $a[0]$ ,  $a[1]$ ,  $\dots$ ,  $a[N-1]$ ; c'è la variabile di algoritmo  $i$ ; non vi sono variabili di uscita (ma la regola 6.5 è soddisfatta per la presenza di istruzioni di uscita).

Dal punto di vista logico il segmento *stampa la lista*  $(a,N)$ , cioè aggiunge allo standard output la lista rappresentata dall'array  $a$  e dal riempimento  $N$ . Tale compito è coerente con le variabili di ingresso individuate e con il fatto che non vi sono variabili di uscita. In altre parole il "prodotto" dell'esecuzione del segmento è esclusivamente esterno all'esecutore.

Il segmento appena esaminato è il corpo di un ciclo **for**, che è a sua volta un'istanza dello schema della scansione. Tenendo presente le considerazioni appena fatte su  $S_{18-21}$ , il segmento  $S_{17-21}$  può essere descritto in termini astratti come segue:

```

for ( j=0; j<M; j++ )
    stampa la lista (a,N);

```

Che mette in evidenza come la stampa ripetuta della lista  $(a,N)$  viene ottenuta annidando la scansione che effettua una stampa della lista all'interno di una scansione delle  $M$  stampe. Il segmento  $S_{17-21}$ , oltre alle stesse variabili di ingresso del segmento  $S_{18-21}$ , ha  $M$  come ulteriore variabile di ingresso.

Vi sono poi due segmenti di codice non ancora considerati:

```

15.     cin >> M;

```

e:

```

22.     cout << endl;

```

Si tratta di segmenti costituiti da una sola istruzione per i quali l'analisi risulta banale.

Vale invece la pena osservare come i segmenti individuati siano composti rispettando la regola 6.1 e, più specificamente le sue conseguenze rappresentate dalle regole 6.2 e 6.3. Infatti, il segmento  $S_{17-21}$  (l'unico che ha variabili di ingresso) è correttamente preceduto da un segmento che ha come variabili di uscita le variabili:  $N$ ,  $a[0]$ ,  $a[1]$ ,  $\dots$ ,  $a[N-1]$  (il segmento dalla  $S_{11-13}$ ) e da un segmento che ha come variabili di uscita la variabile  $M$  (il segmento  $S_{15}$ ). Tale ordine corrisponde infatti alla logica della soluzione che acquisisce dapprima la lista, poi il valore  $M$  e infine aggiunge allo standard output  $M$  copie concatenate della lista letta.

La presenza dell'ultimo segmento  $S_{22}$  ha il solo scopo di terminare con un'andata a capo la sequenza di valori aggiunta allo standard output.

## 6.6 Ricerca di una proprietà in una lista

Esaminiamo ora un ulteriore schema algoritmico di grandissima importanza per la possibilità di impiegarlo, con le opportune generalizzazioni, in un grandissimo numero di situazioni a prima vista completamente diverse.

**Scopo** Lo schema di *ricerca di una proprietà in una lista* ha lo scopo di scandire una lista alla ricerca di un elemento che verifica una determinata proprietà. Anche per questo schema esistono due varianti a seconda che la lista generata sia rappresentata mediante array e riempimento o mediante array e informazione tappo.

**Pseudo-codice** *versione per liste rappresentate con array e riempimento*

```

trovato = false;
i = 0;
while ( (i<n) && !trovato )
    if ( a[i] verifica la proprietà cercata )
        trovato = true;
    else
        i++;

```

versione per liste rappresentate con array e informazione tappo

```

trovato = false;
i = 0;
while ( (a[i]!=TAPPO) && !trovato )
    if ( a[i] verifica la proprietà cercata )
        trovato = true;
    else
        i++;

```

**Commenti** Nello schema oltre alle variabili che rappresentano la lista, sono impiegate altre due variabili essenziali: l'indice *i*, usato per scandire le celle dell'array, e una variabile *trovato* di tipo **bool**, che indica se la proprietà è stata trovata o no.

La prima osservazione da fare sullo schema riguarda la natura della condizione di uscita del ciclo **while**. Si tratta di una condizione composta, costituita da due sottocondizioni. La prima sottocondizione verifica che l'elemento corrente dell'array faccia ancora parte della lista (cioè: ( $i < n$ ) nella prima variante e ( $a[i] \neq \text{TAPPO}$ ) nella seconda variante). La seconda sottocondizione verifica che la proprietà cercata non sia stata ancora trovata (cioè il valore della variabile *trovato* è **false**). Le due sottocondizioni sono composte mediante l'operatore **AND** e dunque il corpo del ciclo viene eseguito fin tanto che *l'elemento corrente dell'array fa parte della lista e la proprietà cercata non è stata trovata*.

Quando il ciclo termina si possono dare due casi:

1.  $i \geq n$  e *trovato* uguale a **false**: in questo caso sono state esaminate una alla volta tutte le caselle dell'array che contengono elementi della lista senza che sia stata trovata la proprietà cercata.
2.  $i < n$  e *trovato* uguale a **true**: la proprietà è stata trovata e il valore di *i* indica la casella dell'array che ha verificato la proprietà cercata; si noti che possono esserci altre caselle successive che verificano la proprietà, ma il ciclo si ferma alla prima; ulteriori considerazioni in proposito verranno svolte negli esempi riportati di seguito.

Si noti che il ciclo è scritto in modo tale che non può accadere che le due sottocondizioni della condizione di uscita del ciclo siano false entrambe contemporaneamente. Questa osservazione permette di concludere che, al termine dell'esecuzione dello schema, il valore della variabile *trovato* permette di sapere quale dei due casi si è verificato.

Un'altra osservazione importante riguarda il valore dell'indice *i* al termine dell'esecuzione dello schema. Nel caso 1 il valore assunto da *i* non ha alcun significato particolare e tale variabile è servita solo per scandire l'intera lista senza che sia stata trovata la proprietà cercata. Al contrario, nel caso 2 il valore assunto da *i* corrisponde all'indice del primo elemento della lista che verifica la proprietà cercata. In questo caso pertanto *i* contiene un'informazione potenzialmente significativa.

Questa considerazione ci permette di valutare meglio il ruolo delle variabili impiegate nello schema. Le variabili di ingresso sono quelle usate per rappresentare la lista (array e riempimento nella prima variante, solo l'array nella seconda variante). Le variabili potenzialmente di uscita sono *trovato* e *i*. La prima è di uscita e contiene il principale risultato dell'algoritmo (se la proprietà è stata trovata o no). La seconda è pure di uscita, ma condizionata al valore di *trovato*, nel senso che il suo valore è significativo solo nel caso 2<sup>7</sup>.

Vale la pena osservare come la classificazione in variabili di ingresso e variabili di uscita fatta sulla base delle proprietà strutturali del codice corrisponda ancora una volta al loro ruolo dal punto di vista concettuale. Poiché lo schema ha lo scopo di cercare se esiste un elemento della lista che verifica una proprietà data, è ovvio che la

<sup>7</sup>Si noti che al termine dello schema, da un punto di vista concettuale, nel caso 1 il valore di *i* va considerato *indefinito* e pertanto tale variabile non può essere usata dai segmenti successivi.

lista rappresenti l'informazione di partenza (variabili di ingresso) e che l'informazione se la proprietà è stata trovata rappresenti il risultato (variabile di uscita). Inoltre, nel caso la proprietà sia stata verificata da un elemento della lista, anche l'informazione su quale elemento ha verificato la proprietà è un risultato dell'algoritmo.

Un ultimo commento riguarda la variante dello schema per liste rappresentate con array e informazione tappo. Se si confronta tale variante con lo schema di scansione di una lista con informazione tappo, si può osservare facilmente che la seconda variante dello schema di ricerca è una *fusione* della prima variante dello schema di ricerca con lo schema di scansione di una lista con informazione tappo. In altre parole, la seconda variante dello schema di ricerca non è propriamente uno schema base, ma un derivato da schemi già noti. L'osservazione mette in luce ancora una volta la capacità degli schemi algoritmici di combinarsi e generare un gran numero di varianti in grado di trattare le situazioni più diverse.

**Esempi** Forniamo di seguito alcuni esempi di applicazione dello schema alla ricerca di proprietà specifiche. In tutti i casi forniamo il segmento corrispondente alla prima variante dello schema, lasciando al lettore il compito di derivare l'equivalente segmento nel caso la lista sia rappresentata con array e informazione tappo.

1. Il seguente segmento verifica se la lista contiene un elemento uguale a un valore  $x$  dato.

```
trovato = false;
i = 0;
while ( (i < n) && !trovato )
    if ( a[i] == x )
        trovato = true;
    else
        i++;
```

Come è ovvio sulla base di quanto osservato in precedenza, sono variabili di ingresso del segmento:  $n$ , le caselle di  $a$  dall'indice 0 all'indice  $n-1$  inclusi, e  $x$ . In altre parole la lista e il valore da cercare. Le variabili `trovato` e `i` sono di uscita, la seconda condizionata alla prima.

È interessante vedere anche come un algoritmo di ricerca possa essere seguito da un segmento che usi i risultati della ricerca stessa. Volendo, ad esempio, semplicemente stampare un messaggio che informi sull'esito della ricerca, il codice riportato sopra potrebbe essere immediatamente seguito dal seguente segmento:

```
if ( trovato ) {
    cout << " Il valore " << x;
    cout << " e' presente nella posizione ";
    cout << i+1 << endl;
}
else {
    cout << " Il valore " << x;
    cout << " non e' presente" << endl;
}
```

Si noti come, rispetto a questo segmento, le variabili `trovato` e `i` siano di ingresso, ma l'uso di quest'ultima sia condizionato al valore della prima (nella parte *else* la variabile `i` non compare). Questo è coerente con il ruolo che tali variabili hanno rispetto al segmento che effettua la ricerca (cfr. regola 6.3). Considerazioni analoghe possono ripetersi nel caso dei segmenti presentati negli esempi successivi.

2. Il seguente segmento verifica se la lista contiene un elemento maggiore di un valore  $x$  dato.

```
trovato = false;
i = 0;
while ( (i < n) && !trovato )
    if ( a[i] > x )
        trovato = true;
    else
        i++;
```

Si noti come la ricerca di una proprietà differente comporti solo la modifica della condizione che determina se cambiare il valore di `trovato` o avanzare nella scansione della lista.

3. Il seguente segmento verifica se due liste di uguale lunghezza sono uguali. Le due liste sono rappresentate dagli array `lista1` e `lista2` e dal riempimento (comune) `n`. Il valore della variabile `uguali` di tipo **bool** indica se le due liste sono uguali (il valore di `uguali` è **true**) o diverse (il valore di `uguali` è **false**).

```
trovato = false;
i = 0;
while ( (i<n) && !trovato )
    if ( lista1[i] != lista2[i] )
        trovato = true;
    else
        i++;
uguali = !trovato;
```

Il corretto funzionamento del segmento si basa sull'osservazione che, avendo supposto uguale la lunghezza, le due liste sono uguali se e solo se nessun elemento della prima è diverso dal corrispondente elemento della seconda. Il problema si trasforma pertanto nella ricerca della seguente proprietà:

*esiste un elemento della prima lista diverso dal corrispondente elemento della seconda lista*

e nell'assegnazione alla variabile `uguali` del corrispondente valore di verità *negato*. In altre parole, si usa lo schema di ricerca per risolvere il problema opposto a quello dato e si ottiene il risultato voluto negando il risultato della ricerca.

#### Osservazione 6.10

L'esempio precedente suggerisce di derivare dallo schema di ricerca un'ulteriore variante, che usa valori opposti per la variabile di tipo **bool**. La variante è la seguente (come al solito diamo solo la versione in cui la lista è rappresentata con array e riempimento, essendo l'altra di facile derivazione):

```
non_trovato = true;
i = 0;
while ( (i<n) && non_trovato )
    if ( a[i] non verifica la proprietà voluta )
        non_trovato = false;
    else
        i++;
```

dove abbiamo cambiato nome alla variabile di tipo **bool** per metterne in risalto il differente ruolo logico. Anche la frase che descrive la condizione cercata è stata modificata per mettere in risalto che questa versione può essere usata in quei casi in cui si vuole stabilire se una proprietà è verificata da *tutti* gli elementi della lista e pertanto il problema si traduce nella ricerca di un elemento che non la verifica.

Se la condizione “`a[i] non verifica la proprietà voluta`” risulta sempre falsa, allora il ciclo termina dopo aver esaminato l'ultimo elemento della lista e la variabile `non_trovato` conserva il valore **true** assegnato inizialmente. Questo indica che tutti gli elementi della lista verificano la proprietà voluta. Se invece la condizione “`a[i] non verifica la proprietà voluta`” risulta vera in almeno un caso, allora la variabile `non_trovato` assume il valore **false** e questo indica che almeno un elemento della lista non verifica la proprietà voluta.

Il segmento di codice che verifica se due liste sono uguali si può allora scrivere:

```
uguali = true;
i = 0;
while ( (i<n) && uguali )
    if ( lista1[i] != lista2[i] )
        uguali = false;
    else
        i++;
```

dove la variabile *uguali*, che ha lo stesso significato di prima, viene usata direttamente per controllare l'eventuale termine prematuro della ricerca.

4. Lo schema della ricerca di una proprietà si può anche ulteriormente generalizzare al caso in cui la proprietà coinvolge più elementi della lista. È sufficiente solo che sia possibile verificare la proprietà esaminando un elemento per volta, usando variabili ausiliarie per tener conto dello stato della ricerca sugli elementi precedentemente esaminati.

Per illustrare la questione consideriamo il problema di stabilire se una lista di numeri contiene almeno *k* elementi uguali a zero. Si tratta evidentemente della ricerca di una proprietà, ma che non riguarda un singolo elemento della lista. D'altra parte noi sappiamo che per contare gli elementi di una lista uguali a zero possiamo usare una scansione con accumulatore che esamina gli elementi della lista per l'appunto uno alla volta.

La soluzione più naturale al problema è pertanto la fusione dello schema di ricerca con lo schema della scansione con accumulatore. Il segmento che ne risulta è il seguente:

```
n_zeri = 0;
almeno_k_zeri = false;
i = 0;
while ( (i<n) && !almeno_k_zeri ) {
    if ( a[i] == 0 )
        n_zeri++;
    if ( n_zeri >= k )
        almeno_k_zeri = true;
    else
        i++; }
```

Il lettore è invitato per esercizio a identificare, nel codice dato, gli elementi appartenenti ai due schemi sopra citati. Quello che invece vale la pena di osservare è come, in questo caso, la condizione cercata sia:

```
n_zeri >= k,
```

e come, al fine di valutarla correttamente, sia necessario aggiungere istruzioni aggiuntive per esaminare l'elemento corrente della lista *a[i]* prima di aggiornare la variabile *n\_zeri*.

Concludiamo il paragrafo svolgendo un esempio completo che ci consenta di applicare gran parte dei concetti fin qui illustrati.

### Esempio 6.17

**Specifica del problema** Scrivere un programma che riceve in ingresso un numero naturale *N1* seguito da una prima lista di *N1* numeri, seguita da un numero naturale *N2* seguito da una seconda lista di *N2* numeri, e che stampa in uscita la lista dei numeri presenti solo in una delle due liste lette.

**Casi di test** Per chiarire il significato della specifica riportiamo i seguenti casi di test.

stdin	4 1 0 3 7 5 1 8 2 3 6
stdout	0 7 8 2 6

stdin	2 1 0 3 7 5 4
stdout	1 0 7 5 4

stdin	0 5 0 3 7 5 4
stdout	0 3 7 5 4

**Soluzione** Sulla base dell'esperienza accumulata nei numerosi esempi visti fino ad ora possiamo abbozzare la seguente soluzione:

```
#include <iostream.h>
#include <stdlib.h>
int main() {
    const int MAX = 100;
    int N1, N2, i;
    double l1[MAX], l2[MAX];

    // lettura della prima lista (l1,N1)

    // lettura della seconda lista (l2,N2)

    // stampa degli elementi di (l1,N1) non presenti in (l2,N2)

    // stampa degli elementi di (l2,N2) non presenti in (l1,N1)

    return 0;
}
```

dove i commenti rappresentano dei segmenti di codice da sviluppare che svolgono il compito descritto brevemente nel commento stesso.

I segmenti di lettura delle due liste sono banali e vengono riportati nella versione finale della soluzione. Per quanto riguarda la *stampa degli elementi di (l1,N1) non presenti in (l2,N2)* osserviamo che si tratta di scandire la lista (l1,N1), stampando ogni elemento che non sia presente in (l2,N2). Il segmento può essere pertanto sviluppato nella seguente soluzione non definitiva che usa lo schema di scansione:

```
for ( i=0; i<N1; i++ ) {
    // cerca l1[i] in (l2,N2) e assegna il risultato della ricerca a trovato
    if ( !trovato ) {
        cout << l1[i];
        cout << ' ';
    }
}
```

Come prima il commento rappresenta un segmento da sviluppare. In questo caso si tratta evidentemente della ricerca del valore di `l1[i]` nella lista rappresentata dall'array `l2` e dal riempimento `N2`. Si ha pertanto la seguente soluzione definitiva:

```
for ( i=0; i<N1; i++ ) {
    trovato = false;
    j = 0;
    while ( (j<N2) && !trovato )
        if ( l2[j] == l1[i] )
            trovato = true;
        else
            j++;
    if ( !trovato ) {
        cout << l1[i];
        cout << ' ';
    }
}
```

dove compaiono le nuove variabili `trovato` e `j` che vanno opportunamente dichiarate.

Osservando che il segmento corrispondente al commento *stampa degli elementi di (l2,N2) non presenti in (l1,N1)* è identico a quello appena scritto purché si scambi `l1` con `l2`, e `N1` con `N2`, si ottiene la seguente soluzione finale:

```

#include <iostream.h>
#include <stdlib.h>
int main() {
    const int MAX = 100;
    int N1, N2, i, j;
    double l1[MAX], l2[MAX];
    bool trovato;

    // lettura della prima lista (l1,N1)
    cin >> N1;
    for ( i=0; i<N1; i++ )
        cin >> l1[i];

    // lettura della seconda lista (l2,N2)
    cin >> N2;
    for ( i=0; i<N2; i++ )
        cin >> l2[i];

    // stampa degli elementi di (l1,N1) non presenti in (l2,N2)
    for ( i=0; i<N1; i++ ) {
        trovato = false;
        j = 0;
        while ( (j<N2) && !trovato )
            if ( l2[j] == l1[i] )
                trovato = true;
            else
                j++;
        if ( !trovato ) {
            cout << l1[i];
            cout << ' ';
        }
    }

    // stampa degli elementi di (l2,N2) non presenti in (l1,N1)
    for ( i=0; i<N2; i++ ) {
        trovato = false;
        j = 0;
        while ( (j<N1) && !trovato )
            if ( l1[j] == l2[i] )
                trovato = true;
            else
                j++;
        if ( !trovato ) {
            cout << l2[i];
            cout << ' ';
        }
    }

    return 0;
}

```

dove sono stati lasciati i commenti per evidenziare i segmenti che compongono l'algoritmo complessivo.