UNITÀ DI APPRENDIMENTO 4
LE STRUTTURE DI DATI

IN QUESTA UNITÀ IMPARERAI...

- Che cosa sono i vettori
- Come si gestiscono i vettori
- Le principali operazioni sui vettori
- Che cosa sono e come si gestiscono le matrici
- Che cosa sono e come si gestiscono i record e gli array di record

1 I vettori

Proviamo a risolvere il seguente problema: *dati in input 4 numeri, stamparli nell'ordine inverso.* La soluzione non appare per nulla complessa: è sufficiente richiedere in input quattro variabili, ad esempio *A*, *B*, *C* e *D*, e stamparle nell'ordine inverso, cioè *D*, *C*, *B* e *A*.

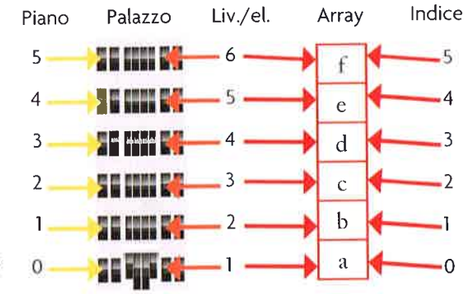
Ora complichiamo il problema: *dati in input 100 numeri, stamparli nell'ordine inverso.* La situazione è un po' diversa, in quanto dovremmo richiedere in input ben 100 variabili con nomi diversi (prova solamente a pensare agli inconvenienti legati alla scelta dei nomi). Utilizzando un così elevato numero di variabili, però, la soluzione impostata, anche se corretta, non rispecchia le regole di una buona programmazione.

Complichiamo ancora un po' il problema: *dati in input N numeri, stamparli nell'ordine inverso.* Questa volta ci troviamo davvero nei guai, in quanto, con le nostre attuali conoscenze, il problema non può essere risolto: non sappiamo, infatti, quante variabili utilizzare. Potrebbero essere 5, o 100, o 1000. Come fare? È necessario utilizzare una struttura in grado di contenere un insieme di dati dello stesso tipo (nel nostro caso, tutti numeri). Abbiamo bisogno di un vettore.

Un **vettore** (o **array monodimensionale**) è una struttura di dati, di tipo sequenziale a carattere statico, identificata da un **nome** e costituita da un insieme di **elementi** omogenei fra loro, individuabili per mezzo di un **indice**. La struttura astratta vettore è memorizzata, di solito, tramite la struttura concreta sequenziale.

Un vettore è un insieme di variabili dello stesso tipo a cui è possibile accedere tramite un nome comune e in cui è possibile referenziare uno specifico elemento tramite un indice. Nelle variabili semplici si accede al valore contenuto specificando il nome della variabile; inoltre, una variabile con un valore diverso avrà un nome diverso. Nel vettore esiste un nome, che però identifica il vettore come struttura: i suoi singoli elementi verranno referenziati specificando la loro posizione relativa all'interno della struttura (l'indice).
Si può immaginare un array come una sorta di casellario, le cui caselle sono dette **elementi (o celle) dell'array stesso**. Ciascuna delle celle si comporta come una variabile tradizionale; tutte le celle sono variabili di uno stesso tipo preesistente, detto **tipo base** dell'array. Si parla perciò di array di interi, array di caratteri, array di stringhe e così via.

Ciascuna delle celle dell'array è identificata da un valore **indice**, generalmente numerico, che può assumere come valori numeri interi contigui che partono da 0 o da 1. Si può quindi parlare della cella di indice 0, di indice 1 e, in generale, di indice *N*, dove *N* è un intero compreso fra 0 (o 1) e il valore massimo per gli indici dell'array (ossia la **dimensione** dell'array).
Nella stesura degli algoritmi si è soliti far partire l'indice dal valore 1, poiché ciò è più naturale e consente di trattare i vettori in modo più intuitivo. Per i linguaggi di programmazione il discorso è diverso, perché alcuni di essi impongono che l'indice del vettore parta da 0 e altri lo fanno partire da 1 (alcuni permettono entrambe le cose). Ad esempio, nel linguaggio C ogni elemento viene identificato da un numero, contando a partire da 0 (invece che da 1) e arrivando a $N-1$ (se la dimensione è uguale a $N=10$, il valore massimo dell'indice sarà 9). Il fatto di far partire l'indice da 0 non deve generare confusione.
Per capire meglio il concetto, osserva la figura qui a lato, che ci riporta a un paragone con il mondo reale. In questa immagine l'array viene paragonato a un palazzo. Pensaci bene: quando diciamo che un palazzo ha 5 piani, in realtà ha sei livelli: infatti il piano terra è il primo livello, il primo piano il secondo, e così via. Lo stesso succede nell'array: se abbiamo un array di dimensione 6, i suoi indici andranno da 0 a 5 e l'elemento richiamato, ad esempio, tramite l'indice 3, è il quarto elemento, perché si inizia a contare da 0.



La sintassi utilizzata nella dichiarazione di un vettore è la seguente:

```
<NomeVettore>: ARRAY[<Dimensione>] di <TipoElemento>
```

dove:

- `<NomeVettore>` rappresenta il nome da assegnare alla variabile di tipo array;
- `ARRAY` è la parola chiave per la definizione di questo nuovo tipo di dati;
- `<Dimensione>` rappresenta il numero massimo di elementi nel vettore;
- `<TipoElemento>` rappresenta il tipo di dati degli elementi del vettore.

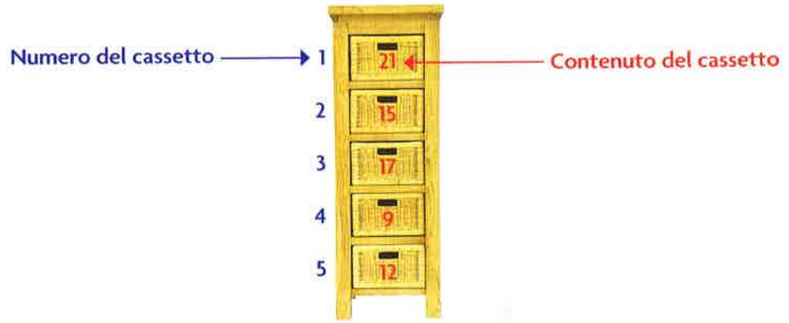
La possibilità di fare riferimento a più informazioni come appartenenti a un unico insieme ordinato logicamente appare, quindi, una delle esigenze fondamentali della programmazione. L'indice, i cui valori devono appartenere a un insieme numerabile, definisce (nell'insieme degli elementi del vettore) una relazione d'ordine totale. Per ogni elemento dell'insieme, infatti, è possibile stabilire se segue o precede un altro.
Quindi, se indichiamo con *Vet* un insieme di *N* informazioni, possiamo affermare che:

Vet[I] precede Vet[I + 1] per ogni I = 1, 2, ..., N – 1

dove:

Vet[I] con I = 1, 2, ..., N

indica un singolo elemento dell'insieme *Vet*, quello di indice *I*.
Vet[I] è da intendersi come l'elemento del vettore *Vet* che si trova nella posizione identificata dal valore dell'indice *I*. Ma che cosa significa dire ciò? Riprendiamo l'analogia con la cassettiera formata da un numero *N* di cassetti tutti identici.



LO SAI CHE...

Le parentesi quadre presenti nella dichiarazione, che racchiudono il campo di validità dell'indice, non indicano opzioni. Facciamo un esempio per consolidare il concetto: dichiariamo un vettore di interi di nome Vet la cui dimensione è 50. Scriveremo:

```
Vet: ARRAY[50] di INTERO
```

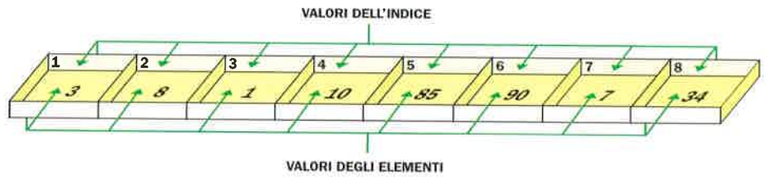
Dichiariamo, ora, un vettore di stringhe di nome Nomi e di dimensione pari a 100. Ogni elemento può contenere una sequenza di caratteri alfanumerici lunga al massimo 35 caratteri. Scriveremo:

```
Nomi: ARRAY[100]
      di STRINGA[35]
```

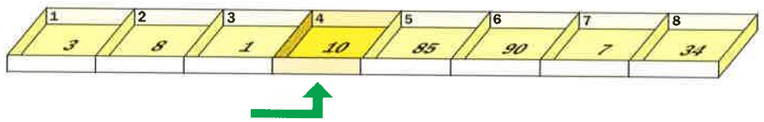
La cassettiera è il vettore, i cassetti sono gli elementi, il numero del cassetto rappresenta la posizione fisica del cassetto all'interno della cassettiera, quindi dell'elemento all'interno del vettore. In ogni cassetto può esserci un valore.

2 Aspetti implementativi dei vettori

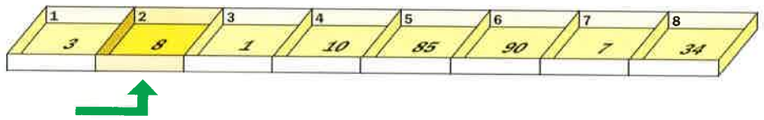
Rappresentiamo graficamente il vettore Vet composto da otto elementi.



Relativamente a questo vettore Vet diciamo che Vet[4] vale 10.



Analogamente Vet[2] vale 8.



Nella nostra trattazione, l'indice potrà essere esclusivamente di tipo intero. Alcuni linguaggi ammettono indici di tipo non numerico, per esempio stringhe. Si parla in questo caso di **hash table**, o di **array associativo**, perché ogni valore stringa utilizzato come indice viene associato a un valore dell'array.
Un esempio:

```
Persona["Nome"] = "Anna";
Persona["Cognome"] = "Verdi";
Persona["Eta"] = 28
```

Come si vede, l'indice dell'array è di tipo stringa. Inoltre, nel vettore Persona vengono memorizzate sia stringhe (il nome e il cognome), sia numeri interi (l'età): non esiste, pertanto, il vincolo di un "tipo base" fissato per tutte le celle dell'array. Questi array prendono il nome di **array densi** e sono gestibili solo da alcuni linguaggi quali PHP, JavaScript e così via, in cui il controllo dei tipi di dato è lasco.

L'organizzazione usata nel vettore è rigida. Ciò comporta alcuni svantaggi nel momento in cui devono essere eseguite alcune manipolazioni sui suoi elementi. Vediamoli.

- **Difficoltà di inserimento o cancellazione**. Sono dovute al fatto che lo spazio della memoria non occupato dal vettore è costituito univocamente dalle posizioni di memoria che precedono il primo elemento del vettore e da quelle che seguono l'ultimo elemento. Di conseguenza l'inserimento di un nuovo elemento nella struttura richiederebbe lo spostamento, mediante riscrittura, di tutti gli elementi che lo seguono; un'operazione analoga sarebbe richiesta nel caso dell'eliminazione di un elemento del vettore, nel caso in cui non si volesse lasciare nessuna posizione di memoria inutilizzata all'interno del vettore stesso.
- **Dimensione statica**: è necessario fissare a priori un limite massimo di elementi che il vettore potrà contenere.

Gli array sono generalmente allocati in aree contigue della memoria del computer. Nel caso in cui il tipo base sia fissato, l'array occuperà un'area di memoria le cui dimensioni complessive sono date dalla dimensione del tipo base moltiplicata per il numero totale degli elementi. Il reperimento di un elemento tramite l'indice avviene sommando all'indirizzo di memoria di partenza dell'array il valore dell'indice moltiplicato per la dimensione del tipo base. Così, se un array di interi (a 2 byte) viene allocato all'indirizzo 500 della memoria, il suo ottavo elemento sarà allocato alla posizione 500 + (2 * 8) = 516.
Questa regola permette al linguaggio un accesso molto efficiente agli elementi del vettore attraverso meccanismi di indirizzamento indiretto, quali sono forniti da tutti i processori e quindi da tutti i linguaggi macchina. Alcuni linguaggi non mantengono in memoria nessun tipo di informazione sulla dimensione di un array, e non sono quindi in grado di eseguire controlli sulla correttezza degli indici utilizzati per accedere a una determinata cella. Ad esempio, nel linguaggio C, se un array di 20 elementi interi (2 byte ciascuno) è allocato all'indirizzo 1000, l'ultima cella valida ha indice 19 e indirizzo 1040; ma se il programmatore, per errore, tenta di accedere alla quarantesima cella (che non esiste), il linguaggio non rileverà l'errore e fornirà accesso alla locazione di memoria 1080, provocando incresciosi malfunzionamenti. Altri linguaggi mantengono in memoria informazioni sulle dimensioni reali degli array e sono quindi in grado di segnalare questo genere di errori senza fornire accesso a elementi erroneamente indirizzati.

3 Come definire un nuovo tipo di dati

Finora abbiamo operato con variabili di tipo semplice o primitivo (ad esempio intero, reale, booleano e così via), mentre ora ci accingiamo a lavorare con **variabili di tipo strutturato**. Attraverso l'utilizzo di queste due strutture, la maggior parte dei linguaggi di programmazione consente di definire nuovi tipi di dati. Anche nel nostro pseudocodice sarà quindi possibile definire dei nuovi tipi personali utilizzabili per dichiarare nuove variabili.
Per creare un nuovo tipo di dati, si utilizza la parola chiave **TIPO**.
La sintassi è la seguente:

```
TIPO <NomeTipo> = <TipoDiDati>
```

Ad esempio, con la seguente dichiarazione di tipo:

```
TIPO Vettore = ARRAY[50] di INTERO
```

definiamo un tipo *Vettore* corrispondente a un vettore di interi di dimensione pari a 50 elementi. Per lavorare con il nuovo tipo *Vettore* dobbiamo dichiarare le variabili. Ad esempio, per dichiararne due scriveremo:

```
VARIABILI
  Vett1, Vett2: VETTORE
```

È da notare che nella dichiarazione di tipo abbiamo utilizzato il simbolo "=", mentre nella dichiarazione delle variabili viene utilizzato il simbolo ":".
Facciamo un altro esempio: consideriamo il vettore *Classe3C* e proviamo a dichiarare più vettori, destinati a contenere i nomi degli studenti che frequentano l'intero corso C. Avremo:

```
TIPO Classe = ARRAY[30] di STRINGA[35]
VARIABILI
  Classe3C, Classe4C, Classe5C: CLASSE
```

dove **Classe** è il nome assegnato al nuovo tipo di dati definito. Per evitare confusioni, ribadiamo la differenza tra dichiarazione di tipo e dichiarazione di variabile. Più volte abbiamo utilizzato la scatola per rappresentare il concetto di variabile. Ricollegandoci a tale concetto, possiamo affermare che la **definizione di tipo** serve per definire la "forma" del contenitore, mentre la **dichiarazione di variabile** serve per creare oggetti aventi una forma precedentemente definita. Per tale motivo, all'interno di un algoritmo è possibile lavorare (cioè leggere, scrivere, assegnare e così via) esclusivamente con le variabili e mai con i tipi.
Riferendoci alle precedenti dichiarazioni, non è corretto scrivere:

```
Classe[I] ← "Rossi"
```

dato che *Classe* non è una variabile ma un tipo; è invece corretto scrivere:

```
Classe3C[I] ← "Rossi"
```

dato che *Classe3C* è una variabile di tipo *Classe*.
È molto importante definire il tipo delle variabili strutturate, in quanto solo in questo modo sarà poi possibile trasferirle facilmente come parametri dal programma chiamante ai sottoprogrammi.

4 Dichiarazione di un vettore in C/C++

Per dichiarare un vettore se ne deve scrivere il tipo, seguito da un nome valido e da una coppia di parentesi al cui interno è racchiusa un'espressione costante, che definisce le dimensioni dell'array. La sintassi generale è dunque la seguente:

```
<Tipo> <NomeArray>[<Dimensione>] [={Valore1, Valore2,....ValoreN}];
```

Ad esempio:

```
int numeri[10];      ← Array non inizializzato di 10 interi
char lettere[20];    ← Array non inizializzato di 20 caratteri
```

In fase di dichiarazione di un array, per definirne la dimensione, non è consentito utilizzare all'interno delle parentesi quadre nomi di variabili; si possono però impiegare delle costanti definite:

```
#define LIMITE_NUMERI 10
#define LIMITE_LETTERE 20
int numeri[LIMITE_NUMERI];
char lettere[LIMITE_LETTERE];
```

Un vettore può essere inizializzato sia esplicitamente, al momento della creazione, fornendo le costanti di inizializzazione dei dati, sia durante l'esecuzione del programma, assegnandogli o copiandovi dati. Per inizializzare l'array *numeri* degli esempi precedenti, in fase di creazione, con dieci numeri interi scriveremo:

```
int numeri[10] = {12, 0, 4, 150, 4500, 2, 34, 5599, 22, 83};
          ↑ Array dimensionato
```

Oppure anche:

```
int numeri[ ] = {12, 0, 4, 150, 4500, 2, 34, 5599, 22, 83};
          ↑ Array non dimensionato
```

In questo secondo modo, viene creato un array di dimensione pari al numero di elementi inseriti.

Per quanto detto, la dichiarazione del vettore *PesoAlunni* risulta:

```
float PesoAlunni[20];
```

che lo dichiara come un vettore di 20 elementi tutti di tipo **float**.
Nota che ogni elemento dell'array può essere utilizzato come se fosse una singola variabile:

```
PesoAlunni[3]=61.9;   ← Al quarto elemento del vettore PesoAlunni è assegnato il valore 61.9
```

Per accedere ai singoli elementi del vettore, basta dunque specificare un diverso valore dell'indice; ed essendo questo un valore intero, lo si può tranquillamente incrementare all'interno di un ciclo.

In C/C++ il passaggio di un array a una funzione avviene sempre per indirizzo e mai per valore. Quando si passa un array, in effetti, non si fa altro che passare l'indirizzo (il puntatore) del suo primo elemento. Ciò significa che, se all'interno della funzione vengono modificati i valori dell'array, tale modifica avrà effetto anche sull'array passato alla funzione.

5 Operazioni di caricamento sui vettori

Un vettore, in un algoritmo, non può mai essere manipolato come se fosse una cosa sola; si deve sempre operare sui singoli elementi. A livello logico, però, si possono definire delle operazioni (alle quali corrispondono dei sottoprogrammi) proprie di questo tipo di strutture.
Esaminiamone alcune, per le quali ci serviremo del seguente tipo di dati:

```
TIPO Vettore = ARRAY[10] di INTERO
```


Il caricamento del vettore

Il caricamento è l'operazione che consente di assegnare un valore agli elementi del vettore.

L'algoritmo risolutivo è il seguente:

PSEUDOCODICE

```
ALGORITMO CaricaVettore
TIPO Vettore = ARRAY[10] di INTERO
Main()
VARIABILI
  V: Vettore
  NumElementi: INTERO
INIZIO
  NumElementi ← OttieniDimensione()
  Caricamento(V, NumElementi)
FINE
FUNZIONE OttieniDimensione(): INTERO
VARIABILI
  Dim: INTERO
INIZIO
  RIPETI
    SCRIVI("Inserisci il numero di elementi")
    LEGGI(Dim)
  FINCHÉ ((Dim > 0) AND (Dim < 10))
  RITORNO(Dim)
FINE
PROCEDURA Caricamento(REF Vet: Vettore; VAL N: INTERO)
VARIABILI
  I: INTERO
INIZIO
  PER I ← 1 A N ESEGUI
    SCRIVI("Inserisci il ", I, "° , elemento")
    LEGGI(Vet[I])
  FINEPER
FINE
```

C++

```
#include <iostream>
using namespace std;
// prototipi
int OttieniDimensione(void);
void Caricamento(int [], int);
int main()
{
  int V[10]; // dichiarazione vettore
  int NumElementi;
  NumElementi=OttieniDimensione();
  Caricamento(V,NumElementi);
  for(int i=0;i<NumElementi;i++)
    cout<< V[i] <<endl;
  system("PAUSE");
  return 0;
}
int OttieniDimensione()
{
  int Dim;
  do
  {
    cout << "inserisci il numero di elementi ";
    cin >> Dim;
  }
  while (Dim<=0 || Dim >=10);
  return Dim;
}
void Caricamento(int Vet[], int N)
{
  int i;
  for (i=0; i<N; i++)
  {
    cout << "inserisci il " << i+1 << " elemento ";
    cin >> Vet[i];
  }
}
```

Quando abbiamo dichiarato un vettore in C/C++ abbiamo visto che il suo primo elemento ha indice pari a zero. Per semplicità didattica, e per comprendere l'analogia tra un vettore e un oggetto reale (come ad esempio una cassettiera), in pseudocodifica il primo elemento del vettore avrà indice 1.

```
    ALLORA
        IndiceMax ← I
    FINESE
    SE(Vet[I] < Vet[IndiceMin])
    ALLORA
        IndiceMin ← I
    FINESE
    I ← I + 1
  FINEMENTRE
FINE
```

C++

```
#include <iostream>
using namespace std;
//prototipi
int OttieniDimensione(void);
void Caricamento(int [], int);
void TrovaMaxMin(int [], int, int &, int &);
int main()
{
  int V[10]; // dichiarazione vettore
  int NumElementi, IndiceMassimo=0,IndiceMinimo=0;
  NumElementi=OttieniDimensione();
  Caricamento(V,NumElementi);
  TrovaMaxMin(V,NumElementi,&IndiceMassimo,&IndiceMinimo);
  cout << "La componente massima si trova in posizione "<< IndiceMassimo+1 << endl;
  cout << "La componente minima si trova in posizione "<< IndiceMinimo+1 << endl;
  system("PAUSE");
  return 0;
}

int OttieniDimensione()
{
  int Dim;
  do
  {
    cout << "inserisci il numero di elementi ";
    cin >> Dim;
  }
  while (Dim<=0 || Dim >=10);
  return Dim;
}
void Caricamento(int Vet[], int N)
{
  int i;
  for (i=0; i<N; i++)
  {
    cout << "inserisci il " << i+1 << " elemento ";
    cin >> Vet[i];
  }
```

```
inserisci il numero di elementi 5
inserisci il 1 elemento 2
inserisci il 2 elemento 6
inserisci il 3 elemento 8
inserisci il 4 elemento 1
inserisci il 5 elemento 4
La componente massima si trova in posizione 3
La componente minima si trova in posizione 4
Premere un tasto per continuare . . .
```



```
}
void TrovaMaxMin(int Vet[], int N, int &IndiceMax, int &IndiceMin)
{
  int i;
  for (i=0; i<N; i++)
  {
    if(Vet[i]>Vet[IndiceMax])
      IndiceMax=i;
    if(Vet[i]<Vet[IndiceMin])
      IndiceMin=i;
  }
}
```

Quando il vettore è passato come parametro non deve essere indicata la dimensione ma solo il nome del parametro seguito da una coppia di parentesi quadre.
Nella procedura *TrovaMaxMin*, infatti, puoi notare il passaggio dell'array indicato con *int Vet[]*, inoltre nel prototipo, poiché non vengono indicati i nomi dei parametri ma solo il tipo, la sintassi corretta è solo *int []*.

2. Scrivere un programma che chieda in input una serie di numeri positivi (finché non viene inserito –1) e ne calcoli la somma e la media. Il numero inserito deve essere compreso tra 1 e 1000.

C++

```
#include <iostream>
using namespace std;
int main()
{
  int valori[200];
  int i, input,numeri=0;
  long somma;
  double media;
  /* ripetiamo il ciclo fino a quando viene inserito -1 */
  /* un valore non valido non viene inserito nel vettore */
  do
  {
    cout << "Inserire il numero" << numeri + 1 <<": ";
    cin >> input;
    if(input>0 && input<1000)
    valori[numeri++]=input;
  } while(input !=-1);
  somma=0;
  for(i = 0; i < numeri; i++)
    somma+=valori[i];
  media=(double)somma / numeri;
  cout << "Numero valori inseriti = " << numeri << endl;
  cout << "Somma = " << somma <<" media ="<< media << endl;
  //system("PAUSE");
  return 0;
}
```

```
Inserire il numero 1: 5
Inserire il numero 2: 8
Inserire il numero 3: 2
Inserire il numero 4: 4
Inserire il numero 5: 3
Inserire il numero 6: -1
Numero valori inseriti = 5
Somma = 22 media =4.4
Premere un tasto per continuare . . .
```

Poiché il vettore deve essere necessariamente dimensionato in fase di dichiarazione, si ipotizza un valore che riesca a coprire il numero massimo di elementi che il programma prevede di gestire: è la variabile *numeri* che effettivamente tiene il conto degli elementi inseriti nel vettore.

```
PSEUDOCODICE
PROCEDURA Sorpresa
VARIABILI
  I: INTERO
INIZIO
  I ← 2
  MENTRE(I ≤ 8) ESEGUI
    SCRIVI(Lettere[I])
    I ← I + 2
  FINEMENTRE
FINE
```

9 Che cosa realizza il seguente spezzone di pseudocodice? Codifica in C/C++.

```
PSEUDOCODICE
TIPO Vettore = ARRAY[50] di INTERO
VARIABILI
  Vet:Vettore
  N: INTERO
PROCEDURA ComponiVet3(VAL Vet1:Vettore; VAL Vet2:Vettore; REF Vet3: Vettore; VAL N:INTERO)
VARIABILI
  I, Resto: INTERO
INIZIO
  I ← 1
  MENTRE(I ≤ N) ESEGUI
    Resto ← I MOD 2
    SE(Resto = 0)
      ALLORA
        Vet3[I] ← Vet1[I]
      ALTRIMENTI
        Vet3[I] ← Vet2[I]*3
    FINESE
    I ← I + 1
  FINEMENTRE
FINE
```

6 | Lo shift degli elementi

Considera il seguente esempio. Vai al cinema con un gruppo di amici. Dopo esserti seduto, noti che un tuo amico non è vicino alla sua ragazza. Decidi, quindi, di far scalare ognuno di un posto, in modo da liberare la poltrona vicina al tuo amico. Hai effettuato uno shift.

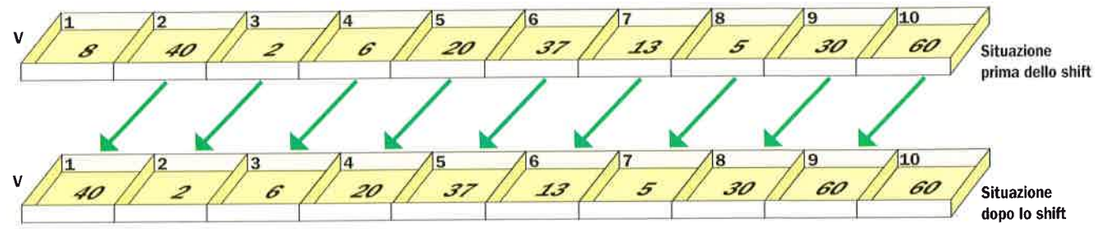
Lo **shift** è l'operazione che consente di spostare di una posizione (a destra o a sinistra) tutti gli elementi del vettore (**shift completo**) o solo una parte di essi (**shift parziale**).

Possiamo quindi effettuare uno:

- shift completo a sinistra;
- shift completo a destra;
- shift parziale a sinistra;
- shift parziale a destra.

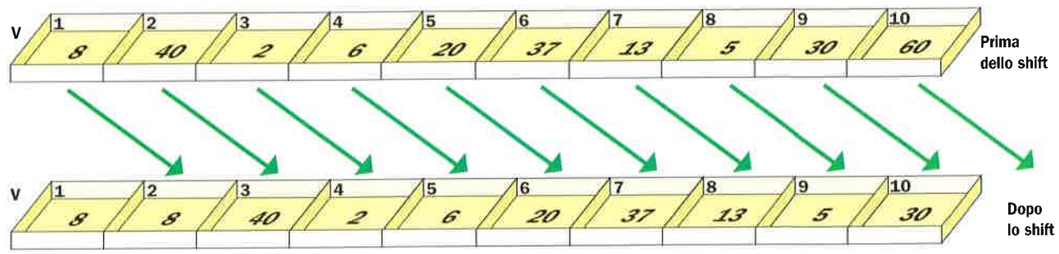
Nello **shift completo a sinistra**, tutti gli elementi presenti nel vettore saranno spostati di una posizione verso sinistra (il contenuto della seconda posizione verrà inserito nella prima, il contenuto della terza sarà a sua volta spostato nella seconda, e così via).
In tal modo, l'elemento presente nella prima posizione sarà perduto, mentre quello presente nell'ultima si presenterà due volte.

Analizziamo la seguente figura:




```
PSEUDOCODICE
ALGORITMO Sposta
TIPO Vettore = ARRAY[10] di INTERO
Main()
VARIABILI
  V: Vettore
INIZIO
  ShiftSinistro(V, 10)
FINE
PROCEDURA ShiftSinistro(REF Vet: Vettore; VAL N: INTERO)
VARIABILI
  I: INTERO
INIZIO
  PER I ← 1 A N - 1 ESEGUI
    Vet[I] ← Vet[I + 1]
  FINEPER
FINE
```

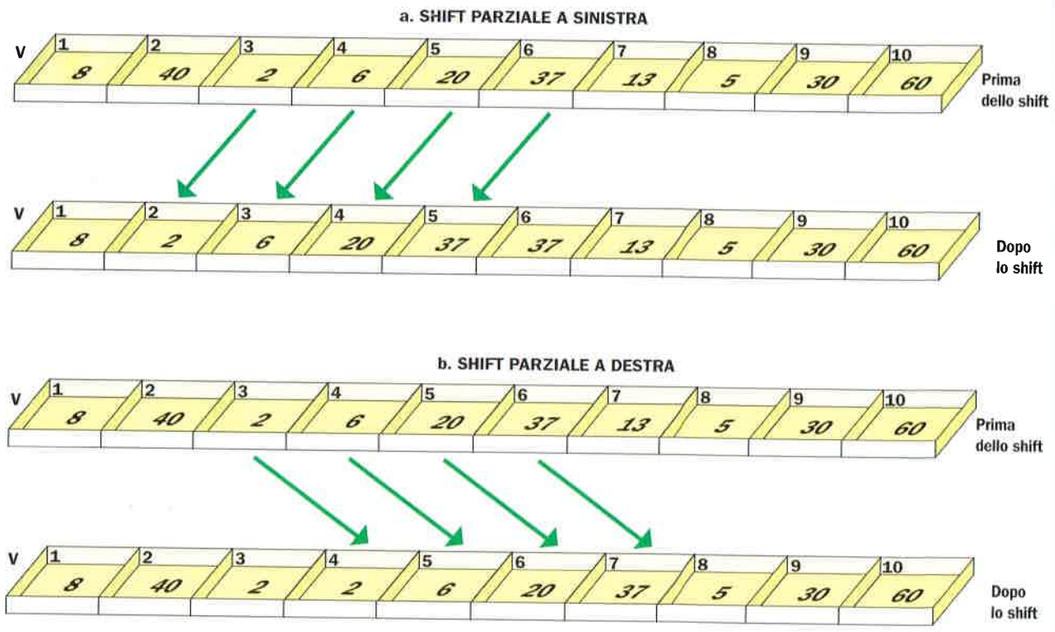
Rispetto all'operazione descritta precedentemente, nello **shift completo a destra** varia solo l'ordine in cui vengono eseguite le assegnazioni. Sarà quindi il contenuto dell'ultima posizione a essere perso, mentre quello della prima si presenterà due volte.



E ora l'algoritmo:

```
PSEUDOCODICE
PROCEDURA ShiftDestro(REF Vet: Vettore; VAL N: INTERO)
VARIABILI
  I: INTERO
INIZIO
  PER I ← N INDIETRO A 2 ESEGUI
    Vet[I] ← Vet[I - 1]
  FINEPER
FINE
```

Lo **shift parziale sinistro o destro** si esegue quando occorre spostare solo alcuni elementi adiacenti del vettore. Per fare questo, è indispensabile conoscere gli indici estremi del gruppo che si intende spostare. Analizziamo la seguente figura, che rappresenta i grafici dei due shift dei quali hai appena letto la descrizione.



Nel seguente codice C++ abbiamo implementato una versione completa di entrambi gli shift con relativo caricamento e stampa del vettore.

```
C++
#include <iostream>
using namespace std;
// prototipi di funzioni
void Caricamento(int [], int);
void ShiftSinistro(int [], int);
void ShiftDestro(int [], int);
void Stampa(int [], int);
```



```
int main()
{
  int V[10];  // dichiarazione del vettore
  Caricamento(V,10);
  ShiftSinistro(V,10);
  cout << "Shift sinistro" << endl;
  Stampa(V,10);
  ShiftDestro(V,10);
  cout << "Shift destro" << endl;
  Stampa(V,10);
  system("PAUSE");
  return 0;
}
void Caricamento(int Vet[], int N)
{
  int i;
  for (i=0; i<N; i++)
  {
    cout << "inserisci il " << i+1 << " elemento ";
    cin >> Vet[i];
  }
}
void ShiftSinistro(int Vet[], int N)
{
  int i;
  for (i=0; i<N-1; i++)
   Vet[i]=Vet[i+1];
}
void ShiftDestro(int Vet[], int N)
{
  int i;
  for (i=N; i>0; i--)
   Vet[i]=Vet[i-1];
}
void Stampa(int Vet[], int N)
{
  int i;
  for (int i=0; i<10; i++)
  cout << Vet[i]<< endl;
}
```

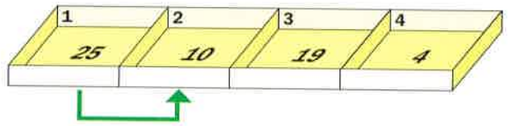
```
inserisci il 1 elemento 8
inserisci il 2 elemento 9
inserisci il 3 elemento 4
inserisci il 4 elemento 1
inserisci il 5 elemento 2
inserisci il 6 elemento 3
inserisci il 7 elemento 10
inserisci il 8 elemento 12
inserisci il 9 elemento 80
inserisci il 10 elemento 18
Shift sinistro
9
4
1
2
3
10
12
80
18
18

Shift destro
9
9
4
1
2
3
10
12
80
18
Premere un tasto per continuare . . .
```

7 L'ordinamento per selezione

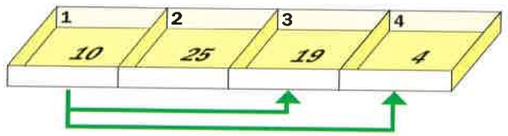
L'**ordinamento** è il processo che permette di ottenere, a partire da un insieme di dati omogenei, un insieme ordinato secondo un ordine crescente o decrescente.

Il più intuitivo e inefficace metodo di ordinamento è quello **per selezione**, detto anche **ordinamento ingenuo**, che consiste nel confrontare ciascun elemento con tutti quelli di indice superiore,

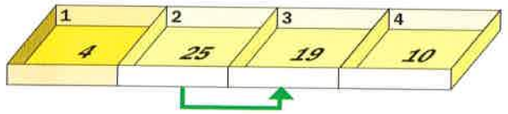
scambiandone il contenuto quando si verifica che il valore dell'elemento preso in considerazione risulta maggiore di quello con cui viene confrontato (nel caso di ordinamento crescente).
Per comprendere meglio la tecnica di questo tipo di ordinamento, proviamo a ordinare il seguente vettore in ordine crescente:



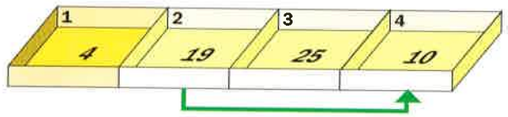
Confrontiamo il primo elemento con il secondo: poiché 25 è maggiore di 10, si effettua lo scambio. Otteniamo:



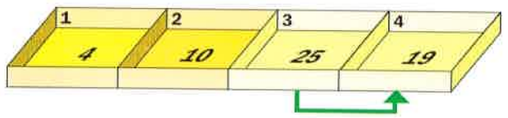
Confrontiamo il primo elemento (che questa volta è 10) con il terzo: poiché 10 è minore di 19, non effettuiamo alcuno scambio. Confrontiamo, quindi, il primo elemento con il quarto: poiché 10 è maggiore di 4 effettuiamo lo scambio. Otteniamo:



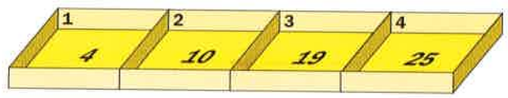
Con la fine della prima scansione abbiamo posizionato l'elemento di valore più piccolo in prima posizione. Continuiamo, ma questa volta partiamo dal secondo elemento e confrontiamolo con tutti gli altri (ossia con il terzo e con il quarto). Poiché 25 è maggiore di 19, si effettua lo scambio. Otteniamo:



Continuiamo e confrontiamo il secondo elemento con il quarto: 19 è maggiore di 10, pertanto eseguiamo uno scambio. Otteniamo:



Abbiamo terminato la seconda scansione e abbiamo posizionato in seconda posizione il secondo elemento di valore più piccolo tra quelli esaminati. Ora consideriamo il terzo elemento e riprendiamo la scansione. Troviamo nuovamente 19 e 25. Poiché 25 è maggiore di 19 effettuiamo ancora lo scambio.
Otteniamo:



Non ci sono altri elementi da confrontare. Termina la terza e ultima scansione: essendo, infatti, il vettore costituito da N = 4 elementi, abbiamo effettuato esattamente N – 1 = 3 scansioni. Il vettore è ora ordinato.

Vediamo l'algoritmo:

PSEUDOCODICE

```
PROCEDURA OrdinamentoIngenuo(REF Vet: Vettore; VAL N: INTERO)
VARIABILI
  I, J, Comodo: INTERO
INIZIO
  PER I ← 1 A N – 1 ESEGUI
    PER J ← I + 1 A N ESEGUI
      SE(Vet[I] > Vet[J])
        ALLORA
          Comodo ← Vet[I]
          Vet[I] ← Vet[J]
          Vet[J] ← Comodo
      FINESE
    FINEPER
  FINEPER
FINE
```

C++

```
void OrdinamentoIngenuo(int Vet[], int N)
{
  int i,j,comodo;
  for (i=0; i<N-1;i++)
    for(j=i+1; j<N; j++)
      if(Vet[i]>Vet[j])
        {
          comodo=Vet[i];
          Vet[i]=Vet[j];
          Vet[j]=comodo;
        }
}
```

Nella prima esecuzione del ciclo interno, per I = 1, si confronta il primo elemento con tutti gli altri scambiando il valore dei due elementi se Vet [1] > Vet [J]. In questo modo, al termine del ciclo Vet[1] conterrà l'elemento più piccolo.
Alla seconda iterazione del ciclo esterno si ripete la stessa procedura: alla fine Vet[2] conterrà il secondo valore in ordine di grandezza. Quando termina tutto il ciclo esterno, il vettore risulterà ordinato.
Osserviamo che per I = 1 l'istruzione del ciclo interno:

```
SE(Vet [I] > Vet [J])
  ALLORA
    .....
```

viene eseguita (N – 1) volte, per I = 2 viene eseguita per (N – 2) volte, per I = 3 viene eseguita per (N – 3) volte e così via fino a I = (N – 1), in cui l'istruzione viene eseguita un'unica volta (il confronto fra Vet[N – 1] e Vet[N]).
In tutto il numero dei confronti effettuati è:

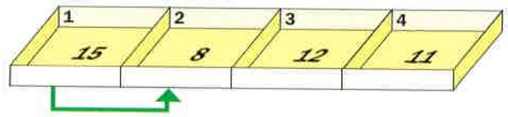
$$(N-1) + (N-2) + (N-3) + \ldots + 2 + 1$$

cioè:

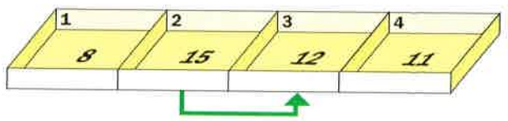
$$\frac{N*(N-1)}{2}$$

8 L'ordinamento a bolle

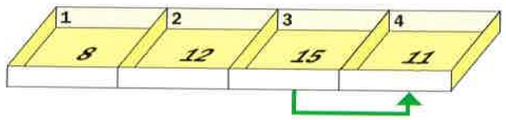
Il metodo di **ordinamento bubble sort** o **a bolle** consiste nel confronto degli elementi scambiandoli di posto, se necessario, a due a due, e cioè primo e secondo, secondo e terzo, ..., penultimo e ultimo.
Questa ripetizione di scambi fra elementi adiacenti permette di far salire verso l'alto, proprio come bolle di sapone, gli elementi più grandi (o più piccoli in caso di ordinamento decrescente); da qui il nome **bubble sort**.
La soluzione è quella di ripetere i confronti ripartendo dal primo elemento, tante volte quante sarà necessario. Se durante una scansione non si effettuano confronti, significa che il vettore è ordinato, altrimenti si continua a confrontare sino alla posizione in cui è avvenuto l'ultimo scambio.
Proviamo a ordinare il seguente vettore di quattro elementi secondo l'ordine crescente:



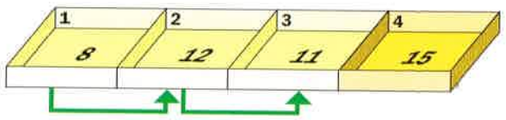
Confrontiamo il primo elemento con il secondo. Poiché 15 è maggiore di 8, effettuiamo lo scambio. Otteniamo:



Ora confrontiamo il secondo con il terzo. Poiché 15 è maggiore di 12, dobbiamo effettuare un ulteriore scambio. Otteniamo:

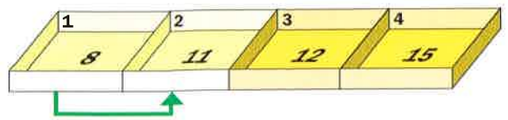


E ora confrontiamo il terzo con il quarto. Anche in questo caso dobbiamo effettuare uno scambio. Otteniamo:

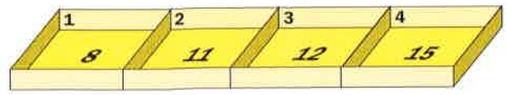


L'elemento più grande è salito pian piano verso l'alto. Ora ricominciamo la scansione, ma questa volta sino al penultimo elemento.

Confrontiamo il primo elemento con il secondo. Poiché 8 è minore di 12, non dobbiamo fare scambi. Confrontiamo il secondo con il terzo e, poiché 12 è maggiore di 11, facciamo lo scambio. Otteniamo:



Abbiamo terminato la seconda scansione e il secondo elemento più grande è migrato in alto verso la penultima posizione. Cominciamo la terza scansione continuando sino al terzultimo elemento. Considerato che 8 è minore di 11 non facciamo alcuno scambio. Il procedimento termina e **il vettore è ordinato**.



Riflettiamo. Se durante una scansione non si effettuano scambi (come nella nostra terza scansione), significa che il vettore è ordinato e quindi non è più necessario effettuare altri confronti.
Ora esaminiamo l'algoritmo. Ci serviremo:

- di una variabile booleana di nome *Continua* che ci segnala se sono avvenuti scambi;
- di una variabile di nome *Sup* (rappresentante il limite superiore) che viene inizializzata a *N* (numero di elementi del vettore);
- di una variabile di nome *K* che, dopo l'esecuzione di ogni ciclo *PER*, indicherà la posizione del nuovo estremo superiore (l'ultima in cui è avvenuto uno scambio).

PSEUDOCODICE

```
PROCEDURA BubbleSort(REF Vet: Vettore; VAL N: INTERO)
VARIABILI
  K, Sup, Comodo, I: INTERO
  Continua: BOOLEANO
INIZIO
  K ← N
  Continua ← VERO
  MENTRE(Continua) ESEGUI
    Sup ← K
    Continua ← FALSO
    PER I ← 1 A Sup - 1 ESEGUI
      SE(Vet[I] > Vet[I + 1])
        ALLORA
          Comodo ← Vet[I]
          Vet[I] ← Vet[I + 1]
          Vet[I + 1] ← Comodo
          Continua ← VERO
          K ← I
      FINESE
    FINEPER
  FINEMENTRE
FINE
```

C++

```
void BubbleSort(int Vet[], int N)
{
  int i,k,Sup,comodo;
  bool Continua=true;
  k=N;
  while(Continua)
   { Sup=k;
     Continua=false;
     for (i=0; i<Sup-1;i++)
       if(Vet[i]>Vet[i+1])
       {
         comodo=Vet[i];
         Vet[i]=Vet[i+1];
         Vet[i+1]=comodo;
         Continua=true;
         k=i;
       }
   }
}
```

Proponiamo inoltre una versione ricorsiva in cui la procedura BubbleSort richiama se stessa per effettuare l'ordinamento del vettore.

C++

```
#include <iostream>
using namespace std;
#define D 10
//prototipi di funzioni
void Caricamento(int [], int);
void BUBBLESORT_Ric (int [], int );
void swap (int [], int i, int j);
void Stampa(int [], int);

int main( )
{
    int V[D];  //dichiarazione del vettore
    Caricamento(V,D);
    cout << "Vettore iniziale" << endl;
    Stampa(V,D);
    // chiama Procedura di ordinamento ricorsiva
    BUBBLESORT_Ric (V, D);
    cout << "Array ordinato BUBBLE SORT ricorsivo: " << endl;
    Stampa(V,D);
    system("PAUSE");
    return 0;
}
void Caricamento(int Vet[], int N)
{
  int i;
  for (i=0; i<N; i++)
    {
      cout << "inserisci il " << i+1 << " elemento ";
      cin >> Vet[i];
    }
}

void BUBBLESORT_Ric (int m[], int n)
{
  int k=0;
  int i;
  for(i=0; i<n-1; i++)
    if(m[i] > m[i+1])
      {
        swap(m,i,i+1);
        k = i;
      }
  if( k > 0 )
  BUBBLESORT_Ric (m, k+1);
}

void swap (int m[], int i, int j)
{
  int temp;
  /* Scambio delle due variabili */
  temp = m[i];
  m[i] = m[j];
  m[j] = temp;
}

void Stampa(int Vet[], int N)
{
  int i;
  for (int i=0; i<N; i++)
    cout << Vet[i]<< endl;
}
```

```
inserisci il 1 elemento 9
inserisci il 2 elemento 8
inserisci il 3 elemento 7
inserisci il 4 elemento 1
inserisci il 5 elemento 2
inserisci il 6 elemento 3
inserisci il 7 elemento 6
inserisci il 8 elemento 4
inserisci il 9 elemento 5
inserisci il 10 elemento 10
Vettore iniziale
9
8
7
1
2
3
6
4
5
10

Array ordinato BUBBLE SORT ricorsivo:
1
2
3
4
5
6
7
8
9
10
Premere un tasto per continuare . . .
```

L'ordinamento a bolle, infine, può ordinare un vettore anche in ordine decrescente; l'unica modifica da apportare è la sostituzione dell'operatore < con l'operatore > nel confronto tra Vet[I] e Vet[I + 1].

9 Il problema della ricerca

La ricerca sequenziale

Il metodo della **ricerca sequenziale** non richiede che i dati siano ordinati. Consiste in una serie di confronti tra il valore *X* da ricercare e tutti gli elementi del vettore.

I confronti possono terminare quando si trova l'elemento cercato, o possono anche continuare sino alla fine del vettore, nel caso in cui si desideri conteggiare il numero di volte che il valore X compare all'interno del vettore (in tal caso si parla di **scansione sequenziale**, o **ricerca completa**, e non di ricerca sequenziale).

Riportiamo di seguito il semplice algoritmo di ricerca sequenziale, che termina nel momento in cui il valore *X* viene trovato.

PSEUDOCODICE

```
FUNZIONE RicercaSequenziale(VAL Vet: Vettore; VAL N: INTERO; VAL X: INTERO): BOOLEANO
VARIABILI
  Trovato: BOOLEANO
  I: INTERO
INIZIO
  Trovato ← FALSO
  I ← 1
  MENTRE((I ≤ N) AND (NOT Trovato)) ESEGUI
    SE(Vet[I] = X)
      ALLORA
        Trovato ← VERO
    FINESE
    I ← I + 1
  FINEMENTRE
  RITORNO(Trovato)
FINE
```

La funzione che implementa la ricerca sequenziale fa uso di una variabile booleana di nome Trovato (così chiamata proprio per fare riferimento al suo significato intrinseco). Tale variabile assume inizialmente il valore Falso, ma nel momento in cui il valore richiesto viene trovato, assume il valore Vero. Considerato che la funzione che abbiamo appena implementato restituisce un valore booleano, all'interno del programma chiamante potrà essere richiamata con una pseudoistruzione del tipo:

```
SE(RicercaSequenziale(V, Dim, Elemento) = VERO)
```

10 La ricerca binaria

Il metodo della **ricerca binaria** o **dicotomica** richiede che gli elementi del vettore siano ordinati. Un esempio: dobbiamo cercare un nominativo in un elenco telefonico (il nostro vettore) di una sola città. Nell'elenco i nomi sono disposti in ordine alfabetico (crescente). Possiamo:

- aprire l'elenco a una pagina che rappresenta più o meno la metà del volume;
- guardare il primo nome in alto. Se il nominativo che stiamo cercando si trova *dopo* quello appena letto ("è maggiore" in ordine alfabetico), scarteremo la metà sinistra della guida, perché il nostro nominativo si troverà sicuramente nella parte destra. Ripeteremo il procedimento, su questa parte di volume, partendo dal passaggio precedente. Ovviamente, se il nominativo che stiamo cercando si trova *prima* di quello che appare in testa alla pagina ("è minore" della parola cercata in ordine alfabetico), considereremo la parte sinistra.

Proseguiremo fino a quando il nome cercato sarà trovato o scopriremo che non è presente nell'elenco.

Nominativo da ricercare:

VERDI GIANNI

Ragioniamo in termini di vettore di *N* elementi in ordine crescente. Il procedimento è il seguente: innanzitutto bisogna identificare l'elemento che occupa la posizione centrale del vettore e confrontare questo elemento con quello X cercato. Si possono verificare tre casi:

1. L'elemento *X* è **più piccolo** dell'elemento centrale: si scartano tutti gli elementi che occupano la metà destra del vettore. È come se, ora, l'ultimo elemento del vettore fosse quello precedente a N/2.
2. L'elemento *X* è **più grande** dell'elemento centrale: si scartano tutti gli elementi che occupano la metà sinistra del vettore. È come se, ora, il primo elemento del vettore fosse quello successivo a N/2.
3. L'elemento *X* è **uguale** all'elemento centrale: *X* appartiene all'insieme e si trova nella posizione centrale.

Per i casi 1 e 2 il discorso va ripetuto sino a quando si verifica il caso 3, oppure, rimanendo un solo elemento da confrontare nel vettore, l'elemento *X* risulta diverso, quindi non è presente nel vettore.

PSEUDOCODICE

```
FUNZIONE RicercaBinaria(VAL Vet: Vettore; VAL N: INTERO; VAL X: INTERO): BOOLEANO
VARIABILI
  Primo, Ultimo, Centro: INTERO
  Trovato: BOOLEANO
INIZIO
  Primo ← 1
  Ultimo ← N
  Trovato ← FALSO
  MENTRE((Primo ≤ Ultimo) AND (NOT Trovato)) ESEGUI
    Centro ← (Primo + Ultimo) DIV 2
    SE(Vet[Centro] = X)
      ALLORA
        Trovato ← VERO
      ALTRIMENTI
        SE(Vet[Centro] < X)
          ALLORA
            Primo ← Centro + 1
          ALTRIMENTI
            Ultimo ← Centro - 1
        FINESE
    FINESE
  FINEMENTRE
  RITORNO(Trovato)
FINE
```

```C++
bool RicercaBinaria (int Vet[], int N, int X)
{
  int Primo,Ultimo,Centro;
  bool trovato;
  Primo=1;
  Ultimo=N;
  while (Primo<=Ultimo && !trovato)
  {
      Centro=(Primo+Ultimo)/2;
        if (Vet[Centro]==X)
          trovato=true;
        else if(Vet[Centro]<X)
          Primo=Centro+1;
        else
          Ultimo=Centro-1;
  }
  return trovato;
}
```

L'acquisizione di una stringa attraverso la funzione **cin** non termina quando l'utente preme il tasto invio, ma **al primo spazio incontrato**. Se dunque la stringa contiene degli spazi, sarà acquisita solo in parte. Per ovviare a questa incresciosa situazione, si ricorre alla funzione:

```
gets(stringa);
```

la quale acquisisce all'interno del vettore stringa tutti i simboli inseriti fino alla pressione del tasto invio, spazi compresi. Esiste una funzione omologa a **gets** per la fase di stampa:

```
puts(<stringa>)
```

Tale funzione sostituisce **cout**, risulta più efficiente in termini di velocità, ma non permette alcun tipo di formattazione sulla stampa del messaggio finale.

11 | Le stringhe

Un vettore può essere anche di tipo **char** e contenere caratteri. Di fatto, un vettore di caratteri è una **stringa,** ossia una sequenza contenente qualsiasi simbolo.

Il C/C++ non prevede un tipo primitivo per la gestione delle stringhe, ma come sappiamo gestisce il tipo carattere e possiamo dunque sopperire a questa mancanza con l'aiuto degli array. La sintassi per creare una stringa coincide pertanto con la normale dichiarazione di un vettore di tipo **char**:

Vettore non inizializzato: stringa vuota

```C++
char stringa[20];
```

se invece, oltre che dichiarare il vettore, vogliamo anche inizializzarlo, abbiamo due possibilità; la prima è, come ci aspettiamo, un elenco di caratteri tra parentesi graffe, ognuno fra apici singoli:

```C++
char stringa[]={'G','a','b','r','i','e','l','e','\0'};
```

Nota però che l'ultimo elemento del vettore è la sequenza speciale **\0**, considerata un unico carattere, che **denota la fine della stringa**: qualsiasi stringa termina con esso, anche se non è espressamente indicato.
Esso va dunque tenuto presente anche quando si valuta la dimensione della stringa; che nel caso precedente, ad esempio, ha valore 9.

Fortunatamente vi è anche un'altra possibilità per inizializzare un vettore di caratteri:

```C++
char stringa[]="Gabriele";
```

Nota però che anche con questa sintassi, sicuramente più immediata e compatta, viene dichiarato un vettore di 9 elementi (l'ultimo dei quali è \0).

Indirizzo in memoria	Contenuto elemento	Carattere rappresentato	Accesso all'elemento
00300	71	G	nome[0]
00301	97	a	nome[1]
00302	98	b	nome[2]
00303	114	r	nome[3]
00304	105	i	nome[4]
00305	101	e	nome[5]
00306	108	l	nome[6]
00307	101	e	nome[7]
00308	0	\0	nome[8]

In conclusione, il seguente programma dichiara e inizializza il vettore stringa con il testo completo, il quale verrà ripartito nelle singole celle del vettore e i cui valori saranno stampati come singoli elementi.

```C++
#include <iostream>
using namespace std;
int main()
{
  int i;
  char stringa[]="Gabriele";
  for (i=0; i<9;i++)
    cout << stringa[i] << endl;
  system("PAUSE");
  return 0;
}
```

Quando vi è la necessità di manipolare la stringa come un vettore, accedendo direttamente ai singoli elementi, il confronto con la fine della stringa deve essere fatto riferendosi non alla sua dimensione, ma all'unico elemento che ne segnala con certezza la fine: il carattere speciale \0.

OSSERVA COME SI FA

1. **Realizziamo un programma che preveda l'acquisizione di un'espressione algebrica e stampi il numero di parentesi tonde, quadre e graffe in essa presenti.**

```C++
#include <iostream>
using namespace std;
int main()
{
  char espressione[30];
  int pargraffe, parquadre, partonde, c;
  cout << "Espressione algebrica ";
  gets(espressione);
```

```
  pargraffe = parquadre = partonde = 0;
  for(c=0;c<30 && espressione[c]!='\0';c++)
  {
    switch(espressione[c])
    {
      case '{': case '}':
        pargraffe++;
        break;
      case '[': case ']':
        parquadre++;
        break;
      case '(': case ')':
        partonde++;
        break;
    }
  }
  cout << "Graffe = " << pargraffe << endl;
  cout << "Quadre = " << parquadre << endl;
  cout << "Tonde = " << partonde << endl;
  system("PAUSE");
  return 0;
}
```

```
Espressione algebrica (2x[1-(2+8)x3]-2)
Graffe = 2
Quadre = 2
Tonde = 2
Premere un tasto per continuare . . .
```

Se l'*espressione* ha più di 30 caratteri, i restanti non verranno acquisiti, come non sarà acquisito neppure il carattere di fine stringa: e ciò provocherà inevitabilmente un errore a livello di sistema operativo durante l'esecuzione.

12 Operazioni con le stringhe

Poiché in C/C++, come si è detto, non vi è un tipo "stringa" predefinito, non esistono neppure delle operazioni disponibili per la loro gestione. Non è dunque possibile, ad esempio, scrivere frammenti di codice come i seguenti:

```
char Benvenuto[30];
Benvenuto="Buongiorno a tutti";
if (Benvenuto== "Buona giornata") ...
```

Realizzare istruzioni equivalenti a queste utilizzando la logica pura dei vettori sarebbe molto laborioso. Esistono però delle librerie standard che ci consentono di risolvere facilmente problemi come questi, almeno per quanto riguarda assegnazione e confronto fra stringhe: si tratta di **string.h** e, in parte, **stdlib.h**.

La tabella a pagina seguente riporta le principali funzioni disponibili al loro interno utili nella gestione delle stringhe.

Funzione	Descrizione	Tipo restituito	Header
strcpy(<Str1>, <Str2>**);**	copia la stringa *Str2* all'interno di *Str1*; la dimensione di *Str1* deve essere almeno pari a quella di *Str2*	array di char	string.h
strncpy(<Str1>, <Str2>,<n>**);**	copia al massimo *n* caratteri della stringa *Str2* in *Str1*	array di char	string.h
strcat(<Str1>, <Str2>**);**	concatena la stringa *Str2* alla fine di *Str1*; la dimensione di *Str1* deve essere tale da poter contenere anche *Str2*.	array di char	string.h
strlen(<Str>**);**	conta il numero dei caratteri nella stringa *Str*, di cui restituisce la lunghezza	unsigned int	string.h
strcmp(<Str1> <Str2>**);**	confronta, carattere per carattere, le due stringhe *Str1* e *Str2* e restituisce un valore intero: • 0, se *Str1* = *Str2*; • < 0, se *Str1* precede *Str2*; • > 0, se *Str1* segue *Str2*.	int	string.h
atof(<Str>**)**	converte la stringa *Str* in numero in virgola mobile	double	stdlib.h
atoi(<Str>**)**	converte la stringa *Str* in numero intero	int	stdlib.h
itoa(<N>, <Str>, <B>**)**	converte il numero intero *N* nella stringa *Str* con formato base specificato dal numero intero *B*	array di char	stdlib.h

OSSERVA COME SI FA

1. **Realizzare un programma che, date una stringa A e una stringa B, calcoli il numero di occorrenze della stringa A in B. Ad esempio, se B contiene "ciao ciao baby" e A = "ciao", deve restituire 2.**

C++

```
#include <iostream>
using namespace std;
#define N 100
int main()
{
  int noccorr=0, i, j, k;
  char s1[N],s2[N];
  cout << "Inserire stringa: ";
  // la gets è necessaria poiché dobbiamo inserire una stringa che contiene spazi
  gets(s1);
  cout << "Inserire sottostringa da cercare: ";
  gets(s2);
  for (i=0; s1[i] != '\0'; i++)
  {
    j=0;
    k=i;
    while((s1[k] == s2[j]) && (s2[j] != '\0') && (s1[k] != '\0'))
    {
      k++;
      j++;
    }
    if (s2[j]=='\0')
    noccorr++;
  }
  cout << s2 <<" compare " << noccorr << " volte in " << s1 << endl;
  system("PAUSE");
  return 0;
}
```

17 | I record

Nella realtà nasce spesso l'esigenza di dover trattare informazioni di tipo diverso relative a un oggetto preso in esame. Ad esempio, per avere delle informazioni su una fattura è necessario ricordare il numero della fattura in questione, la data di emissione, la denominazione dell'impresa emittente e l'importo.
Utilizzando le seguenti variabili semplici:

NOME	TIPO	DESCRIZIONE
NumFat	Intero	Numero della fattura
DataFat	Stringa	Data della fattura
Nome	Stringa	Denominazione dell'impresa emittente
Importo	Reale	Importo della fattura

è possibile risolvere il problema, ma l'utilizzo di tali variabili non indica in alcun modo che i dati in esse contenuti si riferiscono allo stesso oggetto (nel nostro caso, alla stessa fattura).
Un altro esempio: le variabili *Titolo*, *Autore*, *CasaEditrice* e *DataPubb* contengono informazioni relative a un medesimo libro, ma non esiste un modo che ci consenta di assicurarci di ciò.

Dagli esempi esposti si evince che il nostro intento non è quello di descrivere un oggetto, bensì di trovare un metodo per descrivere una classe di oggetti **aventi caratteristiche comuni**, che prendono il nome di attributi.
Riferendoci all'ultimo esempio, possiamo supporre che la classe di oggetti Libri sia composta da tutti i libri presenti nella biblioteca della nostra scuola e che ogni libro sia caratterizzato dai seguenti attributi: *Titolo*, *Autore*, *CasaEditrice* e *DataPubb*. Tali valori rappresentano, perciò, gli attributi comuni a tutti i libri presenti nella biblioteca, quindi tutti gli oggetti appartenenti alla classe.

Un **record**, o **registrazione**, è una struttura di dati a carattere statico composta da un insieme finito di elementi omogenei o eterogenei detti **campi**. I campi sono tra loro logicamente connessi e corrispondono agli **attributi**. Ogni campo accoglie un valore per un attributo.

Un record è caratterizzato da un nome, che lo identifica e si riferisce a esso nella sua globalità. I campi che lo compongono sono caratterizzati da un nome e dal tipo di dato che possono contenere. Possono essere, indifferentemente, di tipo semplice o di tipo strutturato: un singolo campo di un record, infatti, può essere a sua volta un record oppure un array.

Chiamiamo **struttura di un record** la definizione dei campi che compongono il record stesso.

Per definire la struttura di un record possiamo servirci di:

- un **metodo di rappresentazione tabellare** consistente in una tabella simile a quella utilizzata per la dichiarazione delle variabili. Essa indica per ogni campo: il numero, il nome, il tipo, la lunghezza e la descrizione. Tale metodo consente di definire il cosiddetto **tracciato record**:

Tracciato record <IdentificatoreRecord>

N.	NOME	TIPO	LUNGHEZZA	DESCRIZIONE

- un **metodo di dichiarazione sintattica** molto vicino a quello utilizzato da alcuni linguaggi di programmazione. In base a tale metodo, per definire un record occorre specificare:
  - il nome della variabile record;
  - il nome di ogni campo;
  - il tipo di ogni campo.

```
PSEUDOCODICE
<NomeVariabileRecord>: RECORD
    <NomeCampo> {, <NomeCampo>}: <TipoCampo1>
    <NomeCampo> {, <NomeCampo>}: <TipoCampo2>
    <NomeCampo> {, <NomeCampo>}: <TipoCampo3>
    .....
    <NomeCampo> {, <NomeCampo>}: <TipoCampoN>
FINERECORD
```

Esattamente come per gli altri dati strutturati, anche per il record utilizzeremo la parola chiave TIPO per definire un nuovo **tipo record**. Successivamente si potranno dichiarare le variabili di tipo record. Ciò, come ormai sappiamo, non solo ci consente di creare nuovi tipi di dati, ma ci permette di trasferire le variabili di tipo record create come parametri dal programma chiamante ai sottoprogrammi.

OSSERVA COME SI FA

1. Una farmacia ha bisogno di uno schedario dei farmaci e dei prodotti che offre al pubblico. Per ogni prodotto vanno memorizzate e mantenute aggiornate le seguenti informazioni: il nome del prodotto, il nome della casa produttrice, la quantità esistente in farmacia e il prezzo al pubblico.

Rappresentiamo questo record servendoci dei due metodi descritti:

Tracciato record *Prodotti*

N.	NOME	TIPO	LUNGHEZZA	DESCRIZIONE
1	Nome	Stringa	20	Nome del farmaco
2	CasaProduttrice	Stringa	20	Nome della casa produttrice del farmaco
3	Quantità	Intero		Quantità di farmaco presente in farmacia
4	Prezzo	Reale		Prezzo di vendita del farmaco

```
PSEUDOCODICE
Prodotti : RECORD
    Nome, CasaProduttrice: STRINGA[20]
    Quantita: INTERO
    Prezzo: REALE
FINERECORD
```

Riferendoci all'esempio precedente possiamo avere:

Nome:	FarmacoX
Quantita:	10
Prezzo:	10,25
CasaProduttrice:	FarmaXX

Come già sappiamo, gli attributi Nome, CasaProduttrice, Quantita e Prezzo descrivono la classe di oggetti *Prodotti*, e l'insieme dei valori che gli attributi assumono prende il nome di **dominio**. Si chiama **istanza** l'insieme dei valori di una particolare variabile di tipo *Prodotti*.

2. Un docente di informatica vuole organizzare i dati dei suoi studenti. A tale scopo decide di memorizzare, per ognuno di essi: il cognome e il nome, la classe di provenienza, la media dei voti riportati nello scritto e nell'orale e se lo studente è ripetente o meno.

Tracciato record *Studenti*

N.	NOME	TIPO	LUNGHEZZA	DESCRIZIONE
1	Cognome	Stringa	25	Cognome dello studente
2	Nome	Stringa	20	Nome dello studente
3	Provenienza	Stringa	20	Provenienza dello studente
4	MediaScritto	Reale		Media riportata dallo studente nelle prove scritte
5	MediaOrale	Reale		Media riportata dallo studente nelle verifiche orali
5	Ripetente	Booleano		Indica se lo studente è ripetente o meno

PSEUDOCODICE

```
Studenti : RECORD
    Cognome: STRINGA[25]
    Nome, Provenienza: STRINGA[20]
    MediaScritto, MediaOrale: REALE
    Ripetente: BOOLEANO
FINERECORD
```

18 I record in C/C++

Nella terminologia del linguaggio C/C++, i record vengono chiamati **strutture**.
Per **dichiarare** una struttura si utilizza la seguente sintassi:

```
struct <NomeStruttura>
{
 <TipoCampo1> <NomeCampo1>;
 ...
 <TipoCampoN> <NomeCampoN>;
} <identificatore>,<identificatore>, ...;
```

dove *TipoCampo* e *NomeCampo* indicano rispettivamente il tipo e il nome di ciascun elemento (**campo**) che concorre a caratterizzare la struttura che stiamo definendo.
Possiamo allora dire che una struttura, o meglio un particolare tipo di struttura, è dichiarata definendone il nome ed elencando i nomi e i tipi dei suoi campi.
Se è presente *NomeStruttura*, senza alcun identificatore, allora questa viene considerata come una **dichiarazione tipo** (di fatto *NomeStruttura* non definisce alcuna variabile da utilizzare direttamente nel programma, ma rappresenta un nuovo modello di dato aggregato utilizzato successivamente per la dichiarazione delle variabili vere e proprie).
Ad esempio, per dichiarare la struttura **automobile** o equivalentemente "una struttura di tipo **automobile**" o ancora "un tipo di struttura **automobile**", scriveremo:

```
struct automobile
{
  char marca[15];
  char modello[20];
  char targa[7];
  unsigned cilindrata;
  float potenza;
};
```

Campo stringa di 15 caratteri
Campo stringa di 20 caratteri
Campo stringa di 7 caratteri
Campo intero senza segno
Campo a virgola mobile

All'interno della dichiarazione:

- non è consentita alcuna inizializzazione dei campi;
- non è consentito utilizzare per un campo un nome uguale al nome della struttura;
- si possono utilizzare tipi semplici, tipi aggregati o tipi definiti dall'utente.

Quindi, per dichiarare tre variabili di tipo **automobile** scriveremo:

```
struct automobile a1, a2, a3;
```

Le stesse variabili potrebbero essere dichiarate anche in fase di dichiarazione del tipo di **struct**:

```
struct automobile
{
  char marca[15];
  char modello[20];
  char targa[10];
  unsigned cilindrata;
} a1, a2, a3;
```

La dichiarazione:

```
struct automobile a3 = {"Marca1", "Modello1", "XF 345 RT", 1900};
```

costruisce una variabile *a3* di tipo **automobile** e **inizializza** i valori dei suoi campi.
Per riferirci direttamente a un particolare campo di una struttura, utilizziamo la cosiddetta **dot notation**, secondo la seguente sintassi:

```
<VariabileDiTipoStruct>.<NomeCampo>
```

Ad esempio, per assegnare a una variabile *x* di tipo **unsigned** il valore della cilindrata contenuto in una variabile *a3* di tipo **automobile** scriveremo:

```
unsigned x;
x = a3.cilindrata;
```

e nella variabile *x* sarà adesso contenuto il valore 1900.
In modo analogo, per assegnare al campo **cilindrata** della variabile *a1* di tipo **automobile** un nuovo valore, scriveremo:

```
a1.cilindrata = 1600;
```

mentre, per inserire nella struct la targa della variabile *a2*:

```
gets(a2.targa);
```

19 Struct e array

Una struttura può contenere al proprio interno dati di tipo aggregato: **array** e **struct**.
Supponiamo ad esempio di voler rappresentare le informazioni relative ai voli di una compagnia aerea e, insieme ad essi, anche la lista dei passeggeri:

```
struct volo
{
  char nomeVolo[10];
  char partenza[30];
  char destinazione[30];
```