

Algoritmi di ordinamento

1. Introduzione

Il problema di ordinare gli elementi di un insieme è un problema classico dell'informatica, sia da un punto di vista applicativo che didattico. Lo scopo di questa dispensa non è tanto quello di evidenziare gli aspetti fondamentali della progettazione e della costruzione di un algoritmo efficiente, ma è quello di utilizzare la risoluzione di tale problema come utile strumento didattico.

Un **algoritmo di ordinamento** è un algoritmo che viene utilizzato per elencare gli elementi di un insieme secondo una sequenza stabilita da una *relazione d'ordine* (relazione binaria su R che soddisfi determinate proprietà), in modo che ogni elemento sia minore (o maggiore) di quello che lo segue. In generale, la relazione d'ordine viene sempre considerata totale, cioè tale da rendere sempre possibile il confronto tra due elementi dell'insieme. A seconda del verso della relazione considerato, un ordinamento può essere ascendente o discendente.

Il problema dell'ordinamento (**sort**) può dunque essere posto nei seguenti termini: dato un insieme di elementi qualsiasi $A = \{a_1, a_2, \dots, a_n\}$ su cui sia possibile definire una relazione di ordine totale, si richiede di produrre uno scambio degli elementi dell'insieme, in modo tale che

$$a_{ih} \leq a_{ik}, \forall h \leq k$$

con $h, k = 1, 2, \dots, n$.

Ad esempio, se consideriamo l'insieme costituito da $n = 4$ numeri naturali $A = \{a_1 = 7, a_2 = 3, a_3 = 15, a_4 = 6\}$, allora la soluzione del problema dell'ordinamento è data dalla permutazione (a_2, a_4, a_1, a_3) . La soluzione del problema è unica a meno di elementi uguali. Ad esempio nel caso dell'insieme $A = \{a_1 = 7, a_2 = 3, a_3 = 15, a_4 = 7\}$, il problema ammette due soluzioni equivalenti date dalle permutazioni (a_2, a_1, a_4, a_3) e (a_2, a_4, a_1, a_3) .

Il paragrafo successivo illustra la struttura di particolari algoritmi di ordinamento che operano esclusivamente sulla base del confronto dei valori dell'insieme da ordinare. Tali algoritmi sono presentati in pseudo-codice, ma è molto semplice tradurre tale codifica nel linguaggio di programmazione del sistema MATLAB.

2. Selection Sort, Insertion Sort e Bubble Sort

Il **Selection Sort** è un algoritmo decisamente intuitivo ed estremamente semplice. Nella pratica è utile quando l'insieme da ordinare è composto da pochi elementi e dunque anche un algoritmo non molto efficiente può essere utilizzato con il vantaggio di non rendere troppo sofisticata la codifica del programma che lo implementa. L'idea di fondo su cui si basa l'algoritmo è quella di ripetere per n volte una procedura in grado di selezionare alla i -esima iterazione l'elemento più piccolo nell'insieme e di scambiarlo con l'elemento che in quel momento occupa la posizione i .

In altre parole alla prima iterazione dell'algoritmo verrà selezionato l'elemento più piccolo dell'intero insieme e sarà scambiato con quello che occupa la prima posizione; alla seconda iterazione sarà selezionato il secondo elemento più piccolo dell'insieme (ossia l'elemento più piccolo dell'insieme "ridotto" costituito dagli elementi $\{a_2, a_3, \dots, a_n\}$ e sarà scambiato con l'elemento che occupa la seconda posizione, e così via fino ad aver collocato nella posizione corretta tutti gli n elementi. Facciamo un esempio: supponiamo di dover ordinare l'insieme di numeri naturali $A = \{4, 2, 1, 3\}$. Ad ognuno dei passi dell'algoritmo, viene selezionato l'elemento più piccolo nella porzione di insieme ancora da ordinare, scambiandolo con l'elemento che si trova nella prima posizione di tale sottoinsieme. In altri termini al passo i -esimo, viene selezionato l'elemento più piccolo nel sottoinsieme ancora da ordinare costituito dagli elementi $a_i, a_{i+1}, \dots, a_n$ e viene scambiato con l'elemento a_i :

1. $A = (4, 2, \mathbf{1}, 3) \rightarrow A = (1 \mid 2, 4, 3)$
2. $A = (1 \mid \mathbf{2}, 4, 3) \rightarrow A = (1, 2 \mid 4, 3)$
3. $A = (1, 2, \mid \mathbf{4}, 3) \rightarrow A = (1, 2, 3 \mid 4)$
4. $A = (1, 2, 3 \mid \mathbf{4}) \rightarrow A = (1, 2, 3, 4)$

L'algoritmo seguente riporta una pseudo-codifica di Selection Sort. L'insieme da ordinare è l'insieme A e la variabile n rappresenta il numero di elementi di A .

SelectionSort(A, n)

- 1: **per** $i \leftarrow 1, 2, \dots, n$ **ripeti**
- 2: **per** $j \leftarrow i + 1, \dots, n$ **ripeti**
- 3: **se** $a_j < a_i$ **allora**
- 4: scambia a_i e a_j
- 5: **fine-condizione**
- 6: **fine-ciclo**
- 7: **fine-ciclo**

Un altro tipo di algoritmo di ordinamento è l'**Insertion Sort**. Esso si fonda sull'idea di collocare uno dopo l'altro tutti gli elementi a_i dell'insieme nella posizione corretta del sottoinsieme già ordinato costituito dagli elementi $a_1, a_2, \dots, a_{i-1}$, inserendolo facendo scorrere a destra gli elementi maggiori.

Ad esempio supponiamo di voler ordinare utilizzando questa strategia l'insieme $A = \{5, 2, 3, 1, 4\}$. Di seguito riportiamo i passi necessari per risolvere il problema:

1. $A = \{5 \mid \mathbf{2}, 3, 1, 4\} \rightarrow A = \{\mathbf{2}, 5 \mid 3, 1, 4\}$
2. $A = \{2, 5 \mid \mathbf{3}, 1, 4\} \rightarrow A = \{2, \mathbf{3}, 5 \mid 1, 4\}$
3. $A = \{2, 3, 5 \mid \mathbf{1}, 4\} \rightarrow A = \{\mathbf{1}, 2, 3, 5 \mid 4\}$
4. $A = \{1, 2, 3, 5 \mid \mathbf{4}\} \rightarrow A = \{1, 2, 3, \mathbf{4}, 5\}$

Per inserire nella posizione corretta l'elemento a_i , si scorre "all'indietro" il sottoinsieme $a_1, a_2, \dots, a_{i-1}$, ossia cominciando proprio da a_{i-1} , fino ad arrivare eventualmente ad a_1 . Si itera quindi un ciclo controllato dalla variabile $k = i - 1, i - 2, \dots, 2, 1$ in cui, ad ogni iterazione, se $a_k > a_{k+1}$ si scambiano i due elementi contigui a_k e a_{k+1} . Il ciclo termina quando l'elemento a_i è finito all'inizio del sottoinsieme, oppure quando si individua un elemento $a_k \leq a_{k+1}$. L'Algoritmo seguente riporta la pseudocodifica di **Insertion Sort**; anche in questo caso con A indichiamo l'insieme con n elementi da ordinare (n rappresenta la cardinalità di A).

InsertionSort(A, n)

- 1: **per** $i \leftarrow 2, 3, \dots, n$ **ripeti**
- 2: $k \leftarrow i - 1$
- 3: **fintanto che** $k \geq 1$ e $a_k > a_{k+1}$ **ripeti**
- 4: scambia a_k e a_{k+1}
- 5: $k \leftarrow k - 1$
- 6: **fine-ciclo**
- 7: **fine-ciclo**

L'algoritmo **Bubble Sort** (ordinamento "a bolle") si basa sull'idea di far "emergere" man mano gli elementi più piccoli verso l'inizio dell'insieme da ordinare, facendo "sprofondare" al tempo stesso gli elementi più grandi verso il fondo dell'insieme, un po' come avviene con le bollicine gassose in un bicchiere di acqua minerale. La strategia adottata è quella di scorrere più volte la sequenza da ordinare, verificando ad ogni passo l'ordinamento reciproco degli elementi contigui, a_i e a_{i+1} , ed

eventualmente scambiando la posizione di coppie di elementi non ordinate. Procedendo dall'inizio alla fine della sequenza, al termine di ogni scansione si è ottenuto che l'elemento massimo è finito in fondo alla sequenza stessa, mentre gli elementi più piccoli hanno cominciato a spostarsi verso l'inizio della sequenza stessa. Dunque alla fine della prima scansione possiamo essere certi che l'elemento massimo ha raggiunto la sua posizione corretta nell'ultima posizione della sequenza; pertanto, la scansione successiva potrà fermarsi senza considerare l'ultimo elemento dell'insieme e riuscendo così a collocare nella posizione corretta (la penultima) il secondo elemento più grande dell'insieme; e così via fino ad aver completato l'ordinamento dell'intera sequenza. Riportiamo di seguito un esempio in cui è riportata l'evoluzione dell'ordinamento della sequenza durante le quattro scansioni necessarie a risolvere il problema con la strategia che abbiamo appena descritto:

1. $A = \{3,5,2,4,1\} \rightarrow A = \{3,5,2,4,1\} \rightarrow A = \{3,2,5,4,1\} \rightarrow A = \{3,2, 4,5,1\} \rightarrow A = \{3,2, 4,1 \mid 5\}$
2. $A = \{3,2,4,1 \mid 5\} \rightarrow A = \{2,3,4,1 \mid 5\} \rightarrow A = \{2, 3,4,1 \mid 5\} \rightarrow A = \{2, 3,1 \mid 4,5\}$
3. $A = \{2,3,1, \mid 4,5\} \rightarrow A = \{2,3,1 \mid 4,5\} \rightarrow A = \{2,1 \mid 3, 4,5\}$
4. $A = \{2,1 \mid 3, 4,5\} \rightarrow A = \{1,2, 3, 4,5\}$

Osserviamo che effettivamente alla fine di ognuna delle quattro scansioni della sequenza, l'elemento più grande del sottoinsieme ancora da ordinare, finisce in fondo alla sequenza, nella sua posizione definitivamente corretta, mentre la sottosequenza ancora in disordine si riduce di volta in volta di un elemento. Ad esempio alla fine della prima scansione l'elemento 5 finisce in ultima posizione; alla fine della seconda scansione l'elemento 4 (il massimo elemento nel sottoinsieme ancora da ordinare) finisce in fondo, in penultima posizione, e così via. Possiamo formalizzare la strategia risolutiva nell'algoritmo seguente che fornisce una pseudo-codifica di BubbleSort; come al solito n rappresenta il numero di elementi della sequenza A da ordinare.

```

BubbleSort( $A, n$ )
1:  $flag \leftarrow 1$ 
2:  $stop \leftarrow n - 1$ 
3: fintanto che  $flag \leftarrow 1$  ripeti
4:    $flag \leftarrow 0$ 
5:   per  $i \leftarrow 1, 2, \dots, stop$  ripeti
6:     se  $a_i > a_{i+1}$  allora
7:       scambia  $a_i$  e  $a_{i+1}$ 
8:        $flag \leftarrow 1$ 
9:     fine-condizione
10:  fine-ciclo
11:   $stop \leftarrow stop - 1$ 
12: fine-ciclo

```

Anche in questo caso abbiamo un algoritmo strutturato su due cicli nidificati uno dentro l'altro, ma, a differenza dei casi visti nelle pagine precedenti, questa volta il ciclo esterno non viene eseguito un numero prefissato di volte: l'algoritmo Bubble Sort sfrutta la condizione che controlla il ciclo più esterno per verificare la possibilità di interrompere l'esecuzione dell'intero procedimento non appena dovesse risultare evidente che l'insieme è completamente ordinato. In questo modo l'algoritmo è in grado di sfruttare a proprio vantaggio eventuali ordinamenti già presenti nelle sottosequenze dell'insieme da ordinare. Dal punto di vista tecnico gioca quindi un ruolo cruciale la variabile $flag$ che assume il significato di indicatore dello stato di ordinamento della sequenza: il valore $flag = 1$ indica che l'insieme probabilmente ancora non è ordinato, mentre il valore $flag = 0$ segnala che la sequenza risulta ordinata. Pertanto all'inizio della procedura, dal momento che non è ancora chiaro se l'insieme è ordinato o meno, la variabile $flag$ viene impostata con il valore 1. Il

ciclo esterno dell'algoritmo viene controllato unicamente dal valore di questa variabile e viene ripetuto fino a quando la sequenza non risulterà completamente ordinata.

3. Implementazione in MATLAB del Bubble Sort

Consideriamo il problema dell'ordinamento in modo ascendente di un vettore numerico.

I dati del problema sono:

- un certo vettore **v** contenente elementi numerici non ordinati;
- il numero totale degli elementi del vettore ovvero la dimensione del vettore stesso.

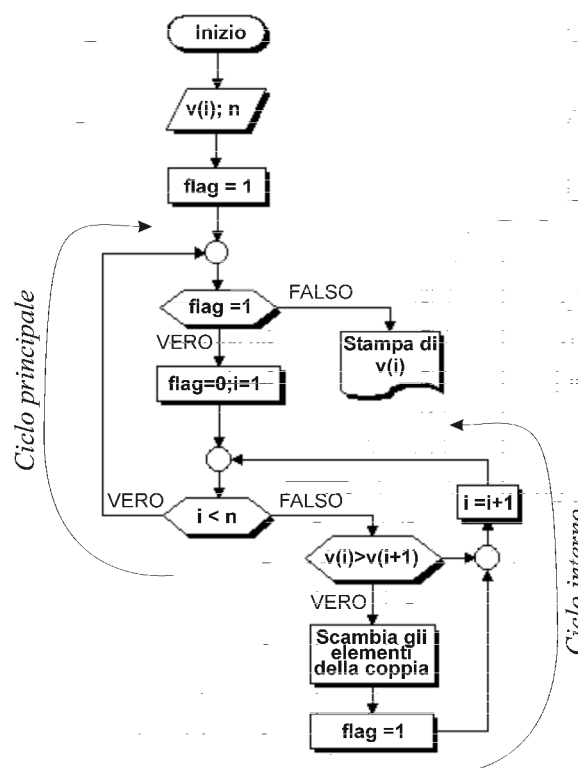
La funzione **sort** esegue questa operazione in modo immediato. Essa opera su un vettore riga o colonna e lo trasforma in un vettore ordinato. L'esempio sotto riportato mostra il modo di operare. Viene generato un vettore riga casuale con 8 elementi compresi tra -5 e +5 e poi ordinato mediante la funzione **sort**.

```
>> n = 8; v = [5*rand(n,1) - 5*rand(n,1)]' % Generazione di un vettore casuale
v =
-2.5341  1.2162  2.6675 -0.8501  2.4392 -1.5137  2.6452 -0.1943

>> sort(v) % Ordinamento del vettore
ans =
-2.5341 -1.5137 -0.8501 -0.1943  1.2162  2.4392  2.6452  2.6675
```

La funzione **sort** appartiene al nucleo di MATLAB come funzione predefinita. Supponiamo di volere costruire una funzione che agisca nello stesso modo mediante un programma che inseriremo in un M-file di nome **ordine.m**.

Lo scopo è solo didattico poiché ci permetterà di utilizzare il linguaggio MATLAB e di esercitarci nella costruzione di un algoritmo risolutivo del problema posto.



Tra i vari algoritmi di ordinamento descritti, risolveremo tale problema mediante l'utilizzo dell'algoritmo Bubble Sort. Di conseguenza, il metodo che useremo consiste semplicemente nello scandire il vettore dall'inizio alla fine, un coppia di elementi alla volta, verificando che tali elementi siano ordinati in modo ascendente. Nel caso in cui ciò non sia vero si scambiano gli elementi tra loro e si segnala l'avvenuto scambio. Alla fine della scansione del vettore ci si chiede se è stato effettuato almeno uno scambio, nel qual caso si ripete la scansione del vettore. Il processo terminerà, fornendo il vettore ordinato, quando durante una scansione non avremo neanche uno scambio tra elementi.

Il flowchart della figura precedente rappresenta l'algoritmo ora descritto:

1. I dati di input sono il vettore $v(i)$ e il numero degli elementi del vettore n .
2. La variabile **flag** viene posta a 1 all'inizio e quando si verifica uno scambio, mentre viene azzerata all'inizio di ogni scansione del vettore.
3. All'ingresso del ciclo principale, quello della scansione di tutto il vettore, viene verificato se il flag è uguale a 0. Se ciò è vero significa che nella scansione precedente del vettore non vi sono stati scambi e quindi il vettore è ordinato.
4. Vi è quindi una struttura **while ... do** che ripete la scansione fino a che la condizione **flag=1** è vera.
5. Internamente al ciclo principale si trova un ciclo (un'altra struttura **while ... do**) che ripete il confronto (struttura **if ... else**) tra elementi della singola coppia finché l'indice si mantiene minore di n , scambiandoli se necessario e ponendo la variabile **flag = 1**.

Vediamo ora il programma in linguaggio MATLAB:

```
n=10; v=5*rand(n,1)-5*rand(n,1); v1=v; % Generazione di 2 vettori casuali v e v1
flag=1; % Il flag viene posto inizialmente a 1 per far eseguire almeno una scansione
% -----
% Scansione ripetuta di tutto il vettore una coppia alla volta mentre vi sono degli
% scambi tra gli elementi delle coppie
while flag==1 % Mentre flag=1 ripeti la scansione
    flag=0; i=1; % Inizializza l'indice i e azzeri il flag
    % -----
    % Scansione delle coppie
    while i < n % Mentre le coppie non sono state esaminate tutte, ripeti ...
        if v(i) > v(i+1) % Condizione per lo scambio degli elementi
            cella=v(i); % -----
            v(i)=v(i+1); % Scambio tra gli elementi della coppia
            v(i+1)=cella; % -----
            flag=1; % Segnalazione avvenuto scambio
        end
        i=i+1; % Nuova coppia da esaminare
    end
end
% -----
ordine = [v1 v] % Visualizzazione del vettore originale e di quello ordinato
```

```
ordine =
    4.1508    -2.3883
   -0.5077    -2.1906
   -0.3248    -1.2624
    1.4651    -0.5077
   -1.2624    -0.3248
   -2.1906     0.0675
    0.5625     0.5625
    1.2060     1.2060
    0.0675     1.4651
   -2.3883     4.1508
```

Il programma genera un vettore casuale di 10 elementi compresi tra -5 e +5 mediante la funzione **rand**. Il vettore generato viene salvato in un vettore **v1** per poterlo confrontare con il vettore dopo l'ordinamento. Il vettore **ordine** è composto dai due vettori.

4. Le sottofunzioni

Questo esempio è dedicato all'esemplificazione della costruzione di una funzione di utente e all'introduzione del concetto di sottofunzione. Vedremo infatti che all'interno del file che contiene la funzione è possibile utilizzare funzioni, dette sottofunzioni, definite nello stesso file.

Lo spunto potrebbe essere il seguente: consideriamo il problema di valutare il valore di due variabili statistiche elementari, quali la *media* e la *mediana*, di un insieme di numeri contenuti in un vettore. Ricordiamo brevemente che la *media* è il rapporto tra la somma degli elementi dell'insieme, diviso il numero totale n degli elementi e la *mediana* è il valore centrale (se n è dispari) oppure la media aritmetica dei due valori centrali (se n è pari). Supponiamo di creare il file **med.m** che conterrà la definizione della funzione che avrà la seguente sintassi:

`[media,mediana] = med(u).`

Con **u** vettore di input del quale bisogna calcolare la media e la mediana.

La sintassi evidenzia che la funzione fornisce due valori di output, il cui calcolo è effettuato all'interno della funzione **med**.

Di seguito è riportato il contenuto del file che definisce la funzione.

```
% ----- Funzione principale -----
function [media,mediana] = med(u) %
% La funzione di utente med determina la MEDIA e la MEDIANA dei valori di un
% vettore per mezzo di funzioni interne (sottofunzioni).
n = length(u); % Determina il numero di elementi di u
media = mean(u,n);
mediana = median(u,n);
% -----

function s = mean(v,n) % Sottofunzione
% Calcola la media.
s = sum(v)/n; % La funzione sum calcola la somma di tutti gli elementi di v

function m = median(v,n) % Sottofunzione
% Calcola la mediana.
w = sort(v); % La funzione sort ordina gli elementi in modo ascendente
if rem(n,2) == 1 % rem restituisce il resto della divisione tra n e 2
    m = w((n+1)/2); % è l'elemento centrale se n è dispari
else
    m = (w(n/2)+w(n/2+1))/2; % è la media dei due elementi centrali se n è pari
end
```

Esaminiamo in dettaglio la funzione:

- la funzione principale **med** utilizza, oltre alla funzione **length**, le due funzioni **mean** e **median** che calcolano rispettivamente la media e la mediana dei valori del vettore **u**;
- le due funzioni **mean** e **median** sono definite nello stesso file;
- poiché la sottofunzione **mean** associa il valore calcolato **s** alla funzione stessa, nella funzione principale la scrittura `media = mean(u,n);` assegnerà alla variabile di output **media** proprio il valore **s** calcolato dalla sottofunzione **mean**. Lo stesso discorso vale per la sottofunzione **median**.

Dopo aver salvato il file è possibile utilizzate direttamente dal **Command Window** la funzione **med**. Proviamo intanto a scrivere:

help med

Otterremo il seguente risultato:

```
med determina la MEDIA e la MEDIANA dei valori di un vettore
per mezzo di funzioni interne(sottofunzioni).
```

che è il commento inserito da noi durante la creazione del file e che serve per ricordarci lo scopo della funzione **med**.

Possiamo ora provare ad utilizzarla:

```
>> u=[(1:3:15) (3:4:34)]
u =
    1     4     7    10    13     3     7    11    15    19    23    27    31

>> [media, mediana] = med(u)
media =     13.1538
mediana =     11
```

Si osservi che l'elemento 11 è quello centrale, considerando il vettore u ordinato in modo ascendente.

5. Algoritmo di inserimento

Un problema molto importante è l'inserimento di un elemento all'interno di un vettore ordinato in modo ascendente. Il problema è chiaramente connesso alla gestione degli archivi per ciò che concerne l'inserimento di nuovi dati. In altre parole si tratta di determinare la posizione nella quale inserire il nuovo elemento, di spostare gli elementi seguenti di una posizione e finalmente di procedere all'inserimento.

Il codice sotto riportato definisce la funzione **inserimento** in grado di effettuare l'operazione. La sintassi che vogliamo creare è la seguente:

```
[v] = inserimento(v);
```

In questo modo il vettore v costituisce sia l'input della funzione che l'output, con la sola differenza che il vettore di output conterrà un elemento in più di quello di input. La funzione è la seguente:

```
function [v] = inserimento(v)           % linea di definizione della funzione
a = input('inserisci elemento:')        % Input dell'elemento da inserire
dim = size(v); % Calcolo della dimensione del vettore v
n = dim(2); % Determinazione del numero totale di elementi del vettore v
i = 1; % Inizializzazione del ciclo
% -----
% Ciclo di confronto tra l'elemento di input ed il vettore
% al fine di determinare la posizione di inserimento
while i < n+1 % Ripete
    if a > v(i) % Posizione non trovata
        i = i + 1;
    else % Posizione individuata
        v = [v 0]; % Estensione del vettore con l'aggiunta di un elemento
        j = n;
        % -----
        % Traslazione degli elementi del vettore di un posto in basso
```

```

        while j > i-1
            v(j+1) = v(j);
            j = j-1;
        end
        % -----
        v(i) = a;      % Inserimento dell'elemento di input al posto esatto
        break;        % Uscita dal ciclo
    end
end
% -----

% -----
% Controllo se l'elemento di input è da aggiungere in coda al vettore
if i == n+1          % L'elemento di input va aggiunto in coda
    v = [v 0];       % Estensione del vettore con l'aggiunta di un elemento
    v(i) = a;        % Inserimento dell'elemento di input al posto esatto
end
% -----

```

Esaminiamo in dettaglio il codice:

- all'inizio la funzione **input** permette di assegnare il valore all'elemento da inserire nel vettore;
- la scrittura **dim=size(v)** crea il vettore **dim** di due elementi: il numero di righe e di colonne del vettore **v**. Per cui **dim(1)** è il numero di righe, mentre **dim(2)** è il numero di colonne che nel nostro caso è il numero totale di elementi del vettore di input;
- una volta che sia determinato il numero di elementi viene innescato un ciclo che serve a confrontare l'elemento da inserire con tutti gli elementi del vettore;
- se l'elemento dovrà essere inserito in testa al vettore o al suo interno sarà il ciclo ad eseguire l'operazione: si procederà all'estensione del vettore, alla traslazione degli elementi seguenti ed all'inserimento;
- nel caso in cui l'elemento sia da inserire in coda al vettore, l'operazione consisterà semplicemente nell'aggiungere un elemento in coda al vettore di input.

Possiamo ora utilizzare la funzione:

```

v = (1:4:45)      % Vettore iniziale ordinato
v =
    1    5    9   13   17   21   25   29   33   37   41   45

[v] = inserimento(v) % La funzione visualizza l'elemento da inserire nel vettore v.
inserisci elemento:12
a =
    12
v =
    1    5    9   12   13   17   21   25   29   33   37   41   45

```

La funzione opera nel modo corretto inserendo il numero 12 dopo il numero 9.