ESEMPIO: ORDINAMENTO CON METODO *BUBBLE SORT*

```
#define MAX 5
#define true 1
#define false 0
typedef float vector [MAX];
     vector v;
     int size; /* dimensione corrente del vettore */

void bubblesort (int iniz, int fine);
void leggi();  void scrivi();

main()      {    printf ("Ordinamento di un vettore");
                 leggi();  bubblesort  (0, size-1);   scrivi();
             }

void bubblesort (int iniz, int fine)
{    int NONSCAMBIO, I;      float     temp;
 do
      { NONSCAMBIO = true;
        for (I = iniz; I < fine; I = I + 1)
         {  if (v[I] > v[I+1])
            {      NONSCAMBIO = false;
                   temp = v[I];  v[I] = v[I+1]; v[I+1] = temp;
            }
        }
      }
  while (! NONSCAMBIO);
}
```


ESEMPIO: ORDINAMENTO CON METODO *QUICK SORT*

```
#define MAX 5
     typedef float vector [MAX];
     vector v;
     int size;

void quicksort (int low, int hi);
/* ordinamento quicksort:
il vettore viene diviso in due parti, una di elementi inferiori al
pivot, una di elementi superiori.
Il procedimento di applica ricorsivamente alle due parti Il
quicksort è particolarmente efficace: il caso peggiore è
limitato in complessità
*/

void leggi();
void scrivi();
main()
{ printf ("Ordinamento di un vettore");
  leggi();   quicksort (0, size-1);  scrivi();
}

void quicksort (int low, int hi)
/* indici iniziali e finali dell'array corrente */
{          int i, j;
           float  pivotelem, temp;

  if ( low < hi )
   {i = low; j = hi;
    pivotelem = v[hi];  /* come pivot l'ultimo elemento */
```

```
    do  /* trova il pivot */
     {        while ((i < j) && (v[i] <= pivotelem)) i = i + 1;
              while ((j > i) && (v[j] >= pivotelem)) j = j - 1;
              if (i < j)      { temp = v[i]; v[i] = v[j]; v[j] = temp; }
     }
    while  (i < j);
    /* si sono determinati i due sottoinsiemi */

    /* posiziona il pivot */
    if ((i != hi) && (v[i] != v [hi]))
           { temp = v[i];     v[i] = v[hi]; v[hi] = temp;}

    /* ricorsione sulle sottoparti*/
    if (low < i - 1)       quicksort (low, i - 1);
    if (i + 1< hi )        quicksort (i + 1, hi); }
}
```

Si determini la **complessità** dell'algoritmo:
il numero di iterazioni/ricorsioni dipende dalla sola dimensione del vettore?

ESEMPIO: ORDINAMENTO CON METODO *SHELL SORT*

```
#define MAX 5
#define true 1
#define false 0
    typedef float vector [MAX];
    vector v;        int size;



void shellsort (int iniz, int fin)
{ int ind, prec, succ, I;     int gap;     int fine;
/* ordinamento Shell:
analisi degli elementi a distanza gap
(il gap va da size a 1).
I confronti, in caso di scambio, si retropropagano
verso l'inizio del vettore, fino all'estremo inferiore */

 gap = (fin - iniz + 1) / 2;
 /* gli indici a partire da 0 fanno considerare il gap */
```

```
 while (gap > 0)
  {  for (ind = gap; ind < size; ind = ind +1)
     { prec = ind - gap;
       do
         { succ = prec + gap;
           if (v[prec] > v [succ])
                 {     fine = false;  temp = v[prec];
                       v[prec] = v[succ];       v[succ] = temp;
                       prec = prec - gap;
                  }
           else   fine = true;
         }    while ((! fine) && (prec >= 0));
      }
     gap = gap / 2;
} }
void leggi();         void scrivi();

main()
{printf ("Ordinamento di un vettore");
   leggi();  shellsort (0, size-1);  scrivi();
}
```


ESEMPIO: ORDINAMENTO CON METODO *MERGE SORT*

```
#define MAX 5

typedef float vector [MAX];
vector v;  int size;
void  merge (iniz1, iniz2, fine);

void mergesort (i, j)
         int i, j;
/*       ordinamento ricorsivo: le due sottoparti di un vettore
   sono ordinate separatamente e poi vengono fuse (merge)
   in un unico vettore
*/
{        int m;
   if ( i < j )
     { m = (j + i) / 2;
       mergesort  (i , m);
       mergesort  (m +1, j);
       merge (i, m + 1, j);
     }
}
void leggi();   void scrivi();
main()
{ printf ("Ordinamento di un vettore: MERGESORT");
   leggi();  mergesort (0, size-1); scrivi();
}
```

```
void  merge (iniz1, iniz2, fine)
int iniz1, iniz2, fine;
/* fusione di due vettori */
/* uso di un vettore temporaneo vout*/
{    vector vout; int i, j, k;

   i = iniz1; j = iniz2; k = iniz1;
   /*confronto degli elementi correnti */
   while (( i <= iniz2 -1) && ( j <= fine ))
   { if (v [i] < v [j])       { vout [k] = v[i]; i= i + 1; }
     else                     { vout [k] = v[j]; j= j + 1; }
     k = k + 1;
   }

/* fasi di trattamento del vettore non terminato */
   while ( i <= iniz2 -1) { vout [k] = v[i]; i= i + 1;  k = k + 1;}
   while ( j <= fine ) { vout [k] = v[j]; j= j + 1; k = k + 1;}

/* copia da vout in uscita */
   for (i = iniz1; i<= fine; i=i+1) v[i] = vout [i];
}
```


ESEMPIO COMPLETO:
DIVERSI METODI DI ORDINAMENTO

```
#define MAX 5
#define true 1
#define false 0

typedef float vector [MAX];
vector v;   int size;

void scambia (float *a, float *b)
 /* passaggio dei parametri per indirizzo in C */
{         float temp;
   temp = *a;  *a = *b;  *b = temp;
}

void bubblesort (int iniz, int fine)
{    int NONSCAMBIO, I;
 do
      { NONSCAMBIO = true;
        for (I = iniz; I < fine; I = I + 1)
         {  if (v[I] > v[I+1])
                    { NONSCAMBIO = false;
                      scambia ( &v[I], &v[I+1] );
      }   }         }
 while (!NONSCAMBIO);
}
```

```
void quicksort (int low, int hi);
void shellsort (int iniz, int fin);
void merge (int iniz1, int iniz2, int fine);
void mergesort (int i, int j);

void leggi ();
void scrivi ();

main()
{   int j, i; char ch;

   printf ("Ordinamento di un vettore con metodi\n ");
   leggi ();
   scrivi ();
   do
   { printf ("Bubble, Shell, Merge, Quick\n");
     scanf ("\n%c", &ch);
   }
   while (ch != 'B' && ch != 'S' && ch != 'M' && ch != 'Q');

   switch (ch)
     {    case 'M': mergesort (0, size-1); break;
          case 'B': bubblesort (0, size-1); break;
          case 'Q': quicksort (0, size-1); break;
          case 'S': shellsort (0, size-1);
     }

     scrivi ();
}
```


ESEMPIO: RICERCA UNA STRINGA IN UN TESTO

METODO INGENUO

per ogni posizione nel testo si ricerca la occorrenza

Il programma si sviluppa in un ciclo esterno
<per ogni carattere del testo>
con un ciclo interno
<controllo della eventuale uguaglianza>

Si veda il filtro seguente che deve lavorare a linee sull'ingresso e passa in uscita la linea se e solo se contiene l'argomento specificato (**patmatch**)

La ricerca va fatta per ogni posizione del testo ==>
indice **indice** scandisce il testo,
l'indice **i** parte da **indice** nel testo e l'indice **j** la stringa
e si confrontano i caratteri successivi
La ricerca ha successo ==>
se abbiamo scandito l'intera stringa da ricercare

In caso di differenze bisogna considerare nel testo posizioni successive ==>
indice nel testo indice mantiene la posizione

se **M** è la dimensione del testo
ed **N** la dimensione della stringa
si effettuano **M*N** confronti

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>
int patmatch ( char * s, char * pattern);
/* ritorna l'indice di match nel testo o valore negativo */

void main  (argc, argv)   int argc;  char **argv;
{    char * s = (char *)malloc (180), * stringa;
     int cont, indice;

if (argc != 2)
{printf (" errore:\n Necessario 1 argomento per ricerca");exit (-
1);}

 cont = 0;
 while ( gets (s))
 { stringa = s;
     while ( (stringa[0] != 0)     &&
          (indice = patmatch (stringa, argv[1])) >= 0)
     { cont +=1;
       stringa = stringa + indice + strlen (argv[1]);
       }

/*       scandiamo la singola stringa (testo) presa in
   ingresso:
   la funzione patmatch porta in uscita l'indice della prima
   occorrenza della stringa argv[1] nella linea di ingresso
   se non c'è match, la funzione restituisce -1*/
 }
 printf ("%d\n", cont);
}
/* si ricerca per ogni posizione del testo un eventuale match:
```



```
   in caso di uguaglianza si avanza nella stringa e nel testo
   fino alla fine della stringa
   in caso di differenza, è necessario tornare indietro alla
   posizione del testo successiva alla iniziale  */


int patmatch ( char * s, char * pattern)
{    int i,j, indice;
      M = strlen (s), N = strlen (pattern);

for (indice = 0; indice < M; indice++)
  {for ( i=indice, j =0;  j < N && i < M; )
       if ( s[i] == pattern [j]) i++, j++;
       else break;
/* si continua fintanto che i caratteri sono uguali */
    if (j == N)   return indice;
/*   se abbiamo scorso l'intero pattern abbiamo trovato,
     altrimenti dobbiamo ripartire dalla posizione successiva
     nel loop esterno, incrementando indice */
  }
  return -1;
}
```

M*N confronti per la ricerca completa (insuccesso)

Vediamo una versione ottimizzata (?)

Usiamo due soli indici: i nel testo e j nel pattern
In caso di caratteri diversi, l'indice i (unico) viene riportato indietro di tante posizioni quante indicate dalle posizioni sulla stringa da ricercare (**j**)

```
int patmatch ( char * s, char * pattern)
 {int i,j, M = strlen (s), N = strlen (pattern);
/* il ciclo for esegue in caso di uguaglianza
    gli indici vengono incrementati e si ripete il confronto  */
     for ( i=0, j =0;  j < N && i < M; i++, j++)
          while ( s[i] && (s[i] != pattern[j]))
               {    i -= j - 1;
                    j = 0;}
/* il ciclo interno esegue solo in caso di differenza:
    gli indici sono riaggiornati.
    In caso di caratteri diversi, si esegue solo nel ciclo interno
*/

/* se la stringa è stata scandita tutta, abbiamo trovato il
   match dalla posizione iniziale (cioé a N posizioni dall'indice
   corrente
*/
     if (j == N) return i - N;
     else return -1;
  }
```


ESEMPIO: RICERCA UNA STRINGA IN UN TESTO

METODI ottimizzati

si cercano di evitare i confronti per ogni posizione nel testo

per esempio sulla base della conoscenza della stringa da ricercare

se ricerchiamo la stringa **babalu** in un testo

assds*ba**babalu***fdfsdfsdfsd**babalu**babal
^ ^ differenza
| *inizio*

il match di **baba**b con **baba**l fino ai caratteri **b** ed **l**
ci consente anche di non tornare indietro e partire dalla posizione corrente nel testo a fare match con la posizione opportuna nella stringa (cioè la seconda b)

Si progetti un algoritmo per tenere conto di questo
==>
è possibile non tornare mai a ricontrollare caratteri già visti (usando un automa a stati finiti)

Esaminiamo altre possibili soluzioni

ESEMPIO: RICERCA UNA STRINGA IN UN TESTO

Confronto a partire dalla fine della stringa

Si consideri una stringa da cercare in un possibile alfabeto di molti simboli e di lunghezza limitata:

si pensi di non confrontare a partire dall'inizio della stringa, ma dalla fine (con una posizione congrua nel testo)

Se i caratteri finali fanno match ==>
allora si torna all'indietro fino ad arrivare all'inizio della stringa e avendo trovato un match nel testo

Se non c'è match, possiamo ragionare sul testo ==>
se abbiamo trovato un carattere del testo che non è contenuto nella stringa, possiamo avanzare nel **testo** della **dimensione** della stringa stessa
se abbiamo trovato un carattere contenuto nella stringa, allora, per stare sicuri, dobbiamo considerare la ultima posizione del carattere nella stringa e muoverci a considerare nel testo la posizione che consenta di averlo nella posizione giusta

L'algoritmo può basarsi su una struttura dati che per ogni possibile simbolo informa di quanto è possibile saltare in avanti nel testo (array skip)


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>

int skip [256];  /* array di supporto */

void initskip ( char * pattern);
int patmatch ( char * s, char * pattern);

void main  (argc, argv)   int argc;  char **argv;
{    char * s = (char *)malloc (180), * stringa;
     int cont, indice;

if (argc != 2)
{printf (" errore:\n Necessario 1 argomento per ricerca"); exit(-
1);}

 initskip (argv[1]);   cont = 0;
 while ( gets (s))
 { stringa = s;
     while ( (stringa[0] != 0)     &&
          (indice = patmatch (stringa, argv[1])) >= 0)
     { cont +=1; stringa = stringa + indice + strlen (argv[1]);}
 }
 printf ("%d\n", cont);
}

/* segue versione ottimizzata */
```

```
int patmatch ( char * s, char * pattern)
{int i,j,t, M = strlen (s), N = strlen (pattern);

if (M < N) return  -1;
  for ( i=N-1, j =N-1;  j > 0 && s[i] != pattern[j] ; i--, j--)
       while ( s[i] != pattern[j])
       {   t= skip[s[i]];
           i += (N-j > t) ? N -j: t;
           if ( i >= M ) return -1;
           j = N-1;
       }
      if (s[i] == pattern[j]) return i;  else return -1;
  }

void initskip ( char * pattern)
{ int i, N = strlen (pattern);
for (i=1; i < 256; i++)  skip [i] = N;
for (i=0; i < N; i++)  skip [pattern[i]] = N - i - 1;
}
```

- Questa strategia usa **M+N** confronti nel caso peggiore
- Se l'alfabeto è sufficientemente ampio ed otteniamo molti salti (stringhe corte rispetto all'alfabeto, allora si può arrivare ad avere solo **M/N** confronti

ESEMPIO: RICERCA STRINGA IN UN TESTO

Confronto basato su sintesi numerica della stringa e del testo

Si consideri la stringa da cercare come una sequenza di caratteri con posizione specificata:

se associamo ad ogni posizione una espressione posizionale opportuna, la stringa può essere rappresentata in modo sintetico da un valore unico senza problemi di collisione

$h_1 = p[j]* d^{N-1} + p[j+1]* d^{N-2} + \ ...+ p[j+N-1]* d^{N-N}$
$h_2 = s[i]* d^{N-1} + s[i+1]* d^{N-2} + \ ...+ s[i+N-1]* d^{N-N}$

Se i due valori (testo e stringa) per una posizione nel testo fanno match ==> h_1 e h_2
abbiamo trovato

Se non c'è match procediamo ==> h_1 fissa e h_2 varia

se riusciamo a passare al nuovo valore nel testo senza rifare tutti i conti abbiamo trovato un buon metodo
tanto migliore è la strategia quanto più è facile passare al nuovo valore

$x_i = h_2 = s[i]* d^{N-1} + s[i+1]* d^{N-2} + \ ...+ s[i+N-1]$
nuovo valore
$x_{i+1} = h_2 = s[i+1]* d^{N-1} + s[i+2]* d^{N-2} + \ ...+ s[i+N]$
$\mathbf{x_{i+1} = (x_i - s[i]* d^{N-1})* d + s[i+N]}$

Per passare da un valore all'altro si usano i principi dell'aritmetica modulare per cui si possano fare sempre operazioni sui moduli (anziché sui valori stessi)

h[**i**] **= x_i mod q**

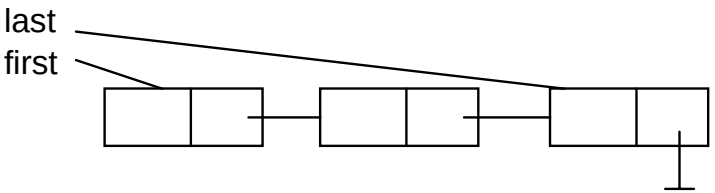
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>

int patmatch ( char * s, char * pattern);

void main  (argc, argv)
int argc;  char **argv;
{    char * s = (char *)  malloc (180),  * stringa;
     int cont, indice;

if (argc != 2)
{ printf (" errore:\n Necessario 1 argomento per ricerca"); exit (-
1);}
 cont = 0;
while ( gets (s))
 { stringa = s;
     while ( (stringa[0] != 0)     &&
          (indice = patmatch (stringa, argv[1])) >= 0)
     { cont +=1; stringa = stringa + indice + strlen (argv[1]);}
 }
 printf ("%d\n", cont); }


/*  versione ottimizzata */
#define q 33554393L      /* elevato numero primo*/
```



```
#define d 32

int patmatch ( char * s, char * pattern)
{    int i;
     long int  dN = 1, h1 = 0, h2 = 0;
     int   M = strlen (s), N = strlen (pattern);

if ( N <= M)
{ for (i = 1; i < N; i++)  dNmeno1 = ( d * dNmeno1) % q;
/* calcolo di d elevato ad N - 1  */

  for ( i= 0; i < N; i++)
  {h1 = (h1 * d + pattern[i]) % q; h2 = (h2 * d + s[i]) % q;}
/* calcolo del valore per la stringa: si noti l'operazione di
   modulo */

  for ( i= 0; h1 != h2 && i < M - N; i++)
  {  h2 = (h2 + q * d - s[i] * dNmeno1)  % q;
     h2 = (h2 * d + s[i + N]) % q;
   }
/* calcolo del valore per ogni posizione del testo  */

  if ( h1 == h2 ) return i; else return -1;
}
else return -1;
}
```

Il fattore ***q * d*** è introdotto per mantenere la somma positiva

Gestione di Liste
(versione non ricorsiva)



Ogni lista è rappresentata dalla coppia di puntatori
(first,last)

Operazioni:

- ⇨ **Create():** inizializza first/last;
- ⇨ **End():** elimina la lista;
- ⇨ **IsIn(i):** verifica la presenza di un valore i nella lista
- ⇨ **Empty():** verifica se la lista è vuota;
- ⇨ **Length():** restituisce il numero degli elementi componenti la lista;
- ⇨ **EnqueueF(i):** inserisce l'elemento i in testa alla lista;
- ⇨ **EnqueueL(i)**: inserisce l'elemento i in coda alla lista;
- ⇨ **Enqueue(i):** inserisce l'elemento dato in testa (solo se non è già presente);

  Dealloca(p): dealloca un elemento (individuato dall'indirizzo p);
- ⇨ **DequeueF()**: estrae il primo elemento;
- ⇨ **DequeueL()**: estrae l'ultimo elemento;
- ⇨ **Dequeue(i)**: estrae l'elemento dato.

ESEMPIO 1: LISTA

```
#include <stdio.h>
#include <alloc.h>
#define NULL 0

struct node
{   int                 item;
    struct node   *     next;
};
struct node *first, *last;

void    Create ();
void    End ();
int     IsIn (int i);
int     Empty ();
int     Length ();
void    EnqueueF (int i);
void    EnqueueL (int i);
void    Enqueue (int i);
int     DequeueF ();
int     DequeueL ();
int     Dequeue (int i);
```

```
main ()
{ int i, b1, b2;
     printf("inizio programma di prova della lista\n");
     Create ();
     EnqueueF (12);   printf("inserito 12 al primo posto\n");
     EnqueueL (13);   printf("inserito 13 all'ultimo posto\n");
     i = Length ();       printf("la lista è lunga %d \n", i);
     b1 = DequeueL ();
     printf(" e' stato estratto l'ultimo elemento %d \n", b1);
     if (b2 = IsIn (13))  printf("13 è in coda\n");
          else printf("13 non e' in coda\n");
     EnqueueF (7);     printf("inserito 7 al primo posto\n");
     if ( Empty ())        printf("la lista è vuota\n");
          else printf("la lista non è vuota\n");
     EnqueueF (12);   printf("inserito 12 al primo posto\n");
     EnqueueL (24);   printf("inserito 24 all'ultimo posto\n");
     i = Length ();
     printf("la lista è lunga %d \n", i);
     b1= Dequeue (11);
     if (b1) printf("11 era in coda\n");
          else  printf("11 non era in coda\n");
     b1= Dequeue (24);
     if (b1) printf("24 era in coda\n");
          else  printf("24 non era in coda\n");
     b1= DequeueF ();
     printf(" è stato estratto il primo elemento %d \n", b1);
     End ();
}
```



```
void Create ()
{    first = NULL;  last = NULL;
}

void End ()
{  int    i;
/* toglie e dealloca tutto  */
while (first != NULL) i = DequeueF();
}

int  IsIn (int i)
{   struct node *t;
t = first;
while (t != NULL && t ->item != i) t = t ->next;
return(t != NULL);
}
```

```
int  Empty ()
{    return (first == NULL);
}

int  Length ()
{   int count = 0;
    struct  node *temp = first;

     while (temp != NULL) { count++; temp = temp -> next;}
     return (count);
}

void EnqueueF (int i)
{   struct node *newnode;

/* inserisci l'elemento  in testa alla lista */
newnode = (struct node *) malloc(sizeof(struct node));
newnode -> next       = first;
newnode ->item        = i;
if (first == NULL)    { last = newnode;}
first                 = newnode;
}
```



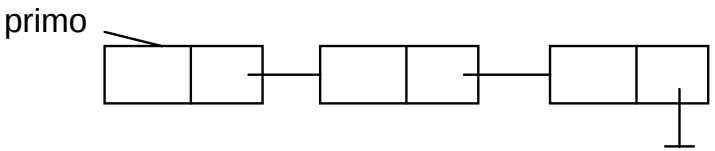
```
void EnqueueL (int i)
{   struct node *newnode;
/* inserisci l'elemento all'ultimo posto in lista */
newnode = (struct node *) malloc(sizeof(struct node));
newnode -> next   = NULL;
newnode ->item    = i;
if (first == NULL)  {   first   = newnode; last  = newnode; }
else
{     last->next = newnode;  last  = newnode; }
}

void Enqueue (int i)
{   struct node *t;
t = first;
while (t != NULL && t ->item != i) t = t ->next;
if (t == NULL)
/* inserisci l'elemento in lista:  non c'è */
          EnqueueF (i);
}

int Dealloca (struct node *temp)
{ int reply;
   reply = temp -> item;
   free ((void*) temp);
   return (reply);
}
```

```
int DequeueF ()
{   struct node *temp = first;
if (first == NULL)  /* lista vuota */
          return (NULL);
else
          { if (first == last)
  /* un solo elemento da togliere */
                {     first  = NULL;
                      last   = NULL;
                }
            else   first = temp -> next;
  return Dealloca(temp);
}
}

int DequeueL ()
{    struct node *old, *new, *temp = last;
if (first == NULL)  /* lista vuota */ return (NULL);
else  { if (first == last)   /* unico elemento da togliere */
            {     last  = NULL;  first = NULL; }
            else
            {  old = first;  new = old -> next;
               while  (new  !=  last)  {  old  =  new;  new  =  new  ->
           next;}
               last = old;
               old ->next = NULL;
            }
return Dealloca(temp); }
}
```



```
int Dequeue (int i)
{
if (first == NULL)   /* lista vuota */ return (NULL);
else { if (first -> item == i) /* primo elemento da togliere */
                return DequeueF ();
else
 {  struct node *t,  *temp;
    t = first; temp = t -> next;
    while (temp != NULL && temp ->item != i)
         { t = temp; temp = t ->next; }
          if (temp != NULL)          /*  l'elemento c'è */
          {   if (t -> next == last ) last = t;
              t -> next = temp -> next;
              return Dealloca(temp);
          }
else return (NULL); /* l'elemento non c'è  */ }
}
}
```

Gestione ricorsiva di Liste



La lista è rappresentata dal puntatore al primo elemento

In genere, semantica per copia

Operazioni:

- ⇨ **Cons(elem, lista):** inserisce l'elemento in testa alla lista (v. EnqueueF);
- ⇨ **Head(lista):** ritorna il valore del primo elemento;
- ⇨ **Tail(lista):** ritorna la lista ottenuta eliminando il primo elemento;
- ⇨ **Length(lista):** restituisce il numero degli elementi componenti la lista;
- ⇨ **Member(i, lista):** verifica se l'elemento i è presente nella lista;
- ⇨ **Append(i, lista):** inserisce l'elemento dato in coda alla lista (v. EnqueueL);
- ⇨ **Inserisci(i, lista):** inserisce l'elemento dato in ordine all'interno dlla lista;
- ⇨ **Sum(lista):** somma tutti gli elementi della lista.

ESEMPIO 2:
LISTA con funzioni RICORSIVE

```
#include <stdio.h>
#include <alloc.h>
#define nil 0

typedef enum { f, t } boolean;
typedef struct ELEMENTO
    {   int ITEM;
        struct ELEMENTO *NEXT;
    } ELEMENTO;

ELEMENTO * CONS (int elem, ELEMENTO *radice)
{ ELEMENTO *Punt;

   Punt = (ELEMENTO *) malloc (sizeof(ELEMENTO));
   Punt -> ITEM = elem;
   Punt -> NEXT = radice;
   return Punt;
};  /*CONS*/

int HEAD (ELEMENTO *radice)
{
  if (radice == nil) return nil;
  else  return radice -> ITEM;
 };  /*HEAD*/
```

```
ELEMENTO * TAIL (ELEMENTO *radice)
 {
  if (radice == nil) return (ELEMENTO *) nil;
  else   return radice -> NEXT;
 };  /*TAIL*/

void writelista (ELEMENTO *radice)
{
    if (radice != nil)
          {    printf ("%d ", HEAD(radice));
               writelista(TAIL(radice));
          }
     else     { printf ("\n"); }
 };  /*writelista*/

int LENGTH (ELEMENTO *radice)
 {
  if (radice == nil) return 0;
  else return (1 + LENGTH(TAIL(radice)));
 };  /*LENGTH*/

int SUM (ELEMENTO *radice)
 {
  if (radice == nil) return 0;
  else return ( HEAD(radice) + SUM(TAIL(radice)));
 };  /*SUM*/
```



```
boolean MEMBER (int el, ELEMENTO *radice)
{
  if (radice != nil)
   { if ( el == HEAD(radice) ) return (t);
     else return ( MEMBER(el,TAIL(radice)) );
   };
  return (f);
 };  /*MEMBER*/

ELEMENTO * APPEND (int el, ELEMENTO *radice)
{ ELEMENTO *p;
  if (radice == nil)
   { p = (ELEMENTO *) malloc (sizeof(ELEMENTO));
     p -> ITEM = el;  p -> NEXT = nil;
     return (p);
   }
  else  return (CONS ( HEAD (radice),
                       APPEND (el, TAIL (radice)) ));
};  /*APPEND*/

void APPENDA (int el, ELEMENTO **radice)
/*versione alternativa ==> con side-effect */
 { if (*radice == nil)
   { *radice = (ELEMENTO *) malloc (sizeof(ELEMENTO));
     (*radice) -> ITEM = el;  (*radice) -> NEXT = nil; }
  else  APPENDA (el, &((*radice) -> NEXT));
};  /*APPENDA*/
```

```
ELEMENTO * INSERISCI (int el, ELEMENTO *radice)
{ ELEMENTO *p;
   if (radice == nil)
    { p = (ELEMENTO *) malloc (sizeof(ELEMENTO));
      p -> ITEM = el;  p -> NEXT = nil;
      return  p;
    }
   else if (MEMBER (el, radice) != t)
              if (HEAD(radice) > el) return ( CONS (el, radice));
  /*inserimento in testa*/
              else
              return (CONS ( HEAD (radice),
                                  INSERISCI (el, TAIL (radice)) ));
          /* else   se c'è già non si inserisce*/
};  /*INSERISCI*/

main ()
{    ELEMENTO *ROOT = nil, *ROOT1;
     int    CHOICE,EL,L;
     boolean FINE = f, R;

/*main*/
 while (FINE != t)
 {   printf ("\n\nQuale funzione vuoi eseguire sulla lista
              di interi ?\n");
    printf ("1. CONS  2. HEAD   3. TAIL  4. LENGTH      \n");
    printf  ("5. SUM   6. MEMBER 7. APPEND   8. INSERT
    \n");
    printf ("9. APPENDA    10. SHOW    11. ESCI      \n");
    scanf ("%d", &CHOICE);
    switch (CHOICE) {
     case 1:
       printf ("Argomento da appendere in testa? ");
```



```
    scanf ("%d", &EL);
    ROOT = CONS(EL,ROOT);
    writelista(ROOT);
    break;

case 2:
    EL = HEAD(ROOT);
    printf("L'elemento ottenuto è  %d\n", EL);
    break;

case 3:
    ROOT1 = TAIL(ROOT); writelista(ROOT1);
    break;

case 4:
     L = LENGTH(ROOT);
     printf ("La lunghezza della lista  è %d\n", L);
     break;
case 5:
     printf ("Somma degli elementi %d\n", SUM(ROOT));
     break;
case 6:
     printf ("Qual è l'argomento da ricercare? \n");
     scanf ("%d", &EL);
     R = MEMBER(EL,ROOT);
     if (R == t) printf ("YES\n");
     else printf ("FALSE\n");
     break;
case 7:
     printf ("L'argomento da appendere in coda?\n ");
     scanf ("%d", &EL);
     ROOT = APPEND(EL,ROOT);
     writelista(ROOT);
```

```
            break;
        case 8:
            printf ("L'argomento da inserire in ordine? ");
            scanf ("%d", &EL);
            ROOT = INSERISCI(EL,ROOT);
            writelista(ROOT);
            break;
        case 9:
            printf ("L'argomento da appendere in coda
                     (con SIDE-EFFECTS)?\n");
            scanf ("%d", &EL);
            APPENDA(EL,&ROOT);
            writelista(ROOT);
            break;
        case 10:
            writelista (ROOT);
            break;
        case 11:
            FINE = t;
            break;
      }  /*case*/
   }    /*while*/
}     /*main*/
```


ESEMPIO 3: PASSAGGIO DI UNA FUNZIONE COME PARAMETRO

```
/* Il programma vuole determinare lo ZERO di una funzione
f ad una variabile mediante il metodo della BISEZIONE.
Lo zero deve essere presente nell'intervallo [inf,sup]       */

#include <stdio.h>
#include <alloc.h>
#include <math.h>
#define  eps 1.0E-14
float fx (float x);
float zero (float (*f)(float x), float a, float b);
/* È possibile anche l'uso della funzione come f (invece di *f)
   all'interno della funzione zero     */

main ()
{ /* bisect */
float inf, sup;
     printf ("\nValore di eps %g", eps);
     printf ("\nScrivi il valore inf e sup dell`ascissa \
della funzione\n");
     scanf ("%f %f",&inf, &sup);
     printf ("\nValore di inf %f e sup %f\n", inf, sup);
     printf ("lo zero è %f\n ", zero (fx,inf,sup));
};  /* bisect */
```

```
float fx (float x)
    { return (x*x - 2*x + 1);}; /* fx */


float zero (float (*f)(float x), float a, float b)
/*  accanto  alla  funzione  f  che  compare  come
    parametro formale  c'è il tipo  dei parametri formali:
    questo permette al compilatore di svolgere controlli
    su tali parametri alla chiamata della funzione f
*/
{ float x, z, s;

  if ( abs ( (*f)(a) ) < eps) x = a;
  else  if (abs ( (*f)(b) ) < eps) x = b;
        else {  s = (*f)(a) < 0;
                do  {    x = (a+b)/2.0;
                         z = (*f)(x);
                         if ((z < 0) == s) a = x;
                         else b = x; }
                while (abs (a-b) >= eps);
             };
  return x;
}; /* zero */
```

cosa succederebbe in caso che zero restituisse un intero?

ESEMPIO 4: ALBERO

```
/* Questo esercizio utilizza i PUNTATORI per memorizzare
caratteri letti da terminale in una struttura ad ALBERO, in un
ordine opportuno.  */
#include <stdio.h>
#include <alloc.h>
#define nil 0
typedef struct nodo   {
            char valore;    struct nodo *prec, *succ;
                        } NODO;
typedef NODO *ptr;

CercaeInserisci (char car, /*var*/ ptr *p)
{    if (*p == nil) /* inserimento */
         {    *p = (ptr) malloc(sizeof(NODO));
              (*p) -> valore = car;
              (*p) -> prec = nil;
              (*p) -> succ = nil;
                  }
  else
    if (car < (*p) -> valore)
         CercaeInserisci(car, &((*p) -> prec));
    else
         if (car > (*p) -> valore)
              CercaeInserisci (car, &((*p) -> succ) );
   };
   /* if car = (*p) -> valore la procedura termina senza fare
      nulla all'albero  */
```

```
void inordine (ptr p)
{
  if (p != nil)
        {   inordine (p -> prec);
            printf (" %c", p -> valore );
            inordine (p -> succ);
        };
}; /* inordine */

void differito (ptr p)
{
  if (p != nil)
        {   differito (p -> prec);
            differito (p -> succ);
            printf (" %c", p -> valore );
        };
}; /* differito */

void anticipato (ptr p)
{
  if (p != nil)
        {   printf (" %c", p -> valore );
            anticipato (p -> prec);
            anticipato (p -> succ);
        };
}; /* anticipato */
```



```
main ()
{   ptr  radice;
    char ch;
radice = nil;
printf ("\nScrivi il testo da processare e termina
        con un punto . \n");

do
    {   ch = getchar();
        if ((ch != ' ') && (ch != '.'))
                CercaeInserisci (ch, &radice);
     }
while (ch != '.')
/* il programma non è un filtro */

printf ("\nStampa in ordine alfabetico\n");
inordine(radice);
printf ("\nStampa in ordine differito\n");
differito(radice);
printf ("\nStampa in ordine anticipato\n");
anticipato(radice);
printf ("\n");
};
```

ESEMPIO 5: SIMULAZIONE DI UN OGGETTO STACK

```
#include <stdio.h>
#include <alloc.h>
#define SIZE 20

typedef struct
    {   struct { int TOP;  int S [SIZE]; } stato;
        struct { void (*init)();
                 void (*push)();
                 int (*pop)();  } operazioni;
    } STACKOBJ;

typedef STACKOBJ *ptr;

void stack_INIT (ptr obj)
{   obj -> stato.TOP = 0;  };

void stack_PUSH (ptr obj, int x)
{   if (obj->stato.TOP == SIZE) printf ("lo stack è saturato");
    else  {  int top;
             top = obj->stato.TOP;
             obj -> stato.S [top] = x;
             obj -> stato.TOP ++;  };
};
```



```
int stack_POP (ptr obj)
{   if (obj->stato.TOP == 0) printf ("lo stack è vuoto");
    else    { int x, top;
              obj->stato.TOP --;
              top = obj->stato.TOP;
              x = obj-> stato.S [top];
              return x; };
};

void stackCREATE (ptr obj)
{
/*inizializzazione della parte di operazioni di un oggetto */
    obj->operazioni.init   = &stack_INIT;
    obj->operazioni.push = &stack_PUSH;
    obj->operazioni.pop  = &stack_POP;
};

main ()
{ /* main */
STACKOBJ OBJ;   stackCREATE(&OBJ);

    /*uso dell'oggetto: inizializzazione top*/
    (*OBJ.operazioni.init)(&OBJ);
    (*OBJ.operazioni.push)(&OBJ, 10);
    (*OBJ.operazioni.push)(&OBJ, 100);
    printf ("primo elemento estratto %d\n",
        (*OBJ.operazioni.pop)(&OBJ));
    printf ("secondo elemento estratto %d\n",
        (*OBJ.operazioni.pop)(&OBJ));
};
```

ESEMPIO 6: OGGETTO STACK ASTRAZIONE CORRETTA

```
/* file di inclusione dell'oggetto stack: stackogg.h */
typedef  struct     {   void     (*   push)  ();
                        int      (*   pop)   ();
                    }   operaztype;
typedef operaztype * ptrinterface;

extern ptrinterface stackCREATE ();


/* file di utilizzo di oggetti stack */

#include "stackogg.h"
main ()  { /* main */
ptrinterface OBJ1, OBJ2;

OBJ1 = stackCREATE();      OBJ2 = stackCREATE();
/*   prima dell'uso dell'oggetto, la sua inizializzazione */

/*   uso dell'oggetto stack creato: sono visibili
le  sole operazioni e non la rappresentazione*/
OBJ1->push(OBJ1, 10);
OBJ1->push(OBJ1, 100);
printf ("primo elemento estratto %d\n", OBJ1->pop(OBJ1));
printf ("secondo elemento %d\n", OBJ1->pop(OBJ1));
     ...
};
```



```
/* file di implementazione degli oggetti  stack */
#define SIZE 20
typedef      struct {       void (* push)();
                            int (*   pop)();
                    } operaztype;
typedef operaztype * ptrinterface;

typedef struct
{    operaztype                          operazioni;
     struct { int TOP;  int S [SIZE]; } stato;
} STACKOBJ;

typedef STACKOBJ *ptr;
/*   il tipo interno alla astrazione allarga la visibilità:
     sono visibili anche la rappresentazione dei dati.
     La creazione alloca spazio per tutta la struttura.
     La visibilità esterna è solo parziale  */

static void stack_PUSH (ptrinterface temp, int x)
{ ptr obj = (ptr) temp;
/* si amplia il tipo ptrinterface */
  if (obj->stato.TOP == SIZE) printf ("lo stack è saturato");
  else
        obj->stato.S [obj->stato.TOP ++] = x;
}
```

```
static int stack_POP (ptrinterface temp)
{ ptr obj = (ptr) temp;  /* ritipaggio */
     if (obj->stato.TOP == 0) printf ("lo stack è vuoto");
       else  return obj->stato.S [-- obj->stato.TOP ];
};

ptrinterface stackCREATE ()

{ ptr obj = (ptr) malloc (sizeof (STACKOBJ));
/*inizializzazione oggetto totale
  sia operazioni sia la parte di stato
*/
     obj->operazioni.push = stack_PUSH;
     obj->operazioni.pop  = stack_POP;
     obj->stato.TOP = 0;
     return (ptrinterface) obj;
};
```

All'esterno è visibile solo una parte della struttura:
La struttura intera è visibile solo all'interno di questo file

====>

La rappresentazione della struttura è **esterna**
ma non visibile

ESEMPIO 7: GESTIONE di un HEAP

uso di una lista con scorrimento a partire dall'inizio e first fit per la allocazione

```
#define NULL 0
#include <alloc.h>

typedef struct header {
     int lung;     /* dimensione della memoria libera */
     struct header *inizio;       /* del blocco stesso */
     struct header *prossimo; /* indirizzo del prossimo */
/*   il blocco di memoria allocato segue in modo contiguo
     l'header **/
                              } HEADER;
static HEADER *  heap;

#define fineb(blocco) \
     ((header *)blocco + 1 + blocco->lung / sizeof (HEADER))
/* espressione per valutare l'indirizzo di fine di un blocco
     (multiplo header).  Notare la aritmetica degli indirizzi */

static HEADER *  cerca  (int size)
{  /* ricerca lo spazio nei blocchi dell'heap */
HEADER *temp=heap;
while ( (temp != NULL) &&
          ((temp -> lung) < (size + sizeof(HEADER))))
                    temp = temp -> prossimo;
return temp;
}
```

```
void * MIAmalloc      (int extsize)
{ HEADER *blocco, *nuovoblocco;
  int size;

size = extsize +  ( (extsize % sizeof (HEADER)) ?
     (sizeof(HEADER) - extsize % sizeof (HEADER)): 0);
/* il size è arrotondato all'header e considera la memoria
   allocabile: non si calcola l'header */

if ((blocco=cerca(size)) == NULL) printf("ERRORE");
else
  {  nuovoblocco = blocco +
              ((blocco->lung - size) /sizeof (HEADER));
/* aritmetica sugli indirizzi */

     nuovoblocco -> inizio = nuovoblocco;
     nuovoblocco -> lung = size;
     nuovoblocco -> prossimo = NULL;
     blocco -> lung -= size + sizeof(HEADER);

     return (void *) (nuovoblocco + 1);
/* aritmetica sugli indirizzi */
  }
}

/* routine di compattamento di due blocchi contigui di heap
*/
void compatta  (HEADER *p, HEADER *s)
{    p-> lung += s-> lung + sizeof(HEADER);
     p-> prossimo = s-> prossimo;
}
```



```
void MIAfree  (void *p)
{ HEADER *vecchioblocco;  HEADER *old, *current;
  vecchioblocco = (HEADER *) p - 1;

   /* inserimento al primo posto nell'heap che si è
      svuotato: il blocco viene aggiunto all'header
      dell'heap */
   if (( heap -> lung == 0 ) && ( fineb(heap) ==
   vecchioblocco))
     { heap -> lung += vecchioblocco -> lung
                                  + sizeof (HEADER);
       if (fineb(heap) == heap -> prossimo)
             compatta (heap, heap-> prossimo);
   /* è possibile che sia necessario un compattamento
      con il primo blocco dell'heap*/
     }

   else


   /* inserimento dopo le indicazioni dell'heap:
      non sono possibili blocchi liberi che
      precedono l'heap stesso */

   {  old = heap; current = heap -> prossimo;

   /* ricerca di due (o un) blocco esistenti nell'heap */
     while ((current != NULL) &&
                    ( fineb (old) > vecchioblocco) ||
                    ( fineb (vecchioblocco) > current))
        { old = current; current = current-> prossimo; }
```

```
 /* inserimento tra blocchi dell'heap */
 if ( (current != NULL) &&
    ( fineb (old) <= vecchioblocco) &&
    ( fineb (vecchioblocco) <= current))
     {   vecchioblocco-> prossimo=current;
         old -> prossimo = vecchioblocco;
         if (fineb(vecchioblocco) == current)
              compatta(vecchioblocco, current);
         if (fineb (old) == vecchioblocco)
              compatta (old, vecchioblocco);
         }
    }

 else
 /* inserimento dopo l'ultimo blocco libero dell'heap*/
    if (vecchioblocco >= fineb(old))
         { if (fineb (old) == vecchioblocco)
                   compatta(old, vecchioblocco);
           else old -> prossimo = vecchioblocco;
         }
  /* inserimento dopo l'ultimo blocco libero dell'heap anche in
   caso di heap svuotato e con solo header*/
   }
}

/*il blocco, assunto con header corretto, viene inserito
  nell'heap unendolo, se possibile, agli altri blocchi liberi
*/
```



```
int CreateHeap (int d)
{
/*l'heap ha una dimensione qualunque, e non
  necessariamente multipla della dimensione dell'header */

 heap = (HEADER *) malloc(d + sizeof(HEADER));
 heap -> lung = d;
 heap -> inizio = heap;
 heap -> prossimo = NULL;
}

void stampaheap ()
{ HEADER * temp = heap; int i=0;
while ( i++, temp != NULL)
{ printf("STAMPAHEAP elemento %d, inizio %p  lung %p\
        prossimo %p, fine %p\n",  i, temp-> inizio, temp->
lung,
        temp->prossimo, fineb(temp));
  temp = temp -> prossimo;
}
}

void stampa ( void * p)
{ HEADER *vecchioblocco;

vecchioblocco = (HEADER *)p - 1;
printf ("BLOCCO indirizzo %p  header %p e lung %p e\
    prossimo %p, fine %p\n", p, vecchioblocco -> inizio,
    vecchioblocco->lung,vecchioblocco->prossimo,
    fineb(vecchioblocco));
}
main ()
{ int size, nbyte, i, choice;
```

```
  void *p [20]; int lung [20];
printf (" size of header %d. Quale dimensione di heap?\n",
sizeof(HEADER));
scanf  ("%d", &size);

CreateHeap(size);
for (i=0; i < 20; i++) lung [i] =0;

while (printf("\n OPERA SU HEAP? 1/0\n"),
        scanf("%d", &choice),
        choice)
{
    do
    {   printf ("\n Operare: malloc/1 free/2 e indice (0/19)\
\nelementi \n");
         for (i=0; i < 20; i++) printf ("%d  ", lung [i]);
         printf("\n"); scanf("%d %d", &choice, &i);
    }

   while ( ((choice !=1) && (choice!=2)) || ((i < 0) || (i > 19)));
```



```
    switch (choice)
   {
    case 1: do
            {   printf ("\n Quanti byte da allocare\n");
                scanf ("%d", &nbyte);
            }
            while ( nbyte <= 0);
            p[i] = MIAmalloc(nbyte);
            stampa (p[i]); lung[i]=nbyte;
            stampaheap ();
            break;
    case 2: printf (" Si libera l'elemento %d lungo %d \n", i,
                    lung[i]);
            MIAfree (p[i]); lung [i]=0;
            stampaheap ();
} }
stampaheap ();
/*  stampa finale dell'heap: deve essere l'iniziale se tutto è
    stato liberato */
}
```

ESEMPIO 8: RICERCA di MATCH di un PATTERN su una STRINGA

Siamo abituati ad usare delle forme abbreviate o pattern che indichino più nomi (vedi UNIX caratteri *, ?, [ ])
È un pattern ***?ab*?**
Il **pattern** fa il match con le stringhe che hanno almeno un carattere qualunque che precede i caratteri ab seguiti da almeno un carattere ulteriore

Manteniamo il programma di pattern matching precedente
Notiamo che il confronto di pattern con wild card richiede un algoritmo più complesso: l'asterisco richiede di fare una ricerca su tutte le possibili posizioni della stringa.
L'idea è di dichiarare insuccesso (o successo) appena si hanno gli elementi per farlo; altrimenti si va avanti a scandire

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>
int patmatch ( char * string, char * pattern);
void main  (argc, argv)   int argc;  char **argv;
{     char * s = (char *)malloc (180);
  if (argc != 2)
      {     printf (" errore:\n Necessario 1 argomento per ricerca"); exit (-1); }
 while ( gets (s))
     if (patmatch (s, argv[1]))
          printf ("Match tra %s e \n%s\n", argv[1]);
     else     printf ("NO Match tra %s e \n%s\n", argv[1]);
}
```

Esaminiamo per il momento una versione limitata del pattern matching, con riconoscimento di ***, ?,** e **caratteri normali**


```
int patmatch ( char * string, char * pattern)
{  register char * p, *s;    register char c; /* corrente */
p = pattern;   s = string;

for (;;) {
if (*p == '\0')    if ( *s == '\0') return 1;
                                 else return 0;
/* se il pattern è finito ed è finita la stringa, OK (restituisce 1)
    altrimenti insuccesso  (restituisce 0).
    Il caso contrario di pattern non esaurito e stringa finita,
   può portare a match  (si pensi a pattern con *)
*/

switch (c = *p++)
{
 case '?':
/* almeno un carattere nella stringa */
     if (*s++ == '\0') return 0;
/* insuccesso, se aspettiamo un carattere */
     else       break;   /* andiamo oltre con la scansione */

 case '*':
/* match con nessuno o qualunque numero di caratteri nella
   stringa */
/* in questo caso ci aspettiamo di fare ricerche ripetute fino
   ad un match o ad una esplorazione completa della stringa
   */
```

```
    c = *p;
/*  consideriamo il carattere successivo c nel pattern
   mantenendo l'analisi sulla posizione p:
   se è normale e nella stringa non c'è altro, insuccesso
    altrimenti scandiamo oltre */
    if (c != '?' && c != '*' && c != '\0')
       while (*s != c)
            if (*s == '\0') return 0;  else  s++;  /* oltre */

/* consideriamo oltre le wildcard del pattern:
   dobbiamo considerare tutti gli eventuali possibili altri
   match in altre posizioni, rimanendo sulla stessa posizione
   della stringa; se non ci sono possibilità, insuccesso*/

/* scansione ricorsiva della stringa e del pattern */
    do
        if ( patmatch (s, p)) return 1;
    while (*s++ != '\0');

/* stringa finita senza match, insuccesso*/
    return 0;

 default:
/* per caratteri diversi proclamiamo insuccesso,
   altrimenti scandiamo oltre  */
          if (*s++ != c) return 0;
          break;
  }
 }
}
```


Ci ispiriamo alla **Bourne shell**

Consideriamo un **pattern matching** con altre wild card
[ab] (un carattere tra quelli specificati)
[a-g] (un carattere nel range specificato)
[!0-9] (**!** rappresenta la **negazione**)
almeno un carattere non compreso tra 0 e 9

```
int patmatch ( char * string, char * pattern)
{  register char * p, *s;    register char c;

p = pattern;   s = string;
for (;;) {
if (*p == '\0')    if ( *s == '\0') return 1;  else return 0;

switch (c = *p++)
{
 case '?':
     if (*s++ == '\0') return 0;   else break;

 case '*':
     c = *p;
     if (c != '?' && c != '*' && c != '[' && c != '\0')
          while (*s != c)  if (*s == '\0') return 0;  else  s++;

     do    if (patmatch(s, p)) return 1;
     while (*s++ != '\0');

     return 0;

case '[':
 {char *endp, chr;  int invert, found;
/* usiamo chr per il carattere nella stringa
```

```
            endp per puntare alla fine (al carattere ])
            invert se abbiamo trovato !
            found se c'è il carattere  nel range o pattern */

    if (*s == '\0') return 0;
/* se non c'è nessun carattere nella stringa, insuccesso;
   altrimenti saltiamo in caso di errore */

    endp = p;   /*  endp cerca la ] */
    if (*endp == '!')     endp++;  /* salta la negazione */

/* cerca la chiusura del pattern segnalata da ]  */
    for (;;)
        {  if (*endp == '\0')
            if (*s++ != '\0') return 0;
            else return 1;
   /* in caso di fine pattern, se la stringa è finita insuccesso,
      altrimenti successo */
            if (*++endp == ']')  break;
        }

/* endp punta a ]
   selezioniamo i casi di negazione o meno attraverso invert
   e usiamo found per verificare il match o meno del
  carattere nel pattern specificato tra [ ]
*/

    invert = 0;
    if (*p == '!') { invert++;  p++; }
    found = 0; chr = *s++;  c = *p++;
/* *p salta l'eventuale negazione ed il carattere è ora in c,
   il corrispondente nel pattern in chr */
```



```
/* scandiamo tutte le possibilità nel pattern e la stringa
  se c'è un range, vediamo se il carattere è compreso
  se un carattere, vediamo se fa match */
    do {
      if (*p == '-' && p[1] != ']')
        { p++;
          if (chr >= c && chr <= *p) found = 1;  /* range */
          p++;
        }
      else if (chr == c) found = 1;  /* carattere uguale */
        }
    while ((c = *p++) != ']');

/* found segnala che abbiamo torvato il match */
    if (found == invert) return 0;   else break;
/* due casi di insuccesso:
   se abbiamo trovato match e c'era negazione,
   se non c'è match e pattern senza negazione (no !)
   negli altri casi si continua a scandire */
}

/* come prima */
 default:          if (*s++ != c) return 0;   break;
} } }
```

Progetto di FILTRI

Filtro che taglia i caratteri numerici (non li porta in uscita)

```
#include <stdio.h>
main()
    {       int c;
      while ((c = getchar()) != EOF)
        if ( ( c < '0' ) || ( c > '9') )
        putchar(c);
    }
```

Altri Filtri

```
#include <stdio.h>
main()
    {       int c;
      while ((c = getchar()) != EOF)
        if ( ( c != 'a' ) && ( c != 'b') && ( c != 'c'))
        putchar(c);
    }
```

```
#include <stdio.h>
main()
    {   int c;
    while ((c = getchar()) != EOF)
      if  ( c != '\n' )      putchar(c);
}
```


FILTRO SIPC (SOLO_I_PRIMI_CARATTERI)

```
#include <stdio.h>
#include <string.h>

void main(argc, argv)
int argc;  char *argv[ ];
{   int c, n, count = 0;

if (argc != 2)
    { printf("errore: Usage %s numero\n", argv[0]);
      exit(1);}

c=0;
while (argv[1][c])
 { if (argv[1][c] < '0' || argv [1][c] > '9')
    { printf("Argomento numerico! NON %s\n", argv[1]);
      exit(2);}
    c++;
  }

n= atoi (argv[1]);
/* atoi da solo non basta: cosa vale atoi(1b4)?  */

  while ((c = getchar()) != EOF)
{  if  (count < n )  putchar(c);
    count++;
 }
exit (0);
}
```

FILTRO
filtro numero c_1 c_2 c_3 c_4 c_5 ... c_n

*filtra le linee che contengono almeno **numero** caratteri degli n passati come argomenti*

```
#include <stdio.h>
#include <string.h>
/* la funzione può essere definita direttamente qui */
int strchr (char *s, char c);

void main(argc, argv)
int argc;  char *argv[ ];
{    char linea[80];  int i,cont;
if (argc < 3)
  { printf("errore: pochi parametri\n"); exit(1);}
for (i = 0; (i < strlen (argv[1])) &&
                    argv[1][i] >= '0' && argv [1][i] <= '9'; i++) ;
if ( i < strlen (argv[1]) || (atoi(argv[1]) > argc-2))
  { printf("Errore sul primo argomento\n"); exit(2);}

while(gets(linea))
 { cont=0;
     for (i=2 ; i<argc ; i++) if (strchr(linea,*argv[i])) cont++;
     if (cont >= atoi(argv[1])) puts(linea);
}}

int strchr (char *s, char c)
 { while (*s)  if ( *s++ == c) return 1;
   return 0;
 }
```


FILTRO ancora

```
#include <stdio.h>
#include <string.h>
void main(argc, argv)
int argc;  char *argv[ ];
{    char linea[80];  int i,cont;

if (argc < 3)
  { puts("errore: pochi parametri"); exit(1);}

for (i = 0; (i < strlen (argv[1])) &&
                    argv[1][i] >= '0' && argv [1][i] <= '9'; i++) ;
if ( i < strlen (argv[1]) || (atoi(argv[1]) > argc-2))
  { puts("Errore sul primo argomento"); exit(2);}

while(gets(linea))
 { cont=0;
     for (i=2 ; i<argc ; i++)
          if (strstr (linea,argv[i])) cont++;
     if (cont >= atoi(argv[1])) puts(linea);
 }
}
```

strstr è definita nella libreria delle stringhe e ricerca la prima occorrenza di una stringa in un'altra

*char ** strstr*(char ** totalstring*, char ** searchstring*)*

21 Febbraio 1997

Si progetti in linguaggio C il filtro **filtro** che si possa ridirigere sia in ingresso, sia in uscita. **filtro** può accettare fino a 4 argomenti, ciascuno un carattere, di tipo diverso. I tipi accettati sono i seguenti: carattere maiuscolo, carattere minuscolo, carattere numerico, carattere non alfanumerico.

filtro deve lavorare sul file di ingresso: viene portato in uscita un insieme di caratteri pari all'insieme dei caratteri di ingresso. Per ogni carattere in ingresso, a secondo del tipo, si produce in uscita o un carattere di default o il corrispondente carattere eventualmente fornito dagli argomenti:

- per ogni carattere alfabetico maiuscolo, in uscita o 'A' (default) o dall'argomento relativo;
- per ogni carattere alfabetico minuscolo, o 'a' (default) o dall'argomento relativo;
- per ogni carattere numerico, o '0' (default) o dall'argomento relativo;
- per ogni carattere non alfanumerico, si porta in uscita o '@' (default) o dall'argomento relativo;
- i fine linea sono portati in uscita sempre.

```
#include <stdio.h>
#include <string.h>
#define MAIUSCOLO    0
#define MINUSCOLO    1
#define NUMERICO     2
#define NOTALFANUM   3
main  (argc, argv)
int argc;   char **argv;
{       int ch, j;
        char chm, chM, chn, chnot;
         int cegia [4] = {0,0,0,0};

/* controllo sul numero massimo argomenti */
  if (argc > 5)
   { printf (" errore:\n al massimo 4 argomenti"); exit (1); }
```



```
/* controllo sulla correttezza degli argomenti  e
   la loro non ripetizione*/
  for (j = 1; j < argc; j++)
  {
     if (strlen (argv[j]) > 1)
       { printf (" errore:\n al massimo 1 carattere in arg %d\n",
j);
          exit (2); }

/* il vettore cegia contiene la indicazione della presenza di
   un argomento di un tipo, le variabili ch* li memorizzano
*/
     if (argv[j][0] >= 'A' && argv[j][0] <= 'Z')
        if ( cegia [MAIUSCOLO])
        { printf (" errore:\n al massimo 1 maiuscolo\n"); exit (3); }
     else {chM = argv[j][0]; cegia [MAIUSCOLO] = 1;}

    if (argv[j][0] >= 'a' && argv[j][0] <= 'z')
         if ( cegia [MINUSCOLO])
         { printf (" errore:\n al massimo 1 minuscolo\n"); exit (3);}
    else {chn = argv[j][0]; cegia [MINUSCOLO] = 1;}

   if (argv[j][0] >= '0' && argv[j][0] <= '9')
    if ( cegia [NUMERICO])
    { printf (" errore:\n al massimo 1 maiuscolo\n"); exit (3); }
    else {chn = argv[j][0]; cegia [NUMERICO] = 1;}

   if ( ! isdigit (argv[j][0]) && ! isalpha (argv[j][0]) )
    if ( cegia [NOTALFANUM])
    { printf (" errore:\n al massimo 1 maiuscolo\n"); exit (3); }
     else {chnot = argv[j][0]; cegia [NOTALFANUM] = 1;}
  }
```

```
while ((ch = getchar ()) != EOF)
{/* non necessario, fatto a default if (ch == '\n') putchar (ch);
*/
  if (ch == '\n') putchar (ch);

 else
 {
  if (( ch >= 'A') && (ch <= 'Z'))
    if ( cegia [MAIUSCOLO]) ch = chM; else  ch = 'A';

  if (( ch >= 'a') && (ch <= 'z'))
    if ( cegia [MINUSCOLO]) ch = chm; else  ch = 'a';

  if (( ch >= '0') && (ch <= '9'))
    if ( cegia [NUMERICO]) ch = chn; else  ch = '0';

  if ( ! isdigit (ch) && ! isalpha (ch))
    if ( cegia [NOTALFANUM]) ch = chnot; else  ch = '@';

/* in ogni caso, da input o dal default
   stampa il carattere in ch */

  putchar (ch);
  }
 }
}
```


FILTRO EXPR

Scrivere in C un filtro che lavora sugli argomenti che specificano una espressione aritmetica: operatori binari **+** e **-** e valori interi. Si vuole fornire il risultato in uscita

expr 2 + 1 - 5 ⇒ ho ottenuto -2

expr 2 + - 5 + 4 ⇒ errore

```
#include <stdio.h>
#include <string.h>

void main(argc, argv)
int argc;  char *argv[ ];
{    char linea[80];
       int j, i, K, cont=0,  addsub =1;

if (argc <  2)
  { puts("Errore:  argomenti operatori e numerici"); exit(1);}
else
if ( argc % 2 == 1)
  { puts("Errore:  argomenti dispari operatori e numerici"); exit(1);}
else
{
/* controllo argomenti numerici */
  for (i = 1; i < argc; i++, i++)
     for (K = 0; (K < strlen (argv[i])); K++)
       if ( argv[i][K] < '0' || argv [i][K] > '9')
     { printf("Errore di argomento %c", argv[i][K]); exit(2);}
```

```
/* controllo argomenti operandi */
for (i = 2; i < argc - 1; i++, i++)
     if ( argv[i][0] != '+' && argv[i][0] != '-')
      {   printf("Arg: %s non operando \n", argv[i]);
          exit(2);}
}

for (i=1; i < argc; i++)
 if (i % 2 ) if (addsub) cont += atoi(argv[i]);
        else cont -= atoi(argv[i]);
 else addsub = argv[i][0] == '+' ? 1: 0;

printf(" ho ottenuto %d\n", cont);
}
```

Estendere il filtro **expr** a considerare altri operatori e confrontarlo con il comando corrispondente di UNIX