# Passaggio di matrici come parametri

- Quando passiamo un vettore ad una funzione, passiamo in realtà il puntatore (costante) all'elemento di indice 0.

  $\Longrightarrow$ non serve specificare la dimensione del vettore nel parametro formale.
- Quando passiamo una matrice ad una funzione, per poter accedere correttamente agli elementi, la funzione deve conoscere il numero di colonne della matrice.

  $\Longrightarrow$ Non possiamo specificare il parametro nella forma `mat[][]`, come per i vettori, ma dobbiamo specificare il numero di colonne.

  **Esempio:** `void stampa(int mat[][5], int righe) {...}`
- Il motivo è semplice: per accedere ad un generico elemento della matrice, `mat[i][j]`, la funzione deve calcolare l'indirizzo di tale elemento `mat + offset`. Per calcolare correttamente `offset` è necessario sapere il numero di colonne `C`.
- L'indirizzo di `mat[i][j]` è infatti:

  `mat + (i · C · sizeof(int)) + (j · sizeof(int))`





Riassumendo:

- per calcolare l'indirizzo dell'elemento `mat[i][j]` è necessario conoscere:
  - il valore di `mat`, ovvero l'indirizzo del primo elemento della matrice
  - l'indice di riga `i` dell'elemento
  - l'indice di colonna `j` dell'elemento
  - il numero `C` di colonne della matrice
- In generale, in un parametro di tipo array vanno specificate tutte le dimensioni, tranne eventualmente la prima.
  1. vettore: non serve specificare il numero di elementi
  2. matrice: bisogna specificare il numero di colonne, ma non serve il numero di righe

## Esercizio

Definire le funzioni/procedure utilizzate nel seguente programma e completare con gli opportuni parametri attuali la chiamata di `swap` in modo che il suo effetto sia di scambiare gli elementi minimo e massimo del vettore.

```
#include <stdio.h>
#define LUNG 10

void leggivet (int [] vet, int dim);
void stampavet (int [] vet, int dim);
int indice_minimo (int vet[], int dim);
int indice_massimo (int vet[], int dim);
void swap (int *, int *);

main()
{
  int vettore[LUNG], pos_min, pos_max;

  leggivet(vettore, LUNG);
  pos_min = indice_minimo(vettore, LUNG);
  pos_max = indice_massimo(vettore, LUNG);
  swap (?, ?);   /* scambio degli elementi minimo e massimo */
  printf("Vettore dopo lo scambio dell'elemento minimo e massimo:\n");
  stampavet(vettore, LUNG);
}
```

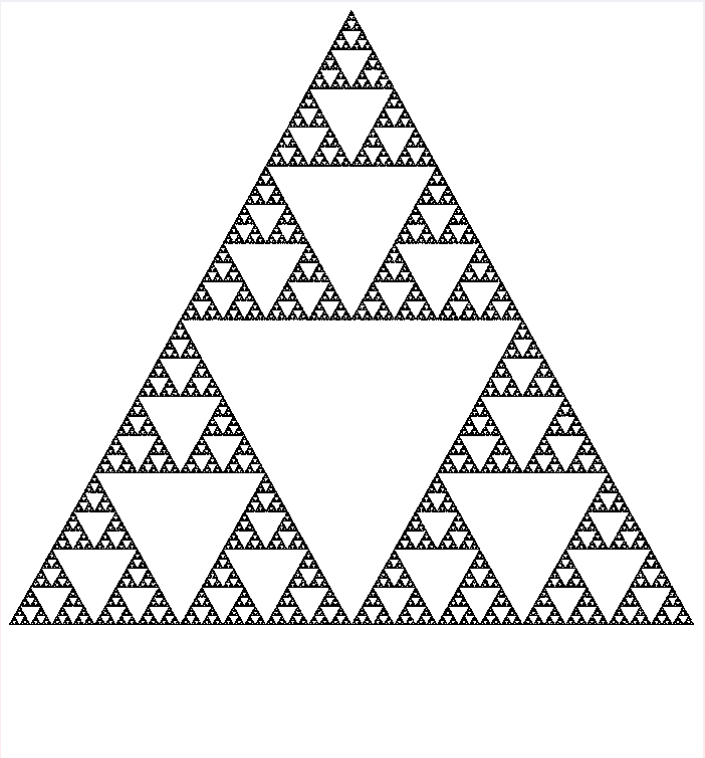




# Ricorsione: C'era una volta un Re

- C'era una volta un Re
  seduto sul sofà
  che disse alla sua serva
  raccontami una storia
  e la serva incominciò:
  - C'era una volta un Re
    seduto sul sofà
    che disse alla sua serva
    raccontami una storia
    e la serva incominciò:
    - C'era una volta un Re
      seduto sul sofà...

# Il triangolo di Sierpinskyi







# Programmazione ricorsiva: cenni

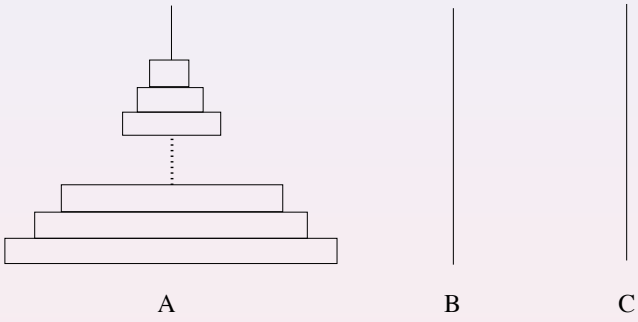
- In quasi tutti i linguaggi di programmazione evoluti è ammessa la possibilità di definire funzioni/procedure ricorsive: durante l'esecuzione di una funzione F è possibile chiamare la funzione F stessa.
- Ciò può avvenire
  - direttamente: il corpo di F contiene una chiamata a F stessa.
  - indirettamente: F contiene una chiamata a G che a sua volta contiene una chiamata a F.
- Questo può sembrare strano: se pensiamo che una funzione è destinata a risolvere un sottoproblema $\mathcal{P}$, una definizione ricorsiva sembra indicare che per risolvere $\mathcal{P}$ dobbiamo . . . saper risolvere $\mathcal{P}$!

- In realtà, la programmazione ricorsiva si basa sull'osservazione che per molti problemi la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema.
- La programmazione ricorsiva trova radici teoriche nel principio di induzione ben fondata che può essere visto come una generalizzazione del principio di induzione sui naturali
- La soluzione di un problema viene individuata supponendo di saperlo risolvere su casi più semplici.
- Bisogna poi essere in grado di risolvere direttamente il problema sui casi più semplici di qualunque altro.



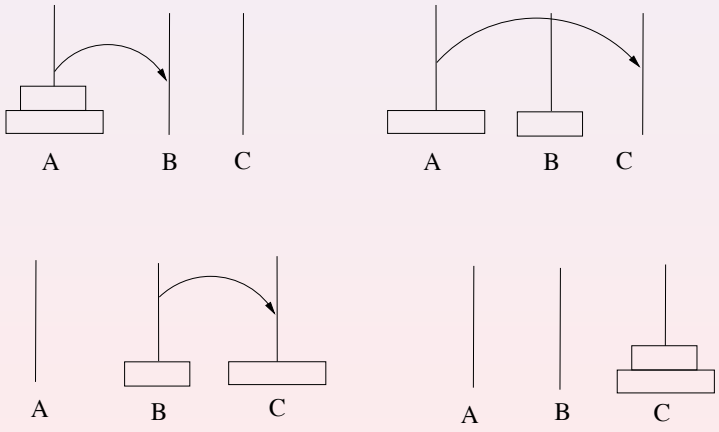


**Esempio:** Torre di Hanoi (leggenda Vietnamita).




- pila di dischi di dimensione decrescente su un perno A
- vogliamo spostarla sul perno C, usando un perno di appoggio B
- vincoli:
  - possiamo spostare un solo disco alla volta
  - un disco più grande non può mai stare su un disco più piccolo
- secondo la leggenda: i monaci stanno spostando 64 dischi: quando avranno finito, ci sarà la fine del mondo

- Come individuare una soluzione per un numero N di dischi arbitrario?
  - per N=1 la soluzione è immediata: spostiamo l'unico disco da A a C
  - se sappiamo risolvere il problema per N=1 lo sappiamo risolvere anche per N=2: come?

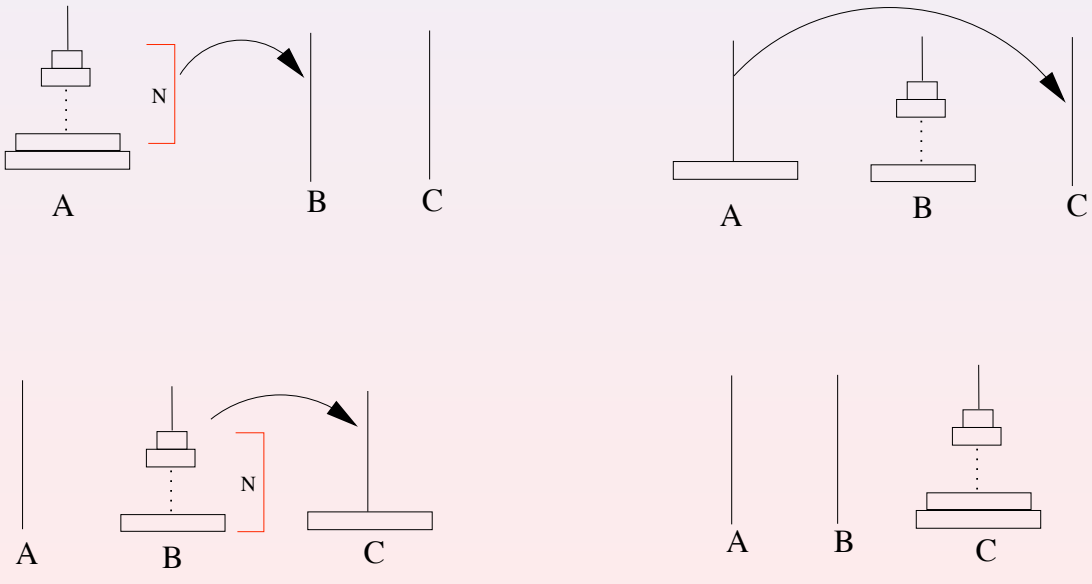



- Notiamo l'utilizzo del perno ausiliario B





- Possiamo generalizzare il ragionamento? Se sappiamo risolvere il problema per N dischi, possiamo individuare una soluzione per lo stesso problema ma con N+1 dischi?



- Formalizziamo il ragionamento
- Indichiamo con hanoi(N, P1, P2, P3) il problema: "spostare N dischi dal perno P1 al perno P2 utilizzando P3 come perno d'appoggio".

```
hanoi(N, P1, P2, P3)
     if (N=1)
         sposta  da P1 a P2;
     else
       {
         hanoi(N-1, P1, P3, P2);
         sposta da P1 a P2;
         hanoi(N-1, P3, P2, P1);
       }
```





**Esempio:** Soluzione di hanoi(3,A,C,B)

$$
\begin{aligned}
\text{hanoi}(3,A,C,B) = {} & \text{hanoi}(2,A,B,C) = \begin{cases} \text{hanoi}(1,A,C,B) = \text{sposta}(A,C) \\ \text{sposta}(A,B) \\ \text{hanoi}(1,C,B,A) = \text{sposta}(C,B) \end{cases} \\
& \text{sposta}(A,C) \\
& \text{hanoi}(2,B,C,A) = \begin{cases} \text{hanoi}(1,B,A,C) = \text{sposta}(B,A) \\ \text{sposta}(B,C) \\ \text{hanoi}(1,A,C,B) = \text{sposta}(A,C) \end{cases}
\end{aligned}
$$

# Quante mosse per $N$ dischi?

Si può dimostrare, per induzione sul numero di dischi $N$, che il numero di mosse è:

$$Mosse(N) = \begin{cases} 1 & \text{se } N = 1 \quad \text{(caso base)} \\ 2^N - 1 & \text{se } N > 1 \quad \text{(caso induttivo)} \end{cases}$$

Supponendo che ogni mossa duri un secondo, i monaci avrebbero da lavorare per più di 580 miliardi di anni.
Sapendo che l'universo ha circa una ventina di miliardi di anni, possiamo dire che i monaci ne avranno ancora per molto ☺





- Le funzioni ricorsive sono convenienti per implementare funzioni matematiche definite in modo induttivo.
  **Esempio:** Definizione induttiva di somma tra due interi non negativi:

$$somma(x, y) = \begin{cases} x & \text{se } y{=}0 \\ 1 + (somma(x, y-1)) & \text{se } y > 0 \end{cases}$$

  - La somma di x con 0 viene definita in modo immediato;
  - la somma di x con il successore di y viene definita come il successore della somma tra x e y.
- **Esempio:** somma di 3 e 2:

$$\begin{aligned} somma(3,2) = \quad & 1 + (\underline{somma(3,1)}) = \\ & 1 + (1 + (\underline{somma(3,0)})) = \\ & 1 + (1 + (3)) = \\ & 1 + 4 = \\ & 5 \end{aligned}$$

**Esempio:** Funzione fattoriale.

- definizione iterativa: $fatt(n) = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- definizione induttiva:

$$fatt(n) = \begin{cases} 1 & \text{se } n = 0 \quad \text{(caso base)} \\ n \cdot fatt(n-1) & \text{se } n > 0 \quad \text{(caso induttivo)} \end{cases}$$

- È essenziale il fatto che, applicando ripetutamente il caso induttivo, ci riconduciamo prima o poi al caso base.

$$\begin{aligned} fatt(3) = \; & 3 \cdot \underline{fatt(2)} = \\ & 3 \cdot (2 \cdot \underline{fatt(1)}) = \\ & 3 \cdot (2 \cdot (1 \cdot \underline{fatt(0)})) = \\ & 3 \cdot (2 \cdot (1 \cdot 1)) = \\ & 3 \cdot (2 \cdot 1) = \\ & 3 \cdot 2 = \\ & 6 \end{aligned}$$





Il codice delle due diverse versioni

- definizione iterativa:

```
int fatt(int n) {
    int i,ris;

    ris=1;
    for (i=1;i<=n;i++)
      ris=ris*i;
    return ris;
}
```

- definizione ricorsiva:

```
int fattric(int n) {
  if (n == 0)
    return 1;
  else
    return n * fattric(n-1);
 }
```

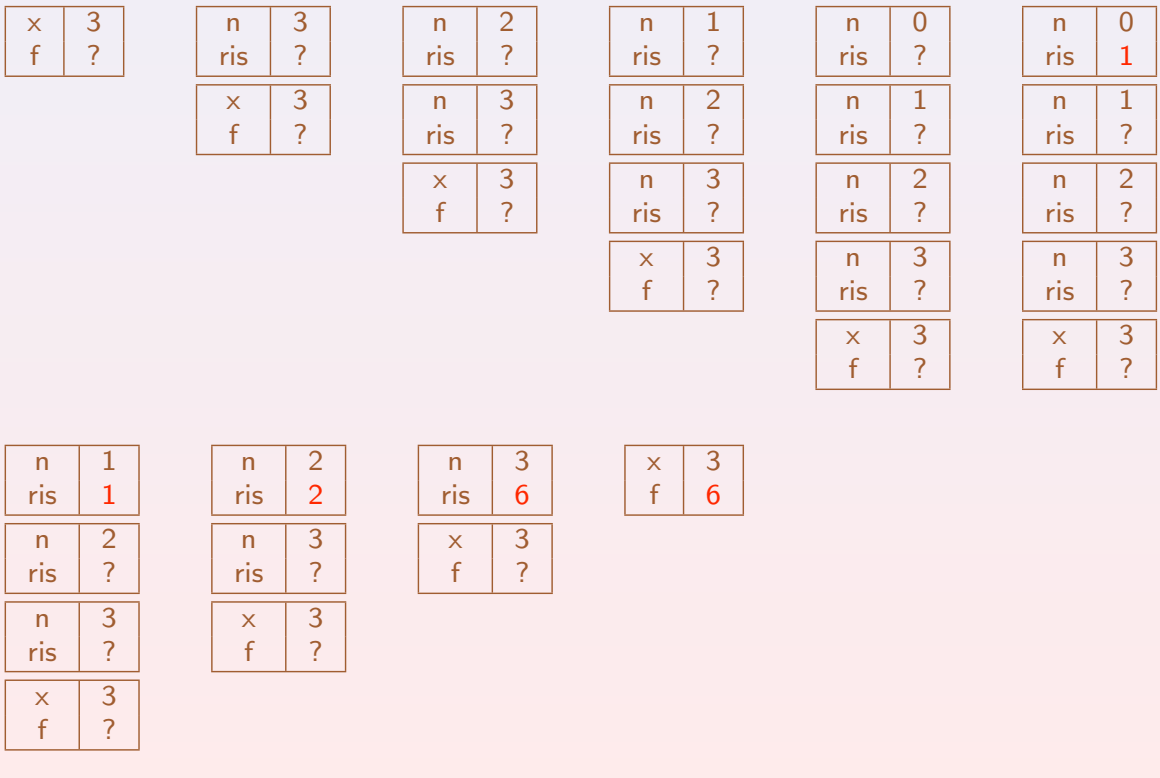
**Esempio:** Programma che usa una funzione ricorsiva.

```
#include <stdio.h>

int fattric (int);

main()
{
int x, f;
scanf("%d", &x);
f = fattric(x);
printf("Fattoriale di %d:  %d\n", x, f);
}

int fattric(int n) {
  int ris;
  if (n == 0)
    ris = 1;
  else
    ris = n * fattric(n-1);
  return ris;
}
```





## Evoluzione della pila (supponendo x=3).



**Esempio:** Leggere una sequenza di caratteri terminata da `'\n'` e stamparla invertita. Ad esempio: `rosa` $\Longrightarrow$ `asor`

- Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma dinamica (liste, le vedremo più avanti)
  2. usando un procedimento ricorsivo.
     - leggiamo un carattere della sequenza, `c1`, leggiamo e stampiamo ricorsivamente il resto della sequenza `c2...cn` e infine stampiamo `c1`;
     - il caso base è la lettura del carattere di fine sequenza.

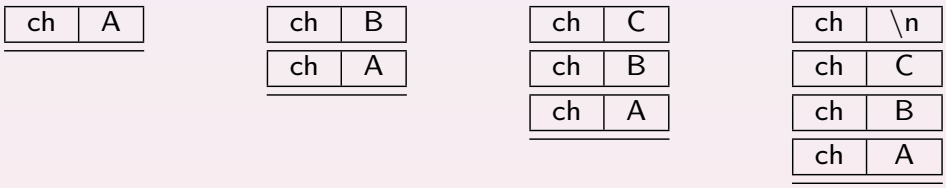
```
void invertInputRic()
 {  char ch;

  ch = getchar();
  if (ch != '\n')
    {
    invertInputRic();
    putchar(ch);
    }
  else
    printf("Sequenza invertita: ");
}
```

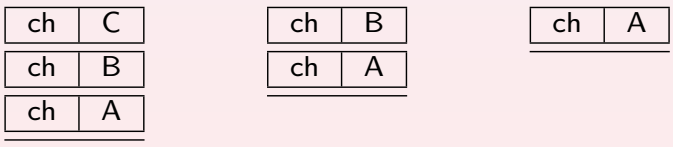




```
 main()
{
  printf("Immetti una sequenza di caratteri\n");
  invertInputRic();
  printf("\n");
}
```

Vediamo come si evolve la pila per l'input `ABC\n`




L'output prodotto è il seguente

```
Sequenza invertita:  CBA
```

# Ricorsione multipla

- Si ha ricorsione multipla quando un'attivazione di una funzione può causare più di una attivazione ricorsiva della stessa funzione (es. torre di Hanoi)

  **Esempio:** Definizione induttiva dei numeri di Fibonacci.

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-2) + F(n-1) \qquad \text{se } n > 1 \end{aligned}$$

- $F(0)$, $F(1)$, $F(2)$, ... è detta sequenza dei numeri di Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...





```
 #include <stdio.h>

int fibonacci (int);

main() {
  int n;

  printf("Inserire un intero >= 0: ");
  scanf("%d", &n);
  printf("Numero %d di Fibonacci: %d\n", n, fibonacci(n));
}

 int fibonacci(int i)
  {
  int ris;
  if (i == 0)
    ris = 0;
  else if (i == 1)
    ris = ;
  else
    ris =  fibonacci(i-1) + fibonacci(i-2);
  return ris;
  }
```

# Esempi di funzioni ricorsive

- Tradurre in C la definizione induttiva già vista:

$$somma(x, y) = \begin{cases} x & \text{se } y = 0 \\ 1 + (somma(x, y-1)) & \text{se } y > 0 \end{cases}$$

```
int somma (int x, int y)
{
  int ris;
  if (y==0)
     ris = x;
  else
     ris = 1 + somma(x, y-1);
  return ris;
}
```





- Calcolo ricorsivo di $x^y$ (si assume $y \geq 0$)

$$x^y = \begin{cases} 1 & \text{se } y = 0 \\ x \cdot x^{y-1} & \textit{altrimenti} \end{cases}$$

```
int exp (int x, int y)
{
  int ris;
  if (y==0)
     ris = 1;
  else
     ris = x * exp(x, y-1);
  return ris;
}
```

- Calcolare ricorsivamente la somma degli elementi nella porzione di un array v compresa tra gli indici from e to.
- Esprimiamo formalmente quanto richiesto:

$$sumVet(v, from, to) = \sum_{i=from}^{to} v[i]$$

- È evidente che:

$$\sum_{i=from}^{to} v[i] = \begin{cases} 0 & \text{se } from > to \\ v[from] + \sum_{i=from+1}^{to} v[i] & \text{se } from \leq to \end{cases}$$

- La traduzione in C è immediata.





```
int sumVet(int *v, int from, int to)
{
  if (from > to)
    return 0;
  else
    return v[from] + sumvet(v,from+1,to);
}
```

```
int sumVet(int *v, int from, int to)
{
  int somma;
  if (from > to)
    somma = 0;
  else
    somma = v[from] + sumvet(v,from+1,to);
  return somma;
}
```

- Calcolare ricorsivamente il numero di occorrenze dell'elemento x nella porzione di un array v compresa tra gli indici from e to.

$$f(v, x, from, to) = \#\{i \in [from, to] \mid v[i] = x\}$$

- Anche in questo caso ragioniamo induttivamente:

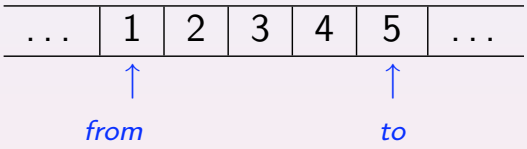
$$f(v, x, from, to) = \begin{cases} 0 & \text{se } from > to \\ f(v, x, from + 1, to) & \text{se } from \leq to \wedge v[from] \neq x \\ 1 + f(v, x, from + 1, to) & \text{se } from \leq to \wedge v[from] = x \end{cases}$$





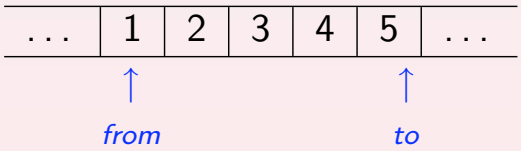
```
int occorrenze (int *v, int x, int from, int to)
{
  int occ;

  if (from > to)
    occ= 0;
  else
    if (v[from]!=x)
        occ = occorrenze(v,x,from+1,to);
    else
        occ = 1+occorrenze(v,x,from+1,to);
}
```

- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici from e to.




- Vogliamo ottenere:

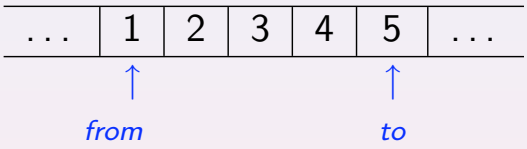

- Induttivamente:



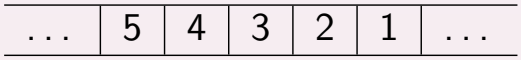




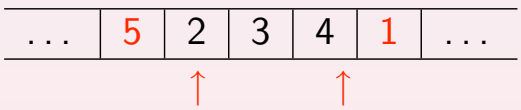

- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici from e to.




- Vogliamo ottenere:




- Induttivamente:




- Questa situazione corrisponde alla chiamata ricorsiva su una porzione più piccola del vettore

```
void swap(int *v, int i, int j)
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

void invertiric (int *v, int from, int to)
{
  if (from < to)
    {
        swap(v, from, to);
        invertiric(v, from+1, to-1);
    }
}
```

- Si noti che la procedura non fa niente se la porzione individuata dal secondo e terzo parametro è vuota (from>to) o contiene un solo elemento (from=to)





# Tipi user-defined

- Il C mette a disposizione un insieme di tipi di dato predefiniti (tipi built-in) e dei meccanismi per definire nuovi tipi (tipi user-defined)
- Vediamo le regole generali che governano la definizione di nuovi tipi e quindi i costrutti linguistici (costruttori) che il C mette a disposizione.
- Tutti i tipi non predefiniti utilizzati in un programma devono essere dichiarati come ogni altro elemento del programma. Una dichiarazione di tipo viene fatta di solito nella parte dichiarativa del programma.
  - parte dichiarativa globale:
    - dichiarazioni di costanti
    - dichiarazioni di tipi
    - dichiarazioni di variabili
    - prototipi di funzioni/procedure

# Dichiarazione di tipo

- Una dichiarazione di tipo (type declaration) consiste nella parola chiave typedef seguita da:
  - la rappresentazione o costruzione del nuovo tipo (ovvero la specifica di come è costruito a partire dai tipi già esistenti)
  - il nome del nuovo tipo
  - il simbolo ; che chiude la dichiarazione

  **Esempio:** `typedef int anno;`
- Una volta definito e nominato un nuovo tipo, è possibile utilizzarlo per dichiarare nuovi oggetti (ad es. variabili) di quel tipo.

  **Esempio:**
  ```
  float x;
  anno a;
  ```
- **Nota:** In `C` si possono anche definire tipi senza usare `typedef`. Quest'ultima consente l'associazione di un nome (identificatore) a un nuovo tipo. Per uniformità e leggibilità del codice useremo spesso `typedef` per definire nuovi tipi.





# Tipi semplici user-defined

**Ridefinizione**: Un nuovo tipo può essere definito rinominando un tipo già esistente (cioè creandone un alias)

**typedef TipoEsistente NuovoTipo;**

dove `TipoEsistente` può essere un tipo built-in o user-defined.

**Esempio:**
```
typedef int anno;
typedef int naturale;

typedef char carattere;
```

**Enumerazione:** Consente di definire un nuovo tipo enumerando i suoi valori, con la seguente sintassi

**typedef enum {v1, v2, ... , vk} NuovoTipo;**

**Esempio:**
```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
typedef enum {gen, feb, mar, apr, mag, giu,
              lug, ago, set, ott, nov, dic} Mese;

typedef enum {m, f} sesso;
```

- I valori elencati nella definizione di un nuovo tipo enumerato, sono identificatori che rappresentano costanti di quel tipo (esattamente come `0, 1, 2, ...` sono costanti del tipo `int`, o `'a', 'b', ...` sono costanti del tipo `char`).
- Dunque, se dichiariamo una variabile
  `Giorno g;`
  possiamo scrivere l'assegnamento
  `g = mar;`
- Le costanti dei tipi enumerati non vanno racchiuse tra virgolette o tra apici!

  N.B. Il compilatore associa ai nomi utilizzati per denotare le costanti dei tipi enumerati valori naturali progressivi.

  **Esempio:** il valore associato a `g` dopo l'assegnamento `g=mar` è il numero naturale (intero) `1`.

  $\Longrightarrow$ mancanza di astrazione: è possibile fare riferimento alla rappresentazione dei valori.





- La relazione tra interi e tipi enumerati consente di applicare a questi ultimi le seguenti operazioni:
  - operazioni aritmetiche: `+,-,*,/,%`
  - uguaglianza e disuguaglianza: `=, !=`
  - confronto: `<,<=,>,>=`
- Si noti che la relazione di precedenza tra i valori (che determina l'esito delle operazioni di confronto) dipende dall'ordine in cui vengono elencati i valori del tipo al momento della sua definzione.

  **Esempio:** Con le dichiarazioni viste in precedenza
  `lun < gio` è vero (un intero diverso da `0`) `apr <= feb` è falso (il valore intero `0`)
- Il C tratta questi tipi come ridefinizione di `int`

# Tipi fai da te: i booleani

Soluzione 1

```
typedef int Boolean;
Boolean b; ...
```

Soluzione 2

```
#define FALSE 0;
#define TRUE 1;...
typedef int Boolean;
Boolean b;
...
```

Soluzione 3

```
typedef enum {FALSE, TRUE} Boolean;...
Boolean b;
...
```

N.B. I valori vanno elencati come sopra, rispettando la convenzione adottata dal C: il valore 0 rappresenta falso.





**Esempio:**

```
typedef enum {false, true} boolean;

boolean even (int n)
{
if (n % 2 == 0)
    return true;
else
    return false;
}


boolean implies (boolean p, boolean q)
{
    if (p)
        return q;
    else
        return true;
}
```

**Esempio:** Uso del costrutto `switch` con tipi enumerati

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
Giorno g;
...
switch (g) {
case lun: case mar: case mer: case gio: case ven:
        printf("Giorno lavorativo");
        break;
case sab: case dom:
        printf("Week-end");
        break;
}


void stampaGiorno(Giorno g) {
switch (g) {
case lun: printf("lun");
          break;
...

case dom: printf("dom");
          break;
}
```





# Tipi strutturati user-defined

- Il `C` non possiede tipi strutturati built-in, ma fornisce dei costruttori che permettono di definire tipi strutturati anche piuttosto complessi.
- Array e puntatori possono essere visti come costruttori di tipo (definiscono un tipo di dato non semplice a partire da tipi esistenti).

**Uso di `typedef` con array e puntatori**

- In generale, una dichiarazione di tipo mediante `typedef` ha la forma di una dichiarazione di variabile preceduta dalla parola chiave `typedef`, e con il nome di tipo al posto del nome della variabile.
- Nel caso di array e puntatori:

```
typedef TipoElemento TipoArray[Dimensione];
typedef TipoPuntato *TipoPuntatore;
```

**Esempio:**

```
typedef int ArrayDieciInteri[10];
typedef int MatriceTreXQuattro[3][4];
typedef int *PuntIntero;
ArrayDieciInteri vet;           /*  int vet[10]; */
PunIntero p;                    /*  int *p; */
MatriceTreXQuattro mat, mat1;   /*  int mat[3][4]; int mat1[3][4]; */
```