

## Ritorniamo sulle Procedure/Funzioni

- ▶ Come astrazione delle istruzioni, le procedure possono dover **modificare lo stato**.

**Esempio:** Procedura `abs` che **assegna** ad una variabile intera il suo valore assoluto

- ▶ il chiamante deve comunicare alla procedura la variabile in questione
- ▶ la procedura deve analizzare il valore della variabile e, se necessario, effettuare il rimpiazzamento
- ▶ La seguente realizzazione della procedura non è corretta
- ▶ Simuliamo il comportamento di una chiamata della procedura (come visto in precedenza)

```
int z = -5;
```

```
abs(z);
```

```
⇒
```

```
x = -5;
```

```
if (x < 0) x = -x;
```

- ▶ La modifica del parametro formale **non** si ripercuote sul parametro attuale (si ricordi che il passaggio dei parametri è **per valore**).

## Passaggio dei parametri per indirizzo

- ▶ Per ottenere l'effetto di modificare il valore dei parametri attuali, alcuni linguaggi (es. Pascal) prevedono un'ulteriore modalità di passaggio dei parametri, il passaggio **per indirizzo**
  - ▶ informalmente: il passaggio per indirizzo fa in modo che, al momento della chiamata, il parametro formale costituisca un modo **alternativo** per accedere al parametro attuale (che **deve** essere una variabile e non può essere una generica espressione)
  - ▶ durante l'esecuzione del corpo, ogni riferimento (e in particolare modifica) al parametro formale è di fatto un riferimento al parametro attuale.
- ▶ In C esiste solo il passaggio per valore, ma quello per indirizzo si può realizzare come segue:
  1. si utilizza un parametro formale di tipo **puntatore**
  2. all'interno del corpo della procedura ogni riferimento al parametro formale avviene attraverso l'operatore di dereferenziazione **\***
  3. al momento della chiamata, si utilizza come parametro attuale un indirizzo di una variabile (utilizzando, se necessario, l'operatore di indirizzo **&**).

**Esempio:** Riprendiamo l'esempio del valore assoluto.

```
void abs(int *x)
{
    if (*x < 0)
        *x = -(*x);
}
```

Nel chiamante si utilizzano chiamate del tipo `abs(&z)` per ottenere l'effetto di rimpiazzare il valore della variabile `z` con il suo valore

```
int z = -5;
```

assoluto. `abs(&z);`  $\implies$  `x = &z;` **N.B.**  
`if (*x < 0) *x = -(*x);`

Il passaggio dei parametri è sempre per **valore**, ma questa volta viene passato un valore puntatore che consente di accedere alla variabile del chiamante. Nell'esempio, l'assegnamento `*x = -(*x);` ha come effetto la modifica della variabile `z` del chiamante, in quanto il valore di `x` è `&z` e dunque `*x` è proprio `z`.

## Esempio: Procedura per lo scambio di due variabili intere

```
► void swap(int *x, int *y)
```

```
{
```

```
    int temp = *x;
```

```
    *x = *y;
```

```
    *y = temp;
```

```
} Esempio di utilizzo: programma che legge due valori interi e li stampa ordinati.
```

```
main() {
```

```
    int a, b;
```

```
    scanf("%d", &a);
```

```
    scanf("%d", &b);
```

```
    if (a > b) swap(&a, &b);
```

```
    printf("Valore minimo: %d\n", a);
```

```
    printf("Valore massimo: %d\n", b); }
```

## Parametri di tipo vettore

- ▶ Il meccanismo del passaggio **per valore** di un **indirizzo** consente il passaggio di vettori come parametri di funzioni/procedure.
- ▶ Quando si passa un vettore come parametro ad una funzione, in realtà si sta passando l'indirizzo dell'elemento di indice **0**.
- ▶ Il parametro formale deve essere di tipo **puntatore** (al tipo degli elementi del vettore)
- ▶ di solito si passa anche la dimensione del vettore in un ulteriore parametro.

### Esempio:

```
void stampaVettore(int *v, int dim)
{ int i;
  for (i = 0; i < dim; i++)
    printf("v[%d]: %d\n", i, v[i]);
}
```

```
main()
{ int vet[5] = {1, 2, 3, 4, 5};
  ...
  stampaVettore(vet, 5);    ... }
```

- ▶ Per evidenziare che il parametro formale è un vettore (ovvero l'indirizzo dell'elemento di indice 0), si può utilizzare la notazione `nome-parametro[]` invece di `*nome-parametro`.

**Esempio:** `void stampa(int v[], int dim) { ... }`

- ▶ Si può anche specificare la dimensione nel parametro, ma questa viene ignorata.

**Esempio:** `void stampa(int v[5], int dim) { ... }`

- ▶ Come al solito, nel prototipo della funzione il nome del parametro (vettore) può anche mancare.

**Esempio:** `void stampa(int [], int);`

- ▶ Il passaggio di un vettore è un **passaggio per indirizzo**.  
⇒ La funzione può modificare gli elementi del vettore passato.

**Esempio:** Lettura di un vettore.

```
void leggiVettore(int v[], int dim)
{
    int i;
    for (i = 0; i < dim; i++)
    {
        printf("Immettere l'elemento di indice %d: ", i);
        scanf("%d", &v[i]);
    }
}
```

## Esempio: Programma che legge, inverte e stampa un vettore di interi

```
#include <stdio.h>
#define LUNG 5

void leggiVettore(int [], int);
void stampaVettore(int [], int);
void invertiVettore(int [], int);

main()
{
    int vett[LUNG];

    leggiVettore(vett, LUNG);
    printf("Vettore prima dell'inversione\n");
    stampaVettore(vett, LUNG);

    invertiVettore(vett, LUNG);
    printf("Vettore dopo l'inversione\n");
    stampaVettore(vett, LUNG);
}
```

La definizione della procedura `void invertiVettore(int [], int);` è lasciata per **esercizio**.



## Passaggio di matrici come parametri

- ▶ Quando passiamo un **vettore** ad una funzione, passiamo in realtà il puntatore (costante) all'elemento di indice 0.  
 ⇒ **non** serve specificare la dimensione nel par. formale.
- ▶ Quando passiamo una **matrice** ad una funzione, per poter accedere correttamente agli elementi, la funzione deve conoscere **il numero di colonne** della matrice.  
 ⇒ 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 quante sono le colonne.
- ▶ L'indirizzo di `mat[i][j]` è infatti:  

$$\text{mat} + (i \cdot C \cdot \text{sizeof}(\text{int})) + (j \cdot \text{sizeof}(\text{int}))$$
 dove **C** è il numero di colonne (gli elementi in ciascuna riga).

## 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 leggiwet (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;

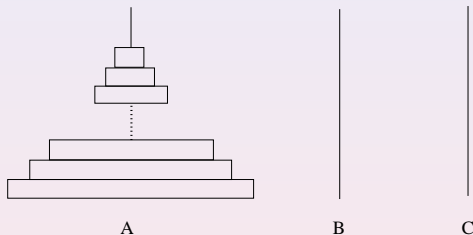
    leggiwet(vettore, LUNG);
    pos_min = indice_minimo(vettore, LUNG);
    pos_max = indice_massimo(vettore, LUNG);
    swap (?, ?); /* scambio degli elementi minimo e massimo */
    printf("Vettore dopo lo scambio dell'elemento minimo e massimo:\n");
    stampavet(vettore, LUNG);
}
```

# 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  $\mathcal{P}$ !

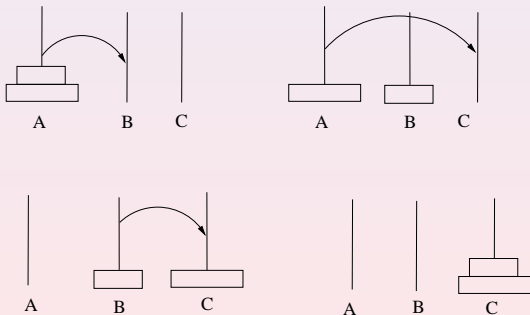
- ▶ In realtà, la programmazione ricorsiva si basa sull'osservazione che per molti problemi **la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema.**
- ▶ La programmazione ricorsiva trova radici teoriche nel **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).



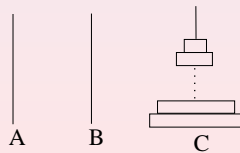
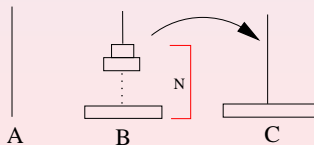
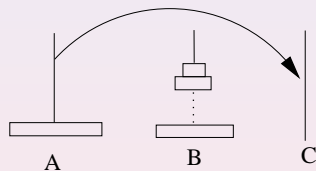
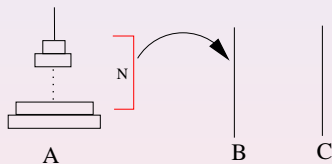
- ▶ pila di dischi di dimensione decrescente su un perno A
- ▶ vogliamo spostarla sul perno C, usando un perno di appoggio B
- ▶ vincoli:
  - ▶ possiamo spostare un solo disco alla volta
  - ▶ un disco più grande non può mai stare su un disco più piccolo
- ▶ secondo la leggenda: i monaci stanno spostando 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)`

|                               |                               |                               |                          |
|-------------------------------|-------------------------------|-------------------------------|--------------------------|
|                               |                               | <code>hanoi(1,A,C,B) =</code> | <code>sposta(A,C)</code> |
|                               | <code>hanoi(2,A,B,C) =</code> | <code>sposta(A,B)</code>      |                          |
|                               |                               | <code>hanoi(1,C,B,A) =</code> | <code>sposta(C,B)</code> |
| <code>hanoi(3,A,C,B) =</code> | <code>sposta(A, C)</code>     |                               |                          |
|                               |                               | <code>hanoi(1,B,A,C) =</code> | <code>sposta(B,A)</code> |
|                               | <code>hanoi(2,B,C,A) =</code> | <code>sposta(B,C)</code>      |                          |
|                               |                               | <code>hanoi(1,A,C,B) =</code> | <code>sposta(A,C)</code> |

- ▶ 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  $2$ :

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

**Esempio:** Funzione fattoriale.

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

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

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

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

## 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  $x=3$ ).

|   |   |
|---|---|
| x | 3 |
| f | ? |

|     |   |
|-----|---|
| n   | 3 |
| ris | ? |

|   |   |
|---|---|
| x | 3 |
| f | ? |

|     |   |
|-----|---|
| n   | 2 |
| ris | ? |

|     |   |
|-----|---|
| n   | 3 |
| ris | ? |

|   |   |
|---|---|
| x | 3 |
| f | ? |

|     |   |
|-----|---|
| n   | 1 |
| ris | ? |

|     |   |
|-----|---|
| n   | 2 |
| ris | ? |

|     |   |
|-----|---|
| n   | 3 |
| ris | ? |

|   |   |
|---|---|
| x | 3 |
| f | ? |

|     |   |
|-----|---|
| n   | 0 |
| ris | ? |

|     |   |
|-----|---|
| n   | 1 |
| ris | ? |

|     |   |
|-----|---|
| n   | 2 |
| ris | ? |

|     |   |
|-----|---|
| n   | 3 |
| ris | ? |

|   |   |
|---|---|
| x | 3 |
| f | ? |

|     |   |
|-----|---|
| n   | 0 |
| ris | 1 |

|     |   |
|-----|---|
| n   | 1 |
| ris | ? |

|     |   |
|-----|---|
| n   | 2 |
| ris | ? |

|     |   |
|-----|---|
| n   | 3 |
| ris | ? |

|   |   |
|---|---|
| x | 3 |
| f | ? |

|     |   |
|-----|---|
| n   | 1 |
| ris | 1 |

|     |   |
|-----|---|
| n   | 2 |
| ris | ? |

|     |   |
|-----|---|
| n   | 3 |
| ris | ? |

|   |   |
|---|---|
| x | 3 |
| f | ? |

|     |   |
|-----|---|
| n   | 2 |
| ris | 2 |

|     |   |
|-----|---|
| n   | 3 |
| ris | ? |

|   |   |
|---|---|
| x | 3 |
| f | ? |

|     |   |
|-----|---|
| n   | 3 |
| ris | 6 |

|   |   |
|---|---|
| x | 3 |
| f | ? |

|   |   |
|---|---|
| x | 3 |
| f | 6 |

**Esempio:** Leggere una sequenza di caratteri terminata da `'\n'` e stamparla invertita. Ad esempio: `casa`  $\implies$  `asac`

- ▶ 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 rappresentato dalla 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 evolve la pila per l'input `ABC\n`

|    |   |
|----|---|
| ch | A |
|----|---|

|    |   |
|----|---|
| ch | B |
|----|---|

|    |   |
|----|---|
| ch | C |
|----|---|

|    |    |
|----|----|
| ch | \n |
|----|----|

|    |   |
|----|---|
| ch | A |
|----|---|

|    |   |
|----|---|
| ch | B |
|----|---|

|    |   |
|----|---|
| ch | C |
|----|---|

|    |   |
|----|---|
| ch | A |
|----|---|

|    |   |
|----|---|
| ch | B |
|----|---|

|    |   |
|----|---|
| ch | A |
|----|---|

|    |   |
|----|---|
| ch | C |
|----|---|

|    |   |
|----|---|
| ch | B |
|----|---|

|    |   |
|----|---|
| ch | A |
|----|---|

|    |   |
|----|---|
| ch | B |
|----|---|

|    |   |
|----|---|
| ch | A |
|----|---|

|    |   |
|----|---|
| ch | A |
|----|---|

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.

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-2) + F(n-1) \quad \text{se } n > 1$$

- ▶  $F(0), F(1), F(2), \dots$  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 = 1;
    else
        ris = fibonacci(i-1) + fibonacci(i-2);
    return ris;
}
```

## Esempi di funzioni ricorsive

- ▶ Tradurre in C la definizione induttiva già vista:

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

```
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  $y \geq 0$ )

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

```
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 di un array  $v$  compreso tra gli indici  $j$  e  $dim-1$ .
- ▶ Esprimiamo formalmente quanto richiesto:

$$sumVet(v, j, dim - 1) = \sum_{i=j}^{d-1} v[i]$$

- ▶ È evidente che:

$$\sum_{i=j}^{d-1} v[i] = \begin{cases} 0 & \text{se } dim = 0 \\ v[j] + \sum_{i=j+1}^{d-1} v[i] & \text{se } dim > 0 \end{cases}$$

- ▶ La traduzione in **C** è immediata.

```
int sumVet(int *v, int dim)
{
    if (dim==0)
        return 0;
    else
        return v[0] + sumvet(v+1,dim-1);
}
```

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

- ▶ Calcolare ricorsivamente il numero di occorrenze dell'elemento  $x$  nell'array  $v$  compreso tra gli indici  $j$  e  $dim-1$ .

$$f(v, x, j, dim) = \#\{i \in [j, dim - 1] \mid v[i] = x\}$$

- ▶ Anche in questo caso ragioniamo induttivamente:

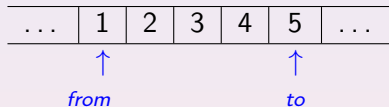
$$f(v, x, j, dim) = \begin{cases} 0 & \text{se } dim = 0 \\ f(v, x, j + 1, dim) & \text{se } dim > 0 \wedge v[j] \neq x \\ 1 + f(v, x, j + 1, dim) & \text{se } dim > 0 \wedge v[j] = x \end{cases}$$



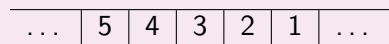
```
int occorrenze (int *v, int dim)
{
    int occ;

    if (dim==0)
        occ= 0;
    else
        if (v[0]!=x)
            occ = occorrenze(v+1,x,dim-1);
        else
            occ = 1+occorrenze(v+1,x,dim-1);
}
```

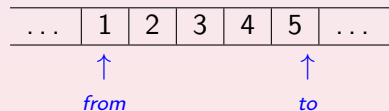
- ▶ Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici *from* e *to*.



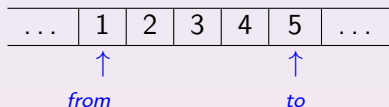
- ▶ Vogliamo ottenere:



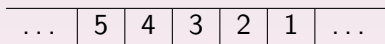
- ▶ Induttivamente:



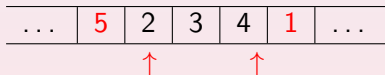
- ▶ Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici *from* e *to*.



- ▶ 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 ( $from = to$ )