Variable-lenght Array (VLA)

- C99 introduce alcune nuove caratteristiche al linguaggio C, alcune già implementate come estensioni in alcuni compilatori.
- C99 introduce i variable-lenght array strutture dati allocate sullo stack la cui lunghezza e' determinata a run time (invece che a tempo di compilazione).
  Esempio: `int vet[exp];`
  dove exp e' una qualsiasi espressione che puo' coinvolgere variabili.
- Una volta trovata la dichiarazione pero' la dimensione e' fissata una volta per tutte.

Array vs. VLA

Svantaggi dovuta all'allocazione sullo stack:

- Le architetture hanno limiti piuttosto bassi per lo stack i VLA devono avere dimensioni ridotte.
- Hanno diverse limitazioni di uso. Ad esempio non possono essere dichiarati come extern. *" Array objects declared with the static or extern storage-class specifier cannot have a variable length array (VLA) type"*.
- L'array e' gestito come una variabile automatica: la sua vita termina quando termina la funzione *readProcess*.

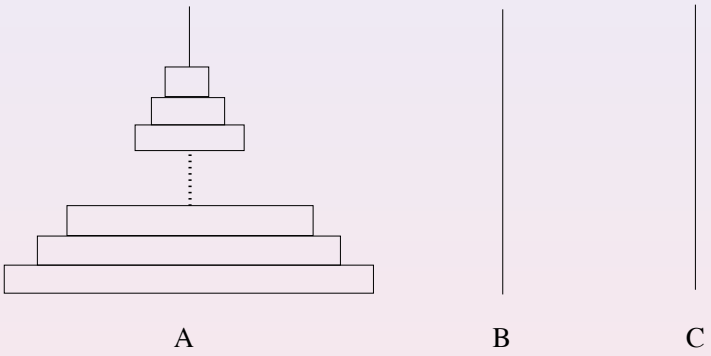
```
float readProcess(int n)
{
    float vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return process(vals, n);
}
```

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 $\mathcal{P}$!

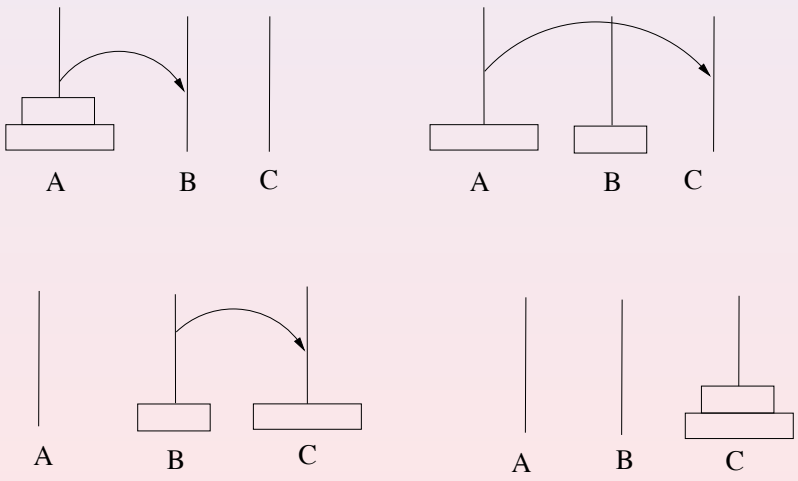
- In realtà, la programmazione ricorsiva si basa sull'osservazione che per molti problemi la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema.
- La programmazione ricorsiva trova radici teoriche nel principio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



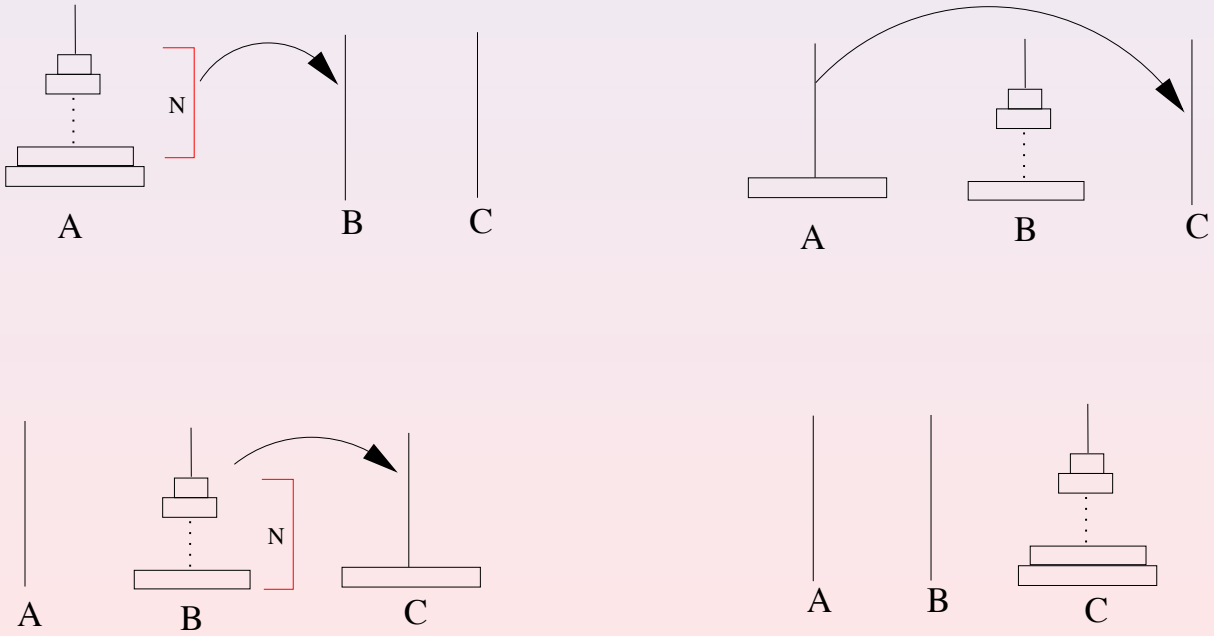
- pila di dischi di dimensione decrescente su un perno A
- vogliamo spostarla sul perno C, usando un perno di appoggio B
- vincoli:
  - possiamo spostare un solo disco alla volta
  - un disco più grande non può mai stare su un disco più piccolo
- secondo la leggenda: i monaci stanno spostando 64 dischi: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

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

- 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:

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

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.

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

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;
}
```

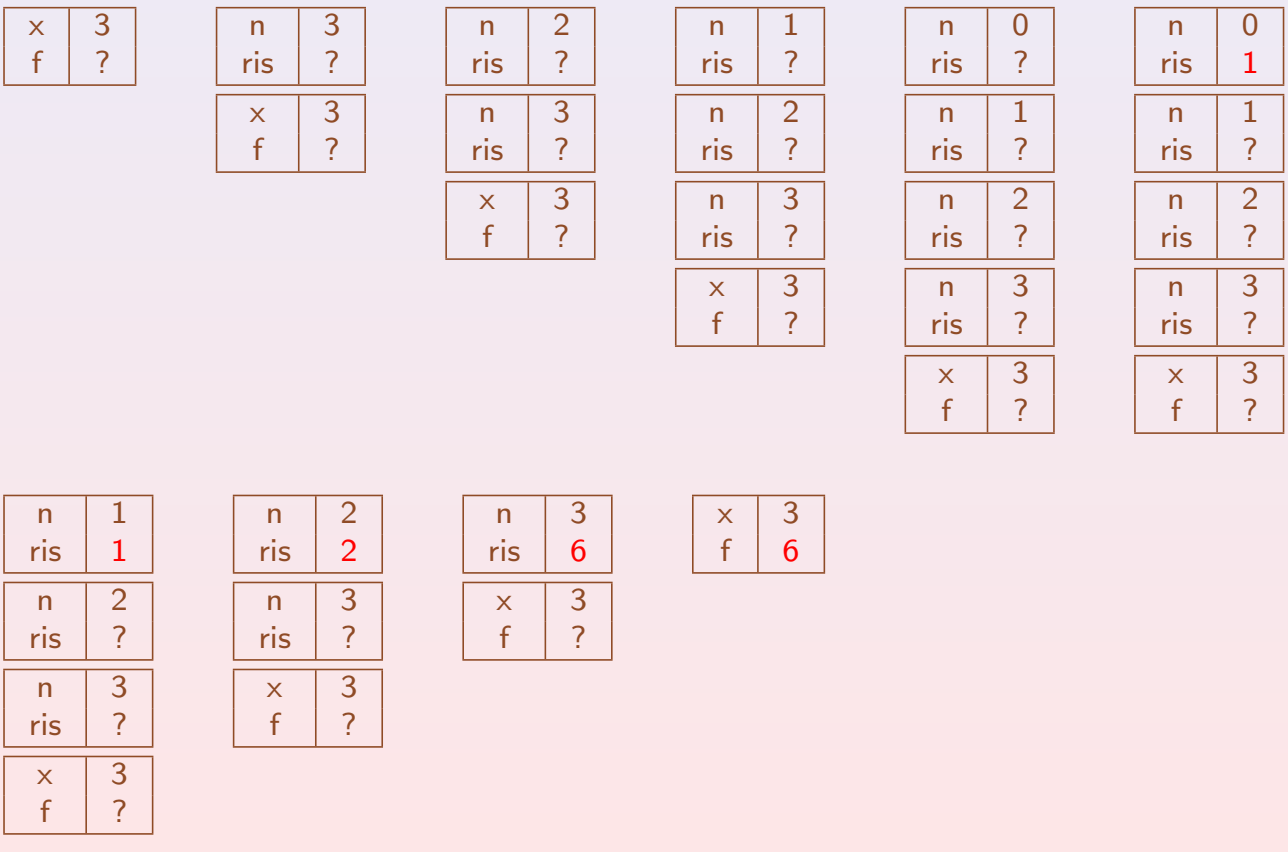
- definizione ricorsiva:

```
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: casa $\Longrightarrow$ asac

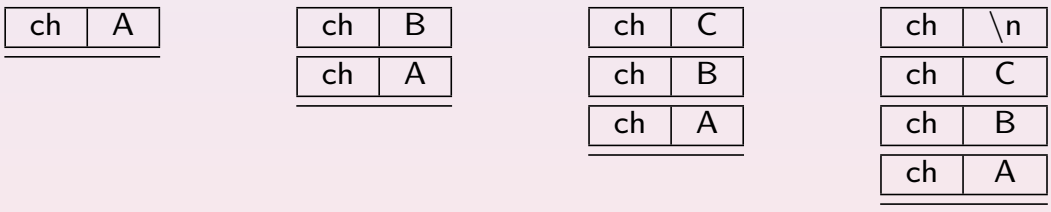
- Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma dinamica (liste, le vedremo più avanti)
  2. usando un procedimento ricorsivo.
     - leggiamo un carattere della sequenza, c1, leggiamo e stampiamo ricorsivamente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

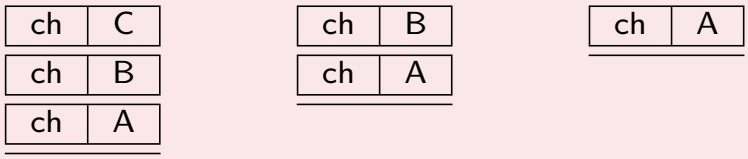
```
void invertInputRic()
 {  char ch;

  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





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.

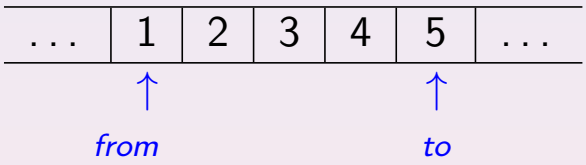
$$f(v, x, from, to) = \#\{i \in [from, to] \mid v[i] = x\}$$

- Anche in questo caso ragioniamo induttivamente:

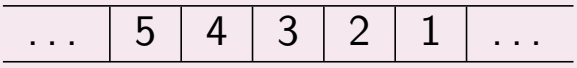
$$f(v, x, from, to) = \begin{cases} 0 & \text{se } from > to \\ f(v, x, from + 1, to) & \text{se } from \leq to \wedge v[from] \neq x \\ 1 + f(v, x, from + 1, to) & \text{se } from \leq to \wedge v[from] = x \end{cases}$$


```
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);
}
```

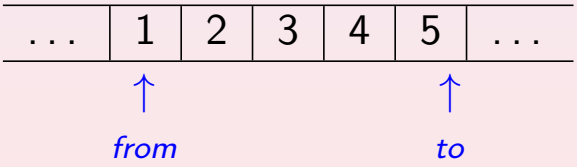
- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici from e to.



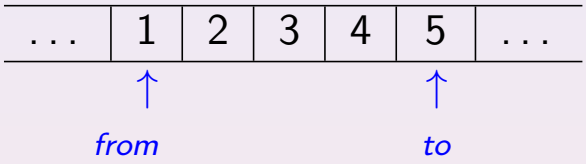
- Vogliamo ottenere:



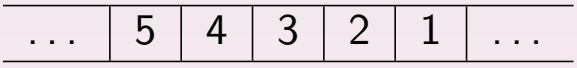
- Induttivamente:



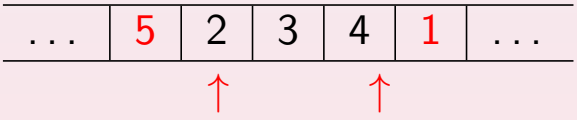
- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici from e to.



- 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)

Problema:

Data una sequenza di elementi in ordine qualsiasi, ordinarla.

- Questo è un problema fondamentale, che si presenta in moltissimi contesti, ed in diverse forme.
- Nel nostro caso formuliamo il problema in termini di ordinamento di vettori:

  Dato un vettore A di n elementi, ordinarlo in modo crescente
- Per semplicità faremo sempre riferimento a vettori di interi.

$$\begin{array}{ccc} & 5\ 2\ 4\ 6\ 1\ 3 & \\ \Longrightarrow & & \\ & 1\ 2\ 3\ 4\ 5\ 6 & \end{array}$$

Ordinamento per inserzione (insertion sort)

- **Esempio:** Ordinamento di una mano di ramino
  - si inizia con la mano sinistra vuota e le carte coperte sul tavolo
  - si prende dalla tavola una carta alla volta e la si inserisce nella corretta posizione nella mano sinistra
  - ...
  - si termina quando si sono finite tutte le carte sul tavolo.
- Stesso procedimento per ordinare un vettore:
  - inizialmente il vettore rappresenta il mazzo sul tavolo
  - si usa un ciclo per analizzare uno alla volta gli elementi del vettore
  - Alla generica iterazione la situazione è la seguente

mano sinistra	carte ancora da scoprire

↑ nuova carta

  - per inserire la nuova carta al posto giusto nella mano sinistra dobbiamo
    - scorrere gli elementi che lo precedono per decidere la posizione che gli compete
    - spostare di un posto verso destra gli elementi maggiori per fargli spazio.

Esempio

In verde le carte ancora da esaminare, in rosso quelle già esaminate (mano sinistra). La nuova carta da esaminare è sottolineata.

5	2	4	6	1	3
5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

- Una volta individuata la posizione k in cui inserire la nuova carta dobbiamo farle spazio, ovvero spostare verso destra di una posizione tutte le carte rosse da k in poi.

  Esempio:

1	2	4	5	6	3
1	2	4	5	6	3
1	2	4	5		6
1	2	4		5	6
1	2		4	5	6

- A questo punto possiamo piazzare la carta in modo ordinato

1	2	3	4	5	6

- Definiamo allora una procedura che sposta tutti gli elementi di un vettore verso destra di una posizione tra due indici dati `from` e `to`

```
void shiftR(int v[], int from, int to)
{
int i;
for (i=to-1; i>=from; i--)
  v[i+1] = v[i];
}
```

- L'elemento in posizione `to` viene perso
- Bisogna procedere da destra verso sinistra (perché?)
- se `to` è minore o uguale a `from` non succede nulla

Possiamo allora definire la procedura di ordinamento per inserzione come segue

```
void sort(int v[], int dim)
{
int h, curr, j=1;
while (j<dim)
  { h=0;
    curr=v[j];
          /* curr e' l'elemento da piazzare */

    while((v[h]<curr) && (h<j))
       h++;
          /* curr va inserito in posizione h */

    shiftR(v,h,j);
     v[h]=curr;
    j++;
   }
}
```


Possiamo anche rendere la procedura più efficiente, nel modo seguente

```
void sort (int v[], int dim) {
  int i, j, prossimo;
  for (i = 1; i < dim; i++) {
     prossimo = v[i];
     j = i;
     while ((j > 0) && (v[j-1] > prossimo)) {
        v[j] = v [j-1];
        j--;
     }
     v[j] = prossimo;
  }
}
```

Ordinamento per selezione del minimo (selection sort)

- **Esempio:** Ordinamento di un mazzo di carte
  - si seleziona la carta più piccola e si mette da parte
  - delle rimanenti si seleziona la più piccola e si mette da parte
  - ...
  - si termina quando rimane una sola carta
- Ordinamento di un vettore:
  - per selezionare l'elemento più piccolo tra quelli rimanenti si utilizza un ciclo
  - mettere da parte significa scambiare con l'elemento che si trova nella posizione che compete a quello selezionato

- in verde la parte che rimane da analizzare
  in blu l'elemento minimo selezionato
  in marrone lo scambio effettuato
  in rosso la parte ordinata

5	2	4	6	1	3	
	5	2	4	6	1	3
	1	2	4	6	5	3
1	2	4	6	5	3	
	1	2	4	6	5	3
	1	2	4	6	5	3
1	2	4	6	5	3	
	1	2	4	6	5	3
	1	2	3	6	5	4
1	2	3	6	5	4	
	1	2	3	6	5	4
	1	2	3	4	5	6
1	2	3	4	5	6	
	1	2	3	4	5	6
	1	2	3	4	5	6
1	2	3	4	5	6	
1	2	3	4	5	6	

Implementazione

```
int minPos(int v[], int from, int to);
/* calcola la posizione del minimo elemento di
   v nella porzione [from,to]                */

void swap(int *p, int *q);
/* scambia le variabili puntate da p e q */

/** PROCEDURA DI ORDINAMENTO PER SELEZIONE **/

void sort(int v[], int dim)
{
  int i, min;
  for(i=0; i<dim-1; i++)
    {
      min = minPos(v, i, dim-1);
      swap(v+i, v+min);
    }
}
```

Scrivere per **esercizio** le procedure `swap` e `minpos`


```
int minPos(int v[], int from, int to) {
/* calcola la posizione del minimo elemento di
   v nella porzione [from,to]                */

    int i, pos;
    pos = from;
    for (i=from+1; i<=to; i++)
        if (v[i] < v[pos])
            pos = i;
    return pos;
}

void swap(int *p, int *q) {
/* scambia le variabili puntate da p e q */
    int temp = *p;
    *p = *q;
    *q = temp;
}
```

Ordinamento a bolle (bubble sort)

- Si fanno salire gli elementi più piccoli ("più leggeri") verso l'inizio del vettore ("verso l'alto"), scambiandoli con quelli adiacenti.
- L'ordinamento è suddiviso in n-1 fasi:
  - fase 0: 0^o elemento (il più piccolo) in posizione 0
  - fase 1: 1^o elemento in posizione 1
  - ...
  - fase n-2: $(n-2)^o$ elemento in posizione n-2, e quindi $(n-1)^o$ elemento in posizione n-1
- Nella fase i: cominciamo a confrontare dal basso e portiamo l'elemento più piccolo (più leggero) in posizione i

5	2	4	6	1	3	
	5	2	4	6	1	3
	5	2	4	1	6	3
	5	2	1	4	6	3
	5	1	2	4	6	3
	1	5	2	4	6	3
1	5	2	4	6	3	
	1	5	2	4	3	6
	1	5	2	3	4	6
	1	5	2	3	4	6
	1	2	5	3	4	6
1	2	5	3	4	6	
1	2	3	5	4	6	
1	2	3	4	5	6	
1	2	3	4	5	6	

```
/** PROCEDURA BUBBLE SORT **/


void sort(int v[], int dim)
{
  int temp,i,j;
  for (i = 0; i < dim-1; i++)     /* fase i-esima */

    for (j = dim-1; j > i; j--)   /* bolla piu' leggera in posizione i */
      if (v[j] < v[j-1])
        swap(v+j, v+j-1);
}
```


Ordinamenti ricorsivi

Selection Sort ricorsivo

- Il metodo del selection sort può essere facilmente realizzato in modo ricorsivo
- si definisce una procedura che ordina (ricorsivamente) la porzione di array individuata da due indici `from` e `to`
- il minimo elemento della porzione viene messo in posizione `from` per poi ordinare ricorsivamente la porzione tra `from+1` e `to`
- Il caso base corrisponde all'ordinamento di una porzione fatta da un solo elemento (è già ordinata)

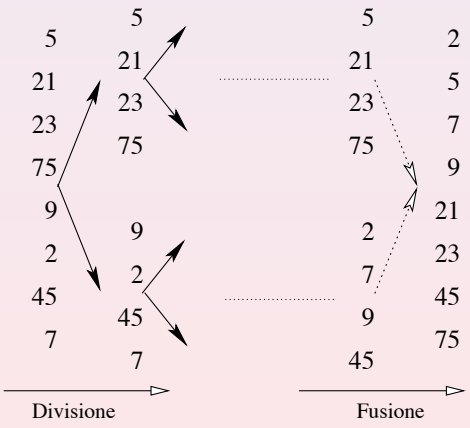
```
void SelectionSort(int v[], int from, int to){
  if (from < to) {
    int min = minPos(v,from,to);
    swap(v+from, v+min);
    SelectionSort(v, from+1, to);
  } }

void sort(int v[], int dim) {
    SelectionSort(v,0,dim-1);
}
```

Merge sort

Si divide il vettore da ordinare in due parti:

- si ordina ricorsivamente la prima parte
- si ordina ricorsivamente la seconda parte
- si combinano (operazione di fusione, merge) le due parti ordinate

Esempio:



Esprimiamo il procedimento in uno pseudo-linguaggio

```
ordina per fusione gli elementi di A da from a to

IF from < to (c'è più di un elemento tra from e to)

THEN
     mid =  (from + to) /2
     ordina per fusione gli elementi di A da from a mid
     ordina per fusione gli elementi di A da mid +1 a to
     fondi
        gli elementi di A da from a mid con
        gli elementi di A da mid+1 a to
        restituendo il risultato nel sottovettore
         di A da from a to
```

Implementiamo l'algoritmo in C, definendo una procedura ricorsiva

```
void mergeRicorsivo(int A[], int from, int to)
```

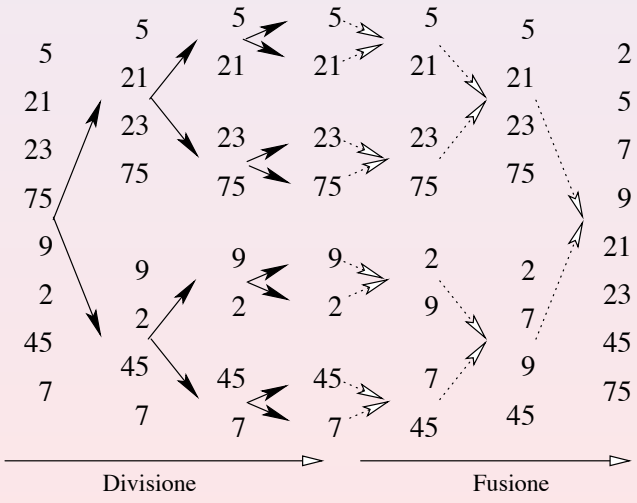
che ordina la porzione dell'array A individuata dagli indici from e to.

```
void mergeRicorsivo(int A[], int from, int to)

{
  int mid;

  if (from < to) {              /* l'intervallo da mid a to, estremi
                                   inclusi, comprende almeno due elementi */
    mid = (from + to) / 2;

    mergeRicorsivo(A, from, mid);
    mergeRicorsivo(A, mid+1, to);

    merge(A, from, mid, to);    /* fonde le due porzioni ordinate [from, mid],
                                   [mid+1, to] nel sottovettore [from, to]  */
  }
}
```


La procedura mergeSort che ordina un array di interi è semplicemente

```
void sort(int v[], int dim)
{
mergeRicorsivo(v, 0, dim-1);
}
```

Esempio:



- Vediamo l'operazione di *fusione*, definendo la procedura

  `void merge(int A[], int from, int mid, int to)`

  che fonde le due porzioni dell'array `A` con indici compresi tra `from` e `mid` e tra `mid+1` e `to`.
- La procedura utilizza un array di supporto `B`: per semplicità, supponiamo di avere una costante `LUNG` che definisce la lunghezza degli array che stiamo trattando.


```
void merge(int A[], int from, int mid, int to)
 {
  int B[LUNG];                              /* vettore di appoggio */
  int primo, secondo, appoggio, da_copiare;

  primo = from;
  secondo = mid + 1;
  appoggio = from;

  while (primo <= mid && secondo <= to) { /* copia in modo ordinato      */
    if (A[primo] <= A[secondo]) {         /* gli elementi della prima e  */
      B[appoggio] = A[primo];             /* della seconda porzione in B */
      primo++;                            /* fino ad esaurire una delle due */
    }
    else {
      B[appoggio] = A[secondo];
      secondo++;
    }
    appoggio++;
  }
```

```
 if (secondo > to)             /* e' finita prima la seconda porzione */
 /* copia da A in B tutti gli elementi della
    prima porzione fino a mid */

    for (da_copiare = primo; da_copiare <= mid; da_copiare++) {
       B[appoggio] = A[da_copiare];
       appoggio++;
    }
  else                          /* e' finita prima la prima porzione */

    for (da_copiare = secondo; da_copiare <= to; da_copiare++) {
    /* copia da A in B tutti gli elementi della
    /* seconda porzione fino a to */

       B[appoggio] = A[da_copiare];
       appoggio++;
    }

/* ricopia tutti gli elementi da from a to da B ad A */
  for (da_copiare = from; da_copiare <= to; da_copiare++)
    A[da_copiare] = B[da_copiare];
}
```