Tecniche della Programmazione, lez.12

Problema dell'ordinamento di una sequenza di elementi

… perché serve?

perché lavorare sulla sequenza ordinata può essere utile in parecchi casi

- ricerca di un elemento
- unicità di un elemento
- numero di occorrenze
- i due elementi più vicini
- elemento più frequente

What's "Sequenza Ordinata"?

Assumiamo che la sequenza sia di **N elementi**, ciascuno di tipo **TipoElemento**

$$a_1,\ a_2,\ \ldots\ a_n$$

... la possiamo definire "ordinata" se

- esiste una **relazione d'ordine** "$<$" (o "$\leq$") applicabile agli elementi

&&

- e per ogni coppia $a_i,\ a_{i+1}$ vale $a_i < a_{i+1}$ $\quad (a_i \leq a_{i+1})$

a

4	9	16	18	21	30

$a[0] \leq a[1] \leq a[2] \leq a[3] \leq a[4] \leq a[5]$

ovviamente viene naturale rappresentare la sequenza come un array ... ma non è l'unico modo ...

What's "Ordinamento"? (anche "Sorting")

Assumiamo che la sequenza sia di **N elementi**, ciascuno di tipo **TipoElemento**

$a_1, a_2, \ldots a_n$

... la possiamo definire "**ordinata**" se

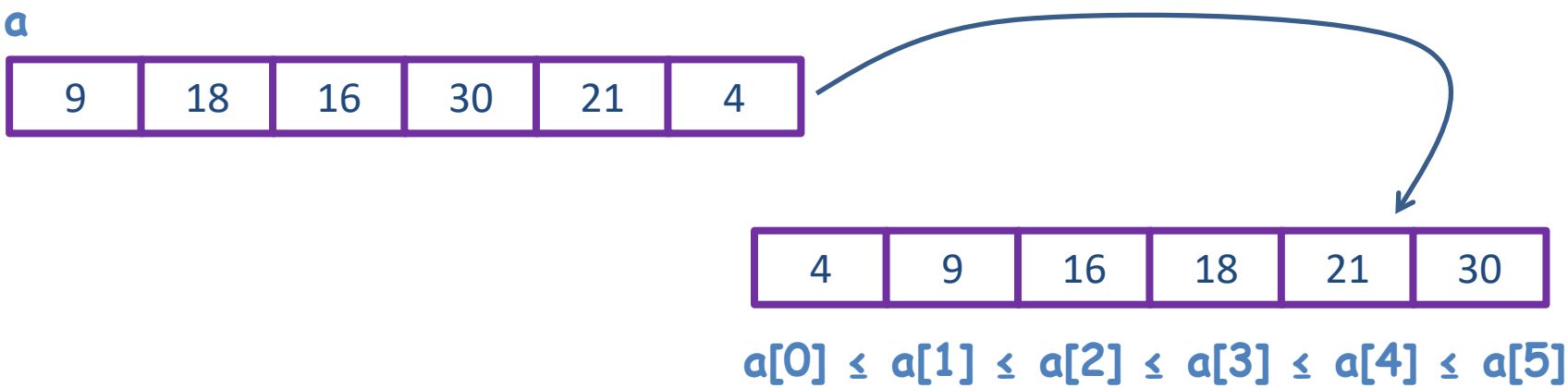
- esiste una **relazione d'ordine** "<" (o "≤") applicabile agli elementi
- e per ogni coppia a_i, a_{i+1} vale $a_i < a_{i+1}$ $\quad (a_i \leq a_{i+1})$

Problema di ORDINAMENTO

Data una sequenza $a_1, a_2, \ldots a_n$ modificarla, spostando gli elementi, in modo che sia ordinata.

Si può modificare la sequenza, oppure ottenere una nuova sequenza, $b_1, b_2, \ldots b_n$, ORDINATA, con i medesimi elementi)

Assumeremo che gli elementi siano INTERI e contenuti in un array, a , i cui elementi vengono, se serve, spostati, in modo che a diventi ordinato



Un algoritmo di ordinamento: sort per scambio, sort a bolle

A Sorting algorithm: Bubble Sort

si procede scorrendo l'array e confrontando gli elementi vicini (a[j],a[j+1]): se i due non sono in ordine tra loro, li si scambia ... E si scorre l'array più volte ...

ordinamento ... crescente

```
per ogni coppia a[j], a[j+1]
  se a[j]>a[j+1]
     scambia a[j] con a[j+1]
```

N=6 int a[N]

30 16 31 9 18 4

30 16 31 9 18 4 j,j+1=0,1

stato attuale *16 30 31 9 18 4*

codice per la parte di algoritmo qui sopra?

☺

16 30 31 9 18 4 j,j+1=1,2

stato attuale *16 30 31 9 18 4*

Un algoritmo di ordinamento: sort per scambio, sort a bolle

A Sorting algorithm: Bubble Sort

si procede scorrendo l'array e confrontando gli elementi vicini (a[j],a[j+1]): se i due non sono in ordine tra loro, li si scambia ... E si scorre l'array più volte ...

ordinamento ... crescente

```
per ogni coppia a[j], a[j+1]
  se a[j]>a[j+1]
     scambia a[j] con a[j+1]
```

```
for (j=0; j<N-1; j++){
  if (a[j]>a[j+1])
     scambia a[j] con a[j+1]
}
```

N=6

int a[N]

30 16 31 9 18 4 j,j+1=0,1

16 30 31 9 18 4 1,2

stato attuale *16 30 31 9 18 4*

16 30 31 9 18 4 2,3

stato attuale *16 30 9 31 18 4*

16 30 9 31 18 4 3,4

stato attuale *16 30 9 18 31 4*

Un algoritmo di ordinamento: sort per scambio, sort a bolle

A Sorting algorithm: Bubble Sort

si procede scorrendo l'array e confrontando gli elementi vicini (a[j],a[j+1]): se i due non sono in ordine tra loro, li si scambia ... E si scorre l'array più volte ...

ordinamento ... crescente

```
per ogni coppia a[j], a[j+1]
  se a[j]>a[j+1]
     scambia a[j] con a[j+1]
```

```
for (j=0; j<N-1; j++){
  if (a[j]>a[j+1])
     scambia a[j] con a[j+1]
}
```

"passata"

N=6

int a[N]

	j,j+1=0,1
30 16 31 9 18 4	
16 30 31 9 18 4	1,2
16 30 31 9 18 4	2,3
16 30 9 31 18 4	3,4
16 30 9 18 31 4	4,5
16 30 9 18 4 **31**	~~5,6~~

Prima passata! (n-1 controlli): l'array non e` ordinato, ma il più grande è "salito" al suo posto

A Sorting algorithm: Bubble Sort

Algoritmo = sequenza di passate

(dopo n-1 passate l'array è ordinato)

Seconda passata (n-2 controlli) il penultimo a posto

16 30 9 18 4 **31**	j=0
16 30 9 18 4 **31**	j=1
16 9 30 18 4 **31**	j=2
16 9 18 30 4 **31**	j=3
16 9 18 4 **30 31**	j=4
	j=5

"passata"

```
for (j=0; j<N-2; j++){
  if (a[j]>a[j+1])
    scambia a[j] con a[j+1]
}
```

A Sorting algorithm: Bubble Sort

TERZA passata (n-3 controlli) il terz'ultimo a posto

16 9 18 4 **30 31**	j=0
9 16 18 4 **30 31**	j=1
9 16 18 4 **30 31**	j=2
9 16 4 **18 30 31**	

"passata"

```
for (j=0; j<N-3; j++){
  if (a[j]>a[j+1])
    scambia a[j] con a[j+1])
}
```

QUARTA passata (n-4 controlli)

```
for (j=0; j<N-4; j++){...
```

dopo 9 4 **16 18 30 31**

QUINTA passata cioè (n-1)-esima passata (n-5 = n-n+1 = 1 controllo)

```
for (j=0; j<N-5; j++){...
```

dopo 4 **9 16 18 30 31**

(dopo n-1 passate l'array è ordinato) **4 9 16 18 30 31**

A Sorting algorithm: Bubble Sort

i-esima passata

```
for (j=0; j<N-i; j++)
  if (a[j]>a[j+1])
    scambia a[j] con a[j+1]
```

algoritmo

Con i che conta da 1 a N-1

esegui la i-esima passata

```
/* ordina l'array arr di N elementi */
void bubblesort (int a[N]) {
 int i, j, aux;
 for (i=1; i<=N-1; i++)
   for (j=0; j<N-i; j++)
     if (a[j]>a[j+1]) {
       aux = a[j];
       a[j] = a[j+1];
       a[j+1] = aux;
     }
return;
}
```

A Sorting algorithm: Bubble Sort

Applicazione sull'array **16 8 2 15 4 9**

16 8 15 2 4 9 j=0

8 16 15 2 4 9 j=1

...

8 15 2 4 9 **16** j=5

Prima passata
5 confronti e
5 scambi

---

8 2 4 9 **15 16**

Seconda passata n-2 confronti
e 3 scambi

---

2 4 8 9 **15 16**

Terza passata n-3 confronti
e 2 scambi

---

2 4 8 9 15 16

Quarta passata, n-4 conf. e 0 scambi

---

2 4 8 9 15 16

Quinta passata, n-5 conf. e 0 scambi

---

2 4 8 9 15 16 !!

A Sorting algorithm: Bubble Sort

Applicazione sull'array **16 18 20 25 34 39**

16 18 20 25 34 39

Prima passata	**n-1 confronti**	**e**	**0 scambi**
seconda	**n-2**	**e**	**0**
terza	**n-3**	**e**	**0**
quarta	**n-4**	**e**	**0**
quinta	**n-5**	**e**	**0**

L'array potrebbe essere già ordinato prima del termine delle n-1 passate: dipende dalla configurazione della sequenza da ordinare

L'algoritmo però esegue sempre N-1 passate

e l'i-esima passata esegue sempre N-i confronti (a[i] > a[i+1]) con eventuale scambio

... si può migliorare, cercando di evitare le passate inutili, il più possibile ...

A Sorting algorithm: Bubble Sort

miglioramento

Mentre non abbiamo finito {

esegui una passata;

~~**se l'array risulta ordinato abbiamo finito**~~

se durante la passata non abbiamo eseguito scambi, abbiamo finito

}

```
while (finito==0) {      /* mentre non ... finito */
  fattoscambio=0;
  for (j=0; ... )
    if (a[j]>a[j+1]) {
      scambia a[j] con a[j+1]
      fattoscambio = 1;
    }
  if (fattoscambio==0)
      finito=1;
```

MA

- init finito

- manca `i` ... sostituita da `finito`, ok, ma ci servirebbe un contatore per non fare più di n-1 passate ...

A Sorting algorithm: Bubble Sort

```
finito=0;
i=0;       /* fatte 0 passate */
while (finito==0) {
  i=i+1;
  fattoscambio=0;
  for (j=0; j<N-i; j++ )
    if (a[j]>a[j+1]) {
      scambia a[j] con a[j+1]
      fattoscambio = 1;
    }
  if ((fattoscambio==0)||(i==N-1))
     finito=1;
}
```

A Sorting algorithm: Bubble Sort

```
void bsort (double a[N]) {
 int i, j, finito, fattoscambio;
 finito=0;
 i=0;    /* fatte 0 passate */
 while (finito==0) {
   i=i+1;
   fattoscambio=0;
   for (j=0; j<N-i; j++ )
     if (a[j]>a[j+1]) {
       scambia a[j] con a[j+1]
       fattoscambio = 1;
     }
   if ((fattoscambio==0)||(i==N-1))
     finito=1;
 }
return;
}
```

A Sorting algorithm: Bubble Sort

analisi

Istruzione dominante

```
if (a[j]>a[j+1]) {
        scambia a[j] con a[j+1]
        fattoscambio = 1;
      }
```

16 18 20 25 34 39

Caso migliore: dopo la prima passata (n-1 confronti e 0 scambi) finiamo

30 16 31 9 18 4

Caso peggiore: numero di confronti massimo

39 34 25 20 18 16

Caso #!%@!!! : massimo numero di confronti, ognuno con scambio e assegn.

Il miglioramento permette di mettere a frutto eventuali configurazioni favorevoli (ad es. array parzialmente ordinato): nei casi sfavorevoli si comporta come la versione iniziale; nei casi favorevoli guadagna un po'.

A Sorting algorithm: Bubble Sort

analisi

prima passata	**n-1 confronti**
seconda passata	**n-2 confronti**
...	
(i-esima passata)	**n-i confronti**
...	
(n-2)-esima passata	**2 confronti**
(n-1)-esima passata	**1 confronto**

Caso peggiore: n-1 passate

$$\sum_{i=1}^{n-1}(n-i) = (n-1)n - \sum_{i=1}^{n-1} i = (n-1)n - \frac{(n-1)n}{2} = \frac{n(n-1)}{2}$$

costo proporzionale a n^2

(stessa complessità del bubblesort senza miglioramento)

Caso migliore: 1 passata **unica passata: n-1 confronti**

Si presenta solo per il bsort (miglioramento)

costo proporzionale a n

Usiamo n invece di N nelle formule

Un altro algoritmo di ordinamento

Another Sorting algorithm: *Selection Sort*

si seleziona il minimo della porzione non ancora sistemata e lo si mette al primo posto di tale porzione … **… porzione??**

0) l'elemento più piccolo dell'array viene messo in a[0]
(NB è il minimo in a[0]-a[N-1])

(ora "a[0] è a posto!")

1) l'elemento più piccolo della porzione a[1]-a[N-1] viene messo in a[1]

(ora a[0] e a[1] sono "a posto!")

2) l'elemento più piccolo della porzione a[2]-a[N-1] viene messo in a[2]

…

(ora a[0] a[1] e a[2] sono "a posto!")

N-2) l'elemento minimo della porzione a[N-2]-a[N-1] viene messo in a[N-2]

(remember, a[N-1] è l'ultimo elemento di a)

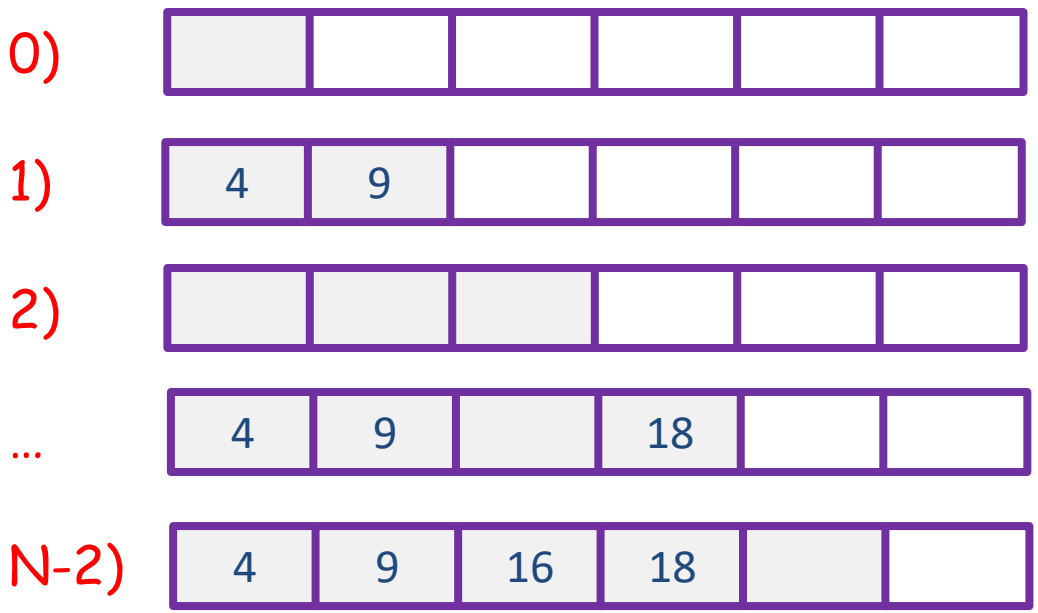
--- stop, dopo N-1 passi sono tutti a posto

Another Sorting algorithm: Selection Sort

applicazione intuitiva: ☺ disegnare i 5 stati successivi dell'array

`int a[6]` N=6

9	18	16	4	30	21



0) l'elemento più piccolo dell'array viene messo in a[0]

1) l'elemento più piccolo della porzione a[1]-a[N-1] viene messo in a[1]

2) l'elemento più piccolo della porzione a[2]-a[N-1] viene messo in a[2]

...

N-2) l'elemento più piccolo della porzione a[N-2]-a[N-1] viene messo in a[N-2]

--- stop,

NB
quando mettiamo il minimo della porzione a[i]...a[N-1] in a[i], l'elemento che era in a[i] non deve andare perso ... quindi lo mettiamo dove stava il minimo ... (scambio)

Another Sorting algorithm: Selection Sort

applicazione intuitiva
disegnare i 5 stati successivi dell'array

`int a[6]` N=6

9	18	16	4	30	21

0)	4	18	16	9	30	21	scambio tra 4 e 9 (a[0] e a[3])
1)	4	9	16	18	30	21	scambio tra a[1] e a[3]
2)	4	9	16	18	30	21	16 rimane a[2]
3)	4	9	16	18	30	21	18 rimane in a[3]
N-2)	4	9	16	18	21	30	scambio tra a[N-2] e a[N-1]

ora anche a[N-1] è a posto

NB
quando mettiamo il minimo della porzione a[i]...a[N-1] in a[i], l'elemento che era in a[i] non deve andare perso ... quindi scambiamo le posizioni del minimo e di a[i] ...

Another Sorting algorithm: Selection Sort

applicazione intuitiva
disegnare i 5 stati successivi dell'array

`int a[6]` N=6

9	18	16	4	30	21

0)	4	18	16	9	30	21	scambio tra 4 e 9 (a[0] e a[3])
1)	4	9	16	18	30	21	scambio tra a[1] e a[3]
2)	4	9	16	18	30	21	16 rimane a[2]
3)	4	9	16	18	30	21	18 rimane in a[3]
N-2)	4	9	16	18	21	30	scambio tra a[N-2] e a[N-1] ora anche a[N-1] è a posto

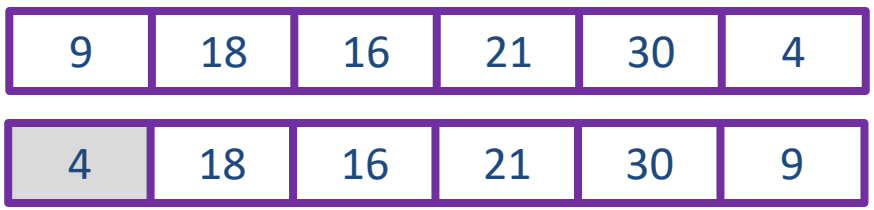
chiamare "0" il primo passo è una buona tattica quando si lavora con gli array.
Notare la corrispondenza tra ... la numerazione di un passo
... e l'indice dell'elemento sistemato dal passo ...

Selection Sort: algoritmo

0) lavoriamo su int a[N], ci serve 'min', … i …
1) per i da 0 a N-2
   1.1) trovare 'min' tale che a[min] è il minimo elemento in a[i]-a[N-1]
   1.2) scambiare a[i] con a[min]
2) fine

`int a[6]` N=6

i=0

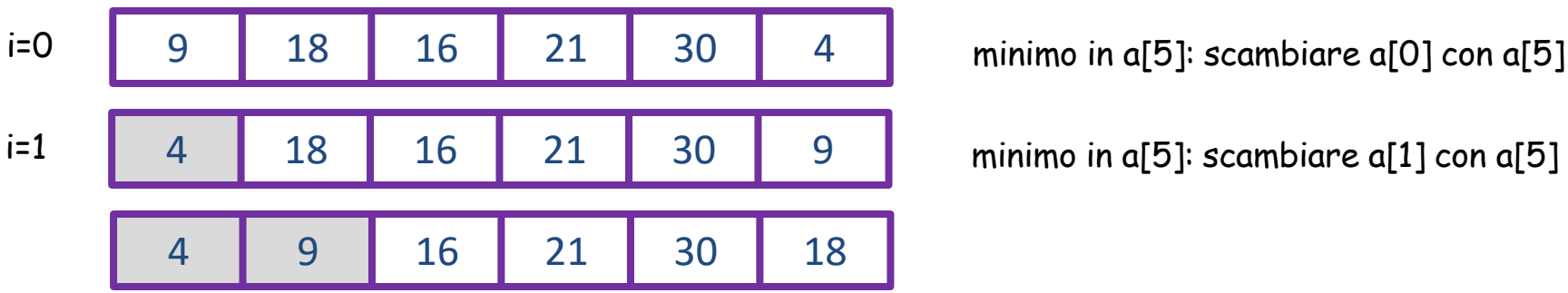


minimo in a[5]: scambiare a[0] con a[5]

Selection Sort: algoritmo

0) lavoriamo su int a[N], ci serve 'min', … i …
1) per i da 0 a N-2
   1.1) trovare 'min' tale che a[min] è il minimo elemento in a[i]-a[N-1]
   1.2) scambiare a[i] con a[min]
2) fine

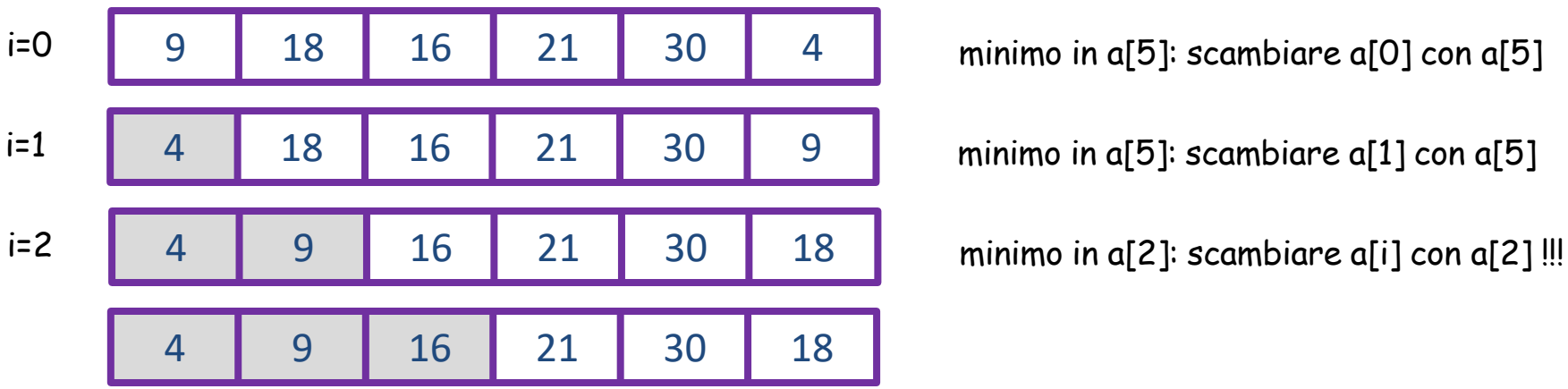
`int a[6]` n=6



Selection Sort: algoritmo

0) lavoriamo su int a[N], ci serve 'min', … i …
1) per i da 0 a N-2
   1.1) trovare 'min' tale che a[min] è il minimo elemento in a[i]-a[N-1]
   1.2) scambiare a[i] con a[min]
2) fine

`int a[6]` n=6



minimo in a[5]: scambiare a[0] con a[5]

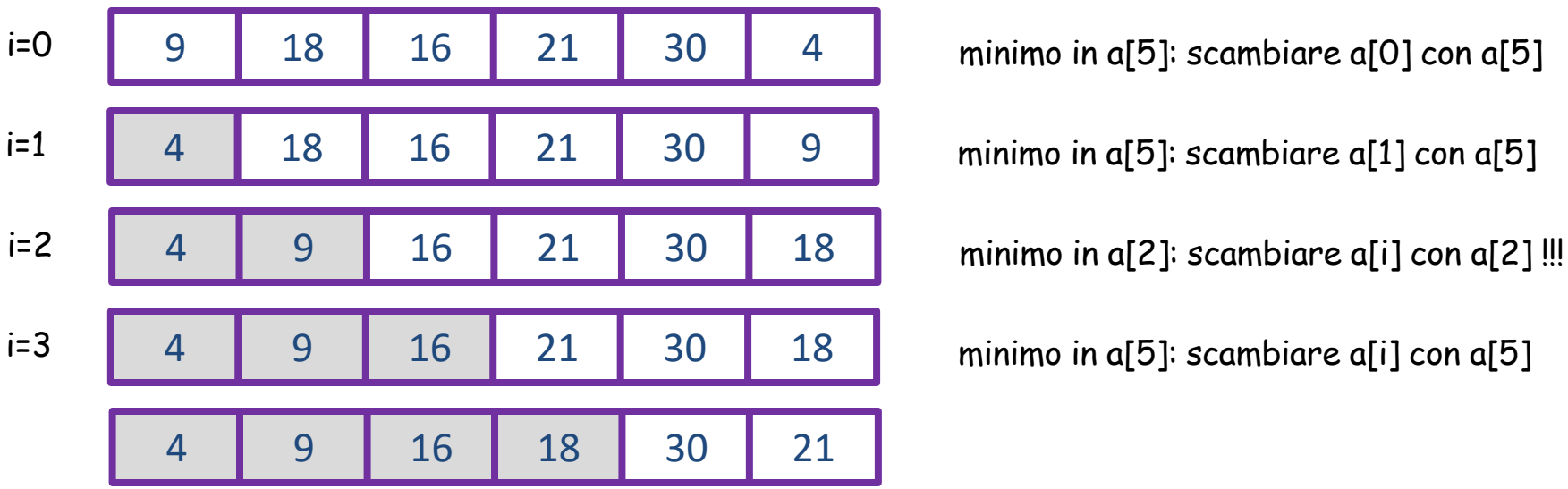
minimo in a[5]: scambiare a[1] con a[5]

minimo in a[2]: scambiare a[i] con a[2] !!!

Selection Sort: algoritmo

0) lavoriamo su int a[N], ci serve 'min', … i …
1) per i da 0 a N-2
   1.1) trovare 'min' tale che a[min] è il minimo elemento in a[i]-a[N-1]
   1.2) scambiare a[i] con a[min]
2) fine

`int a[6]` n=6



Selection Sort: algoritmo

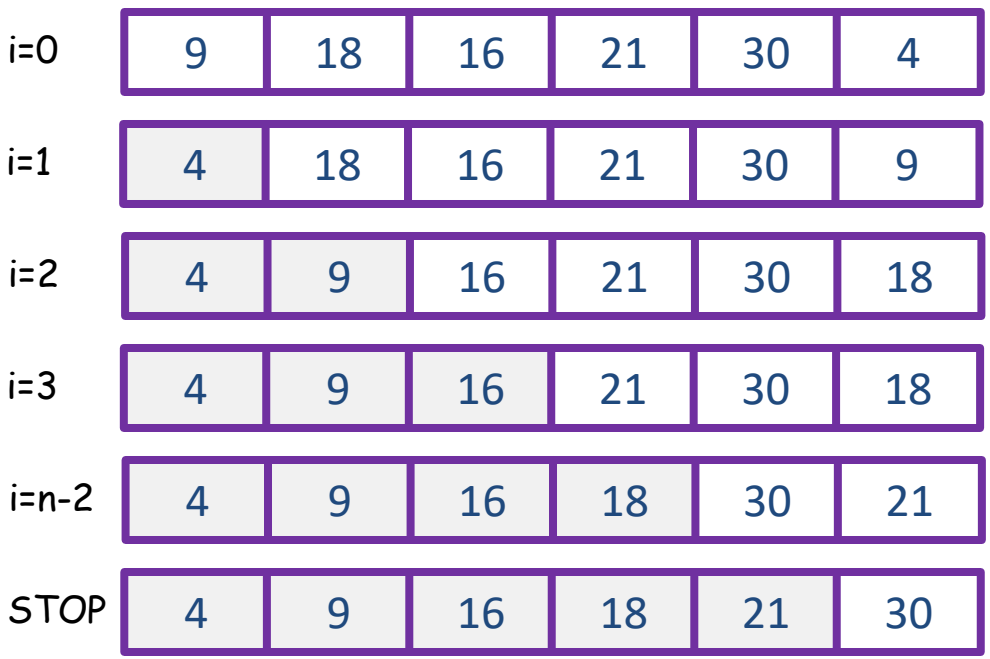
0) lavoriamo su int a[N], ci serve 'min', ... i ...
1) per i da 0 a N-2
   1.1) trovare 'min' tale che a[min] è il minimo elemento in a[i]-a[N-1]
   1.2) scambiare a[i] con a[min]
2) fine

`int a[6]` n=6

Selection Sort: algoritmo

0) lavoriamo su int a[N], ci serve 'min', … i …
1) per i da 0 a N-2
   1.1) trovare 'min' tale che a[min] è il minimo elemento in a[i]-a[N-1]
   1.2) scambiare a[i] con a[min]
2) fine

`int a[6]` n=6



minimo in a[5]: scambiare a[0] con a[5]

minimo in a[5]: scambiare a[1] con a[5]

minimo in a[2]: scambiare a[i] con a[2] !!!

minimo in a[5]: scambiare a[i] con a[5]

minimo in a[5]: scambiare a[4] con a[5]

a[5] può solo essere al posto giusto - STOP

Selection Sort: funzione che implementa l'algoritmo

funzione selectionSort(?)

0) void; riceve int a[N]; usa i, min … j

1) per i da 0 a N-2
   1.1) trovare 'min' tale che a[min] è il minimo elemento in a[i]-a[N-1]

      1.1.1) ricerca del minimo in a[i]-a[N-1]
         - min inizializzato con i
         - ciclo con tecnica del "minimo parziale", scorrendo gli elementi a[i+1]-a[N-1]
           per j da i+1 a N-1 se a[j] < a[min] min cambia

   1.2) scambiare a[i] con a[min]

2) fine

funzione selectionSort()

```
void selectionSort (int arr[N]) {
  int i, j, min;

 for (i=0; i<N-1; i++) {          /* min è l'indice del minimo parz */
    min=i;                        /* min inizializzato con i */
    for (j=i+1; j<=N-1; j++)
       if (arr[j] < arr[min])
          min=j;

    …                  /*scambio tra arr[i] e arr[min] */   … come si fa? ☺
 }
return;
}
```

funzione selectionSort()

```
void selectionSort (int arr[N]) {
  int i, j, min, aux;              /* aux usato per lo scambio */

 for (i=0; i<N-1; i++) {
                                   /* min sarà l'indice del minimo */
    min=i;                         /* init minimo parziale */

    for (j=i+1; j<=N-1; j++)       /* calcolo del minimo */
       if (arr[j] < arr[min])
          min=j;

                                   /*scambio tra arr[i] e arr[min] */
    aux = arr[i];
    arr[i] = arr[min];
    arr[min] = aux;
 }
return;
}
```

funzione selectionSort() --- analisi

```
void selectionSort (int arr[N]) {
  int i, j, min, aux;
 for (i=0; i<N-1; i++) {
    min=i;
    for (j=i+1; j<=N-1; j++)
       if (arr[j] < arr[min])
          min=j;

    ... /* scambio tra arr[i] e arr[min] */
 }
return;
}
```

Quanto ci vuole ad eseguire il sort?

quante volte viene eseguita min=i ?

quante volte viene eseguito il confronto? Più di min=i? ☺

quante volte viene eseguita min=j ?

funzione selectionSort() --- analisi

```
void selectionSort (int arr[N]) {
  int i, j, min, aux;
 for (i=0; i<N-1; i++) {
    min=i;
    for (j=i+1; j<=N-1; j++)
       if (arr[j] < arr[min])
          min=j;

    ... /* scambio tra arr[i] e arr[min] */
 }
return;
}
```

Quanto ci vuole ad eseguire il sort?

quante volte viene eseguita min=i ? N-1

quante volte viene eseguito il confronto ? più di tutti

quante volte viene eseguita min=j ? mah! … tra 0 e #confronti

funzione selectionSort() --- analisi

```
void selectionSort (int arr[N]) {
  int i, j, min, aux;
 for (i=0; i<N-1; i++) {
    min=i;
    for (j=i+1; j<=N-1; j++)
       if (arr[j] < arr[min])
          min=j;

    ... /*
 }
return;
}
```

Usiamo n invece di N nelle formule

il confronto viene eseguito
- **n-1 volte alla prima passata**
- **n-2 …**
- **… n-k volte alla k-esima**
- **1 volta alla n-1-esima passata**

cioè SEMPRE un numero di volte proporzionale a n^2

$$\sum_{i=1}^{n-1}(n-i) = (n-1)n - \sum_{i=1}^{n-1} i = (n-1)n - \frac{(n-1)n}{2} = \frac{n(n-1)}{2}$$

Dal punto di vista del numero di confronti non c'è un caso peggiore.
Se proprio vogliamo identificare un caso #!%@!!!, questo potrebbe essere quando l'array è così messo male che min=j viene eseguito sempre, per ogni j … costa un po' di più, ma l'ordine di grandezza del costo è sempre n^2.

funzione selectionSort() --- riassumendo l'analisi

```
void selectionSort (int arr[N]) {
  int i, j, min, aux;
 for (i=0; i<N-1; i++) {
    min=i;
    for (j=i+1; j<=N-1; j++)
       if (arr[j] < arr[min])
          min=j;

    ... /* scambio tra arr[i] e arr[min] */
 }
return;
}
```

il confronto viene eseguito un numero di volte "di un ordine di grandezza maggiore di qualsiasi altra" (eccetto min=j in un caso)

Quindi basiamo su questa operazione, nel caso peggiore, l'analisi dell'algoritmo (della sua efficienza)

Il costo di ordinare un array di n elementi con il selection sort è proporzionale ad n^2

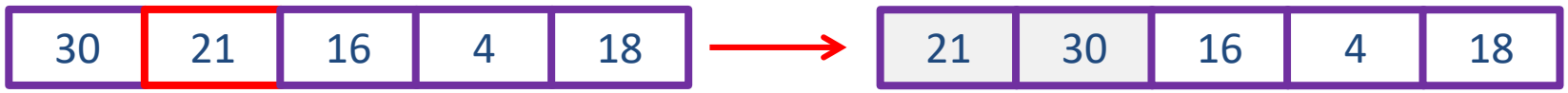
$$\alpha n^2$$

(maggiori informazionisul calcolo della complessità di un algoritmo e sull'ordine di grandezza del costo, non riusciamo a darle qui ... saranno oggetto di due brevi discussioni in altre lezioni; ma soprattutto saranno oggetto di studio approfondito in un altro corso ...)

A third *Sorting algorithm:* *Insertion Sort*

(*ordine crescente*)	**viene considerato ogni elemento dal secondo in poi; ogni volta si piazza l'elemento considerato nella posizione relativa giusta rispetto ai suoi predecessori.**

1) si mette l'elemento a[1] nella giusta relazione con a[0]



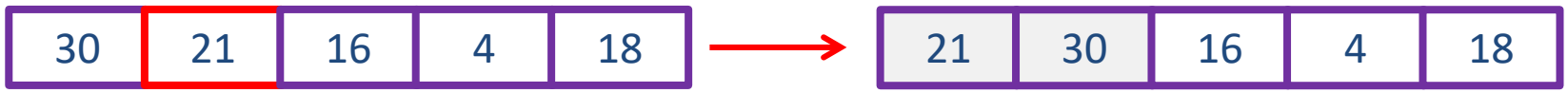
ora a[0]-a[1] sono ok (tra loro … ma non è detto che rimangano dove sono)

2)

Insertion Sort

(*ordine crescente*)	**viene considerato ogni elemento dal secondo in poi; ogni volta si piazza l'elemento in considerazione nella giusta posizione rispetto ai suoi predecessori.**

1) si mette l'elemento a[1] nella giusta relazione con a[0]



ora a[0]-a[1] sono ok (tra loro … ma non è detto che rimangano dove sono)

2) si mette l'elemento a[2] nella giusta relazione con a[0]-a[1]

abbiamo inserito 16 e shiftato i primi due: ora a[0]-a[2] sono ok (tra loro …)

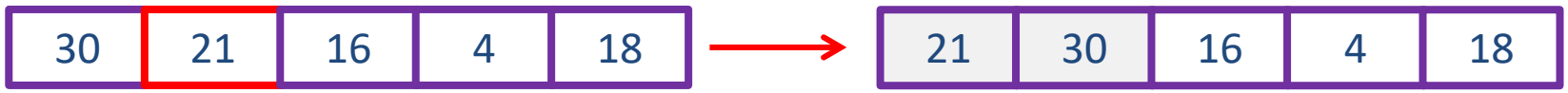
3) si mette l'elemento a[3] nella giusta relazione con a[0]-a[2]

NB ogni volta si esegue una "passata" su una porzione dell'array, per identificare la posizione giusta dell'elemento in considerazione, rispetto a quelli di indice inferiore

Insertion Sort

(*ordine crescente*)	**viene considerato ogni elemento dal secondo in poi; ogni volta si piazza l'elemento in considerazione nella giusta posizione rispetto ai suoi predecessori.**

passata 1) si mette l'elemento a[1] nella giusta relazione con a[0]



ora a[0]-a[1] sono ok (tra loro … ma non è detto che rimangano dove sono)

passata 2) si mette l'elemento a[2] nella giusta relazione con a[0]-a[1]

abbiamo inserito 16 e shiftato i primi due: ora a[0]-a[2] sono ok (tra loro …)

passata 3) si mette l'elemento a[3] nella giusta relazione con a[0]-a[2]

abbiamo inserito 4 e shiftato i primi tre: ora a[0]-a[3] sono ok

? ☺

Insertion Sort

viene considerato ogni elemento dal secondo in poi (per ordine crescente); ogni volta si piazza l'elemento in considerazione nella giusta posizione rispetto ai suoi predecessori.

passata 1) si mette l'elemento a[1] nella giusta relazione con a[0]
ora a[0]-a[1] sono ok

passata 2) si mette l'elemento a[2] nella giusta relazione con a[0]-a[1]
abbiamo inserito 16 e shiftato i primi due: ora a[0]-a[2] sono ok

passata 3) si mette l'elemento a[3] nella giusta relazione con a[0]-a[2]
abbiamo inserito 4 e shiftato i primi tre: ora a[0]-a[3] sono ok

passata 4) si mette l'elemento a[4] nella giusta relazione con a[0]-a[3]

abbiamo inserito 18 e shiftato due suoi ex predecessori:
ora a[0]-a[4] sono ok, cioè l'array è ordinato.

Insertion Sort: esercizio: ordinare usando insertion sort

61	42	86	18	41	53	47

passata 1) →

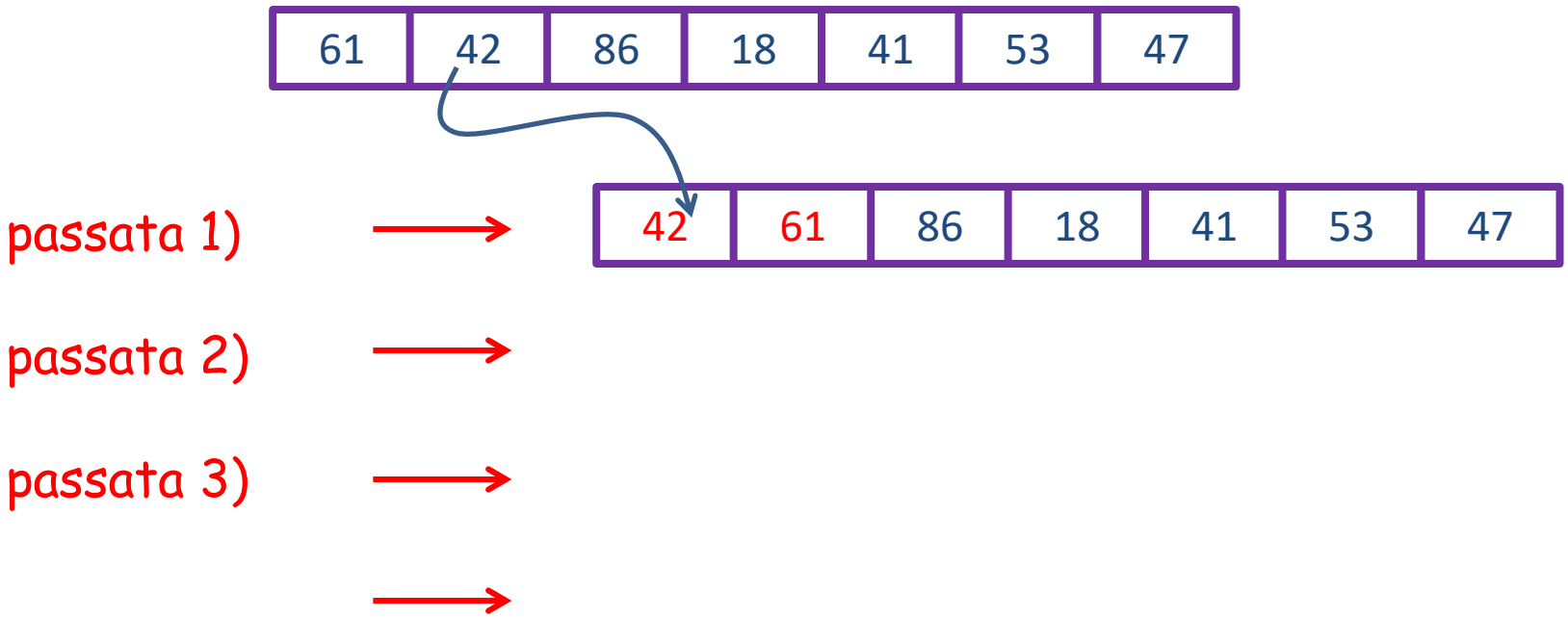
passata 2) →

passata 3) →

→

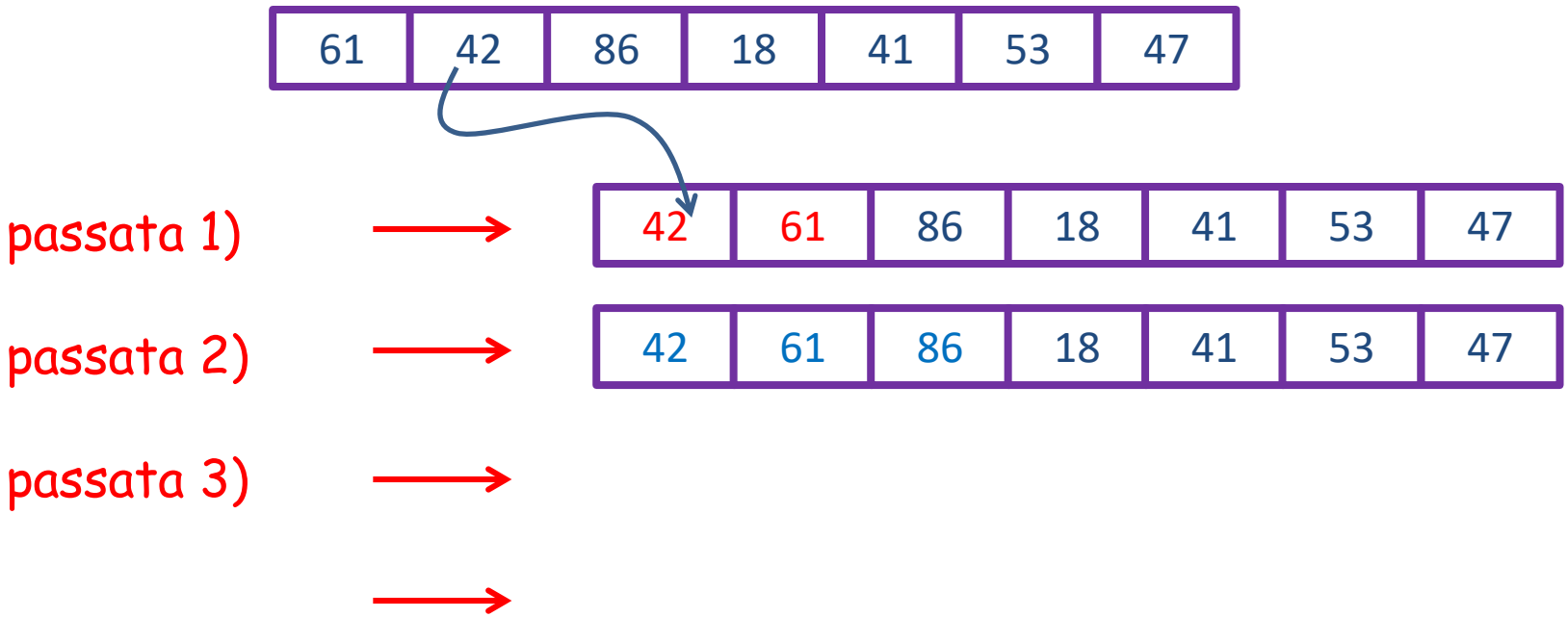
→

Insertion Sort: esercizio: ordinare usando insertion sort



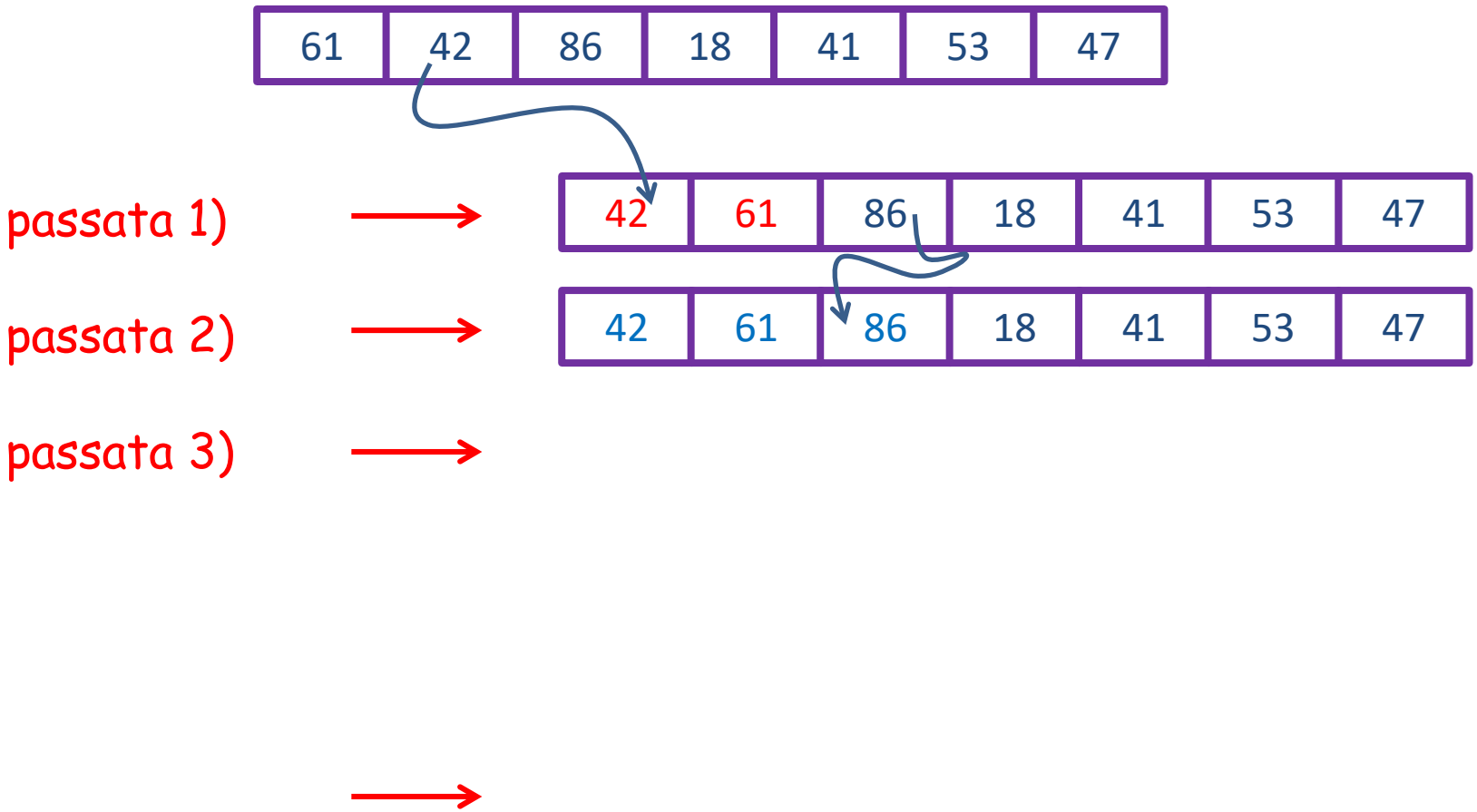
NB In rosso l'elemento messo a posto e tutti quelli che si deono essere spostati per farlo inserire al suo posto

Insertion Sort: esercizio: ordinare usando insertion sort

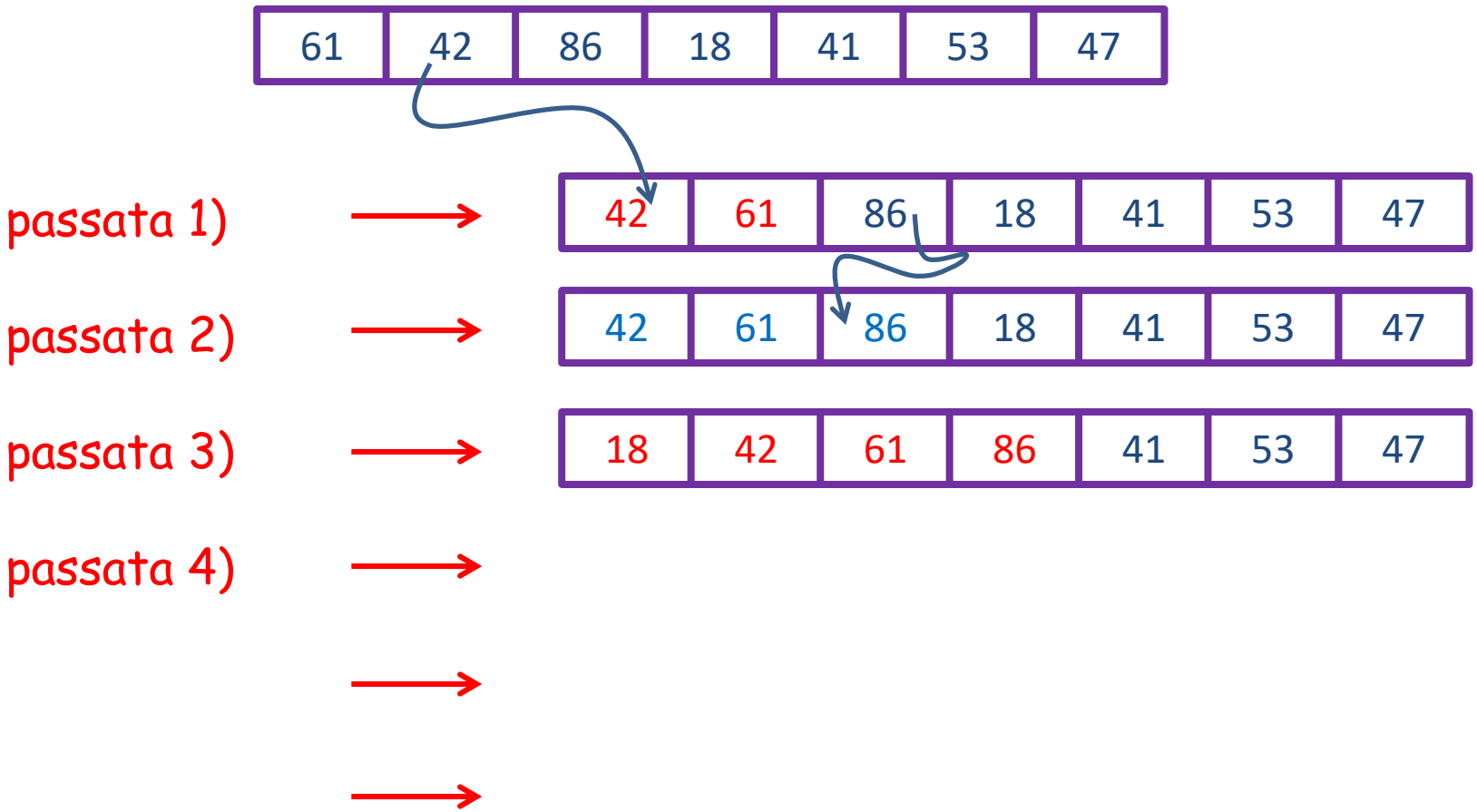


NB In rosso l'elemento messo a posto e tutti quelli che si deono essere spostati per farlo inserire al suo posto

Insertion Sort: esercizio: ordinare usando insertion sort

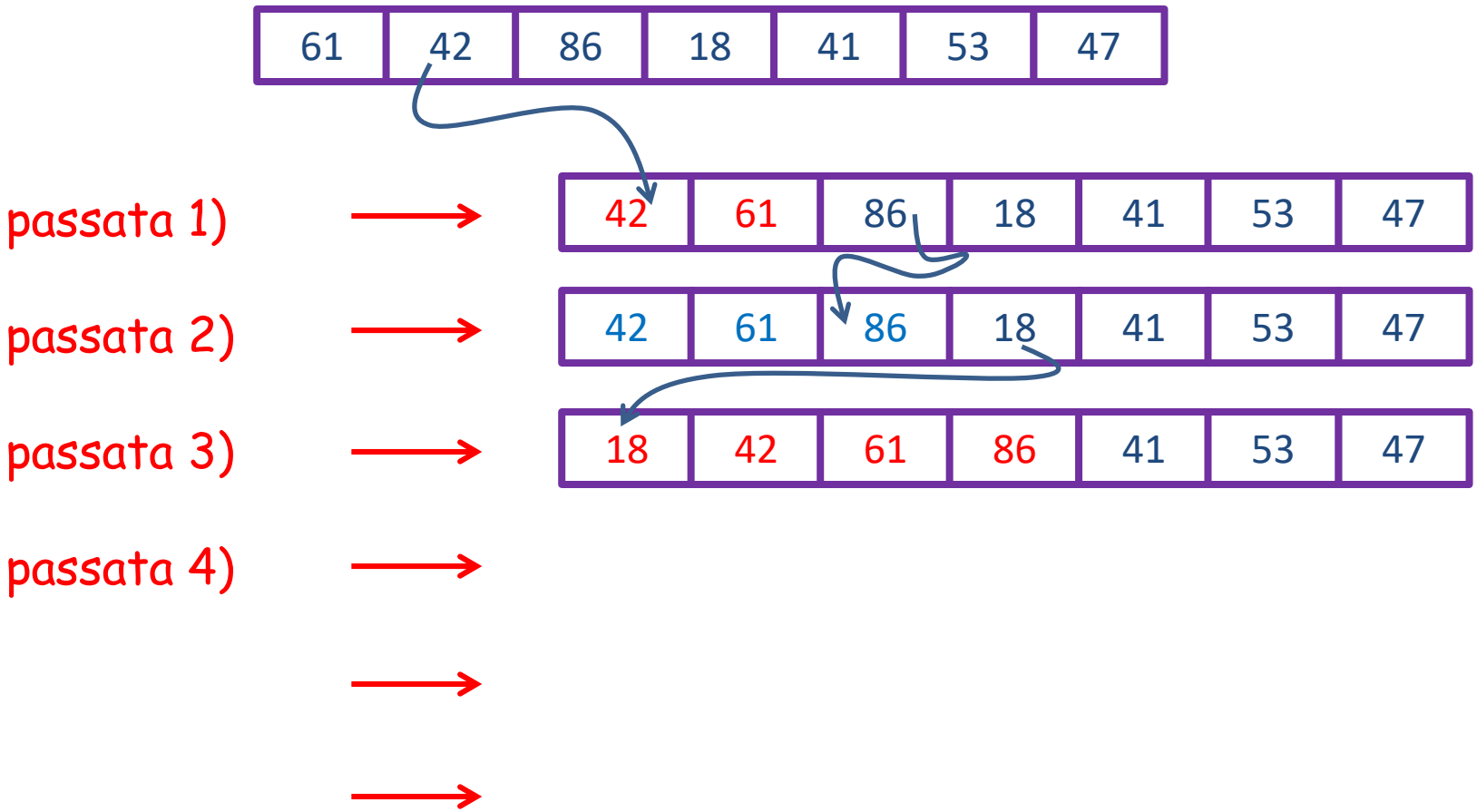


Insertion Sort: esercizio: ordinare usando insertion sort

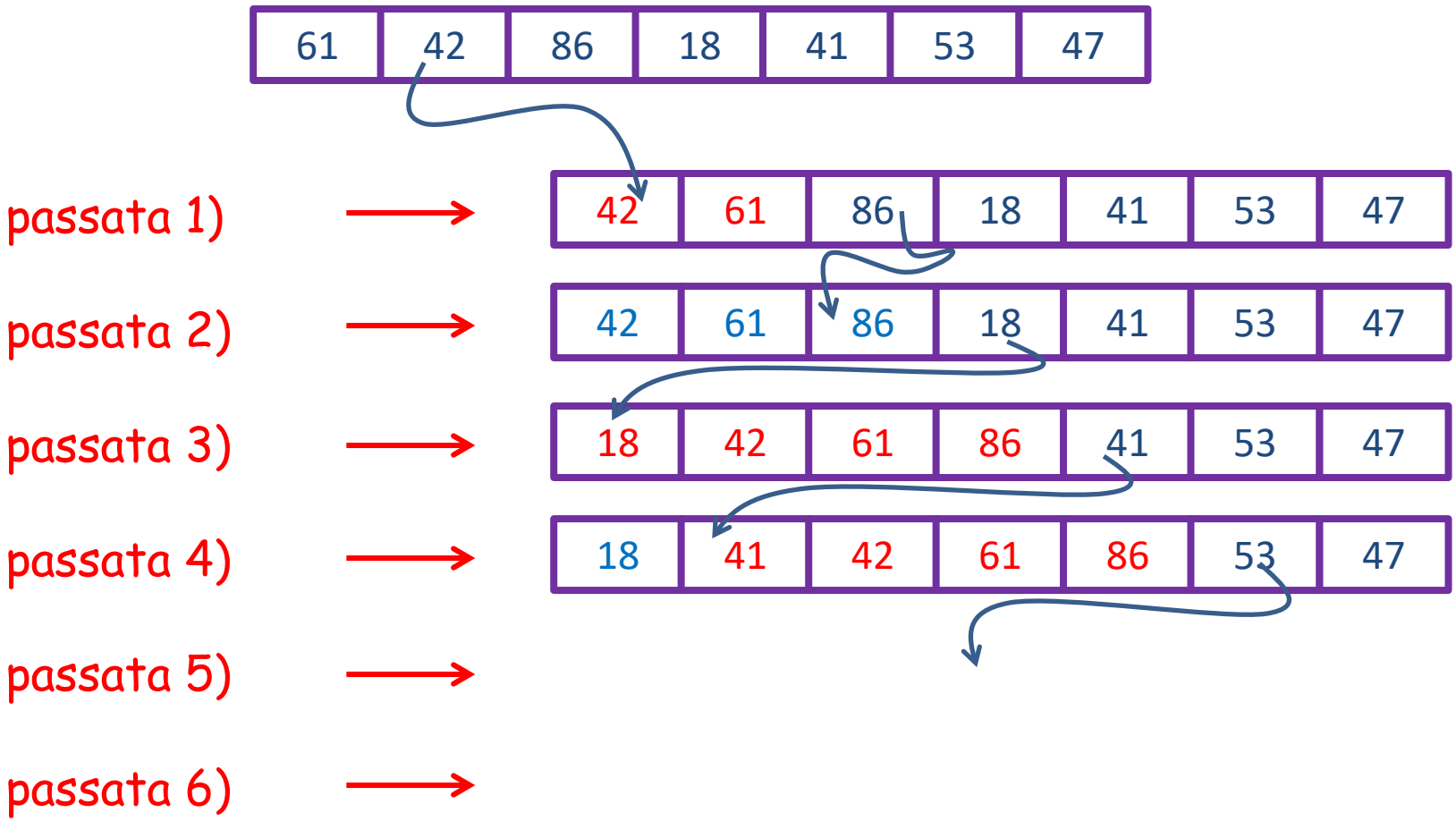


NB In rosso l'elemento messo a posto e tutti quelli che si deono essere spostati per farlo inserire al suo posto

Insertion Sort: esercizio: ordinare usando insertion sort

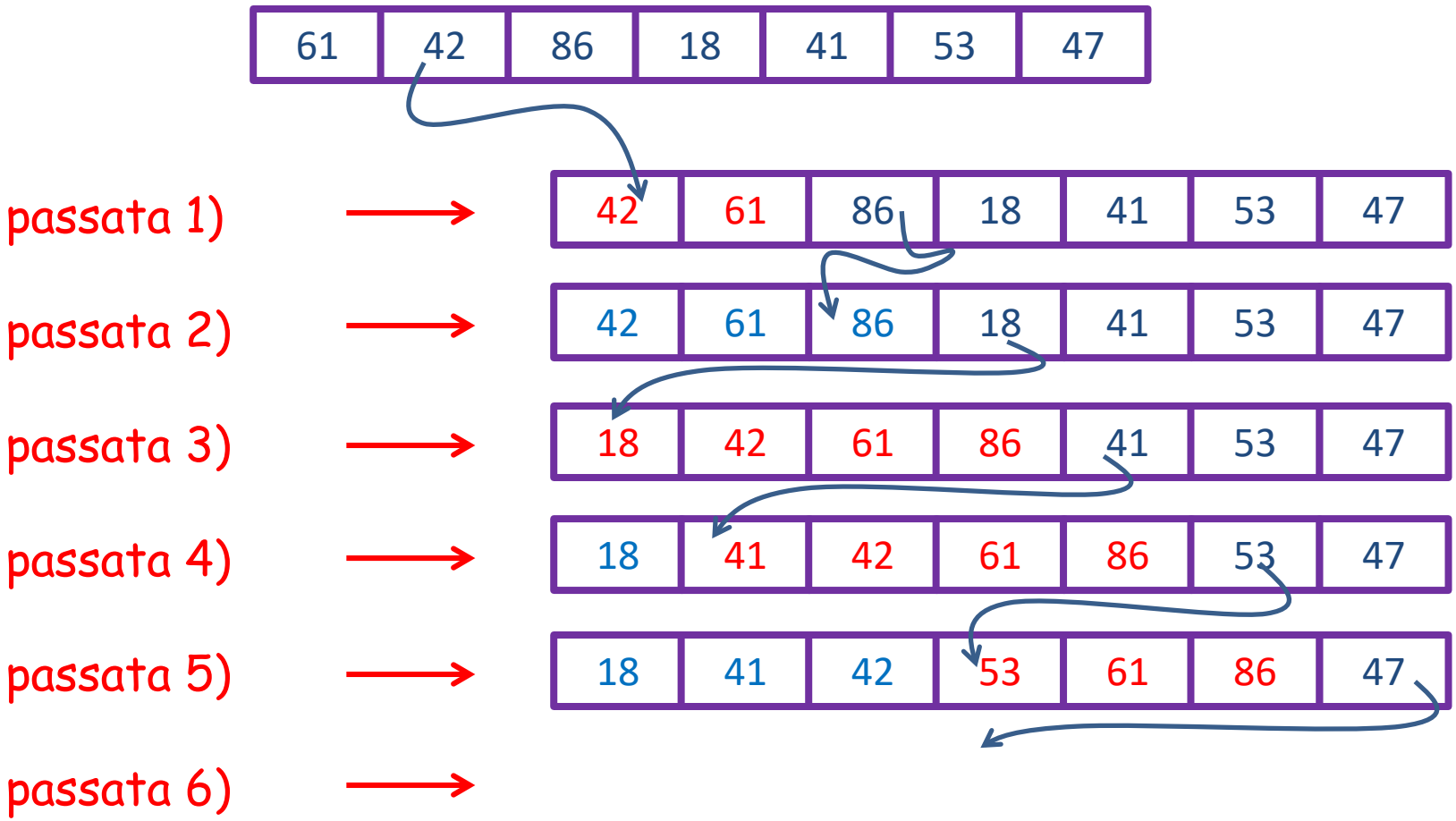


Insertion Sort: esercizio: ordinare usando insertion sort



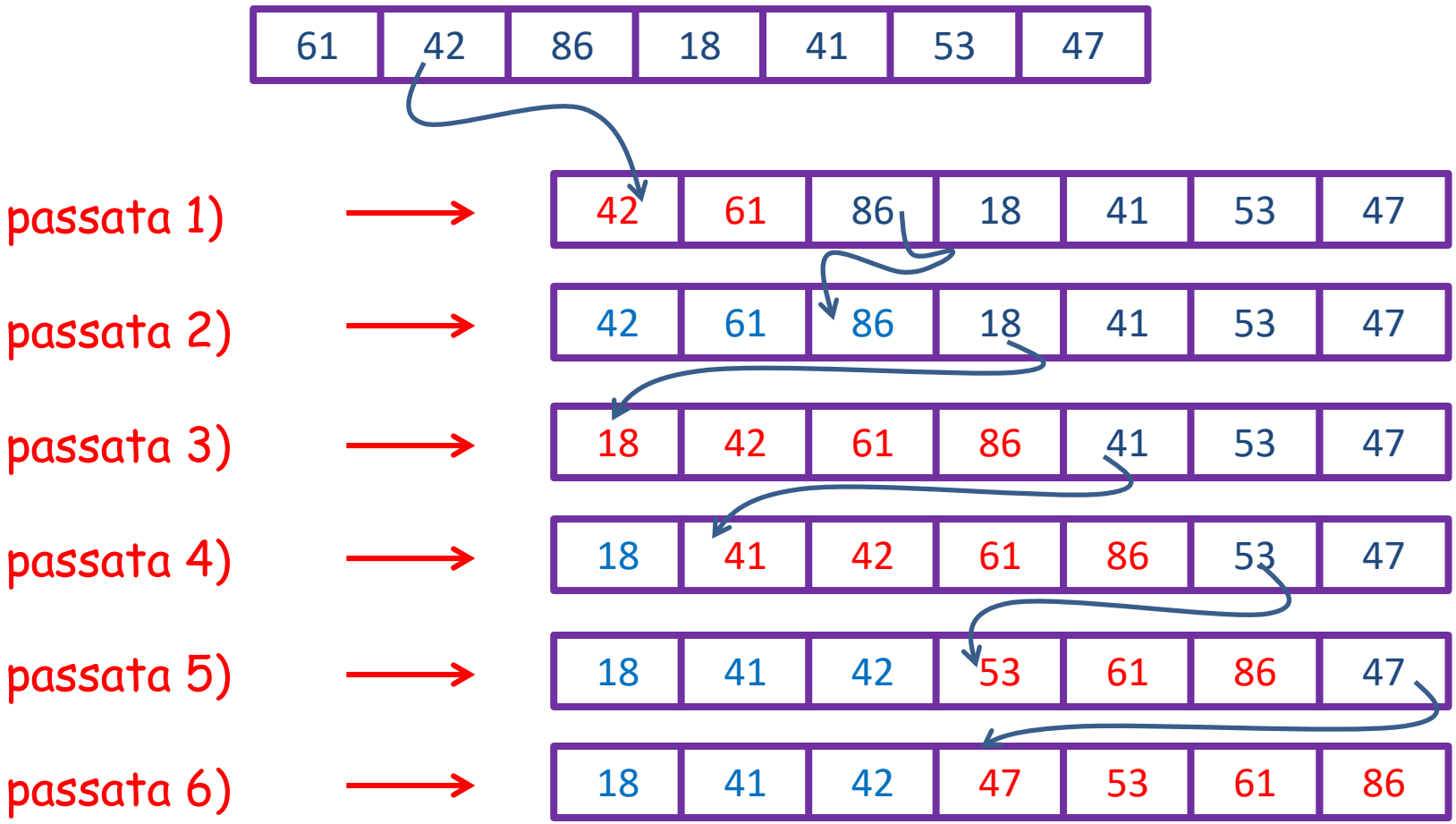
NB In rosso l'elemento messo a posto e tutti quelli che si deono essere spostati per farlo inserire al suo posto

Insertion Sort: esercizio: ordinare usando insertion sort



NB In rosso l'elemento messo a posto e tutti quelli che si deono essere spostati per farlo inserire al suo posto

Insertion Sort: esercizio: ordinare usando insertion sort



NB In rosso l'elemento messo a posto e tutti quelli che si devono essere spostati per farlo inserire al suo posto

Insertion Sort: secondo esercizio: ordinare usando insertion sort

passata 1) ⟶

passata 2) ⟶

passata 3) ⟶

passata 4) ⟶

passata 5) ⟶

passata 6) ⟶

Insertion Sort: secondo esercizio: ordinare usando insertion sort

10	30	15	4	3	2

passata 1) →	10	30	15	4	3	2
passata 2) →	10	15	30	4	3	2
passata 3) →	4	10	15	30	3	2
passata 4) →	3	4	10	15	30	2
passata 5) →	2	3	4	10	15	30

passata 6) → no (l'algoritmo è previsto faccia N-1 passate)

colori? ☺

Insertion Sort: secondo esercizio: ordinare usando insertion sort

qui i colori sono usati come nell'esercizio precedente, per mettere in evidenza ... cosa? ☺

passata 1) → 10 30 15 4 3 2

passata 2) → 10 15 30 4 3 2

passata 3) → 4 10 15 30 3 2

passata 4) → 3 4 10 15 30 2

passata 5) → 2 3 4 10 15 30

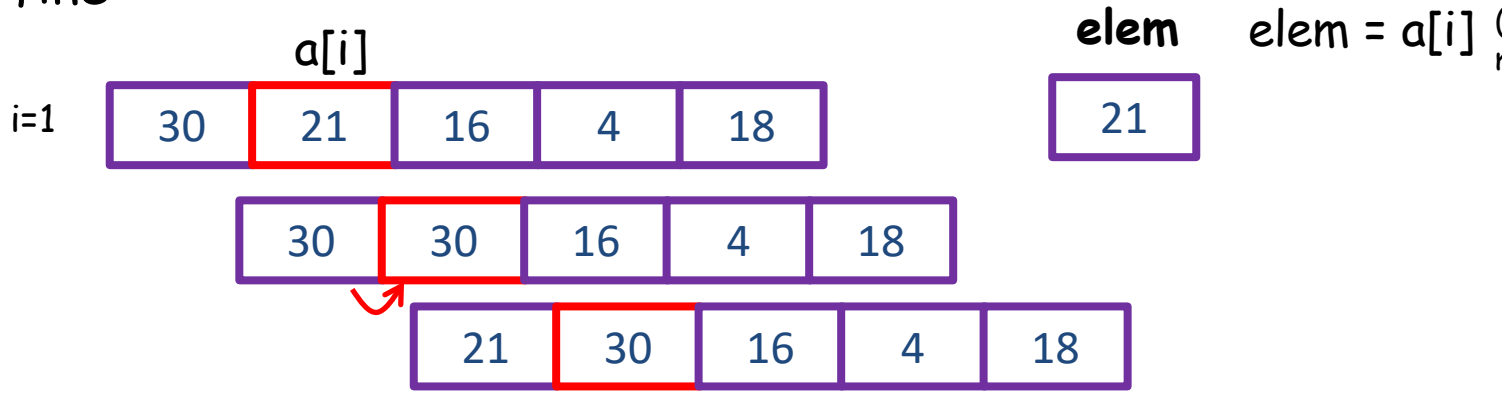
Insertion Sort: implementazione in una funzione

funzione insertionSort()

0) void; riceve int a[]; usa i, …

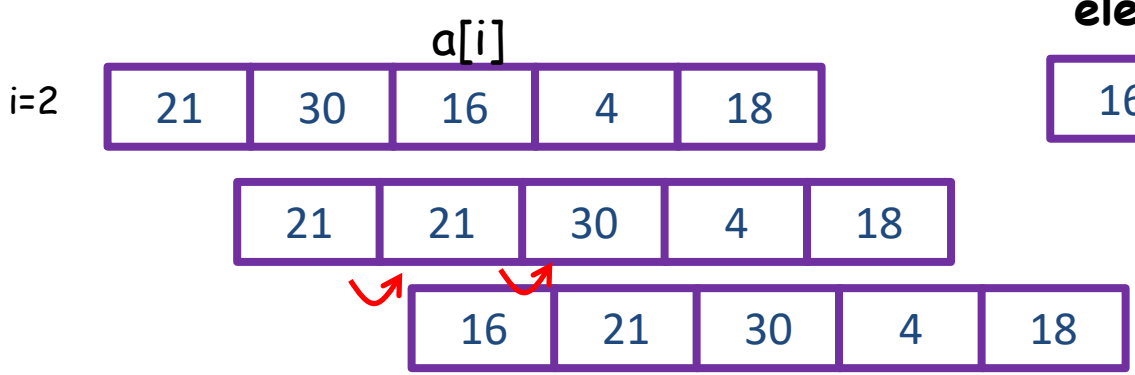
1) **per i da 1 a N-1**, mettere a posto a[i] assumendo che a[0]-a[i-1] siano a posto (cioè fare una passata)

2) fine



elem elem = a[i] (l'elemento da mettere a posto)

21



elem elem = a[i]

16

scorrendo all'indietro la porzione a[i]-a[N-1] si trova l'indice "primo"

Poi elem va in a[primo] doo che gli a[primo]-a[i-1] sono stati shiftati

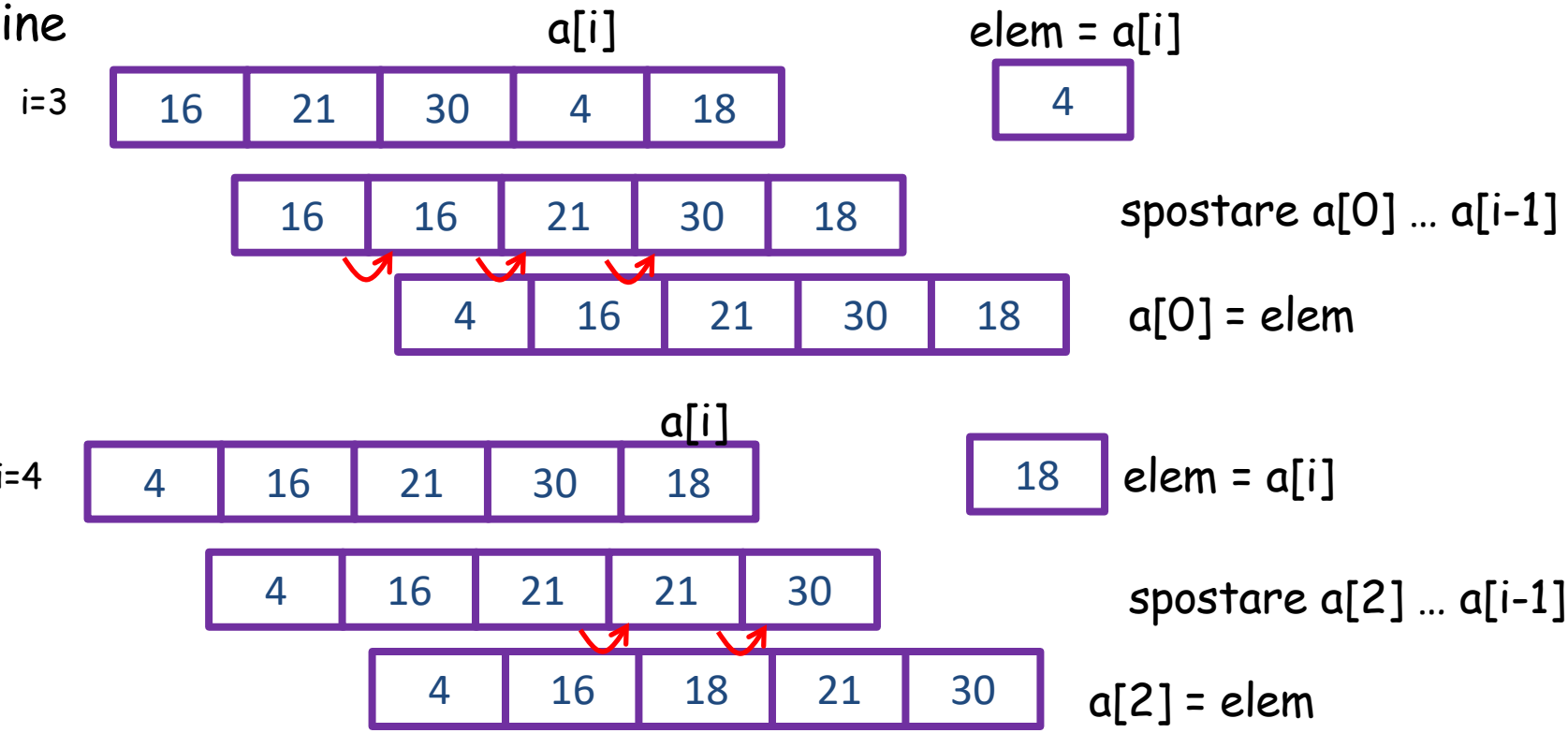
Insertion Sort: implementazione in una funzione

funzione insertionSort()

0) void; riceve int a[]; usa i, …

1) **per i da 1 a n-1**, mettere a posto a[i] assumendo che a[0]-a[i-1] siano a posto (cioè fare una passata)

2) fine



Insertion Sort: implementazione in una funzione

funzione insertionSort()

0) void; riceve int a[]; usa i, … elem, primo, k

1) per i da 1 a n-1, mettere a posto a[i] assumendo che a[0]-a[i-1] siano a posto (cioè fare una passata)

   1.1) elem = a[i]

   1.2) trovare 'primo' = indice del **primo** elemento in a[0]-a[i-1] che è maggiore di a[i]

   1.3) spostare a destra gli elementi da a[**primo**] ad a[i-1] (usa k)

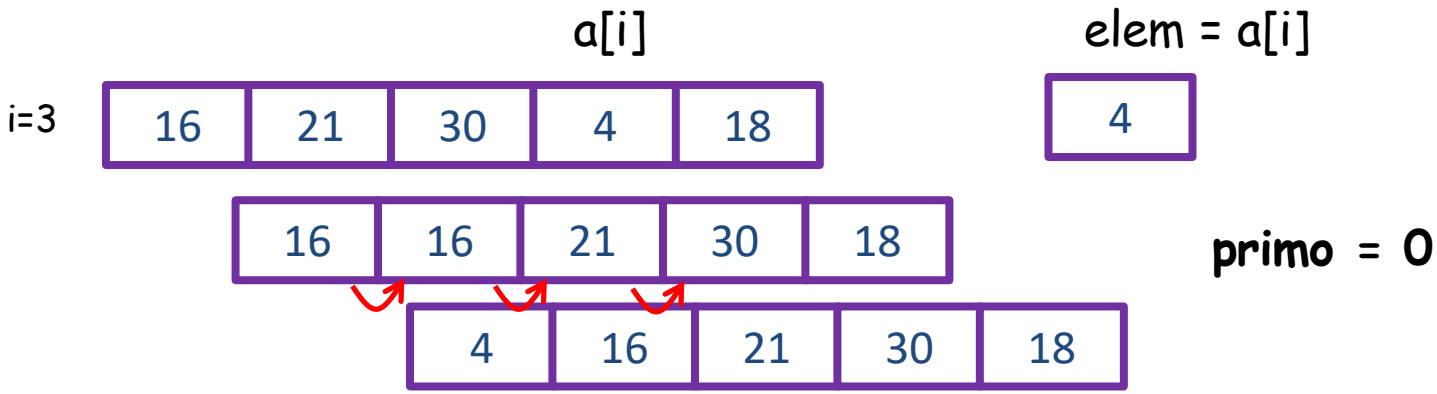
   1.4) a[**primo**] = a[i]

2) fine

Insertion Sort: implementazione in una funzione

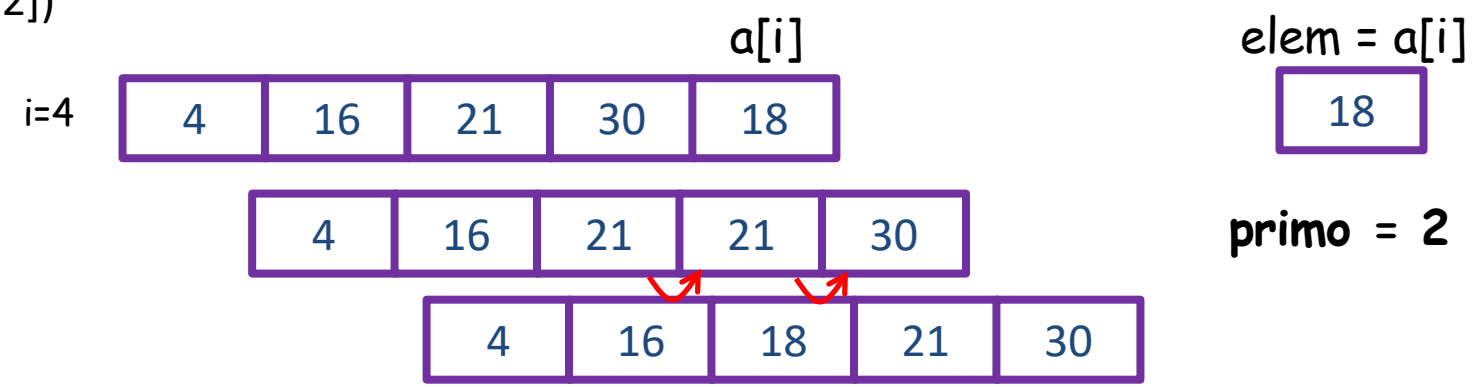
riguardando la passata 3 …

dopo lo scorrimento all'indietro, in cui confrontiamo elem con a[2], a[1] e a[0], siamo arrivati a 0 e quindi primo=0 (il primo element che dovrà spostarsi è a[0])



riguardando la passata 4 …

dopo lo scorrimento all'indietro, in cui confrontiamo elem con a[3], a[2], a[1] e a[0], siamo arrivati a 2 (perché a[1]<elemn) e quindi primo=2 (il primo element che dovrà spostarsi è a[2])



funzione insertionSort()

```
void insertionSort (int arr[N]) {
  int i, elem, primo, k;

  for (i=1; i<=N-1; i++) {

     elem = arr[i];       /* 1.1) */
     primo=i;       /* primo è inizializzato con i; poi dovremo trovare il
                       suo giusto valore procedendo all'indietro */

/* trovare 'primo' = "indice del primo elemento in a[0]…a[i-1] che è > di a[i]" */
     while ( ... arr[primo-1]>elem) {
                       /* è più grande … vediamo se ce n'è uno prima */
        primo--;
     }
/* si esce appena arr[primo-1] NON è maggiore di elem */

/* ora 1.3) … spostare a destra …*/
...
/* e poi 1.4) */
     arr[primo] = elem;
  } /* fine for */
                                          return;}
```

funzione insertionSort()

```
void insertionSort (int arr[N]) {
  int i, elem, primo, k;

  for (i=1; i<=N-1; i++) {

     elem = arr[i];       /* 1.1) */
     primo=i;       /* primo è inizializzato con i; poi dovremo trovare il
                       suo giusto valore procedendo all'indietro */

/* trovare 'primo' = "indice del primo elemento in a[0]…a[i-1] che è > di a[i]" */
while ( ... arr[primo-1]>elem) {
        primo--;
     }
                    /* ora 1.3) … spostare a destra …*/
     for(k=i-1; k>=primo; k--)
       arr[k+1] = arr[k];

/* e poi 1.4) */
     arr[primo] = elem;
  } /* fine primo for */
                                        return;}
```

funzione insertionSort()

```
void insertionSort (int arr[N]) {
  int i, elem, primo, k;

  for (i=1; i<=N-1; i++) {

     elem = arr[i];        /* 1.1) */
     primo=i;       /* primo è inizializzato con i; poi dovremo trovare il
                       suo giusto valore procedendo all'indietro */

/* trovare 'primo' = "indice del primo elemento in a[0]…a[i-1] che è > di a[i]" */
     while ( ... arr[primo-1]>elem) {
        primo--;
     }
                    /* ora 1.3) … spostare a destra …*/
     for(k=i-1; k>=primo; k--)
       arr[k+1] = arr[k];

/* e poi 1.4) */
     arr[primo] = elem;
  } /* fine primo for */
                                   return;}
```

primo-1 potrebbe diventare -1?

funzione insertionSort()

```
void insertionSort (int arr[N]) {
  int i, elem, primo;

  for (i=1; i<=N-1; i++) {

     elem = arr[i];        /* 1.1) */
     primo=i;       /* primo è inizializzato con i; poi dovremo trovare il
                       suo giusto valore procedendo all'indietro */

/* trovare 'primo' = "indice del primo elemento in a[0]…a[i-1] che è > di a[i]" */
     while ( primo>0 && arr[primo-1]>elem) {
        primo--;
     }
                    /* ora 1.3) … spostare a destra …*/
     for(k=i-1; k>=primo; k--)
       arr[k+1] = arr[k];

/* e poi 1.4) */
     arr[primo] = elem;
  } /* fine primo for */
                                    return;}
```

Insertion Sort: seconda implementazione in una funzione

funzione insertionSort2()

Algoritmo più sintetico … i passi 1.2 e 1.3 della versione precedente sono condensati

0) void; riceve int a[]; usa i, … elem, primo

1) per i da 1 a n-1, mettere a posto a[i] assumendo che a[0]-a[i-1] siano a posto

   1.1) elem = a[i]

   1.2) prosegui all'indietro, da primo=i, decrementando primo, verso l'elemento > elem con indice più piccolo possibile;
   se primo diventa zero, fermati;
   se a[primo-i] non è >= elem fermati;
   ad ogni iterazione (quindi mentre primo>0 and a[primo-i] >= elem),
   - sposta a[primo-i] a destra
   - decrementa primo

   1.3) a[**primo**] = a[i]

2) FINE

funzione insertionSort2()

```
void insertionSort2 (int arr[N]) {
  int i, elem, primo;

  for (i=1; i<=N-1; i++) {
     elem = arr[i];        /* dopo lo metteremo in arr[primo] */
     primo=i;              /* primo è init con i; poi dovremo trovare il
                              suo giusto valore procedendo all'indietro */

/* prosegui all'indietro cercando il primo elemento > elem; se non sei arrivato a
   zero, e se arr[primo-1] è più grande di elem, arr[primo-1] va spostato a
   destra; sennò hai trovato dove mettere elem (e hai spostato tutti gli elementi
   che dovevi spostare per fargli spazio) */
     while (primo>0 && arr[primo-1]>elem) {
        arr[primo] = arr[primo-1];
        primo--;
     }
/* all'uscita dal ciclo rimane solo da piazzare elem in arr[primo] */
     arr[primo] = elem;
  } /* fine for */
return;
}
```

funzione insertionSort2() --- analisi

```
void insertionSort2 (int arr[N]) {
  int i, j, min;
  for (i=1; i<=N-1; i++) {
     elem = arr[i];        /* dopo lo metteremo in arr[primo] */
     primo=i;              /* da trovare */
     while (primo>0 && arr[primo-1]>elem) {
        arr[primo] = arr[primo-1];
        primo--;
     }
   arr[primo] = elem;
return;
}
```

il confronto viene eseguito
- **1 volta alla prima passata**
- **2 …**
- **… k volte alla k-esima**
- **n-1 volte alla n-1-esima passata**

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

(Usiamo n invece di N nelle formule)

cioè
- **nel caso peggiore** (il ciclo while arriva sempre a 0, ovvero, ogni passata sposta tutti i possibili elementi) **il confronto è fatto un numero di volte proporzionale a n^2**

- nel caso migliore (sequenza già ordinata e quindi un solo confronto per passata) il costo è proporzionale a n