

anno 2014-15  
Introduzione all'Algoritmica per i Licei.

7 – Algoritmi di ordinamento elementari.

Elio Giovannetti  
Dipartimento di Informatica  
Università di Torino

versione 25 febbraio 2015



Quest'opera è distribuita con [Licenza Creative Commons](http://creativecommons.org/licenses/by-nc-sa/3.0/it/legalcode)  
[Attribuzione - Non commerciale - Condividi allo stesso modo 3.0 Italia.](http://creativecommons.org/licenses/by-nc-sa/3.0/it/legalcode)  
<http://creativecommons.org/licenses/by-nc-sa/3.0/it/legalcode>

# Il problema dell'ordinamento

Data una sequenza di elementi, ordinarla rispetto ad una relazione d'ordine definita sugli elementi (di solito su un particolare campo o componente di tali elementi);  
ad es. ordinare un insieme di oggetti **Studente** rispetto al numero di matricola (costituente la chiave identificativa dell'oggetto), oppure rispetto al cognome, ecc.

Gli algoritmi che risolvono tale problema, cioè gli algoritmi di ordinamento, sono classificabili secondo diversi criteri, quali la semplicità di realizzazione, l'efficienza, oppure il fatto che godano o no di certe proprietà interessanti.

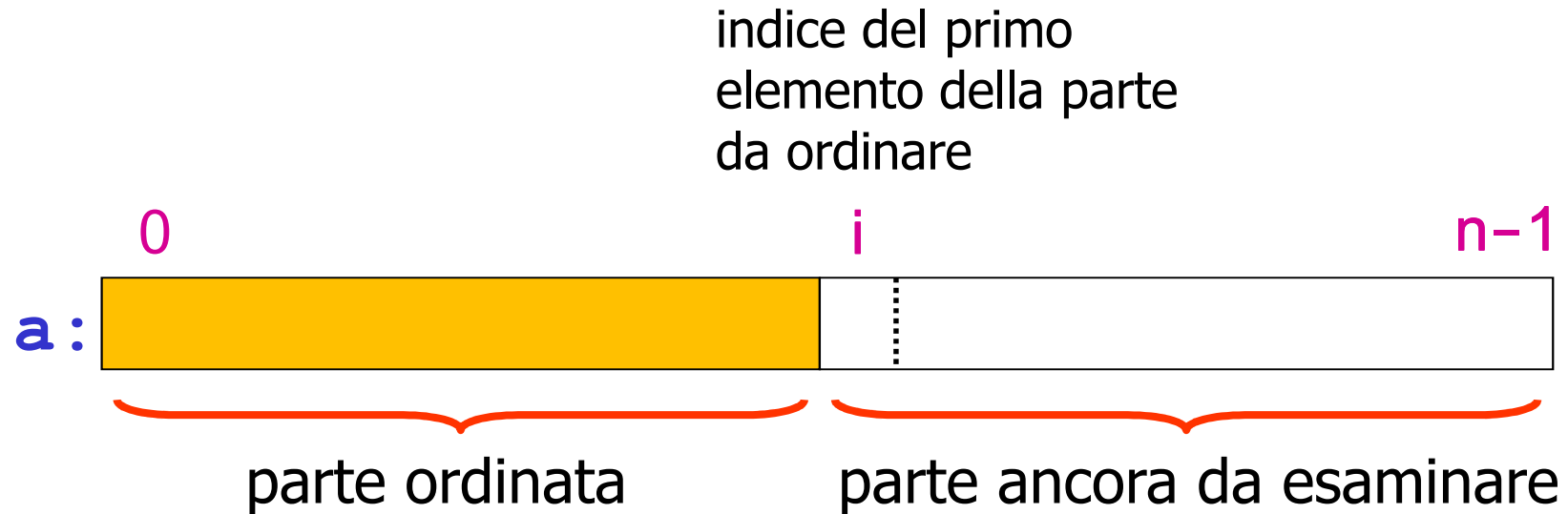
Algoritmo di ordinamento per selezione  
o **selection sort**  
o ordinamento per estrazione successiva del minimo

**Descrizione informale.**

Si cerca il minimo dell'array e lo si mette al primo posto, mettendo il primo al posto del minimo; poi si cerca il minimo nella parte di array dal secondo elemento alla fine, e lo si mette al secondo posto, e così via.

# Qual è la situazione al passo generico?

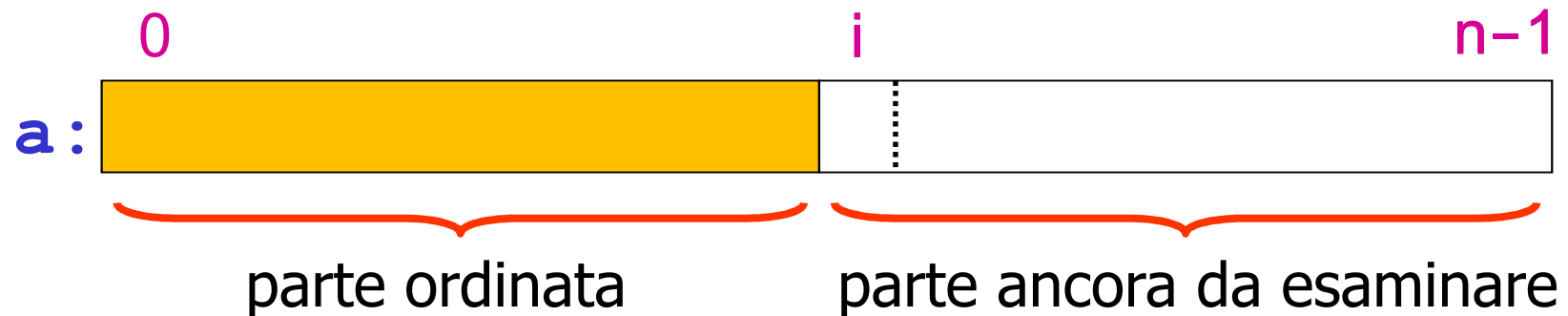
lista **a**, di lunghezza **n**



Ogni elemento della parte ordinata è minore (o uguale) di tutti gli elementi della parte ancora da esaminare.

Ciascun elemento della parte ordinata è quindi già al suo posto definitivo nella lista.

## Cosa bisogna fare nel corpo del ciclo principale?



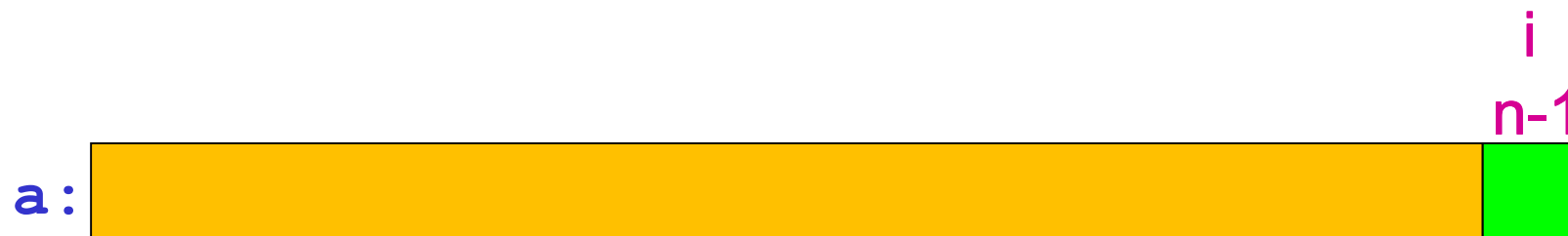
Cercare il minimo nella parte  $a[i : n]$  e scambiarlo con  $a[i]$ :

$iMin$  = indice dell'elemento minimo in  $a[i : n]$ ;

$a[i], a[iMin] = a[iMin], a[i]$

Per quali valori di  $i$  devono essere eseguite tali istruzioni?

Da  $i = 0$  (al primo passo si cerca il minimo di tutta la lista)  
fino a  $i = \dots ?$



Non c'è bisogno di controllare l'ultimo elemento, di indice  $n-1$ , poiché esso risulta automaticamente al posto giusto: essendosi estratto ogni volta il minimo degli elementi rimasti da esaminare, l'ultimo elemento rimasto è il massimo. Quindi:

```
n = len(a)
for i in range(n-1):
    iMin = = indice dell'elemento minimo in a[i : n];
    a[i], a[iMin] = a[iMin], a[i]
```

## Lo schema della procedura

```
def ssort(a):  
    n = len(a)  
    for i in range(n-1):  
        iMin =  
  
        a[i], a[iMin] = a[iMin], a[i]
```

Troviamo l'indice del minimo in  $a[i : n]$

```
def ssort(a):  
    n = len(a)  
    for i in range(n-1):  
        iMin = i  
        for j in range(i+1, n):  
            if a[j] < a[iMin]: iMin = j  
        a[i], a[iMin] = a[iMin], a[i]
```



## Tempo di calcolo.

Quanti passi fa l'algoritmo per ordinare una lista di  $n$  elementi?

- cerca il minimo della lista, percorrendola tutta:  $n$  passi
- cerca il minimo dal secondo in poi:  $n-1$  passi
- cerca il minimo dal terzo in poi:  $n-2$  passi
- ...

Quanti passi fa in tutto?

$$n + (n-1) + (n-2) + (n-3) + \dots + 2$$

Cioè:  $[n + (n-1) + (n-2) + (n-3) + \dots + 2 + 1] - 1$

è la somma dei numeri interi da 1 a  $n$  (meno uno).

Ma è la formula scoperta da Gauss bambino!

$$n(n+1)/2 \quad \text{che è uguale a } (n^2 + n)/2$$

Il numero di passi, e quindi il tempo di calcolo, cresce come il quadrato del numero degli elementi.

## Crescita quadratica del tempo

Se  $n$  raddoppia diventando  $2n$ , il quadrato diventa  $(2n)^2 = 4n^2$ , cioè quadruplica.

Se  $n$  si triplica diventando  $3n$ , il quadrato diventa  $(3n)^2 = 9n^2$ , cioè si moltiplica per  $9$ . Ecc.

Quindi se il numero di elementi della lista raddoppia, il tempo necessario per ordinare la lista quadruplica;

se il numero di elementi si triplica, il tempo necessario per ordinare la lista si moltiplica per  $9$ ;

se il numero di elementi si decuplica, il tempo necessario si moltiplica per  $100$ ; ecc.

Inoltre il numero di passi, per una lista di una data lunghezza, è sempre lo stesso, **indipendente dalla disposizione degli elementi**.

Ad esempio è lo stesso anche se la lista è già ordinata, o quasi ordinata.

# Algoritmo di ordinamento per inserimento o **insertion sort** (isort)

## Descrizione informale.

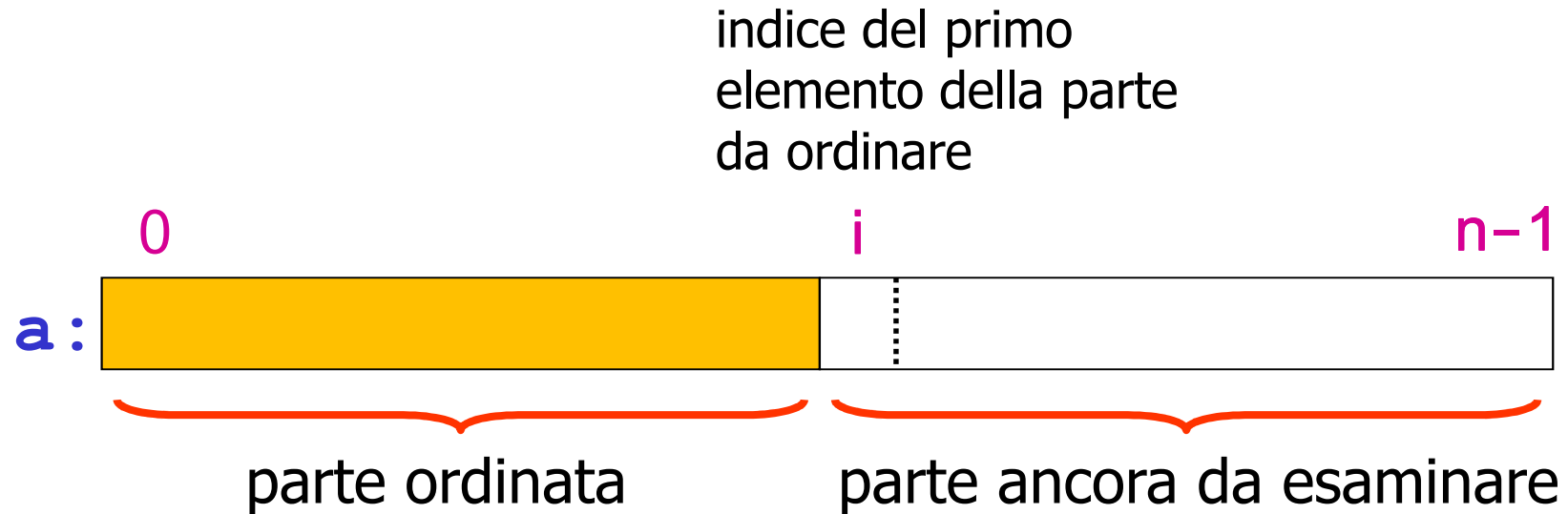
Si prende via via l'elemento successivo della lista e lo si inserisce, rispettando l'ordine, nella parte già ordinata, come quando, ricevute il proprio mazzetto di carte, se ne prende in mano una dopo l'altra.

Attenzione: "ordinamento per inserimento" è il nome di un *algoritmo di ordinamento*, cioè di un algoritmo che prende come argomento una sequenza esistente e la ordina;

NON è una procedura che inserisce in modo ordinato degli elementi in una sequenza vuota, pur basandosi sullo stesso principio di essa.

# Qual è la situazione al passo generico?

lista **a**, di lunghezza **n**



A differenza del selection sort, degli elementi ancora da esaminare non sappiamo nulla!



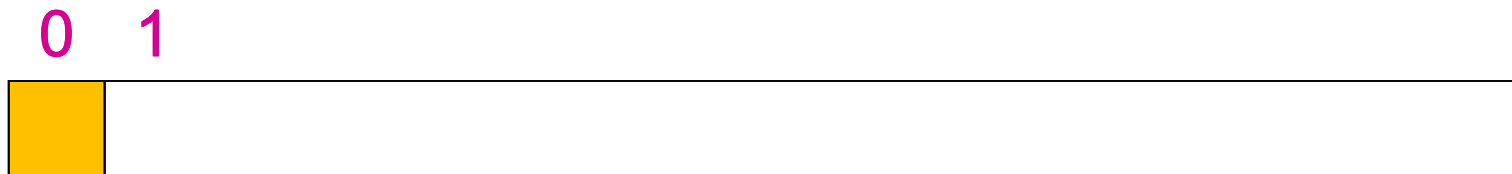
# Schema dell'algoritmo

```
for i in range(?, ?):  
    prendi l'elemento a[i]  
    e inseriscilo al posto giusto in a[0 .. i-1],  
    spostando gli elementi successivi per fargli spazio.
```

Da quale valore di  $i$  si inizia?

Per poter effettuare l'inserimento ci deve essere una porzione iniziale già ordinata. ma la porzione costituita dal solo primo elemento  $a[0]$  è banalmente ordinata!

Si inizia quindi dal secondo elemento,  $a[1]$ .



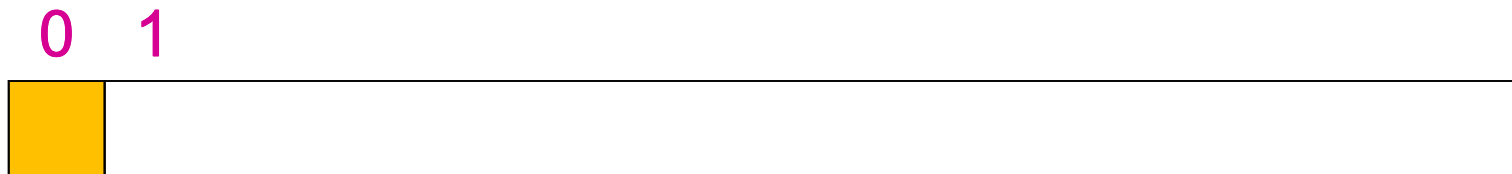
# Schema dell'algoritmo

```
for i in range(1, ?):  
    prendi l'elemento a[i]  
    e inseriscilo al posto giusto in a[0 .. i-1],  
    spostando gli elementi successivi per fargli spazio.
```

Da quale valore di  $i$  si inizia?

Per poter effettuare l'inserimento ci deve essere una porzione iniziale già ordinata. ma la porzione costituita dal solo primo elemento  $a[0]$  è banalmente ordinata!

Si inizia quindi dal secondo elemento,  $a[1]$ .



# Schema dell'algoritmo

```
for i in range(1, n):  
    prendi l'elemento a[i]  
    e inseriscilo al posto giusto in a[0 .. i-1],  
    spostando gli elementi successivi per fargli spazio.
```

Con quale valore di  $i$  si termina?

Quando arriviamo all'ultimo elemento, a differenza del `ssort` non sappiamo nulla di tale elemento, che va quindi inserito al posto giusto come tutti gli altri.





## Differenza fra il ssort e l'isort.

Si noti la differenza nella forma del ciclo for nei due algoritmi.

**ssort** (ordinamento per estrazione successiva del minimo):

```
for i in range(0, n-1):
```

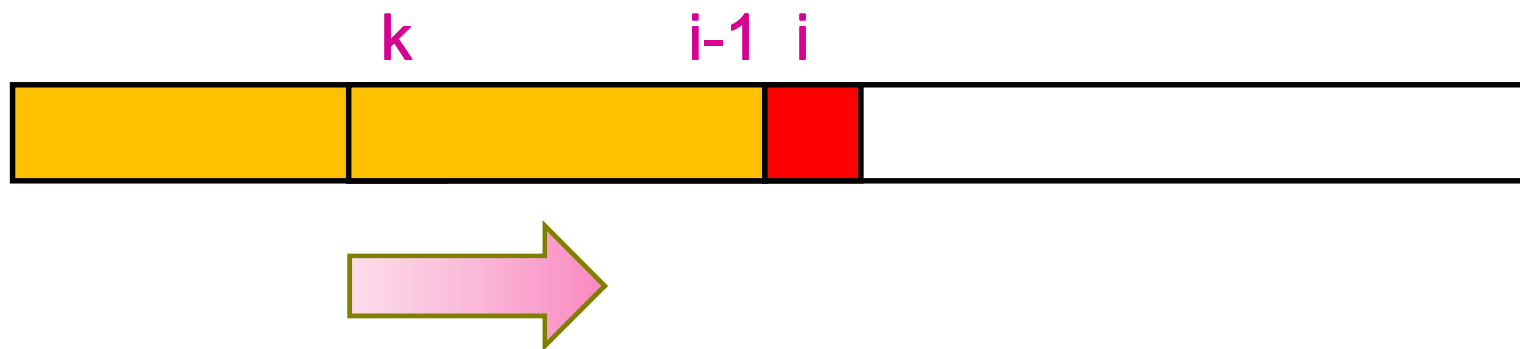
**isort** (ordinamento per inserimento):

```
for i in range(1, n):
```

# Come inserire un elemento al posto giusto

## Il passo da fare

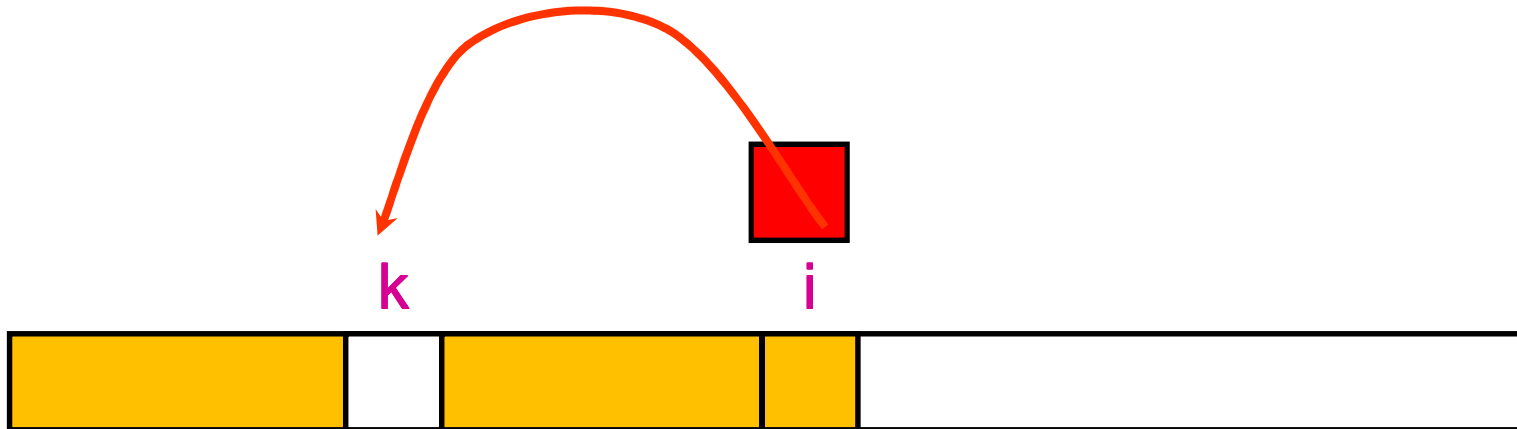
Vogliamo inserire l'elemento  $a[i]$  al posto giusto  $k$  (che dobbiamo scoprire qual è) in  $a[0 : i]$ :



# Ordinamento per inserimento

## Il passo da fare

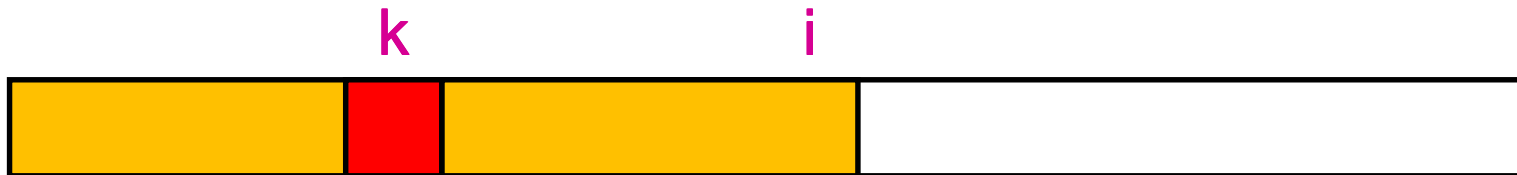
Vogliamo inserire l'elemento  $a[i]$  al posto giusto  $k$  (che dobbiamo scoprire qual è) in  $a[0 : i]$ :



# Ordinamento per inserimento

## Il passo da fare

Vogliamo inserire l'elemento  $a[i]$  al posto giusto  $k$  (che dobbiamo scoprire qual è) in  $a[0 : i]$ :



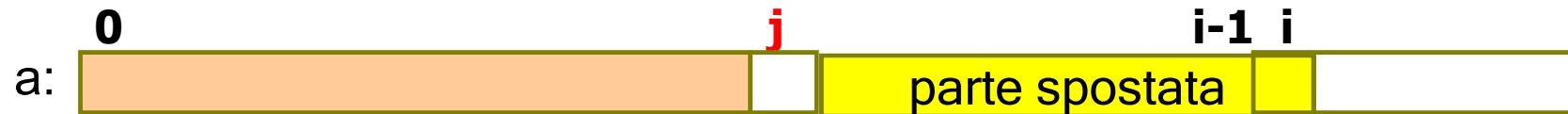
# Inserimento di $a[i]$ in $a[0 .. i-1]$

Situazione iniziale: la porzione  $a[0 : i]$  è ordinata



$elem = a[i]$  è l'elemento da inserire

## Inserimento di $a[i]$ in $a[0 .. i-1]$



`elem = a[i]` è l'elemento da inserire

A partire da  $a[i-1]$  compreso, sposto a destra di un posto tutti gli elementi di  $a[0 : i]$ :

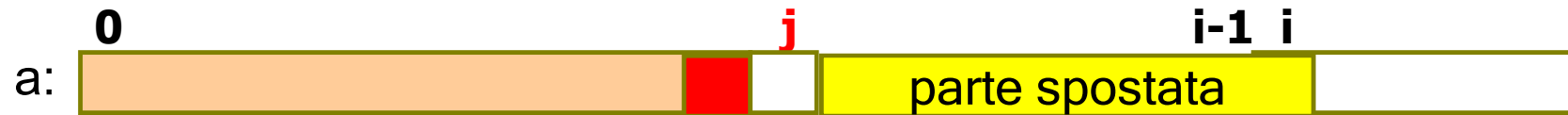
```
j = i
```

```
while j > 0:
```

```
    a[j] = a[j-1]
```

```
    j = j-1
```

## Inserimento di $a[i]$ in $a[0 .. i-1]$



`elem` è l'elemento da inserire

A partire da  $a[i-1]$  compreso, sposto a destra di un posto tutti gli elementi di  $a[0 : i]$ :

```
j = i
```

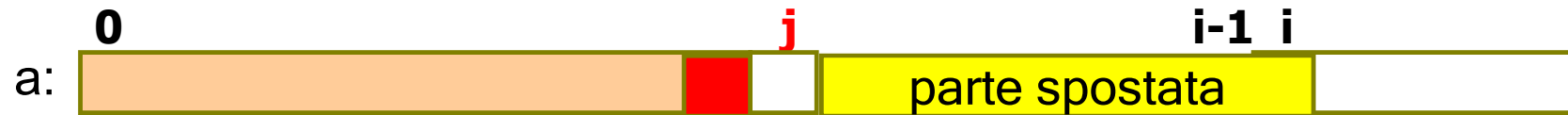
```
while j > 0:
```

```
    se però elem è  $\geq$  dell'elemento da spostare, mi fermo
```

```
    a[j] = a[j-1]
```

```
    j = j-1
```

## Inserimento di $a[i]$ in $a[0 .. i-1]$



`elem` è l'elemento da inserire

A partire da  $a[i-1]$  compreso, sposto a destra di un posto tutti gli elementi di  $a[0 : i]$ :

```
j = i
while j > 0:
    if elem >= a[j-1]: break # va inserito in a[j]
    a[j] = a[j-1]
    j = j-1
a[j] = elem
```



## Inserimento di $a[i]$ in $a[0 .. i-1]$

Se si è usciti dal while regolarmente, il posto vuoto è  $a[0]$ , quindi elem deve essere inserito in  $a[0]$

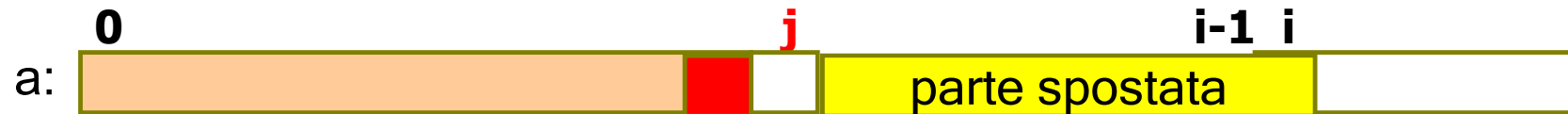


$elem$  è l'elemento da inserire

ma in tal caso  $j = 0$ , quindi  $elem$  va inserito in ogni caso in  $a[j]$

```
j = i
while j > 0:
    if elem >= a[j-1]: break # va inserito in a[j]
    a[j] = a[j-1]
    j = j-1
a[j] = elem
```

## Inserimento di $a[i]$ in $a[0 .. i-1]$



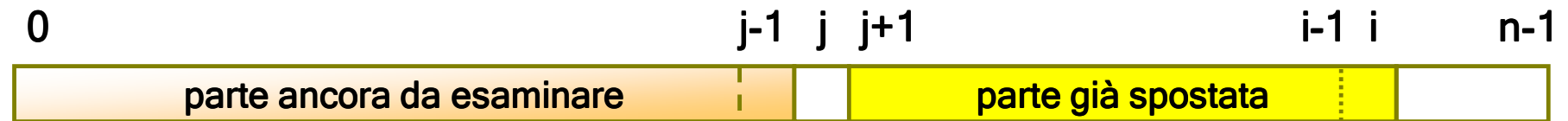
`elem` è l'elemento da inserire

ma uscire dal ciclo per `elem >= a[j-1]` equivale a continuare per `elem >= a[j-1]`

```
j = i
while j > 0 and elem < a[j-1]:
    a[j] = a[j-1]
    j = j-1
a[j] = elem
```

Ci si può arrivare anche con un ragionamento diretto.

Situazione al passo generico.




L'elemento da inserire  $elem$  è minore di tutti gli elementi "gialli" già spostati, e  $a[j]$  è il posto libero (che si è creato spostando a destra di uno tutti gli elementi successivi)





## La procedura completa

```
def isort(lista):  
    n = len(lista)  
    for i in range(1, n):  
        elem = lista[i]  
        j = i  
        while j > 0 and elem < lista[j-1]:  
            lista[j] = lista[j-1]  
            j = j-1  
        #end while  
        lista[j] = elem  
    #end for
```



## La procedura completa

```
def isort(lista):  
    n = len(lista)  
    for i in range(1, n):  
        elem = lista[i]  
        j = i  
        while j > 0 and elem < lista[j-1]:  
            lista[j] = lista[j-1]  
            j = j-1  
        lista[j] = elem
```

## Tempo di calcolo

L'algoritmo è costituito da un ciclo for di  $n$  passi, senza uscite forzate. Tuttavia all'interno di tale ciclo viene eseguito un ciclo interno di inserimento, il cui tempo di calcolo non è costante, ma in generale sarà dipendente dal valore di  $i$ .

**In media**, tale ciclo interno troverà il posto giusto all'incirca a metà della parte già ordinata, e farà quindi un numero di passi circa uguale a  $i/2$ ,

cioè  $1/2$  la prima volta,  $2/2$  la seconda,  $3/2$  la terza, ...;

approssimativamente, il numero totale di passi sarà allora:

$$1/2 (1 + 2 + 3 + \dots + n) = 1/2 (n(n+1)/2) = (n^2 + n)/4$$

Il tempo di calcolo cresce in modo **quadratico** rispetto alla lunghezza dell'array da ordinare, cioè è circa proporzionale al quadrato di tale lunghezza, come per il `ssort`.



## Tempo di calcolo (continua)

**caso peggiore:** l'array di partenza è ordinato inversamente; allora ogni esecuzione dell'algoritmo *inserisci* effettua  $k$  passi, con  $k$  successivamente uguale a  $1, 2, \dots, n$  (verificare simulando a mano l'esecuzione);

il numero totale di passi è quindi:

$$1+2+3+ \dots + n = n(n+1)/2 = (n^2 + n)/2$$

**quadratico**

**caso migliore:** l'array di partenza è già ordinato o ha pochi elementi "fuori posto"; allora ogni inserimento, inserendo un elemento che è maggiore o uguale dei precedenti, fa un solo passo (eccetto eventualmente i pochi elementi fuori posto).

Il numero totale di passi è quindi

$$1+1+1+ \dots (n \text{ volte}) (+ i \text{ passi per i pochi "fuori posto"})$$

cioè circa **proporzionale a  $n$**  (o **lineare in  $n$** ).

## Uso dell'algorithmo isort.

L'algorithmo di "ordinamento per inserimento" è quindi da scegliersi nei casi in cui, per qualche motivo, si sa a priori che la sequenza da ordinare è già quasi ordinata, cioè contiene un numero piccolo di elementi fuori posto.