

# *Con ordine, per favore*

corso di laurea in **Matematica**

*Informatica Generale*, Lezione **7(a)**

**Ivano Salvo**



**SAPIENZA**  
UNIVERSITÀ DI ROMA

# Il fascino della forza bruta ...



Cominciamo con un algoritmo **folle**: generare **tutte** le permutazioni del vettore, e **verificare** ciascuna di esse se sia ordinata o meno.

Facciamo conoscenza con **forall** che serve a enumerare tutti gli elementi che appartengono a un insieme (o a una sequenza). In realtà il ciclo **for** di Python equivale al nostro forall.

In questo caso è un'idea balorda ( $\theta(n!)$ , la transitività di  $\leq$  ci aiuterà molto), ma in alcuni casi è **l'unica cosa da fare** (eventualmente, con l'aiuto di qualche **euristica**).

```
def ordinaBruteForce(v):  
    forall u ∈ permutazioni(v):  
        if ordinato(u): return u
```



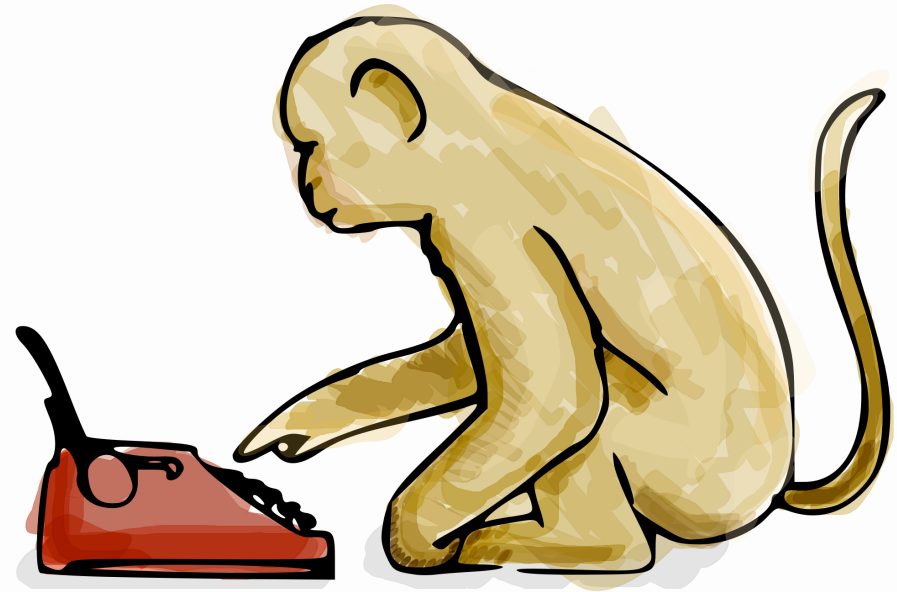
## ... e quello del caso

In alternativa, potremo incaricare una scimmia di “**scombinare**” o **mescolare**, gli elementi del vettore, fino a raggiungere una configurazione ordinata (**Emil Borel** docet).

Ovviamente sto cercando un **ago in un pagliaio**: c'è un'unica permutazione ordinata tra  $n!$ .

Riflettete su quante volte, mescolando un mazzo di carte, vi riconoscete un qualsiasi ordine!

Anche questa è un'idea balorda, ma in **alcuni casi**, se usati bene, usare **alcuni tentativi casuali** possono essere **molto efficaci**.



```
def ordinamentoRandom(v):  
    v = estraiPermutazione(v)  
    if !ordinato(v):  
        ordinamentoRandom(v)
```

# *Limiti inferiori/superiori dei problemi*

Cercare **esaustivamente** nello spazio delle soluzioni dà un **limite superiore** (usualmente **banale**) alla complessità di un problema.

Il tempo di **verifica** che una **soluzione sia corretta** dà un **limite inferiore** alla complessità di un problema.

**Esempio:** Abbiamo visto che verificare se una sequenza sia ordinata costa  $\theta(n)$  e quindi possiamo dire che l'ordinamento è almeno  $\Omega(n)$ .

Vedremo un **limite più stretto**  $\Omega(n \log n)$ .

La **complessità di un algoritmo corretto** per un problema  $P$  stabilisce un altro **limite superiore** (usualmente **più stretto** della ricerca esaustiva) alla complessità di  $P$ .

Oggi vedremo che l'ordinamento è un problema  $\mathcal{O}(n^2)$  esibendo degli algoritmi semplici di complessità quadratica.

Vedremo poi algoritmi che realizzano il limite inferiore  $\Omega(n \log n)$ .  
Un tale algoritmo è detto **asintoticamente ottimo**.

# *Algoritmi naif di ordinamento*

corso di laurea in **Matematica**

*Informatica Generale*, Lezione **9.1**

**Ivano Salvo**



**SAPIENZA**  
UNIVERSITÀ DI ROMA

# Selezionare i minimi successivi

Un' idea un po' più intelligente di quelle viste finora, è quella di selezionare i **minimi successivi**.

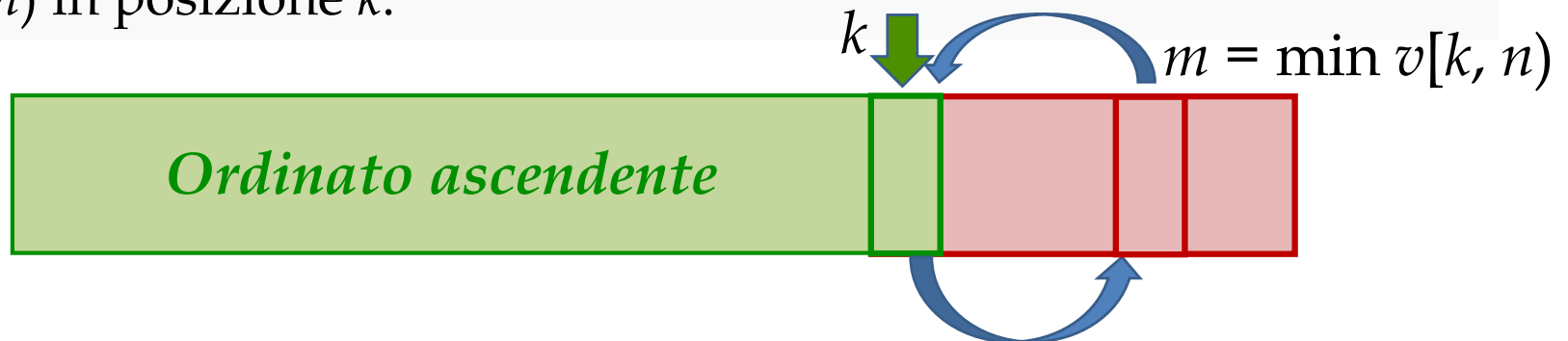
Il **minimo** del vettore viene messo al **primo posto** scambiandolo con l'elemento al primo, poi si seleziona il **minimo dei rimanenti** e lo si mette al **secondo posto** e così via...

A una generica iterazione, avremo sistemato  $k$  elementi al loro posto nella parte sinistra del vettore, e i  $k$  elementi della parte sinistra sono tutti minori di tutti gli  $n - k$  nella parte destra.

Abbiamo cioè soddisfatto l'invariante:

$$\varphi(k) \equiv \text{Asc}(v[0, k]) \ \& \ v[0, k] < v[k, n)$$

Siccome se  $\forall a \in A, b \in B. a \leq b$  allora  $\max A \leq \min B$ , si può soddisfare l'invariante per  $k+1$  semplicemente mettendo il minimo di  $v[k, n)$  in posizione  $k$ .



# Selection Sort: algoritmo e analisi

`selectionSort` termina perché fa sempre  $n$  cicli.

L'operazione **più costosa** del ciclo `for` è il **calcolo del minimo**, che è lineare nella lunghezza della porzione di vettore, quindi  $\theta(n - k)$ .  
Le altre sono operazioni  $\theta(1)$ .

Quindi la complessità è data  $\sum_{k=0, \dots, n-1} \theta(n - k) = n + (n+1) + \dots + 2 + 1$   
che è  $\frac{n(n+1)}{2}$  per la formula di Gauss, quindi  $\theta(n^2)$ .

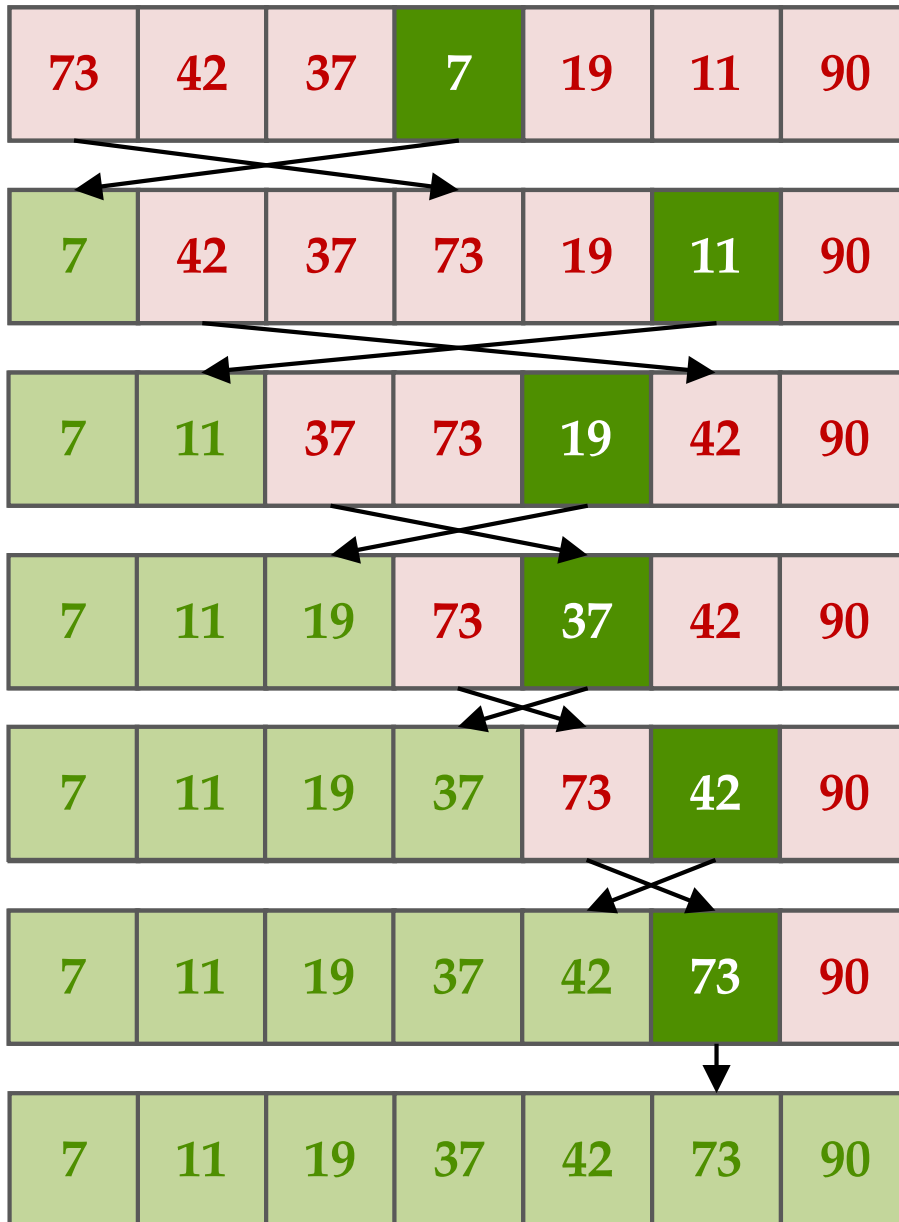
Osservate che, scritto così, fa **sempre esattamente  $n - 1$  scambi** e non trae vantaggio da situazioni fortunate, ad esempio, vettore ordinato.

**Esercizio:** dimostrate che  $\varphi(n-2)$  implica **Asc**( $v$ ), da cui la guardia.

**Esercizio:** verificare che l'invariante vale all'ingresso del `for`

```
def selectionSort(v):
    n = len(v)
    for k=0 to n-2:
        #INV: ASC(v[0,k) & v[0,k) ≤ v[k,n)
        m = minV(v, k, n)           # minimo parte destra
        v[k], v[m] = v[m], v[k]     # scambia
```

# Selection Sort: esempio



all'inizio il vettore  
è tutto rosso

calcolo il minimo della  
parte rossa...

...e lo metto al suo posto  
(a sinistra parte rossa).

osservate che la parte verde  
contiene gli elementi più piccoli

a volte non occorre scambiare  
nulla.

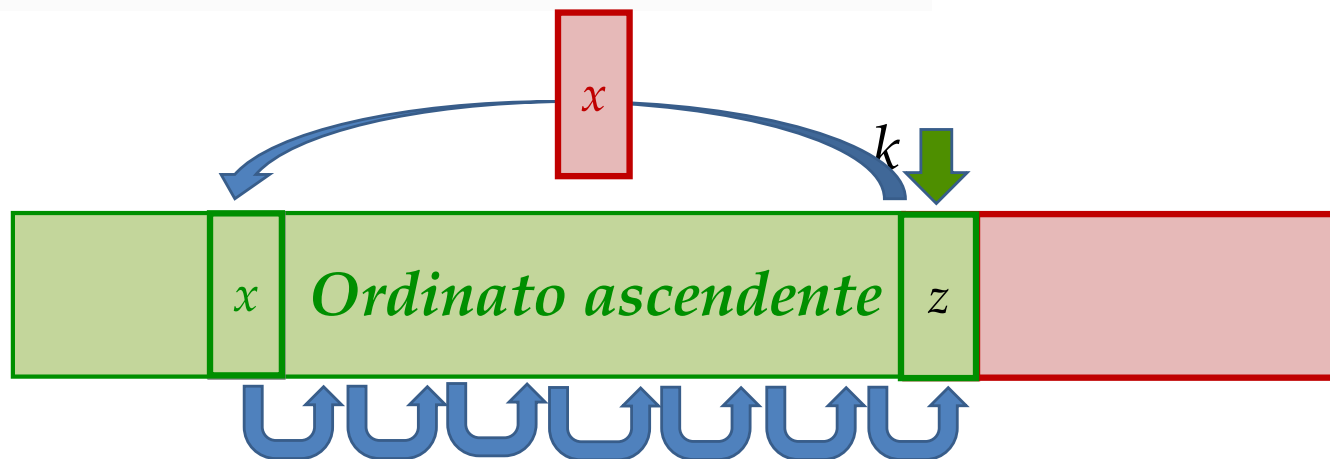
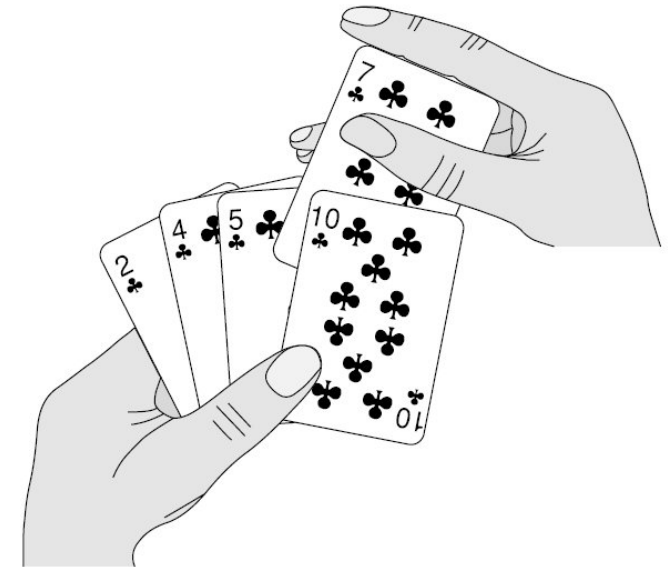
quando arrivo a sistemare il  
posto  $n-1$ , ho finito.

# Ordinamento del giocatore di carte

Un'idea leggermente diversa: mantenere ancora la **parte sinistra ordinata**, ma **senza richiedere** che **contenga gli elementi più piccoli**.

Occorre **'infilare' il nuovo elemento** (che potrebbe essere più piccolo di quelli già nella parte ordinata) **al posto giusto**, come fanno i **giocatori di carte**.

L'invariante è semplicemente  $\text{Asc}(v[0, k])$ . Occorre stare attenti a **non perdersi** gli elementi mentre si fanno scivolare a destra.



# Insertion Sort: algoritmo e analisi

Ecco la classica presentazione di insertionSort con **cicli annidati**.

Alla generica iterazione  $k$ , insertionSort garantisce **un'invariante più debole** di selectionSort:  $\text{Asc}(v[0, k])$ , che comunque corrisponde alla post-condizione  $\text{Asc}(v)$  per  $k = n$ .

Nel **caso pessimo** ( $\text{Disc}(v)$ ), selectionSort fa  $k$  scivolamenti per inserire  $v[k]$  (al primo posto) e quindi  $\sum_{k \in [0, n)} k = \frac{n(n+1)}{2} = \theta(n^2)$ .

Il **caso ottimo** è quando  $v$  soddisfa  $\text{Asc}(v)$ , in cui il ciclo interno non fa nessuno scivolamento, e quindi termina in  $\theta(n)$  confronti.

```
def insertionSort(v):  
  # ENS: Asc(v)  
  n = len(v)  
  for k=1 to n-1:  
    #INV: ASC(v[0,k])  
    x, j = v[k], k-1 # salvo in x val. da ins.  
    while x < v[j] and j ≥ 0: # trovo il punto  
      v[j+1], j = v[j], j-1 # e scivolo a dx  
    v[j+1]=x # sistemo x
```

Quali sono il caso  
ottimo e pessimo?

## Questione di stile...

Io ovviamente preferirei la seguente versione (che però **scorre due volte**  $v[0, k)$  per **inserire**  $v[k]$  al posto giusto:

- **prima si cerca il punto giusto**  $m$  (con apposita funzione *ricercaIns*, variazione della ricerca sequenziale),
  - poi usa la funzione *shiftRight* per **far scivolare a destra** gli elementi di  $v$  e **riempire il "buco"** lasciato **con**  $v[k]$ .
- ▶ **Esercizio:** scrivere pre- post- condizioni e codice di *ricercaIns* e *shiftRight*. Valutare la loro complessità e di *smartInsertionSort*.
- ▶ **Esercizio:** **modificare** la **ricerca binaria** per **trovare il punto dove inserire**. Come cambia la complessità di *smartInsertionSort*?

```
def smartInsertionSort(v):  
    # ENS: Asc(v)  
    n = len(v)  
    for k=1 to n-1:  
        #INV: ASC(v[0,k])  
        m = ricercaIns(v, 0, k, v[k])  
        shiftRight(v, m, k, v[k])
```

# Insertion Sort: esempio



il vettore  $v[0:1)$  è ordinato

prendo il primo elemento...

e lo inserisco ordinatamente,  
nella parte verde...

...e faccio scivolare gli altri

osservate che la parte verde  
al passo  $k$  contiene gli elementi  
che già erano nei primi  $k$  posti

Il 90 non deve sorpassare  
nessuno

# Ordinamento a Bolle

Tradizionalmente questo algoritmo viene ritenuto il più ovvio.

L'**idea** è molto semplice: si scorre il vettore e si **confronta ogni elemento con il successivo**. Se i due elementi **non stanno nell'ordine giusto, si scambiano**.

Alla **prima passata**, sono sicuro che ho portato il **massimo in fondo al vettore** (che è venuto **a galla** come una **bolla di sapone**, da cui il nome **Bubble Sort** o Ordinamento a Bolle). E poi via via si sistemano per certo il penultimo, e così via.

A ben vedere, questo è **del tutto simile a SelectionSort** (che potrebbe essere scritto a massimi successivi): la differenza è che selectionSort sistema il minimo con **1 solo** scambio.

**Conseguenza 1:** BubbleSort è in genere **meno efficiente** di SelectionSort, anche se il numero di **confronti è lo stesso**

**Conseguenza 2:** il lavoro in più fatto da BubbleSort si può sfruttare per fare delle **piccole ottimizzazioni**, che **non migliorano il caso pessimo** ma possono trarre vantaggio da situazioni fortunate.

# Bubble Sort: algoritmo e analisi

Versione **1** a cicli **fissi**: l'analisi è identica a SelectionSort.

Il ciclo esterno viene eseguito  $n - 1$  volte. Il costo del ciclo interno invece dipende da  $k$  ed è  $n - k - 1$ .

Ritroviamo la sommatoria  $n + (n - 1) + (n - 2) + \dots + 2 + 1 = \dots$

**Osservazione:** se il vettore **fosse già ordinato**, non ci sarebbe mai **nessuno scambio**, e la prima passata (interna) sarebbe sufficiente ad accorgersi di questa situazione.

```
def bubbleSort(v):
    n = len(v)
    for k=0 to n-2:
        #INV: ASC(v[n-k,n) & v[0,n-k) ≤ v[n-k,n)
        for j=1 to n-k-1:
            if a[j]<a[j-1]:          # scambia
                v[j], v[j-1] = v[j-1], v[j]
```

# Bubble Sort con sentinella

Possiamo subito sfruttare a buon mercato l'ultima osservazione.

È sufficiente introdurre una **variabile booleana** per memorizzare se durante una passata (ciclo interno) **si siano fatti scambi** o **meno**.

Se non ci sono stati scambi, abbiamo verificato che il vettore è ormai ordinato anche nella porzione  $v[0, n - k)$  e possiamo uscire.

Il **caso ottimo** diventa quando  $v$  è **ordinato**, che diventa  $\theta(n)$ .

```
def bubbleSort(v):
    n = len(v)
    for k=0 to n-2:
        #INV: ASC(v[n-k,n) & v[0,n-k) ≤ v[n-k,n)
        ord = True
        for j=1 to n-k-1:
            if a[j]<a[j-1]: # scambia
                v[j], v[j-1] = v[j-1], v[j]
                ord = False
        if ord: return
```

# Bubble Sort ottimizzato

Possiamo ricavare un'informazione più fine di quella booleana.

Possiamo memorizzare **l'indice *us* dell'ultimo scambio**: se non ci sono scambi, *us* sarà 0 e questo sarà **equivalente ad avere il valore *True* in *ord*** nel precedente algoritmo.

Ma possiamo accelerare la prossima passata che si può fermare a *us*.

```
def bubbleSort(v):
    n = len(v)
    sup = n-1
    while sup > 0:
        #INV:ASC(v[n-sup,n]&v[0,n-sup)≤v[n-sup,n)
        us = 0
        for j=1 to sup-1:
            if a[j]<a[j-1]: # scambia
                v[j], v[j-1] = v[j-1], v[j]
                us = j
        sup = us
```

Termina perché *sup*  
scende almeno di 1  
(sistemo il massimo  
in fondo)

# Bubble Sort: esempio, 1ma passata

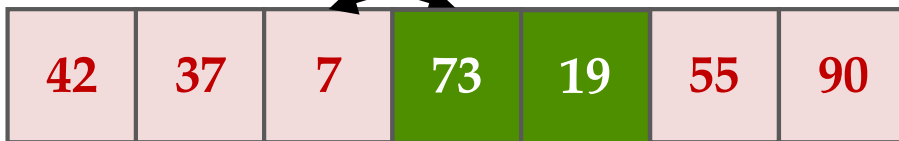
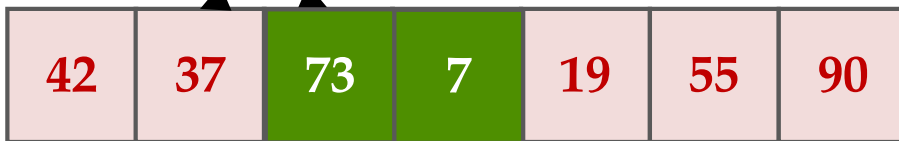


comincio coi primi due



e li scambio

e poi continuo

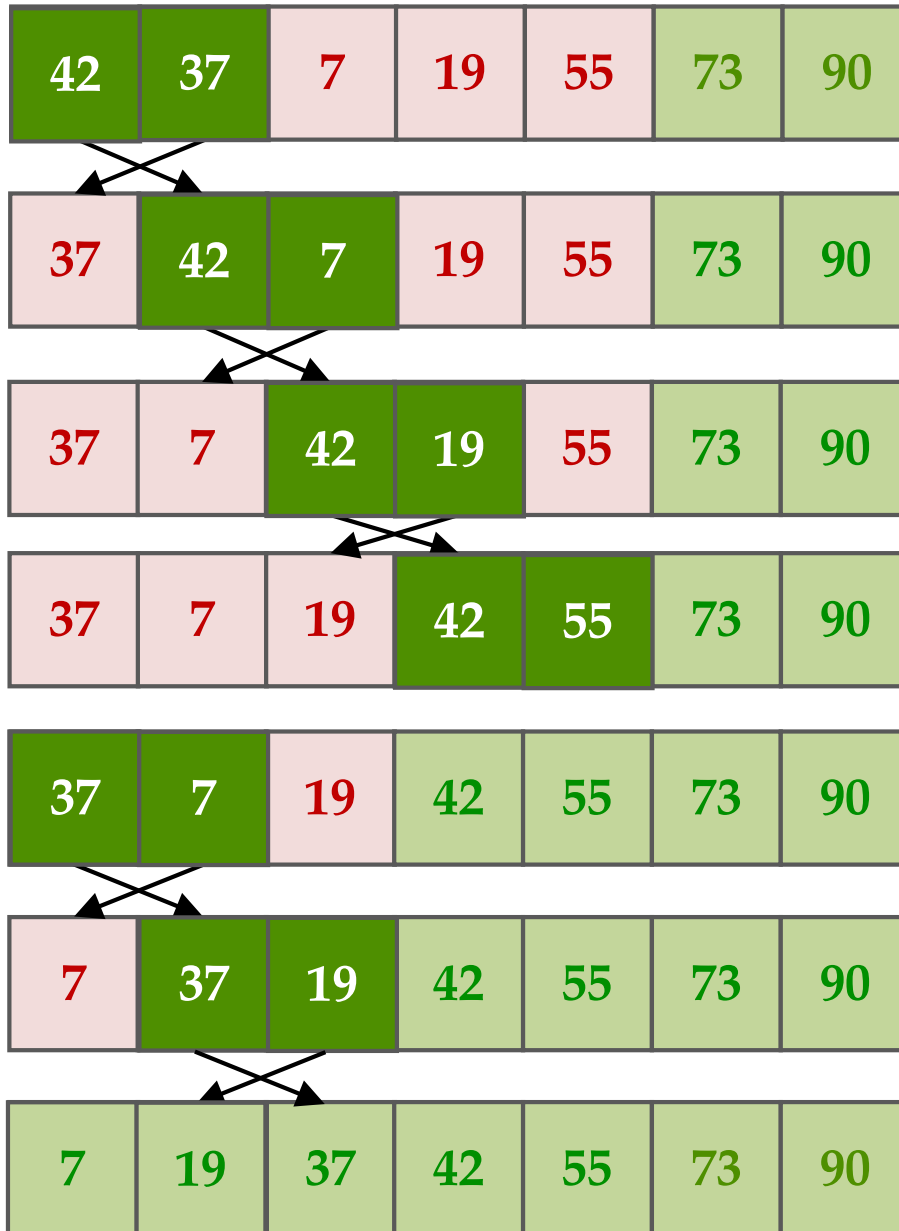


a volte non occorre scambiare nulla.



ho sistemato gli ultimi due, perché  $us = 4$

# Bubble Sort: esempio, altre passate



**R**icomincio coi primi due

e li scambio  
e poi continuo

fine 2da passata

fine 3za passata

fine ultima passata