

# Bubblesort e ordinamenti per confronti

Informatica@SEFA 2018/2019 - Lezione 14

Massimo Lauria <massimo.lauria@uniroma1.it>  
<http://massimolauria.net/courses/infosefa2018/>

Venerdì, 9 Novembre 2018

# Bubblesort

# Bubble sort stupido (I)

Abbiamo una sequenza di  $n$  elementi.

1. Mettiamo il più grande elemento nella posizione  $n - 1$
2. Mettiamo il secondo più grande elemento nella posizione  $n - 2$

...

Al passo  $i$ -esimo mettiamo l' $i$ -esimo elemento più grande alla posizione  $n - i$ .

# Invarianti

- ▶ Immediatamente prima di eseguire il passo  $i$  gli elementi nelle posizioni  $n - i + 1, \dots, n - 1$  sono nella loro posizione **definitiva**;
- ▶ al passo  $i$  mettiamo alla posizione  $n - i$  il più grande tra gli elementi nelle posizioni  $0, \dots, n - i$ ;
- ▶ la ricerca del più grande il suo spostamento sono fatte simultaneamente.

# Bubble up

Mette il massimo elemento alla fine, effettuando scambi a due a due.

```
def bubbleup(seq,end):                                     1
    for j in range(0,end): # si ferma a end-1              2
        seq[j], seq[j+1] = min(seq[j],seq[j+1]),max(seq[j],seq3
        [j+1])
```

```
bubbleup([2,4,1,3,6,5,2],6)                               1
```

```
Initial: [2, 4, 1, 3, 6, 5, 2]
0 vs 1 : [2, 4, 1, 3, 6, 5, 2]
1 vs 2 : [2, 1, 4, 3, 6, 5, 2]
2 vs 3 : [2, 1, 3, 4, 6, 5, 2]
3 vs 4 : [2, 1, 3, 4, 6, 5, 2]
4 vs 5 : [2, 1, 3, 4, 5, 6, 2]
5 vs 6 : [2, 1, 3, 4, 5, 2, 6]
```

Notate come il massimo sia messo sul fondo della lista.

# Bubble sort stupido (II)

L'ordinamento finale `stupid_bubblesort` si ottiene ripetendo la procedura: al passo  $i$  si opera sulla sottolista dalla posizione 0 alla posizione  $n - i$ .

```
def stupid_bubblesort(seq):           1
    for i in range(1, len(seq)):      2
        bubbleup(seq, len(seq)-i)    3
```

```
stupid_bubblesort([5, -4, 3, 6, 19, 1, -5])  1
```

```
Start   : [5, -4, 3, 6, 19, 1, -5] |
Step 1   : [-4, 3, 5, 6, 1, -5] | [19]
Step 2   : [-4, 3, 5, 1, -5] | [6, 19]
Step 3   : [-4, 3, 1, -5] | [5, 6, 19]
Step 4   : [-4, 1, -5] | [3, 5, 6, 19]
Step 5   : [-4, -5] | [1, 3, 5, 6, 19]
Step 6   : [-5] | [-4, 1, 3, 5, 6, 19]
```

# Osservazione su bubbleup

Sulla sequenza

```
[3, 2, 7, 1, 8, 9]
```

l'ultimo scambio effettuato è tra la posizione 2 e 3, ovvero quando si passa da

```
[2, 3, 7, 1, 8, 9]
```

a

```
[2, 3, 1, 7, 8, 9]
```

Nessuno scambio viene effettuato dalla posizione 3 in poi: quegli elementi sono già ordinati.

# Garanzie ulteriori di bubbleup

La funzione bubbleup ci fornisce delle garanzie che

- un'inversione tra posizione  $i$  e  $i + 1$  vuol dire:

*L'elemento alla posizione  $i + 1$  è maggiore di tutti i precedenti.*

- nessuna inversione dopo la posizione  $i$  vuol dire:

*Gli elementi dalla posizione  $i + 1$  in poi sono ordinati.*

Stiamo usando la prima ma non la seconda.



# Modifichiamo bubbleup

Memorizziamo l'ultimo scambio effettuato. Se la funzione restituisce una posizione  $j$

- ▶ gli elementi in  $\text{pos} > j$  sono ordinati
- ▶ sono maggiori degli elementi in  $\text{pos} \leq j$ .

```
def bubbleup(seq, end, log=False):           1
    last_swap = 0                             2
    for j in range(0, end):                   3
        if seq[j] > seq[j+1]:                 4
            last_swap = j                     5
            seq[j], seq[j+1] = seq[j+1], seq[j] 6
    return last_swap                          7
```

# Bubblesort

Usiamo queste garanzie che ci da bubbleup

```
def bubblesort(seq):  
    end=len(seq)-1  
    while end>0:  
        end=bubbleup(seq,end)
```

1  
2  
3  
4

```
bubblesort([3,2,7,1,8,9])
```

1

```
Start : [3, 2, 7, 1, 8, 9] |  
Step 1 : [2, 3, 1] | [7, 8, 9]  
Step 4 : [2, 1] | [3, 7, 8, 9]  
Step 5 : [1] | [2, 3, 7, 8, 9]
```

Questa versione fa meno passi: deve ordinare solo la parte di sequenza che precede l'ultimo scambio.

# Bubblesort su sequenze ordinate

```
stupid_bubblesort([1, 2, 3, 4, 5, 6, 7, 8])
```

1

```
Start   : [1, 2, 3, 4, 5, 6, 7, 8] |  
Step 1  : [1, 2, 3, 4, 5, 6, 7] | [8]  
Step 2  : [1, 2, 3, 4, 5, 6] | [7, 8]  
Step 3  : [1, 2, 3, 4, 5] | [6, 7, 8]  
Step 4  : [1, 2, 3, 4] | [5, 6, 7, 8]  
Step 5  : [1, 2, 3] | [4, 5, 6, 7, 8]  
Step 6  : [1, 2] | [3, 4, 5, 6, 7, 8]  
Step 7  : [1] | [2, 3, 4, 5, 6, 7, 8]
```

```
bubblesort([1, 2, 3, 4, 5, 6, 7, 8])
```

1

```
Start   : [1, 2, 3, 4, 5, 6, 7, 8] |  
Step 1  : [1] | [2, 3, 4, 5, 6, 7, 8]
```

# Bubblesort su sequenze invertite

```
stupid_bubblesort([8, 7, 6, 5, 4, 3, 2, 1])
```

1

```
Start   : [8, 7, 6, 5, 4, 3, 2, 1] |  
Step 1  : [7, 6, 5, 4, 3, 2, 1] | [8]  
Step 2  : [6, 5, 4, 3, 2, 1] | [7, 8]  
Step 3  : [5, 4, 3, 2, 1] | [6, 7, 8]  
Step 4  : [4, 3, 2, 1] | [5, 6, 7, 8]  
Step 5  : [3, 2, 1] | [4, 5, 6, 7, 8]  
Step 6  : [2, 1] | [3, 4, 5, 6, 7, 8]  
Step 7  : [1] | [2, 3, 4, 5, 6, 7, 8]
```

```
bubblesort([8, 7, 6, 5, 4, 3, 2, 1])
```

1

```
Start   : [8, 7, 6, 5, 4, 3, 2, 1] |  
Step 1  : [7, 6, 5, 4, 3, 2, 1] | [8]  
Step 2  : [6, 5, 4, 3, 2, 1] | [7, 8]  
Step 3  : [5, 4, 3, 2, 1] | [6, 7, 8]  
Step 4  : [4, 3, 2, 1] | [5, 6, 7, 8]  
Step 5  : [3, 2, 1] | [4, 5, 6, 7, 8]  
Step 6  : [2, 1] | [3, 4, 5, 6, 7, 8]  
Step 7  : [1] | [2, 3, 4, 5, 6, 7, 8]
```

# Prestazioni del Bubblesort

Come abbiamo visto ci sono casi in cui il bubblesort impiega  $O(n)$  operazioni ma anche casi in cui non si comporta meglio della versione stupida.

**Esercizio:** generate liste casuali e osservate la differenza di prestazioni tra

- insertion sort
- bubblesort stupido
- bubblesort

# Ordinamenti per confronti

# Risultati di impossibilità

Vogliamo migliorare il più possibile gli algoritmi che utilizziamo. Esistono tuttavia dei limiti insuperabili.

*E.g. La ricerca in una sequenza non ordinata richiede  $\Omega(n)$  operazioni.*

**Dimostrazione:** qualunque sia l'algoritmo, sappiamo che non deve saltare nessuna posizione nella sequenza, poiché l'elemento cercato potrebbe essere lì.

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

1	1	1	2	1	1	1	1
---	---	---	---	---	---	---	---

# Come si dimostra l'impossibilità?

Vogliamo trovare un algoritmo con determinate prestazioni, e questo può esistere (anche se è difficile scoprirlo/inventarlo) oppure non esistere affatto.

Come dimostrare che **esiste**?

- basta esibirlo

Come dimostrare che **non esiste**?

- **nessun algoritmo** ha le prestazioni richieste



# Limite degli ordinamenti per confronto

Gli algoritmi di ordinamento che abbiamo visto

- insertion sort
- bubble sort

sono algoritmi di **ordinamento per confronti** e usano  $O(n^2)$  operazioni. É possibile fare di meglio? E quanto meglio?

# Limite degli ordinamenti per confronto

Gli algoritmi di ordinamento che abbiamo visto

- insertion sort
- bubble sort

sono algoritmi di **ordinamento per confronti** e usano  $O(n^2)$  operazioni. É possibile fare di meglio? E quanto meglio?

## Theorem

*Un algoritmo di ordinamento per confronti **necessita** di  $\Omega(n \log n)$  operazioni per ordinare una lista di  $n$  elementi.*

# Esempio di ordinamento di tre elementi

Ci sono 6 modi di disporre  $\{a_0, a_1, a_2\}$  in sequenza, e possiamo effettuare **confronti** tra elementi per scoprire quale dei 6 modi mette  $\{a_0, a_1, a_2\}$  in ordine crescente.

- Se  $a_0 \leq a_1$ 
  - se  $a_1 \leq a_2$  **output**  $\langle a_0, a_1, a_2 \rangle$
  - altrimenti
    - se  $a_0 \leq a_2$  **output**  $\langle a_0, a_2, a_1 \rangle$
    - altrimenti **output**  $\langle a_2, a_0, a_1 \rangle$
- altrimenti
  - se  $a_0 \leq a_2$  **output**  $\langle a_1, a_0, a_2 \rangle$
  - altrimenti
    - se  $a_1 \leq a_2$  **output**  $\langle a_1, a_2, a_0 \rangle$
    - altrimenti **output**  $\langle a_2, a_1, a_0 \rangle$

# Prerequisito: permutazioni

Una permutazione su  $\{0, 1, \dots, n-1\}$  è una funzione

$$\pi : \{0, 1, \dots, n-1\} \rightarrow \{0, 1, \dots, n-1\}$$

tale che  $\pi(i) = \pi(j)$  se e solo se  $i = j$ .

Una permutazione è completamente descritta da

$$(\pi(0), \pi(1), \dots, \pi(n-1))$$

Esempi per  $n = 6$ :

- $(1, 4, 3, 2, 0, 5)$
- $(5, 4, 3, 2, 1, 0)$
- $(0, 1, 2, 3, 4, 5)$

# Prerequisito: permutazioni (II)

Le permutazioni su  $\{0, \dots, n-1\}$  con operazione  $\pi\rho$  che denota  $i \mapsto \rho(\pi(i))$ .

Struttura di gruppo algebrico

- ▶ **Identità:** esiste  $\pi$  per cui  $\pi(i) = i$  per ogni  $i$ ;
- ▶ **Associatività:**  $\pi_1(\pi_2\pi_3) = (\pi_1\pi_2)\pi_3$
- ▶ **Inversa:** per ogni  $\pi$  esiste **un'unica** permutazione, che denotiamo come  $\pi^{-1}$  per cui  $\pi\pi^{-1}$  è la permutazione identica.

$$n = 6 \quad \pi = (1, 4, 3, 5, 2, 0) \quad \pi^{-1} = (5, 0, 4, 2, 1, 3)$$

# Ordinamento

Un algoritmo di ordinamento prende in input una sequenza

$$\langle a_0, a_1, a_2, a_3, \dots, a_{n-1} \rangle \quad (1)$$

ed essenzialmente calcola una permutazione  $\pi$  sugli indici  $\{0, 1, \dots, n-1\}$  per cui

$$\langle a_{\pi(0)}, a_{\pi(1)}, a_{\pi(2)}, a_{\pi(3)}, \dots, a_{\pi(n-1)} \rangle \quad (2)$$

è una sequenza crescente.

# Esempio

Da un input

$$\langle a_0, a_1, a_2, a_3, a_4 \rangle = \langle 32, -5, 7, 3, 12 \rangle$$

un algoritmo di ordinamento produce la permutazione

$$(1, 3, 2, 4, 0)$$

che corrisponde all'output

$$\langle a_1, a_3, a_2, a_4, a_0 \rangle = \langle -5, 3, 7, 12, 32 \rangle$$

# Ordinamenti per confronti

Tutte le decisioni prese dall'algoritmo di basano sul confronto  $\leq$  tra due elementi della sequenza. Nel senso che

- le entrate e uscite dai cicli `while` e `for`
- la strada presa negli `if/else`
- come spostati gli elementi tra le posizioni della lista
- ...

non dipendono dai valori nella sequenza, ma solo dall'esito dei confronti.



# Esempio (ancora tre elementi)

- Se  $a_0 \leq a_1$ 
  - se  $a_1 \leq a_2$  **output**  $\langle a_0, a_1, a_2 \rangle$
  - altrimenti
    - se  $a_0 \leq a_2$  **output**  $\langle a_0, a_2, a_1 \rangle$
    - altrimenti **output**  $\langle a_2, a_0, a_1 \rangle$
- altrimenti
  - se  $a_0 \leq a_2$  **output**  $\langle a_1, a_0, a_2 \rangle$
  - altrimenti
    - se  $a_1 \leq a_2$  **output**  $\langle a_1, a_2, a_0 \rangle$
    - altrimenti **output**  $\langle a_2, a_1, a_0 \rangle$

La scelta della permutazione (delle 6 disponibili) dipende solo dall'esito dei confronti.

# Osservazione 1

*Se per due sequenze in input tutti i confronti danno lo stesso esito, un algoritmo di ordinamento per confronti produce la stessa permutazione  $\pi$  per entrambe.*

## Osservazione 2

Per ogni permutazione  $\pi$  di  $\{0, 1, \dots, n-1\}$ , esiste un input per cui questa permutazione è l'unico output corretto per un ordinamento.

**Dimostrazione:** Si prenda l'unica permutazione  $\pi^{-1}$  e si dia in input la sequenza

$$\pi^{-1}(0), \pi^{-1}(1), \pi^{-1}(2), \dots, \pi^{-1}(n-1)$$

che, una volta ordinata, risulta essere  $\{0, 1, 2, \dots, n-1\}$ . Questa sequenza può essere ordinata solo dalla permutazione  $\pi$  per definizione.

# Dimostrazione del limite $\Omega(n \log n)$

## Theorem

*Per qualunque algoritmo di ordinamento per confronti, esiste una sequenza di  $n$  elementi per cui l'algoritmo esegue  $\Omega(n \log n)$  confronti.*

## Proof.

- Sia  $h$  il massimo numero di confronti dell'algoritmo;
- al massimo  $2^h$  output distinti (**osservazione 1**);
- Ci sono  $n!$  permutazioni di  $n$  elementi e sono tutte possibili output (**osservazione 2**);
- Quindi  $2^h \geq n! \geq (n/2)^{n/2}$ , ovvero  $h \geq \Omega(n \log n)$ .



# Ricapitolando

- ordinamento per confronti richiede  $\Omega(n \log n)$  passi
- insertion sort  $\Theta(n^2)$  operazioni
- bubblesort  $\Theta(n^2)$  operazioni

vedremo

- ordinamenti per confronti con  $O(n \log n)$  operazioni
- (forse) un ordinamento che utilizza  $O(n)$  operazioni