Sapienza — Università di Roma
Dip. Scienze Statistiche — Informatica 2019/2020
II canale

Appunti su algoritmi e complessità

Massimo Lauria
<massimo.lauria@uniroma1.it>
https://massimolauria.net/courses/informatica2019

Versione degli appunti aggiornata al 11 ottobre 2019.

Nota bibliografica: Molto del contenuto di questi appunti può essere approfondito nei primi capitoli del libro *Introduzione agli Algoritmi e strutture dati* di Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein.

Indice

Capitolo 1

Ordinare i dati con Bubblesort

Il Bubblesort è una altro algoritmo di ordinamento, il cui comportamento è molto differente dall'insertion sort. Vedete dunque che per risolvere lo stesso problema, ovvero ordinare una lista, esistono più algoritmi anche profondamente diversi.

Per introdurre il Bubblesort partiamo da una sua versione molto semplificata, descritta nella prossima sezione.

1.1 Bubble sort semplificato

La versione semplificata del bubblesort ha uno schema che a prima vista è simile a quello dell'insertion sort, infatti la lista ordinata viene costruita passo passo, mettendo al suo posto definitivo un elemento alla volta. Partendo da una lista di n elementi facciamo i seguenti $n-1$ passi

1. Effettuando degli scambi di elementi nella lista che vedremo successivamente, mettiamo il più grande elemento nella posizione $n-1$. Questo elemento non verrà più toccato o spostato.

2. Mettiamo il secondo più grande elemento nella posizione $n-2$. Poiché al passo precedente abbiamo messo l'elemento più grande alla posizione $n-1$, vuol dire che il secondo elemento più grande non è altri che l'elemento più grande tra quelli nelle posizioni $0, 1, \ldots, n-2$.

3. Mettiamo il terzo elemento più grande nella posizione $n-3$. Poiché alle posizioni $n-2$ e $n-1$ ci sono i due elementi più grandi, il terzo

elemento più grande non è altri che l'elemento più grande tra quelli nelle posizioni $0, 1, \ldots, n-3$.

Continuando su questa falsariga, al passo i-esimo mettiamo l'i-esimo elemento più grande alla posizione $n-i$. Arrivati al passo $n-1$, ci vtroviamo con gli elementi nelle posizioni 0 e 1, e dobbiamo semplicemente mettere il più grande in posizione 1 e lasciare il più piccolo in posizione 0.

Da questa descrizione del bubble sort mancano molti dettagli: come si cerca l'elemento più grande nella lista? Come lo si mette alla fine della lista? Come sono effettuati questi scambi?

Da questa descrizione parziale osserviamo subito che immediatamente prima di eseguire il passo i gli elementi nelle posizioni $n-i+1, \ldots, n-1$ sono ordinati e nella loro posizione **definitiva**.

Una cosa da chiarire è come venga trovato l'elemento più grande della lista e messo all'ultima posizione. E più in generale, per il passo i-esimo, come venga trovato l'elemento più grande tra le posizioni $0, \ldots, n-i$ e messo in posizione $n-i$.

Vediamo che il seguente algoritmo, applicato alla lista `seq`, la scorre e scambia ogni coppia di elementi adiacenti, se non sono tra di loro in ordine crescente.

```
for j in range(0,len(seq)-1):
    if seq[j] > seq[j+1]:
        seq[j], seq[j+1] = seq[j+1], seq[j]
```

Vediamo un esempio di esecuzione di questa procedura sulla sequenza `[2,4,1,3,6,5,2]`

```
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, perché ad un certo punto l'elemento massimo risulterà sempre maggiore di quelli con cui viene confrontato e la sua posizione verrà sempre fatta avanzare. Notate anche come gli altri elementi della lista si trovano nelle posizioni $0, \ldots, n-2$ e quindi la procedura può essere ripetuta su questa parte di lista.

Al passi i-esimo la procedura sarà la stessa, ma limitata alle posizioni da 0 a $n-i$.

```
for j in range(0,len(seq)-i):
    if seq[j] > seq[j+1]:
        seq[j], seq[j+1] = seq[j+1], seq[j]
```

L'algoritmo finale sarà quindi ottenuto ripetendo la procedura: al passo i si opera sulla sottolista dalla posizione 0 alla posizione $n - i$. L'algoritmo completo è il seguente.

```
def stupid_bubblesort(seq):
    for i in range(1,len(seq)):
        for j in range(0,len(seq)-i):
            if seq[j] > seq[j+1]:
                seq[j], seq[j+1] = seq[j+1], seq[j]
```

Vediamo un esempio di esecuzione dell'algoritmo, mostrata passo passo.

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

```
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]
```

1.2 Miglioriamo l'algoritmo

Se eseguiamo la sequenza di scambi iniziali sulla lista `[3, 2, 7, 1, 8, 9]` ,vediamo che l'ultimo scambio effettuato è tra la posizione 2 e 3, ovvero quando la lista passa da `[2, 3, 7, 1, 8, 9]` a `[2, 3, 1, 7, 8, 9]`

```
Initial: [3, 2, 7, 1, 8, 9]
0 vs 1 : [2, 3, 7, 1, 8, 9]
1 vs 2 : [2, 3, 7, 1, 8, 9]
2 vs 3 : [2, 3, 1, 7, 8, 9]
3 vs 4 : [2, 3, 1, 7, 8, 9]
4 vs 5 : [2, 3, 1, 7, 8, 9]
```

Nessuno scambio viene effettuato dalla posizione 3 in poi, e questo ci garantisce che da quella posizione gli elementi sono già ordinati.

1.3 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.

1.4 Modifichiamo bubbleup

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

- gli elementi in pos $> j$ sono ordinati
- sono maggiori degli elementi in pos $\leq j$.

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

1.5 Bubblesort

Usiamo queste garanzie che ci da `bubbleup`

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

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

```
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.

1.6 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]
```

1.7 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]
```

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.

Per esercizio, generate liste casuali e osservate la differenza di prestazioni tra

- insertion sort
- bubblesort stupido
- bubblesort

Capitolo 2

Ordinamenti per confronti

2.1

Ordinamenti per confronti

2.2 Risultati di impossibilita

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. Testo:

1	1	1	1	1	1	1	1

2. Testo:

1	1	1	**2**	1	1	1	1

2.3 Come si dimostra l'impossibilita?

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

2.4 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?

Teorema 1. *Un algoritmo di ordinamento per confronti* ***necessita*** *di* $\Omega(n \log n)$ *operazioni per ordinare una lista di n elementi.*

2.5 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** $\langle a_2, a_1, a_0 \rangle$

2.6 Prerequisito: permutazioni

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

$$\pi : \{0, 1, \ldots n-1\} \to \{0, 1, \ldots n-1\}$$

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

Una permutazione è completamente descritta da

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

Esempi per $n = 6$:

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

2.7 Prerequisito: permutazioni (II)

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

Struttura di gruppo algebrico

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

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

2.8 Ordinamento

Un algoritmo di ordinamento prende in input una sequenza

$$\langle a_0, a_1, a_2, a_3, \ldots, a_{n-1} \rangle \tag{2.1}$$

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

$$\langle a_{\pi(0)}, a_{\pi(1)}, a_{\pi(2)}, a_{\pi(3)}, \ldots, a_{\pi(n-1)} \rangle \tag{2.2}$$

è una sequenza crescente.

2.9 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

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

2.10 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.

2.11 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.

2.12 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 π per entrambe.

2.13 Osservazione 2

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

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

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

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

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

Teorema 2. *Per qualunque algoritmo di ordinamento per confronti, esiste una sequenza di n elementi per cui l'algoritmo esegue* $\Omega(n \log n)$ *confronti.*

Dimostrazione.

- 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 $h \geq \Omega(n \log n)$.

□

2.15 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