# 2. Algoritmi di Ordinamento Introduzione – Algoritmi Elementari

Abbiamo studiato che negli array ordinati è possibile utilizzare la ricerca binaria, molto più veloce di quella sequenziale. Nella pratica spesso le applicazioni devono "mettere in ordine" dei dati. Daremo pertanto ampio spazio allo studio degli *algoritmi di ordinamento* (o *sorting*).

Vista l'importanza del tema, esistono numerose tecniche di ordinamento e diverse varianti del problema. Iniziamo a distinguere tra due tipi fondamentali di ordinamento:

- *Ordinamento interno:* i dati da ordinare sono contenuti all'interno della memoria centrale, tipicamente in strutture in cui è possibile l'accesso diretto ai singoli elementi, come gli array, nei quali il tempo di accesso a un elemento *non dipende dalla sua posizione*.
- *Ordinamento esterno:* gli elementi da ordinare si trovano nella memoria di massa. In questo caso non si accede direttamente ai singoli dati, ma a blocchi di dati che devono essere trasferiti nella memoria centrale per essere elaborati. Le prestazioni sono influenzate dalla velocità delle periferiche, che hanno tempi di accesso decisamente più elevati rispetto alla memoria centrale.

Presenteremo alcuni algoritmi per l'ordinamento interno, in cui ordineremo in modo non decrescente vettori di elementi. Per semplicità, negli esempi utilizzati per illustrare gli algoritmi, considereremo l'ordinamento di vettori di interi. Non dobbiamo tuttavia dimenticarci che, nella realtà, in genere si devono ordinare vettori i cui elementi sono record o strutture formate da diversi campi (ad esempio se un record descrive i dati anagrafici di una persona, i campi potrebbero essere nome, cognome, data di nascita, luogo di nascita, codice fiscale, ecc.). Uno dei campi del record è scelto come *campo chiave*. L'ordinamento dell'array avviene rispetto tale campo.

## Stabilità

Tra gli elementi da ordinare ci potrebbero essere record con la stessa chiave. Diciamo che un algoritmo di ordinamento è *stabile* quando due record che hanno la stessa chiave vengono mantenuti nello stesso ordine relativo dopo l'ordinamento. Ad esempio, dovendo ordinare i seguenti record contenenti un campo intero e un campo stringa, in cui la chiave è il campo intero:

Il presente materiale integra *ma non sostituisce* il libro di testo consigliato.
Data pubblicazione: 18 ottobre 2019

```
<10, tigre>  <12, cane>  <10, lepre>
```

dopo l'applicazione di un ordinamento stabile, il record `<10, tigre>` dovrà continuare a precedere il record `<10, lepre>`, come nella sequenza iniziale.

## Analisi degli algoritmi di ordinamento

Analizzeremo gli algoritmi di ordinamento valutandone la complessità in base al *numero di confronti* tra chiavi e allo *spazio* utilizzati, in funzione del numero $n$ di elementi da ordinare. Consideriamo il numero di confronti, e non direttamente il tempo, perché le operazioni di confronto saranno le operazioni "più costose" che utilizzeremo negli algoritmi. Per ordinare vettori di interi di dimensione costante (ad esempio rappresentati mediante tipi `int` o `long`), il tempo richiesto da ciascun confronto è costante. Pertanto dalla stima sul numero dei confronti otteniamo direttamente una stima del tempo di esecuzione, che sarà dello stesso ordine. In generale, una stima del tempo di calcolo potrà essere ottenuta moltiplicando il numero dei confronti, per il costo di ciascun confronto. Se ad esempio dovessimo ordinare delle stringhe di lunghezza al più $m$, ogni confronto tra stringhe costerà $O(m)$. Pertanto, una stima del tempo di calcolo potrà essere ottenuta moltiplicando il numero dei confronti per una funzione lineare in $m$.

Riguardo alla spazio, valuteremo quanto spazio *aggiuntivo* (cioè oltre a quello contenente inizialmente i dati da ordinare) venga utilizzato da un algoritmo. Per alcuni algoritmi tale spazio contiene, oltre a un numero fissato di variabili, delle strutture di ausiliarie la cui dimensione dipende dal numero $n$ di chiavi da ordinare (ad esempio, array ausiliari o stack per la ricorsione).

## Algoritmi di ordinamento interno basati su confronti

Esistono innumerevoli tecniche di ordinamento basate sui confronti tra chiavi. Ne vedremo alcune delle principali, suddivise in due gruppi.

- *Tecniche di base.*
  Si tratta di algoritmi elementari, di facile comprensione e implementazione, che ordinano effettuando, nel caso peggiore, un numero quadratico di confronti:
  - *Ordinamento per selezione* o `selectionSort`.
  - *Ordinamento per inserimento* o `insertionSort`.
  - *Ordinamento a "bolle"* o `bubbleSort`.
- *Tecniche avanzate.*
  Si tratta di algoritmi più sofisticati dei precedenti e meno intuitivi, che permettono di ottenere prestazioni migliori, con un numero di confronti dell'ordine di $n \log n$ nel caso peggiore (salvo `quickSort` per il quale, tuttavia, il caso peggiore risulta raro)
  - *Ordinamento per fusione* o `mergeSort`[1].
  - *Ordinamento veloce* o `quickSort`.
  - *Ordinamento mediante heap* o `heapSort`.

Nella descrizione di *tutti questi algoritmi*, ordineremo un array $A$ di $n$ elementi, i cui indici partono da 0. Alla fine dell'esecuzione, il risultato dell'ordinamento sarà lasciato nell'array $A$ stesso.

[1] Questo metodo, con alcuni adattamenti, può essere utilizzato anche per l'ordinamento esterno.

## 2.1 Algoritmi elementari

Gli algoritmi di ordinamento per selezione e per inserimento lavorano in maniera incrementale: ognuno di essi è costituito da una serie di passi principali:

- all'inizio di ciascun passo principale l'array contiene un segmento iniziale già ordinato, seguito da una parte da ordinare,
- al termine del passo il segmento iniziale contiene un elemento in più. In questo modo, dopo un numero finito di passi (come vedremo $n-1$), l'array risulta ordinato.

## 2.2 Ordinamento per selezione

Al passo principale $k$, $k = 0, \ldots, n-1$, viene selezionato l'elemento che deve essere collocato nella posizione $k$. L'elemento viene collocato in tale posizione, dalla quale l'elemento non sarà più spostato. Più in dettaglio:

1. *Prima del passo principale* $k$, i primi $k$ elementi dell'array sono al loro posto definitivo, cioè sono ordinati tra loro e minori o uguali degli elementi successivi, i.e., $A[0] \leq A[1] \leq \cdots \leq A[k-1]$ e $A[k-1] \leq A[j]$ per $j \geq k$;
2. si seleziona l'elemento che andrà collocato in posizione $k$, cioè il minimo della parte non ordinata (quindi il minimo tra $A[k], \ldots, A[n-1]$),
3. lo si colloca in posizione $k$, scambiandolo con l'elemento ivi presente,
4. in questo modo, *dopo il passo principale* $k$, i primi $k$ elementi risultano collocati nella loro posizione definitiva.

Si può facilmente osservare che dopo il passo $n-2$ la parte non ordinata contiene solo un elemento e, in base al punto 1, questo è maggiore o uguale dei precendenti e, dunque, si trova nella sua posizione definitiva. Pertanto non è necessario eseguire il passo $n-1$.

**Codice 2.1** Ordinamento per selezione

```
Algoritmo selectionSort (array A[0..n − 1])
 for k ← 0 to n − 2 do
    // ricerca del minimo in A[k..n − 1]
    m ← k                                   // m indica la posizione del minimo
    for j ← k + 1 to n − 1 do
       if A[j] < A[m] then m ← j
    scambia A[m] con A[k]   // sistema il minimo nella sua posizione definita k
```

### Numero di confronti

Consideriamo l'iterazione $k$ del ciclo principale. All'interno di essa viene ricercato il minimo della porzione di vettore da posizione $k$ a posizione $n-1$, effettuando $n-k-1$ confronti (numero iterazioni ciclo interno). Sommando su tutte le iterazioni $k$ del ciclo principale otteniamo il numero totale di confronti:

$$C(n) = \sum_{k=0}^{n-2} (n-k-1)$$

Osserviamo che nella sommatoria precedente stiamo sommando tutti i numeri da $n-1$ (per $k=0$) decrescendo fino a 1 (per $k=n-2$). Con il cambio di variabile $i=n-k-1$, possiamo riscrivere la sommatoria come:

$$C(n)=\sum_{k=0}^{n-2}(n-k-1)=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}=\Theta(n^2)\,.$$

Si noti che $\frac{n(n-1)}{2}$ è il *numero esatto di confronti*, che vengono sempre eseguiti indipendentemente dal contenuto dell'array.

### Spazio

L'algoritmo, oltre all'array da ordinare, utilizza un numero costante di variabili. Pertanto la quantità di spazio aggiuntivo è costante.

## 2.3 Ordinamento per inserimento

Prima del passo principale $k$, $k=1,\ldots,n-1$, gli elementi $A[0],\ldots,A[k-1]$ sono già ordinati tra di loro. Scopo del passo $k$ è quello di inserire l'elemento $A[k]$ nella posizione appropriata tra $A[0],\ldots,A[k-1]$. A tale scopo:

1. si memorizza l'elemento $A[k]$ da sistemare in una variabile $x$,
2. si ispeziona la porzione di array $A[0..k-1]$ *da destra verso sinistra*, cioè *a partire dalla posizione* $k-1$, spostando avanti di una posizione ogni elemento maggiore di $x$, in modo da "far posto" all'elemento da inserire;
3. individuata la posizione in cui inserire $x$ (quando si raggiunge un elemento che non è maggiore di $x$ o si è ispezionata tutta la porzione iniziale di array), si inserisce $x$ (gli elementi successivi sono già stati spostati durante il passo 2).

**Codice 2.2** Ordinamento per inserimento

```
Algoritmo insertionSort (array A[0..n − 1])
 for k ← 1 to n − 1 do
     x ← A[k]                                  // elemento da inserire in A[0..k − 1]
     // ricerca da destra la posizione in cui inserire x,
     // spostando man mano in avanti gli elementi maggiori
     j ← k − 1
     while j ≥ 0 and A[j] > x do
         A[j + 1] ← A[j]                       // sposta in avanti l'elemento A[j]
         j ← j − 1
     A[j + 1] ← x                                                  // inserisce x
```

### Numero di confronti

Consideriamo l'iterazione $k$ del ciclo principale. All'interno di essa il valore $x$ di $A[k]$ viene confrontato con *al più* tutti gli elementi precedenti da $A[k-1]$ fino ad $A[0]$. Pertanto, vengono effettuati al più $k$ confronti (ciclo `while` interno).

Sommando su tutte le iterazioni otteniamo che il numero di confronti è, nel caso peggiore,

$$C(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)\,.$$

Tale numero massimo di confronti viene effettuato quando l'array è inizialmente ordinato al contrario in quanto, ad ogni iterazione del ciclo principale, l'elemento $A[k]$ deve essere inserito nella posizione 0. D'altra parte, se inizialmente il vettore è già ordinato in modo non decrescente, per ogni iterazione del ciclo principale, il ciclo `while` interno termina immediatamente, dopo avere effettuato un solo confronto. In questo caso, che risulta il migliore per questo algoritmo, vengono effettuati $n-1$ confronti.

### Spazio

L'algoritmo, oltre all'array da ordinare, utilizza un numero costante di variabili. Pertanto la quantità di spazio aggiuntivo è costante.

### Note

- La ricerca della posizione in cui inserire l'elemento $A[k]$ inizia dalla posizione $k-1$ verso la posizione 0 e *non* dalla posizione 0 verso la posizione $k-1$. Cosa cambierebbe effettuando la ricerca a partire dalla posizione 0?

- Cosa cambia se nel ciclo `while` si sostituisce la condizione $A[j] > x$ con $A[j] \geq x$?

## 2.4 Ordinamento a bolle

L'idea base dell'algoritmo è quella di scandire ripetutamente l'array dal primo all'ultimo elemento, scambiando tra loro gli elementi adiacenti che non risultino ordinati. L'array sarà ordinato quando si riuscirà ad effettuare un'intera scansione senza alcuno scambio.

**Codice 2.3** Ordinamento "a bolle" (versione base)

```
Algoritmo bubbleSort (array A[0..n − 1])
 do
    scambiato ← false          // per ricordare se durante la scansione corrente
                               // è stato fatto almeno uno scambio
    for j ← 1 to n − 1 do
       if A[j] < A[j − 1] then
          scambia A[j − 1] con A[j]
          scambiato ← true
 while scambiato
```

*Esempio.*
Passi di ordinamento di un array (le righe successive alla prima indicano il contenuto dell'array dopo ogni iterazione del ciclo principale; gli elementi indicati in grassetto hanno raggiunto la posizione definitiva):

| *indici* | *0* | *1* | *2* | *3* | *4* | *5* | |
|---|---|---|---|---|---|---|---|
| contenuto iniziale | 7 | 2 | 4 | 5 | 3 | 1 | |
| dopo la 1a iterazione | 2 | 4 | 5 | 3 | 1 | **7** | |
| dopo la 2a iterazione | 2 | 4 | 3 | 1 | **5** | **7** | |
| dopo la 3a iterazione | 2 | 3 | 1 | **4** | **5** | **7** | |
| dopo la 4a iterazione | 2 | 1 | **3** | **4** | **5** | **7** | |
| dopo la 5a iterazione | **1** | **2** | **3** | **4** | **5** | **7** | |
| dopo la 6a iterazione | **1** | **2** | **3** | **4** | **5** | **7** | nessuno scambio effettuato! |

□

In generale, si può osservare che, come effetto della prima iterazione, l'elemento più grande raggiunge l'ultima posizione dell'array; come effetto della seconda scansione, il penultimo elemento nell'ordine raggiunge la penultima posizione, e così via (in alcuni casi, come per l'array `7 1 4 3 5 6`, tali posizioni possono essere raggiunte prima). In generale, dopo l'$i$-esima iterazione, gli ultimi $i$ elementi dell'array sono al loro posto definitivo e, dunque, *non è più necessario esaminarli*. Per la stessa ragione, dopo $n-1$ scansioni gli $n-1$ elementi più grandi hanno raggiunto la loro posizione finale. Dunque, l'elemento più piccolo deve trovarsi nell'unica posizione che resta, cioè quella di indice 0. Ciò implica che l'array è ordinato. Pertanto, dopo avere effettuato $n-1$ scansioni l'algoritmo può terminare anche se nell'ultima scansione ci sono stati scambi. Tenendo conto di queste considerazioni, possiamo scrivere una versione migliorata dell'algoritmo che effettui al più $n-1$ scansioni e che all'$i$-esima scansione non esamini gli ultimi $i-1$ elementi dell'array.

**Codice 2.4** Ordinamento "a bolle" (versione migliorata)

```
Algoritmo bubbleSort (array A[0..n − 1])
  i ← 1
  do
      scambiato ← false          // per ricordare se durante la scansione corrente
                                 // è stato fatto almeno uno scambio
      for j ← 1 to n − i do
          if A[j] < A[j − 1] then
              scambia A[j − 1] con A[j]
              scambiato ← true
      i ← i + 1
  while scambiato and i < n
```

Osserviamo che l'algoritmo potrebbe effettuare anche meno di $n-1$ iterazioni, come nel seguente esempio:

| *indici* | *0* | *1* | *2* | *3* | *4* | *5* | |
|---|---|---|---|---|---|---|---|
| contenuto iniziale | 7 | 1 | 4 | 3 | 5 | 6 | |
| dopo la 1a iterazione | 1 | 4 | 3 | **5** | **6** | **7** | |
| dopo la 2a iterazione | **1** | **3** | **4** | **5** | **6** | **7** | |
| dopo la 3a iterazione | **1** | **3** | **4** | **5** | **6** | **7** | nessuno scambio effettuato! |

In particolare, se l'array inizialmente è già ordinato, l'algoritmo effettua un'unica iterazione.

## Numero di confronti

Consideriamo l'iterazione $i$ del ciclo principale. In essa si effettuano esattamente $n-i$ confronti (ciclo `for` interno). Il ciclo principale viene eseguito a partire da $i=1$, incrementando $i$ fino al più

a $n-1$. Pertanto, sommando su tutte le iterazioni, il numero di confronti è al massimo

$$C(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)\,.$$

Tale numero viene raggiunto quando l'array è inizialmente ordinato al contrario. Se invece il vettore inizialmente è già ordinato, l'algoritmo termina dopo avere effettuato una sola scansione dell'intero array. In tal caso il numero di confronti è $n-1$.

### Spazio

L'algoritmo, oltre all'array da ordinare, utilizza un numero costante di variabili. Pertanto la quantità di spazio aggiuntivo è costante.

### Un ulteriore miglioramento

Un ulteriore miglioramento dell'algoritmo è stato proposto da Knuth in *The Art of Computer Programming, Volume 3* (seconda edizione, 1998, Addison Wesley Longman Publishing Co.), ed è presentato nel Codice 2.5. A differenza della versione precedente, in cui la parte da ispezionare ad ogni iterazione viene ridotta di un solo elemento, in questo caso la parte può essere ridotta di più elementi, sulla base degli spostamenti fatti nell'iterazione precedente, Questo permette di ridurre il numero di confronti in alcuni casi di array parzialmente ordinati (ad esempio `3 2 1 4 5 6 7`), ma non nel caso peggiore. La variabile *primoInOrdine* memorizza una posizione dell'array per la quale è noto che tutti elementi a partire da essa si trovano nella loro posizione definitiva. Alla fine di ogni iterazione del ciclo principale la variabile $t$ contiene l'indice del più grande elemento che è stato spostato nella propria posizione definitiva. Questo garantisce che tutti gli elementi dalla posizione $t$ in poi siano nella posizione definitiva, permettendo dunque di aggiornare *primoInOrdine* per l'iterazione successiva e, nel caso $t$ contenga 0, di stabilire che l'array è ordinato.

**Codice 2.5** Ordinamento "a bolle" (ulteriore miglioramento proposto da Knuth)

```
Algoritmo bubbleSort (array A[0..n − 1])
  primoInOrdine ← n
  do
      t ← 0
      for j ← 1 to primoInOrdine − 1 do
          if A[j] < A[j − 1] then
              scambia A[j − 1] con A[j]
              t ← j
      primoInOrdine ← t
  while t > 0
```

## 2.5 Confronti e spostamenti

Abbiamo detto che la stima del tempo di calcolo di questi algoritmi può avvenire a partire da quella del numero di confronti, moltiplicando il numero di confronti per il tempo necessario per effettuare ciascun confronto, Questo è vero a patto che i confronti tra chiavi siano le operazioni più costose effettuate dagli algoritmi. Potremmo calcolare anche il numero di spostamenti di elementi. Da questo calcolo possiamo scoprire che tra i tre algoritmi presentati sopra, l'ordinamento per selezione

è quello che effettua un numero di spostamenti più basso. Ma quanto costano gli spostamenti in termini di tempo? Come per i confronti, se siamo ordinando numeri interi di grandezza fissata, come i valori dei tipi `int` e `long`, gli spostamenti sono effettuati mediante assegnamenti che copiano un numero fissato di bit, e quindi avvengono in tempo costante. Se tuttavia, come avviene nella pratica, stiamo ordinando rispetto a un campo chiave dei record di grandi dimensioni, la copia di interi record diventa costosa in termini di tempo e il numero di spostamenti può essere un parametro critico, anche più importante del numero di confronti, per valutare il tempo impiegato da un algoritmo.

Questo problema può essere evitato utilizzando i puntatori: anziché memorizzare negli elementi dell'array i record da ordinare, possiamo memorizzare i puntatori ad essi. Quindi, ogni cella dell'array conterrà il puntatore a un record, memorizzato altrove. In questo modo, per effettuare uno spostamento, non è necessario copiare l'intero record, ma solo copiare dei puntatori ai record. La dimensione dei puntatori può essere considerata costante (dipende dalla grandezza delle memoria che può essere indirizzata).