4. Algoritmi di Ordinamento QuickSort

4.1 Ordinamento veloce (`quickSort`)

Presentiamo ora un algoritmo di ordinamento che, come `mergeSort`, utilizza uno schema ricorsivo, in cui ogni array di almeno due elementi viene diviso in due parti che vengono ordinate separatamete e poi ricombinate. Mentre in `mergeSort` la parte di suddivisione è immediata e quella di ricombinazione è più complicata, nell'algoritmo `quickSort`, che presentiamo ora, la parte di suddivisione risulta laboriosa al contrario di quella di ricombinazione che è banale.

Illustriamo l'idea base dell'algoritmo partendo da un esempio. Supponiamo di dovere ordinare la sequenza di numeri

```
44  55  12  42  94  6  18  67
```

Scegliamo all'interno di essa un *qualunque* elemento, ad esempio `42` (che chiameremo *perno* o *pivot*), e costruiamo due sequenze nelle quali collochiamo rispettivamente tutti gli elementi minori o uguali al perno e tutti quelli maggiori, in un qualunque ordine:

```
12  42  6  18          44  55  94  67
```

Ordinando separatamente le due sequenze otteniamo:

```
6  12  18  42          44  55  67  94
```

Concatenando le sequenze ottenute al passo precedente, otteniamo la sequenza ordinata:

```
6  12  18  42  44  55  67  94
```

Nel Codice 4.1 è schematizzata questa strategia. Studiamo ora più in dettaglio come realizzarla.

Iniziamo ad esaminare come costruire la *partizione*, cioè sulla suddivisione dell'array in due parti. Una soluzione molto semplice è quella di utilizzare delle strutture ausiliarie. In questo caso, dopo avere scelto il perno, si possono confrontare gli altri elementi con esso, collocandoli

Il presente materiale integra *ma non sostituisce* il libro di testo consigliato.
Data pubblicazione: 30 ottobre 2019

Codice 4.1 Quicksort: schema ad alto livello

Nello schema si utilizza impropriamente la notazioni insiemistica per semplicità e per enfatizzare il fatto che l'ordine con cui vengono collocati gli elementi in ciascuna delle due parti B e C non è importante.

Algoritmo *quickSort (array A)*
if *lunghezza di* $A > 1$ **then**
scegli un elemento di A come *perno*
$B \leftarrow \{y \in A \mid y \leq perno\}$
$C \leftarrow \{y \in A \mid y > perno\}$
$quickSort(B)$
$quickSort(C)$
$A \leftarrow$ concatenazione di B e C

nella parte appropriata. Il numero di confronti è $n - 1$. Tuttavia vogliamo evitare il ricorso a strutture ausiliarie, che richiedono ulteriore memoria. Vogliamo anche mantenere lineare il numero di confronti utilizzati per costruire la partizione. A tale scopo possiamo procede come segue:

1. Scegliamo come perno l'elemento più a sinistra dell'array.
2. Scansioniamo l'array da destra verso sinistra fino al primo elemento minore o uguale al perno.
3. Scansioniamo l'array da sinistra verso destra fino al primo elemento maggiore del perno.
4. Se due le scansioni non si sono incontrate, scambiamo i due elementi individuati e proseguiamo le scansioni ai passi 2 e 3.
5. Quando ogni elemento è stato confrontato con il perno (nella scansione da destra o in quella da sinistra), scambiamo il perno con l'elemento su cui si è arrestata la scansione da destra.

Per le scansioni da destra e da sinistra utilizziamo due indici, di nomi dx e sx, rispettivamente, che indicano gli elementi correntemente ispezionati dalle due scansioni. Osserviamo che, ad ogni passo, tutti gli elementi a sinistra dell'indice sx risultano minori o uguali al perno, mentre quelli a destra dell'indice dx risultano maggiori del perno. Quando i due indici si incontrano o si scambiano, cioè quando $dx \leq sx$, tutti gli elementi sono stati ispezionati. Inoltre, l'elemento di indice dx è minore o uguale al perno (passo 2). A questo punto è sufficiente scambiare questo elemento con il perno per ottenere la partizione.

Esempio.
Alcuni passi nella costruzione di una partizione:

contenuto iniziale	12	33	15	7	19	9	56	2	perno = 12
prima coppia da scambiare	*12*	33 $\uparrow_{sx}$	15	7	19	9	56	2 $\uparrow_{dx}$	
seconda coppia da scambiare	*12*	2	15 $\uparrow_{sx}$	7	19	9 $\uparrow_{dx}$	56	33	
gli indici si incontrano	*12*	2	9	7 $\uparrow^{\uparrow_{dx}}_{sx}$	19	15	56	33	
partizione	$\underbrace{7 \quad 2 \quad 9}_{\leq 12}$			*12*	$\underbrace{19 \quad 15 \quad 56 \quad 33}_{>12}$				

contenuto iniziale	25	7	4	33	2	27	55	12	24	42	perno = 25
prima coppia da scambiare	*25*	7	4	33 $\uparrow_{sx}$	2	27	55	12	24 $\uparrow_{dx}$	42	
seconda coppia da scambiare	*25*	7	4	24	2	27 $\uparrow_{sx}$	55	12 $\uparrow_{dx}$	33	42	
gli indici si sono incrociati	*25*	7	4	24	2	12 $\uparrow_{dx}$	55 $\uparrow_{sx}$	27	33	42	
partizione	$\underbrace{12 \quad 7 \quad 4 \quad 24 \quad 2}_{\leq 25}$					*25*	$\underbrace{55 \quad 27 \quad 33 \quad 42}_{>25}$				

□

Lo pseudocodice per la strategia di partizione appena descritta è riportato nel Codice 4.2. L'algoritmo effettua una partizione della porzione di array A delimitata dall'indice i (incluso) fino all'indice f (escluso), *restituendo la posizione finale del perno.*

Codice 4.2 Partizionamento di un array

```
Algoritmo partiziona (array A, indice i, indice f)→ indice
  /* Riorganizza gli elementi all'interno di A[i..f − 1] e restituisce un
     indice j in modo tale che tutti gli elementi di A[i..j − 1] siano minori o
     uguali ad A[j] e tutti gli elementi di A[j + 1..f − 1] siano maggiori
     di A[j]                                                              */
  perno ← A[i]
  sx ← i
  dx ← f
  while sx < dx do
      do dx ← dx − 1 while A[dx] > perno
      do sx ← sx + 1 while sx < dx and A[sx] ≤ perno
      if sx < dx then scambia A[sx] con A[dx]
  scambia A[i] con A[dx]
  return dx
```

Si osservi che, dopo avere effettuato la partizione, il perno ha raggiunto la sua posizione definitiva nell'array, cioè la stessa posizione che avrà nell'array ordinato. Pertanto l'algoritmo `quickSort` può limitarsi a riordinare ricorsivamente la parte di array a sinistra del perno e quella a destra del perno, senza più considerare il perno. Una volta riordinate queste due parti, l'intero array risulterà ordinato.

Lo pseudocodice di `quickSort` è riportato nel Codice 4.3. La procedura ricorsiva `quickSort` ordina la porzione di array dall'indice i all'indice f escluso. L'algoritmo principale si limita a richiamare la procedura ricorsiva sull'intero array.

Numero di confronti

Calcoliamo ora il numero di confronti effettuati da `quickSort` per ordinare un array di n elementi. Osserviamo, prima di tutto, che per effettuare la partizione ogni elemento dell'array deve essere confrontato con il perno (eccetto il perno stesso). Pertanto vi sono almeno $n-1$ confronti. Inoltre, in alcuni casi al termine del processo di costruzione della partizione l'elemento su cui si trova l'indice dx viene confrontato un'ulteriore volta con il perno (si veda il secondo esempio precedente).

Codice 4.3 Quicksort

Algoritmo *quickSort (array $A[0..n-1]$)*
 $quickSort(A, 0, n)$

Procedura *quickSort (array A, indice i, indice f)* `/* Ordina` $A[i..f-1]$ `*/`
 if $f - i > 1$ **then**
  $m \leftarrow partiziona(A, i, f)$
  $quickSort(A, i, m)$
  $quickSort(A, m+1, f)$

Pertanto, il numero di confronti per la costruzione della partizione è $n-1$ o n (nei calcoli successivi, per semplicità utilizzeremo sempre n, scelta che non modifica le stime ottenute).

Caso peggiore

Calcoliamo ora in numero di confronti di `quickSort` nel *caso peggiore*, che indichiamo con $\mathcal{C}_w(n)$. Possiamo scrivere la seguente equazione di ricorrenza:

$$\mathcal{C}_w(n) = \begin{cases} n + \max\{\mathcal{C}_w(k) + \mathcal{C}_w(n-k-1) \mid 0 \leq k < n\} & \text{se } n > 1 \\ 0 & \text{altrimenti.} \end{cases} \tag{4.1}$$

Nell'espressione per il caso $n > 1$, il termine n rappresenta il numero di confronti per effettuare la partizione, mentre ciascuna somma $\mathcal{C}_w(k) + \mathcal{C}_w(n-k-1)$ rappresenta il numero di confronti effettuati nelle chiamate ricorsive nell'ipotesi che, dopo la partizione, vi siano k elementi a sinistra del perno e $n-k-1$ a destra.[1] Poiché stiamo studiando il caso peggiore, consideriamo il valore di k che massimizza tale somma. È possibile dimostrare che tale valore si ottiene quando la partizione è completamente sbilanciata, cioè $k = 0$ o $k = n-1$ (ad esempio, se come perno viene scelto il primo elemento, nel caso l'array sia già ordinato si ottiene $k = 0$). Otteniamo dunque che, per $n > 1$, nel caso peggiore si ha $\mathcal{C}_w(n) = n + \mathcal{C}_w(n-1)$ da cui, procedendo per sostituzione sino ad arrivare al caso base, ricaviamo:

$$\mathcal{C}_w(n) = n + \mathcal{C}_w(n-1) = n + n - 1 + \mathcal{C}_w(n-2) = \cdots = \sum_{i=2}^{n} i + \mathcal{C}_w(1) = \frac{n(n+1)}{2} - 1 = \Theta(n^2).$$

Pertanto nel caso peggiore `quickSort` effettua un numero di confronti dello stesso ordine degli algoritmi elementari che abbiamo analizzato.

Caso migliore

Abbiamo detto che il caso peggiore si ottiene quando ad ogni livello della ricorsione la partizione risulta sbilanciata. Se, al contrario, l'array viene sempre suddiviso in due parti circa della stessa lunghezza (come avviene per `mergeSort`), il numero di confronti diminuisce drasticamente. In tal caso infatti possiamo approssimare il numero di confronti con la seguente equazione di ricorrenza che, come quella di `mergeSort`, ha soluzione $\mathcal{C}_b(n) = \Theta(n \log n)$:

$$\mathcal{C}_b(n) = \begin{cases} n + 2\mathcal{C}_b(n/2) & \text{se } n > 1 \\ 0 & \text{altrimenti.} \end{cases} \tag{4.2}$$

Pertanto nel caso migliore l'algoritmo si comporta come `mergeSort`. Ad esempio, se come perno viene scelto il primo elemento, la seguente sequenza rappresenta un caso migliore di `quickSort`:

`8 5 2 7 10 9 11`

[1] In altri termini, considerando indici a partire da 0, ciò significa che il perno è stato collocato nella posizione di indice k.

Caso medio

Perché questo algoritmo si chiama `quickSort` *(quick= veloce)* se nel caso peggiore si comporta come gli algoritmi di ordinamento elementari? La risposta si ha studiando il *numero medio* di confronti effettuati dall'algoritmo.

Abbiamo visto che nel caso base ($n = 0$ e $n = 1$) l'algoritmo non effettua confronti. Pertanto $\mathcal{C}(0) = \mathcal{C}(1) = 0$. Se $n > 1$ vi sono n confronti per effettuare la partizione, più $\mathcal{C}(k)$ confronti per ordinare ricorsivamente la parte sinistra e $\mathcal{C}(n-k-1)$ confronti per ordinare ricorsivamente la parte destra, dove k è la lunghezza della parte sinistra. Il valore di k dipende dal contenuto dell'array. Assumendo una distribuzione uniforme, supponiamo che la probabilità che il perno si trovi alla posizione k, sia $1/n$, per $k = 0, \ldots, n-1$. Pertanto, possiamo calcolare il numero medio di confronti con la seguente formula

$$\begin{aligned}
\mathcal{C}(n) &= \frac{1}{n}\sum_{k=0}^{n-1}(n + \mathcal{C}(k) + \mathcal{C}(n-k-1)) \\
&= \frac{1}{n}\sum_{k=0}^{n-1} n + \frac{1}{n}\sum_{k=0}^{n-1}\mathcal{C}(k) + \frac{1}{n}\sum_{k=0}^{n-1}\mathcal{C}(n-k-1) \\
&= n + \frac{2}{n}\sum_{i=0}^{n-1}\mathcal{C}(i)
\end{aligned}$$

Abbiamo pertanto la seguente equazione di ricorrenza per il numero medio di confronti:

$$\mathcal{C}(n) = \begin{cases} n + \frac{2}{n}\sum_{i=0}^{n-1}\mathcal{C}(i) & \text{se } n > 1 \\ 0 & \text{altrimenti} \end{cases} \tag{4.3}$$

Dimostriamo per induzione su n che $\mathcal{C}(n) \leq 2n \ln n$, per $n \geq 1$. La base $n = 1$ è ovvia. Per il passo induttivo, poiché $\mathcal{C}(0) = \mathcal{C}(1) = 0$ e utilizzando l'ipotesi di induzione otteniamo:

$$\begin{aligned}
\mathcal{C}(n) &= n + \frac{2}{n}\sum_{i=0}^{n-1}\mathcal{C}(i) = n + \frac{2}{n}\sum_{i=2}^{n-1}\mathcal{C}(i) \\
&\leq n + \frac{2}{n}\sum_{i=2}^{n-1} 2i\ln i = n + \frac{4}{n}\sum_{i=2}^{n-1} i\ln i
\end{aligned}$$

Il valore della sommatoria può essere calcolato mediante la seguente maggiorazione per mezzo di un integrale, calcolabile per parti:

$$\begin{aligned}
\sum_{i=2}^{n-1} i\ln i &\leq \int_2^n x\ln x dx = \left[\frac{x^2\ln x}{2} - \frac{x^2}{4}\right]_2^n \\
&= \frac{n^2\ln n}{2} - \frac{n^2}{4} - 2\ln n + 1 \leq \frac{n^2\ln n}{2} - \frac{n^2}{4}
\end{aligned}$$

Sostituendo, concludiamo la dimostrazione per induzione ottenendo:

$$\mathcal{C}(n) \leq n + \frac{4}{n}\left(\frac{n^2\ln n}{2} - \frac{n^2}{4}\right) = 2n\ln n$$

Nella formula precedente è presente il logaritmo naturale. Convertendo alla base 2 otteniamo che il numero medio di confronti è $\mathcal{C}(n) \leq 1.39 n \log_2 n$. Questo numero è estremamente basso, molto

vicino al numero di confronti di `mergeSort`, per il quale avevamo calcolato al più $n \log_2 n - n + 1$ confronti per n potenza di 2.

È importante osservare che, poiché nel caso peggiore il numero di confronti è dell'ordine di n^2, mentre sia nel caso medio che nel caso migliore tale numero è dell'ordine di $n \log n$, il caso peggiore è *estremamente raro*. Per questa ragione e per il fatto che la costante 1.39 è molto bassa, questo algoritmo viene utilizzato molto spesso nella pratica, ed è classificato tra gli algoritmi veloci.

Ulteriori osservazioni

Possiamo osservare che le prestazioni di `quickSort`, su uno stesso array, possono variare notevolmente in base alla strategia utilizzata per scegliere il perno. Ad esempio, con la scelta che abbiamo utilizzato, se l'array inizialmente è già ordinato si ottiene il caso peggiore. Al contrario, se scegliessimo come perno l'elemento centrale, nel caso di un array già ordinato otterremmo il caso migliore.

Spesso, per cercare di evitare il caso peggiore, si utilizzano delle strategie randomizzate. Una possibilità è quella di disordinare in maniera casuale gli elementi dell'array, prima di applicare l'algoritmo. Un'altra possibilità può essere quella di scegliere come perno un elemento a caso dell'array da ordinare (lo si può scambiare con il primo elemento, applicando poi la strategia di partizione che abbiamo presentato).

Si può inoltre osservare che questo metodo di ordinamento non è stabile (per esercizio trovate un esempio che giustifichi questa affermazione). Esistono delle versioni stabili di `quickSort` che asintoticamente hanno le stesse prestazioni, ma con constanti più alte (D. Motzkin: A Stable Quicksort. *Softw., Pract. Exper.* 11(6): 607-611 (1981).)

Spazio di lavoro

L'algoritmo `quickSort` è un algoritmo *in loco*, nel senso che non vengono utilizzate strutture aggiuntive destinate a contenere i dati da ordinare come, al contrario, avviene per `mergeSort`, in cui si utilizza un array di supporto per la fase di `merge`. Tuttavia `quickSort` utilizza memoria aggiuntiva per la ricorsione. Possiamo osservare che ogni record di attivazione deve contenere i parametri i e f che delimitano la parte di array da ordinare, oltre alla variabile m. Dunque la grandezza di ciascun record di attivazione è costante. Pertanto, la quantità di memoria utilizzata sullo stack sarà proporzionale al numero massimo di record di attivazione che possono essere presenti contemporaneamente, cioè all'altezza massima raggiunta dallo stack.

Nel caso peggiore, tale altezza è n. Ciò avviene quando ad ogni chiamata ricorsiva il perno è l'elemento maggiore e dunque tutti gli altri elementi vengono collocati nella parte sinistra della partizione. Un esempio in cui avviene ciò, con la nostra scelta del perno, è quando la prima posizione contiene l'elemento maggiore dell'array, seguito da tutti gli altri in ordine crescente, come

```
8  1  2  3  4  5  6  7
```

Si osservi che in questo caso viene anche effettuato un numero massimo di confronti dell'ordine di n^2. Anche nel caso di array già ordinato, come

```
1  2  3  4  5  6  7  8
```

si ha un numero di confronti dell'ordine di n^2 e altezza massima dello stack n. In generale, quando l'array viene sempre diviso in maniera sbilanciata si ottiene sia il caso peggiore del numero di confronti, sia il caso peggiore per l'altezza dello stack.

Viceversa, se la partizione risulta sempre bilanciata, cioè l'algoritmo viene applicato ricorsivamente a due parti della stessa grandezza, oltre ad avere un numero di confronti dell'ordine di $n \log n$, anche l'altezza dello stack risulta dell'ordine di $\log n$ (possiamo mostrarlo nello stesso modo utilizzato per `mergeSort`).

È possibile modificare `quickSort`, in maniera che l'altezza dello stack sia sempre $O(\log n)$. Per fare ciò iniziamo ad osservare che nella procedura ricorsiva nel Codice 4.3 la seconda chiamata ricorsiva si trova in coda. Anziché effettuare una chiamata ricorsiva, possiamo ripetere dall'inizio (sostituendo la selezione con un'iterazione), dopo avere aggiornato la variabile i (Codice 4.4). Applicando questa nuova versione dell'algoritmo al caso di array già ordinato, che utilizzava stack di altezza n, ogni chiamata ricorsiva per la parte sinistra (che contiene 0 elementi) termina immediatamente, senza effettuare ulteriori chiamate. Quindi l'altezza dello stack è costante. Tuttavia, nell'altro caso presentato, in cui la parte destra contiene 0 elementi, l'altezza dello stack resta n anche in questa versione.

Codice 4.4 Procedura ricorsiva di Quicksort senza ricorsione in coda

```
Procedura quickSort (array A, indice i, indice f)        /* Ordina A[i..f − 1] */
  while f − i > 1 do
      m ← partiziona(A, i, f)
      quickSort(A, i, m)
      i ← m + 1
```

Possiamo anche osservare che nel `quickSort` non è importante l'ordine con cui vengono ordinate le due parti (cioè l'ordine delle chiamate ricorsive nel Codice 4.3): se ordinassimo la parte destra prima di quella sinistra il risultato resterebbe corretto. Combinando questa idea, con l'eliminazione della ricorsione in coda presentata nel Codice 4.4, scriviamo una nuova versione dell'algoritmo nella quale ordiniamo prima la parte più piccola, utilizzando la ricorsione, e poi l'altra parte ripetendo iterativamente (Codice 4.5).

Codice 4.5 Procedura ricorsiva di Quicksort con riduzione dell'altezza dello stack

Dopo il partizionamento, ordina subito la parte più piccola ottenuta (chiamata ricorsiva), lasciando "in sospeso" (per la successiva iterazione del ciclo) la parte più grande. In tal modo, ogni record di attivazione sullo stack rappresenta una chiamata per ordinare un array di lunghezza al più metà rispetto a quella dell'array del record del chiamante. Di conseguenza l'altezza dello stack è al massimo dell'ordine di $\log n$.

```
Procedura quickSort (array A, indice i, indice f)        /* Ordina A[i..f − 1] */
  while f − i > 1 do
      m ← partiziona(A, i, f)
      if m − i < f − m then
          quickSort(A, i, m)
          i ← m + 1
      else
          quickSort(A, m + 1, f)
          f ← m
```

In questo modo, se dobbiamo ordinare un array di lunghezza n, la chiamata ricorsiva verrà effettuata per l'array più corto della partizione, che quindi ha lunghezza al massimo $\frac{n}{2}$. Pertanto l'altezza dello stack è:

$$\mathcal{H}(n) \leq \begin{cases} \mathcal{H}(n/2) + 1 & \text{se } n > 1 \\ 1 & \text{altrimenti,} \end{cases} \tag{4.4}$$

da cui otteniamo $\mathcal{H}(n) = O(\log n)$. Tale limite superiore viene raggiunto quando, ad ogni livello della ricorsione, le due parti hanno circa la stessa lunghezza. Pertanto possiamo concludere che $\mathcal{H}(n) = \Theta(\log n)$.

Poiché, come discusso in precedenza, ogni record di attivazione ha dimensione costante, lo spazio utilizzato per implementare questa versione di `quickSort` è $\Theta(\log n)$.