QuickSort

Segue il paradigma "divide et impera".

Decomposizione: se la sequenza ha almeno due elementi, scegli un elemento pivot e dividi la sequenza in due sotto-sequenze (eventualmente vuote) dove

- la prima contiene elementi minori o uguali al pivot
- la seconda contiene elementi maggiori o uguali al pivot

Ricorsione: ordina ricorsivamente le due sotto-sequenze.

Ricombinazione: concatena (implicitamente) le due sotto-sequenze ordinate in un'unica sequenza ordinata.

Indichiamo al solito con Scambia (i, j) la funzione che $a[i] \leftrightarrow a[j]$

```
Scambia (i, j):
temp = a[i];
a[i] = a[j];
a[j] = temp;
```

dove a indica l'array contenente gli n elementi da ordinare

In codice:

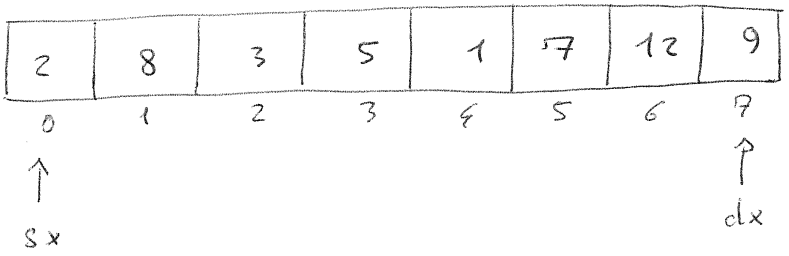
```
                                  ← array da ordinare
QuickSort (a, sinistra, destra):
                                  0 ≤ sinistra , destra ≤ n-1.

IF (sinistra < destra){
   scegli pivot nell'intervallo [sinistra ... destra];
   rango = Distribuzione (a, sinistra, pivot, destra);
   QuickSort (a, sinistra, rango - 1);
   QuickSort (a, rango+1, destra);
}
```

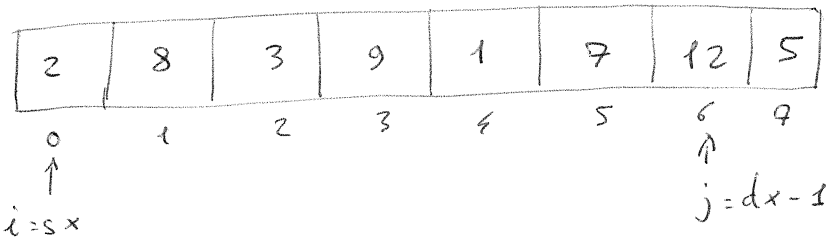
dove Distribuzione è invece:

```
Distribuzione (A, sx, px, dx)
IF (px != dx) Scambia (px, dx);          /* mette il pivot nell'ultima cella di A
i = sx;   j = dx - 1;
  WHILE (i ≤ j){                         /* si esce quando i > j
     WHILE ((i ≤ j) && (A[i] ≤ A[dx]))   /* avanza i fino a A[i]>A[dx]
        i = i + 1;
     WHILE ((i ≤ j) && (A[j] ≥ A[dx]))   /* avanza j fino a A[j]<A[dx]
        j = j - 1;
     IF (i < j) Scambia (i, j);          /* scambia per far avanzare ancora i e j
  }
  IF (i != dx) Scambia (i, dx);          /* inserisce il pivot al "posto giusto" nella sequenza ordinata.
  RETURN i;
```

Esempio di esecuzione di Partizione

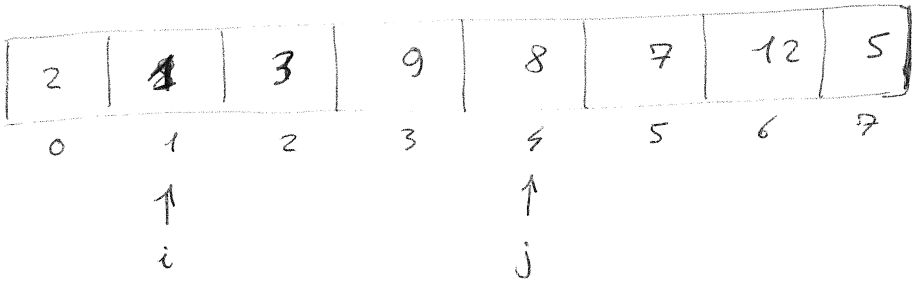


Supponiamo che $px = 3$. Allora Scambia (3, 7) produce:

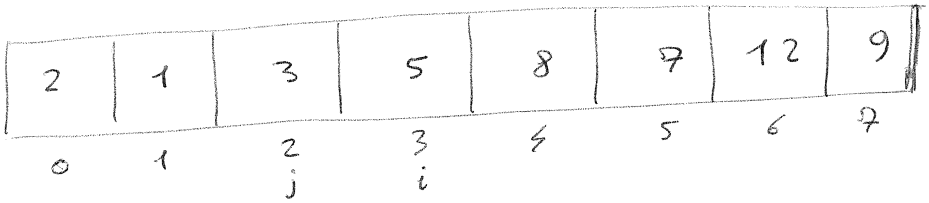


1ª ITERAZIONE

All'interno del WHILE principale, il primo WHILE sposta l'indice i fino a $i = 1$. Il secondo WHILE sposta l'indice j fino a $j = 4$. A[1] e A[4] vengono scambiati



Alla seconda iterazione, i diventa $i = 3$ e l'indice j avanza fino a $j = 2$ (si esce dal WHILE principale). A[3] e A[7] vengono scambiati.



Viene restituito $i = 3$;

Osservazioni:

La posizione i restituita da Distribuzione è la posizione finale nell'array ordinato in cui si troverà il pivot scelto.

Distribuzione essenzialmente divide l'array in due metà:

- nella metà a sinistra mette tutti gli elementi uguali o minori del pivot
- in quella destra tutti gli elementi maggiori o uguali al pivot

Il valore $z = i + 1$ rappresenta il rango del pivot.

- i: indice restituito da Distribuzione
- z: # elementi minori o uguali al pivot

Tecnicamente, Distribuzione fa avanzare gli indici i e j, il primo da sinistra a destra, il secondo da destra a sinistra, fino a quando $i > j$.

L'indice i avanza fino a quando trova un elemento maggiore del pivot.

L'indice j avanza fino a quando trova un elemento minore del pivot.

Lo scambio degli elementi $A[i] \leftrightarrow A[j]$ permette

ai due indici di riprendere la scansione dell'array fino a quando non si incrociano.

Il costo della funzione Distribuzione è chiaramente $O(n)$ (viene scandito l'intero vettore).

Si noti pure che l'algoritmo QuickSort ha un costo di ricombinazione nullo: una volta che le due metà, quella sinistra e quella destra, sono state ricorsivamente ordinate, non c'è nulla da fare ancora.

Il tempo di esecuzione di QuickSort può essere espresso attraverso una relazione di ricorrenza. Chiaramente, nel caso base (array di taglia 1) $T(n) \leq c_0$ per $n \leq 1$.

Nel passo ricorsivo, se indichiamo con r = rango + 1, le due chiamate (1° passo della ricorsione)

QuickSort(a, sinistra, r-2) → a[0, ... r-2] → (r-1) elementi

QuickSort(a, r, destra) → a[r, ..., n-1] → (n-r) elementi

possiamo scrivere che

$$T(n) \leq T(r-1) + T(n-r) + c \cdot n$$

($c \cdot n$: costo di Distribuzione)

Naturalmente, il valore di q dipende dal pivot che scegliamo. Pertanto non è noto a priori.

Tuttavia, volendo valutare il caso pessimo, è facile osservare che le condizioni più sfavorevoli si verificano quando:

$$q = 1 \quad \text{(pivot a sinistra)} \quad \text{o} \quad q = n \text{ (pivot a destra)}$$

In entrambi i casi:

$$T(n) \leq T(n-1) + T(0) + cn \leq T(n-1) + c'n.$$

Applicando il metodo iterativo (- il Master Theorem non è applicabile) è facile vedere che $T(n) = O(n^2)$.

Infatti:

$$\begin{aligned} T(n) &\leq T(n-1) + c' \cdot n \\ &\leq T(n-2) + c'(n-1) + c'n \\ &\leq T(n-3) + c'(n-2) + c'(n-1) + c'(n \\ &\vdots \\ &\leq T(\underbrace{n-(n-1)}_{1}) + c'(n-(n-2)) + \dots + c'(n-1) + c'n \\ &\leq \underbrace{c' \sum_{k=2}^{n} k}_{c' \cdot \left(\frac{n \cdot (n+1)}{2} - 1\right)} + \underbrace{T(1)}_{c_0} = O(n^2) \end{aligned}$$

Osservazioni: il libro di testo, per semplicità, in prima istanza suggerisce di scegliere il pivot automaticamente come l'ultimo elemento a[dx] dell'array a.

Con tale scelta, quando l'array in input a è già ordinato, si verifica il caso peggiore.

Se, invece, la distribuzione fosse sempre perfettamente bilanciata, cioè $s = n/2$, la relazione di ricorrenza diventerebbe, al passo ricorsivo

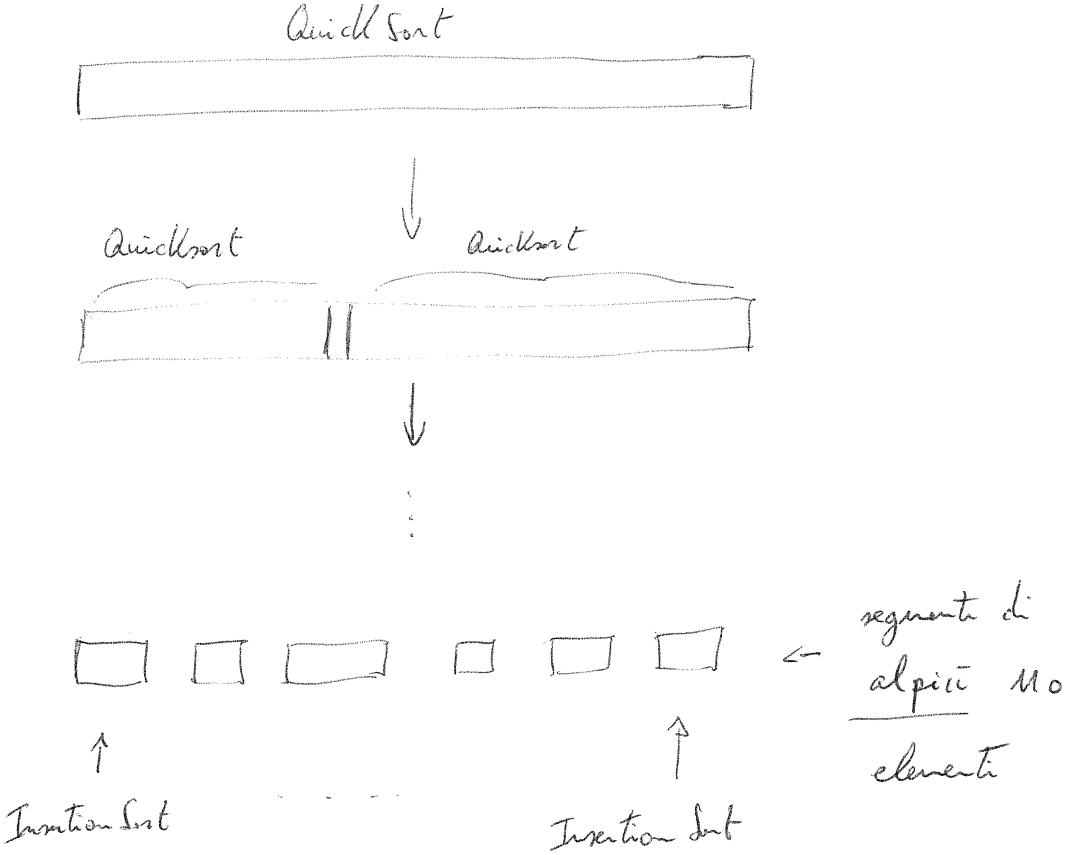
$$T(n) \leq 2\,T(n/2) + c \cdot n$$

che, per il caso 2. del master theorem ha soluzione $O(n \log n)$.

È possibile mostrare (ma NON lo faremo) che nel caso medio l'algoritmo richiede tempo $O(n \log n)$ perché si alternano distribuzioni bilanciate con distribuzioni sbilanciate (e queste ultime sono in qualche modo "ammortizzate" da quelle bilanciate).

Come il nome suggerisce, l'algoritmo è veloce in pratica.

Spesso nelle implementazioni se ne usa una versione ibrida.



Invece di applicare la ricorsione fino alla fine, quando l'array contiene un numero costante n_0 di elementi viene ordinato direttamente usando l'InsertionSort.

Il costo di questo passo è n_0^2 per ogni segmento.

Supponendo siano tutti di taglia n_0, allora sono in totale $\frac{n}{n_0}$. Pertanto il costo di questa fase è

$$\frac{n}{n_0} \times n_0^2 = n \times n_0 = O(n)$$

Selezione dell'elemento di rango r

Dato un array a di n elementi occorre individuare l'elemento che nell'array ordinato occuperebbe la posizione r-1 (gli array indicizzano le posizioni a partire da zero...).

senza ordinare l'array.

Un elemento ha rango r se ci sono r elementi minori o uguali in a.

Se $r = 1$, l'elemento richiesto è il minimo in a.

Se $r = n$, l'elemento richiesto è il massimo in a.

Il codice che implementa l'algoritmo QuickSort può essere facilmente modificato per individuare l'elemento di rango r, per qualsiasi valore di r, $1 \le r \le n$.

QuickSelect (a, sinistra, z, destra):

↘ array di n elementi

$0 \leq sinistra \leq z-1 \leq destra \leq n-1$

↓

```
IF (sinistra == destra) {
    RETURN a[sinistra];
} ELSE { ->
    scegli pivot nell'intervallo [sinistra ... destra];
    rango = Distribuzione (a, sinistra, pivot, destra);
    IF (z-1 == rango) {
        RETURN a[rango];
    } ELSE IF (z-1 < rango) {
        RETURN QuickSelect (a, sinistra, z, rango-1);
    } ELSE {
        RETURN QuickSelect (a, rango+1, z, destra);
    }
-> }
```

Esempio di esecuzione.

Per semplicità consideriamo l'array a utilizzato in precedenza

a

2	8	3	5	1	7	12	9
0	1	2	3	4	5	6	7

e supponiamo di voler individuare l'elemento di rango $2 = 2$.

QuickSelect (a, 0, 2, 7) invoca Distribuzione (a, 0, 3, 7)

che restituisce il valore rango = 3 (... sto utilizzando l'esecuzione già vista in cui px = 3)

e il vettore modificato risulta

a

2	1	3	5	8	7	12	9
0	1	2	3	4	5	6	7

Poiché $2-1 = 2-1 = 1 < 3 = \text{rango}$, viene invocata la chiamata ricorsiva QuickSelect (a, 0, 2, 2) sull'array

a

2	1	3
0	1	2

sx dx

Supponendo che il pivot scelto sia px = 2, Distribuzione (a, 0, 2, 2) opera come segue:

2	1	3
1°	1'	2

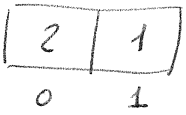
i = sx j = dx-1 dx

l'indice i avanza fino a i = 2.

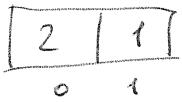
l'indice j resta j = 1

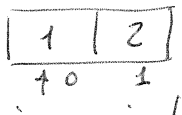
Restituisce rango = 2

Poiché $2-1 = 1 < 2 = \text{rango}$, l'algoritmo invoca

QuickSelect $(a, 0, 2, 1)$ sull'array $a =$ 

Supponendo di scegliere come pivot $px = 0$, Distribuzione scambia gli elementi e procede:

$a =$ 

$\longrightarrow$ 

$i = sx = j = dx - 1$

L'indice i avanza fino a $i = 1$.

L'indice j resta $j = 0$.

Restituisce $\text{rango} = 1$

Poiché $(2-1 = 1 \equiv 1 = \text{rango})$

QuickSelect restituisce $a[\text{rango}] = a[1] = 2$

Nell'array considerato 2 è l'elemento di rango 2.

Osservazione: se il pivot scelto quando $a =$ 

fosse stato $px = 1$ e, a seguito della successiva invocazione di QuickSelect $(a, 1, 2, 2)$, si scegliesse $px = 1$ per Distribuzione $(a, 1, 1, 2)$, l'algoritmo terminerebbe nell'altro caso (array a di taglia 1). Provate!

L'idea alla base della trasformazione del codice di QuickSort per ottenere QuickSelect è che, ad ogni invocazione,

> Distribuzione trova il rango del pivot
>
> Posiziona gli elementi di rango minore a sinistra, quelli di rango maggiore a destra.

Pertanto possiamo modificare QuickSort per selezionare l'elemento di interesse, procedendo ricorsivamente soltanto "sulla metà" di interesse dell'array a.

Non lo mostriamo formalmente ma, con ragionamenti analoghi a quelli che si applicano all'analisi dell'algoritmo QuickSort nel caso medio, è possibile far vedere che l'algoritmo QuickSelect, nel caso medio, costa $O(n)$.

Nota: QuickSelect, nel cercare l'elemento di rango 2, modifica lo stato iniziale dell'array a.

Diversi elementi possono essere spostati.