

Algoritmi e Strutture Dati per la Fisica dei Dati

Quicksort

Analisi e confronto con altri algoritmi di ordinamento

Edoardo Tronconi

Matricola: 974734

23 febbraio 2021

Problema del Sorting

Problema del Sorting

Sorting

Input Una sequenza di n oggetti $\{a_1, a_2, \dots, a_n\}$

Output Una permutazione $\{a_{\sigma_1}, a_{\sigma_2}, \dots, a_{\sigma_n}\}$ tale che $a_{\sigma_1} \leq a_{\sigma_2} \leq \dots \leq a_{\sigma_n}$

È un problema fondamentale in quanto:

- Molti problemi necessitano intrinsecamente di qualche forma di ordinamento
- Molti algoritmi hanno l'ordinamento come subroutine fondamentale
- Può essere approcciato con numerose tecniche diverse

Proprietà degli algoritmi di sorting

Complessità temporale: Il tempo utilizzato dall'algoritmo al variare della dimensione dell'input.
Esiste un lower bound teorico di $\Omega(\log n!) = \Omega(n \log n)$ al numero di confronti necessari

Spazio utilizzato: Lo spazio in memoria utilizzato. Algoritmi che occupano spazio $O(1)$ sono detti *in place*

Stabilità: Un algoritmo di sorting è detto *stabile* se due elementi con valore (key) uguale compaiono nello stesso ordine nell'input e nell'output

Quicksort

Quicksort è un algoritmo sviluppato nel 1961 da Tony Hoare.

- Basato sull'approccio *Divide et Impera*
- Complessità temporale $O(n^2)$ nel caso peggiore e $O(n \log n)$ nel caso medio
- Algoritmo *in place*
- Non è stabile

Algoritmo

Divide: La procedura PARTITION suddivide *in place* l'array di input $A[p, \dots, r]$ in due subarray $A[p, \dots, q - 1]$ e $A[q + 1, \dots, r]$

- I subarray sono tali che $A[p, \dots, q - 1] \leq A[q] \leq A[q + 1, \dots, r]$
- L'indice q è calcolato durante la procedura

Impera: I subarray sono ordinati applicando ricorsivamente Quicksort.
Il caso “base” della ricorsione partiziona l'array di due elementi $A[p, r]$ in $\{A[p'], A[r']\}$ con $A[p'] \leq A[r']$

Combina: Poiché i subarray sono già ordinati tra di loro, non serve alcuna procedura per ricombinare i risultati parziali: l'array $A[p, \dots, r]$ è ordinato

Pseudocode

Partition: PARTITION(A, p, r)

```
1   $x = A[r]$ 
2   $i = p$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5          exchange  $A[i]$  with  $A[j]$ 
6           $i = i + 1$ 
7  exchange  $A[i]$  with  $A[r]$ 
8  return  $i$ 
```

Quicksort: QUICKSORT(A, p, r)

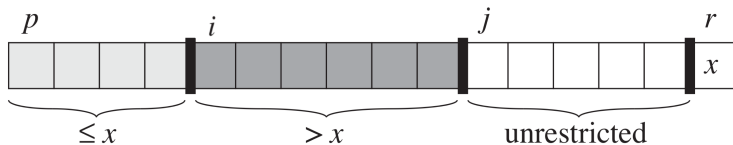
```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )
```

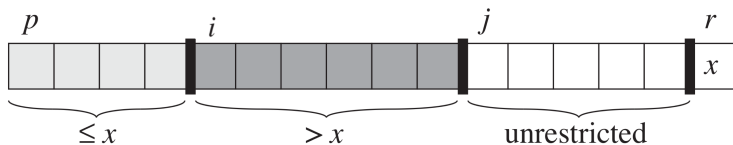

Correttezza

Per dimostrare la correttezza dell'algoritmo si utilizza un *Loop Invariant*:

All'inizio di ogni iterazione del ciclo for in PARTITION:

- ① $p \leq k \leq i - 1 \Rightarrow A[k] \leq x$
- ② $i \leq k \leq j - 1 \Rightarrow A[k] > x$
- ③ $k = r \Rightarrow A[k] = x$

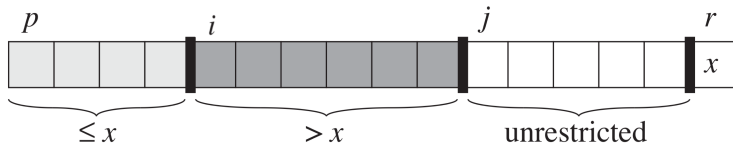




Inizializzazione: $i = j = p \Rightarrow$ ① e ② sono soddisfatte in modo banale.
La prima riga di PARTITION assicura che ③ sia soddisfatta

$j \Rightarrow j + 1$: A seconda del risultato del test su $A[j]$ si ottiene:

- $A[j] > x \rightarrow j = j + 1$.
 - ① rimane vera
 - ② è vera per $A[j - 1]$ appena aggiunto (altri uguali)
- $A[j] \leq x \rightarrow$ scambiati $A[i] \leftrightarrow A[j]$; $i = i + 1$; $j = j + 1$.
 - ① e ② sono vere in quanto ho scambiato il valore $\leq x$ con uno $> x$



Termine: $j = r \Rightarrow$ Ho suddiviso l'intero array nelle tre regioni identificate dalle condizioni ①, ② e ③

L'ultima riga di PARTITION scambia $A[i]$ e $A[r]$



L'output corrisponde a quello atteso: $A[p, \dots, i-1] \leq A[i] \leq A[i+1, r]$

Complessità Temporale PARTITION

Tempo impiegato da PARTITION

La procedura partition su un subarray $A[p, \dots, r]$ ha complessità temporale $\Theta(n)$ (con $n = r - p$)

- Le righe di codice all'esterno del ciclo **for** sono eseguite in tempo $\Theta(1)$
- Le righe di codice all'interno del ciclo **for** sono eseguite in tempo $\Theta(1)$ e il ciclo è eseguito n volte

Pertanto:

$$T_{\text{PARTITION}}(n) = \Theta(1) + n \cdot \Theta(1) = \Theta(n)$$

Complessità Temporale QUICKSORT

La complessità temporale di QUICKSORT dipende da come PARTITION divide l'array

$$T_{\text{QUICKSORT}}(n) = T_{\text{QUICKSORT}}(q) + T_{\text{QUICKSORT}}(n - q - 1) + \Theta(n)$$

Complessità Temporale QUICKSORT

Best Case

Se PARTITION produce sottoproblemi bilanciati l'equazione di ricorrenza per QUICKSORT diventa:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$

\Downarrow

$$T(n) = \Theta(n \log n)$$

La complessità in tempo di QUICKSORT pertanto è asintoticamente ottimale nel caso migliore.

Complessità Temporale QUICKSORT

Worst Case

Se PARTITION produce sottoproblemi completamente sbilanciati (ovvero di dimensione $[n - 1]$ e $[0]$) l'equazione di ricorrenza diventa:

$$T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n)$$

\Downarrow

$$T(n) = \Theta(n^2)$$

Si può dimostrare che si tratta del caso peggiore considerando:

$$T(n) = \max_{0 \leq q \leq n-1} [T(q) + T(n - q - 1)] + \Theta(n)$$

e provando come soluzione $T(n) \leq c \cdot n^2$, da cui si ottiene:

$$\Rightarrow T(n) \leq c \cdot \max_{0 \leq q \leq n-1} [q^2 + (n - q - 1)^2] + \Theta(n) \leq c \cdot n^2 + \Theta(n) = O(n^2)$$

Complessità Temporale QUICKSORT

Worst Case

La procedura PARTITION mostrata produce sottoproblemi sbilanciati quando il pivot selezionato è il minimo o il massimo dell'array.

Il caso peggiore si ha se i sottoproblemi sono sbilanciati ad ogni iterazione ricorsiva di QUICKSORT.

Ma quindi i “casi peggiori” sono:

Array già ordinato \Rightarrow Scelta migliore del pivot (RANDOMIZED-PARTITION¹, MEDIAN-OF-THREE, MEDIAN-OF-N²)

Array con molti valori uguali \Rightarrow HOARE-PARTITION o THREE-WAY-PARTITION

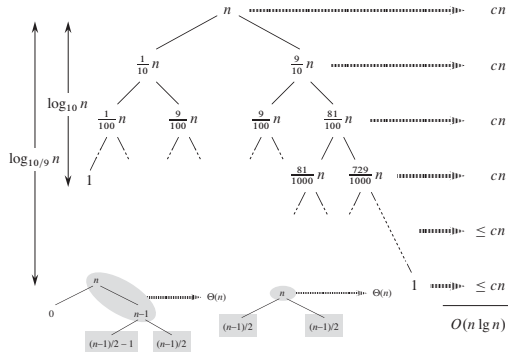
¹QUICKSORT con RANDOMIZED-PARTITION impiega comunque tempo $\Theta(n^2)$ con probabilità $\frac{1}{n!}$, ma non sempre su array ordinati

²Garantisce $\Theta(n \log n)$ ma con costanti sufficientemente elevate da renderlo non pratico

Complessità Temporale QUICKSORT

Average Case

Il tempo medio di esecuzione di QUICKSORT è molto più vicino al caso migliore che al caso peggiore



- Se PARTITION divide in modo arbitrariamente sbilanciato, ma con proporzionalità costante $\Rightarrow O(n \log n)$
- Anche se alcuni *splits* sono completamente sbilanciati ($[n-1] \longleftrightarrow [0]$) $\Rightarrow O(n \log n)$
- Il tempo è $O(n^2)$ solo se PARTITION divide sempre in modo "lineare" $[n-c-1] \longleftrightarrow [c]$

Complessità Temporale QUICKSORT

Average Case

Per studiare la complessità temporale del caso medio di QUICKSORT si suppone che:

- 1 Tutte le permutazioni possibili di ogni sequenza di input siano ugualmente probabili
In pratica, per evitare di avere il *worst case* su un caso specifico (come array già ordinati) → RANDOMIZED-QUICKSORT: scelgo il pivot casualmente
- 2 Non ci siano elementi con valore uguale¹

¹Per adeguate scelte di PARTITION la complessità in tempo in presenza di elementi con valore uguale è \leq di quella con elementi tutti diversi

Complessità Temporale QUICKSORT

Average Case

$T_{\text{QUICKSORT}}(n)$ è dominato dal tempo speso in PARTITION

Il pivot scelto da PARTITION non è incluso nelle successive chiamate a QUICKSORT
 \Rightarrow PARTITION è eseguita al più n volte

Il tempo impiegato dalla i -esima chiamata a PARTITION è $O(1) + O(\#_{\text{for}}^i)$

Il tempo impiegato da tutte le n chiamate a partition è $O(n + X)$ ($X = \sum_{i=1}^n \#_{\text{for}}^i = \#_{\text{confronti}}^{\text{tot}}$)

$$T_{\text{QUICKSORT}}(n) = O(n + X)$$

PARTITION(A, p, r)

```
1   $x = A[r]$ 
2   $i = p$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5          exchange  $A[i]$  with  $A[j]$ 
6           $i = i + 1$ 
7  exchange  $A[i]$  with  $A[r]$ 
8  return  $i$ 
```

QUICKSORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )
```

Complessità Temporale QUICKSORT

Average Case

Definisco $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$, dove $\{z_\ell\}_{\ell=1}^n$ sono gli elementi dell'array $A[1, \dots, n]$ disposti in ordine crescente

Si ottiene il numero totale di confronti tra due elementi dell'array (X) con le considerazioni:

- 1 Due elementi sono confrontati al più una volta
- 2 Utilizzando la variabile indicatrice $X_{ij} = I\{z_i \text{ è confrontato a } z_j\}$ il numero totale di confronti è dato da:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}$$

- 3 Il valore atteso del numero di confronti è dato da:

$$E[X] = E \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr\{z_i \text{ è confrontato a } z_j\}$$

Complessità Temporale QUICKSORT

Average Case

La probabilità di confrontare due valori z_i e z_j è calcolata considerando che:

- 1 Se un pivot x è scelto con $z_i < x < z_j \Rightarrow z_i$ e z_j **non** sono confrontati
- 2 z_i e z_j sono confrontati se uno dei due è scelto come pivot prima di ogni altro elemento in Z_{ij}

Ma tutte le permutazioni sono ugualmente probabili, quindi:

$$\Pr\{z_\ell \text{ è il primo pivot scelto da } Z_{ij}\} = \frac{1}{|Z_{ij}|} = \frac{1}{j-i+1}$$



$$\Pr\{z_i \text{ è confrontato a } z_j\} = \Pr\{z_i \text{ o } z_j \text{ sono il primo pivot scelto da } Z_{ij}\} = \frac{2}{j-i+1}$$

Complessità Temporale QUICKSORT

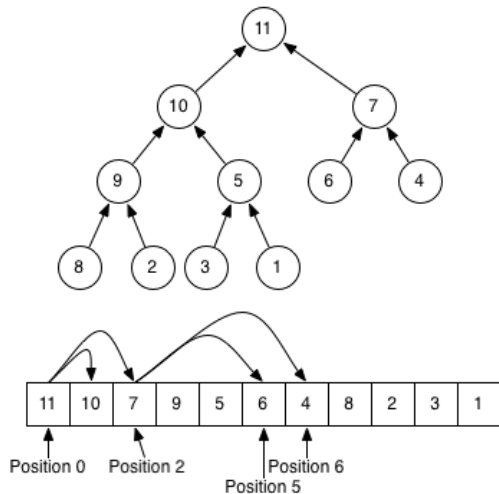
Average Case

È possibile quindi calcolare il valore atteso del numero di confronti:

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} \\ &= \sum_{i=1}^{n-1} O(\log n) \\ &= O(n \log n) \end{aligned} \quad \Rightarrow \quad \begin{aligned} T_{\text{QUICKSORT}}(n) &= O(n + X) = O(n + n \log n) \\ &= O(n \log n) \end{aligned}$$

Heapsort

Heapsort



- Basa il suo funzionamento su un *max heap*
 - Complessità temporale $\Theta(n \log n)$
 - In place
 - Non stabile
-
- Non ha worst case $O(n^2)$ come Quicksort, ma è generalmente più lento