

COMPLESSITÀ QUICKSORT

vediamo anche ie caso medio
senza calcolare

CASO MEDIO $\sim \Theta(n \log n)$ = MERGESORT
CASO MIGLIORATO \sim

MARTINA
POLINELLI

Idea base di quickSort: partizionare rispetto a un perno (“pivot”)

(in place)

- Prendi un elemento dell'array (il “pivot”)
- Partiziona gli elementi dell'array in modo tale che
 - quelli a sinistra del pivot siano minori del pivot
 - quelli a destra siano maggiori (o uguali) del pivot
- Dopo queste operazioni, il pivot è nella sua posizione finale
- Richiama quickSort sulle due parti dell'array ottenute

che non sono
per forza
metà e metà

contenuto
di una
cella

Due fatti importanti della partizione

- Richiede tempo lineare nella dimensione dell'array, $\Theta(n)$.
- Non richiede spazio extra
- Riduce la dimensione del problema

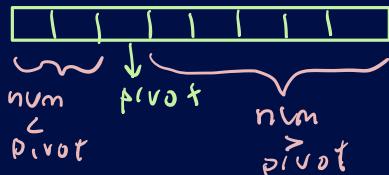
Quicksort, descrizione ad alto livello

[C.A.R. Hoare, 1961]

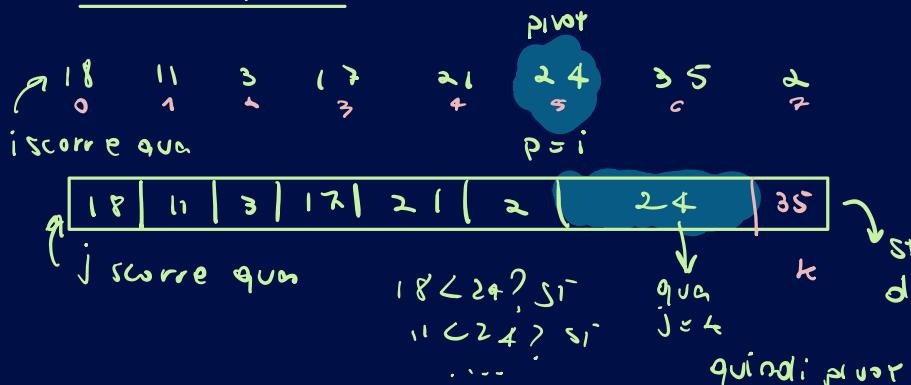
```
QuickSort(array A)
{
    if lunghezza(A)=1 return;
    p=scegliPivot(A);
    Partiona A rispetto al pivot p (metti a sinistra di p gli elementi < e a destra gli elementi >);
    Riordina ricorsivamente la prima parte di A (gli elementi minori di p);
    Riordina ricorsivamente la seconda parte di A (gli elementi maggiori di p);
}
```

idea Quicksort → 2 chiamate ricorsive QS (su 2 parti di array)

partizione(in place) →



* NON IN PLACE



N.B. → MOLTO importante
scelta del pivot
(adesso sceglie a caso)

Ogni n confronto con pivot → complessità $\Theta(n)$

* in place

Assumo che 1° elem della seq che va da "inizio" e "fine" è pivot



Come partizionare (modo "difficile", senza array di appoggio)

$p = \text{pivot}$
(esem)

$l = \text{index}$
del
pivot

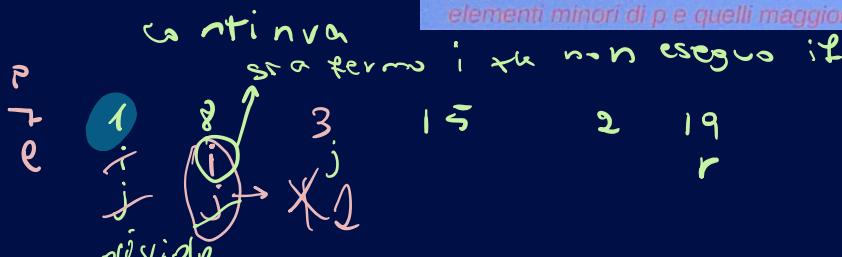
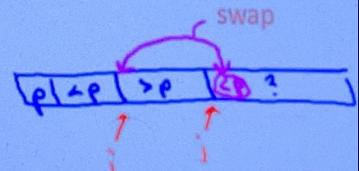


Partition (A, l, r)

[input corresponds to $A[l \dots r]$]

- $p := A[l]$
- $i := l+1$
- for $j = l+1$ to r
 - if $A[j] < p$ [if $A[j] > p$, do nothing]
 - swap $A[j]$ and $A[i]$
 - $i := i+1$
- swap $A[l]$ and $A[i-1]$

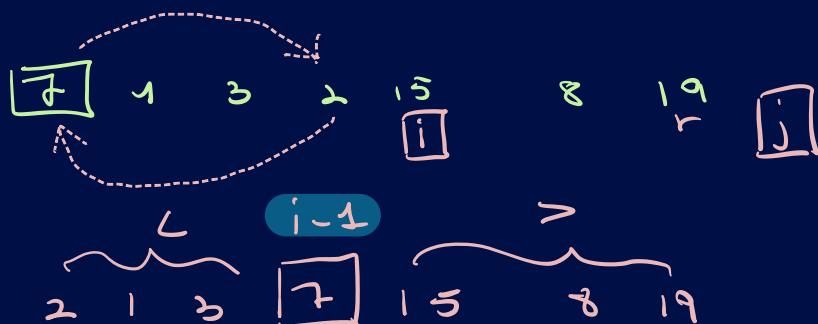
Questa è una descrizione ad alto livello e trascura alcuni dettagli implementativi che nell'implementazione bisogna assolutamente considerare: in particolare, partition deve restituire l'indice del pivot p per poter effettuare le due chiamate ricorsive sugli elementi minori di p e quelli maggiori o uguali



$j =$ tra elementi scenditi e ancora da scandire
 $i =$ "muro" / tra quelli scenditi / tra + piccoli e + grandi del pivot

$A[j] < p ?$ si swap tra $A[j]$ e $A[i]$

ultimo swap \rightarrow pivot swap in $i-1$



* $i < x < j$
elementi > pivot

* $x < i$
elementi < pivot

* Δ

In questo sto dicendo che I numeri prima di J sono quelli analizzati e anche quelli prima di I sono più piccoli di P

come si sceglie il pivot? + ramite mediana

in questo modo chiamiamo quicksort su 2 parti uguali

EX CASO PEGGIORI

QS 12 3 8 ≤ 21 n elem

* se sceglio 3...

[3] 5 12 21 8 n operaz

QS 5 12 21 8

5 12 8 21 n-1 operazioni

QS 5 12 8

5 8 12 n-2 operaz

etc ... 1 operaz

caso peggiore: $\Theta \sum_{i=1}^n i = \Theta n^2$

caso in cui sceglio

pivot a caso e

mi capita sempre

o ie + piccolo o ie + grande

* scegliamo come pivot sempre l'elem mediano

① QS n

		P
--	--	---

partition

QS $\frac{n}{2}$

	P
--	---

 → QS

--

②

QS $\frac{n}{2}$

P

 → QS

--

FINE

LIVELLO

0

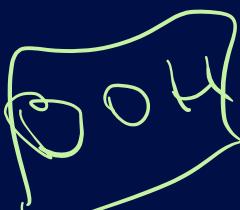
$\frac{n}{2}$
 $n/5$

1

$\frac{n}{2}$

2

$\frac{n}{4}$



caso migliore =
 $= \Theta n \log n$

(solito discorso
per mergesort)

Come scegliere un buon pivot?

- IDEA: scelta casuale (random) del pivot

ovvero, ad ogni chiamata ricorsiva, scegliamo come pivot uno dei numeri nell'array, a caso (la probabilità di essere scelto è la stessa per ogni elemento).

Con questo approccio alla scelta del pivot, abbiamo una versione di quickSort "randomizzata": è il primo esempio di algoritmo randomizzato, nel quale due esecuzioni diverse sullo stesso input possono svolgersi in modo diverso! Naturalmente, il risultato finale delle due esecuzioni deve essere lo stesso!



Non possiamo scegliere il mediano Perché per definizione il mediano possiamo trovarlo in una lista di numeri ordinata, ma dato che noi stiamo ordinando questo non è possibile.

Infatti ricadiamo in complessità $N \log N$ anche se non dividiamo perfettamente a metà

- Per ogni array di dimensione n , il tempo di esecuzione di quickSort randomizzato nel caso medio è $\Theta(n \log n)$
[non lo dimostriamo]

Miglioramento: scegliere 3 elem random, ordinare e ie mediano tra quelli \rightarrow pivot

Altro miglioramento: se sequenza molto piccola
Chiamo insertion sort invece che quick

Numeri casuali in C++

- La funzione `rand()` genera un numero compreso nell'intervallo `[0, RAND_MAX]`, dove `RAND_MAX` è una costante che dipende dal compilatore usato.
- Il generatore di numeri pseudo casuali produce una sequenza di numeri apparentemente casuali; in realtà la sequenza ad un certo punto si ripete e risulta prevedibile
- Per ovviare a questo inconveniente è necessario far generare la sequenza partendo ad ogni esecuzione con un valore diverso (un seme diverso).
- La funzione `srand` inizializza il generatore di numeri con un valore che passiamo come argomento
- La funzione `time(NULL)` restituisce un valore intero ottenuto dal clock pari al numero di secondi trascorsi dal 1/1/1970



QuickSort, verso il codice

```
void quickSort(array A) /* A è un array di interi */  
{  
    qs(A, 0, size(A)-1); /* chiamo qs, la funzione che  
    definiremo ricorsivamente, su tutto l'array A,  
    ovvero la parte di array identificata dagli estremi 0  
    e size(A)-1 */  
}
```

QuickSort, verso il codice

```
void qs(array A, int inizio, int fine)
{
    if (condizione per cui non ho ancora raggiunto il caso base)
    {
        int pivot_index = partizionalInPlace(A, inizio, fine); /* la funzione
partizionalInPlace fa due cose: 1) modifica la parte di A compresa tra inizio e
fine, indici inclusi, effettuando la partizione "in place"; per fare questo deve
selezionare il pivot_index, mettere l'elemento in posizione pivot_index -- ovvero
il pivot -- all'inizio della sottosequenza compresa tra inizio e fine, implementare il
partizionamento in place, rimettere a posto il pivot; 2) restituisce l'indice del pivot
(non il pivot ma il suo indice!!!); questo è necessario perché dobbiamo
richiamare qs sulle due sottoparti di A comprese rispettivamente tra inizio e
pivot_index -1, e tra pivot_index+1 e fine */
        richiamo ricorsivamente qs tra inizio e pivot_index-1
        richiamo ricorsivamente qs tra pivot_index+1 e fine
    }
}
```

All'esame:

per sapere i
passaggi ricevi
MERGESORT

- * complessità caso migliore (dettagliato) (O(n log n))
che è un caso ipotetico in cui pivot = elem mediani
- * NO analisi caso medio (che comunque è quello randomico)
- * spiegare che i sotto problemi sono di tipo partition
- * profondità dell'albero delle chiamate ricursive
- * complessità caso peggiore (Ti basta punti se dici che è quello in cui le pivot è ie primo numero! QUELLO È SOLO UN CASO DI CASO PEGGIORER)
bisogna invece dire che è quello in cui le pivot è sempre o num + piccole o + grande. ($\Theta(n^2)$)