

Come funziona il metodo partition Quicksort

Consideriamo l'array

{ 7, 2, 1, 8, 6, 3, 5, 4 }

L'idea generale è quella di avere un pivot, e tutti gli elementi prima del pivot sono minori del pivot, e tutti gli el dopo sono più grandi.

Ci sono diversi modi per effettuare il partitioning, ma vediamo il più semplice:

- Dato un array non ordinato  $a[low : high]$ , scegliamo come pivot l'ultimo elemento:

int pivot = arr[high]

Di conseguenza, tenendo conto dell'array riportato prima, il nostro pivot è 4.

Abbiamo due contatori:

- I Iniziamo impostando  $i$  come: int  $i = low - 1$

ovvero:  $i = low - 1$

- j Iniziamo j da  $low$  fino ad arrivare ad  $high$   
ovvero: for (int  $j = low$ ;  $j < high$ ;  $j++$ )

Ci troviamo quindi nella situazione:

$j \leftarrow j = low$

$i \uparrow 7, 2, 1, 8, \dots$

$i = low - 1$

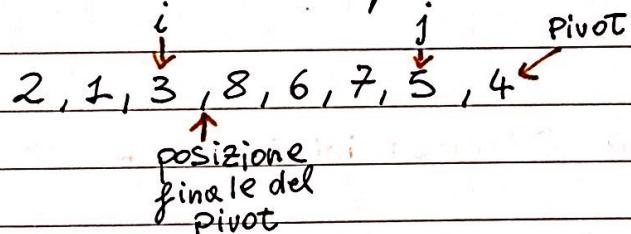
Una volta inizializzate le variabili... prima cosa da fare (nel loop, quindi si rigete) è effettuare un confronto tra  $j$  e pivot.

Se  $j$  è più grande del pivot, allora non facciamo nulla, ed incrementiamo  $j$  (sempre nel loop).

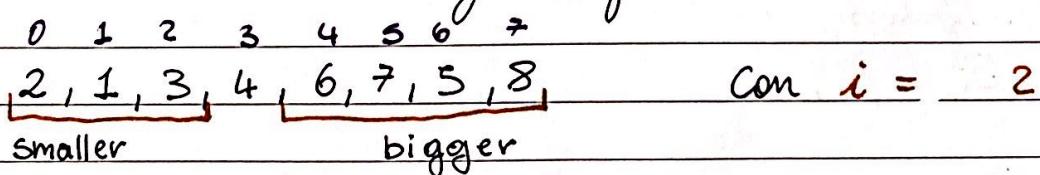
Quando incontriamo un elemento  $arr[j]$  minore del pivot, incrementiamo  $i$  e scambiamo  $i$  e  $j$ .

Continuiamo quindi incrementando  $j$  ed effettuando il confronto con il pivot.

Continuiamo in questo modo finché  $j$  non arriva alla fine. Arrivati alla fine, scambiato l'ultimo elemento (ovvero il pivot) con l'indice  $i+1$ . Questo perché ci troveremo in questa situazione:



Effettuando questo scambio, avremo che tutti gli elementi a sinistra del pivot sono più piccoli, e tutti quelli a destra sono più grandi.



L'ultima cosa da fare è quella di tornare  $i+1$ , ovvero la posizione del pivot, in modo da partizionare l'array allo suo sx e allo suo dx con lo stesso algoritmo.

## QuickSort Recap

Come il MergeSort, il QuickSort è un algoritmo ricorsivo, ma quando parliamo di QS. dobbiamo tenere a mente la parola **Pivot**.

Un Pivot è semplicemente un elemento nell'array che soddisfa le seguenti condizioni:

- È nella posizione finale dell'array ordinato.
- Gli elementi a sinistra sono più grandi.
- Gli elementi a destra sono più piccoli.

$\{2, 6, 5, \underline{3}, 8, 7, 1, 0\}$

Pivot

1) Scambiamo il pivot con l'ultimo el dell'array.

2) Ora cerchiamo due elementi:  
• itemFromLeft che è più grande del pivot  
• itemFromRight che è il primo elemento dalla dx che è più piccolo del pivot.

$2, 6, 5, 0, 8, 7, 1, 3$

item<sup>↑</sup>FromLeft

item<sup>↑</sup>From Right

3) Scombiemo i due valori 0 e 1, e ripetiamo il processo.

• Ci fermiamo quando itemFromLeft ha un indice maggiore di iFR.

4) Scombiemo iFL con il nostro pivot che si trova in ultima posizione.

→ A questo punto, 3, è nella sua posizione finale.

$\{2, 1, 0, 3, 8, 7, 6, 5\}$  5 è l'indice del pivot

Abbiamo ora che tutti gli elementi a Sx del nostro pivot sono più piccoli, e tutti gli el a Dx sono più grandi.

Abbiamo detto che il QS è un algoritmo ricorsivo, quindi non ci resta altro che partizionare l'array e lasciare che la ricorsione faccia il resto.

Alla fine avremo che le due partizioni Sx e Dx sono completamente ordinate.

Come scegliere il Pivot?

Nel caso migliore, vogliamo che il nostro pivot divida l'array a metà, o il più vicino possibile.

Median of three

In questo metodo guardiamo il primo, ultimo ed elemento centrale, li ordiniamo e scegliamo l'el centrale come pivot.

Worst case :  $O(N^2)$

Average Case :  $O(N \log N)$

## Quick SORT

Il cuore dell'algoritmo è costituito dal partizionamento.  
Nel primo step si effettua lo shuffle dell'array per protezione contro il caso peggiore.

Successivamente si effettua una partizione dell'array, in modo tale che per un dato  $j$  (avendo una data posizione dell'array) il valore di  $a[j]$  risulta ordinato, ovvero in place, cioè non abbiamo chiavi che siano maggiori della chiave scelta, ovvero dell'elemento pivot.

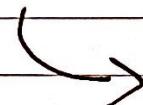
Quindi NON avremo chiavi maggiori a sinistra e NON avremo chiavi minori a dx.

In questo modo sono ordinare in modo ricorsivo la parte sinistra e destra senza preoccuparmi dell'elemento pivot che risulterà ad ogni passo in place, ovvero nel suo posto finale.

### Come Funziona il Partizionamento?

Effettuo il loop finché i due puntatori  $i$  e  $j$  non si incontrano, quindi faccio lo scan da destra sinistra a destra finché non trovo un elemento minore dell'elemento pivot.

In modo duale faccio lo scan da destra a sinistra usando il puntatore  $j$  finché non trovo un elemento maggiore dell'elemento pivot; a quel punto faccio lo scambio tra  $a[i]$  e  $a[j]$ .



Quando i due partitatori si incontrano, effettua lo scambio tra il pivot ( $\alpha[lo]$ ) ed  $\alpha[sj]$ .

Una volta partizionato, mi ritrovo che  $k$  (il vecchio pivot) si trova nella sua posizione finale, e sono andare ad ordinare ricorsivamente, con lo stesso metodo precedente, il sotto array sinistro e destro.

In altre parole...

Tutti gli elementi maggiori del pivot li devo portare a destra, tutti gli elementi minori li devo portare a sinistra.

## client example

Challenge: Vogliamo Trovare gli  $m$ -items più grandi in uno Stream di  $n$  elementi, dove lo Stream è molto grande e  $m$  è grande.  
Questo potrebbe essere un metodo di Fraud detection, dove andiamo ad isolare le Transazioni che hanno un costo / importo maggiore.

Il vincolo in questo caso è che abbiamo una memoria limitata e non possiamo effettuare l'ordinamento di TUTTI gli  $n$ -elementi per Trovare il max.

## La soluzione

Possiamo usare una coole e priorità minima in cui leggiamo da sinistra le Transazioni e creo una nuova Transaction, inserendola nella coole. Ogni volta che leggo la Transaz. effettuo il controllo: se il size della coole è maggiore di  $m$ , vedo ogni volta a rimuovere il minimo. ( $m$  è il numero di item che voglio Trovare). Alla fine Troverò all'interno delle coole le Transaz. che hanno un importo maggiore.

(1:06 - 3/30)

## COLLEZIONI

Una collezione è un insieme di oggetti, precisamente è un tipo di dato che raggruppa oggetti; finora abbiamo visto lo stack e la coda, ed ora introduciamo la:

### CODA A PRIORITÀ

Questo tipo di coda serve principalmente ad avere un ordinamento parziale degli elementi.

Come rimuoviamo gli elementi nelle varie collezioni?

Stack: Rimuoviamo l'elemento aggiunto per ultimo.

Queue: Rimuoviamo <sup>il primo</sup> elemento aggiunto.

Randomized Queue: Rimuoviamo un elemento random.

Cose vogliamo rimuovere in una Priority Queue?

L'obiettivo è rimuovere l'elemento più grande/piccolo.

### API

Public class MaxPQ < Key extends Comparable<Key>

- MaxPQ() Creo una coda vuota
- void insert(Key a) inserire una chiave
- Key delMax() ritornare e rimuovere il più grande
- boolean isEmpty() ritorna se la collez. è vuota
- Key max() ritorna la chiave più grande.
- int size() ritorna il numero degli elementi

### Applicazioni

- Grafi
- Statistica
- Event simulation
- Sistemi operativi

Come si implementa una coda a priorità?  
Se utilizzassi delle code standard avrei ciclare negli elementi ogni volta che voglio trovare il massimo.

Challenge: Il nostro obiettivo è implementare tutte le operazioni in modo efficiente:

IMPLEMENTAZIONE	INSERT	del Max	max
Array non ordinato	1	n	n
Array ordinato	n	1	1
goal	log n	log n	log n

Soluzione: Array parzialmente ordinati con una struttura dati particolare: Binary Heap

Albero binario

Def: è vuoto oppure un nodo che è collegato a destra e sinistra ad un sottoalbero.

Albero completo: È perfettamente bilanciato, tranne per l'ultimo livello (base).

↳ L'altezza dell'albero cresce a potenze di 2; quindi l'h di un albero completo con n nodi è  $\lg n$ .  
ES:  $n = 15 \rightarrow h = 3$ ;  $n = 16 \rightarrow h = 4$

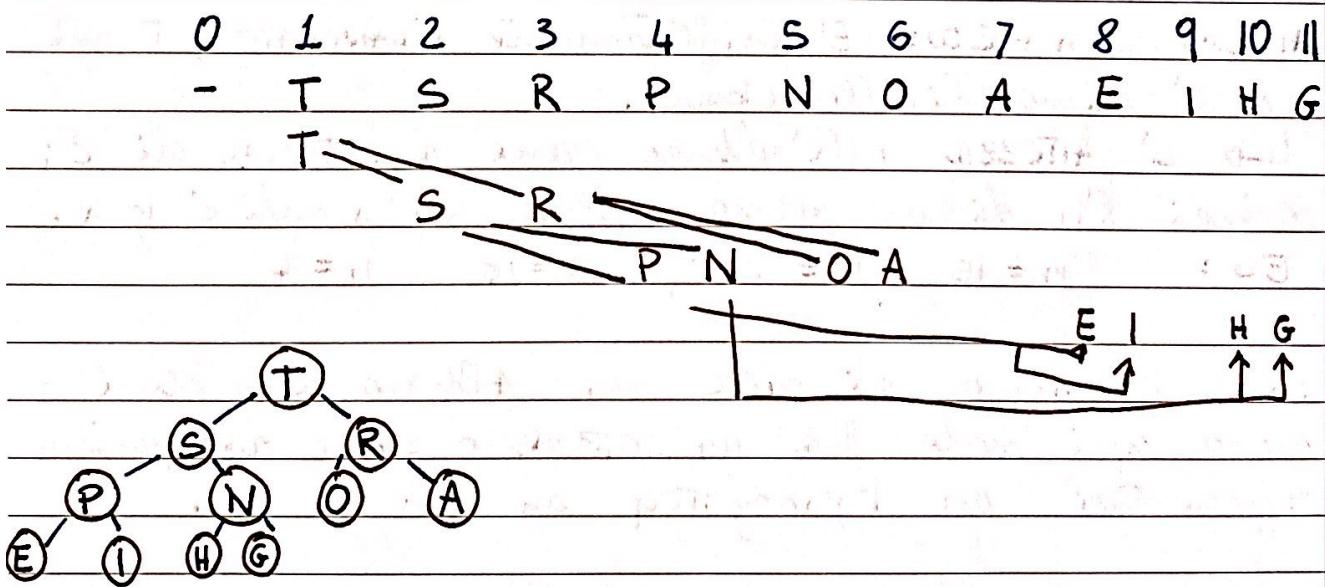
Se eliminiamo di avere un albero completo (avendo ogni nodo ha un sottoalbero sx e dx) posiamo rappresentare un Binary Heap con un Array.

Rappresentare un Binary Heap

Come detto in precedenza assumiamo che l'albero sia completo, poniamo usare un array per rappresentare un binary heap.

Nell'array avremo quindi le chiavi dei nodi seguendo una regola precisa: l'albero deve essere heap ordered, ovvero deve avere le chiavi in ordine, quindi non deve esistere una chiave GENITORE più piccola di una chiave FIGLIO.

Per rappresentare con un array facciamo partire gli indici da 1 (per una motivazione matematica); i nodi figlio del nodo  $k$  saranno agli indici  $2k$  e  $2k+1$ . In questo modo non avremo bisogno di tenere traccia dei links!



## PROPRIETA'

Proposizione: la chiave piu' grande si trova all'indice  $a[1]$ , che e' la radice dell'albero binario.

Quindi sappiamo sempre dove si trova l'elemento piu' grande.

Proposizione: Possiamo usare gli indici dell'array per tenere traccia delle 'parenteli':

Genitore del nodo a K si trova a  $K/2$

I figli del genitore a K si trovano a  $2K$  e  $2K+1$

## Operazioni sul binary heap

### Inserimento:

- Aggiungo l'elemento nella PROSSIMA posizione rispetto all'ultima posizione utile dell'array.
- Verifico se viene violato l'heap Order
- Se si faccio salire l'el di un livello scambiandolo col padre.
- Ripeto finche' non rispetta l'heap order.

### Rimozione:

- Porto il primo elemento all'ultima posizione (scambio)
- Elimino e' el. in ultima posizione
- da nuova radice viola l'H.O.  $\rightarrow$   
Entrambi i suoi figli sono maggiori; effettuo lo scambio con il figlio maggiore tra i due.
- Ripeto finche' l'elemento non viola l'H.O.

## Eliminare una chiave random

Per eliminare una chiave random all'interno del binary heap basterebbe effettuare le scambi dell'elemento da eliminare con l'ultimo elemento; successivamente eliminare l'ultimo elemento e, in fine verificare che gli elementi rispettino l'H.O.