Algoritmo di tipo divid: et impera

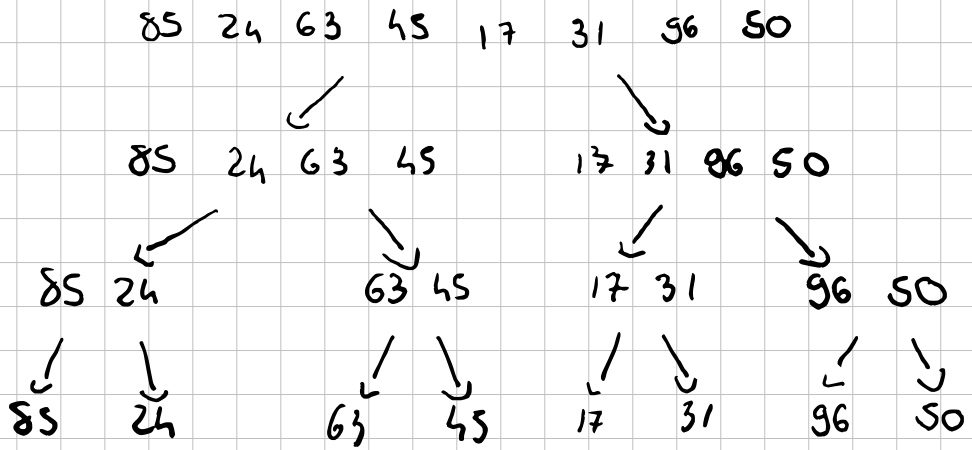
MERGE SORT (O(nlogn))

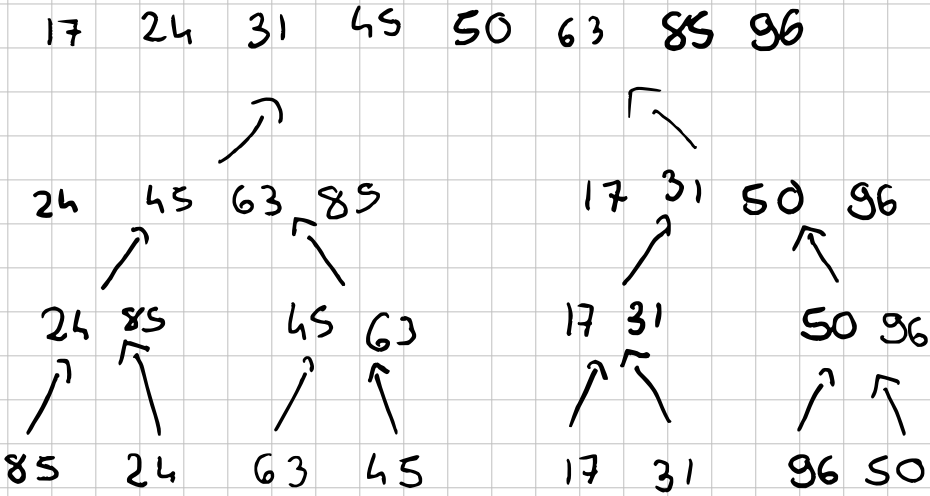
- **DIVIDI:**
  Se dimensione dei dati da elaborare è minore di una certa somma ordina con metodo banale, altrimenti DIVIDI i dati in 2 sottinsiemi

- **CONQUISTA:**
  Risolvi ricorsivamente i sottoproblemi associati ai sottoinsiemi di dati

- **COMBINA:**
  Prendi le soluzioni dei sottoproblemi e combinale

ES.





L'algoritmo ha complessità $O(n \log n)$ poiché:
il costo dell'algoritmo nel codice "merge" (1° parte di divisione) è, dato S_1 e S_2: due sottoinsiemi e n_1 e n_2 rispettiv. la loro dimensione, $O(n_1 + n_2)$, in quanto per ordinarlo sarà lineare.

Un albero merge-sort, come quello dell'esempio, ha un altezza $\log n$.
Considerando un nodo v generico dell'albero, si avrà la fase di scissione, proporzionale alla dimensione di v, come anche la fusione, che è anch'esse lineare.
Se si guarda con: l'altezza, il tempo sarà $O(n/2^i)$.

Guardando all'intero nodo, si può vedere come il numero dei nodi sia uguale a 2^i, quindi il costo è $(2^i \cdot n/2^i) = O(n)$. Per ogni altezza, il tempo di computazione dell'ordinam. e scissione è proprio $O(n)$.

Il numero di altezza è $\log(n)$, quindi il costo totale è $O(n \log n)$

Eq di ricorrenza

$$t(n) = \begin{cases} b & \text{se } n \leq 1 \\ 2t(n/2) + cn & \text{altrimenti} \end{cases}$$

$t(n)$ è ricorsiva, quindi inseriamo $t(n)$ nella funzione

$t(n) = 2t(n/2) + cn =$

$t(n/2) = 2t(n/4) + cn/2 \Rightarrow 4t(n/4) + 2cn = 8t(n/8) + 3cn$

$\vdots$

$t(n) = 2^i t(n/2^i) + icn$

esso continua finché $n/2^i = 1 \rightarrow i = \log_2 n$
e a quel punto $t(n) = b$

$\rightarrow t(n) = 2^{\log_2(n)} b + \log_2 n \cdot cn =$

$= cn \cdot \log n + nb \Rightarrow O(t(n)) = O(n \log n)$

QUICK SORT

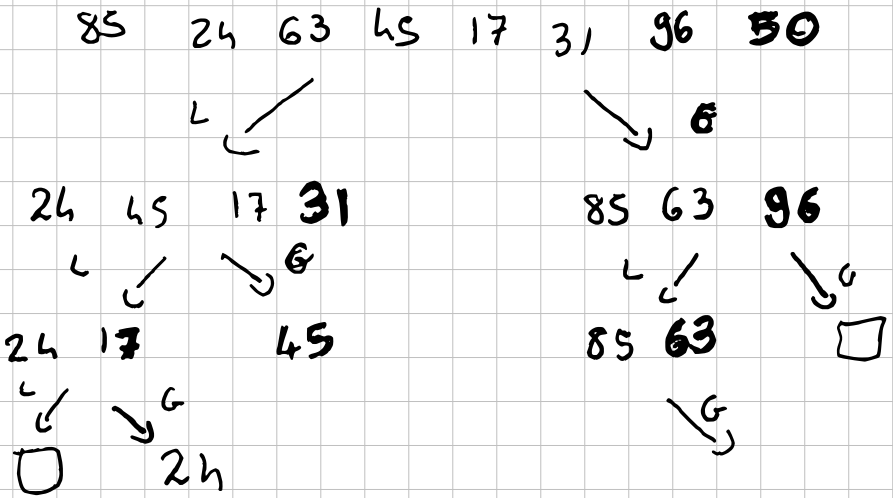
1 dividi : se S è vuota o ha un solo elemento allora restituisci S. Altrimenti seleziona un elemento $x \in S$, che sarà il pivot. A questo punto dividi gli elementi di S in 3 categorie
- L : elementi MINORI di x
- E : elementi UGUALI ad x
- G : elementi MAGGIORI di x

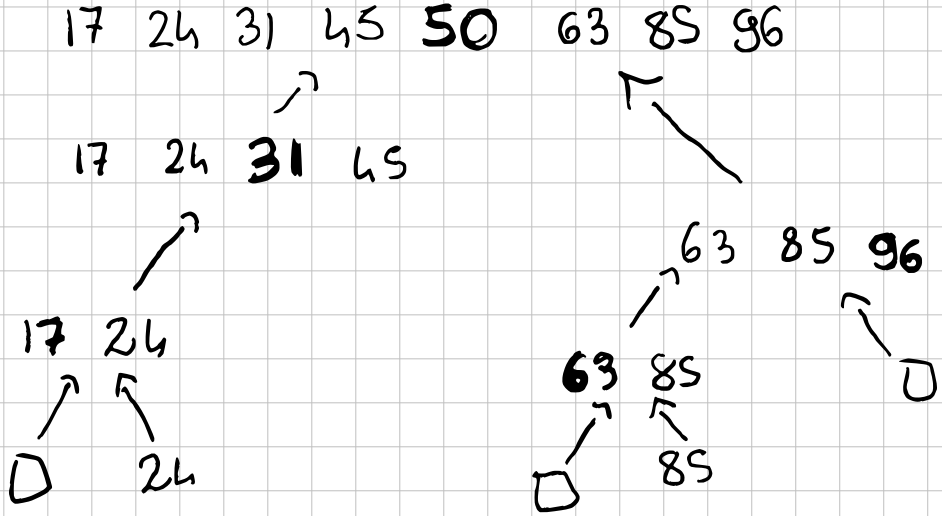
2 conquista: ordina ricorsivamente le sequenze L e G

3 Combina: Memorizza gli elementi in S, prima L, poi E, poi G

NB: come pivot si sceglie di solito l'ultimo EL.

es.





ottima l'implementazione con Coda

```
quickSort(Queue K)
  se  n<2  return
  else
      pivot
      while(K.isEmpty)
        c=K.deQueue;
        if (c<k)
           L.enqueue(c)
        else if (c=k)
           E.enqueue(c)
        else
           G.enqueue(c)
        quickSort(L)
        QuickSort(G)
        svuota L in K
        svuota E in K
        svuota G in K
```

Pseudo codice

il costo dell'algoritmo, nel caso peggiore è $O(n^2)$.
Il caso peggiore consiste proprio nel fatto di avere scelto come pivot L'ULTIMO ELEMENTO, ogni volta, in quanto il costo sarebbe

$$t(n) = n + (n-1) + (n-2) : \frac{n(n+1)}{2} = O(n^2).$$

$O(n \cdot n)$

Se scegliemo un elemento RANDOMICO, invece, il valore atteso ci dice che

$$E(x) = \frac{\sum_{i=0}^{n} x_i}{n}$$

è a metà! Se si sceglie come elemento di pivot quello centrale si avrà come valore atteso dell'altezza $h = O(\log n)$

$\rightarrow O(n \log n)$

Esso sarà il tempo atteso dell'algoritmo QuickSort

ORDINAMENTO in tempo LINEARE

Questi algoritmi, che si vedranno successivamente, hanno costo asintotico minore di $\Omega(n \log n)$, ma hanno bisogno di ipotesi.
ES. Coppie chiave/valore, con vincolo sull'e vari di appartenza

Bucket sort

Secondo le ipotesi, il limite inferiore sul costo asintotico consiste nel fatto di avere CONFRONTI.
l'algoritmo non fa confronti, infatti consiste nell'ordinare le chiavi secondo bucket, con chiave come indice.

Dopo aver inserito ogni elemento di S nel bucket, si scandiscono i bucket secondo B[0], B[1] ... B[n-1], così che saranno ORDINATI.

S: fa l'ipotesi che si CONOSCA il numero massimo delle chiavi!

```
Algoritmo bucketSort(S)
 input : sequenza S
output. sequenza S ordinata
B array di n bucket
for (e in S)
    k = e.getKey();
    elimina e da S e assegnalo e in B[k]
for i in range (0, n-1)
   for (e in B[i])
     delete e from B[i] e inseriscilo alla fine di S
```

il costo è $O(n+N)$
n = numero di elementi in S
N = intervallo di valori.

Se $N >> n$ prestazioni peggiorano

Radix Sort e Ordinamento Stabile

Quando si ordinano coppie chiave-valore è possibile che vogliamo gestire le chiavi uguali.
Sia $S = ((k_0, v_0), \ldots, (k_{n-1}, v_{n-1}))$ una sequenza di voci.

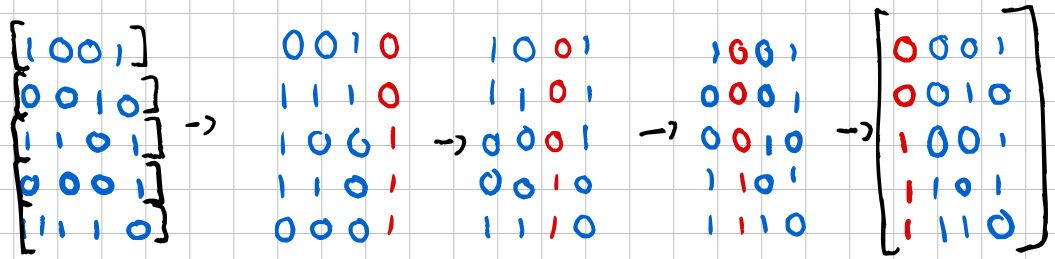
Un algoritmo di ordinamento è **STABILE** se $\forall$ coppia di valori, in cui $k_i = k_j$ e $v_i < v_j$, la voce (k_i, v_i) precede quella (k_j, v_j) in S

Per effettuare un ordinamento stabile si usa un'altro algoritmo che sfrutta il bucket Sort

Radix Sort

L'algoritmo ordina una sequenza S di voci, applicando 2 volte l'algoritmo di ordinamento bucket Sort, una volta per la prima componente, la seconda per l'altra

Bisogna ordinare prima per la seconda, poi per la prima. Es. sequenza di interi a 4 bit



Costo $O(d(n+N))$

con d il numero di componenti del dato da ordinare.

Cosa scegliere?

QuickSort:

Buona scelta predefinita, molto veloce nella pratica, scegliendo il pivot randomicamente

HeapSort:

Ottimo se non si può assolutamente avere $O(n^2)$ e se si vuole avere un basso impatto nella memoria.
Lo usa il kernel linux

Merge Sort:

buona scelta per algoritmo stabile. Merge sort è estensibile per tipi di dato non inseribili in RAM. Costo dato da lettura, scrittura su disco

Radix Sort:

Sembra veloce, ma nasconde b elevato a volte (es. ordinamento numeri binari). Tende ad essere lento in pratica. $\log N > b$
$N < 16$ mld di mbd. Molto lento.

Insertion Sort:

Costo $O(n+m)$, con m numero di inversioni. Ottimo per ordinare piccole sequenze ($n < 50$), e anche per liste già "quasi ordinate" m, vicino a 0
Si rischiano prestazioni quadratiche, si tende ad evitarlo.
(Si può usare in-place selection sort)