

## i. INSERTION SORT

TECNICA INCREMENTALE:  $n$  elementi già ordinati, estendo la soluzione all'elemento  $n+1$

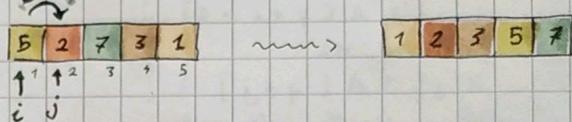
ORDINATO	$j$
----------	-----

ORDINAMENTO IN LOCO (SOL POSIZIO): in ogni istante al più un numero costante di elementi dell'array di input sono registrati all'esterno dell'array

@ POST: ordina  $A[1 \dots A.length]$

• INSERTION-SORT ( $A$ )

```
for j = 2 * to A.length * il 1° elemento è già
    key = A[j]      ordinato (è da solo)
    i = j - 1
    while i > 0 AND A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```



key = 2     $i=1 \rightarrow 5731$

$i=0 \rightarrow 25731$

worst case: ordine inverso

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

è STABILE!

best case:  $A$  ordinato non decrescente  $\Theta(n)$

• INT = il sottoarray  $A[1 \dots j-1]$  è formato dagli elementi ordinati che originariamente erano in  $A[1 \dots j-1]$

CONCLUSIONE: quando il ciclo termina  $j = A.length + 1$  e l'INT è vero

INT  $\left[\frac{A.length+1}{j}\right] =$  il sottoarray  $A[1 \dots A.length + 1 - 1]$  è formato dagli elementi ordinati che originariamente erano in  $A[1 \dots A.length + 1 - 1]$

ma allora ho ordinato l'array di pertinenza ottenendo una permutazione degli elementi dell'input di pertinenza

## • COMPLESSITÀ

L'algoritmo ORDINA in loco  $n$  elementi, eseguendo, nel caso peggiore,  $\Theta(n^2)$  confronti

• Dimostrazione

- ordina in loco perché ho un solo elemento memorizzato fuori dall'array di input

- il ciclo ESTERNO è eseguito  $n-1$  volte ed esegue un numero di confronti:

$$\sum_{j=2}^n \sum_{i=j-1}^{n-1} n = \sum_{i=1}^{n-1} (n-i)(n-i+1) = \frac{n(n-1)}{2} = \Theta(n^2) \quad (n = A.length)$$

⚠️ Algoritmo sensibile all'ordinamento dei dati in ingressi ⚠️ (usato per input di piccole dimensioni)

## ii. MERGE-SORT

• TECNICA DIVIDE-ET-IMPERA: voglio ordinare il sottoarray  $A[p \dots r]$ , all'inizio  $p=1$  e  $r=A.length=n$

(1) DIVIDE: l'array in due sottoarray  $A[p \dots q]$  e  $A[q+1 \dots r]$  dove  $q = L(p+r)/2$

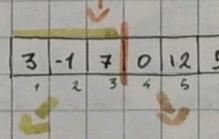
(2) IMPERA: ordina i due sottoarray (ricorsivamente) utilizzando il MERGE-SORT

se il problema è sufficientemente piccolo (zero/uno elemento) risolve direttamente

(3) COMBINA: fonde insieme i due sottoarray ordinati per generare un singolo array ordinato  $A[p \dots r]$

• MERGE-SORT ( $A, p, r$ )

```
if p < r /* ho almeno 2 elementi */
    q = L(p+r)/2
    MERGE-SORT (A, p, q) ] T(1/2)
    MERGE-SORT (A, q+1, r) ] T(1/2)
    MERGE (A, p, q, r) ->  $\Theta(n)$ 
```



$$q = L(p+r)/2 = 3$$

effetto delle  
merge sort  
come? non mi  
importa! pensa all'effetto

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

> merge: \* input  $A, p, q, r$   
output due sottoarray fusi in un unico array ordinato  $A[p \dots r]$

\* vincoli:  $p \leq q < r \Rightarrow$  nessun sottoarray è vuoto!  
il sottoarray  $A[p \dots q]$  è ordinato  
il sottoarray  $A[q+1 \dots r]$  è ordinato

è STABILE!

idea: prendo il 1° elemento da ciascuno, li confronto, metto in ordine

• MERGE(A, p, q, n)  
 $n_1 = q - p + 1$  /\* numero elementi di  $A[p \dots q]$  \*/  
 $n_2 = n - q$  /\* numero elementi di  $A[q+1 \dots n]$  \*/  
 crea gli array  $L[1 \dots n_1+1]$  e  $R[1 \dots n_2+1]$   
 /\* copio gli elementi dei due sottovettori in L e R \*/

```

    for i=1 to n1
      L[i] = A[p+i-1]
    for j=1 to n2
      R[j] = A[q+j]
    /* valore sentinelle, nella realtà uso un while */
    L[n1+1] = +∞
    R[n2+1] = +∞
  
```

$i=1, j=1$   
 for  $k=p$  to  $n$  ←  $\Theta(n-p+1) = \Theta(n)$   
 if  $L[i] \leq R[j]$   
   A[k] = L[i]  
   i = i + 1  
 else  
   A[k] = R[j]  
   j = j + 1

L	-1	3	7	∞
	2			
R	0	5	12	∞

A	-1	0	3	5	7	12
	2					

$$T_{\text{merge}}(n) = \Theta(n)$$

$$T(n) = \Theta(n_1 + n_2) + \Theta(n) = \Theta(n)$$

$$(q-p+1) + (n-q) = n-p+1 = n$$

• INV = il sottoarray  $A[p \dots n-1]$  contiene ordinati i  $n-p$  elementi più piccoli di  $L[1 \dots n_1+1]$  e  $R[1 \dots n_2+1]$ . Inoltre  $L[i]$  e  $R[j]$  sono i più piccoli elementi dei loro array che non sono stati ancora copiati in A  
 CONCLUSIONE: quando il ciclo termina  $n = n+1$

$\text{INV} \left[ \frac{n+1}{n} \right] =$  il sottoarray  $A[p \dots n+1-1]$  e contiene ordinati i  $n+1-p$  elementi più piccoli di  $L[1 \dots n_1+1]$  e  $R[1 \dots n_2+1]$ . Inoltre  $L[i]$  e  $R[j]$  sono i più piccoli elementi dei loro array che non sono ancora stati copiati in A

~ #elementi in L e R:

$$n_1+1 + n_2+1 = (q-p+1)+1 + (n-q)+1 = n-p+3 \quad \text{MA io ne ho copiati } n-p+1 \Rightarrow \text{ho escluso i valori sentinella!}$$

### • COMPLESSITÀ MERGE-SORT

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

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

[caso master]  $f(n) = \Theta(n)$ ;  $g(n) = n^d = n^{\log_2 e} = n^{\log_2 2} = n \Rightarrow f(n) = \Theta(n^d) \Rightarrow T(n) = \Theta(n^d \log n)$

### • VANTAGGI MERGE-SORT

i. complessità è  $\Theta(n \log n)$

ii. metodo **STABILE**: i numeri con lo stesso valore (duplicati) si presentano nell'array di output nello stesso ordine in cui si trovano nell'array di input.

«info»	1	-3	1	7	-3	8	merge set	-3	-3	1	1	7	8
«dati satelliti»	A	B	C	D	E	F	(anche Insertion set)	B	E	A	C	D	F

### • SVANTAGGI MERGE-SORT

- i. NON è in loco, necessita di memoria aggiuntiva  $\Theta(n)$  proporzionale al numero di elementi da ordinare
- ii. NON è sensibile all'ordinamento degli elementi, il tempo di calcolo dipende solo dal numero degli elementi (chiavi) da ordinare

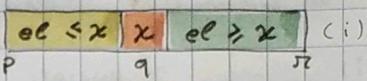
### iii. QUICK - SORT

(inizializzati:  $p=1$ ;  $n = A.length = n$ )

$q$  non è considerato

TECNICA DIVIDE - ET - IMPERA  $\Rightarrow A[p \dots n]$

- 1. DIVIDE: partiziona l'array  $A[p \dots n]$  in due sottoarray  $A[p \dots q-1]$  e  $A[q+1 \dots n]$  anche se vuoti!  
sono tali che, ogni elemento di  $A[p \dots q-1]$  è minore o uguale ad  $A[q]$   
che a sua volta è minore o uguale ad ogni elemento di  $A[q+1 \dots n]$ 
  - $q$  è parte del PARTIZIONAMENTO
  - $A[q]$  è chiamato «PIVOT»



- 2. IMPERA: ordina ricorsivamente i due sottoarray  $A[p \dots q-1]$  e  $A[q+1 \dots n]$   
chiamando il Quick-Sort; se il problema è

sufficientemente piccolo risolve direttamente (zero/uno elemento)

- 3. COMBINNA: NON fa niente dato che i due sottoarray sono ordinati in loco, ma allora  $A[p \dots n]$  è ordinato!

• POST: ordina  $A[p \dots n]$

• QUICK-SORT ( $A, p, n$ )

if  $p < n$

PIVOT: scelgo l'ultimo elemento!

•  $q = \text{PARTITION}(A, p, n)$

QUICK-SORT ( $A, p, q-1$ )

QUICK-SORT ( $A, q+1, n$ )

elemento  $\leq x$

oscilla tra  
nlogn e  $n^2$

• PARTITION ( $A, p, n$ )

•  $x = A[n]$  /\* pivot, ultimo elemento \*/

•  $i = p-1$

for  $j=p$  to  $n-1$

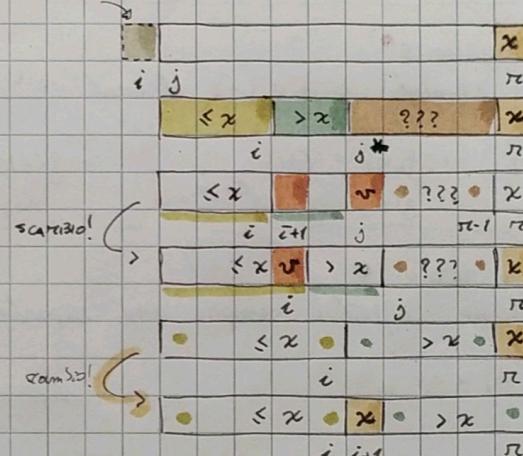
if  $A[j] \leq x$

•  $i = i + 1$

• SCAMBIO  $A[i]$  e  $A[j]$

SCAMBIO  $A[i+1]$  e  $A[n]$

return  $A[i+1]$  /\* indice pivot \*/



joue del for

• INV =  $x = A[n] \wedge \forall k \in [p \dots i]: A[k] \leq x \wedge \forall k \in [i+1 \dots j-1]: A[k] > x$

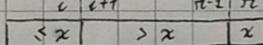
$\wedge (p \leq j \leq n) \wedge (p-1 \leq i \leq j-1)$  da  $j$  inizia la parte NON SO!\*

CONCLUSIONE: alla fine del ciclo for  $j=n$

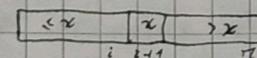
ho finito la parte de

INV [ $\frac{\pi}{\delta}$ ] =  $x = A[n] \wedge \forall k \in [p \dots i]: A[k] \leq x \wedge \forall k \in [i+1 \dots n-1]: A[k] > x$  ed ora!!  $\wedge \dots$

↳ ho ripartito i valori in 3 parti



• La ultime due righe inseriscono il PIVOT  $x$  nella partizione corretta scambiando l'elemento più a sx che è maggiore di  $x$



\* SIMULAZIONE PARTITION!

2	8	7	1	3	5	6	4
i	j						n

(i)  $2 \leq 4 \checkmark$

2	8	7	1	3	5	6	4
i	j						n

(ii)  $8 \leq 4 \times$

$7 \leq 4 \times$

$1 \leq 4 \checkmark$

2	1	7	8	3	5	6	4
i	j						n

(iii)  $3 \leq 4 \checkmark$

2	1	3	8	7	5	6	4
i	j						n

(iv)  $5 \leq 4 \times$

$6 \leq 4 \times$

scambio!

2	1	3	4	7	5	6	8
i	i+1						n

PRESTAZIONI Quick Sort \* (-1) il punto è già impostato finale  
 $T(n) = \begin{cases} c & n \leq 1 \\ T(k) + T(n-k-1) + \Theta(n) & n > 1 \end{cases}$

# elementi di un sottovettore # elementi altro sottovettore PARTITION!

\* dipendono dal partizionamento dell'array!

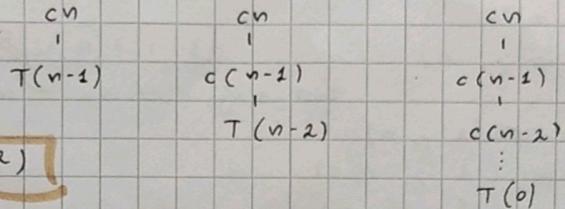
→ oscilla tra  $n \log n$  e  $n^2$

- (i) sottovettori BILANCIATI: complessità come il MERGE SORT ( $n \log n$ )
- (ii) sottovettori SBILANCIATI: prestazioni pessime come l'INSERTION SORT nel caso peggiore ( $n^2$ )

### PARTIZIONAMENTO NEL CASO PEGGIORIO

- VETTORE ORDINATO: una partizione ha  $n-1$  elementi e zero dall'altra (AD OGNI CHIARATTA RICORSIVA!)

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= T(n-1) + cn \\ &= \sum_{i=1}^n ci + T(0) = c \frac{n(n+1)}{2} + T(0) = \Theta(n^2) \end{aligned}$$



### PARTIZIONAMENTO NEL CASO MIGLIORIO

- sottoproblemi con  $\lfloor \frac{n}{2} \rfloor$  elementi e  $\lceil \frac{n}{2} \rceil - 1$  elementi

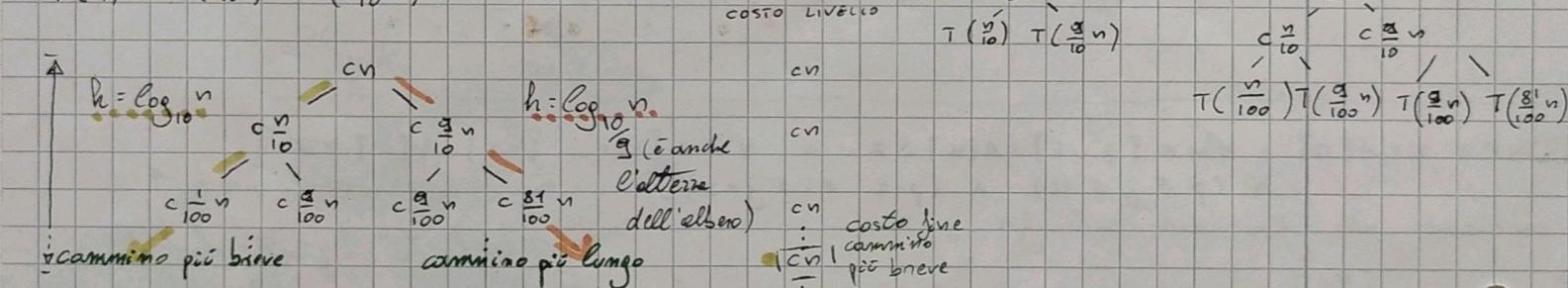
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \rightsquigarrow f(n) = \Theta(n) \Rightarrow f(n) = \Theta(n^d) \Rightarrow T(n) = \Theta(n \log n)$$

[t. master]  $a=2, b=2$   
 $g(n) = n^d = n \log_2 n = n$

### PARTIZIONAMENTO NEL CASO MEDIO

- suppongo che l'algoritmo effettui sempre una ripartizione proporzionale (esempio 9 a 1)

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9}{10}n\right) + cn$$



$$\Rightarrow T(n) \leq cn \cdot \log_{\frac{9}{10}} n$$

$$T(n) = T(\alpha n) + T((1-\alpha)n) + cn \quad \left\{ \begin{array}{l} \alpha < 1 \\ 0 < \alpha < 1 \end{array} \right. \Rightarrow T(n) = \Theta(n \log n) \quad \text{costo se mantengo la proporzionalità}$$

ogni livello ha un costo cn finché NON raggiunge la profondità  $\log_{\frac{9}{10}} n$  dopo la quale il livello avrà al più un costo limitato superiormente da cn. La ricorsione termina alla profondità  $\log_{\frac{9}{10}} n = \Theta(\log n)$ .  
 dunque il costo complessivo è dato da  $\Theta(n \log n)$

### ALTERNATIVA: PARTIZIONI BUONE E PESSIME

$$\begin{cases} L(n) = 2L\left(\frac{n}{2}\right) + \Theta(n) & \rightsquigarrow (\text{Lucky}) \text{ PARTIZIONE BILANCIATA} \\ M(n) = L(n-1) + \Theta(n) & \rightsquigarrow (\text{Unlucky}) \text{ PARTIZIONE FORTEMENTE SBILANCIATA} \end{cases}$$

] → l'altro!

$$L(n) = 2 [L\left(\frac{n}{2}-1\right) + \Theta\left(\frac{n}{2}\right)] + \Theta(n)$$

$$\text{essomiglia a } = 2L\left(\frac{n}{2}-1\right) + 2\Theta\left(\frac{n}{2}\right) + \Theta(n)$$

$$\text{quella del caso} \quad \triangleright = 2L\left(\frac{n}{2}-1\right) + \Theta(n) = \Theta(n \log n)$$

virgine (sì può dimostrare che è vlogn)

sono entrambi lineari!

è difficile  
 da gestire

• **QUICK SORT RANDORIZZATO**: scambia  $A[\pi]$  con un elemento scelto a caso da  $A[p \dots \pi]$

• **RANDOMIZED-PARTITION** ( $A, p, \pi$ )

•  $i = \text{random}(p, \pi)$

scambia  $A[i]$  e  $A[\pi]$

return PARTITION ( $A, p, \pi$ )

migliora le prestazioni, è più probabile avere partizioni alternate!

• **RANDOMIZED-QUICKSORT** ( $A, p, \pi$ )

if  $p < \pi$

$q = \text{RANDOMIZED-PARTITION}(A, p, \pi)$

RANDOMIZED-QUICKSORT ( $A, p, q-1$ )

RANDOMIZED-QUICKSORT ( $A, q+1, \pi$ )

algoritmo eccellente!

• **VANTAGGI** (assumendo che TUTTE le chiavi siano DISTINTE!)

- il tempo di esecuzione è INDEPENDENTE dall' ordinamento dell' input

- non faccio assunzioni sulla distribuzione dell' input

- il CASO PEGGIORE è determinato solo dal generatore di numeri casuali (no input!)

se fossero duplicate non posso evitare il caso peggiore

(ad esempio chiavi tutte uguali)

• **OTTIMIZZAZIONI QUICK SORT**

i. utilizzare l' INSERTION-SORT su vettori di piccole dimensioni (risolve direttamente)

a. if  $(\pi-p \leq M)$

INSERTION-SORT ( $A, p, \pi$ )

b. if  $((\pi-p) \leq M)$

return ; \* non fa nulla (enq piccole dimensioni)

M è un parametro costante il cui valore dipende dalle implementazioni (spesso  $5 \leq M \leq 25$ )

void SORT ( $A, p, \pi$ )

• QUICK-SORT ( $A, p, \pi$ ) rende un array quasi ordinato!

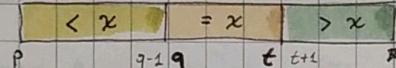
• INSERTION-SORT ( $A, p, \pi$ )

• completa l'ordinamento (è sensibile agli elementi ordinati)

ii. si sceglie il più come la mediana di tre elementi del vettore

→ un elemento è sx, uno è dx e uno al centro; calcolo mediana e scambio con  $A[\pi]$

iii. **TRIPARTIZIONE**:



in presenza di chiavi duplicate!

• **POST**: permuta gli elementi di  $A[p \dots \pi]$  e restituisce due indici  $q$  e  $t$ , con  $p \leq q \leq t \leq \pi$ , tale che

\* ~ elementi di  $A[q \dots t]$  sono uguali

\* ~ ogni elemento di  $A[p \dots q-1]$  è minore di  $A[q]$

\* ~ ogni elemento di  $A[t+1 \dots \pi]$  è maggiore di  $A[q]$

\*/ ~ partiton dura un tempo lineare  $\Theta(\pi-p)$ .

• **PARTITION** ( $A, p, \pi$ )

$x = A[\pi]$   
min = eq = ?  
mag =  $\pi$

(\*!) while (eq < mag)

if  $A[eq] < x$

(a) scambia  $A[min]$  e  $A[eq]$

eq = eq + 1

min = min + 1

else if  $A[eq] == x$

(b) eq = eq + 1

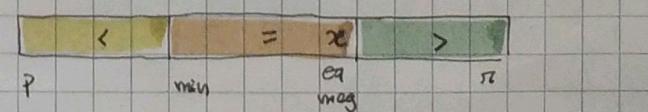
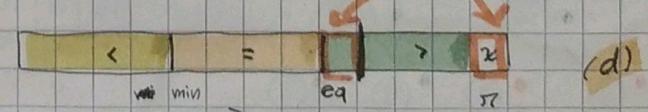
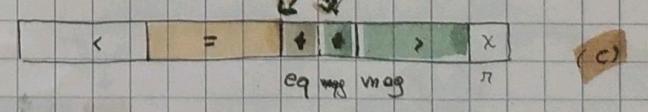
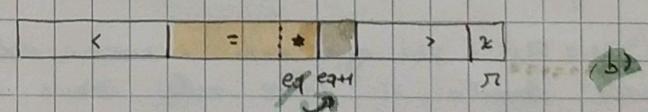
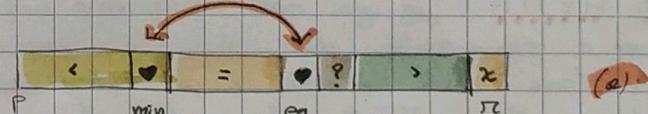
else

mag = mag - 1

(c) scambia  $A[mag]$  e  $A[eq]$

scambia  $A[mag]$  e  $A[\pi]$

return  $\langle min, mag \rangle$



COMPLICATITÀ

(\*!) il corpo del ciclo è eseguito  $\pi-p$  volte

→  $T(n) = \Theta(\pi-p) = T(n) = \Theta(n) [ \pi-p+1 = n ]$

- $\text{INV} \equiv x = A[\pi] \wedge \forall \kappa \in [p \dots \min]: A[\kappa] < x$
- $\wedge \forall \kappa \in [\min, \text{eq}]: A[\kappa] = x$
- $\wedge \forall \kappa \in [\text{mag}, \tau]: A[\kappa] > x$
- $\wedge p \leq \min \leq \text{eq} \leq \text{mag} \leq \tau$

CONCLUSIONE: quando il ciclo termina [ $\text{eq} = \text{mag}$ ]

- $$\text{INV} \left[ \frac{\text{mag}}{\text{eq}} \right] \equiv x = A[\pi] \wedge \forall \kappa \in [p \dots \min]: A[\kappa] < x$$
- $\wedge \forall \kappa \in [\min, \text{mag}]: A[\kappa] = x$
  - $\wedge \forall \kappa \in [\text{mag}, \tau]: A[\kappa] > x$

non ho più elementi non so!

$< x$	$= x$	$> x$	$x$
p	min	eq	mag

La penultima riga posiziona  $x$  in modo che il vettore sia posizionato come richiesto! ~

$< x$	$= x$	$x$	$> x$
p	min	mag	τ

• QUICK-SORT' ( $A, p, \tau$ )

if  $p < \tau$

•  $\langle q, t \rangle = \text{PARTITION}'(A, p, \tau)$

quick-sort' ( $A, p, q-1$ )

quick-sort' ( $A, q, t+1, \tau$ )

meglio se chiamassi

RANDOMIZED-PARTITION' ( $A, p, \tau$ )

algoritmo della  
bandiera  
olandese

⚠ se ho elementi tutti uguali fa solo la PARTITION! ~  $T(n) = \Theta(n)$

algoritmo di ordinamento in loco

in media è eseguito in  $\mathcal{O}(n \log n)$

nel caso peggiore il tempo di esecuzione è  $\mathcal{O}(n^2)$

NON è stabile

vettore posizionale!

• HEAP: memorizzato in un array  $A$  con 2 attributi \*

ALBERO BINARIO quasi completo, dove tutti i livelli dell'albero sono completamente riempiti tranne eventualmente l'ultimo in cui TUTTE le FOGLIE sono addossate a SINISTRA!

( $\rightarrow$  vettore senza buchi!)

•  $A.\text{length}$ : # elementi array

•  $A.\text{heap-size}$ : # elementi dell'HEAP memorizzati in  $A$  ~ $A[1 \dots A.\text{heap-size}]$

•  $A[1] = \text{radice albero}$

• se  $i$  è l'indice d'un nodo =>

PARENT( $i$ )  
return  $\lfloor \frac{i}{2} \rfloor$

LEFT( $i$ )  
return  $i * 2$

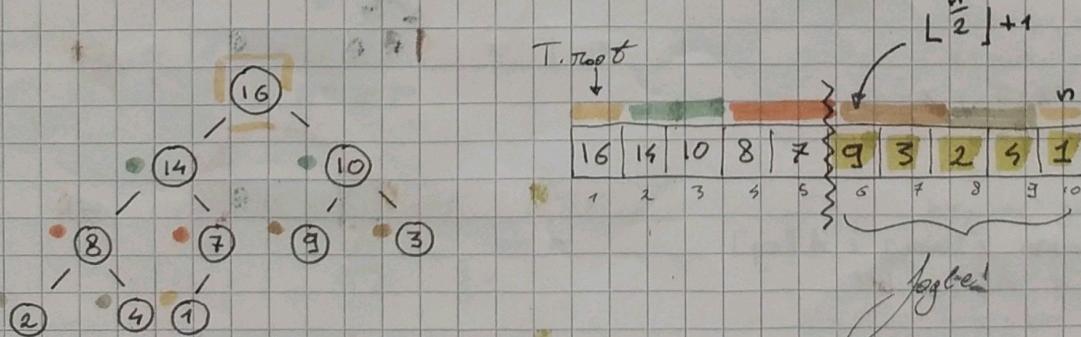
RIGHT( $i$ )  
return  $i * 2 + 1$

(i) MAX HEAP: per ogni nodo  $i$  DIVERSO dalla RADICE per insuzione e transitività del  $\leq$ , la proprietà di MAX HEAP garantisce che il massimo elemento si trova nella RADICE e che il sottoalbero di un nodo contiene valori NON maggiori di quelli contenuti nel modo stesso

•  $A[\text{parent}(i)] \geq A[i]$

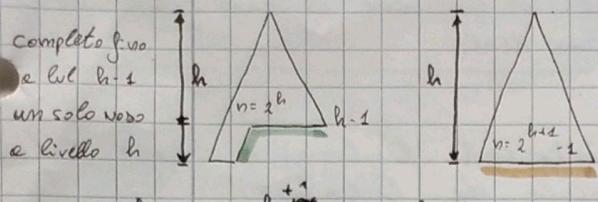
(ii) MIN HEAP: per ogni nodo  $i$  DIVERSO dalla RADICE il più piccolo elemento si trova nella RADICE dell'albero

•  $A[\text{parent}(i)] \leq A[i]$



i. L'alterazione di un heap di  $n$  elementi è  $\mathcal{O}(n)$ .

dato che un vettore è un'altra cosa quasi completa, se ha un'altezza  $h$  allora ha  $2^h \leq n \leq 2^{h+1} - 1$



$$\sum_{i=0}^{h-t} 2^{\underline{i+1}} = \frac{2^{h-t+1} - 1}{2-1} + 1 = 2^h$$

add all terms  $h$

$$\sum_{i=0}^h 2^i = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$

$$\text{ma allora } 2^h \leq n < 2^{h+1} - 1 < 2^{h+1} = h \leq \log_2 n < h+1 \Rightarrow h = \lfloor \log_2 n \rfloor$$

ii. Nell'array che rappresenta un heap di  $n$  elementi, le foglie sono i nodi con indici  $\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, \dots, n$ .

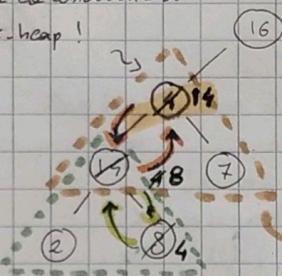
iii. Ci sono al massimo  $\left\lceil \frac{n}{2^{h+1}} \right\rceil$  nodi di altezza  $h$  in un qualsiasi heap di  $n$  elementi.  
 ↳ ci sono tanti nodi che hanno altezze piccole (massima lunghezza cammino da nodo a foglia)  
 N.B. se  $h = 0$  (altezza foglie!)  $\Rightarrow \# \text{foglie} \leq \left\lceil \frac{n}{2^0} \right\rceil = \left\lceil \frac{n}{2} \right\rceil$

\* @PRE: gli alberi bimani con radice left(i) e right(i) sono max HEAP  
\* @Post: l'albero radicato in i è un max HEAP  
\*/

• MAX-HEAPIFY ( $A$ ,  $i$ )

$l = \text{left}(i)$   
 $r = \text{right}(i)$   
 if  $l \leq A.\text{heap\_size}$  AND  $A[l] > A[i]$   
     •  $\text{max} = l$   
 else  
     •  $\text{max} = i$   
 if  $r \leq A.\text{heap\_size}$  AND  $A[r] > A[\text{max}]$   
     •  $\text{max} = r$   
 if  $i \neq \text{max}$   
     • SWAP(A[i], A[max])  
     • MAX-HEAPIFY(A, max)

viale la condizione  
max-heap!



$$f(n) = O(b)$$

\* O(log n) dato che c'è l'attesa dell'heap

prendo il max  
Era i 3 nodi  
e scambiò figlio con padre  
riapplicando la max - neopoli

il tempo di esecuzione è  $O(h)$  dove  $h$  è l'altezza del nodo  $i$  a cui applico la funzione, perché ad ogni chiamata ricorsiva scendo di un livello dell'albero! \*

`/* @post: costuisce un heap dato un vettore disordinato`

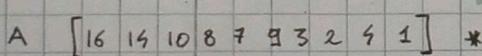
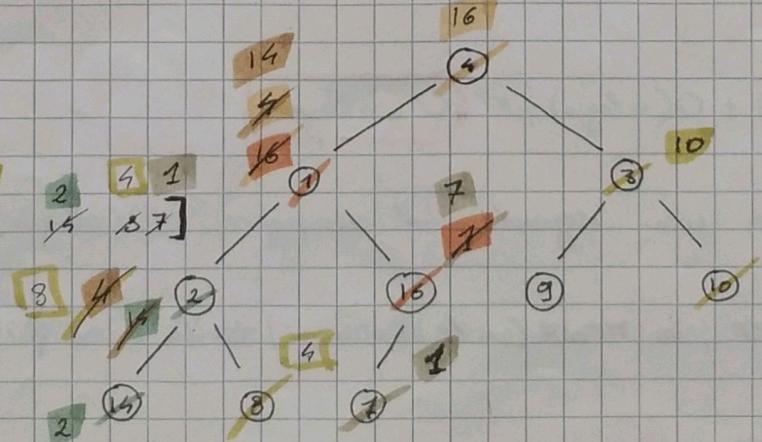
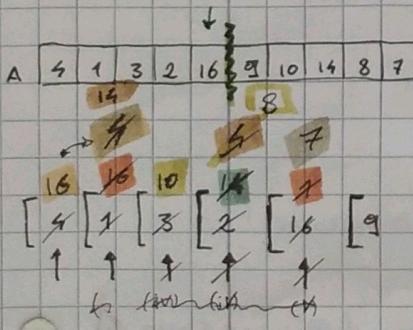
### • BUILD-MAX-HEAP (A)

A.heap.size = A.length  
 for  $i = \lfloor A.length / 2 \rfloor$  down to 1  
 $\bullet$  MAX-HEAPIFY (A, i)

La appello a metà ormai perché i suoi figli sono FOGLI! dunque sono dei dannati. MAX HEAP \*

$$T(n) = O(n)$$

$\text{IN}^\Gamma \equiv$  ogni nodo  $i+1, i+2, \dots, n$  è rotarè di un MAX HEAP dove  $n = \text{A.length}$



CONCLUSIONE: quando il ciclo termina  $i=0$

INV  $\left[ \frac{0}{i} \right] =$  ogni nodo  $1, 2, \dots, n$  è radice di MAX-HEAP. in particolare il nodo  $1$  che contiene la radice dell'albero è un MAX-HEAP

### COMPLICATITÀ

- limite superiore (NON STRETTO): ogni chiamata di max-heapify costa  $O(\log n)$  e ci sono  $O(n)$  chiamate. quindi il tempo d'esecuzione è limitato da  $O(n \log n)$
- limite superiore (STRETTO): il tempo di max-heapify è  $O(h)$  e varia con l'altezza del nodo a cui si applica

C. se applico le max-heapify a tutti i nodi dell'albero

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) = \star$$

$\star \leadsto \text{max node altezza } h$

$$= O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(2n) = O(n)$$

POSSO TAGLIODDARE!

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2(1-\frac{1}{2})} = 2 \Rightarrow \sum_{h=0}^{\infty} h x^h = \frac{x}{(1-x)^2}$$

### HEAP SORT

#### • HEAP-SORT (A)

BUILD-MAX-HEAP (A)  $\sim O(n)$

/\* estraevo  $n-1$  volte l'elemento massimo e lo metto come ultimo dentro \*/

for  $i := A.length$  DOWN TO 2  $\leftarrow n-1$  volte

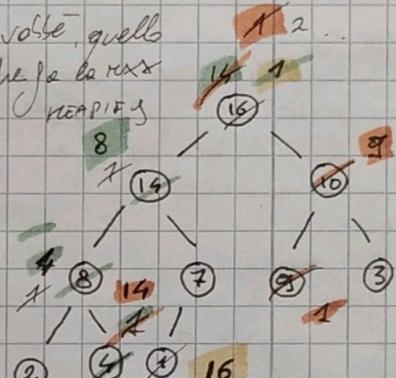
SCAMBIA  $A[1]$  con  $A[i]$

$A.heap-size = A.heap-size - 1$

MAX-HEAPIFY ( $A, 1$ )  $\rightarrow$  chiama sempre sulla radice  $O(\log n)$

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

voglio quelli  
che fa la MAX  
HEAPIFY



... 14 16  
 $i \leftarrow i+1$

INT: il sottovettore  $A[1 \dots i]$  è un max-heap che contiene gli  $i$  elementi più piccoli del vettore di partenza  $A[1 \dots n]$  e il sottovettore  $A[i+1 \dots n]$  contiene gli  $(n-i)$  elementi più grandi di  $A[1 \dots n]$  ordinati.

CONCLUSIONE: quando il ciclo termina  $i=1$

1  $\quad i$   $\quad i+1$   $\quad n$   
max-heap | ordinati

INV  $\left[ \frac{1}{i} \right] =$  il sottovettore  $A[1]$  è un max-heap che contiene gli elementi più piccoli del vettore di partenza  $A[1 \dots n]$  e il sottovettore  $A[2 \dots n]$  contiene gli  $(n-1)$  elementi più grandi di  $A[1 \dots n]$  ordinati.

$\Rightarrow$  il vettore  $A$  è ordinato  $\square$

### COMPLICATITÀ

$$T_{\text{heap-sort}}(n) = O(n) + O(n \log n) = O(n \log n)$$

L'heap sort ordina in loco eseguendo, nel caso peggiore,  $O(n \log n)$  confronti!  
NON È STABILE

**⚠️** il Quick Sort, in MEDIA (con le ottimizzazioni) risulta essere più veloce dell'HEAP SORT **⚠️**

- **CODE DI PRIORITÀ**: struttura dati che mantiene un insieme dinamico  $S$  di elementi con valori chiave/peso
- code di MASSIMA priorità
  - (i) **INSERT** ( $S, x$ ) inserisce l'elemento  $x$  in  $S$ :  $S = S \cup \{x\}$
  - (ii) **MAXIMUM** ( $S$ ) restituisce l'elemento di  $S$  con la chiave più grande (sola lettura)
  - (iii) **EXTRACT-MAX** ( $S$ ) ~~elimina~~ restituisce l'elemento di  $S$  con la chiave più grande
  - (iv) **INCREASE-KEY** ( $S, x, v$ ) aumenta il valore della chiave di  $x$  al nuovo valore  $v$ , dove  $v >$  chiave di  $x$
- code di MINIMA priorità
  - (i) **INSERT** ( $S, x$ ) inserisce l'elemento  $x$  in  $S$ :  $S = S \cup \{x\}$
  - (ii) **MINIMUM** ( $S$ ) restituisce l'elemento di  $S$  con la chiave più piccola (sola lettura)
  - (iii) **EXTRACT-MIN** ( $S$ ) ~~elimina~~ e restituisce l'elemento di  $S$  con la chiave più piccola
  - (iv) **DECREASE-KEY** ( $S, x, v$ ) ~~decremente~~ il valore della chiave  $x$  al nuovo valore  $v$ , dove  $v <$  chiave di  $x$

• **HEAP-MAXIMUM** ( $A$ )  $T(n) = O(1)$

```

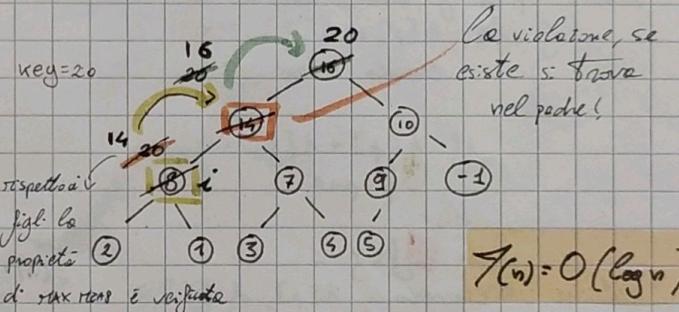
if A.heap-size < 1
  errore "heap underflow"
else
  return A[1]
  
```

• **HEAP-EXTRACT-MAX** ( $A$ )  $T(n) = O(\log n)$

```

if A.heap-size < 1
  errore "heap underflow"
else
  permette di mantenere le
  proprietà di albero completo
  max = A[1]  $\leftarrow$  proprieà di
  A[1] = A[A.heap-size]
  A.heap-size = A.heap-size - 1
  MAX-HEAPIFY ( $A, 1$ )
  return max
  
```

• **HEAP-INCREASE-KEY** ( $A, i, key$ ) \* nodo, è la posizione del vettore!!!
   
 $i$   $\leftarrow$  posizione del vettore!!!
   
 if key < A[i]
 errore "la nuova chiave è minore di quella consentita"
 A[i] = key  $\leftarrow$  se è  $i=1$  sono la radice e allora è ok!
 while  $i > 1$  AND A[parent(i)] < A[i]
 scambia A[i] e A[parent(i)]
  $i = parent(i)$  /\* prepara l'eventuale eccezione \*/



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

• **INV** = l'array  $A[1 \dots \text{heap-size}]$  soddisfa le proprietà di MAX-HEAP, tranne una possibile violazione:  $A[i]$  potrebbe essere più grande dell'elemento presente in  $A[\text{parent}(i)]$

CONCLUSIONE: il ciclo termina in due casi:

- (i)  $i=1$ : l'**INV** è soddisfatto perché è la radice e non possono esserci violazioni
- (ii)  $A[i] \leq A[\text{parent}(i)]$ : l'unica violazione potrebbe essere in  $i$ , ma, la grande del ciclo mi assicura che NON ci sono violazioni

• **MAX-HEAP-INSERT** ( $A, key$ )

$A.\text{heap-size} + 1$   
 $A[A.\text{heap-size}] = -\infty$   $\leftarrow$  se inserisco subito la key potrei  
 violare la condizione di max-heap  
 $(\text{heap-increase-key si aspetta che } key > A[A.\text{heap-size}])$

**HEAP-INCREASE-KEY** ( $A, A.\text{heap-size}, key$ )  $\leftarrow$  in realtà posso salvare il codice direttamente e non diamerla!

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

⚠️ un heap può svolgere ciascuna operazione con le code di PRIORITÀ nel tempo  $O(\log n)$  su un insieme dinamico di  $n$  elementi





scrivere una funzione che dato un ALBERO BINARIO di RICERCA  $T$ , memorizza le chiavi di  $T$  in un array e strutturato come MAX-HEAP e restituisce lo HEAP-SIZE

### • COSTRUISCI-MAX-HEAP ( $T, A$ )

if  $T.root = \text{NIL}$

return 0

else

$u = \text{tree\_maximum}(T.root)$

$i = 1$

while  $u \neq \text{NIL} \leftarrow \Theta(n)$

$A[i] = u.key$

$i = i + 1$

$u = \text{tree\_predecessor}(u)$

return  $i - 1$



visito l'elenco partendo dal rassero e vedo al predecessore, ottenendo un array ordinato in senso non crescente, quindi un max-heap

MAX-HEAP!

9	7	6	5	4	3	1
---	---	---	---	---	---	---

se un array è ordinato in senso non crescente  $\Rightarrow$  è un max heap

progettare un algoritmo che ricevuto un intero  $k \leq n$  e un array  $v$  non ordinato di  $n$  elementi distilire restituisce il  $k$ -esimo elemento più piccolo di  $v$ . (3 algoritmi con diversi costi)

### (i) SOLUZIONE COSTO $\Theta(n \log n)$ $\Rightarrow$ « ordinamento »

- CERCA- $k$ -ESIMO ( $v, k$ )

mergesort ( $v, 1, v.length$ )

return  $v[k]$

operazione preparatoria

### (ii) SOLUZIONE COSTO $O(n + k \log n)$ $\Rightarrow$ $O(n) + k \log n$ $\Rightarrow$ eseguo $k$ volte un'operazione

- CERCA- $k$ -ESIMO ( $v, k$ )

build-min-heap ( $v$ )  $\leftarrow \Theta(n)$

for  $i = 1$  to  $k$   
 $x = \text{heap\_extract-min}(v)$  }  $k$  volte  $\log n$ !

return  $x$

### (iii) SOLUZIONE COSTO $O(n \log k)$ $\Rightarrow$ « code di priorità »

- CERCA- $k$ -ESIMO ( $v, k$ )

heap = nuovo max-heap ( $v$ ) /\* crea un max-heap root d'dimensione  $k$  \*/  $\xrightarrow{\text{length} = k}$  heap size = 0

for  $i = 1$  to  $v.length - n$  volte

if heap.heap-size <  $k$

max-heap-insert (heap,  $v[i]$ )  $\leftarrow O(\log k)$

else

if heap-maximum (heap)  $> v[i]$   $\leftarrow O(1)$

$O(\log k)$  heap-extract-max (heap)

$O(\log n)$  max-heap-insert (heap,  $v[i]$ ) se il  $k$ -esimo elemento più piccolo è

return heap-maximum (heap)



$\xrightarrow{\text{se il } k\text{-esimo elemento più piccolo è}}$   
 $\xrightarrow{\text{il max non sarà in quel el. più piccolo}}$

$\xrightarrow{\text{posso scartare il max del mio heap perché ho trovato un elemento } v[i] \text{ nel primo el. più piccolo e sicuramente il max non sarà in quel el. più piccolo}}$

sia  $T$  un BST contenente  $n$  nodi, sia  $v$  una chiave che compare in  $T$  e sia  $x$  un nodo di  $T$

i. La ricerca di  $v$  ha costo  $O(n)$   $\Rightarrow$  VERO (giustificare)

- La ricerca di  $v$  in  $T$  ha costo  $O(h)$ , dove  $h$  è l'altezza di  $T$ . Nel caso peggiore, in cui  $T$  sia completamente sbilanciato  $h = n-1$ , di conseguenza  $O(n)$

ii. il predecessore di  $x$  si trova nel sottoalbero radicato in  $x.left$   $\Rightarrow$  FALSO (controesempio)

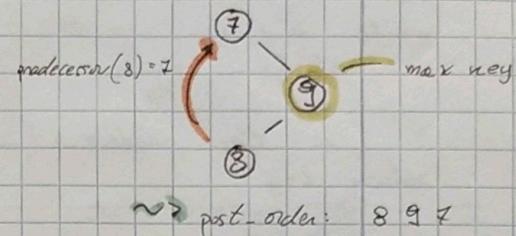
- il predecessore di  $8$  non si trova in  $x.left$ !

iii. La chiave massima si trova in una foglia  $\Rightarrow$  FALSO

-  $9$  è la chiave massima ma non è una foglia

iv. eseguendo una visita in post orden d' $T$  le chiavi sono stampate in ordine decrescente  $\Rightarrow$  FALSO

- la visita in post orden produce  $8 9 7$ !



Sia  $v$  un array di  $n$  interi distinti dove  $v$  è dispari,  $v$  è detto ALTERNANTE se vale la seguente condizione

$$v[1] > v[2] < v[3] > v[4] < \dots > v[n-1] < v[n]$$

Scrivere una procedura che renda  $v$  alternante!

• ALTERNA ( $v$ )

```
for i=2 to v.length - 1 step=2
    if v[i] > v[i-1]
        • Scambia v[i] e v[i-1]
    else
        if v[i] > v[i+1]
            • scambia v[i] e v[i+1]
```

altra soluzione  
ordina  
scambia gli elementi  
ed è a due  
 $T(n) = \Theta(n \log n)$

! è una PROPRIETÀ  
LOCALE fra i 3 valori !

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

scrivere un algoritmo INTERSECT ( $H_1, H_2$ ) che, dati due min heap  $H_1$  e  $H_2$  contenenti rispettivamente  $n_1$  e  $n_2$  interi positivi, ritorna in output un nuovo min heap contenente tutti e solo gli elementi che appartengono sia ad  $H_1$  che ad  $H_2$ .

• INTERSECT ( $H_1, H_2$ )

HEAP-RIS = new\_min\_heap(min{ $H_1.\text{heapsize}, H_2.\text{heapsize}$ })

while NOT is-heapEmpty( $H_1$ ) and NOT is-heapEmpty( $H_2$ )

    min 1 = heap-minimum( $H_1$ )

    min 2 = heap-minimum( $H_2$ )

    if min 1 = min 2

        heap-insert(HEAP-RIS, min 1)  $\Theta(\log(\min\{n_1, n_2\}))$

        heap-extract-min( $H_1$ )  $\Theta(\log n_1)$

        heap-extract-min( $H_2$ )  $\Theta(\log n_2)$

    else

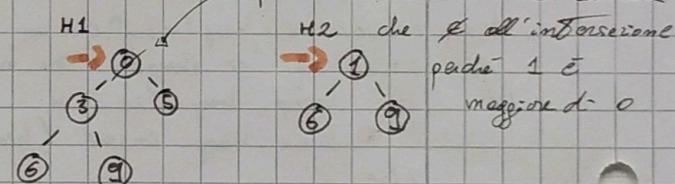
        if min 1 < min 2

            heap-extract-min( $H_1$ )  $\Theta(\log n_1)$

    else

        heap-extract-min( $H_2$ )  $\Theta(\log n_2)$

return HEAP-RIS



escludo elementi leggendo  
sempre la testa dei  
due min heap!

$$T(n_1 + n_2) = O((n_1 + n_2) \cdot (\log n_1 + \log n_2))$$

dato un vettore  $v$  di numeri interi distinti, se  $i < j$  e  $v[i] > v[j]$ , allora la coppia  $(i, j)$  è detta INVERSIONE di  $v$ . Utilizzando la tecnica divide et impera scrivere una ricorsiva efficiente che restituisce il numero di inversioni.

$$(1, 5) (2, 5) (3, 5) (4, 5)$$

$$(3, 4)$$

$\Rightarrow$  5 inversioni

2	3	8	6	1
1	2	3	4	5

modificare il merge sort!!!

## ALBERI DI DECISIONE

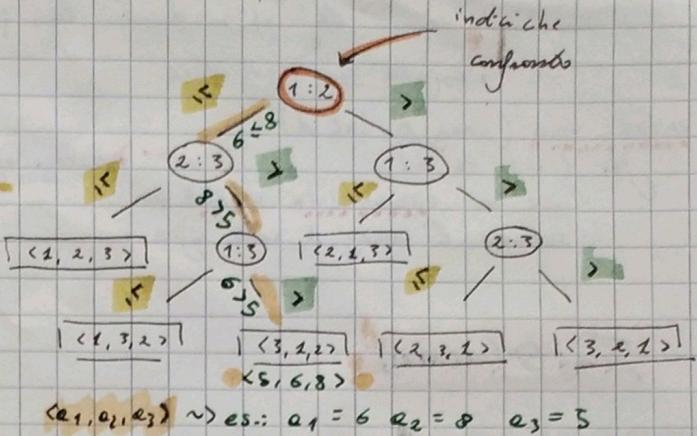
Tengono traccia dei confronti trascurando altre operazioni

per input  $a_1, \dots, a_n$ , ogni nodo è etichettato da

$i:j$  con  $i, j \in \{1, \dots, n\}$

confronto  $a_i < a_j$

il sottosalvo sx da i successivi confronti se  $a_i \leq a_j$   
il sottosalvo dx da i successivi confronti se  $a_i > a_j$



Ogni foglia è una permutazione  $\pi(a_1), \dots, \pi(a_n)$  t.c.  
 $a_{\pi(1)} < a_{\pi(2)} < \dots < a_{\pi(n)}$

$\Delta \# \text{foglie} \geq n!$

Dato un qualsiasi algoritmo di ordinamento per confronti, posso costruire un albero di decisione  $T_n$ , esso modella tutte le tracce (ogni carico) di esecuzione!

$\rightarrow$  TEMPO DI ESECUZIONE (numero di confronti): lunghezza di un cammino nell'albero (worst case = altezza)

qualsiasi algoritmo di ordinamento basato sui confronti richiede  $\Omega(n \log n)$  confronti nel caso peggiore!

- dimostrazione:

ve determinate l'altezza di un albero di decisione, dove ogni permutazione appare come una foglia.  
si consideri un albero di decisione di altezza  $h$  e  $2^h$  foglie che corrisponde a un ordinamento per confronti di  $n$  elementi.

allora  $h \geq n!$  e  $h \leq 2^h \Rightarrow n! \leq h \leq 2^h \Rightarrow h \geq \log n!$

Approssimazione di Stirling:  $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + O(\frac{1}{n}))$

estendere di  $n$  all'inf. donca:

$$\Rightarrow h \geq \log n! \Rightarrow \log\left(\frac{n}{e}\right) = n \log\left(\frac{n}{e}\right) = n(\log n - \log e) = h = \Theta(n \log n)$$

## ALGORITMI DI ORDINAMENTO NON BASATI SUI CONFRONTI

### (ii) COUNTING SORT

ASSUNZIONE: i numeri da ordinare sono interi in un intervallo (insieme limitato) che vale da 0 a  $k$  ( $k \in \mathbb{N}$ )

COUNTING-SORT (array A, array B, int n, int k)

for ( $i=0$  to  $n$ ) /\* elenco C di k+1 elementi \*/

$c[i] = 0$

(a) for ( $j=1$  to  $n$ )  $\rightarrow c[k-A[j]]++$

(b) for ( $i=1$  to  $n$ )  $\rightarrow c[i] = c[i] + c[i-1]$   $\rightarrow$  v-volti

(c) for  $j=n$  down to 1  $\rightarrow$  v-volti

$B[c[A[j]]] = A[j]$   $\rightarrow$  v-volti

$c[A[j]] = c[k-A[j]]$

A  $\begin{matrix} 3 & 0 & 2 & 3 & 2 & 1 \end{matrix}$

(a) C  $\begin{matrix} 1 & 0 & 2 & 2 \\ 0 & 1 & 2 & 3 \end{matrix} \rightarrow c[:] = \{x \in \{1..n\} | A[x] = i\}$

(b) C  $\begin{matrix} 1 & 1 & 3 & 5 \\ 0 & 1 & 2 & 3 \end{matrix} \rightarrow c[:] = \{x \in \{1..n\} | A[x] \leq i\}$

(c) C  $\begin{matrix} 1 & 1 & 2 & 5 \\ 0 & 1 & 2 & 3 \end{matrix}$

• C  $\begin{matrix} 1 & 1 & 2 & 5 \\ 0 & 1 & 2 & 3 \end{matrix}$

• C  $\begin{matrix} 1 & 1 & 2 & 4 \\ 0 & 1 & 2 & 3 \end{matrix}$

• C  $\begin{matrix} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 3 \end{matrix}$

• C  $\begin{matrix} 0 & 1 & 1 & 4 \\ 0 & 1 & 2 & 3 \end{matrix}$

• C  $\begin{matrix} 0 & 1 & 1 & 3 \\ 0 & 1 & 2 & 3 \end{matrix}$

B  $\begin{matrix} 0 & 2 & 2 & 3 & 3 \\ 0 & 1 & 2 & 3 & 4 \end{matrix}$

la mette qui:

perché ci sono 3 elementi  $\leq 2$ !

ordine senso NON crescente

\* C  $\begin{matrix} 1 & 1 & 1 & 3 \\ 0 & 1 & 2 & 3 \end{matrix}$

input:  $A[1..n]$ , dove  $A[j] \in [0..k]$   $\forall j \in \{1, \dots, n\}$

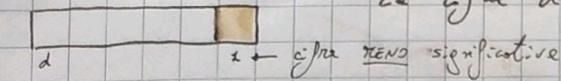
output:  $B[1..n]$  ordinato

mem. aux:  $c[0..k]$  vettore delle occorrenze

STABILITÀ: se sono delle fine dell'array!

$T_{cs}(n) = \Theta(n+k)$  se  $k = O(n) \Rightarrow T(n) = \Theta(n)$   
ma se  $k = n^2 \Rightarrow T(n) = \Theta(n^2)$

(iii) RADIX-SORT: ordinare elementi con  $d$  cifre, dove la cifra 1 è la meno significativa e la cifra  $d$  è la più significativa



RADIX-SORT (array A, int d)

for ( $i=1$  to  $d$ )

usa un ORDIMENTO STABILE per ordinare l'array A sulla cifra i

$$T(n) = \Theta(d(n+k))$$

$n=10$	326	690	706	326
es. COUNTING SORT	$d=3$	453	704	688
		608	453	435
		835	835	326
		435	435	453
		704	326	608
		690	608	690
				835

i duplicati sono  
seletti perché sono  
delle i-esime cifre!

CORRETEZZA (sull' i-esima colonna da ordinare)

- $i=1$  ordina la colonna
- $i \neq 1$  assumo che le cifre delle colonne 1 alla  $i-1$  siano ordinate. dimostro che un ALGORITMO STABILE sulla colonna  $i$  lascia le colonne  $1, 2, \dots, i$  ordinate
  - se due cifre sulla colonna  $i$  sono uguali, restano ordinate per stabilità e, per ip. induttiva, sono ordinate.
  - se due cifre sulla colonna  $i$  sono diverse, allora l'algoritmo di ordinamento sulla colonna  $i$  le ordina e le mette in posizione corretta.

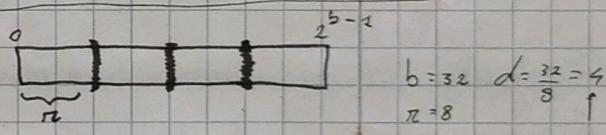
### COMPLICATEZZA

- $n$  numeri
  - $d$  cifre
  - max  $n$  valori per cifra
- $$\Rightarrow \Theta(d(n+k)) \rightsquigarrow \Theta(n+k) \text{ dell'algoritmo stabile!}$$

### OSSERVAZIONI

se  $n = O(n)$   $\Rightarrow T(n) = \Theta(nd)$ ; se  $d$  è costante  $\Rightarrow T(n) = \Theta(n)$   
NON è in loco! ha bisogno d' memoria aggiuntiva non costante!

Suppongo d' avere  $n$  interi, ciascuno de  $b$  bits  
e diviso in cifre da  $n$  bit:  $d = \frac{b}{n}$



soddisfro ogni intero in  $\lceil \frac{b}{n} \rceil$  cifre con ogni cifra lunga n bit

$$n = 2^k \text{ cifre} \in [0 \dots 2^n - 1] \rightsquigarrow \Theta(d(n+k)) = \Theta\left[\left(\frac{b}{n}\right)(n+2^k)\right]$$

scelgo  $n \leq b$  tale che  $\frac{b}{n}(n+2^k)$  sia minima:

$n$  minima!

i.  $b < \lfloor \log_2 n \rfloor \Rightarrow$  per qualsiasi valore di  $n \leq b$ :  $(n+2^k) = \Theta(n)$  ( $n \leq b \Rightarrow 2^n \leq 2^b \leq n$ )

ii.  $n = b \Rightarrow \left(\frac{b}{b}(n+2^k)\right) = \Theta(n)$

iii.  $b \geq \lfloor \log_2 n \rfloor \Rightarrow \begin{cases} \frac{b}{n} n & \rightsquigarrow n \text{ grande} \\ \frac{b}{n} 2^n & \rightsquigarrow n \text{ non deve crescere troppo} \end{cases}$

$$\rightsquigarrow \text{se } n = \Theta(b) \Rightarrow \Theta\left(\frac{bn}{\log n}\right)$$