Algoritmi e Strutture Dati (25/10/2021)

* Ordinamento in tempo lineare

limite inferiore $\Omega(n \log n)$

vincoli: range / distribuzione

limite inferiore: $\Omega(n)$

* Counting Sort

assunzioni su elementi da ordinare

- interi
- in $[0, k]$ con k noto

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

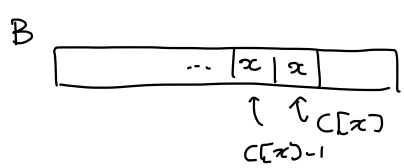
OUTPUT: $B[1..n]$ copia ordinata di A

↓

array $C[0..k]$ di supporto

Counting Sort (A, B, k)

```
(4)(k) {  C[0..k] ← 0
(4)(n) {  for j = 1 to A.length
              C[A[j]]++                  // C[x] = num. di occorrenze di x in A
(4)(k) {  for i = 1 to k
              C[i] = C[i-1] + C[i]       // C[x] = num. di occorrenze di elem ≤ x in A
(4)(n) {  for j = A.length downto 1
              B[C[A[j]]] = A[j]
              C[A[j]]--
```



Esempio:

	1	2	3	4	5	6	7	8
A	1	4	2	0	1	2	0	2
B	0	0	1	1	2	2	2	4

$k=4$

	0	1	2	3	4
C	2	2	3	0	1
	0	2	4	7	7

Costo?

$\Theta(k+n)$

$k = O(1)$ $\rightarrow$ $\Theta(n)$

$k = O(n)$

Perché inizio a scrivere A dal fondo nell'ultimo ciclo?

	1	2	3	4	5	6	7	8
A	1	4	2	0	1	2	0	2
B	0	0	1	1	2	2	2	4

$k=4$

	0	1	2	3	4
C	2	2	3	0	1
	0	1	2	3	4
	0	3	4	7	7

$A[i] = x$ $\qquad$ $x.key$

Per avere un algoritmo stabile!

→ Insertion Sort, MergeSort } stabili

→ QuickSort, HeapSort } non stabili

* semplificazione:

```
Counting Sort (A, B, k)
    C[0..k] ← 0
    for j = 1 to A.length
        C[A[j]]++

    i = 1
    for j = 0 to k
        while C[j] > 0
            B[i] = j
            C[j]--
            i++
```

OK solo se non ci sono dati satellite

* <u>spazio</u>

$$\Theta(n + k)$$

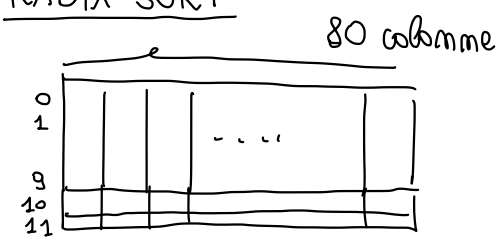
n ↑ spazio per B

k ↖ spazio per C

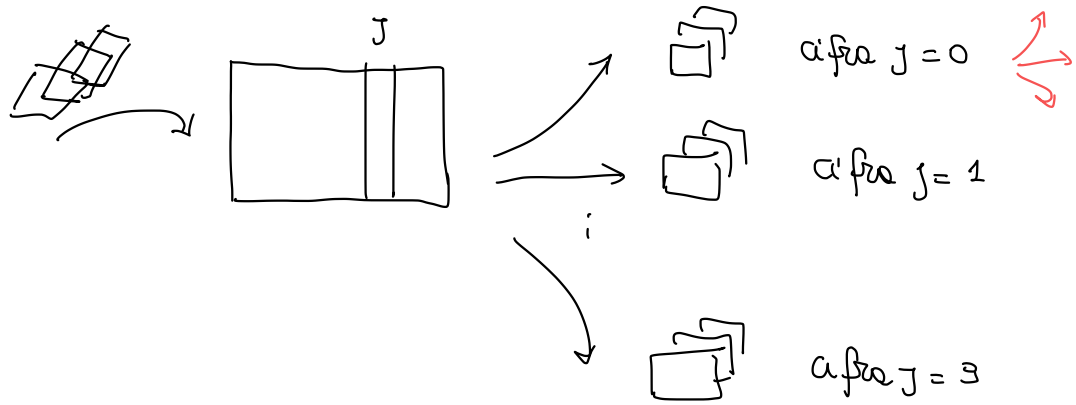
può essere un problema

# bit	dim. C
8	$2^8 \times 1$ byte $\simeq$ 256 byte
16	$2^{16} \times 2$ byte $\simeq$ 128 Kbyte
32	$2^{32} \times 4$ byte $\simeq$ 16 GB
64	$2^{64} \times 8$ byte $\simeq$ 128 exabyte (10^{18})

* <u>RADIX SORT</u>



Hermann Hollerith



1ª idea: ordino a partire dalla più significativa

2ª idea: ordino dalla cifra meno significativa alla più significativa

IBM

ESEMPIO

3	2	9	←	7	2	0	-	→	7	2	0
4	5	7	←	3	5	5	-	→	3	2	9
6	5	7	←	4	5	7	-	→	8	3	9
8	3	9	←	6	5	7	-	→	4	3	9
4	3	9	←	3	2	9	-	→	3	5	5
7	2	0	←	8	3	9	-	→	4	5	7
3	5	5	←	4	3	9	-	→	6	5	7

3	2	9
3	5	5
4	3	9
4	5	7
6	5	7
7	2	0
8	3	9

input $A[1..n]$ con $A[j]$ di d cifre in base b

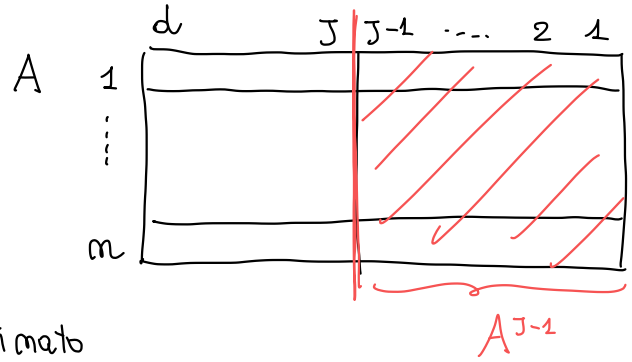
$\in [0, b^d - 1]$

$A[j] = \underset{\text{cifra più signif.}}{a_d} \; a_{d-1} \dots a_2 \; \underset{\text{meno signif.}}{a_1}$

Radix Sort (A, d)
  for $j = 1$ to d
    ordino A rispetto alla cifra j
    con un algoritmo stabile

// A è ordinato nelle cifre $j-1 \dots 1$

A^{j-1} ordinato



INV: A^{J-1} ordinato

- inizio: $J = 1$ $\quad A^{J-1} = A^0$ vacuamente ordinato

- mantenimento: $\boxed{A^{J-1} \text{ ordinato}}$ e ordino rispetto alla Jma cifra con algoritmo stabile

$\rightsquigarrow$ A^J ordinato

$\forall\, i, i' \qquad i \leq i' \qquad A^{J-1}[i] \leq A^{J-1}[i']$

ordiniamo con algoritmo stabile rispetto alla cifra J

e vorremmo A^J ordinato

$\forall\, i, i' \qquad i \leq i' \qquad A^J[i] \leq A^J[i']$

$$a_J \underbrace{a_{J-1} .. a_1}_{A^{J-1}[i]} \qquad a'_J \underbrace{a'_{J-1} .. a'_1}_{A^{J-1}[i']}$$

2 possibilità:

① $a_J \neq a'_J \Rightarrow a_J < a'_J \Rightarrow$

$$a_J\, a_{J-1} .. a_1 < a'_J\, a'_{J-1} .. a'_1$$

② $a_J = a'_J \Rightarrow$ dato che abbiamo usato un algoritmo stabile

$\Rightarrow A^{J-1}[i] \leq A^{J-1}[i']$

$$\Rightarrow \underbrace{a_J\, A^{J-1}[i]}_{A^J[i]} \leq \underbrace{a'_J\, A[i']}_{A^J[i']}$$

conclusione: $J = d + 1$

$\rightsquigarrow A^{J-1} = A^d = A$ ordinato

* <u>complessità</u> :

<u>Radix Sort</u> (A, d)

for $j = 1$ to d

ordino A rispetto alla cifra j

con un algoritmo stabile

n = # elem. dell'array

d cifre

b base

supponiamo che sia il Counting Sort

$\Theta(n + b)$

$\times\ d$ iterazioni

costo totale Radix Sort

$\Theta(d\,(n+b))$

?

$b = O(1)$ oppure $b = O(n)$

$\rightsquigarrow \Theta(d\ n)$

se $d = O(1)$ $\rightarrow \Theta(n)$

<u>spazio</u>: $\Theta(n + b)$

output

n elem. distinti

$d \geq \log n$