

Algoritmi e Strutture Dati

Luciano Gualà

guala@mat.uniroma2.it

www.mat.uniroma2.it/~guala

Sommario

- Delimitazioni inferiori e superiori (di algoritmi e problemi)
- Quanto velocemente si possono ordinare n elementi?
 - una soglia (asintotica) di velocità sotto la quale non si può scendere: un **lower bound**
 - (per una classe di algoritmi ragionevoli - quelli basati su confronti)
 - una tecnica elegante che usa gli **alberi di decisione**
- E se si esce da questa classe di algoritmi?
 - **integer sort** e **bucket sort** (per interi "piccoli")
 - **radix sort** (per interi più "grandi")

Delimitazioni
inferiori e superiori
(di algoritmi e problemi)

Delimitazioni superiori (*upper bound*)

Definizione

Un algoritmo A ha complessità (costo di esecuzione) $O(f(n))$ su istanze di dimensione n rispetto ad una certa risorsa di calcolo, se la quantità r di risorsa **sufficiente** per eseguire A su una qualunque istanza di dimensione n verifica la relazione $r(n) = O(f(n))$

Definizione

Un problema P ha una complessità $O(f(n))$ rispetto ad una risorsa di calcolo se **esiste** un algoritmo che risolve P il cui costo di esecuzione rispetto a quella risorsa è $O(f(n))$

Delimitazioni inferiori (*lower bound*)

Definizione

Un algoritmo A ha complessità (costo di esecuzione) $\Omega(f(n))$ su istanze di dimensione n e rispetto ad una certa risorsa di calcolo, se la quantità r di risorsa usata da A nel caso peggiore su istanze di dimensione n verifica la relazione $r(n) = \Omega(f(n))$

Definizione

Un problema P ha una complessità $\Omega(f(n))$ rispetto ad una risorsa di calcolo se **ogni algoritmo** che risolve P ha costo di esecuzione nel caso peggiore $\Omega(f(n))$ rispetto quella risorsa

Ottimalità di un algoritmo

Definizione

Dato un problema P con complessità $\Omega(f(n))$ rispetto ad una risorsa di calcolo, un algoritmo che risolve P è **ottimo** se ha costo di esecuzione $O(f(n))$ rispetto a quella risorsa

complessità temporale del problema dell'ordinamento

- Upper bound: $O(n^2)$
 - Insertion Sort, Selection Sort, Quick Sort, Bubble Sort
- Un upper bound migliore: $O(n \log n)$
 - Merge Sort, Heap Sort
- Lower bound: $\Omega(n)$
 - banale: ogni algoritmo che ordina n elementi li deve almeno leggere tutti

Abbiamo un **gap di $\log n$** tra upper bound e lower bound!

Possiamo fare meglio?

Sui limiti della velocità: una
delimitazione inferiore
(lower bound) alla
complessità
del problema



Ordinamento per confronti

Dati due elementi a_i ed a_j , per determinarne l'ordinamento relativo effettuiamo una delle seguenti operazioni di confronto:

$$a_i < a_j ; a_i \leq a_j ; a_i = a_j ; a_i \geq a_j ; a_i > a_j$$

Non si possono esaminare i valori degli elementi o ottenere informazioni sul loro ordine in altro modo.

Notare: Tutti gli algoritmi citati prima sono algoritmi di ordinamento per confronto.

Teorema

Ogni algoritmo basato su confronti che ordina n elementi deve fare nel caso peggiore $\Omega(n \log n)$ confronti.

Nota: il #di confronti che un algoritmo esegue è un **lower bound** al #di passi elementari che esegue

Corollario

Il **Merge Sort** e l'**Heap Sort** sono algoritmi **ottimi** (almeno dentro la classe di algoritmi basati su confronti).

Uno strumento utile: albero di decisione

Gli algoritmi di ordinamento per confronto possono essere descritti in modo astratto in termini di **alberi di decisione**.

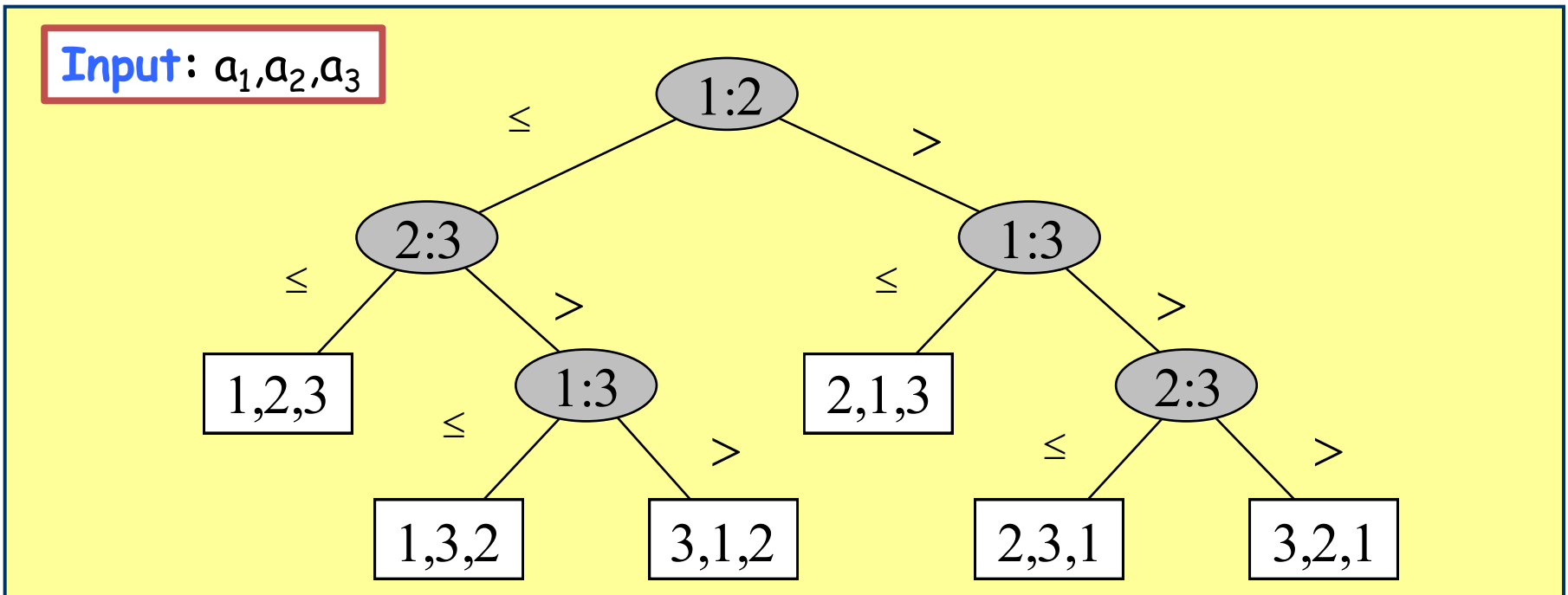
Un generico algoritmo di ordinamento per confronto lavora nel modo seguente:

- confronta due elementi a_i ed a_j (ad esempio effettua il test $a_i \leq a_j$);
- a seconda del risultato - riordina e/o decide il confronto successivo da eseguire.

Albero di decisione - Descrive i confronti che l'algoritmo esegue quando opera su un input di una **determinata dimensione**. I movimenti dei dati e tutti gli altri aspetti dell'algoritmo vengono ignorati

Alberi di decisione

- Descrive le diverse sequenze di confronti che A potrebbe fare su istanze di dimensione n
- Nodo interno (non foglia): $i:j$
 - modella il confronto tra a_i e a_j
- Nodo foglia:
 - modella una risposta (output) dell'algoritmo: permutazione degli elementi



Osservazioni

- L'albero di decisione **non è** associato ad un problema
- L'albero di decisione **non è** associato **solo** ad un algoritmo
- L'albero di decisione è associato ad un **algoritmo** e a una **dimensione dell'istanza**
- L'albero di decisione descrive le diverse sequenze di confronti che un certo algoritmo può eseguire su istanze di una **data dimensione**

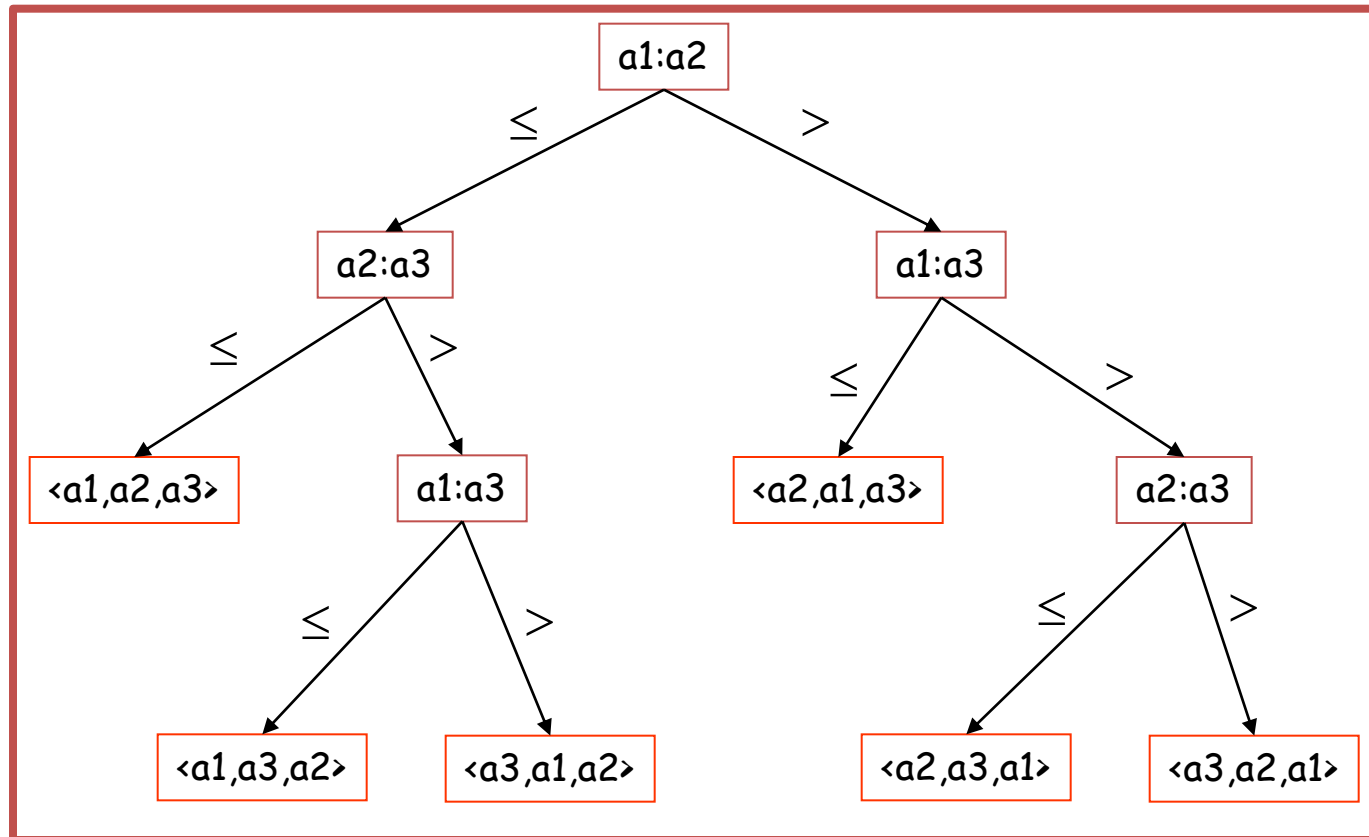
Esempio

Fornire l'albero di decisione del seguente algoritmo per istanze di dimensione 3.

InsertionSort2 (A)

1. **for** k=1 **to** n-1 **do**
2. x = A[k+1]
3. j = k
4. **while** j > 0 e A[j] > x **do**
5. A[j+1] = A[j]
6. j = j-1
7. A[j+1]=x

...eccolo:



Proprietà

- Per una particolare istanza, i confronti eseguiti dall'algoritmo su quella istanza rappresentano un **cammino radice - foglia**
- L'algoritmo segue un cammino diverso a seconda delle caratteristiche dell'istanza
 - **Caso peggiore**: cammino più lungo
- Il numero di confronti nel caso peggiore è pari **all'altezza dell'albero di decisione**
- Un albero di decisione di un algoritmo che risolve il problema dell'ordinamento di n elementi deve avere necessariamente **almeno $n!$ foglie**

Lemma

Un albero binario T con k foglie, ha altezza almeno $\log_2 k$

dim (per induzione sul k)

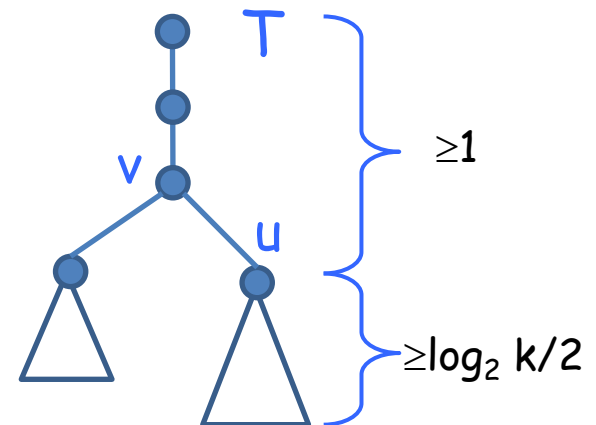
caso base: $k=1$ altezza almeno $\log_2 1=0$

caso induttivo: $k>1$

considera il nodo interno v più vicino alla radice che ha due figli (v potrebbe essere la radice). nota che v deve esistere perché $k>1$.

v ha almeno un figlio u che è radice di un (sotto)albero che ha almeno $k/2$ foglie e $< k$ foglie.

T ha altezza almeno
 $1 + \log_2 k/2 = 1 + \log_2 k - \log_2 2 = \log_2 k$



Il lower bound $\Omega(n \log n)$

- Consideriamo l'albero di decisione di un qualsiasi algoritmo che risolve il problema dell'ordinamento di n elementi
- L'altezza h dell'albero di decisione è almeno $\log_2 (n!)$
- Formula di Stirling: $n! \approx (2\pi n)^{1/2} \cdot (n/e)^n$

$$\begin{aligned} h &\geq \log_2(n!) > \log_2 (n/e)^n = \\ &= n \log_2 (n/e) = \\ n! &> (n/e)^n &= n \log_2 n - n \log_2 e = \\ &= \Omega(n \log n) \end{aligned}$$

Esercizio

Dimostrare usando la tecnica dell'albero di decisione che l'algoritmo di pesatura che esegue (nel caso peggiore) $\lceil \log_3 n \rceil$ pesate per trovare la moneta falsa fra n monete è ottimo.

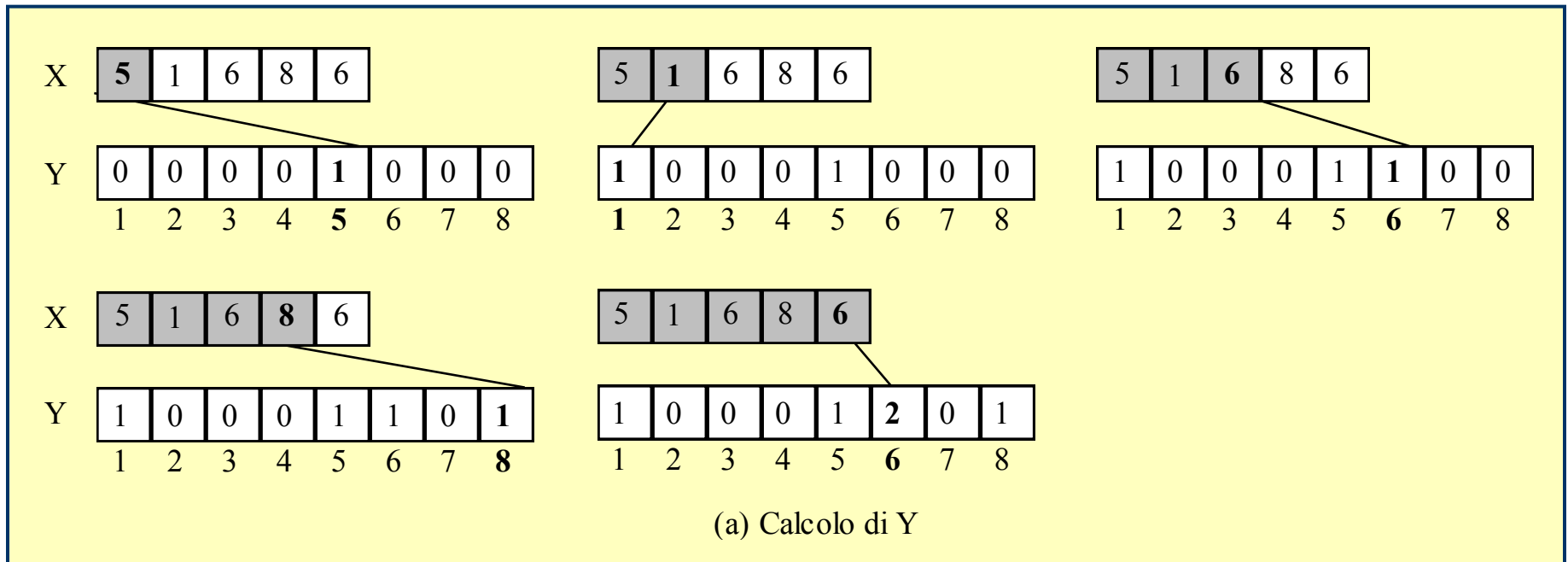
può un algoritmo basato su confronti ordinare n interi piccoli, diciamo compresi fra 1 e $k=O(n)$, in (asintoticamente) meno di $n \log n$?

...no, la dimostrazione funziona anche sotto questa ipotesi!

IntegerSort: fase 1

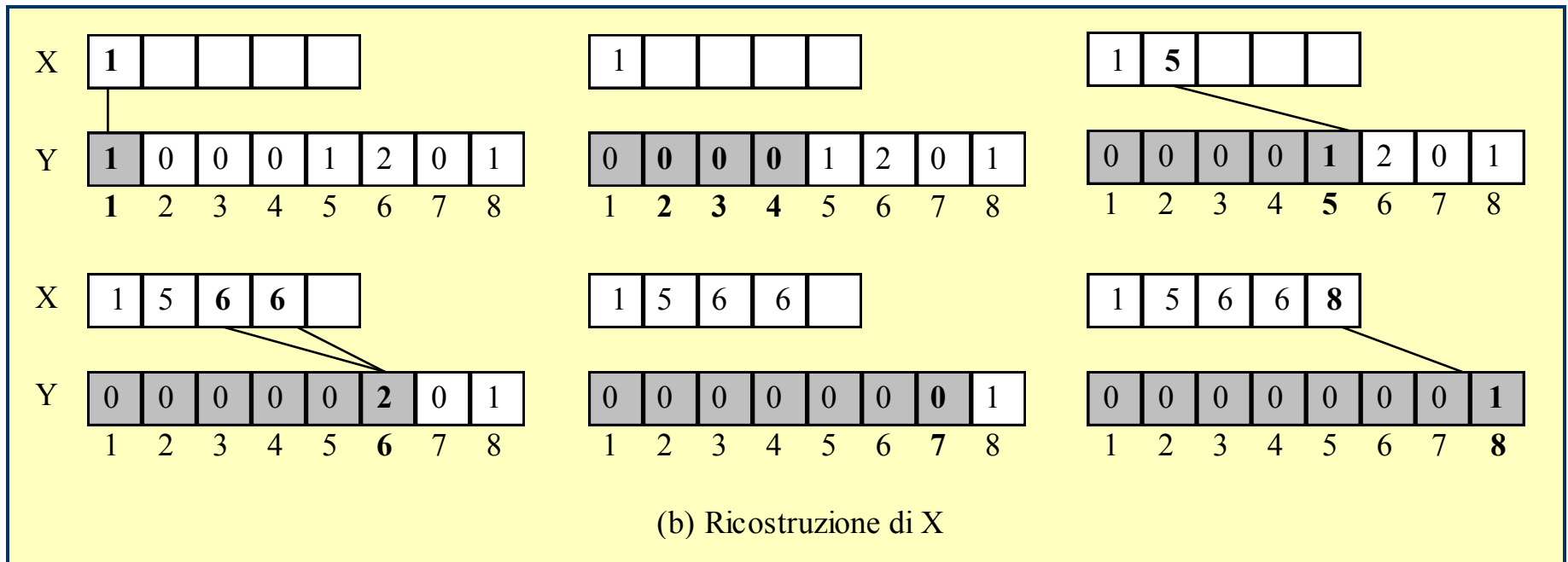
Per ordinare n interi con valori in $[1, k]$

Mantiene un array Y di k contatori tale che $Y[x] =$ numero di volte che il valore x compare nell'array di input X



IntegerSort: fase 2

Scorre Y da sinistra verso destra e, se $Y[x]=k$, scrive in X il valore x per k volte



IntegerSort (X, k)

1. Sia Y un array di dimensione k } $O(1)$ - tempo costante
2. **for** i=1 **to** k **do** Y[i]=0 } $O(k)$
3. **for** i=1 **to** n **do** incrementa Y[X[i]] } $O(n)$
4. j=1 } $O(1)$
5. **for** i=1 **to** k **do** } $O(k)$
6. **while** (Y[i] > 0) **do** }
7. X[j]=i } per i fissato
8. incrementa j } #volte eseguite
9. decrementa Y[i] } è al più $1+Y[i]$ $\Rightarrow O(k+n)$

$$\sum_{i=1}^k (1+Y[i]) = \sum_{i=1}^k 1 + \sum_{i=1}^k Y[i] = k + n$$

IntegerSort: analisi

- Tempo $O(1)+O(k)=O(k)$ per inizializzare Y a 0
- Tempo $O(1)+O(n)=O(n)$ per calcolare i valori dei contatori
- Tempo $O(n+k)$ per ricostruire X



$O(n+k)$

Tempo lineare se $k=O(n)$

Contraddice il lower bound di $\Omega(n \log n)$?

No, perché l'Integer Sort non è un algoritmo basato su confronti!

Una domanda

Che complessità temporale ha l'IntegerSort quando $k = \omega(n)$,
per esempio $k = \Theta(n^c)$, con c costante?

$$\dots T(n) = \Theta(n^c) \dots$$
$$\dots = \omega(n \log n) \text{ per } c > 1 \dots$$

BucketSort

Per ordinare n record con chiavi intere in $[1, k]$

- **Esempio:** ordinare n record con campi:
 - nome, cognome, anno di nascita, matricola,...
- si potrebbe voler ordinare per matricola o per anno di nascita

Input del problema:

- n record mantenuti in un array
- ogni elemento dell'array è un record con
 - campo chiave (rispetto al quale ordinare)
 - altri campi associati alla chiave (informazione satellite)

BucketSort

- Basta mantenere un array di liste, anziché di contatori, ed operare come per `IntegerSort`
- La lista `Y[x]` conterrà gli elementi con chiave uguale a `x`
- Concatenare poi le liste

Tempo $O(n+k)$ come per `IntegerSort`

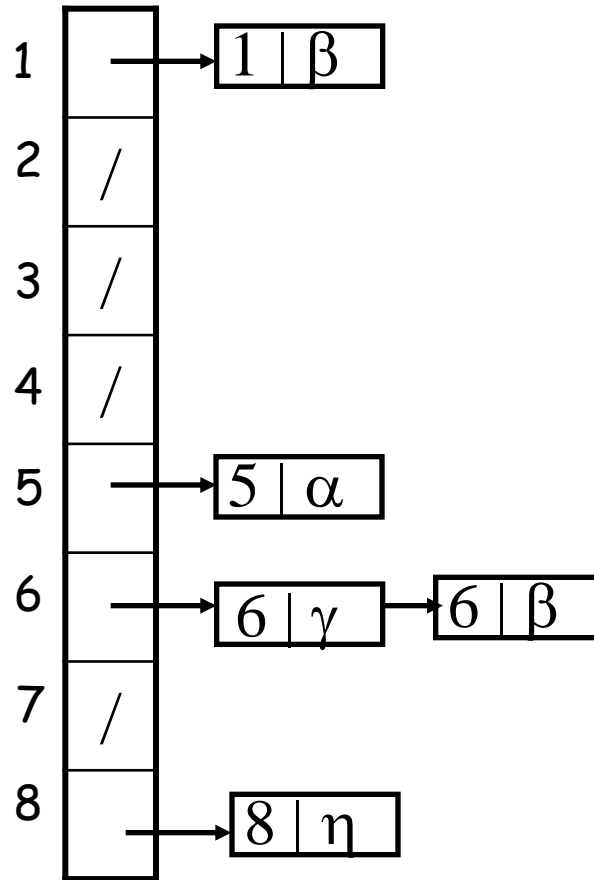
esempio

X

chiave info satellite

1	5	α
2	1	β
3	6	γ
4	8	η
5	6	β

γ



X (ordinato)

chiave info satellite

1	1	β
2	5	α
3	6	γ
4	6	β
5	8	η

BucketSort (X, k)

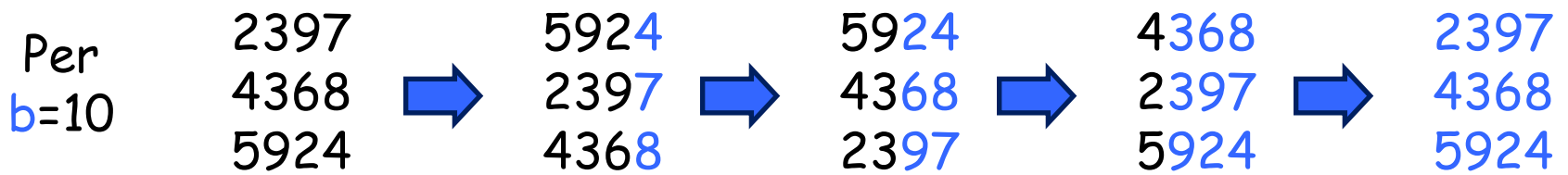
1. Sia Y un array di dimensione k
2. **for** i=1 **to** k **do** Y[i]=lista vuota
3. **for** i=1 **to** n **do**
4. **if** (chiave(X[i]) \notin [1,k]) **then errore**
5. **else** appendi il record X[i] alla lista Y[chiave(X[i])]
6. **for** i=1 **to** k **do**
7. copia ordinatamente in X gli elementi della lista Y[i]

Stabilità

- Un algoritmo è **stabile** se preserva l'ordine iniziale tra elementi con la stessa chiave
- domanda: il **BucketSort** è stabile?
- Il **BucketSort** è stabile se si appendendo gli elementi di **X in coda** alla opportuna lista **Y[i]**

RadixSort

- Rappresentiamo gli elementi in base b , ed eseguiamo una serie di BucketSort
- Partiamo dalla cifra meno significativa verso quella più significativa



Correttezza

- Se x e y hanno una diversa t -esima cifra, la t -esima passata di **BucketSort** li ordina
- Se x e y hanno la stessa t -esima cifra, la proprietà di stabilità del **BucketSort** li mantiene ordinati correttamente



Dopo la t -esima passata di **BucketSort**, i numeri sono correttamente ordinati rispetto alle t cifre meno significative

Tempo di esecuzione

- $O(\log_b k)$ passate di bucketsort
- Ciascuna passata richiede tempo $O(n+b)$



$$O((n+b) \log_b k)$$

$$\log_2 k = \log_n k \log_2 n$$

Se $b = \Theta(n)$, si ha $O(n \log_n k) = O \left(n \frac{\log k}{\log n} \right)$

➔ Tempo **lineare** se $k = O(n^c)$, c costante

esempio

- Si supponga di voler ordinare 10^6 numeri da 32 bit
- Come scelgo la base b ?
- 10^6 è compreso fra 2^{19} e 2^{20}
- Scegliendo $b=2^{16}$ si ha:
 - sono sufficienti 2 passate di bucketSort
 - ogni passata richiede tempo lineare

Problema 4.10

Descrivere un algoritmo che, dati n interi in $[1, k]$, processa la sequenza in modo da poter poi rispondere a interrogazioni del tipo: “quanti interi cadono nell’intervallo $[a, b]$?”, per ogni a e b in tempo $O(1)$. L’algoritmo deve richiedere tempo di preprocessing $O(n + k)$.

Soluzione Problema 4.10

Idea: Costruire in tempo $O(n+k)$ un array Y di dimensione k dove $Y[i]$ è il numero di elementi di X che sono $\leq i$

CostruisciOracolo (X, k)

1. Sia Y un array di dimensione k
2. **for** $i=1$ **to** k **do** $Y[i]=0$
3. **for** $i=1$ **to** n **do** incrementa $Y[X[i]]$
4. **for** $i=2$ **to** k **do** $Y[i]=Y[i]+Y[i-1]$
5. **return** Y

InterrogaOracolo (Y, k, a, b)

1. **if** $b > k$ **then** $b=k$
2. **if** $a \leq 1$ **then return** $Y[b]$
else return $(Y[b]-Y[a-1])$