Ordinamenti per confronto: albero di decisione

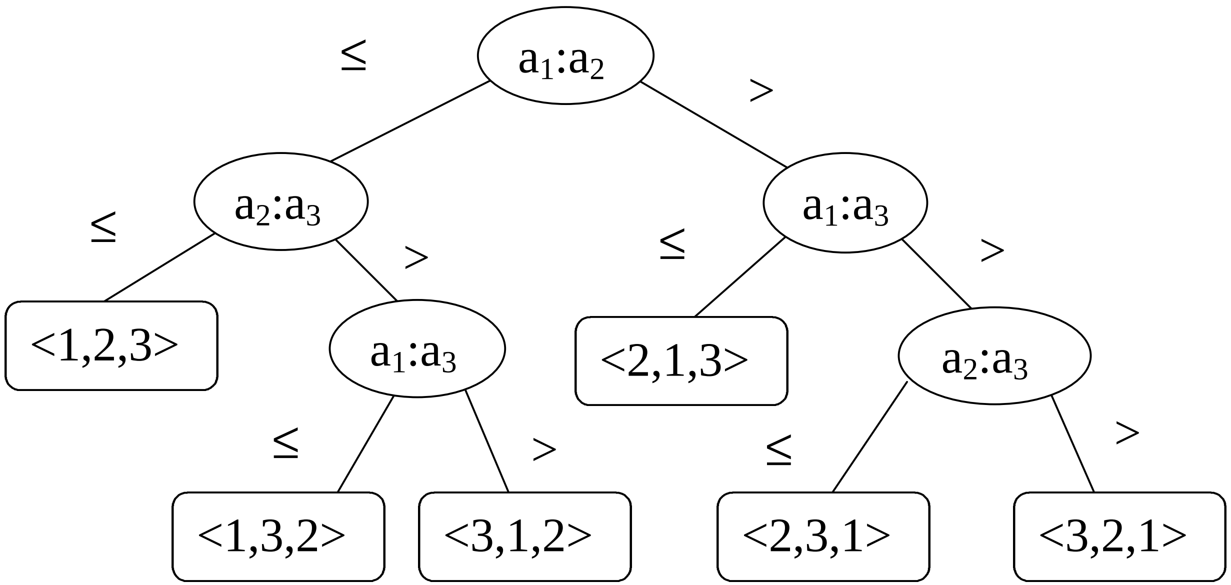
<u>Albero di decisione</u> = rappresentazione grafica di tutte le possibili sequenze di confronti eseguite da un algoritmo assegnato di ordinamento per confronto quando opera su un input di una data dimensione

Assunzioni semplificative:

- tutti gli elementi di input sono distinti ($\rightarrow$ i confronti $a_i = a_j$ sono inutili)
- tutti i confronti hanno la forma $a_i \leq a_j$

- Cammino dalla radice a una foglia: una esecuzione dell'algoritmo di ordinamento
- Nodo interno: esecuzione di un confronto $a_i \leq a_j$ (etichetta $a_i : a_j$) per qualche i,j tale che $1 \leq i,j \leq n$
- Foglia: permutazione degli indici dell'input ($\rightarrow$ n° foglie $\geq$ n° permutazioni dell'input, n!) che costituisce la soluzione per l'esecuzione dell'algoritmo che porta a tale foglia
- Altezza dell'albero di decisione = n° di confronti che l'algoritmo di ordinamento esegue nel caso pessimo

Albero di decisione di INSERTION-SORT



Ordinamenti per confronto: limite inferiore per il caso pessimo

Limite inferiore sull'altezza degli alberi di decisione = limite inferiore sul tempo di esecuzione di qualunque algoritmo di ordinamento per confronto

Teorema: qualunque albero di decisione che ordina n elementi ha altezza $\Omega(n\,lg\,n)$.

Dimostrazione

$n! \leq 2^h$ (cioè n° permutazioni ≤ n° foglie di un albero binario di altezza h) $\rightarrow$

$h \geq lg(n!) \rightarrow$

(per l'approssimazione di Stirling, $lg(n!) = \Theta(n\,lg\,n)$)

$h \geq \Theta(n\,lg\,n) \rightarrow h = \Omega(n\,lg\,n)$

Ordinamenti per confronto: limite inferiore per il caso pessimo (cont.)

Corollario: MERGE-SORT e HEAPSORT sono ordinamenti per confronto asintoticamente ottimi

Dimostrazione

Il limite superiore del tempo di esecuzione dei due algoritmi coincide con quello inferiore del caso pessimo descritto dal teorema

COUNTING-SORT

Ipotesi: ognuno degli n elementi di input è un intero che cade nell'intervallo da 1 a k, per qualche intero k

L'algoritmo determina, per ogni elemento x dell'array di input, il n° di elementi ≤ x e poi usa questa informazione per porre x nella posizione che gli compete nell'array di output B (con qualche aggiustamento se esistono più elementi uguali)

COUNTING-SORT (cont.)

COUNTING-SORT(A,B,k)

▷ l'array B ha le stesse dimensioni di A
e mantiene l'output ordinato
for i ← 1 **to** k .... O(k)
  do C[i] ← 0
▷ l'array C[1 .. k] fornisce la memoria di lavoro temporanea
for j ← 1 **to** length[A] .... O(n)
  do C[A[j]] ← C[A[j]]+1
▷ C[i] contiene ora il numero di elementi uguali a i
for i ← 2 **to** k .... O(k)
  do C[i] ← C[i] + C[i – 1]
▷ C[i] contiene ora il numero di elementi ≤ i
for j ← length[A] **downto** 1 .... O(n)
  do B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] – 1

Esempio: COUNTING-SORT

	1	2	3	4	5	6	7
A	3	6	3	6	4	1	2

	1	2	3	4	5	6
C	1	1	2	1	0	2

	1	2	3	4	5	6
C	1	2	4	5	5	7

	1	2	3	4	5	6	7
B		2					

	1	2	3	4	5	6
C	1	1	4	5	5	7

Esempio: COUNTING-SORT (cont.)

	1	2	3	4	5	6	7
B	1	2					

	1	2	3	4	5	6
C	0	1	4	5	5	7

	1	2	3	4	5	6	7
B	1	2			4		

	1	2	3	4	5	6
C	0	1	4	4	5	7

Esempio: COUNTING-SORT (cont.)

B

1	2	3	4	5	6	7
1	2			4		6

C

1	2	3	4	5	6
0	1	4	4	5	6

B

1	2	3	4	5	6	7
1	2		3	4		6

C

1	2	3	4	5	6
0	1	3	4	5	6

Esempio: COUNTING-SORT **(cont.)**

B

1	2	3	4	5	6	7
1	2		3	4	6	6

C

1	2	3	4	5	6
0	1	3	4	5	5

B

1	2	3	4	5	6	7
1	2	3	3	4	6	6

C

1	2	3	4	5	6
0	1	2	4	5	5

Analisi di COUNTING-SORT

T(n) = O(k+n)

Se k = O(n), come di solito accade, T(n) = O(n)

L’algoritmo è stabile, cioè elementi con lo stesso valore compaiono nell’array di output nello stesso ordine in cui compaiono in quello di input (proprietà molto importante quando l’ordinamento avviene in base a una chiave ma ciascun elemento è un intero record)

RADIX-SORT

Scheda perforata: 12 righe * 80 colonne, dove ogni elemento della matrice può essere perforato

Rappresentazione di un n° in base 10: sono usate solo 10 righe e 1 colonna per ogni cifra → per rappresentare un n° di d cifre si usano d colonne

La macchina ordinatrice di schede può esaminare una sola colonna per volta

Per ordinare dei numeri, si ordina prima secondo la cifra meno significativa, e poi via via <u>in modo stabile</u> fino ad arrivare alla più significativa

L'algoritmo può essere usato anche per ordinare record in info con chiavi multiple

RADIX-SORT (cont.)

```
RADIX-SORT(A,d)
for i ← 1 to d
    do usa un ordinamento stabile
       per ordinare l'array A sulla cifra i        O(k+n)
                                                   (se COUNTING-SORT)
```

<u>Esempio</u>

200	200	200	122
451	200	200	200
418	451	418	200
122 ⇒	122 ⇒	418 ⇒	222
222	222	122	418
200	418	222	418
418	418	451	451
A: situazione iniziale	↑ i = 1	↑ i = 2	↑ i = d

Analisi di RADIX-SORT

T(n) dipende da quale ordinamento stabile viene usato

Se ogni cifra è nell'intervallo da 1 a k, con k non troppo grande, viene scelto COUNTING-SORT (che però non ordina "in loco") →

$T(n) = O(dn+dk) = O(n+k) = O(n)$

se d è costante

se $k = O(n)$

BUCKET-SORT

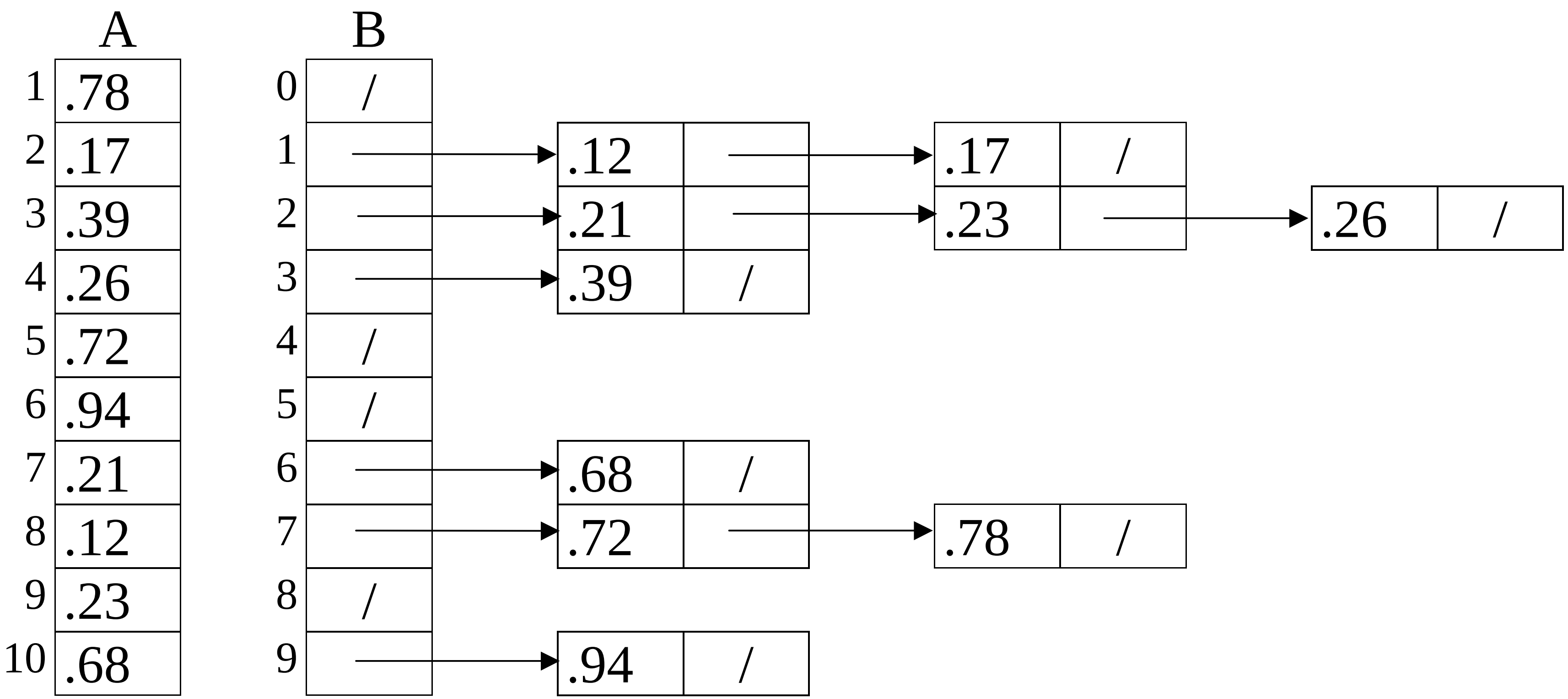
Ipotesi: l’input è generato da un processo casuale che distribuisce gli n elementi in modo uniforme nell’intervallo [0,1)

L’algoritmo divide l’intervallo [0,1) in n sottointervalli di uguale dimensione, detti bucket (secchi), in cui distribuisce gli elementi, poi ordina gli elementi di ogni bucket

BUCKET-SORT(A)
n $\leftarrow$ length[A]
$\triangleright$ richiede un array ausiliario B[0 .. n – 1] di liste concatenate
for i $\leftarrow$ 1 **to** n
    do inserisci A[i] nella lista B[$\lfloor$nA[i]$\rfloor$] O(n)
for i $\leftarrow$ 0 **to** n – 1
    do ordina la lista B[i] con INSERTION-SORT
concatena le liste B[0], B[1], .., B[n – 1] in quest’ordine O(n) nel caso peggiore

Esempio: BUCKET-SORT

Situazione al termine dell'esecuzione dell'algoritmo



Prova di correttezza di BUCKET-SORT

CASO 1

Ipotesi: A[i] e A[j], $i \neq j$, sono due elementi distinti che cadono nello stesso bucket B[k]

Tesi (da dimostrare): A[i] e A[j] compaiono nella sequenza di output in ordine appropriato

Dimostrazione: la tesi è dimostrata perché il contenuto di B[k] viene ordinato con INSERTION-SORT

Prova di correttezza di BUCKET-SORT (cont.)

CASO 2

Ipotesi: A[i] e A[j], $i \neq j$, sono due elementi distinti che cadono in due bucket distinti, rispettivamente B[i'] e B[j'], $i' < j'$ ($\rightarrow$ nella sequenza di output A[i] precede A[j])

Tesi: $A[i] \leq A[j]$

Dimostrazione: assumiamo, per assurdo, la negazione della tesi, cioè $A[i] > A[j]$; se ciò è vero, segue che

$$i' = \lfloor nA[i] \rfloor > \lfloor nA[j] \rfloor = j'$$

che contraddice l'ipotesi $\rightarrow$ la tesi è dimostrata

BUCKET-SORT: caso medio

n_i = variabile casuale che rappresenta il n° di elementi memorizzati nel bucket B[i]

Valor medio

Tempo di esecuzione della linea 6 : $E[O(n_i^2)] = O(E[n_i^2])$

Tempo di esecuzione delle linee 5 + 6: $\sum_{i=0}^{n-1} O(E[n_i^2]) = O\left(\sum_{i=0}^{n-1} E[n_i^2]\right) = O(n)$

$$E[n_i^2] = Var[n_i] + E^2[n_i] = 1 - \frac{1}{n} + 1^2 = 2 - \frac{1}{n}$$

La distribuzione di n_i è binomiale

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