

Algoritmi e Strutture Dati

Capitolo 4

Ordinamento: Quicksort (*) e metodi di ordinamento lineari

Punto della situazione

- Problema dell'ordinamento:
 - Lower bound – $\Omega(n \log n)$
 - Upper bound – $O(n \log n)$
 - Algoritmi **ottimi**:
 - Mergesort (*non in loco* e complessità $\Theta(n \log n)$)
 - Heapsort (*in loco* e complessità $\Theta(n \log n)$)
- Proviamo a costruire un nuovo algoritmo che ordini *in loco*, che costi **mediamente** $\Theta(n \log n)$, e che sia **molto efficiente nella pratica** (per ordinare sequenze fino a circa 10.000 elementi)

QuickSort

Usa la tecnica del **divide et impera**:

1. **Divide**: scegli a caso un elemento x della sequenza (**perno**) e **partiziona** la sequenza in 2 sottosequenze di elementi: A_1 che conterrà tutti gli elementi $\leq x$, ed A_2 che conterrà tutti gli elementi $> x$ (si noti che A_1 e A_2 in questo momento in generale sono disordinate)
2. Risolvi i due sottoproblemi ricorsivamente
3. **Impera**: restituisci la concatenazione delle due sottosequenze A_1 e A_2 (che a questo punto sono state ordinate ricorsivamente), separate dall'elemento x

Rispetto al MergeSort, divide complesso ed impera semplice

Partizione in loco

- Supponiamo di dover partizionare la sottosequenza $A[i;f]$, con $i < f$; scegliamo come perno il primo elemento della sequenza, cioè $A[i]$, e scorriamo $A[i;f]$ “in parallelo” da sinistra verso destra (indice inf) e da destra verso sinistra (indice sup)
 - da sinistra verso destra, ci si ferma su un elemento **maggiore** del perno
 - da destra verso sinistra, ci si ferma su un elemento **minore o uguale** al perno
 - quindi, si scambiano gli elementi e si riprende la scansione
- Quando gli indici si invertono (cioè, $inf = sup + 1$), ci si ferma; in questo momento inf punta ad un elemento **maggiore** del perno, mentre sup punta ad un elemento **minore-uguale** al perno (in particolare, potrebbe puntare al perno), e quindi si scambia il perno con l'elemento puntato da sup . In tal modo, il perno si troverà a dividere gli elementi più piccoli di lui da quelli più grandi.

Tempo di esecuzione di una scansione: $\Theta(n)$

QuickSort *in loco*

```

procedura partition(array A, indici i e f) → indice
1.   x ← A[i]
2.   inf ← i
3.   sup ← f + 1
4.   while ( true ) do
5.       do inf ← inf + 1 while ( inf ≤ f and A[inf] ≤ x )
6.       do sup ← sup - 1 while ( A[sup] > x )
7.       if ( inf < sup ) then scambia A[inf] ed A[sup]
8.       else break
9.   scambia A[i] ed A[sup]
10.  return sup

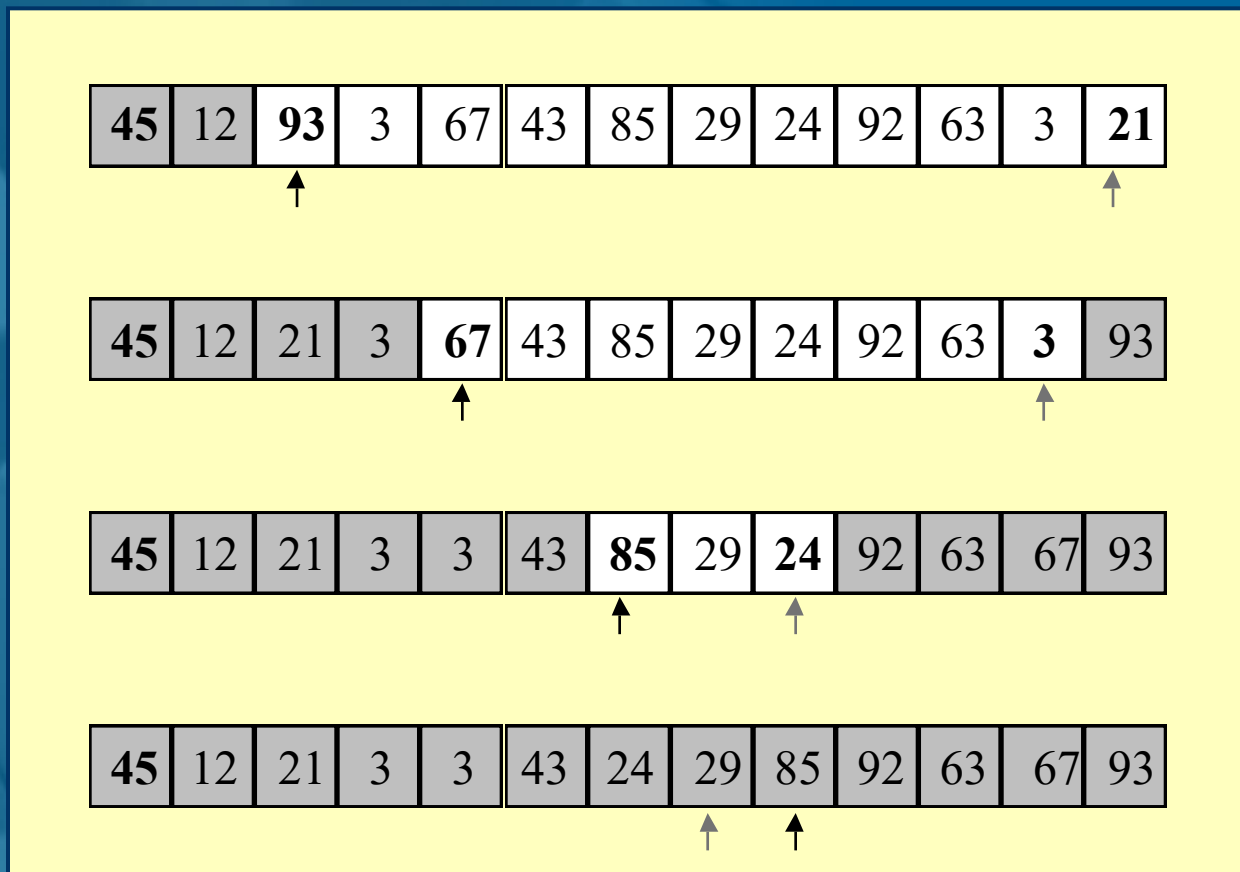
algoritmo quickSort(array A, indici i e f)
11.  if ( i ≥ f ) then return
12.  m ← partition(A, i, f)
13.  quickSort(A, i, m - 1)
14.  quickSort(A, m + 1, f)

```

Si noti che se inf supera il limite destro f , ciò significa che il perno $A[i]$ è \geq di tutti gli elementi della sequenza $A[i;f]$, incluso quindi $A[f]$, quindi il **while** di riga 6 decrementerà sup a f , mentre il **while** di riga 4 andrà in **break** a riga 8, con scambio finale a riga 9 tra $A[i]$ e $A[f]$

Nota: Viene lanciato chiamando `quickSort(A, 1, n)`

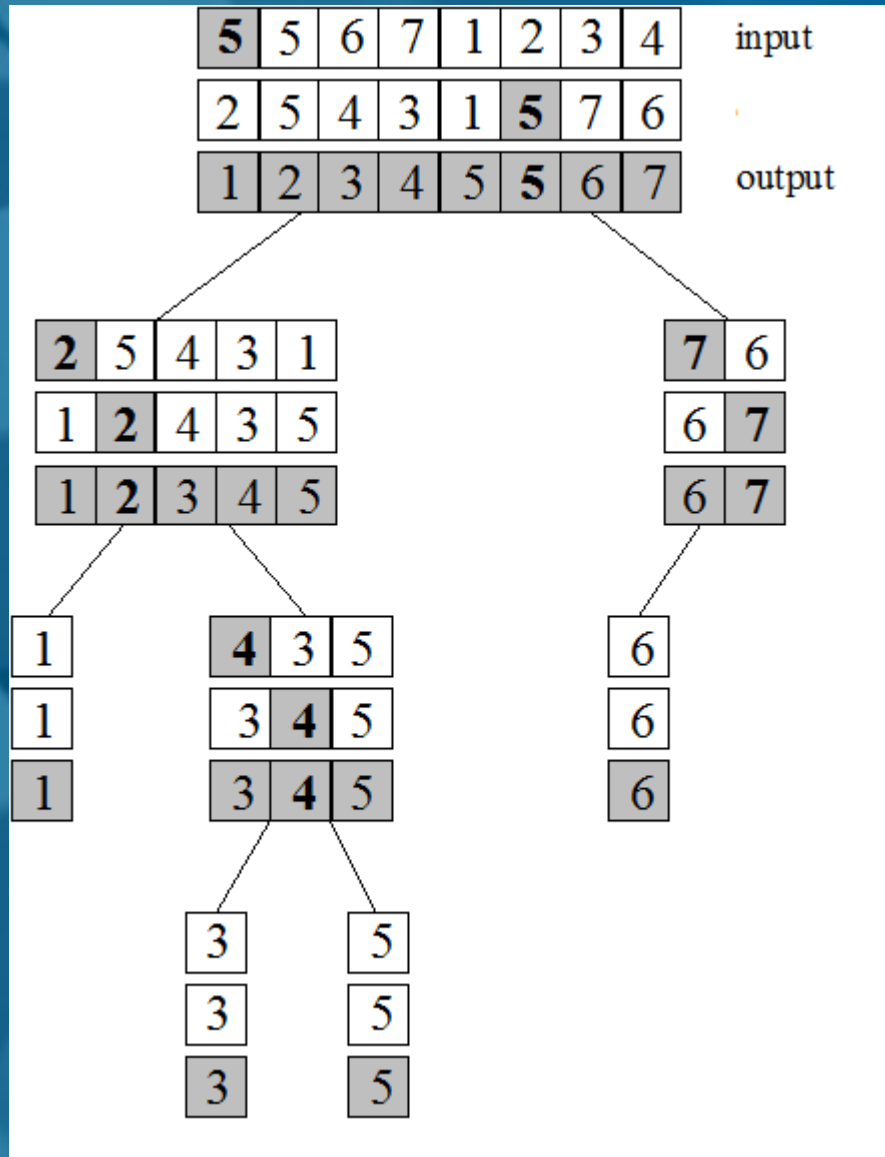
Partizione in loco: un esempio



Infine, si scambia 45 con 29, ottenendo $\langle 29, 12, 21, 3, 3, 43, 24, 45, 85, 92, 63, 67, 93 \rangle$



Esempio di esecuzione



L'albero delle chiamate ricorsive può essere sbilanciato... cosa succede nel caso peggiore?

Analisi di Quicksort

Siano a e b le dimensioni delle sottosequenze A_1 (elementi minori o uguali al perno) e A_2 (elementi maggiori del perno); allora, il numero di confronti $C(n)$ (operazione dominante) è pari a:

$$C(n) = C(a) + C(b) + (n-1) \quad \text{per } n > 1, \quad C(1) = 0$$

Confronti eseguiti ad ogni passata bidirezionale

Analisi del caso peggiore

- Nel caso peggiore, il perno scelto ad ogni passo è il minimo o il massimo degli elementi nell'array ($a=0$ e $b=n-1$, oppure $a=n-1$ e $b=0$)

- Il numero di confronti diventa pertanto:

$$C(n) = C(n-1) + (n-1) \quad \text{per } n > 1, \quad C(1) = 0$$

- Svolgendo per iterazione si ottiene

$$\begin{aligned} C(n) &= C(n-1) + (n-1) = [C(n-2) + (n-2)] + (n-1) = \dots \\ &= [C(1) + 1] + 2 + \dots + (n-2) + (n-1) = 1 + 2 + \dots + (n-2) + (n-1) \Rightarrow \end{aligned}$$

$$C(n) = \Theta(n^2) \Rightarrow T_{\text{WORST}}(n) = \Theta(n^2)$$

Analisi del caso migliore

- Nel caso migliore, il perno scelto ad ogni passo è il **mediano** nell'array ($a=b=n/2$)

- Il numero di confronti diventa pertanto:

$$C(n) = 2 \cdot C(n/2) + (n-1) = 2C(n/2) + \Theta(n)$$

e applicando il teorema master (caso 2)

$$C(n) = \Theta(n \log n) \Rightarrow T_{\text{BEST}}(n) = \Theta(n \log n)$$

Analisi del caso medio

- Possiamo affinare l'analisi del caso peggiore, dimostrando che $T_{AVG}(n) = \Theta(n \log n)$. Osserviamo infatti che il perno è un elemento che contiene un valore scelto uniformemente a caso nello spazio dei valori ammissibile, e quindi tale elemento ha la stessa **probabilità**, pari a $1/n$, di occupare una **qualsiasi** posizione dell'array dopo il partizionamento, il numero di confronti nel **caso atteso** è:

$$C(n) = \sum_{a=0}^{n-1} \frac{1}{n} \left[C(a) + C(n-a-1) + (n-1) \right]$$

dove a e $(n-a-1)$ sono le dimensioni dei sottoproblemi risolti ricorsivamente

Analisi del caso medio (2)

Osserviamo che $C(a)$ e $C(n-a-1)$ generano esattamente gli stessi termini nella sommatoria, e quindi:

$$C(n) = \sum_{a=0}^{n-1} \frac{1}{n} \left[C(a) + C(n-a-1) + (n-1) \right] = n-1 + \frac{2}{n} \sum_{a=0}^{n-1} C(a)$$

Moltiplicando entrambi i membri per n :

$$n C(n) = n(n-1) + 2 \sum_{a=0..n-1} C(a) \quad (1)$$

o anche, passando da n a $n-1$:

$$(n-1) C(n-1) = (n-1)(n-2) + 2 \sum_{a=0..n-2} C(a) \quad (2)$$

e sottraendo la (2) dalla (1):

$$n C(n) - (n-1) C(n-1) = 2 C(n-1) + n(n-1) - (n-1)(n-2)$$

e svolgendo e semplificando:

$$n C(n) = (n+1) C(n-1) + 2(n-1) \text{ cioè } n C(n) \leq (n+1) C(n-1) + 2n$$

Analisi del caso medio (3)

Dividiamo ora $n C(n) \leq (n+1) C(n-1) + 2n$ per $n(n+1)$:

$$\begin{aligned} C(n)/(n+1) &\leq C(n-1)/n + 2/(n+1) \leq (C(n-2)/(n-1) + 2/n) + 2/(n+1) \\ &\leq ((C(n-3)/(n-2) + 2/(n-1)) + 2/n) + 2/(n+1) \leq \dots \end{aligned}$$

e proseguendo fino a che l'argomento di C diventa 1, e quindi $C(1)=0$

$$C(n)/(n+1) \leq C(1)/2 + 2 \sum_{i=1..n} 1/(i+1) = 2 \sum_{i=1..n} 1/(i+1)$$

e per n che tende all'infinito la sommatoria (**serie armonica**) tende a $\ln n + 0.577\dots$, cioè:

$$C(n)/(n+1) \leq 2(\ln n + 0.577) = O(\log n), \text{ cioè } C(n) = O(n \log n).$$

Poiché chiaramente $C(n) = \Omega(n \log n)$ in quanto il caso medio deve costare almeno come il caso migliore, ne consegue che

$$T_{AVG}(n) = \Theta(n \log n).$$

Un confronto con l'Insertion Sort

Ricordiamo che nell'IS-2:

- $T_{\text{BEST}}(n) = \Theta(n)$
- $T_{\text{AVG}}(n) = \Theta(n^2)$
- $T_{\text{WORST}}(n) = \Theta(n^2)$

mentre nel QS:

- $T_{\text{BEST}}(n) = \Theta(n \log n)$
- $T_{\text{AVG}}(n) = \Theta(n \log n)$
- $T_{\text{WORST}}(n) = \Theta(n^2)$

⇒ È interessante notare che il caso medio dell'IS-2 costa come il suo caso peggiore, mentre il caso medio del QS costa come il suo caso migliore! Perché secondo voi?

Ordinamenti lineari

(per dati di input di tipo **numero intero** e con ulteriori proprietà particolari)

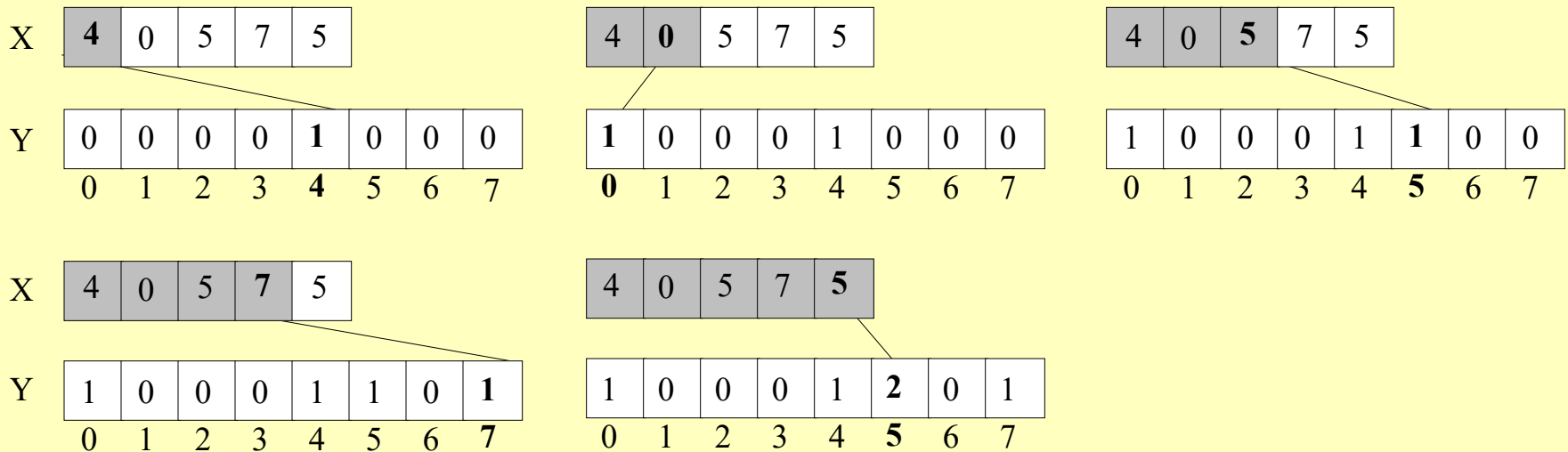
Un semplice esempio

- Supponiamo che gli n elementi da ordinare siano tutti distinti e appartenenti all'intervallo $[0, n-1]$
- In quanto tempo possiamo ordinarli?
- ☺ $\Theta(n)$: utilizzo un array di appoggio di dimensione n nel quale vado a scrivere in $\Theta(n)$ i valori $0, 1, 2, \dots, n-1$. Si noti che in questo modo sono anche in grado di verificare se l'input soddisfa realmente le ipotesi.
- Contraddice il lower bound? No, perché non è un **algoritmo basato su confronti** tra elementi: sfrutta una particolare proprietà dei dati in input che deve essere nota a priori!

IntegerSort: fase 1

Ordina n interi con valori in $[0, k=O(n)]$ in tempo $\Theta(n)$

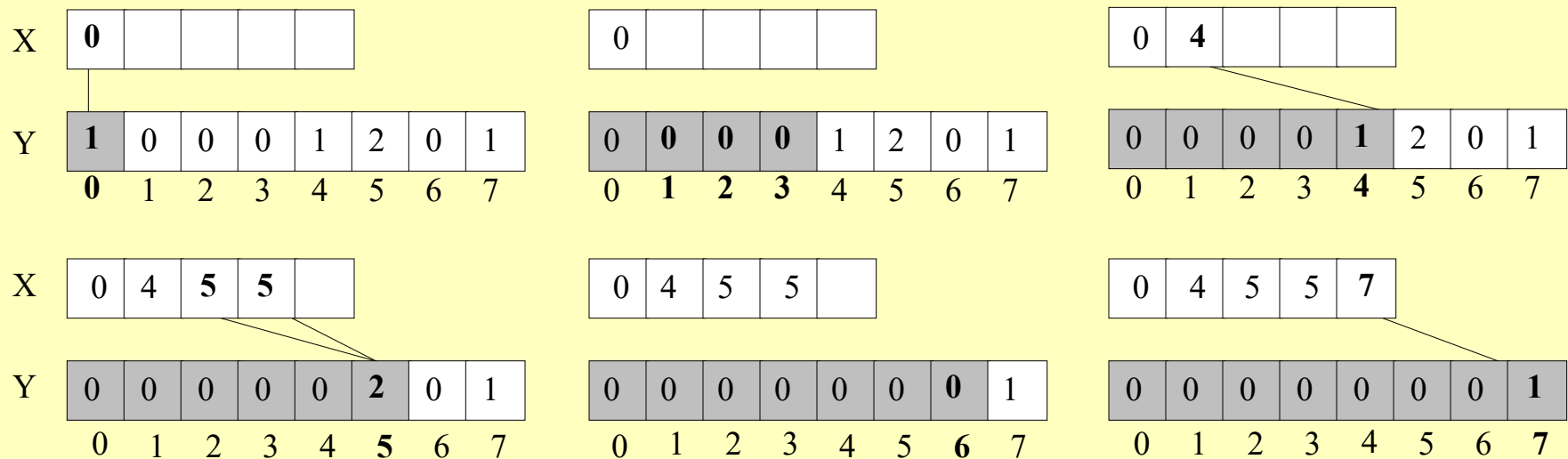
Inizialmente, scorre l'array di input X da sinistra verso destra, e mantiene un array Y di appoggio di $k+1$ contatori tale che $Y[i] = \text{numero di volte che il valore } i \text{ compare in } X$ (suppongo che il primo indice dell'array sia pari a 0)



(a) Calcolo di Y

IntegerSort: fase 2

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



(b) Ricostruzione di X

IntegerSort: analisi

- Tempo $\Theta(k)$ per inizializzare Y a 0 (si noti che devo conoscere **a priori** k)
- Tempo $\Theta(n)$ per calcolare i valori dei contatori
- Tempo $\Theta(n+k)$ per ricostruire X



$\Theta(n+k)$ \longrightarrow Tempo lineare $\Theta(n)$ se $k=O(n)$

Spazio utilizzato: $\Theta(n+k)$, cioè $\Theta(n)$ se $k=O(n)$

BucketSort

Ordina n record con “chiavi” intere in $[0, k=O(n)]$
in tempo $\Theta(n)$

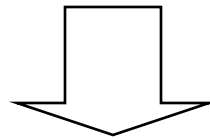
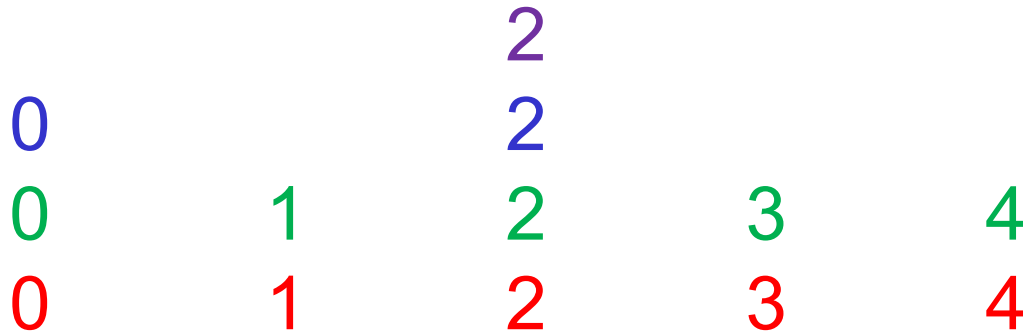
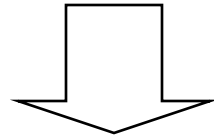
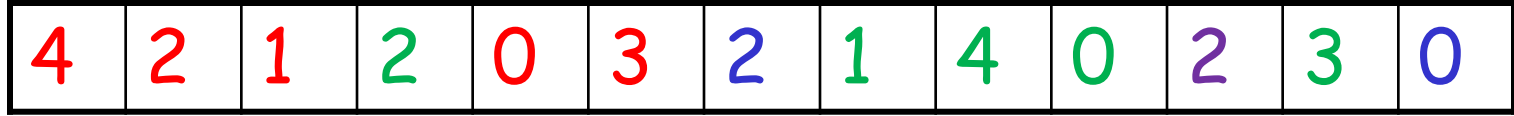
- Basta mantenere un array Y di $k+1$ liste (i bucket appunto), anziché di contatori, ed operare come per IntegerSort
- La lista $Y[i]$ conterrà gli elementi con chiave uguale a i
- Concatena infine le liste in ordine per $i=0, 1, \dots, k$

Tempo e spazio $\Theta(n+k)=\Theta(n)$ se $k=O(n)$,
come per IntegerSort

Stabilità

- Un algoritmo di ordinamento viene detto **stabile** se preserva l'ordine iniziale (ovvero nella sequenza di input) tra elementi aventi la stessa chiave
- Il BucketSort può essere reso stabile appendendo gli elementi di input **in coda** alla opportuna lista in **Y**, man mano che essi si presentano

Esempio



RadixSort

Ordina n interi con valori in $[0, k=O(n^c)]$, $c > 1$, in tempo $\Theta(n)$

- Rappresentiamo gli elementi in base b (che come vedremo deve essere scelta **opportunamente**), ed eseguiamo una serie di BucketSort con chiavi in $[0, b-1]$
- Partiamo dalla cifra meno significativa verso quella più significativa

Si noti la stabilità del bucket sort: 368 continua a precedere 387



Correttezza (per induzione)

- Dimostriamo per induzione che dopo la t -esima passata di BucketSort, i numeri sono correttamente ordinati rispetto alle t cifre meno significative
- Passo base: per $t=1$, è banalmente vero;
- Alla t -esima passata:
 - se x e y hanno una diversa t -esima cifra meno significativa, la t -esima passata di BucketSort li ordina rispetto a tale cifra, e quindi l'enunciato è vero;
 - altrimenti, se x e y hanno la stessa t -esima cifra meno significativa, la proprietà di stabilità del BucketSort li mantiene nell'ordine stabilito durante la passata **precedente**, e quindi l'enunciato è vero perché per ipotesi induttiva alla fine della $(t-1)$ -esima passata x e y sono ordinati correttamente rispetto alle $t-1$ cifre meno significative.

Tempo di esecuzione (e spazio utilizzato)

- $\Theta(\log_b k)$ passate di BucketSort
- Ciascuna passata richiede tempo (e spazio) $\Theta(n+b)$, e quindi $T(n) = \Theta((n+b) \log_b k)$

Scegliendo $b = \Theta(n)$, si ha $\log_b k = \frac{\log k}{\log b} = \Theta\left(\frac{\log k}{\log n}\right)$

e quindi $T(n) = \Theta\left[n \frac{\log k}{\log n}\right] \Rightarrow$ se $k = O(n^c)$, c costante

$$\Rightarrow T(n) = \Theta\left[n \frac{\log n^c}{\log n}\right] = \Theta(nc) = \Theta(n)$$

- E il cambiamento di base? E' facile vedere che costa $\Theta(n \log_b n^c) = \Theta(n \log_n n^c) = \Theta(nc) = \Theta(n)$

Riepilogo Ordinamento

- Nuove tecniche:
 - Incrementale (SelectionSort, InsertionSort)
 - Divide et impera (MergeSort, QuickSort)
 - Strutture dati efficienti (HeapSort)
- Alberi di decisione per la dimostrazione di delimitazioni inferiori
- Proprietà particolari dei dati in ingresso possono aiutare a progettare algoritmi più efficienti: algoritmi lineari