

# 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 un elemento **x** della sequenza (**perno**) e partiziona la sequenza in elementi  $\leq x$  ed elementi  $>x$
  2. Risolvi i due sottoproblemi ricorsivamente
  3. **Impera**: restituisci la concatenazione delle due sottosequenze ordinate

Rispetto al MergeSort, divide complesso ed impera semplice

# QuickSort *non in loco*

## Quicksort(array A)

1. Scegli un elemento  $x$  in  $A$
2. Partiziona  $A$  rispetto ad  $x$  calcolando
3.  $A_1 = \{y \in A : y \leq x\}$
4.  $A_2 = \{y \in A : y > x\}$
5. **if** ( $|A_1| > 1$ ) **then** Quicksort( $A_1$ )
6. **if** ( $|A_2| > 1$ ) **then** Quicksort( $A_2$ )
7. Copia la concatenazione di  $A_1$  e  $A_2$  in  $A$

**Nota:** Si usano 2 array ausiliari, cioè l'ordinamento non è *in loco*

# Partizione in loco

- Scegli un perno a caso, e scorri l'array “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
- Scambia gli elementi e riprendi la scansione
- Quando gli indici si invertono (cioè,  $\text{inf} = \text{sup} + 1$ ), fermati; in questo momento **inf** punta ad un elemento **maggiore** del perno, mentre **sup** punta ad un elemento **minore-uguale** al perno; allora, se il perno non è puntato da **inf** o **sup**, scambia il perno con:
  - Caso 1:** l'elemento puntato da **sup**, se il perno si trova alla sua sinistra;
  - Caso 2:** l'elemento puntato da **inf**, se il perno si trova alla sua destra.

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

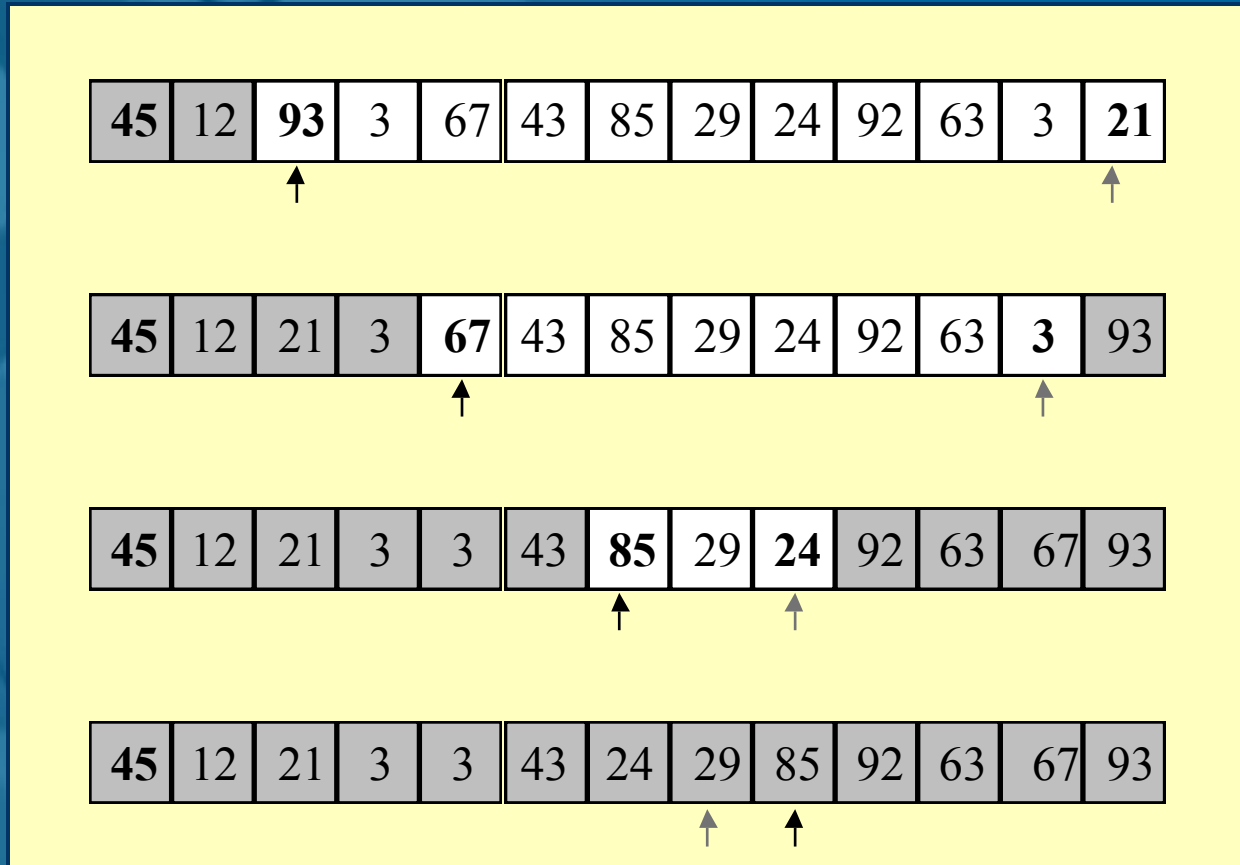
# QuickSort *in loco*

```
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)
```

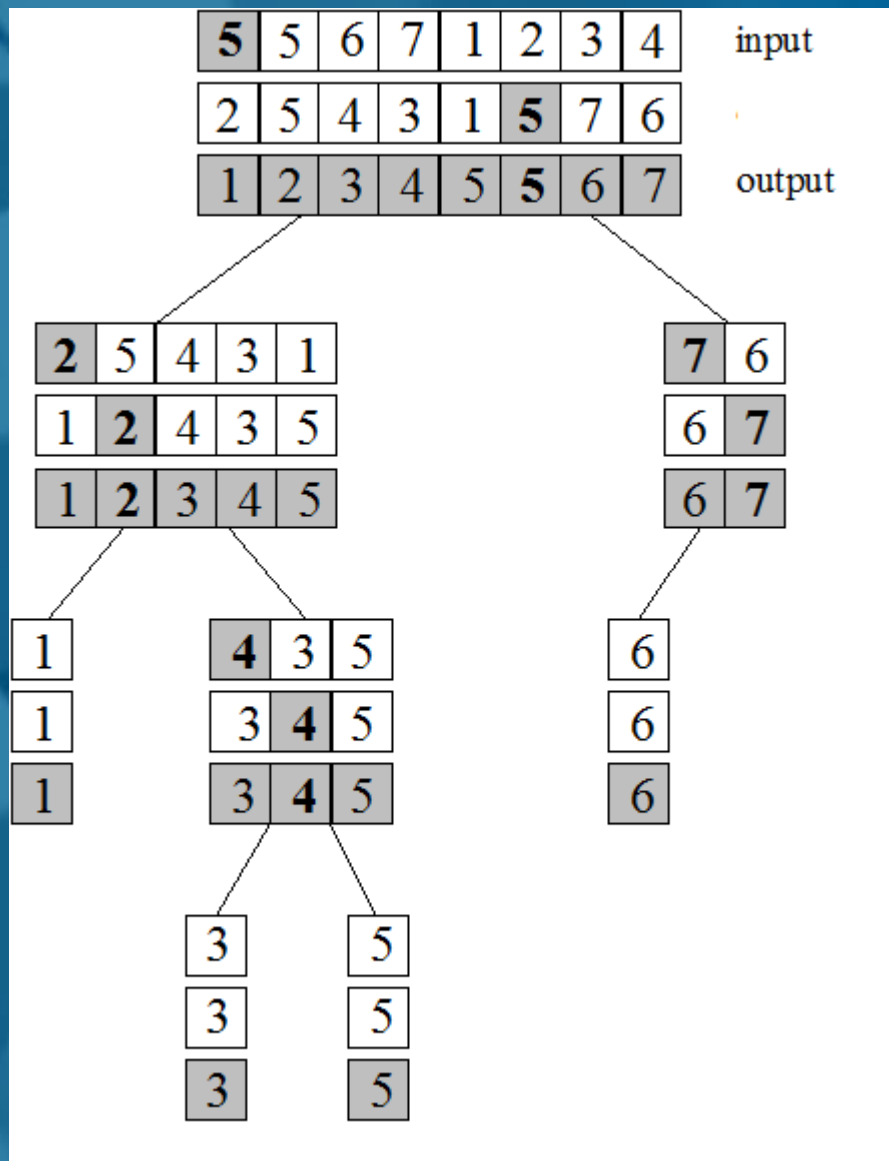
**Nota:** Viene lanciato chiamando `quickSort(A, 1, n)`; si noti che viene scelto come perno sempre l'elemento in **prima posizione** nella sequenza, e quindi devo sempre scambiare il perno con `A[sup]` (**Caso 1**)

# 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  $A_1$  e  $A_2$ ; allora, il numero di confronti  $C(n)$  (operazione dominante) è pari a:

$$C(n) = C(a) + C(b) + (n-1)$$


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:

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

- Svolgendo per iterazione si ottiene

$$C(n) = (n-1) + (n-2) + \dots + 1 \Rightarrow$$

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

# 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  $T_{AVG}(n) = \Theta(n \log n)$ . Osservo che il perno è un elemento scelto a caso...
- ...e poiché 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:

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

Dimostriamo che  $C(n) \leq 2n \log n$  per induzione su  $n$

1. Passo base,  $n=1$ :  $C(1)=0 \leq 2 \cdot 1 \cdot \log 1=0$ ;
2. Assumiamo che  $C(a) \leq 2a \log a$  per  $a \leq n-1$ ; allora:

$$C(n) \leq n-1 + \frac{4}{n} \sum_{a=1}^{n-1} a \log a \leq n-1 + \frac{4}{n} \int_2^n a \log a \, da$$

e integrando per parti e manipolando si ottiene

$$C(n) \leq 2n \log n, \text{ ovvero } T_{\text{AVG}}(n) = O(n \log n),$$

e poiché analogamente si può dimostrare che  $C(n) \geq n \log n$ , si ottiene

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

# Un confronto con l'Insertion Sort

Ricordiamo che nell'IS:

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

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 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 con proprietà particolari)

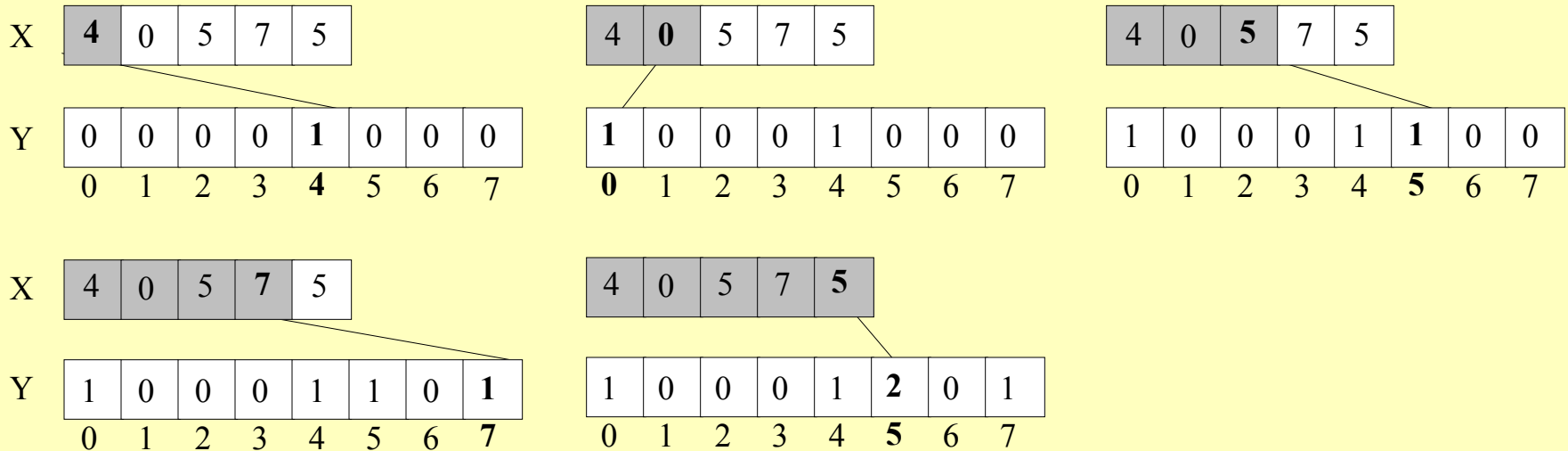
# Un semplice esempio

- 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 che l'input soddisfi realmente le ipotesi.
- Contraddice il lower bound? No, perché non è un **algoritmo basato su confronti** tra elementi!



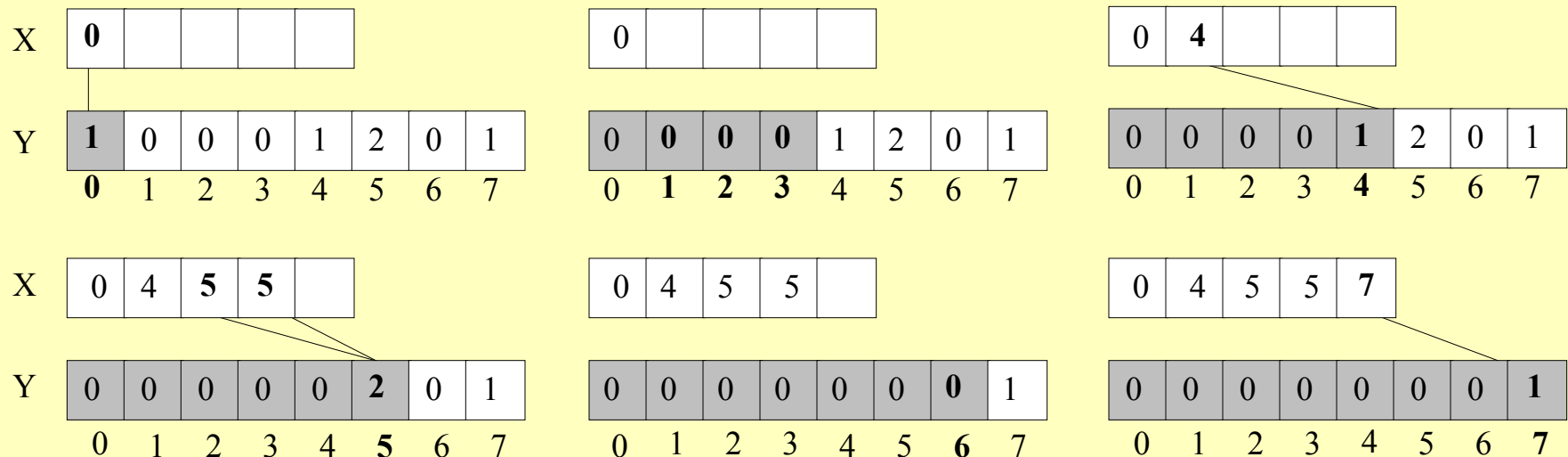
# IntegerSort: fase 1

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 se  $k=O(n)$

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

# BucketSort

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

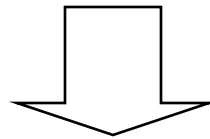
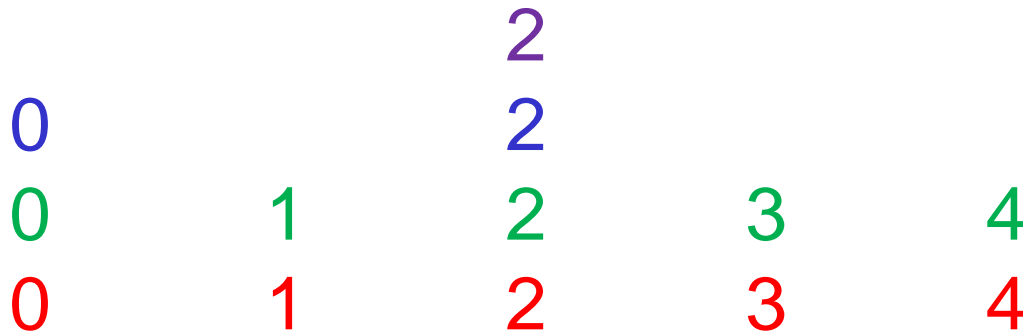
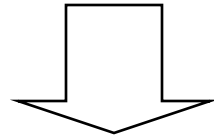
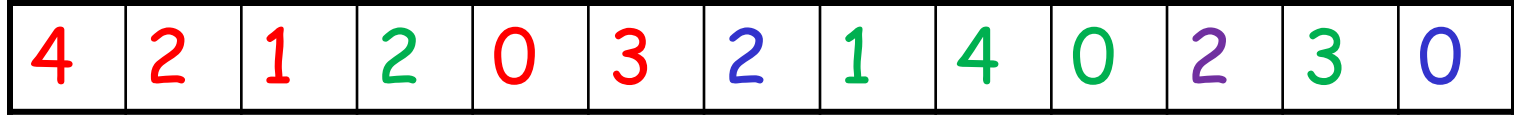
- Basta mantenere un array  $Y$  di  $k$  liste (i bucket appunto), anziché di contatori, ed operare come per IntegerSort
- La lista  $Y[i]$  conterrà gli elementi con chiave uguale a  $i$
- Concatena infine le liste in ordine per  $i=1, \dots, k$

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

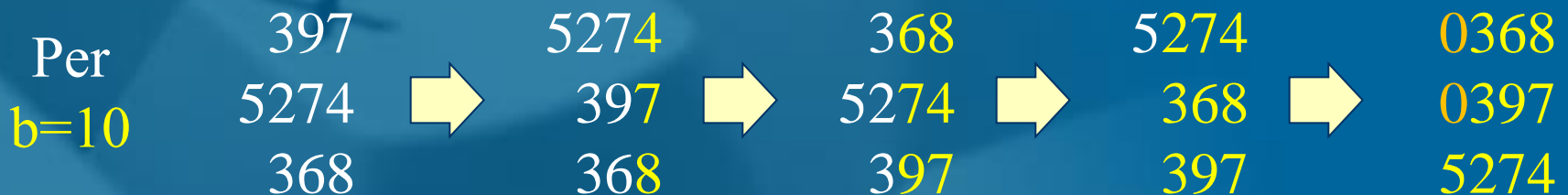
# Esempio



# RadixSort

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

- Cosa fare se il massimo valore  $k = \omega(n)$ , ad esempio se  $k = \Theta(n^c)$ ?
- 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



# 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  $\Theta(nc) = \Theta(n)$

# 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