

CMP461: Algorithms



Lecture 04: Linear Time Sorting and Order Statistics

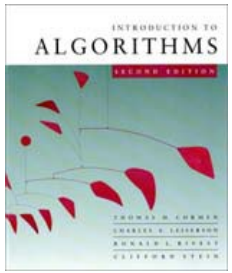
Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Fall 2013

Agenda

- Sorting Lower Bound
- Linear Time Sorting
 - Counting Sort
 - Radix Sort
- Order Statistics

Acknowledgment

A lot of slides adapted from the slides of Erik Demaine and Charles Leiserson



How fast can we sort?

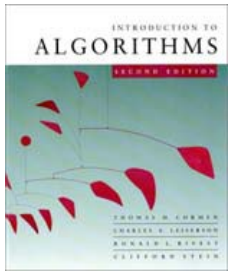
All the sorting algorithms we have seen so far are **comparison sorts**: only use comparisons to determine the relative order of elements.

- *E.g.*, insertion sort, merge sort, quicksort, heapsort.

The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

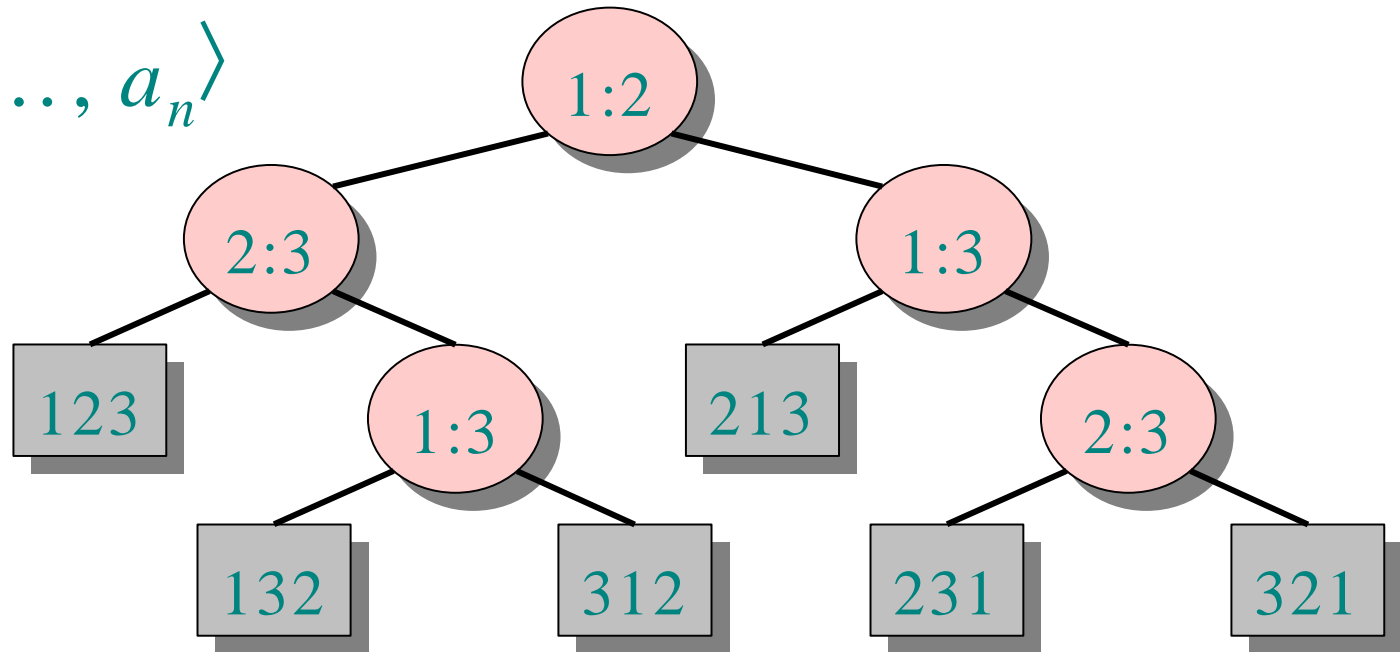
Is $O(n \lg n)$ the best we can do?

Decision trees can help us answer this question.



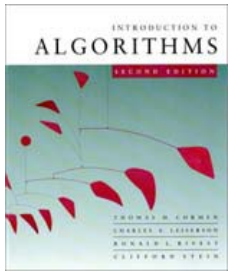
Decision-tree example

Sort $\langle a_1, a_2, \dots, a_n \rangle$



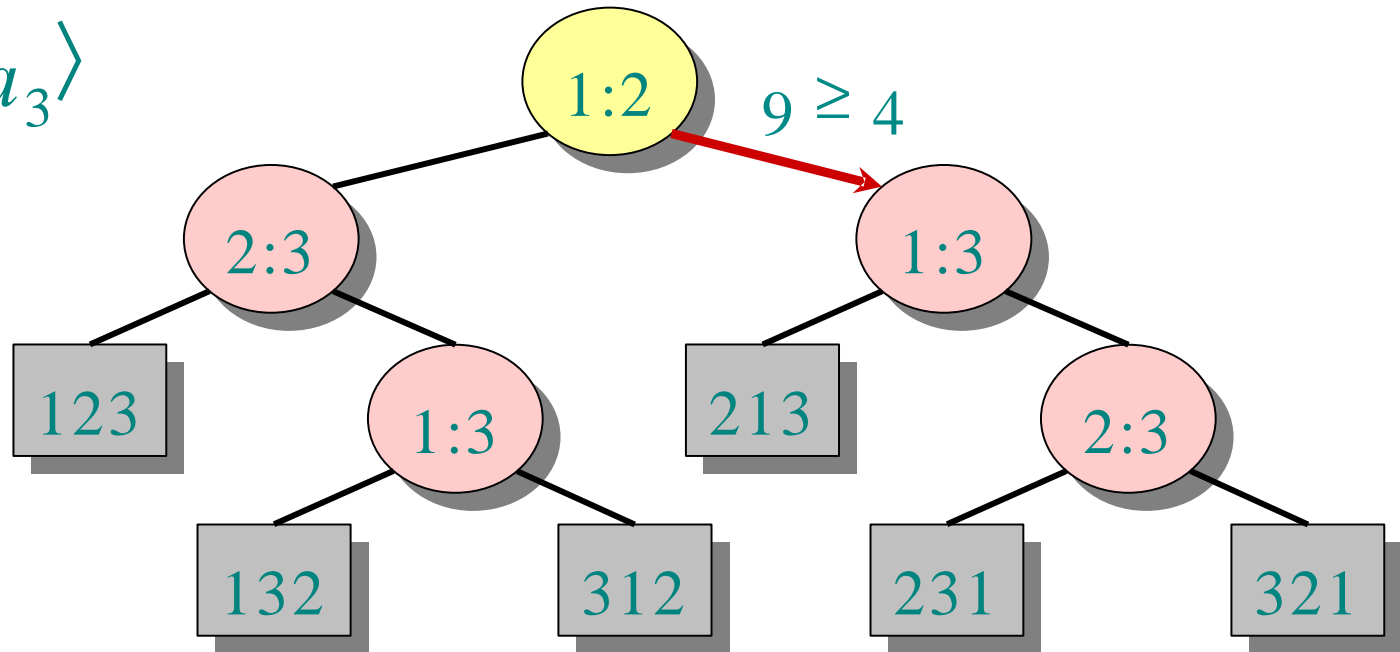
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



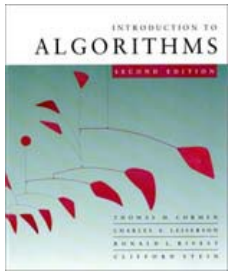
Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 9, 4, 6 \rangle$:



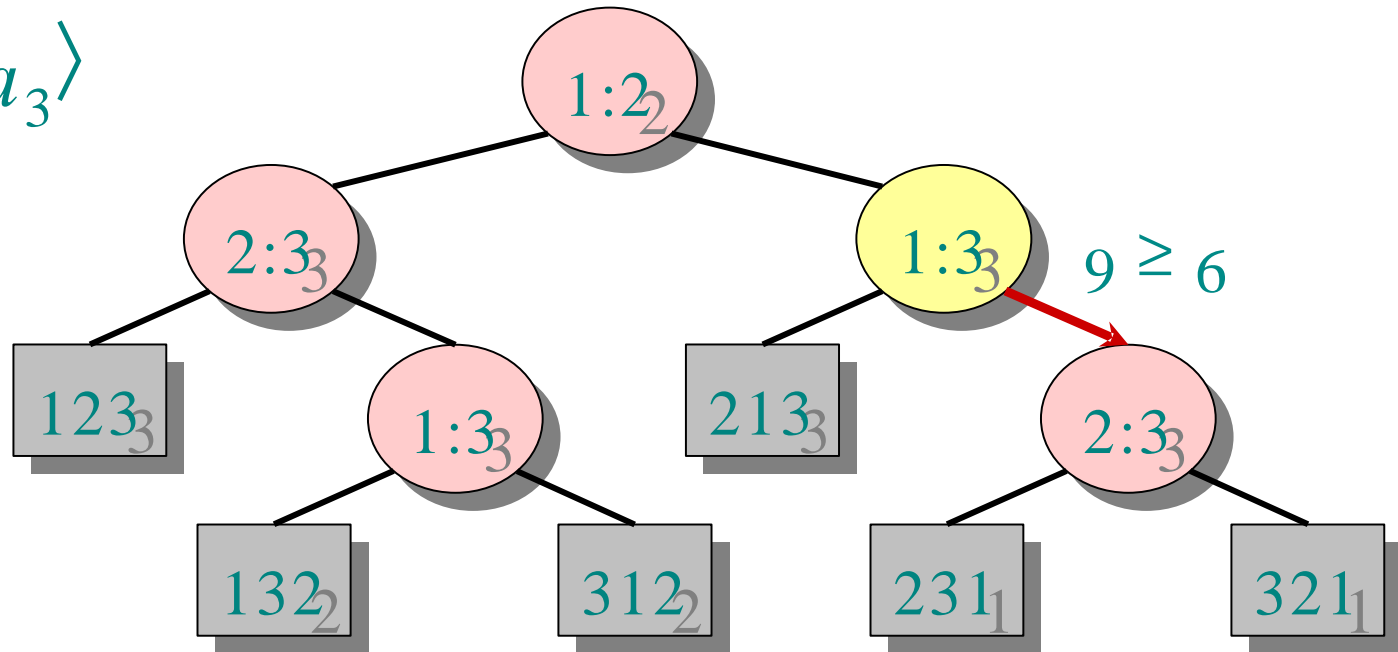
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



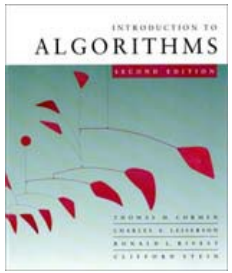
Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 9, 4, 6 \rangle$:



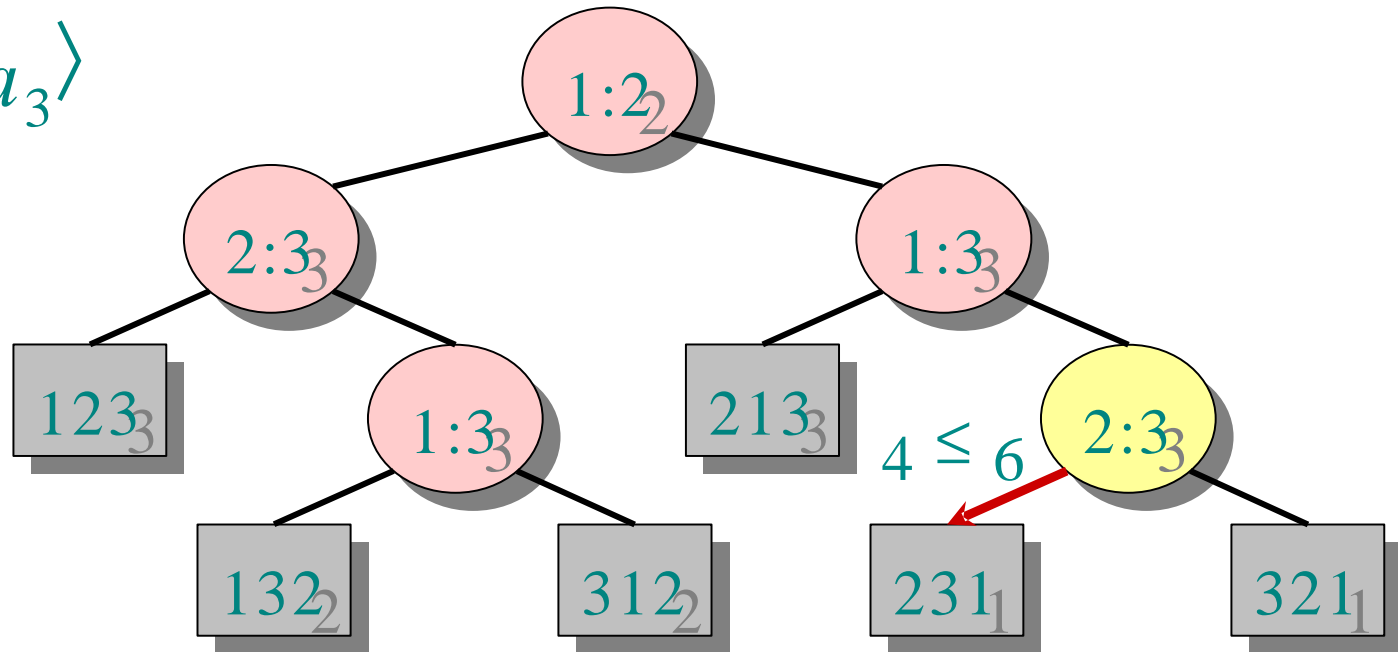
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



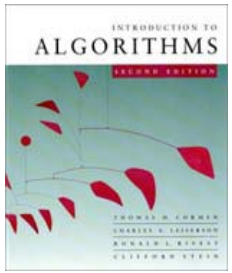
Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 9, 4, 6 \rangle$:



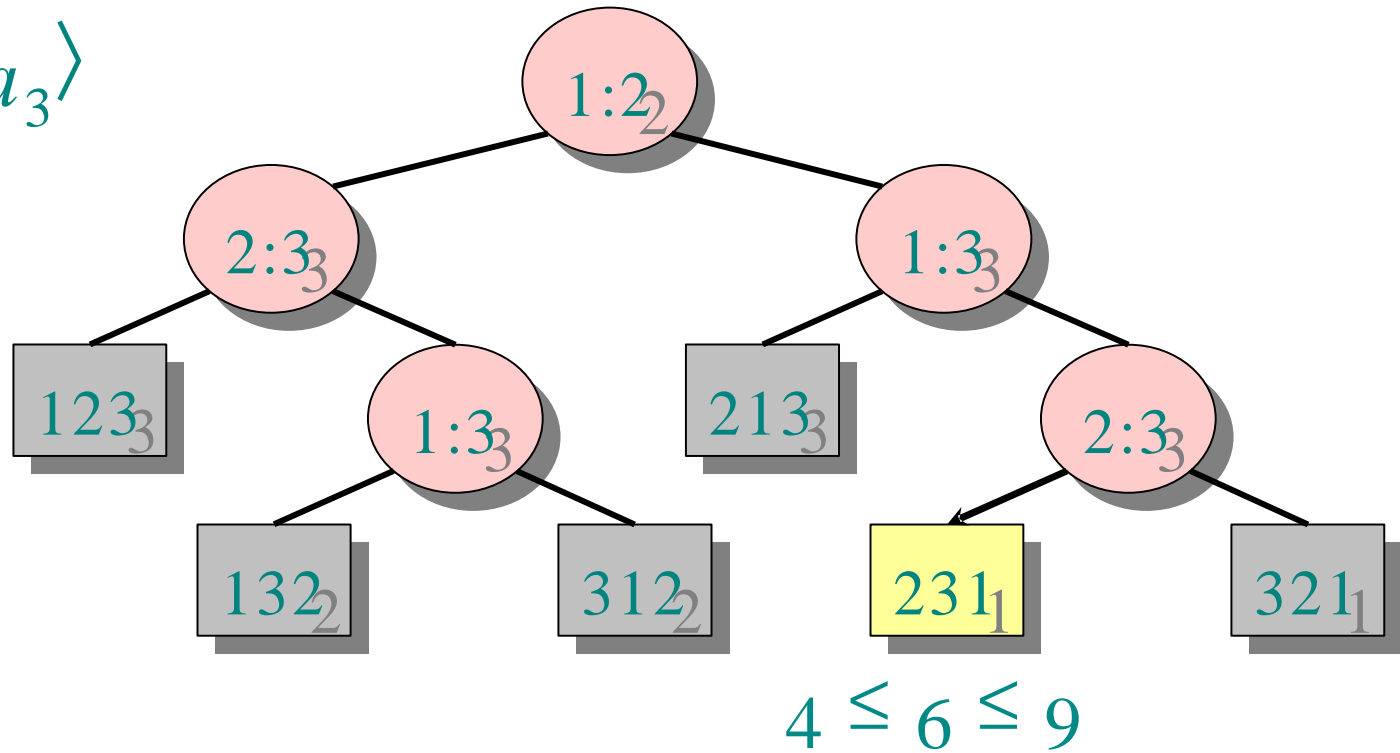
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

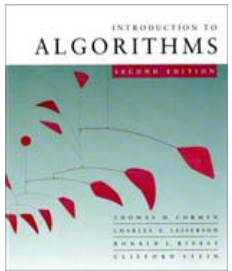


Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 9, 4, 6 \rangle$:



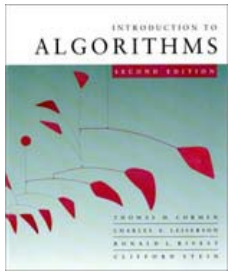
Each leaf contains a permutation $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ has been established.



Decision-tree model

A decision tree can model the execution of any comparison sort:

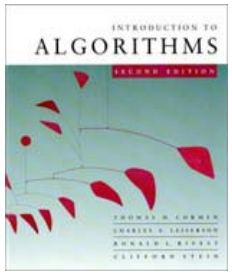
- One tree for each input size n .
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.



Lower bound for decision-tree sorting

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height- h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

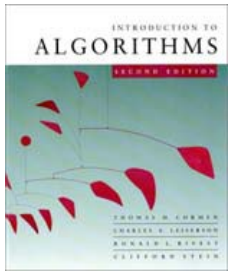


Lower bound for decision-tree sorting

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height- h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$\therefore h \geq \lg(n!)$ (\lg is mono. increasing)

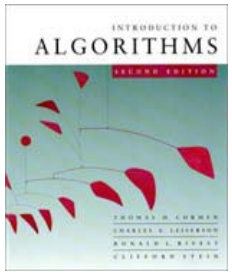


Lower bound for decision-tree sorting


Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height- h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$$\begin{aligned} \therefore h &\geq \lg(n!) && (\lg \text{ is mono. increasing}) \\ &\geq \lg((n/e)^n) && (\text{Stirling's formula}) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n). \quad \square \end{aligned}$$

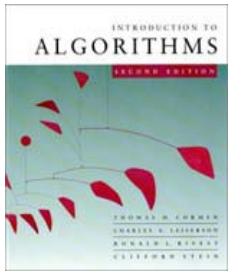


Lower bound for comparison sorting

Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms. 

Linear Time Sorting

- So how is linear time sorting possible?
- No comparisons!



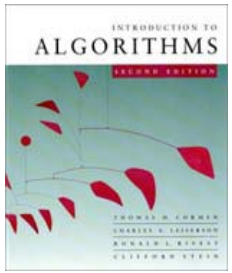
Sorting in linear time

Counting sort: No comparisons between elements.

- **Input:** $A[1 \dots n]$, where $A[j] \in \{1, 2, \dots, k\}$.
- **Output:** $B[1 \dots n]$, sorted.
- **Auxiliary storage:** $C[1 \dots k]$.

Idea

For every number, find how many number precede it in the sorted array



Counting sort

for $i \leftarrow 1$ **to** k

do $C[i] \leftarrow 0$

for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

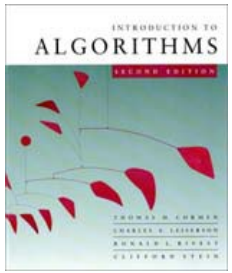
for $i \leftarrow 2$ **to** k

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

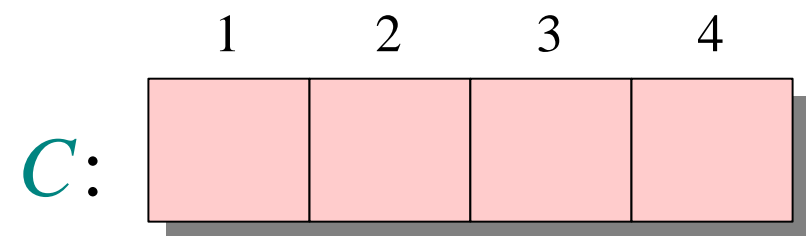
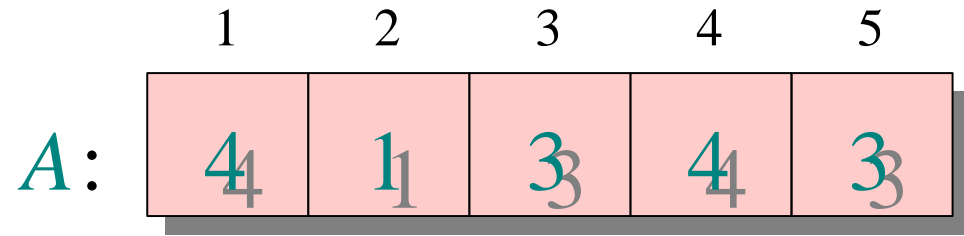
for $j \leftarrow n$ **downto** 1

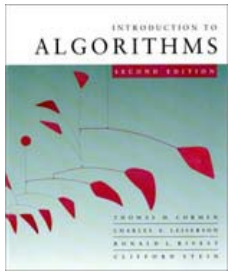
do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

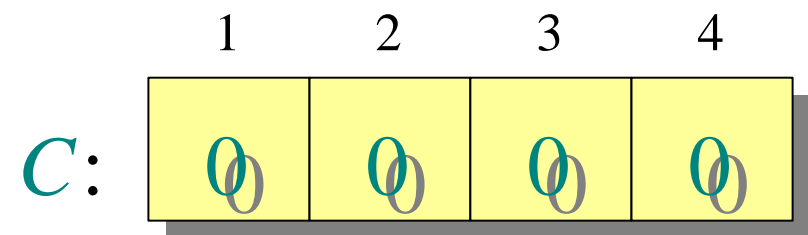
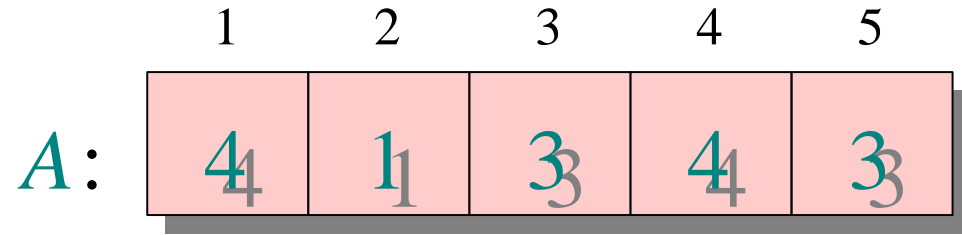


Counting-sort example

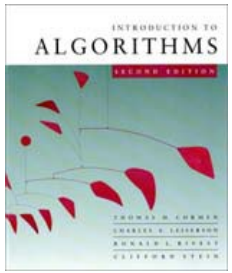




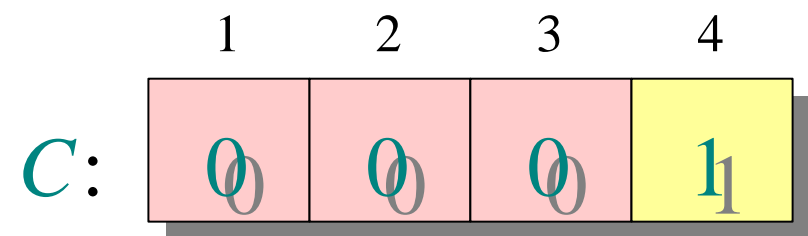
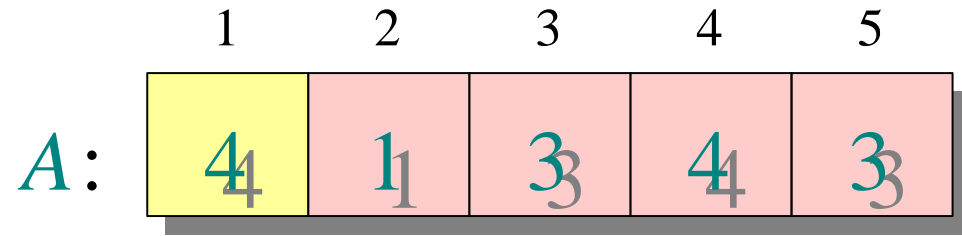
Loop 1



```
for  $i \leftarrow 1$  to  $k$   
  do  $C[i] \leftarrow 0$ 
```

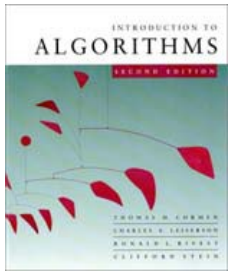


Loop 2

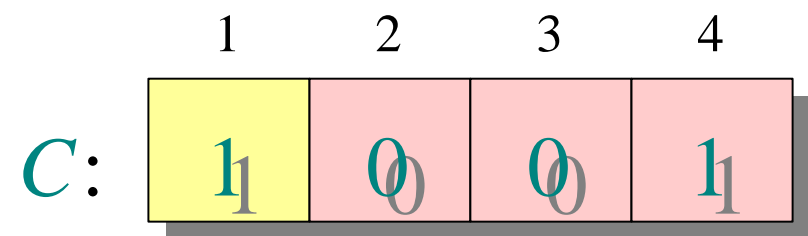
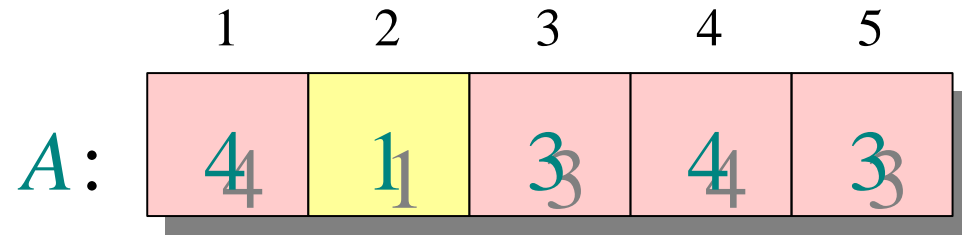


for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i] = |\{\text{key} = i\}|$

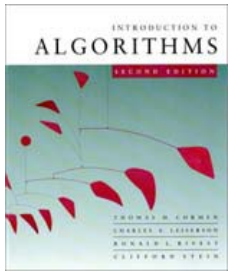


Loop 2

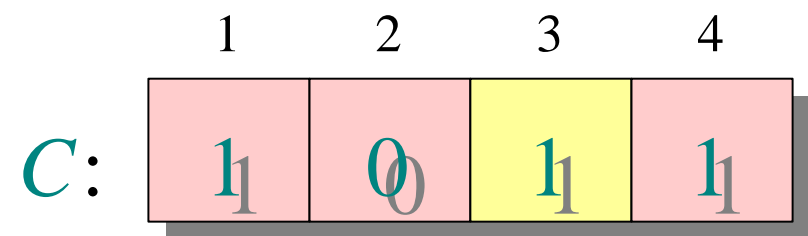
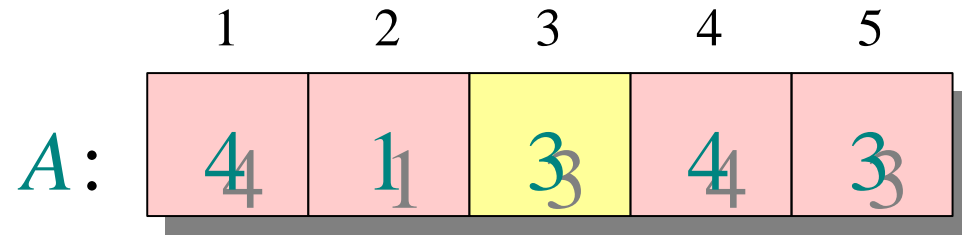


for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i] = |\{\text{key} = i\}|$

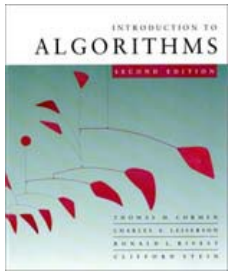


Loop 2

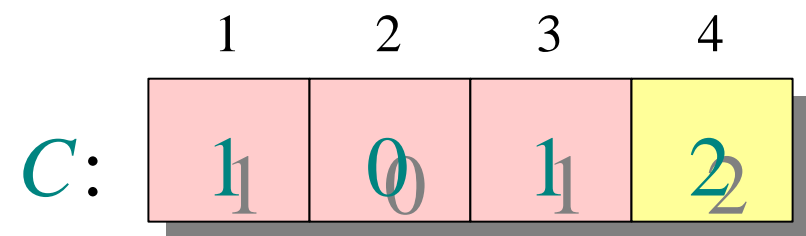
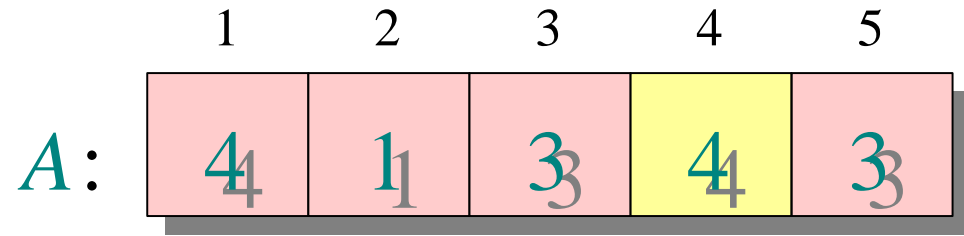


for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i] = |\{\text{key} = i\}|$

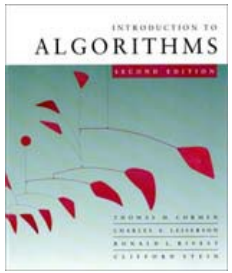


Loop 2

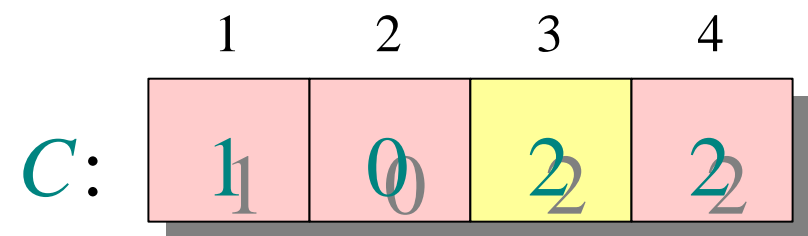
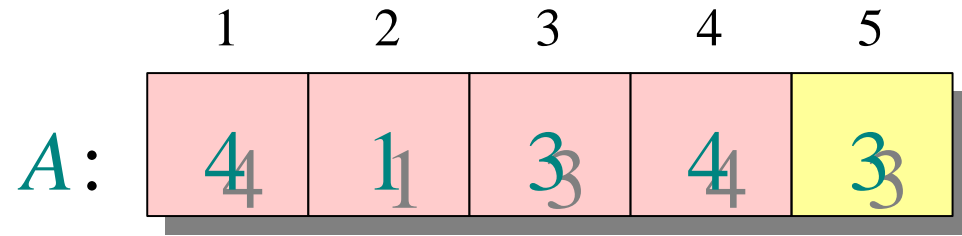


for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i] = |\{\text{key} = i\}|$

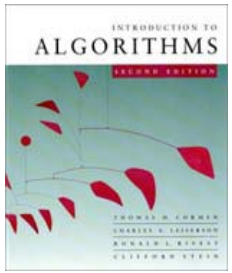


Loop 2

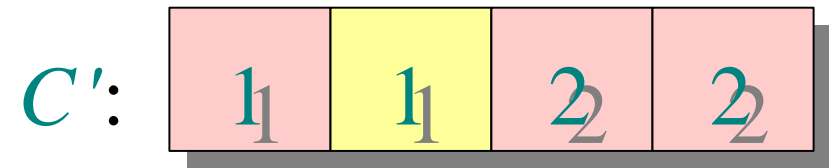
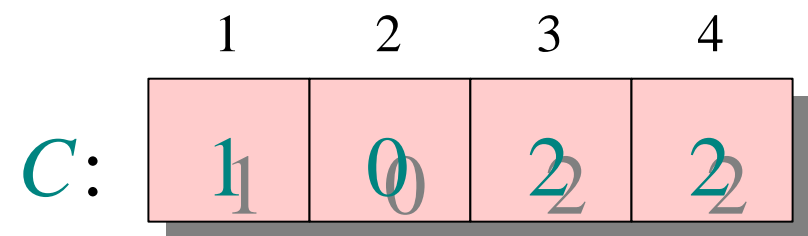
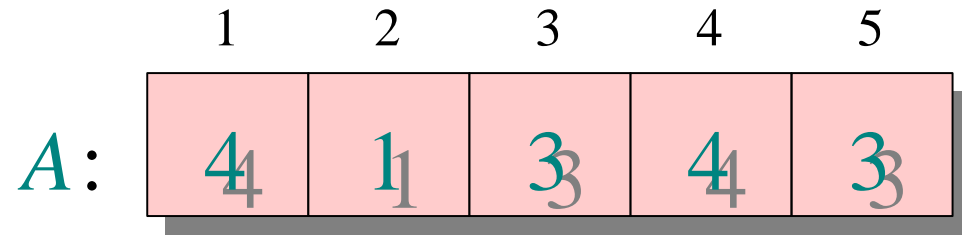


for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key} = i\}|$



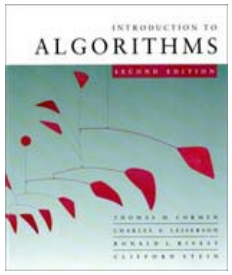
Loop 3



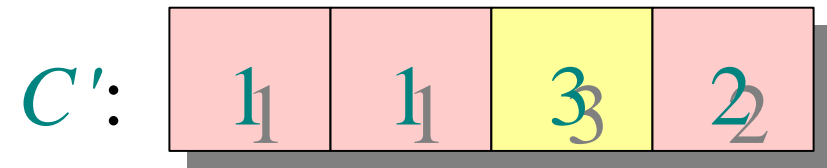
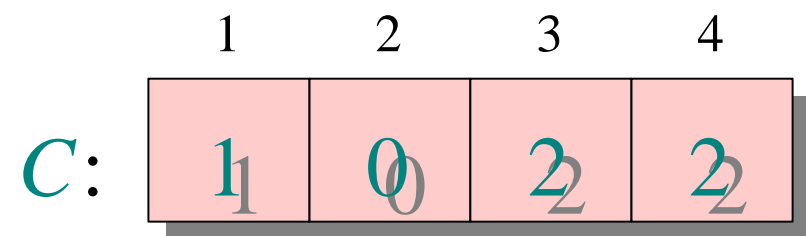
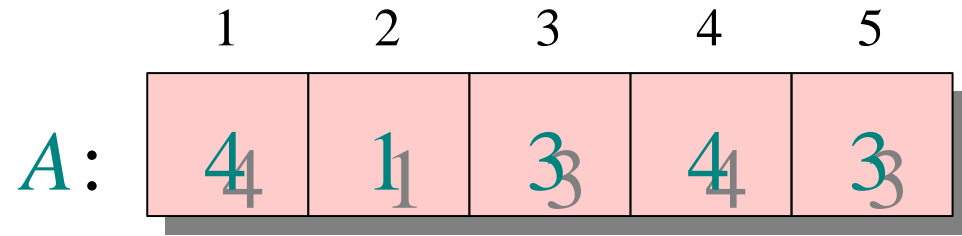
for $i \leftarrow 2$ **to** k

do $C[i] \leftarrow C[i] + C[i-1]$

$\triangleright C[i] = |\{\text{key} \leq i\}|$



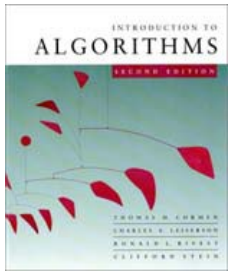
Loop 3



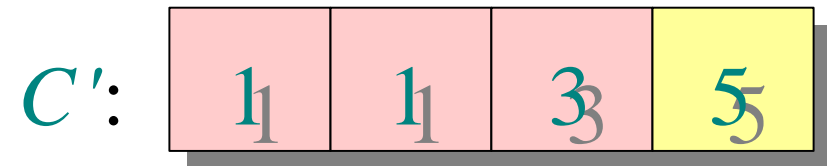
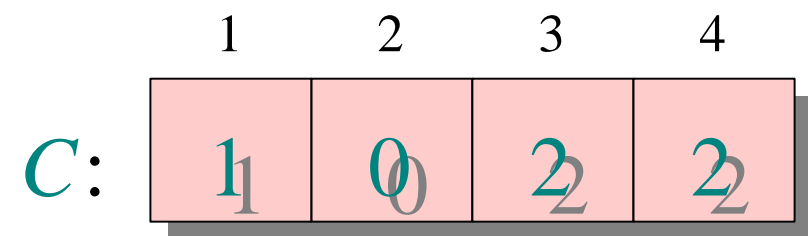
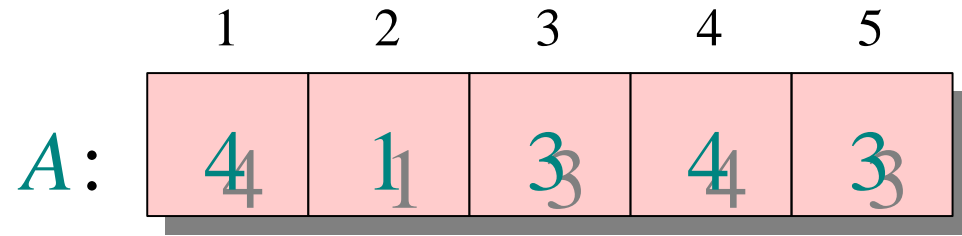
for $i \leftarrow 2$ **to** k

do $C[i] \leftarrow C[i] + C[i-1]$

$\triangleright C[i] = |\{\text{key} \leq i\}|$



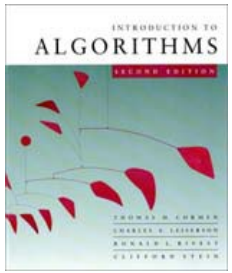
Loop 3



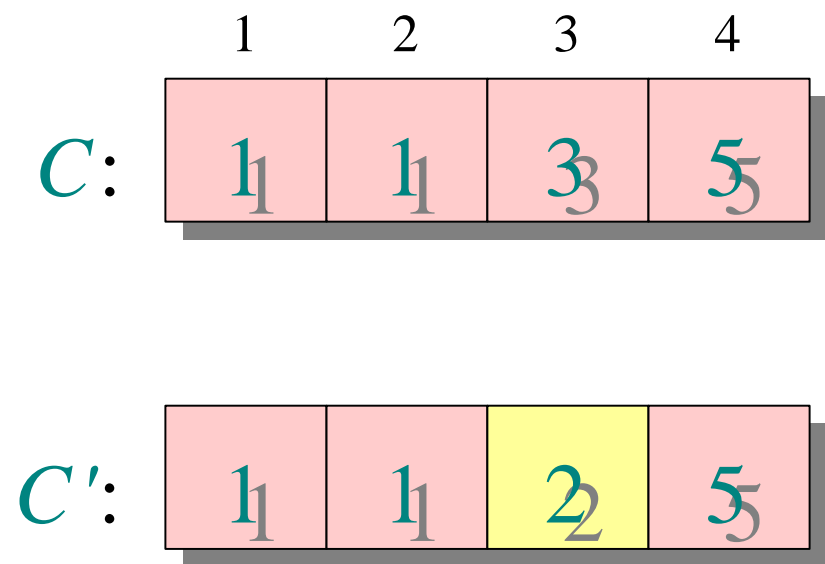
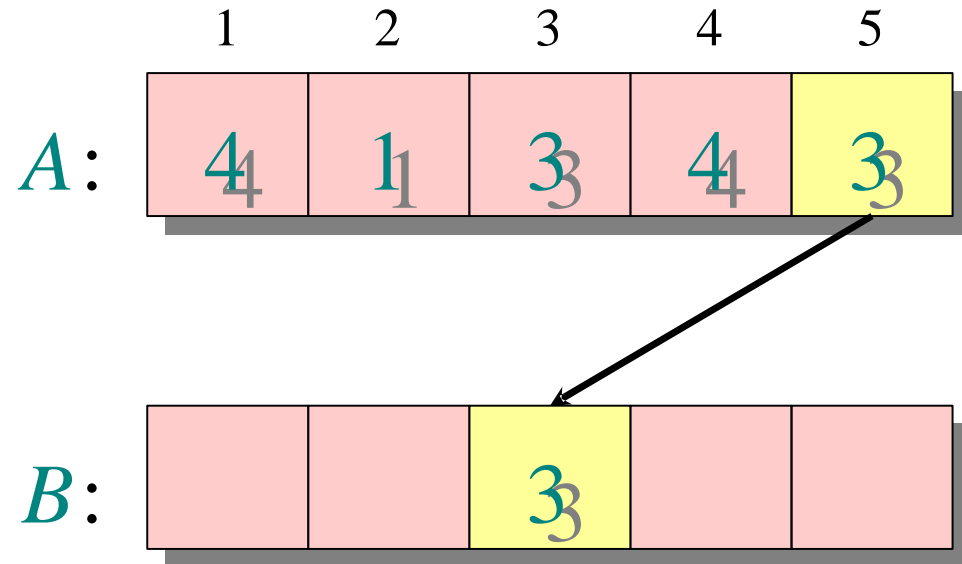
for $i \leftarrow 2$ **to** k

do $C[i] \leftarrow C[i] + C[i-1]$

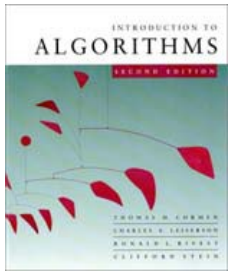
$\triangleright C[i] = |\{\text{key} \leq i\}|$



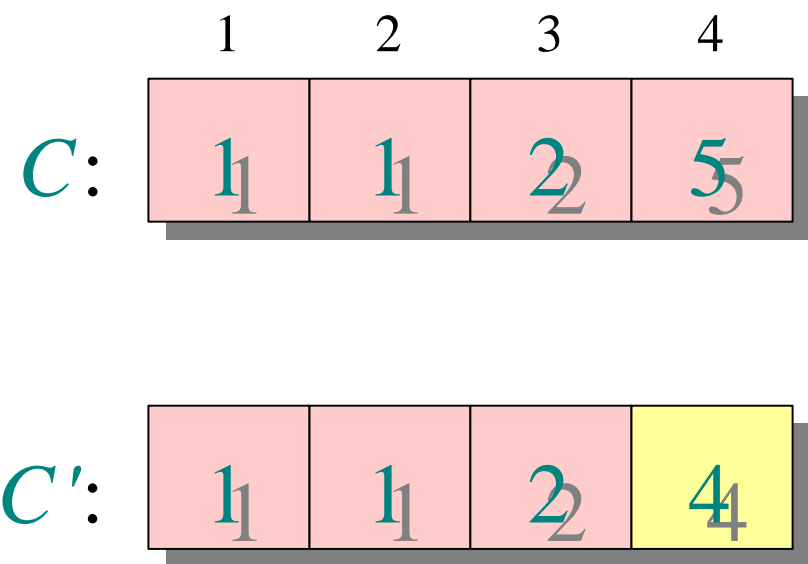
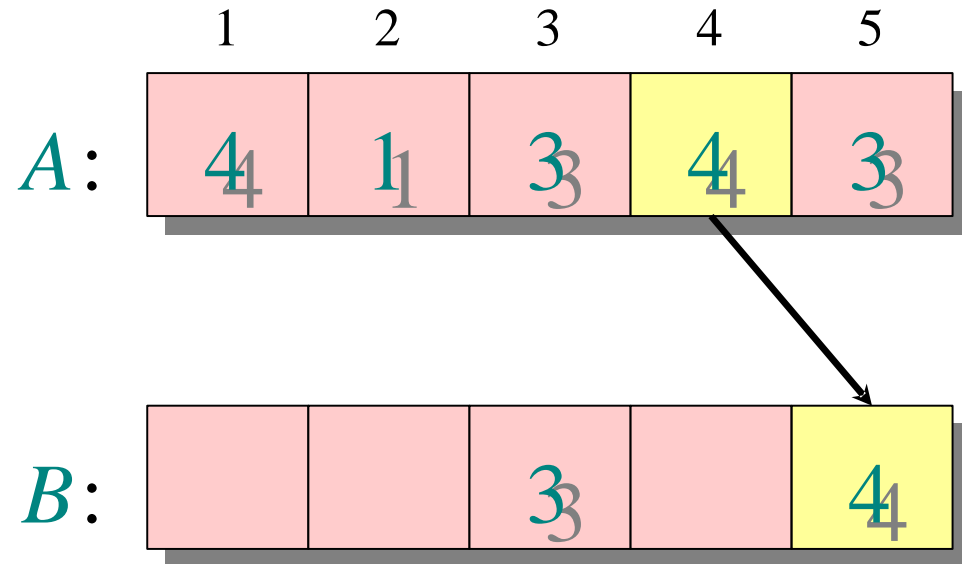
Loop 4



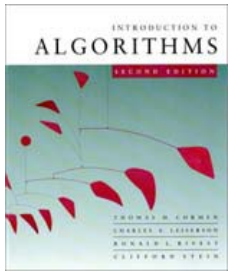
```
for  $j \leftarrow n$  downto 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



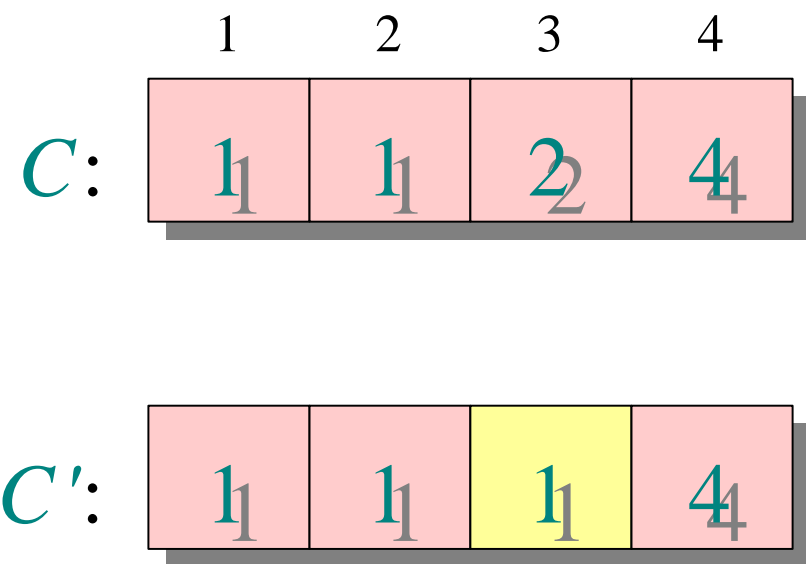
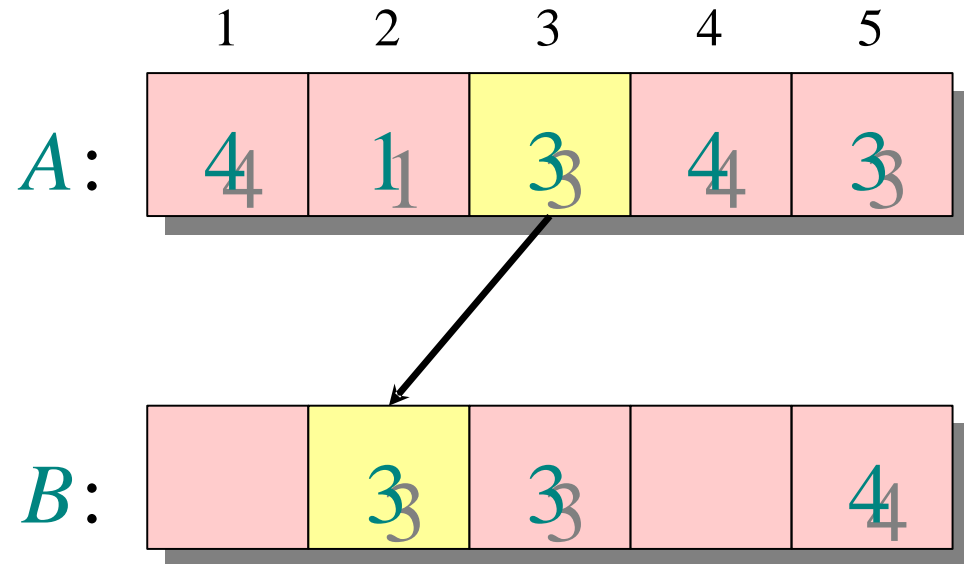
Loop 4



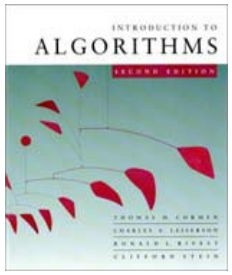
```
for  $j \leftarrow n$  downto 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



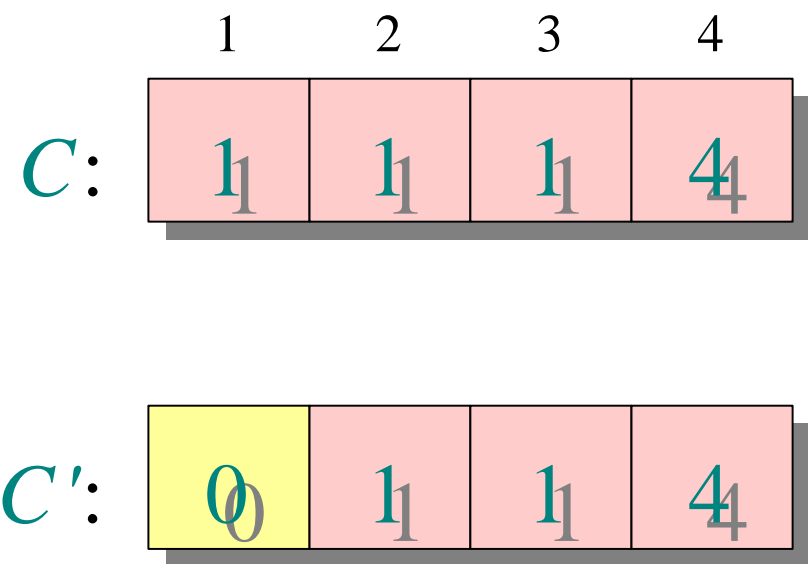
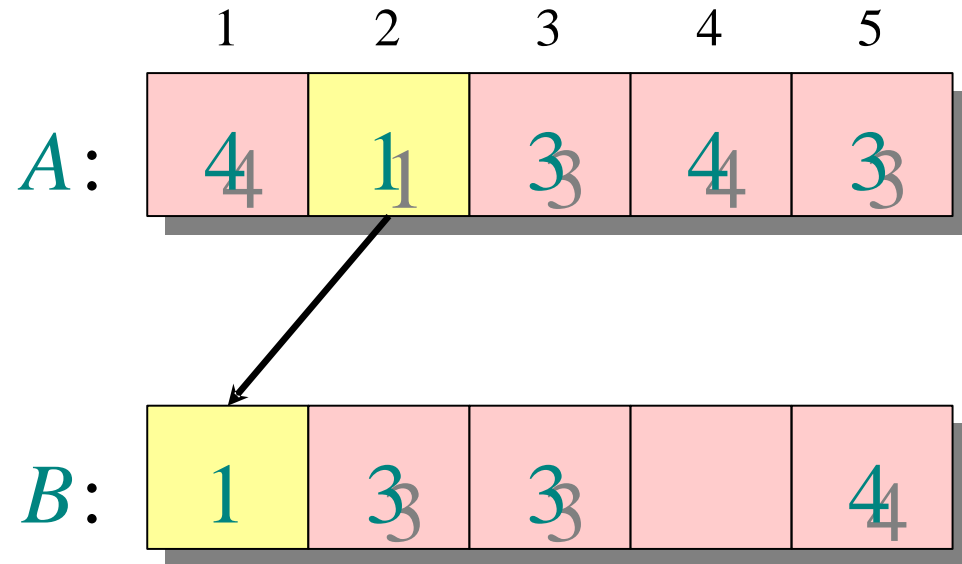
Loop 4



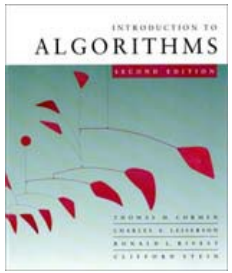
```
for  $j \leftarrow n$  downto 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



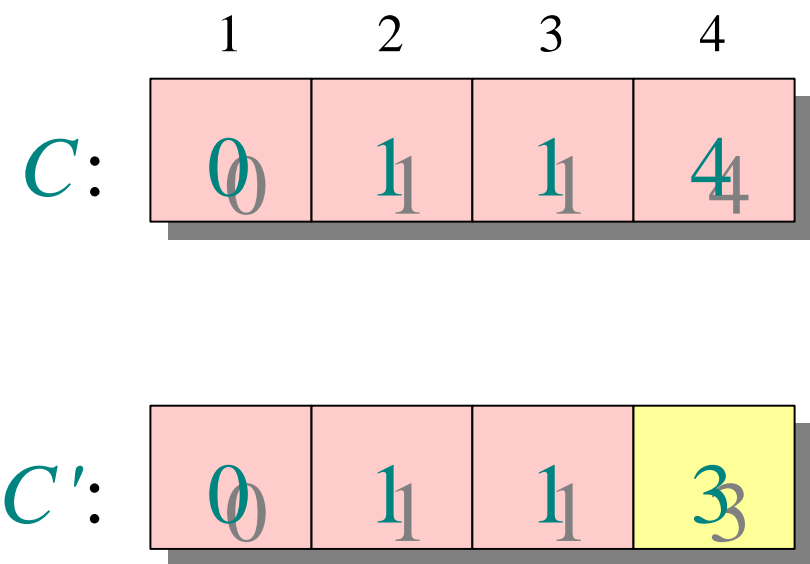
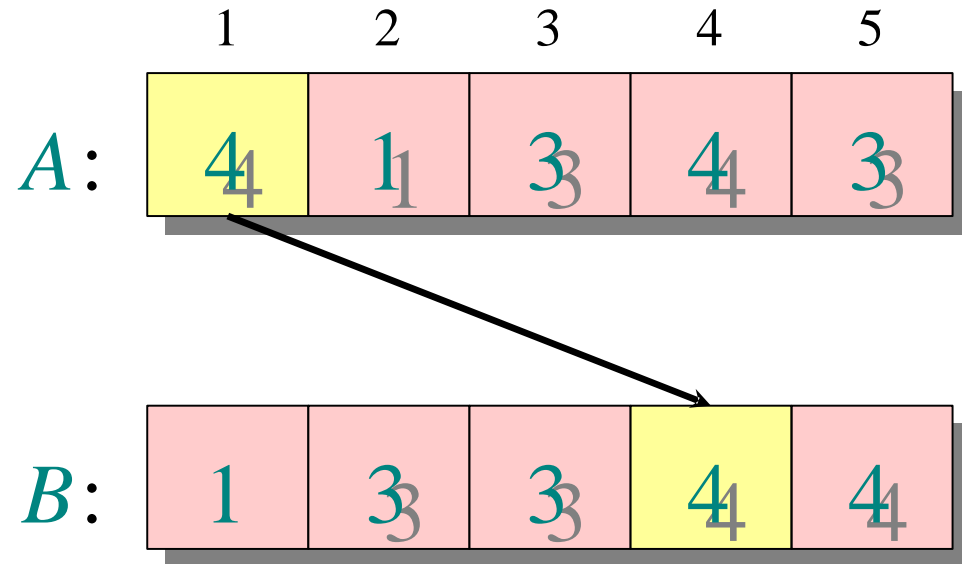
Loop 4



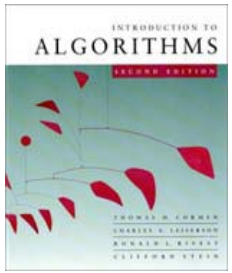
```
for  $j \leftarrow n$  downto 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



Loop 4



```
for  $j \leftarrow n$  downto 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



Analysis

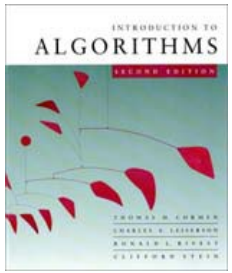
$\Theta(k)$ { **for** $i \leftarrow 1$ **to** k
 do $C[i] \leftarrow 0$

$\Theta(n)$ { **for** $j \leftarrow 1$ **to** n
 do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ { **for** $i \leftarrow 2$ **to** k
 do $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$ { **for** $j \leftarrow n$ **downto** 1
 do $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

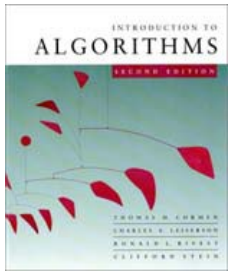
$\Theta(n + k)$



Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?



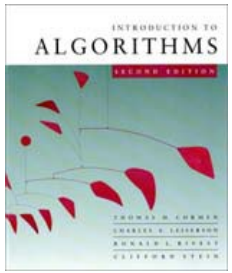
Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

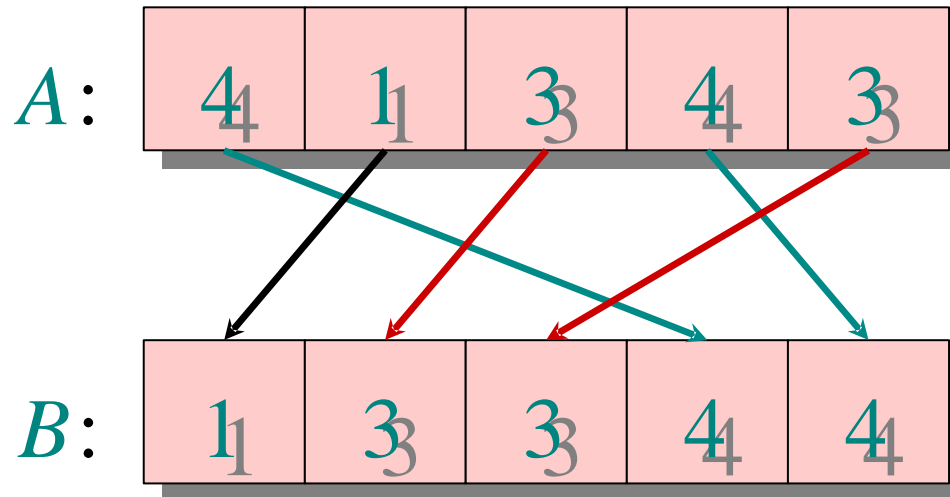
Answer:

- **Comparison sorting** takes $\Omega(n \lg n)$ time.
- Counting sort is not a **comparison sort**.
- In fact, not a single comparison between elements occurs!

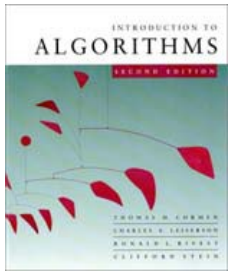


Stable sorting

Counting sort is a **stable** sort: it preserves the input order among equal elements.



Exercise: What other sorts have this property?



Radix sort

- **Origin:** Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on **least-significant digit first** with auxiliary **stable** sort.

Operation of Radix Sort

3 2 9

4 5 7

6 5 7

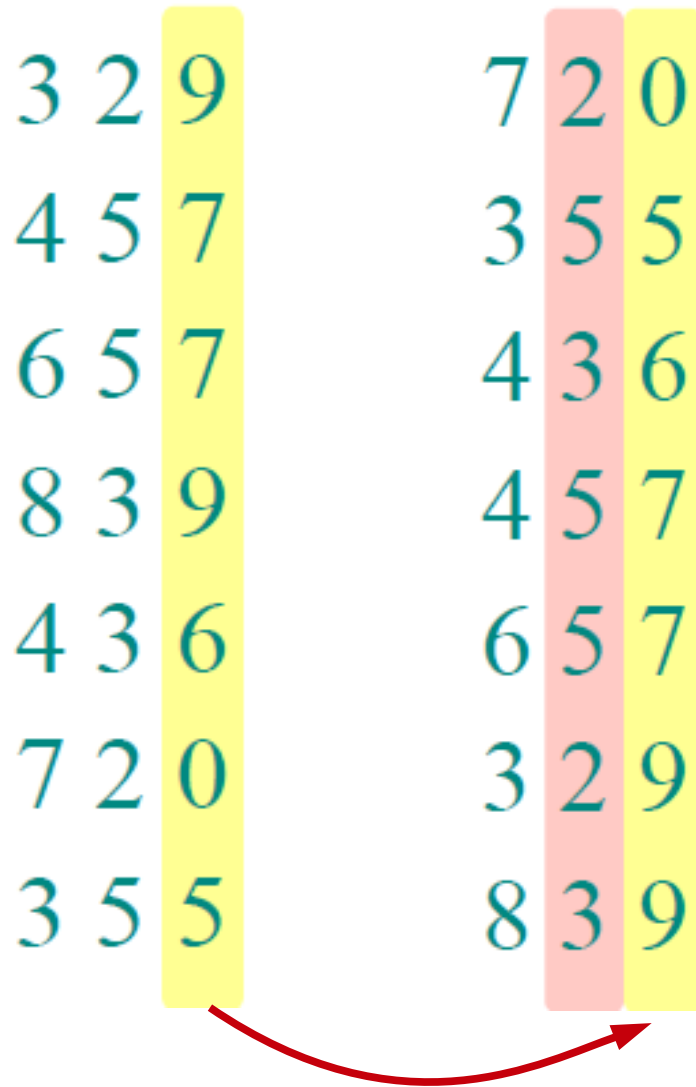
8 3 9

4 3 6

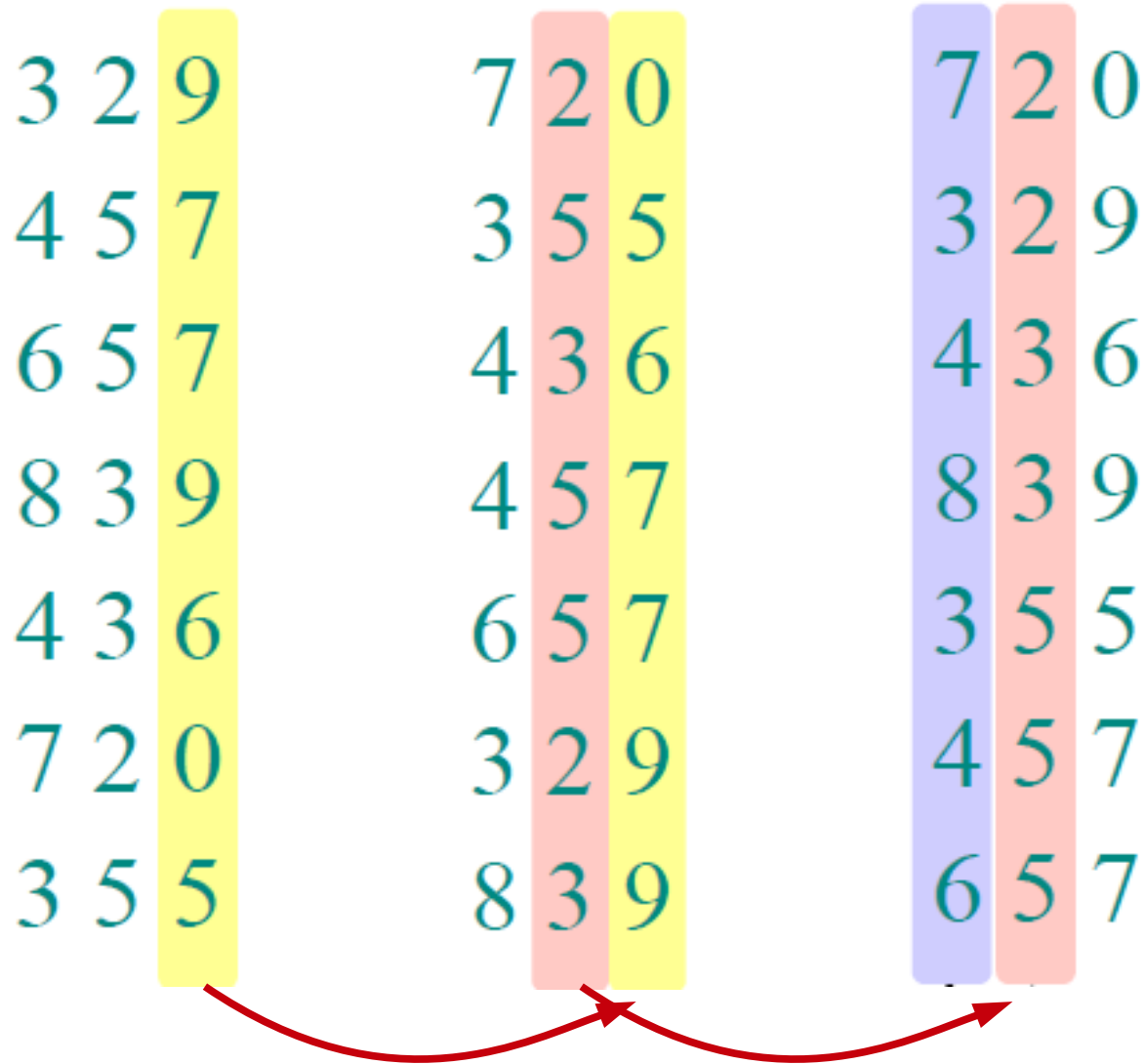
7 2 0

3 5 5

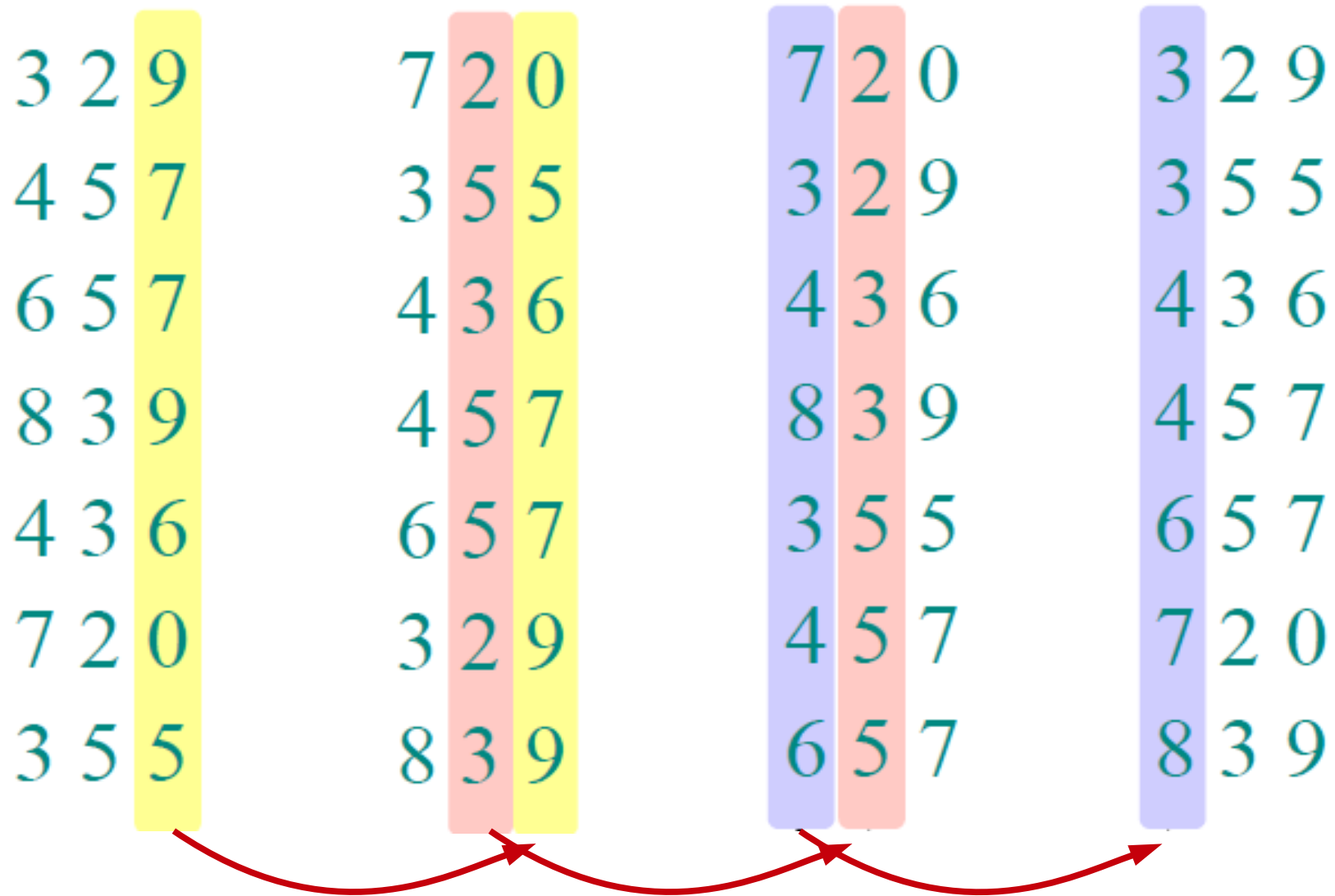
Operation of Radix Sort

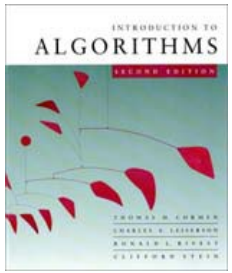


Operation of Radix Sort



Operation of Radix Sort

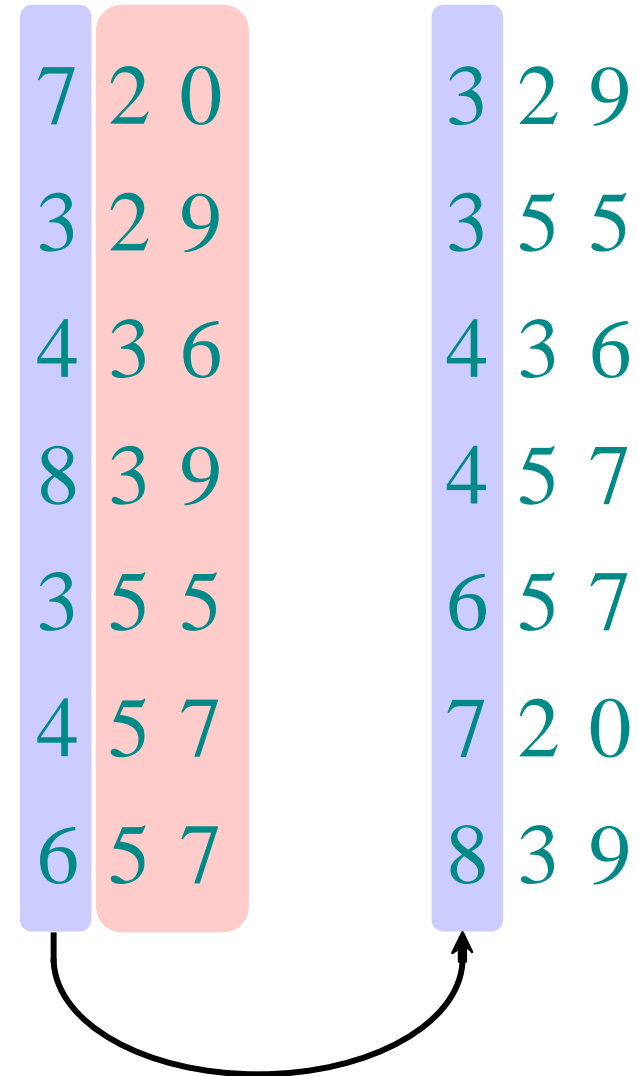


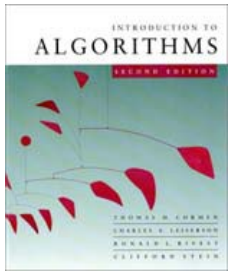


Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t

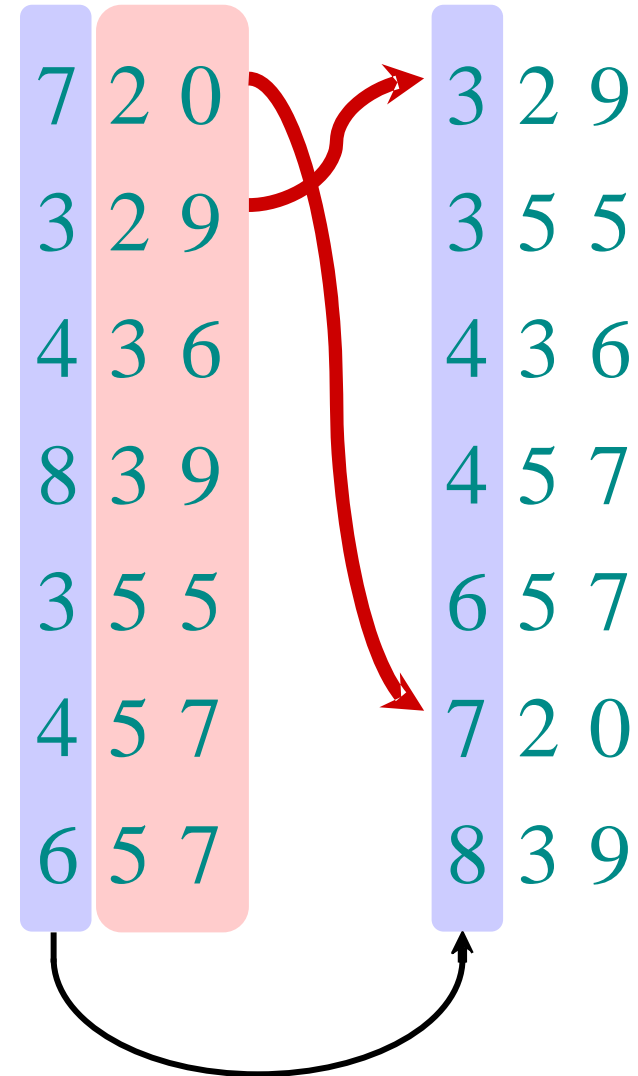


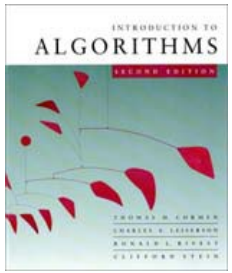


Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
Two numbers that differ in digit t are correctly sorted.





Correctness of radix sort

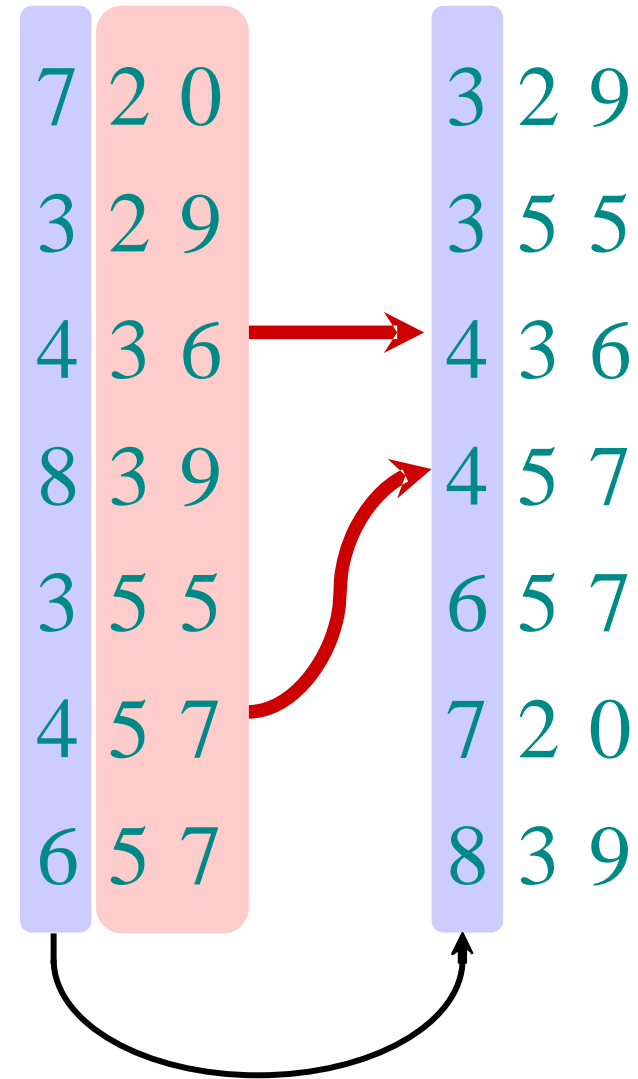
Induction on digit position

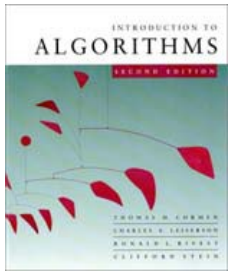
- Assume that the numbers are sorted by their low-order $t - 1$ digits.

- Sort on digit t

Two numbers that differ in digit t are correctly sorted.

Two numbers equal in digit t are put in the same order as the input \Rightarrow correct order.

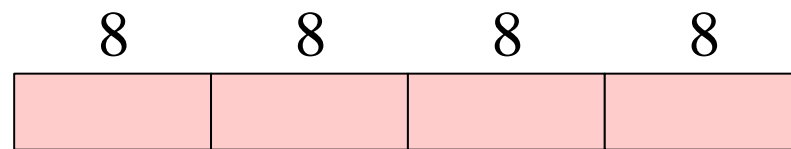




Analysis of radix sort

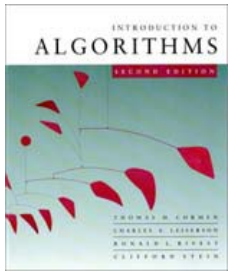
- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base- 2^r digits.

Example: 32-bit word



$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base- 2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base- 2^{16} digits.

How many passes should we make?



Analysis (continued)

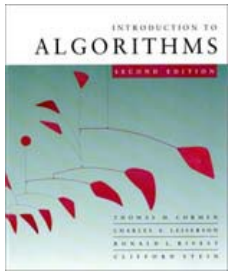
Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

If each b -bit word is broken into r -bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have

$$T(n, b) = \Theta\left(\frac{b}{r}(n + 2^r)\right).$$

Choose r to minimize $T(n, b)$:

- Increasing r means fewer passes, but as $r \gg \lg n$, the time grows exponentially.



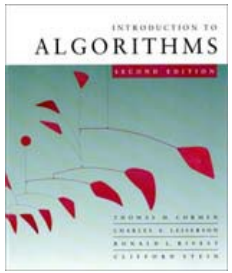
Choosing r

$$T(n, b) = \Theta\left(\frac{b}{r}(n + 2^r)\right)$$

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don't want $2^r \gg n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.

Choosing $r = \lg n$ implies $T(n, b) = \Theta(bn/\lg n)$.



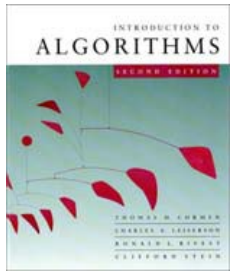
Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers): $r = \lg n \approx 11$ and passes = $b / r \approx 3$

- At most 3 passes when sorting ≥ 2000 numbers.
- Merge sort and quicksort do at least $\lceil \lg 2000 \rceil = 11$ passes.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.



Order statistics

Select the i th smallest of n elements (the element with *rank* i).

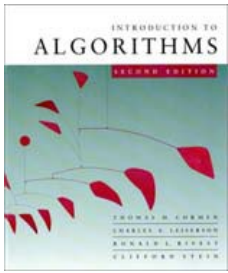
- $i = 1$: *minimum*;
- $i = n$: *maximum*;
- $i = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$: *median*.

Naive algorithm: Sort and index i th element.

$$\begin{aligned}\text{Worst-case running time} &= \Theta(n \lg n) + \Theta(1) \\ &= \Theta(n \lg n),\end{aligned}$$

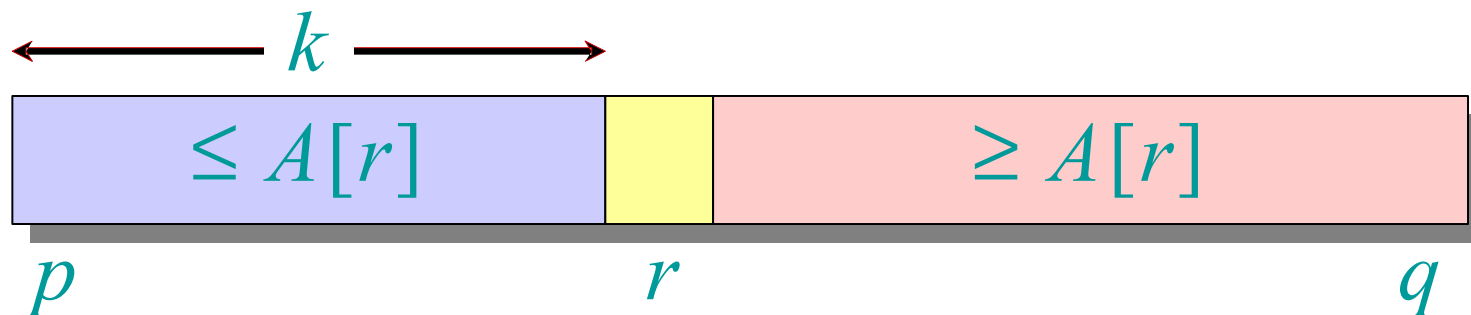
using merge sort or heapsort (*not* quicksort).

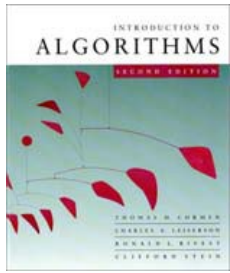
Can we do better?



Randomized divide-and-conquer algorithm

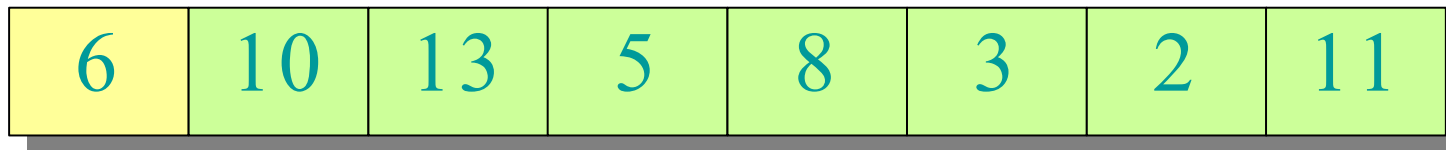
RAND-SELECT(A, p, q, i) \triangleright i th smallest of $A[p..q]$
if $p = q$ **then return** $A[p]$
 $r \leftarrow$ **RAND-PARTITION**(A, p, q)
 $k \leftarrow r - p + 1$ $\triangleright k = \text{rank}(A[r])$
if $i = k$ **then return** $A[r]$
if $i < k$
then return **RAND-SELECT**($A, p, r - 1, i$)
else return **RAND-SELECT**($A, r + 1, q, i - k$)





Example

Select the $i = 7$ th smallest:



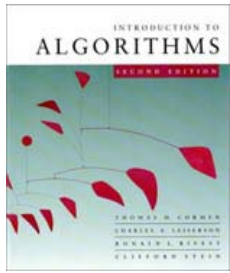
$i = 7$

pivot

Partition:

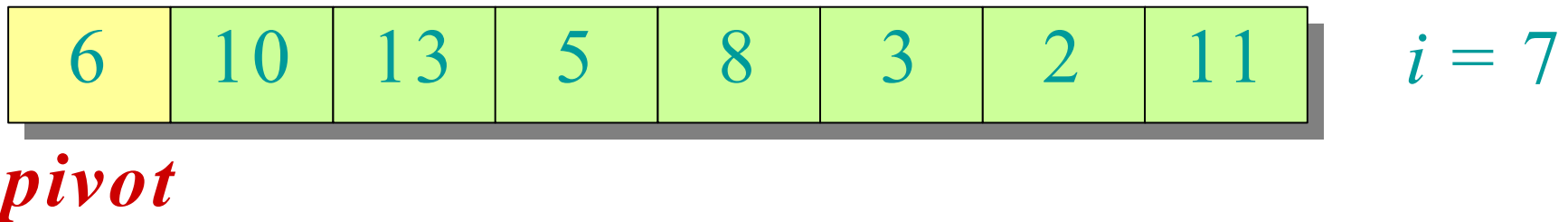


$k = 4$

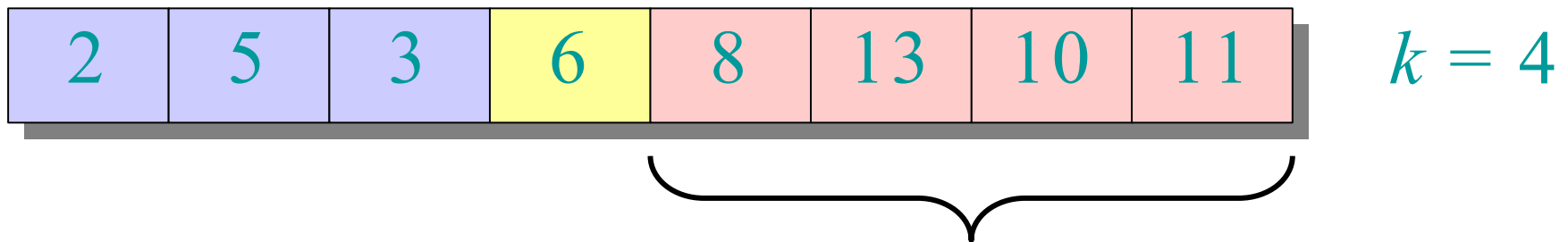


Example

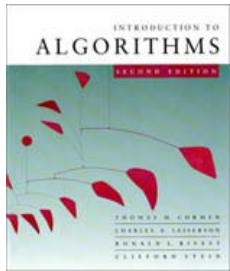
Select the $i = 7$ th smallest:



Partition:



Select the $7 - 4 = 3$ rd smallest recursively.



Intuition for analysis

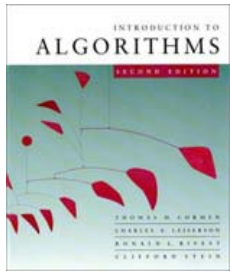
(All our analyses today assume that all elements are distinct.)

Lucky:

$$\begin{aligned} T(n) &= T(9n/10) + \Theta(n) \\ &= \Theta(n) \end{aligned}$$

$$n^{\log_{10/9} 1} = n^0 = 1$$

CASE 3



Intuition for analysis

(All our analyses today assume that all elements are distinct.)

Lucky:

$$\begin{aligned} T(n) &= T(9n/10) + \Theta(n) \\ &= \Theta(n) \end{aligned}$$

$$n^{\log_{10/9} 1} = n^0 = 1$$

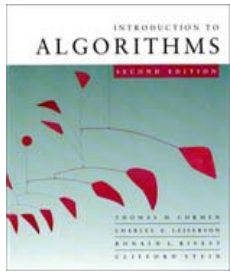
CASE 3

Unlucky:

$$\begin{aligned} T(n) &= T(n - 1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

arithmetic series

Worse than sorting!



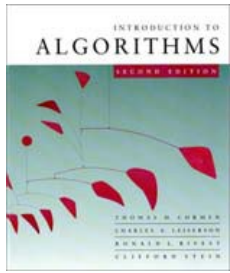
Analysis of expected time

The analysis follows that of randomized quicksort, but it's a little different.

Let $T(n)$ = the random variable for the running time of RAND-SELECT on an input of size n , assuming random numbers are independent.

For $k = 0, 1, \dots, n-1$, define the *indicator random variable*

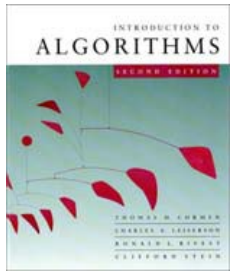
$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$



Analysis (continued)

To obtain an upper bound, assume that the i th element always falls in the larger side of the partition:

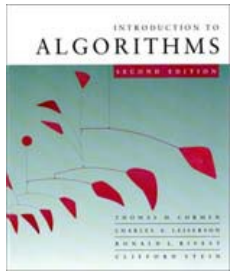
$$T(n) = \begin{cases} T(\max\{0, n-1\}) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(\max\{1, n-2\}) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(\max\{n-1, 0\}) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$
$$= \sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)).$$



Calculating expectation

$$E[T(n)] = E \left[\sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right]$$

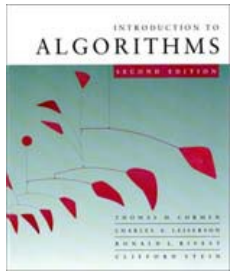
Take expectations of both sides.



Calculating expectation

$$\begin{aligned} E[T(n)] &= E \left[\sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right] \\ &= \sum_{k=0}^{n-1} E[X_k (T(\max\{k, n-k-1\}) + \Theta(n))] \end{aligned}$$

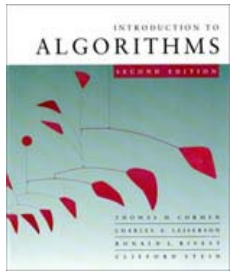
Linearity of expectation.



Calculating expectation

$$\begin{aligned} E[T(n)] &= E \left[\sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right] \\ &= \sum_{k=0}^{n-1} E[X_k (T(\max\{k, n-k-1\}) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)] \end{aligned}$$

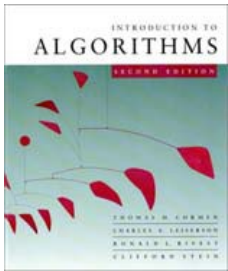
Independence of X_k from other random choices.



Calculating expectation

$$\begin{aligned} E[T(n)] &= E \left[\sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right] \\ &= \sum_{k=0}^{n-1} E[X_k (T(\max\{k, n-k-1\}) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \end{aligned}$$

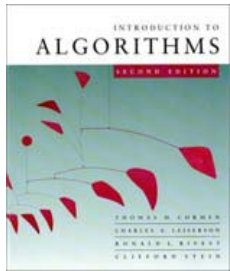
Linearity of expectation; $E[X_k] = 1/n$.



Calculating expectation

$$\begin{aligned} E[T(n)] &= E \left[\sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right] \\ &= \sum_{k=0}^{n-1} E[X_k (T(\max\{k, n-k-1\}) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \\ &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n) \end{aligned}$$

Upper terms
appear twice.



Hairy recurrence

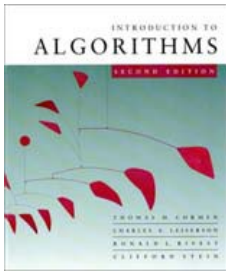
(But not quite as hairy as the quicksort one.)

$$E[T(n)] = \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n)$$

Prove: $E[T(n)] \leq cn$ for constant $c > 0$.

- The constant c can be chosen large enough so that $E[T(n)] \leq cn$ for the base cases.

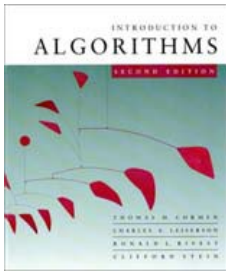
Use fact: $\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \leq \frac{3}{8}n^2$ (exercise).



Substitution method

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n)$$

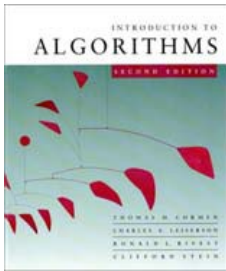
Substitute inductive hypothesis.



Substitution method

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \\ &\leq \frac{2c}{n} \left(\frac{3}{8} n^2 \right) + \Theta(n) \end{aligned}$$

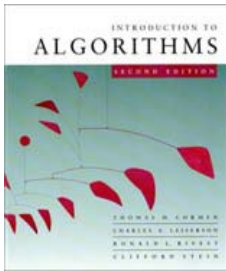
Use fact.



Substitution method

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \\ &\leq \frac{2c}{n} \left(\frac{3}{8} n^2 \right) + \Theta(n) \\ &= cn - \left(\frac{cn}{4} - \Theta(n) \right) \end{aligned}$$

Express as *desired – residual*.



Substitution method

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \\ &\leq \frac{2c}{n} \left(\frac{3}{8} n^2 \right) + \Theta(n) \\ &= cn - \left(\frac{cn}{4} - \Theta(n) \right) \\ &\leq cn, \end{aligned}$$

if c is chosen large enough so that $cn/4$ dominates the $\Theta(n)$.

Summary

- Randomized Order Statistics
 - Works fast in practice
 - $\Theta(n)$ time on average
 - Worst case very bad $\Theta(n^2)$

Recap

- Sorting Lower Bound
- Linear Time Sorting
 - Counting Sort
 - Radix Sort
- Order Statistics
- Next:
 - Hashing