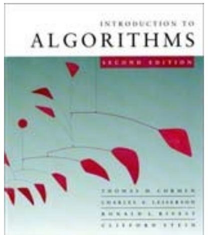
CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya



How fast can we sort?

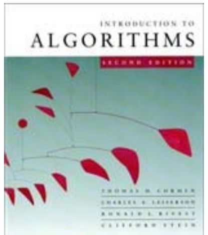
All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.

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

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

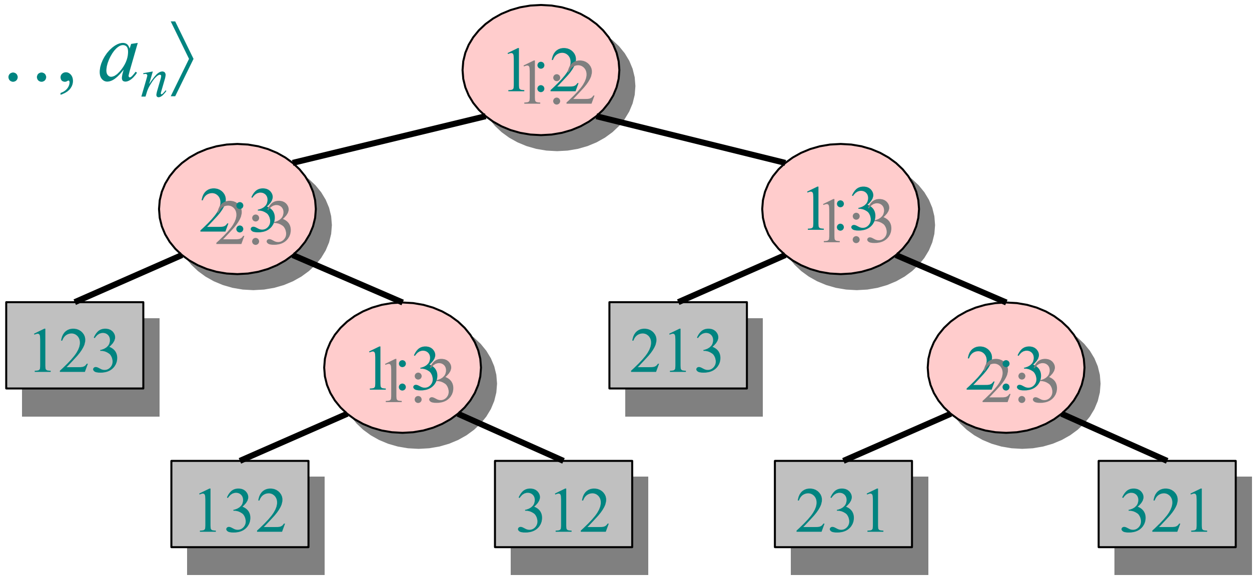
Is $O(n \lg n)$ the best we can do?

Decision trees can help us answer this question.



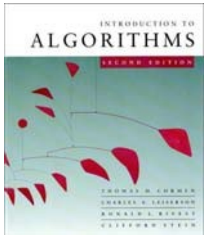
Decision-tree example

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



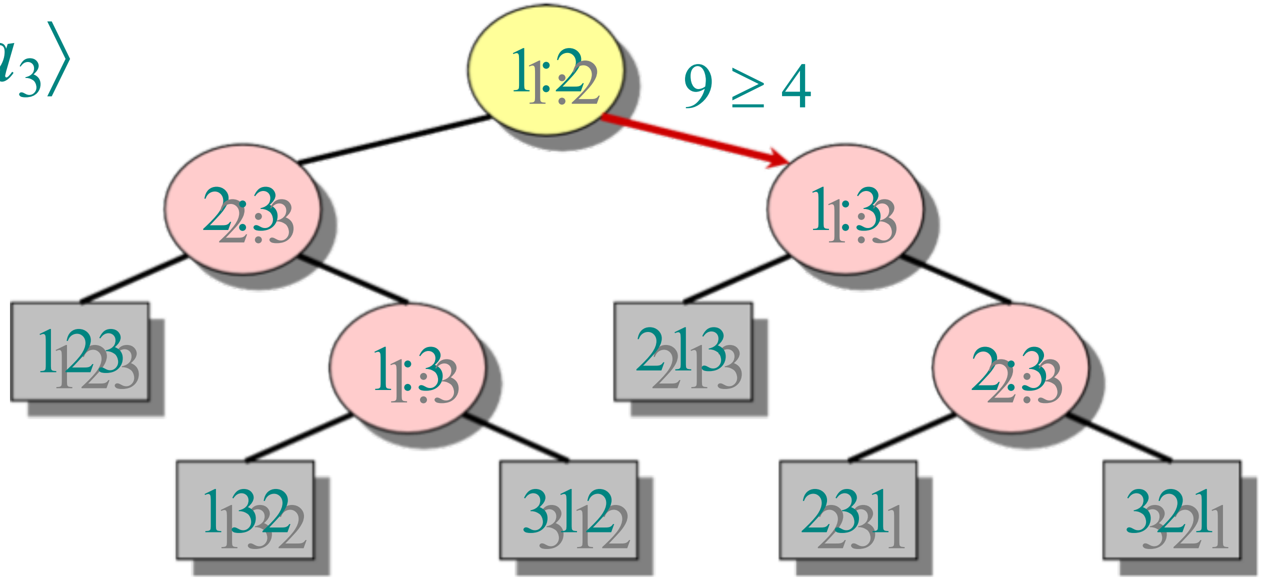
Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.



Decision-tree example

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

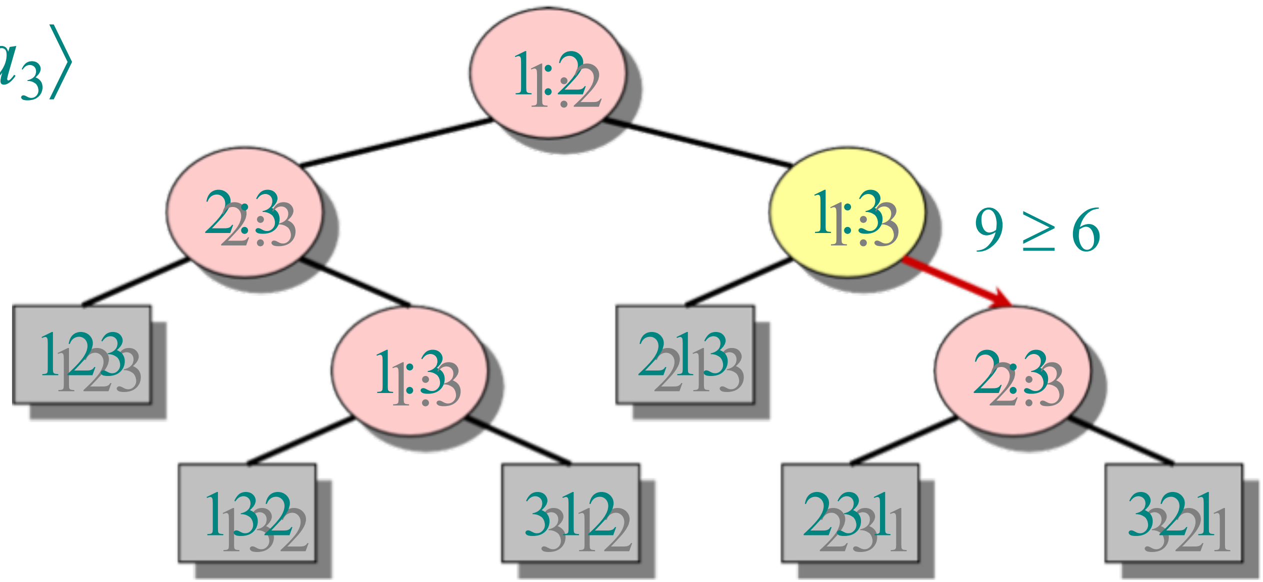


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

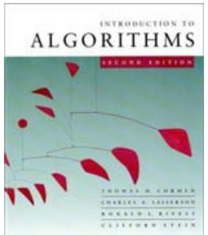
Decision-tree example

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



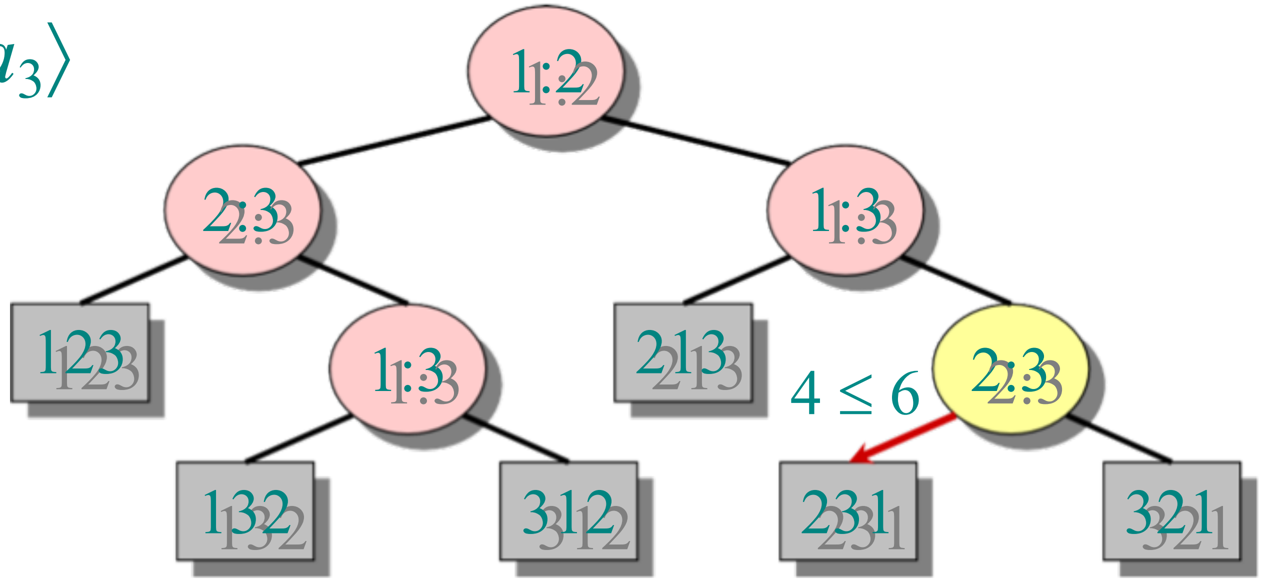
Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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



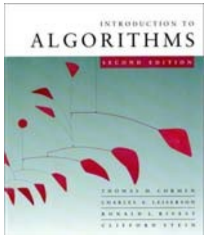
Decision-tree example

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



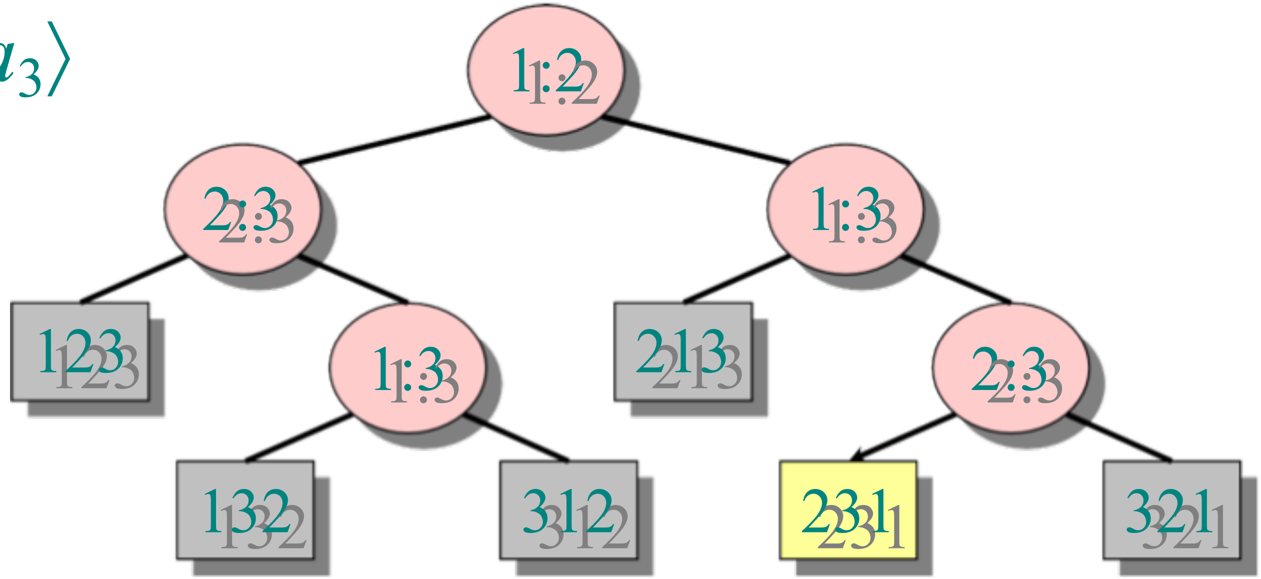
Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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



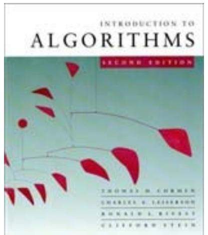
Decision-tree example

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



$4 \le 6 \le 9$

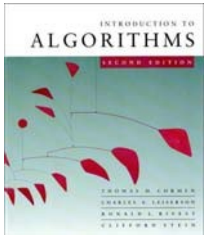
Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \le a_{\pi(2)} \le$ L $\le a_{\pi(n)}$ has been established.



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.

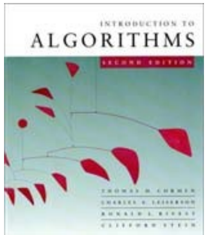


Lower bound for decision- tree sorting

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

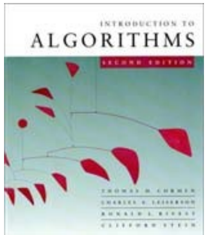
Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

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



Lower bound for comparison sorting

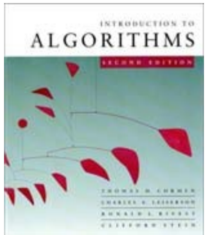
Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms.



Sorting in linear time

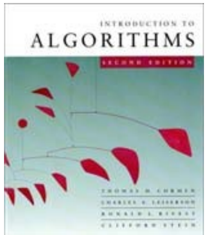
Counting sort: No comparisons between elements.

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



Counting sort

$$
\begin{array}{ll}
\textbf{for } i \leftarrow 1 \textbf{ to } k & \\
\quad \textbf{do } C[i] \leftarrow 0 & \\
\textbf{for } j \leftarrow 1 \textbf{ to } n & \\
\quad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 & \triangleleft\, C[i] = |\{\text{key} = i\}| \\
\textbf{for } i \leftarrow 2 \textbf{ to } k & \\
\quad \textbf{do } C[i] \leftarrow C[i] + C[i\text{–}1] & \triangleleft\, C[i] = |\{\text{key} \le i\}| \\
\textbf{for } j \leftarrow n \textbf{ downto } 1 & \\
\quad \textbf{do } B[C[A[j]]] \leftarrow \mathrm{A}[j] & \\
\quad\quad C[A[j]] \leftarrow C[A[j]] - 1 &
\end{array}
$$

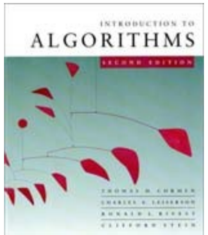


Counting-sort example

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:				

B:					



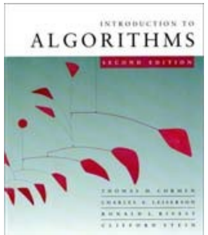
Loop 1

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	0

B:					

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

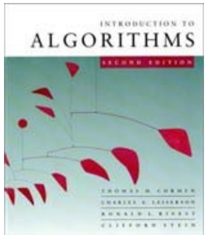
C:

1	2	3	4
0	0	0	1

B:

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

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

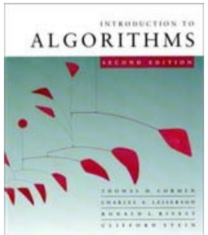
C:

1	2	3	4
1	0	0	1

B:

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

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



Loop 2

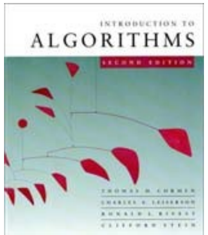
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	1

B:					

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

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

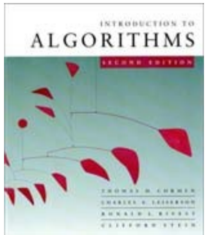
C:

1	2	3	4
1	0	1	2

B:

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

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



Loop 2

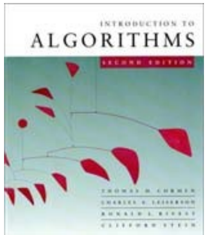
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

B:					

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

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



Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

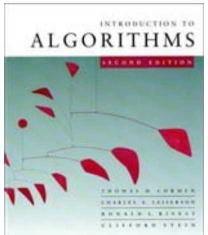
B:

C':

1	1	2	2

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

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



Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

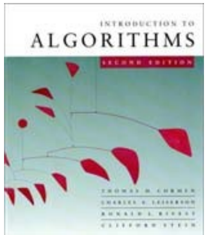
B:

C':

1	1	3	2

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

do $C[i] \leftarrow C[i] + C[i–1]$ $\quad \triangleleft C[i] = |\{\text{key} \leq i\}|$



Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

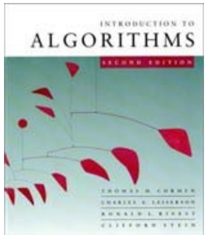
B:

C':

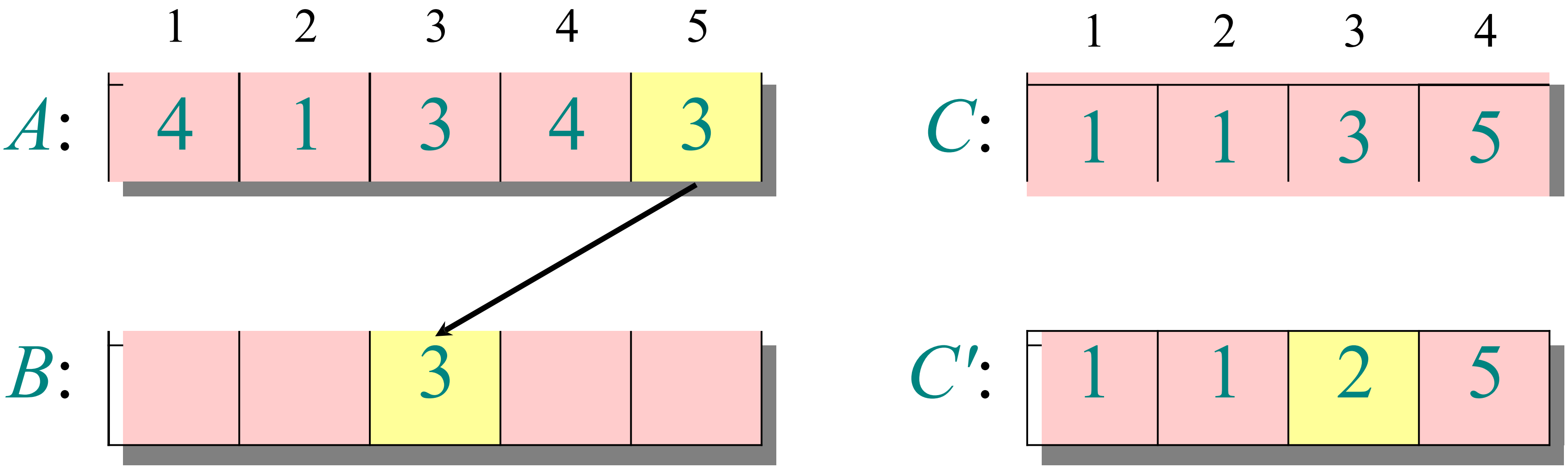
1	1	3	5

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

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



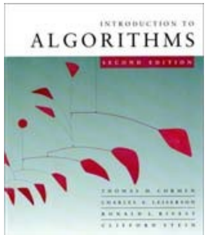
Loop 4



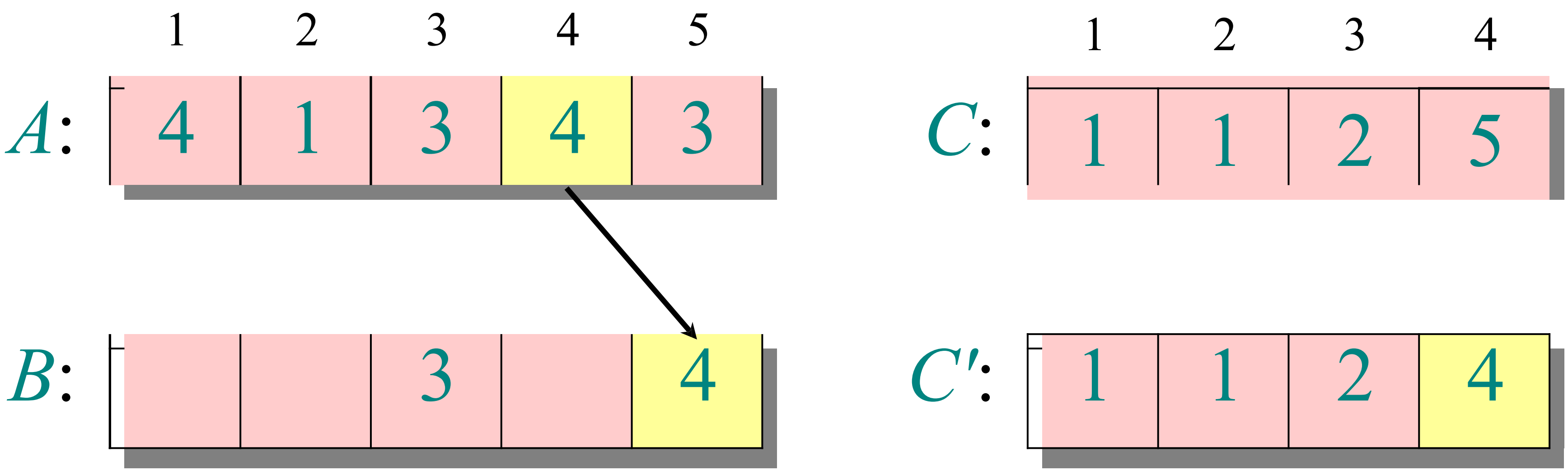
for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

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



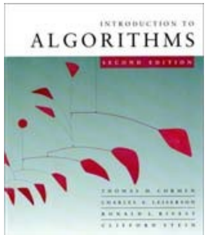
Loop 4



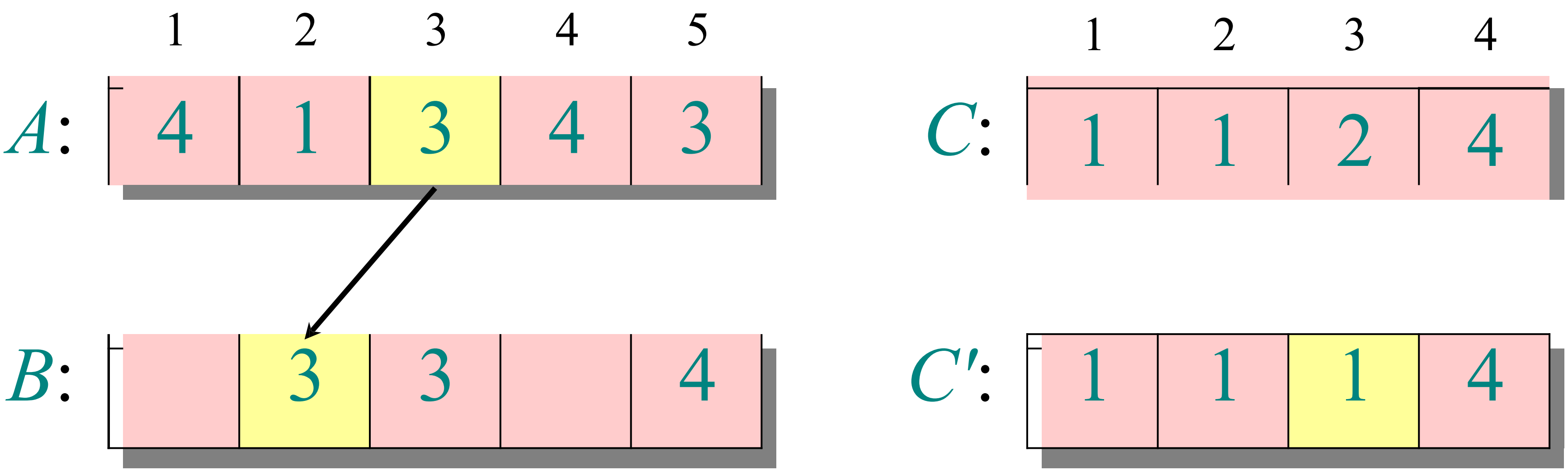
for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

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



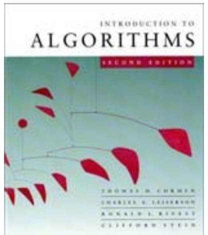
Loop 4



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

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

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



Loop 4

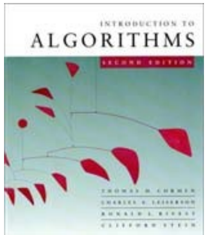
	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3		4

	1	2	3	4
C:	1	1	1	4
C':	0	1	1	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

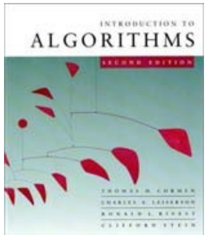
	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3	4	4

	1	2	3	4
C:	0	1	1	4
C':	0	1	1	3

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

  do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

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



Analysis

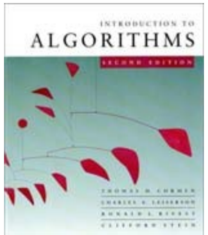
$\Theta(k)$
for $i \leftarrow 1$ **to** k
    do $C[i] \leftarrow 0$

$\Theta(n)$
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 $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$



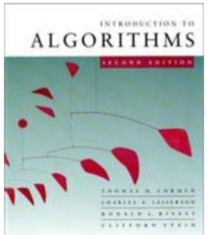
Running time

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

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

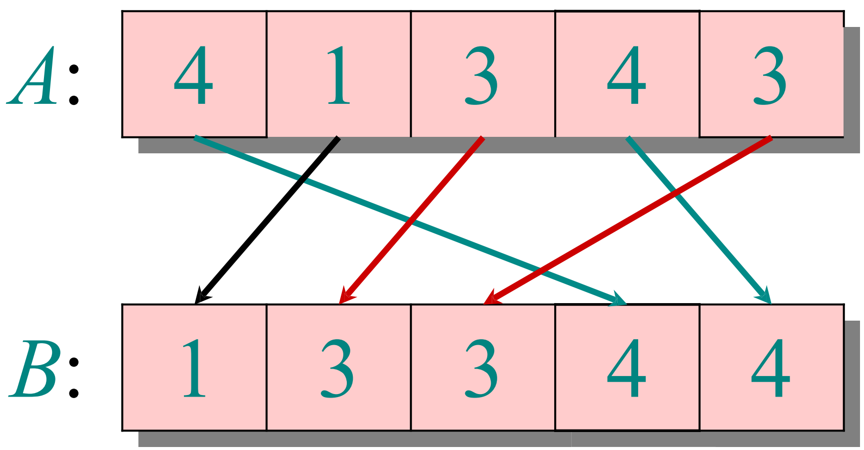
Answer:

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

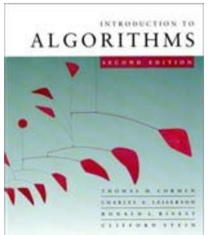


Stable sorting

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

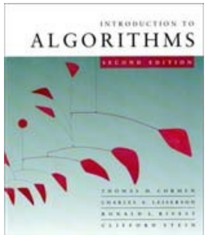


Exercise: What other sorts have this property?



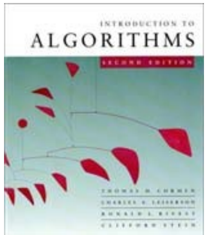
Radix sort

- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census. (See Appendix
- 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	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

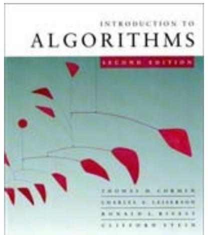


Correctness of radix sort

Induction on digit position

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

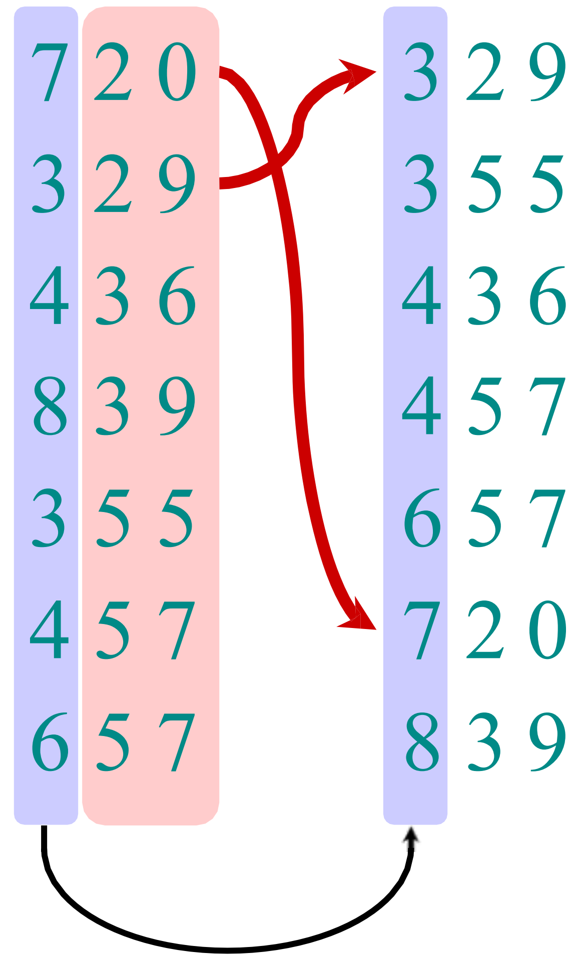
7 2 0	3 2 9
3 2 9	3 5 5
4 3 6	4 3 6
8 3 9	4 5 7
3 5 5	6 5 7
4 5 7	7 2 0
6 5 7	8 3 9

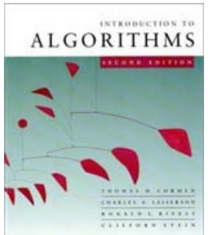


Correctness of radix sort

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.

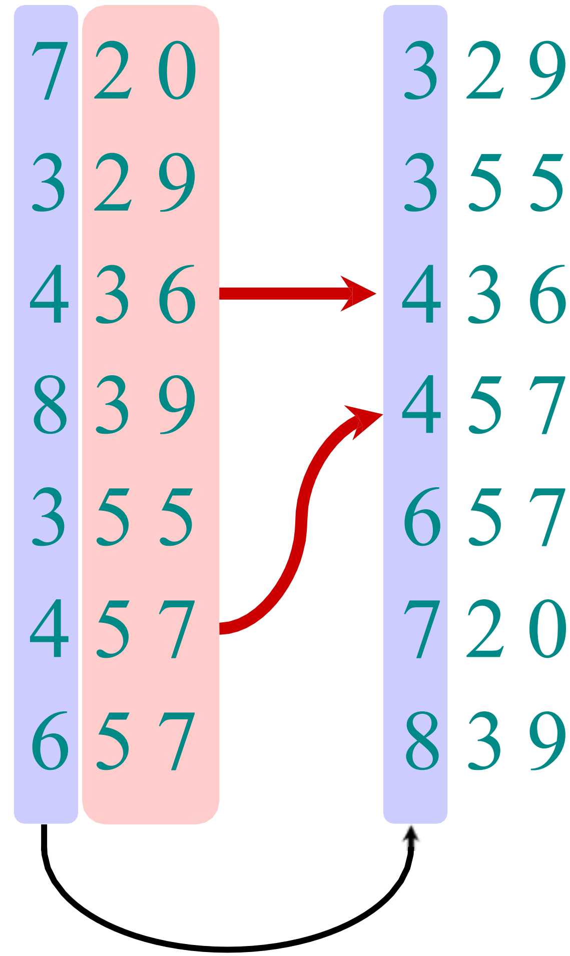


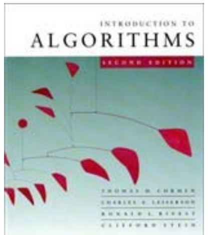


Correctness of radix sort

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.

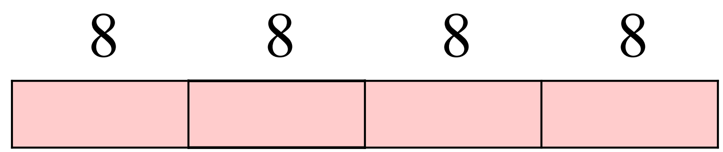




Analysis of radix sort

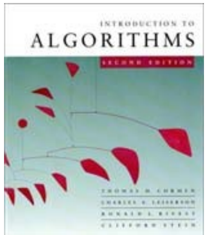
- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

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)

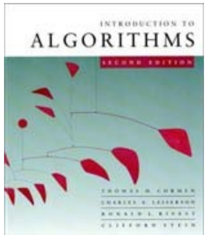
Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

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

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

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

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



Choosing r

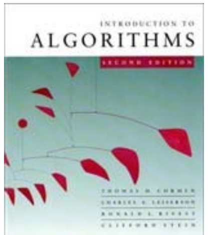
$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right).$$

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

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(d\, n)$ time.



Conclusions

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

Example (32-bit numbers):

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