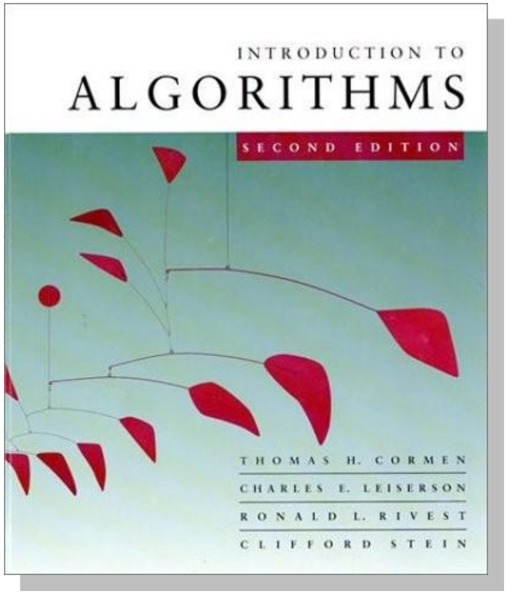
Design and Analysis of Algorithms

Linear Time Sort

Fall 2022

National University of Computer and Emerging Sciences,
Islamabad

Introduction to Algorithms

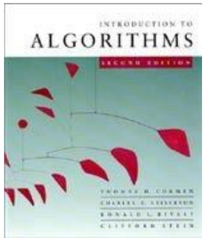


Sorting Lower Bounds

- Decision trees

Linear-Time Sorting

- Counting sort
- Radix sort
- Bucket sort



How fast can we sort?

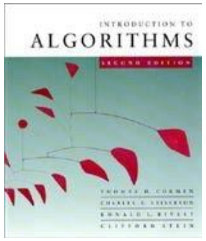
All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.

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

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

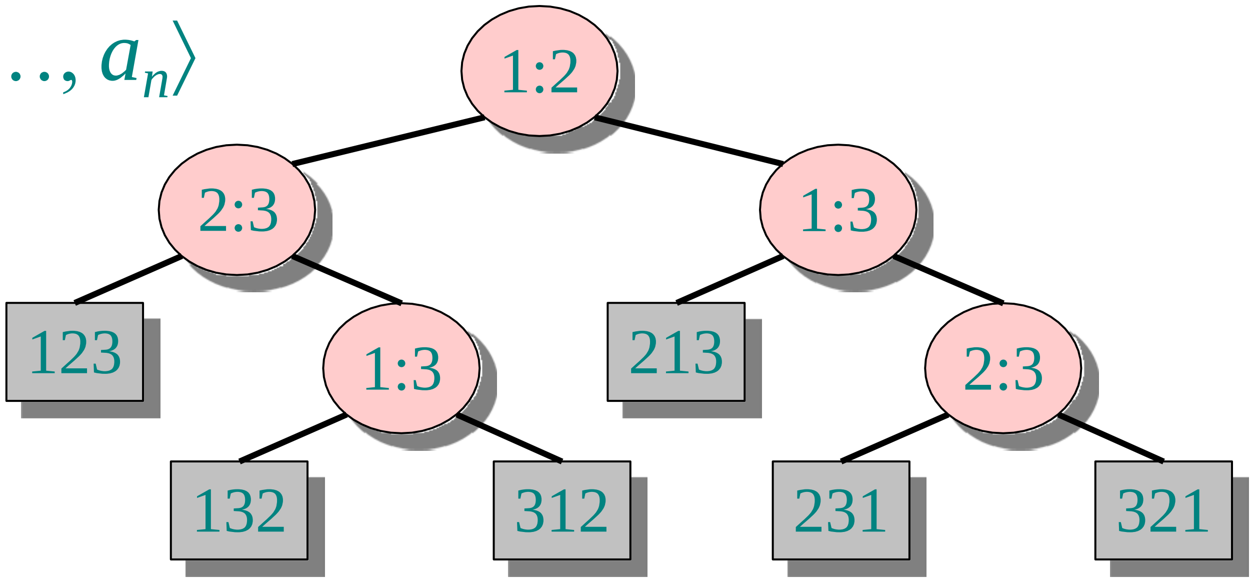
Is O(n lgn) the best we can do?

Decision trees can help us answer this question.



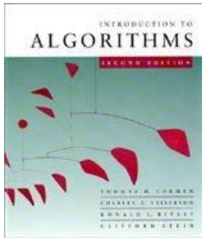
Decision-tree example

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



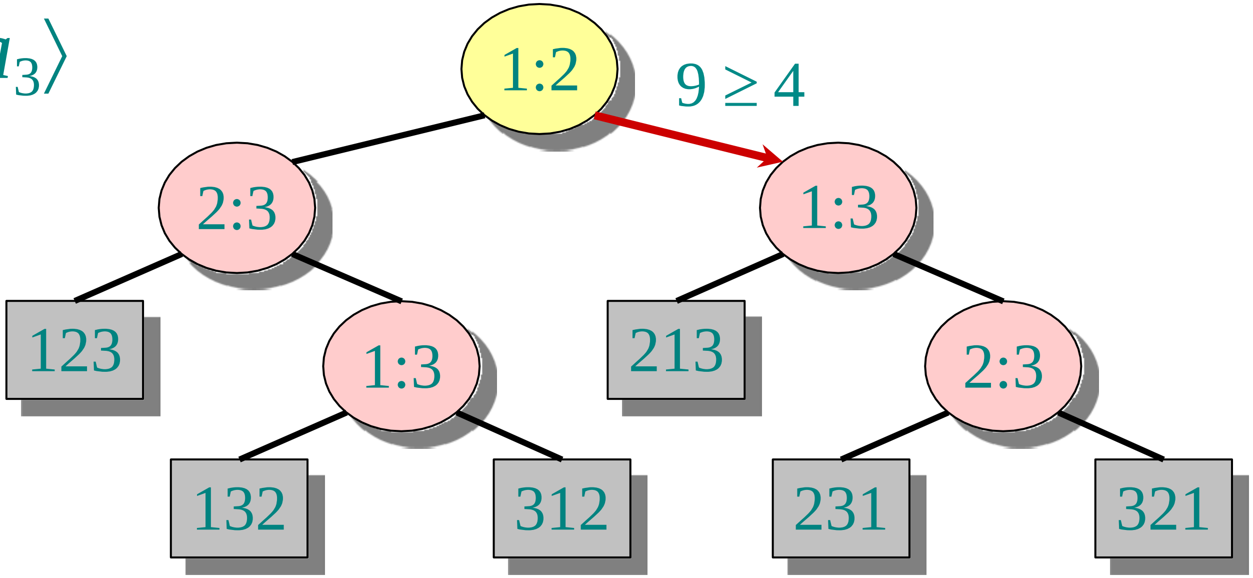
Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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



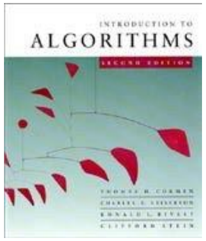
Decision-tree example

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



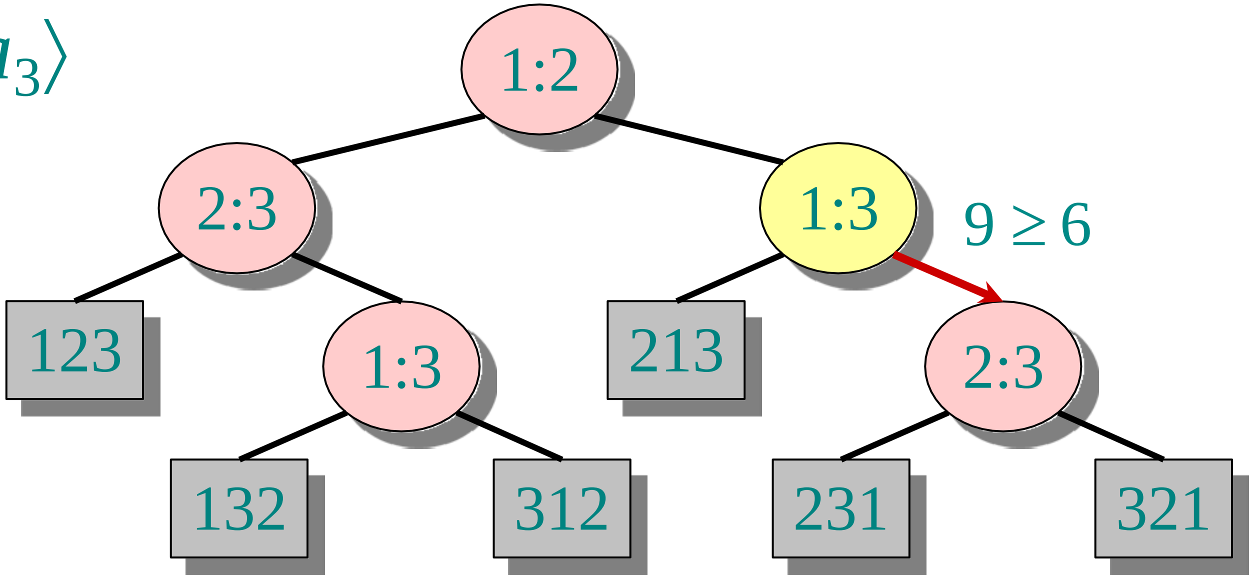
Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2,\dots, n\}$.

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



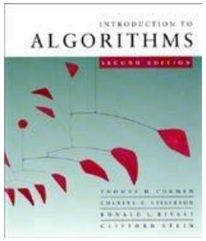
Decision-tree example

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



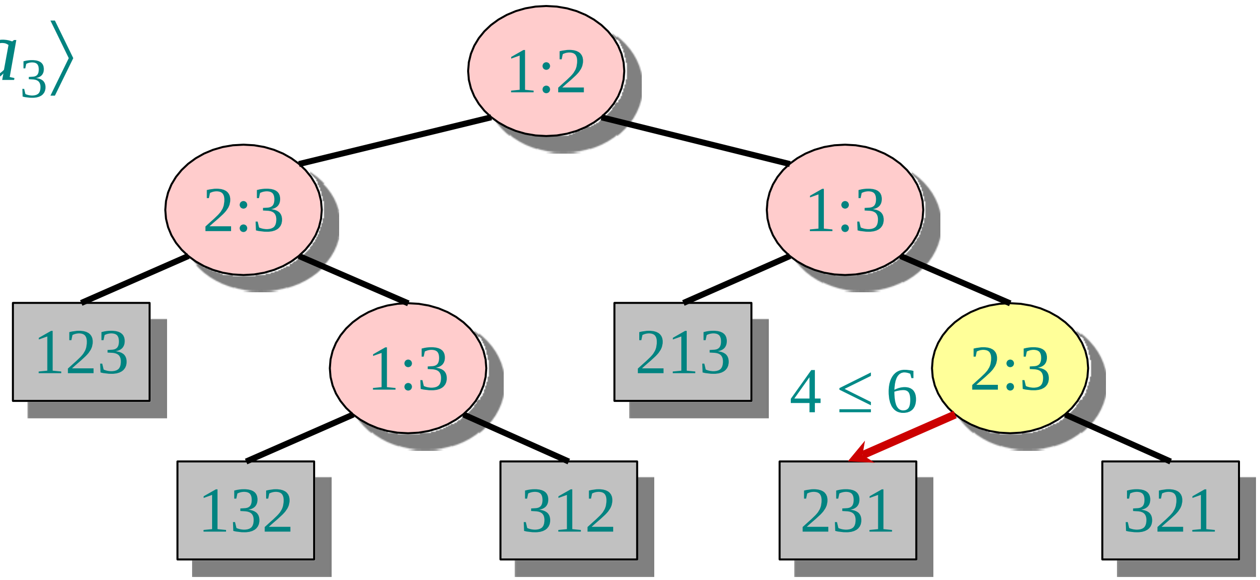
Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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



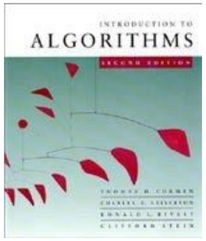
Decision-tree example

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



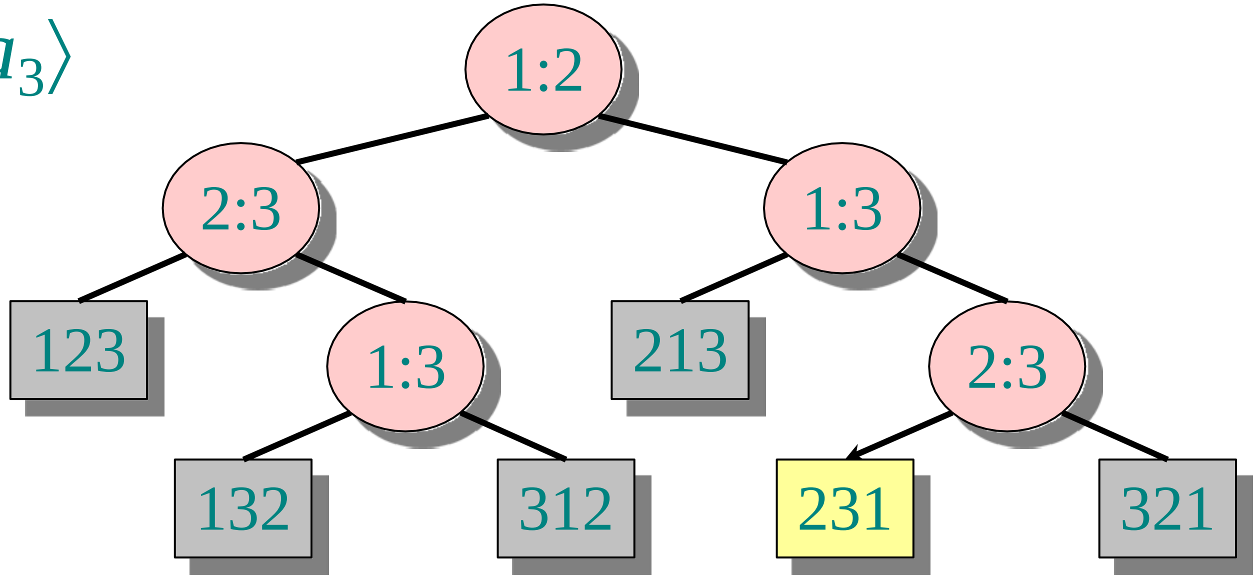
Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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



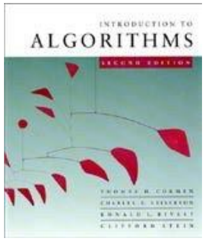
Decision-tree example

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



$4 \leq 6 \leq 9$

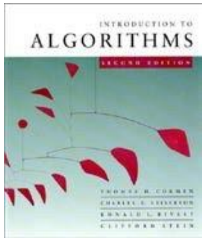
Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ has been established.



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.

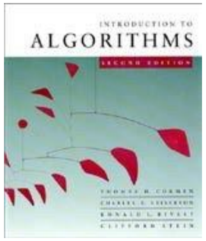


Lower bound for decision-tree sorting

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, $n! \leq 2^h$.

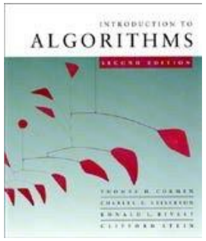
$$
\begin{aligned}
\therefore\ h &\geq \lg(n!) && (\lg \text{ is mono. increasing}) \\
&\geq \lg((n/e)^n) && (\text{Stirling's formula}) \\
&= n \lg n - n \lg e \\
&= \Omega(n \lg n).\ \square
\end{aligned}
$$



Lower bound for comparison sorting

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

LINEAR SORT

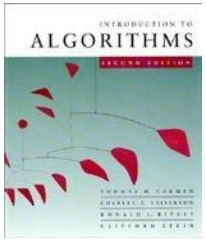


Sorting in linear time

Counting sort: No comparisons between elements.

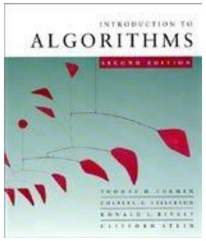
- ***Input***: $A[1\,.\,.\,n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- ***Output***: $B[1\,.\,.\,n]$, sorted.
- ***Auxiliary storage***: $C[1\,.\,.\,k]$.

Counting sort assumes that each of the n input elements is an integer in the range 0 to k,



Counting sort

for $i \leftarrow 1$ **to** k
    do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** n
    do $C[A[j]] \leftarrow C[A[j]] + 1$ $\quad \rhd\, C[i] = |\{\text{key} = i\}|$
for $i \leftarrow 2$ **to** k
    do $C[i] \leftarrow C[i] + C[i-1]$ $\quad \rhd\, C[i] = |\{\text{key} \leq i\}|$
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

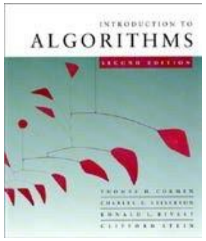
A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4

B:

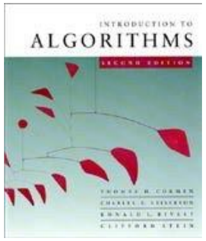


Loop 1

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

	1	2	3	4
C:	0	0	0	0

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

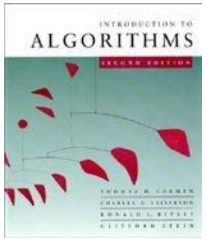
C:

1	2	3	4
0	0	0	1

B:

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

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



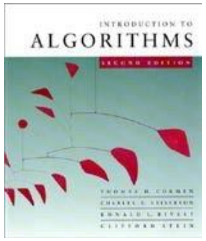
Loop 2

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

	1	2	3	4
C:	1	0	0	1

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

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

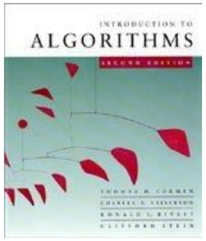
C:

1	2	3	4
1	0	1	1

B:

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

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

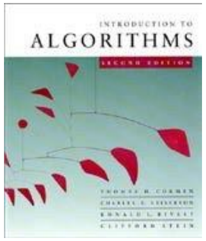
C:

1	2	3	4
1	0	1	2

B:

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

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

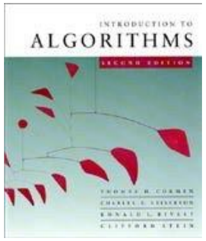
C:

1	2	3	4
1	0	2	2

B:

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

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



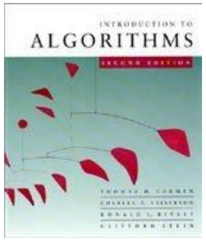
Loop 3

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

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

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

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



Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

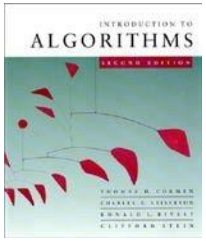
B:

C':

1	1	3	2

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

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



Loop 3

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

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

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

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

Loop 4

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	1	3	5

B:

		3		

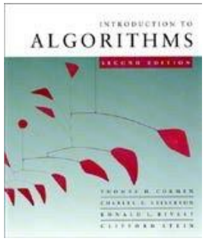
C':

1	1	2	5

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

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

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



Loop 4

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	1	2	5

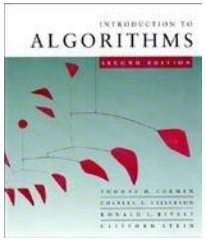
B:

		3		4

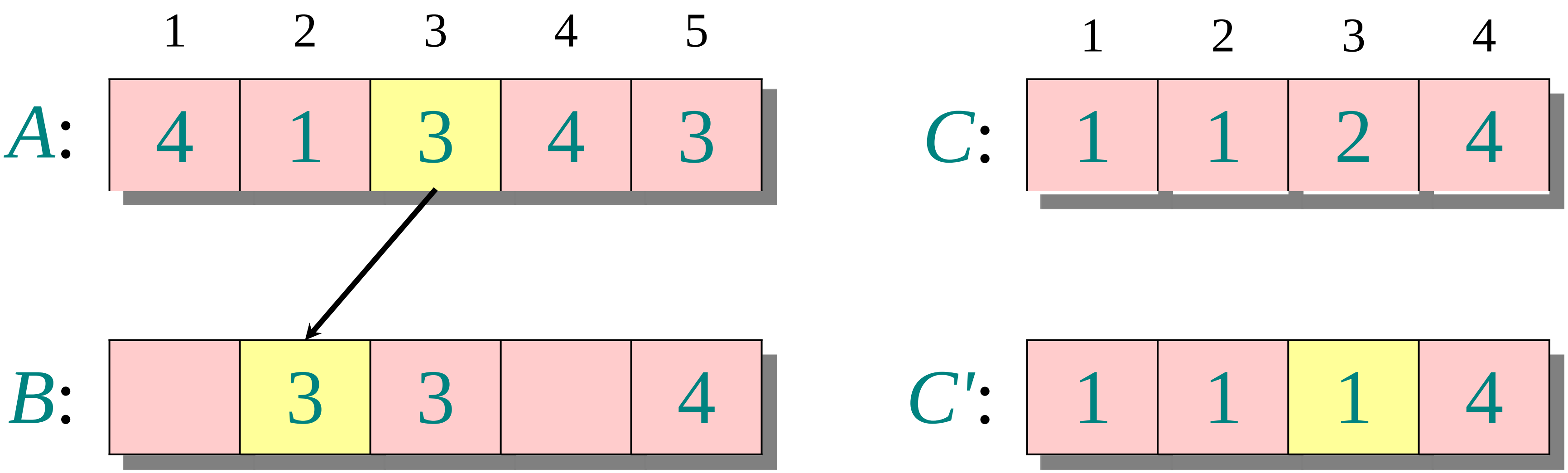
C':

1	1	2	4

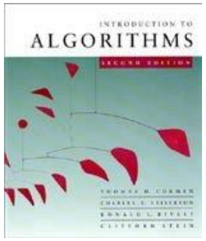
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$



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

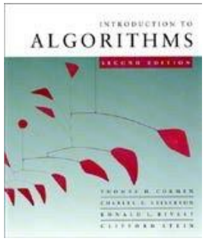
	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3		4

	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

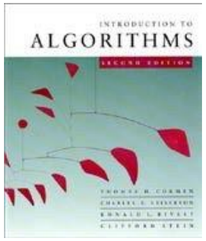
	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3	4	4

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

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

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

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



Analysis

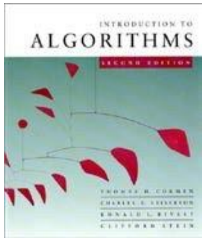
$\Theta(k)$ **for** $i \leftarrow 1$ **to** k
    do $C[i] \leftarrow 0$

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

$\Theta(n + k)$



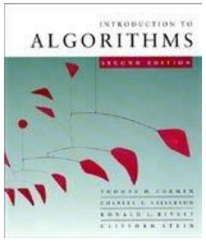
Running time

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

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

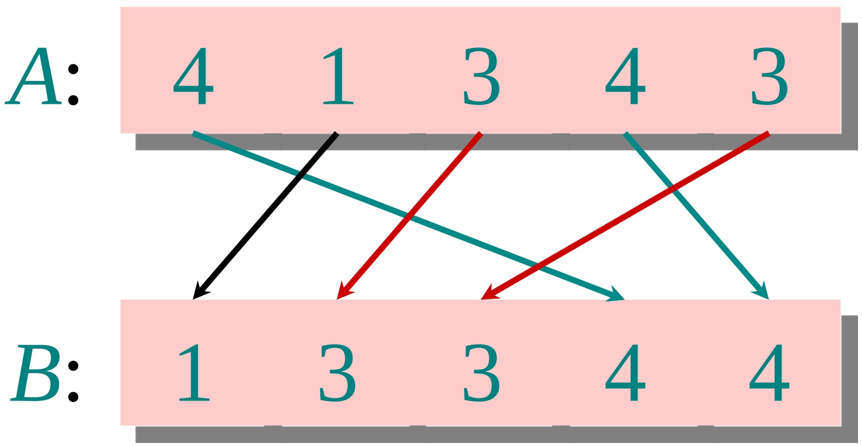
Answer:

- ***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?

Stable Sorting Algorithms

- **A sorting algorithms is *stable* if for any two indices i and j with $i < j$ and $a_i = a_j$, element a_i precedes element a_j in the output sequence.**

Input

2_1	7_1	4_1	4_2	2_2	5_1	2_3	6_1

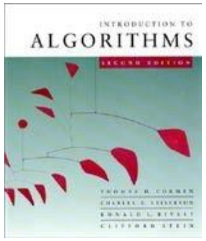
Output

2_1	2_2	2_3	4_1	4_2	5_1	6_1	7_1

Observation: *Counting Sort is stable.*

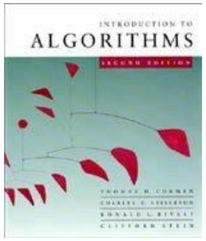
Counting Sort

- Linear Sort! Cool! *Why don't we always use counting sort?*
- Because it depends on range *k* of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, *k* too large (2^{32} = 4,294,967,296)



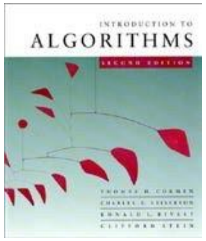
Radix sort

- ***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

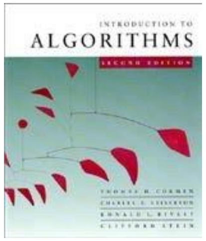


Correctness of radix sort

Induction on digit position

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

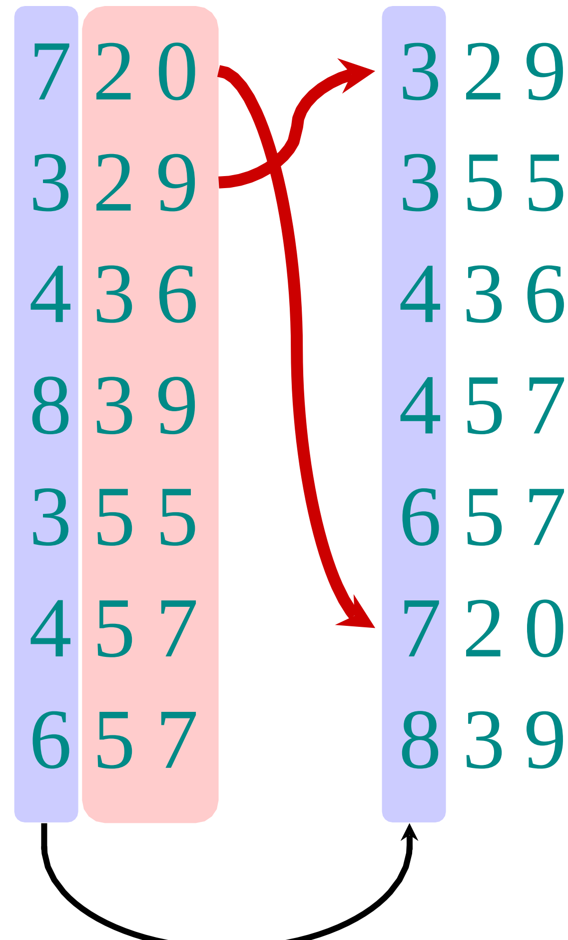
7 2 0	3 2 9
3 2 9	3 5 5
4 3 6	4 3 6
8 3 9	4 5 7
3 5 5	6 5 7
4 5 7	7 2 0
6 5 7	8 3 9

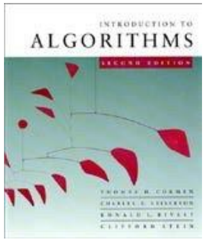


Correctness of radix sort

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.

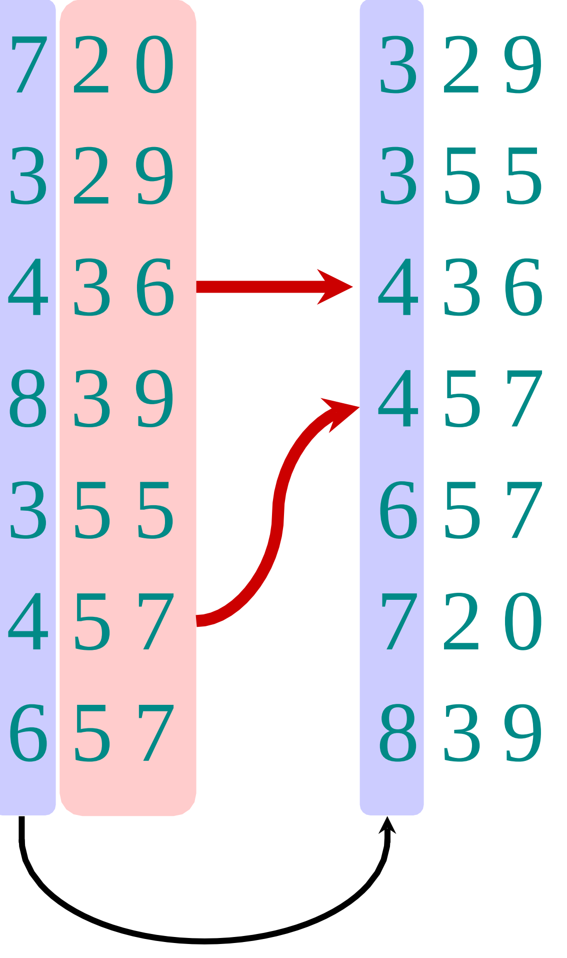




Correctness of radix sort

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.



Radix Sort
Correctness and Running Time

- What is the running time of radix sort?

When k is not too large, use counting sort as a stable sort

counting sort

- Each pass over the d digits takes time **O(n+k),**

- **Running Time=** Running time of stable Sort x d
    - $=\Theta$ d.$(n+k)$
    - K and d constants
    -

    T(n)= Θ (n)
- Stable, Fast
- Doesn't sort in place (because counting sort is used)

Bucket Sort

- **Assumption: input - *n* real numbers from [0, 1)**
- **Basic idea:**
  - Create *n* linked lists (*buckets*) to divide interval [0,1) into subintervals of size 1/*n*
  - Add each input element to appropriate bucket and sort buckets with insertion sort
- **Uniform input distribution → O(1) bucket size**
  - **Therefore the expected total time is O(n)**

Bucket Sort

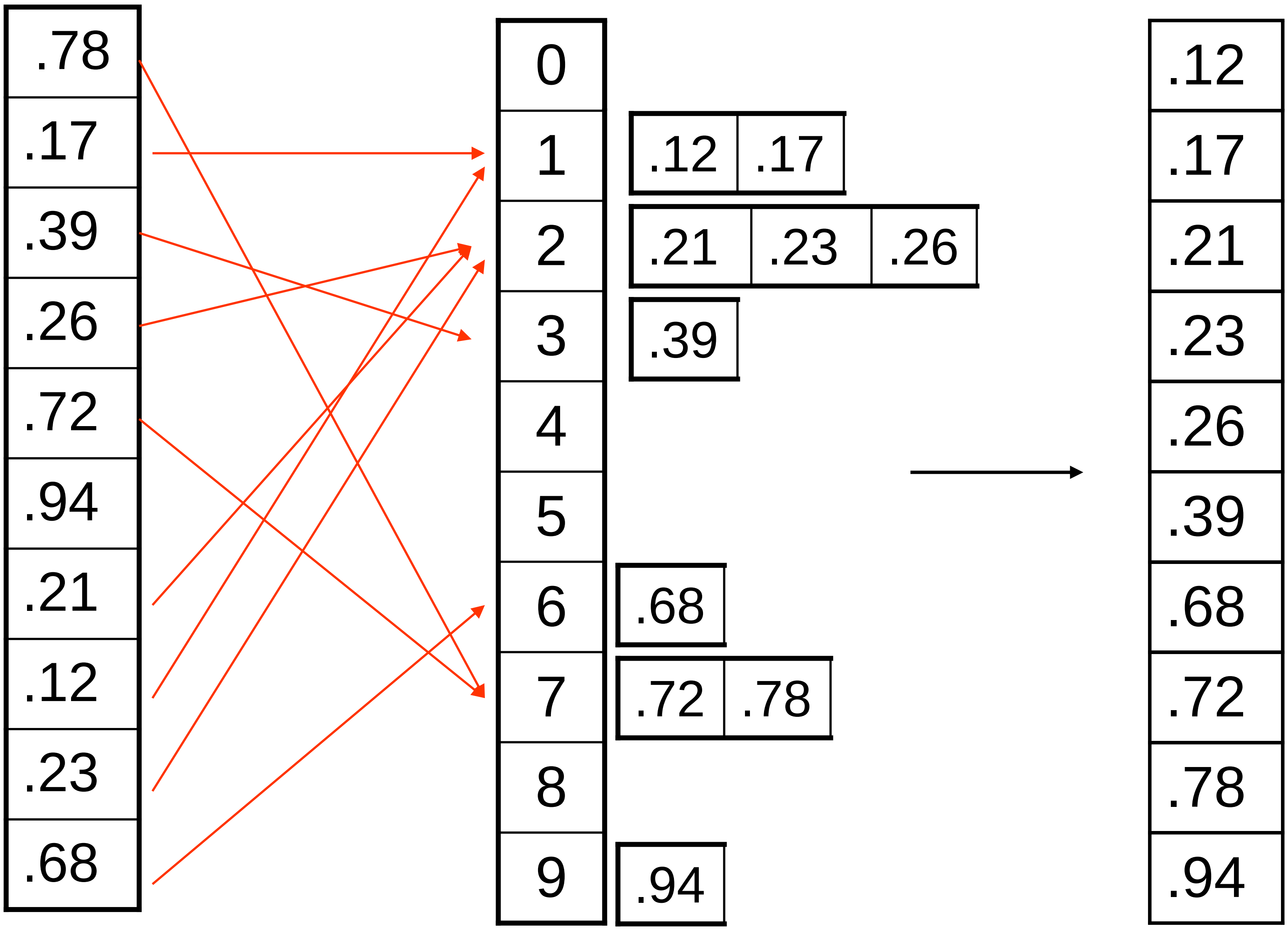
<u>**Bucket-Sort**</u>**(A)**

1. **$n \leftarrow$ length(A)**
2. **for $i \leftarrow$ 0 to n**

   Distribute elements over buckets
3. **do insert $A[i]$ into list B[floor(n*$A[i]$)]**
4. **for $i \leftarrow$ 0 to n –1**

   Sort each bucket
5. **do Insertion-Sort($B[i]$)**
6. **Concatenate lists $B[0]$, $B[1]$, … $B[n-1]$ in order**

Bucket Sort Example



Bucket Sort – Running Time

- All lines except line 5 (Insertion-Sort) take $O(n)$ in the worst case.

- In the worst case, $O(n)$ numbers will end up in the same bucket, so in the worst case, it will take $O(n^2)$ time.

- Lemma: *Given that the input sequence is drawn uniformly at random from* [0,1)*, the expected size of a bucket is* $O(1)$*.*

- So, in the *average case*, only a constant number of elements will fall in each bucket, so it will take $O(n)$ (see proof in book).

- Use a different indexing scheme (hashing) to distribute the numbers uniformly**.**

Sorting Algorithms

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Reference

- **Chapter# 8**
  - **Sorting in Linear Time**
- **Introduction to Algorithms by Cormen**