CS473-Algorithms I

Lecture 9

Sorting in Linear Time

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.

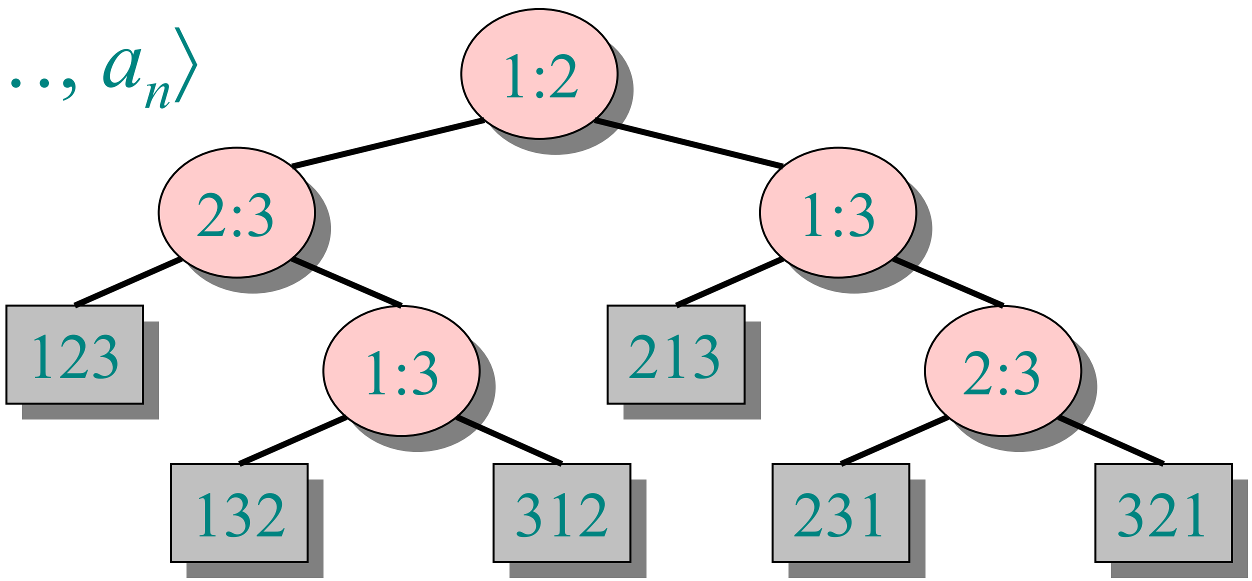
The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

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

Decision trees can help us answer this question.

Decision-tree example

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

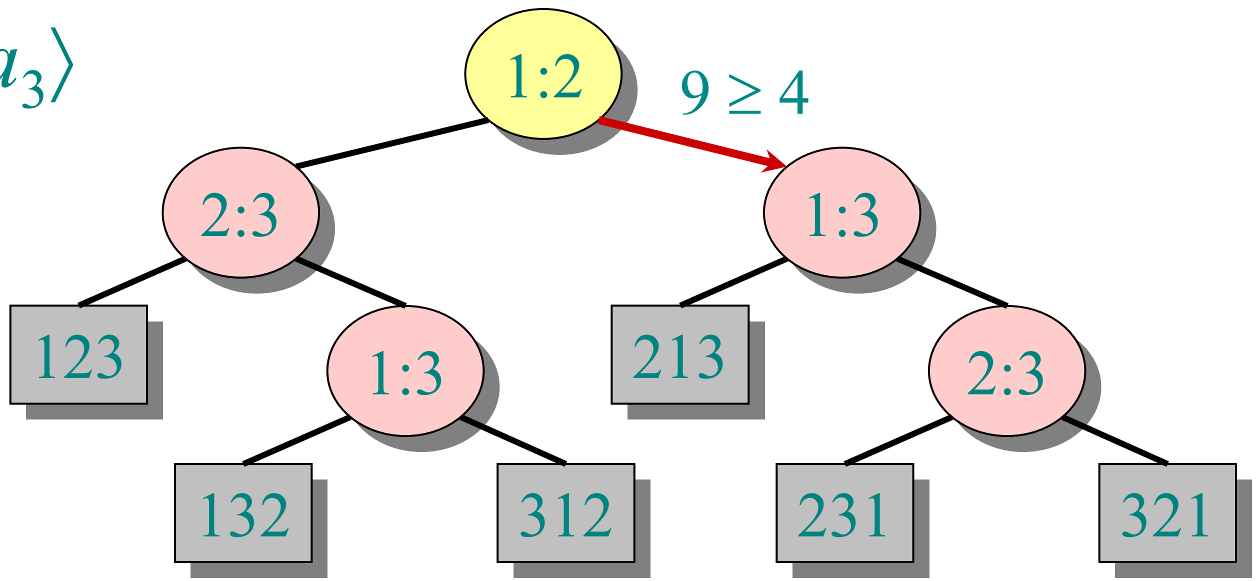


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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

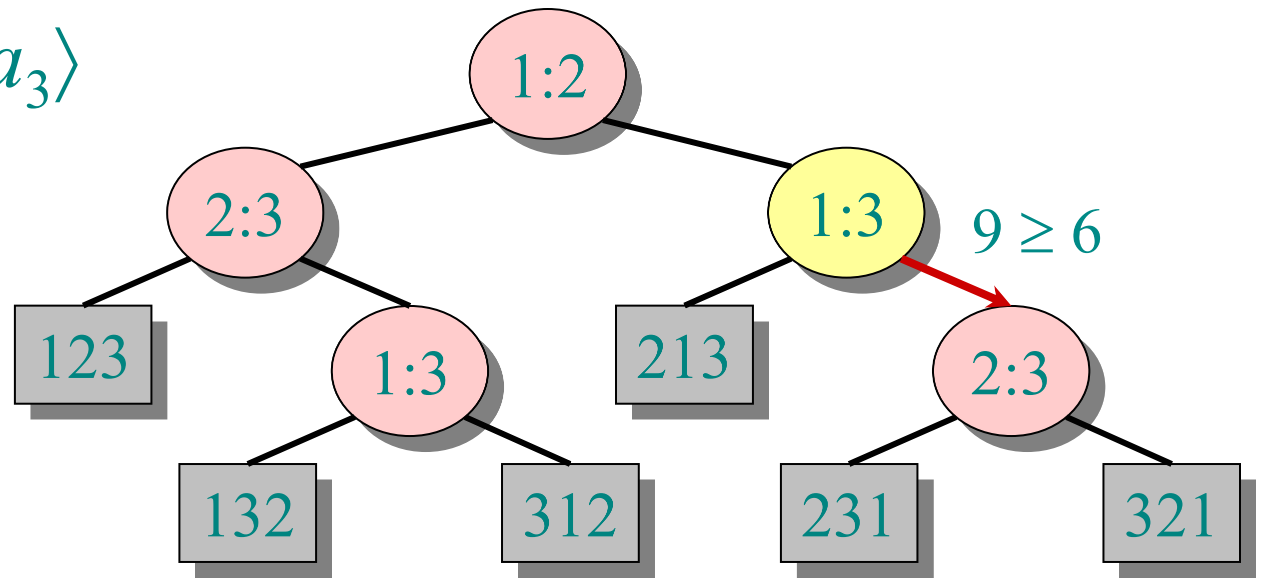


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2,\ldots, n\}$.

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

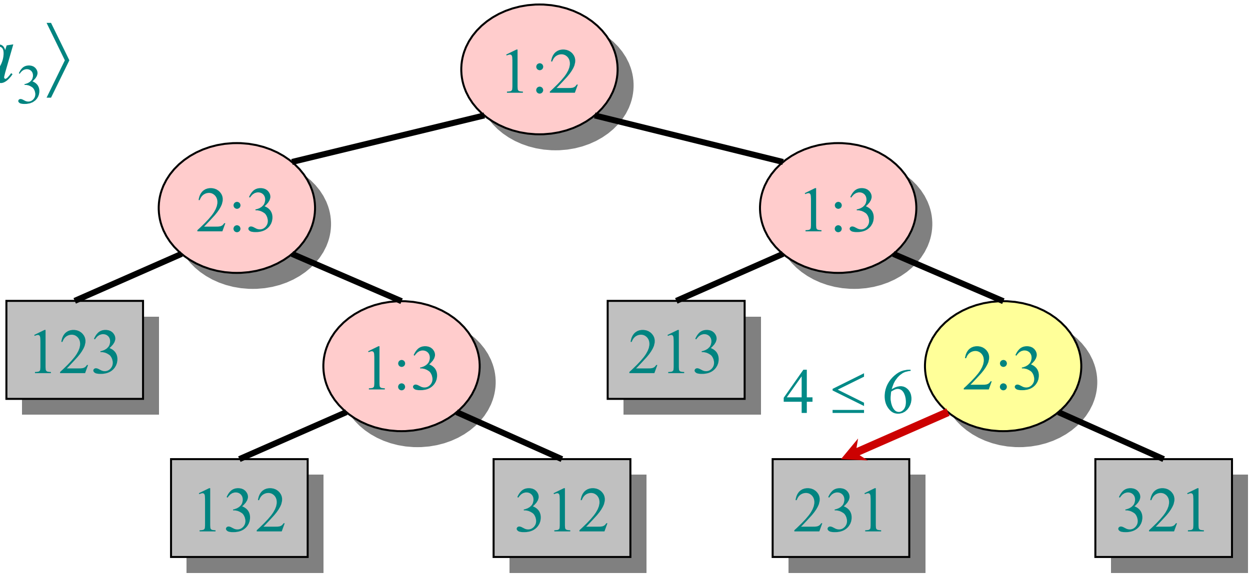


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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

Decision-tree example

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

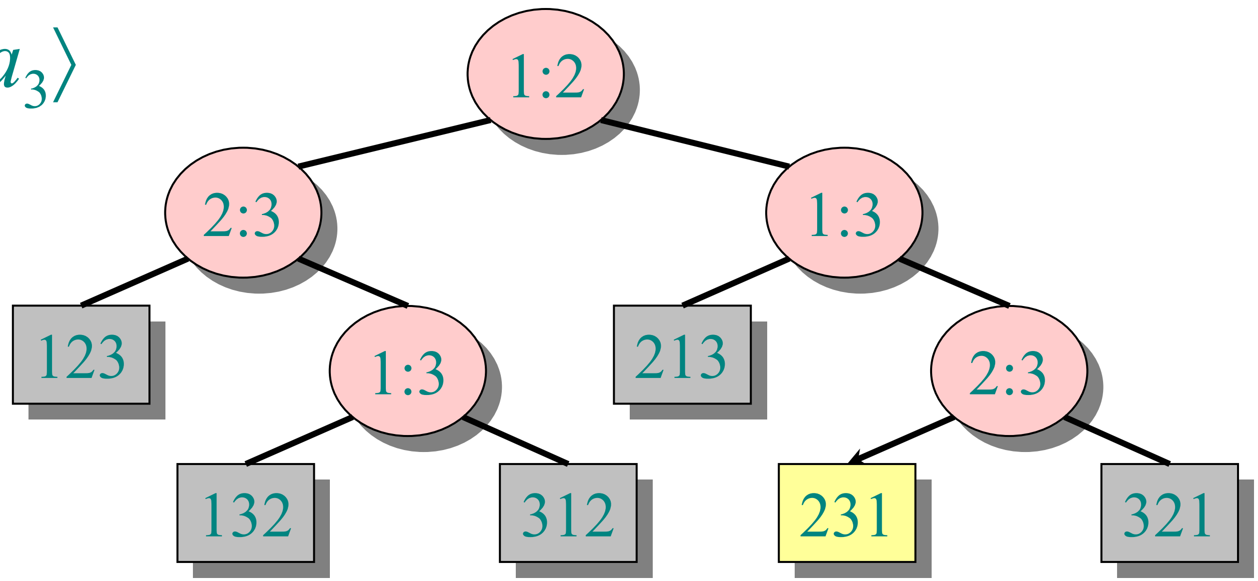


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2,\ldots, n\}$.

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



$4 \le 6 \le 9$

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

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

Lower bound for comparison sorting

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

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

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

    do $C[i] \leftarrow 0$

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

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

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

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

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

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

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

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

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

	1	2	3	4
C:	0	0	0	1

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	0	1

B:

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

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

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

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	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 $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright\ C[i] = |\{\text{key} = i\}|$

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

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	2

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ $\triangleright$ $C[i] = |\{\text{key} \leq i\}|$

Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

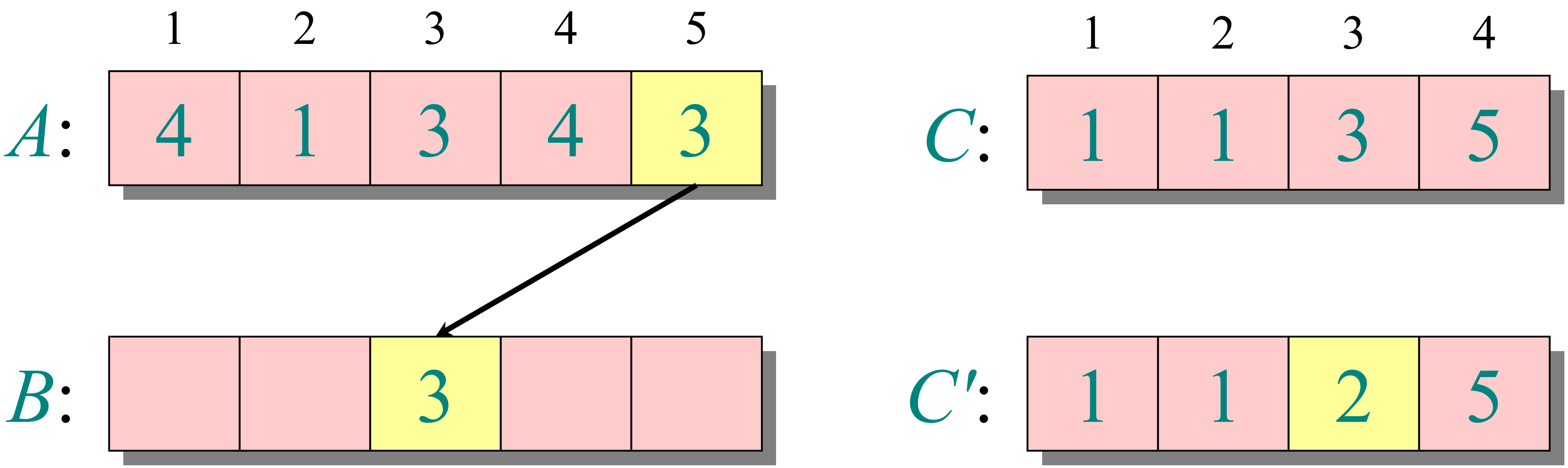
C′:

1	1	3	5

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

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

Loop 4

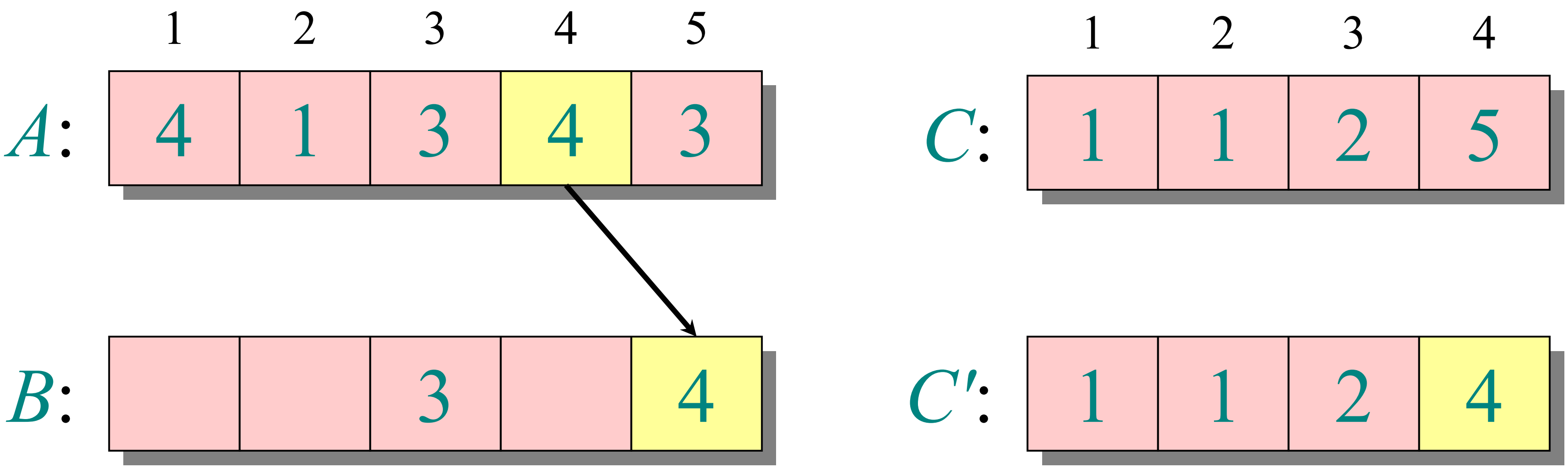


for $j \leftarrow n$ **downto** 1

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

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

Loop 4

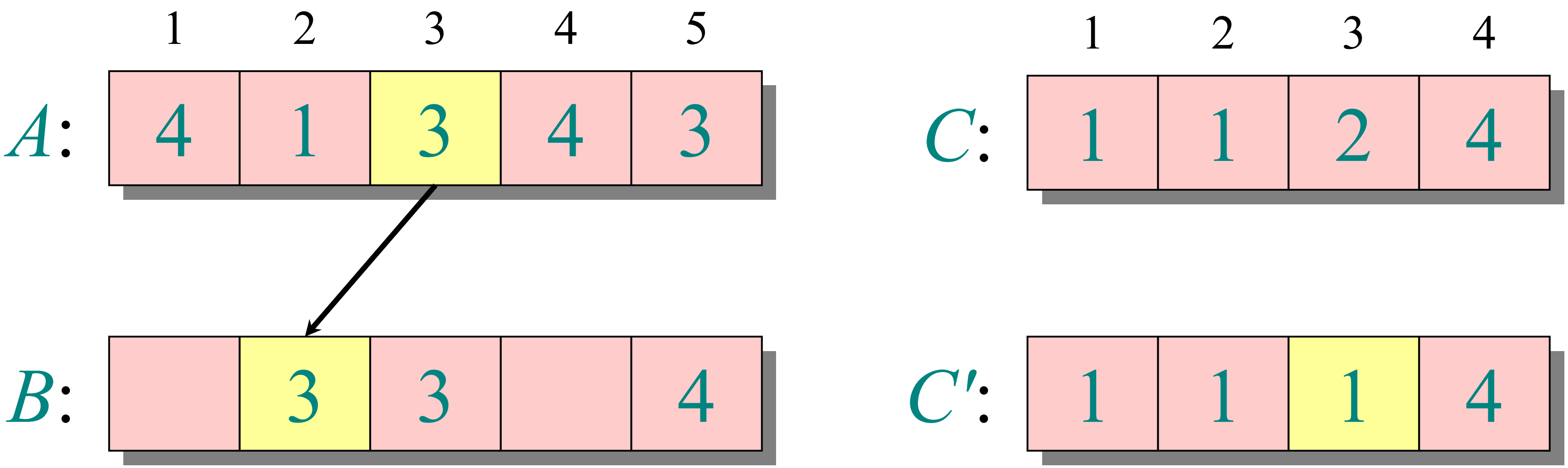


for $j \leftarrow n$ **downto** 1

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

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

Loop 4

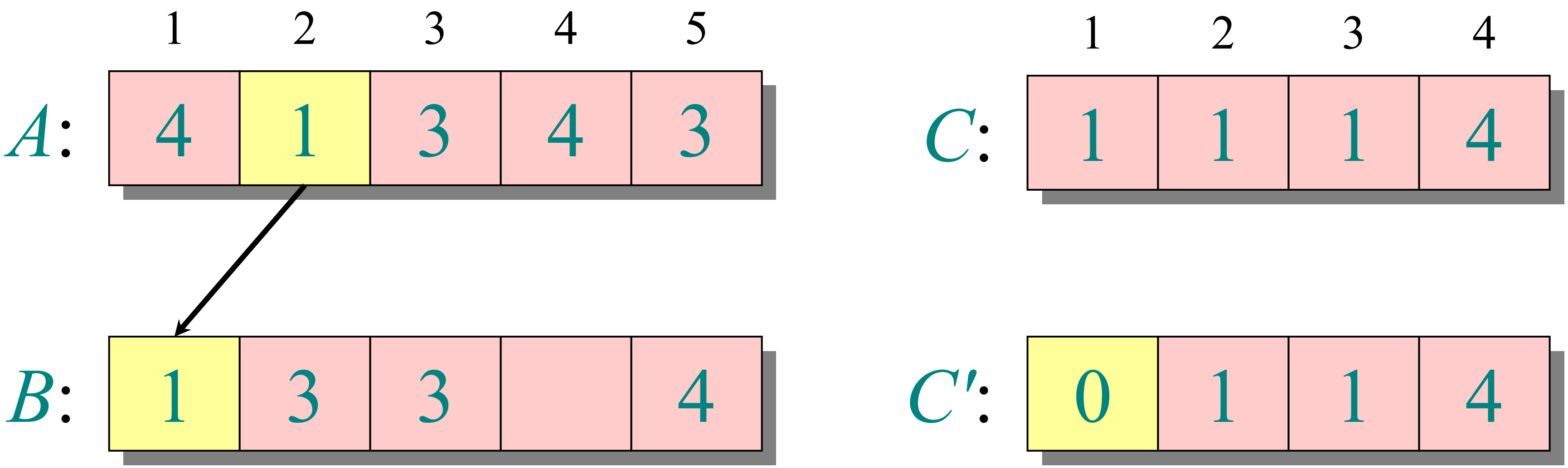


for $j \leftarrow n$ **downto** 1

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

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

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

	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

$$
\begin{array}{l}
\textbf{for } j \leftarrow n \textbf{ downto } 1 \\
\quad \textbf{do } B[C[A[j]]] \leftarrow \mathrm{A}[j] \\
\qquad C[A[j]] \leftarrow C[A[j]] - 1
\end{array}
$$

Analysis

$$\Theta(k) \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow 0 \end{array} \right.$$

$$\Theta(n) \left\{ \begin{array}{l} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \qquad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \end{array} \right.$$

$$\Theta(k) \left\{ \begin{array}{l} \textbf{for } i \leftarrow 2 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow C[i] + C[i-1] \end{array} \right.$$

$$\Theta(n) \left\{ \begin{array}{l} \textbf{for } j \leftarrow n \textbf{ downto } 1 \\ \qquad \textbf{do } B[C[A[j]]] \leftarrow \mathrm{A}[j] \\ \qquad\qquad C[A[j]] \leftarrow C[A[j]] - 1 \end{array} \right.$$

$$\Theta(n + k)$$

Running time

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

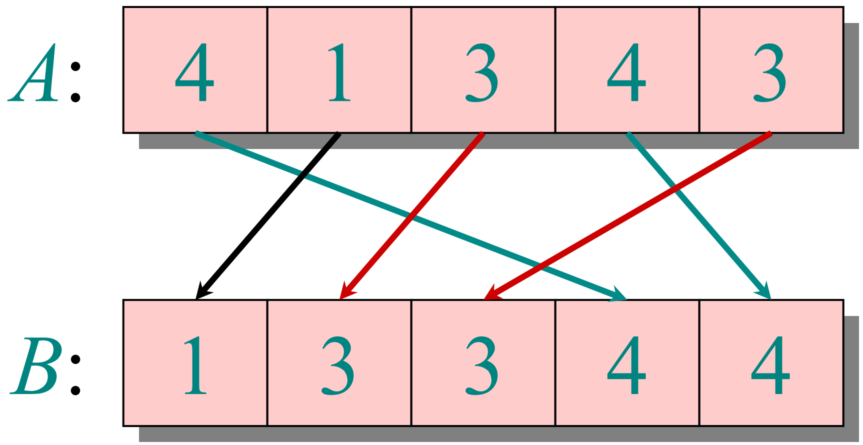
- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

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?

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.

Herman Hollerith (1860-1929)

- The 1880 U.S. Census took almost 10 years to process.
- While a lecturer at MIT, Hollerith prototyped punched-card technology.
- His machines, including a “card sorter,” allowed the 1890 census total to be reported in 6 weeks.
- He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International Business Machines.

“Modern” IBM card

- One character per column.

Produced by the WWW Virtual Punch-Card Server.

So, that’s why text windows have 80 columns!

Origin of radix sort

Hollerith's original 1889 patent alludes to a most-significant-digit-first radix sort:

"The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards."

Least-significant-digit-first radix sort seems to be a folk invention originated by machine operators.

Hollerith's MSD-First Radix Sort

- Sort numbers on most-significant-digit (MSD)
  - sort each of the resulting bins recursively
  - then, combine the decks in order
- Cards in 9 out of 10 bins must be put aside to sort each bin
  - may require very large number of sorting passes
  - may generate very large number of intermediate card piles to maintain

Hollerith's MSD-First Radix Sort

$S(d)$: # of sorting passes needed to sort d-digit numbers (worst-case)

Recurrence: $S(d) = 10S(d-1) + 1$ with $S(1) = 1$

$$S(d) = 10S(d-1)+1 = 10(10S(d-2)+1)+1 = 10^2 S(d-2)+10^1+10^0$$

$$= 10^i S(d-i)+10^{i-1}+10^{i-2}+\ldots+10^1+10^0$$

Iteration terminates when $i=d-1$ with $S(d-(d-1))=S(1)=1$

$$S(d) = \sum_{i=0}^{d-1} 10^i = \frac{10^d - 1}{10 - 1} = \frac{1}{9}(10^d - 1) \Rightarrow \boxed{S(d) = \frac{1}{9}(10^d - 1)}$$

Hollerith's MSD-First Radix Sort

$P(d)$: # of intermediate card piles maintained (worst-case)

Each sorting pass generates 9 intermediate piles except the sorting passes on LSDs (there are 10^{d-1} such sortings)

$$P(d) = 9(S(d) - 10^{d-1}) = 9\left(\frac{10^d - 1}{9} - 10^{d-1}\right)$$

$$= 9 \times \frac{1}{9}\,(10^{d-1} - 1 - 9 \times 10^{d-1})$$

$$\boxed{P(d) = 10^{d-1} - 1}$$

Alternative solution by solving the reccurrence:

$$\boxed{\begin{aligned} P(d) &= 10P(d-1) + 9 \\ P(1) &= 0 \end{aligned}}$$

LSD-First Radix Sort

Radix Sort: Counter-intuitive solution

- Sort numbers on their least significant digit (LSD) first
- Combine the cards into a single deck with
  - the cards in the 0-bin preceeding
  - the cards in the 1-bin preceeding
  - the cards in the 2-bin, and so on.
- Continue this sorting process for the other digits
  - from the LSD towards the MSD

➢ Requires only *d* sorting passes

➢ Does not generate any intermediate card piles

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

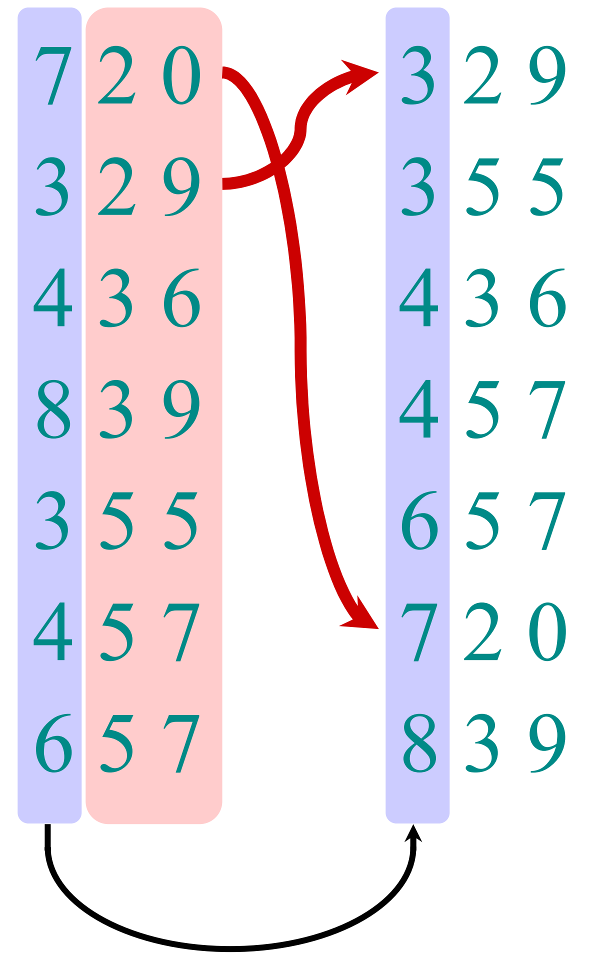
- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t

720	329
329	355
436	436
839	457
355	657
457	720
657	839

Correctness of radix sort

Induction on digit position

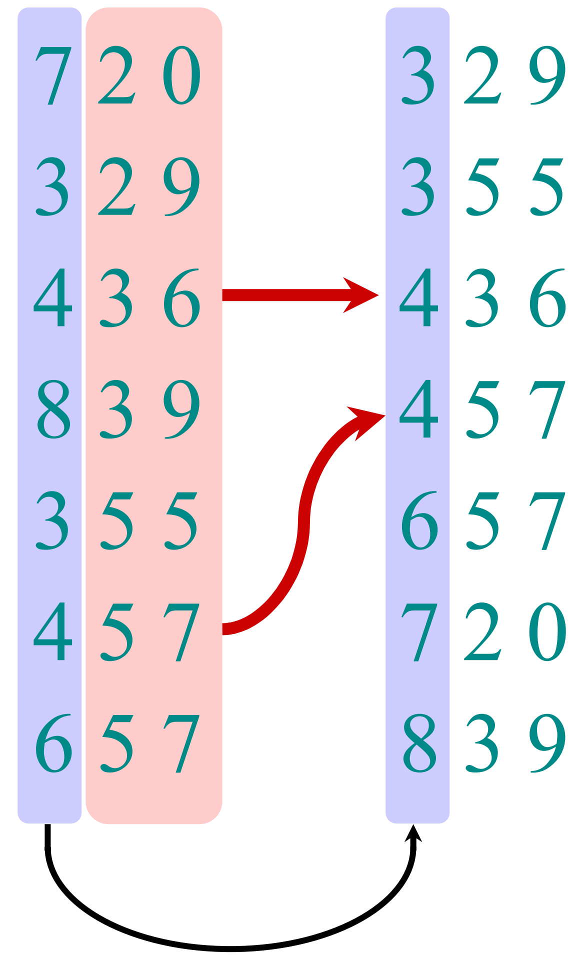
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.



Correctness of radix sort

Induction on digit position

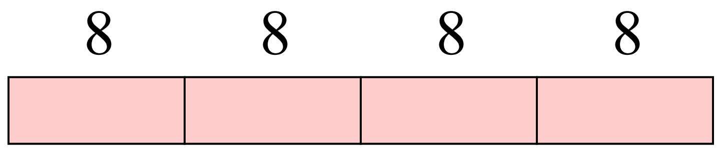
- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.



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 $\lceil \lg 2000 \rceil = 11$ 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.