Design and Analysis of Algorithms

CSE 5311

Lecture 8 Sorting in Linear Time

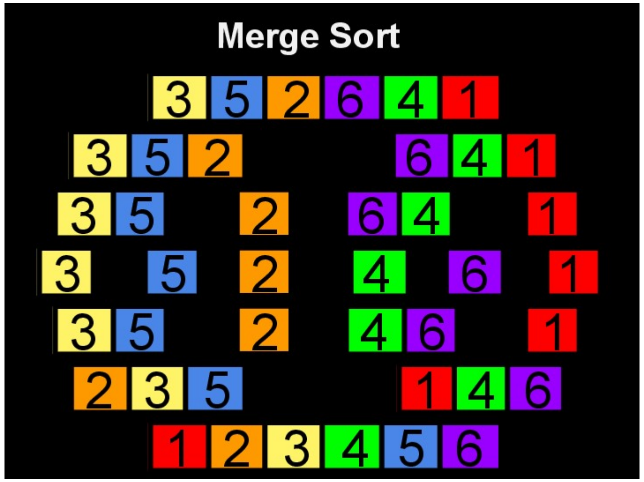
Junzhou Huang, Ph.D.

Department of Computer Science and Engineering

Sorting So Far

- **Insertion sort:**
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

- **Merge sort:**
  - Divide-and-conquer:
    - ➢ Split array in half
    - ➢ Recursively sort subarrays
    - ➢ Linear-time merge step
  - O(n lg n) worst case



Sorting So Far

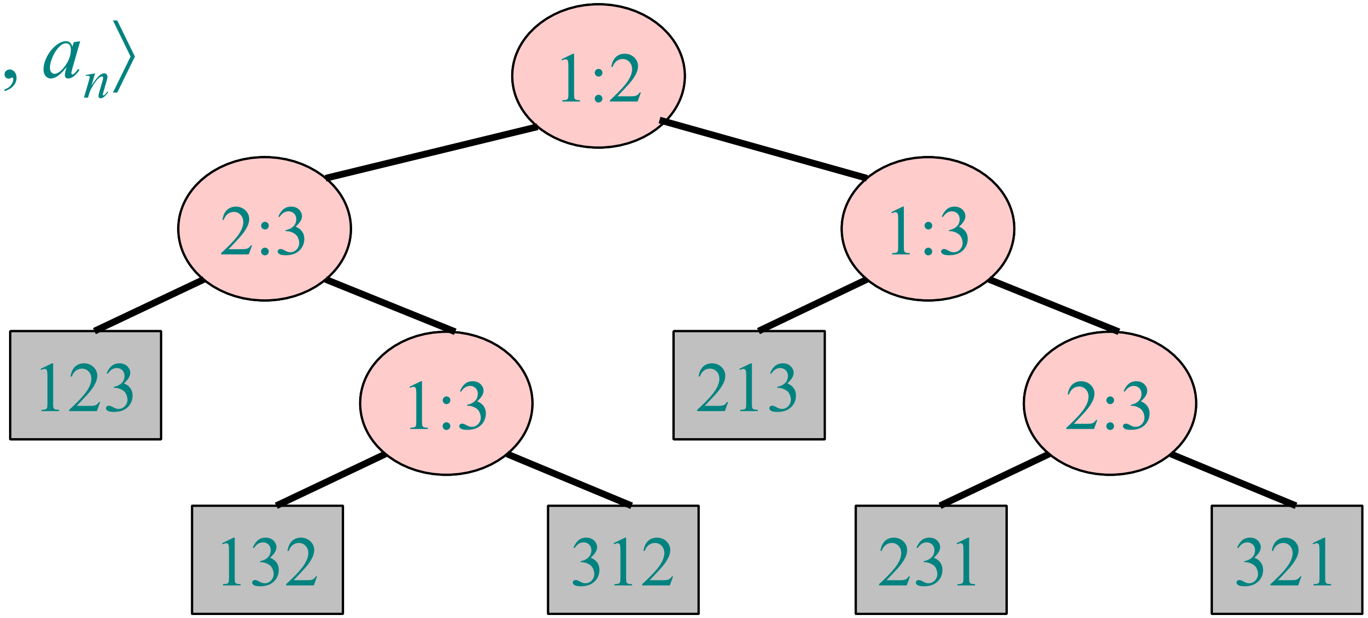
- **Heap sort:**
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around
- **Quick sort:**
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - O(n lg n) average case
  - Fast in practice
  - O(n^2) worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort

How Fast Can We Sort?

- **Lower bound**
  - Prove a Lower Bound for ***any comparison based algorithm*** for the Sorting Problem
  - How? Decision trees help us.
- **Observation:** sorting algorithms so far are ***comparison sorts***
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are Ω(n lg n)
    - ➢ A comparison sort must do O(n) comparisons (***why?***)
    - ➢ What about the gap between O(n) and O(n lg n)

Decision-tree Example

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

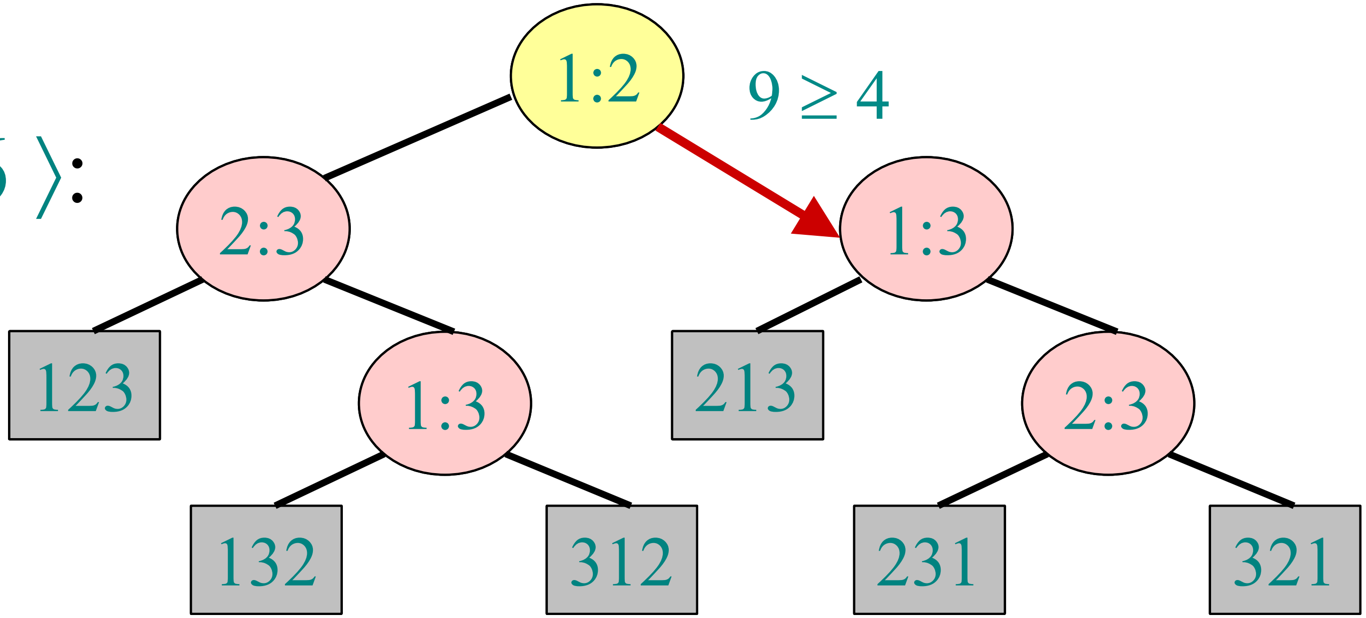


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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

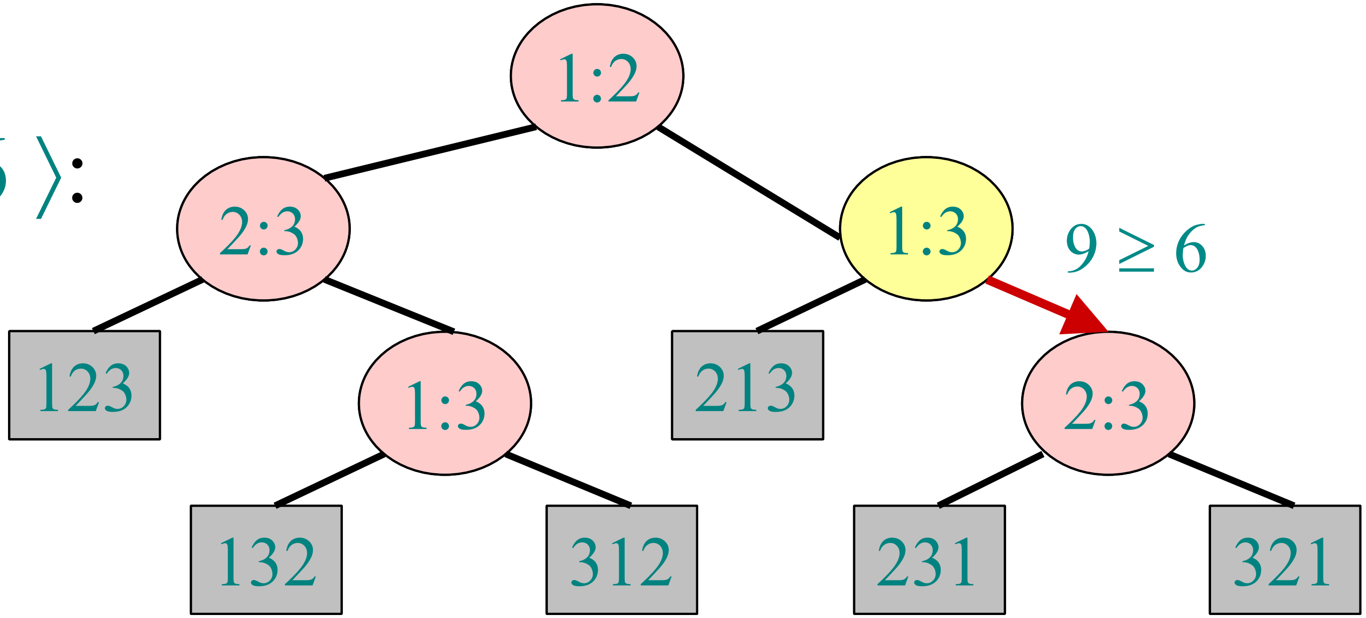


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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

Decision-tree Example

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

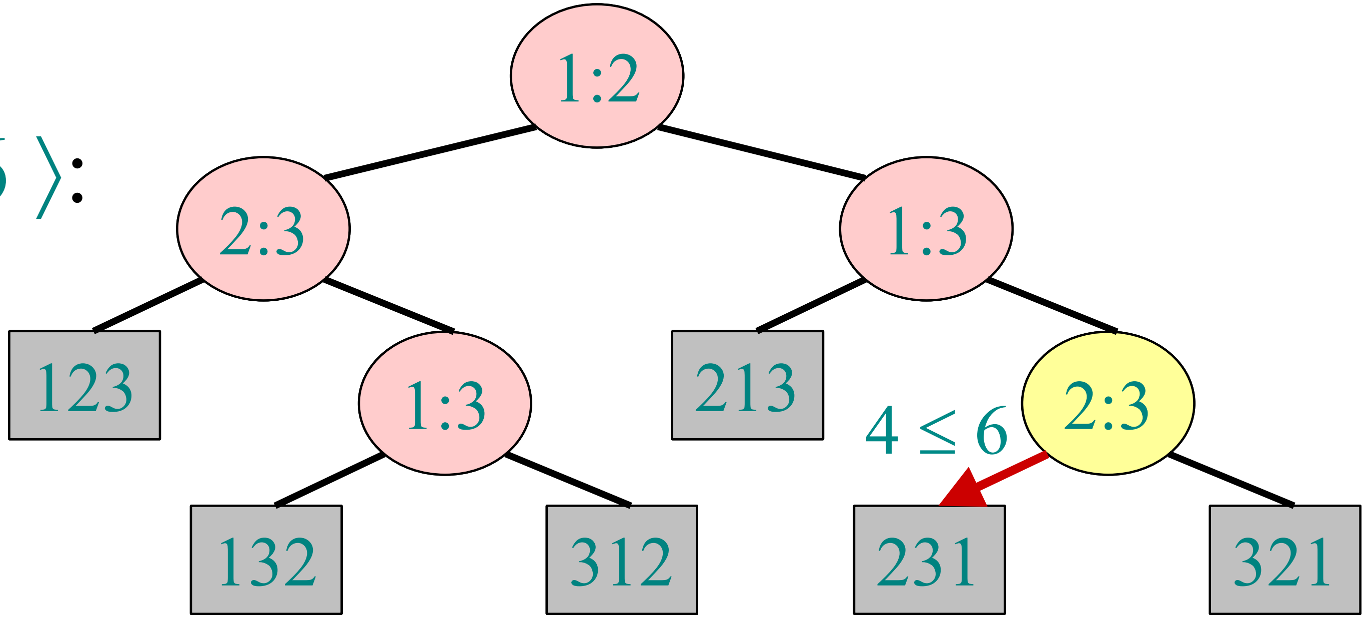


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

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

Decision-tree Example

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

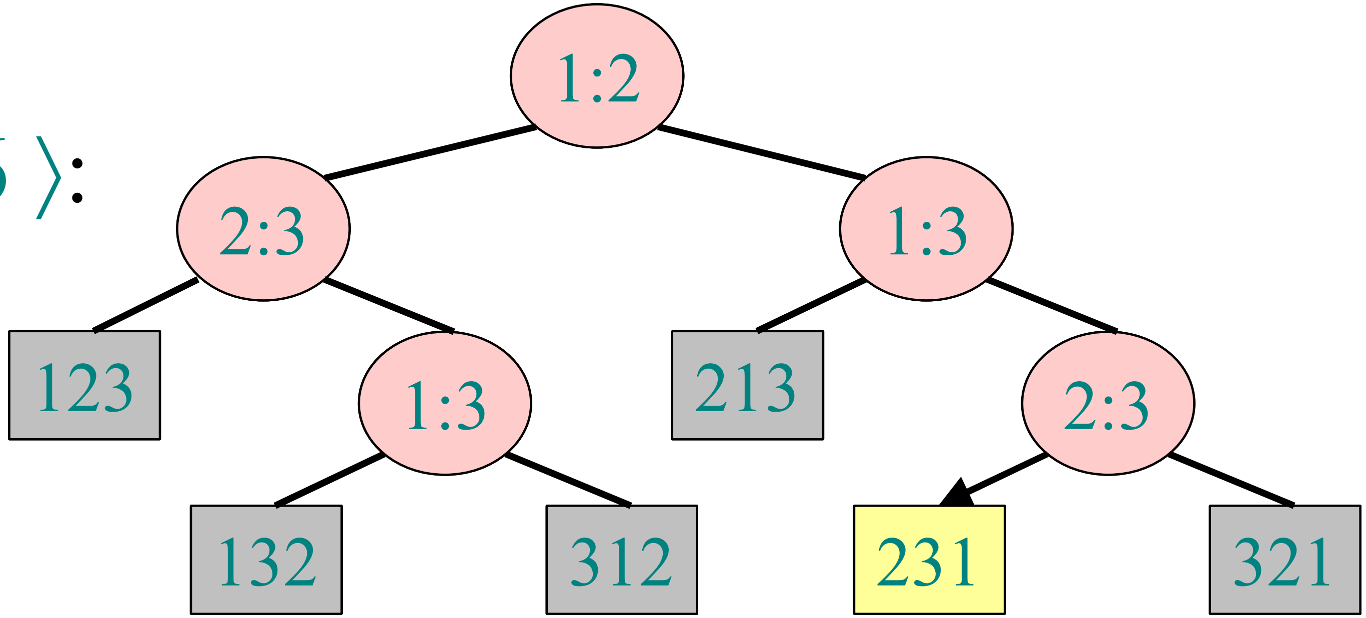


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- 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 Example

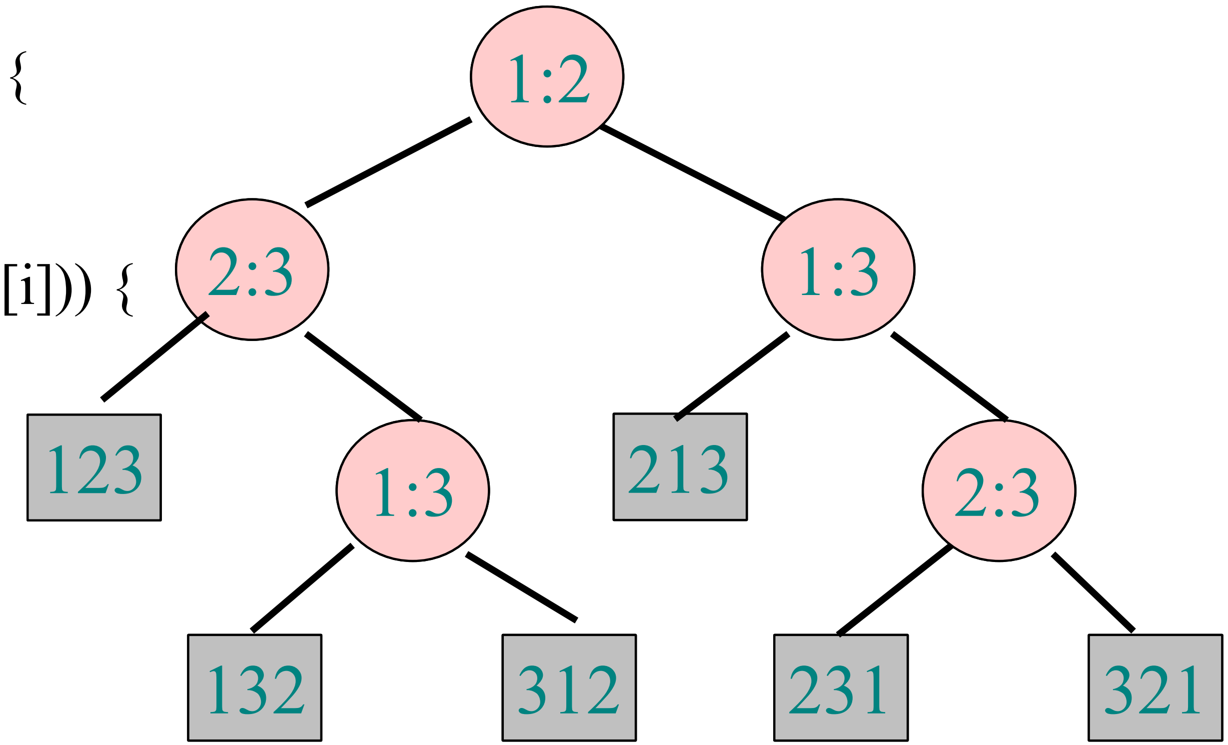
A decision tree can model the execution of any comparison sort:

- 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.

How?

Any comparison sort can be turned into a Decision tree

```
class InsertionSortAlgorithm  {
    for (int i = 1; i < a.length; i++) {
        int j = i;
        while ((j > 0) && (a[j-1] > a[i])) {
                a[j] = a[j-1];
                j--;  }
        a[j] = B;  }}
```



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, $n! \leq 2^h$.

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

n log n − n < log(n!) < n log n

Decision Tree

- ***Decision trees*** **provide an abstraction of comparison sorts**
  - A decision tree represents the comparisons made by a comparison sort. Every thing else ignored
  - What do the leaves represent?
  - How many leaves must there be?
- **Decision trees can model comparison sorts. For a given algorithm:**
  - One tree for each n
  - Tree paths are all possible execution traces
  - *What's the longest path in a decision tree for insertion sort? For merge sort?*

- ***What is the asymptotic height of any decision tree for sorting n elements?***
- **Answer: $\Omega(n \lg n)$ (now let's prove it…)**

Lower Bound For Comparison Sorting

- **Theorem:** Any decision tree that sorts n elements has height $\Omega(n \lg n)$
- What's the minimum # of leaves?
- What's the maximum # of leaves of a binary tree of height h?
- Clearly the minimum # of leaves is less than or equal to the maximum # of leaves
- So we have $n! \leq 2^h$; Taking logarithms: $\lg(n!) \leq h$
- Stirling's approximation tells us: $n! > \left(\frac{n}{e}\right)^n$
- Thus $h \geq \lg\left(\frac{n}{e}\right)^n = n \lg n - n \lg e = \Omega(n \lg n)$

The minimum height of a decision tree is $\Omega(n \lg n)$

Lower Bound For Comparison Sorting

- Thus the time to comparison sort n elements is $\Omega(n \lg n)$
- **Corollary**: Heapsort and Mergesort are asymptotically optimal comparison sorts
- But the name of this lecture is "**Sorting in linear time**"!
  - ***How can we do better than $\Omega(n \lg n)$?***

Sorting In Linear Time

- **Counting sort**
  - No comparisons between elements!
  - ***But***…depends on assumption about the numbers being sorted
    - We assume numbers are in the range *1*… k
  - **The algorithm:**
    - **Input**: A[1..n], where A[j] $\in$ {1, 2, 3, …, k}
    - **Output**: B[1..n], sorted (notice: not sorting in place)
    - **Also:** Array C[1..k] for auxiliary storage

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Work through example: A={4 1 3 4 3}, k = 4

Counting Sort

```
CountingSort(A, B, k)
      for i=1 to k
            C[i]= 0;
      for j=1 to n
            C[A[j]] += 1;
      for i=2 to k
            C[i] = C[i] + C[i-1];
      for j=n downto 1
            B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What will be the running time?

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

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

	1	2	3	4
C:	1	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

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

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

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

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

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

Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

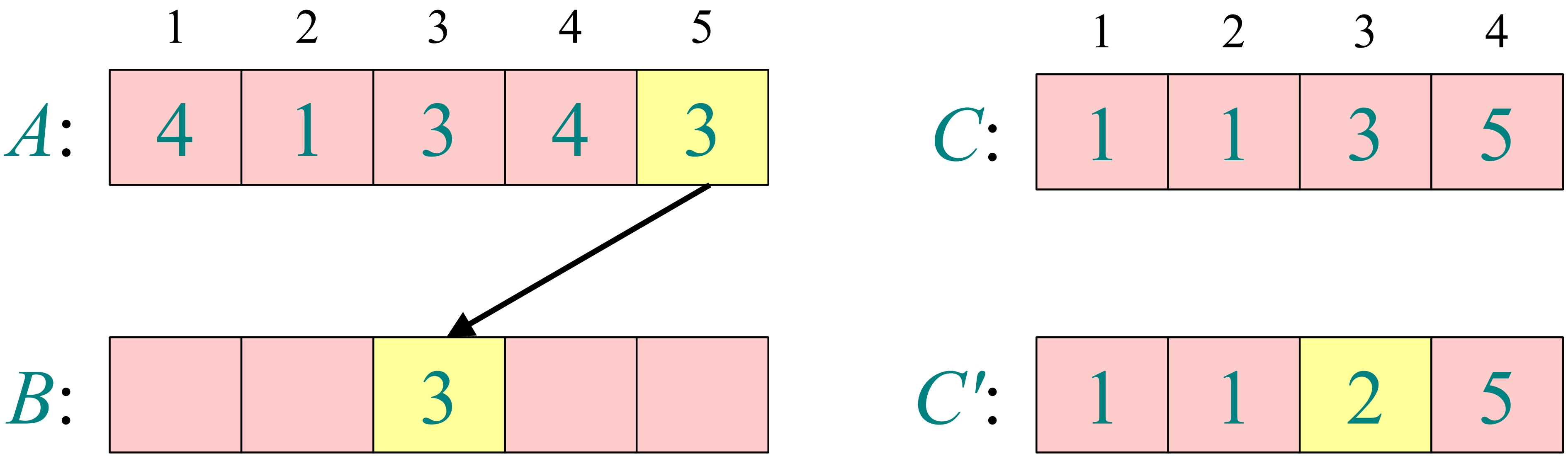
C':

1	1	3	5

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

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

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

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

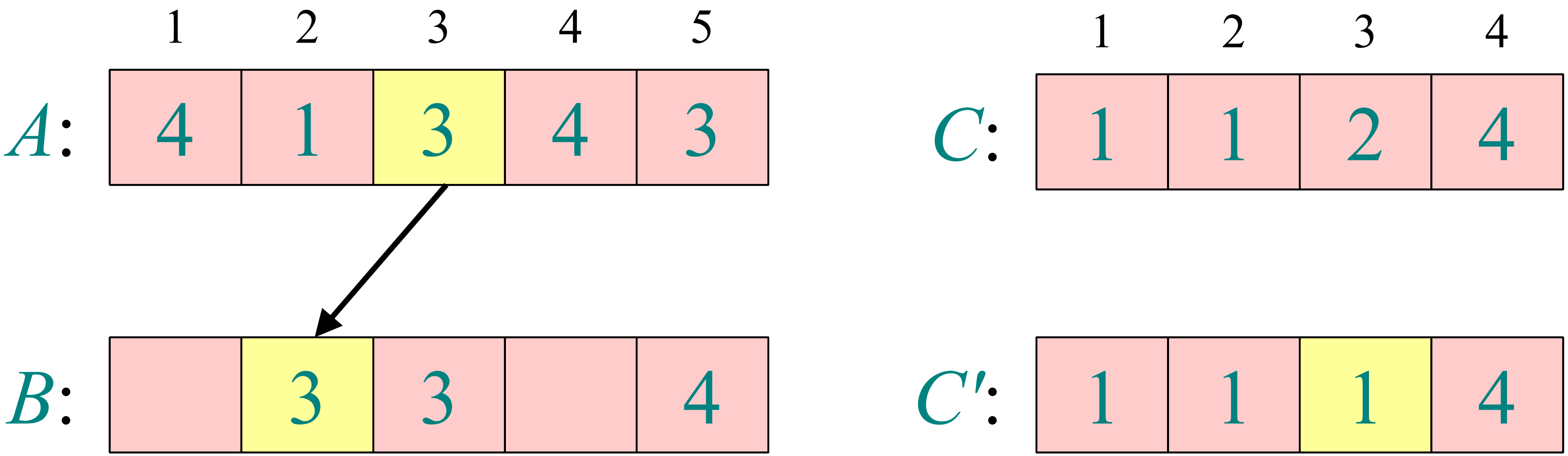
	1	2	3	4
C:	1	1	2	5
C':	1	1	2	4

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

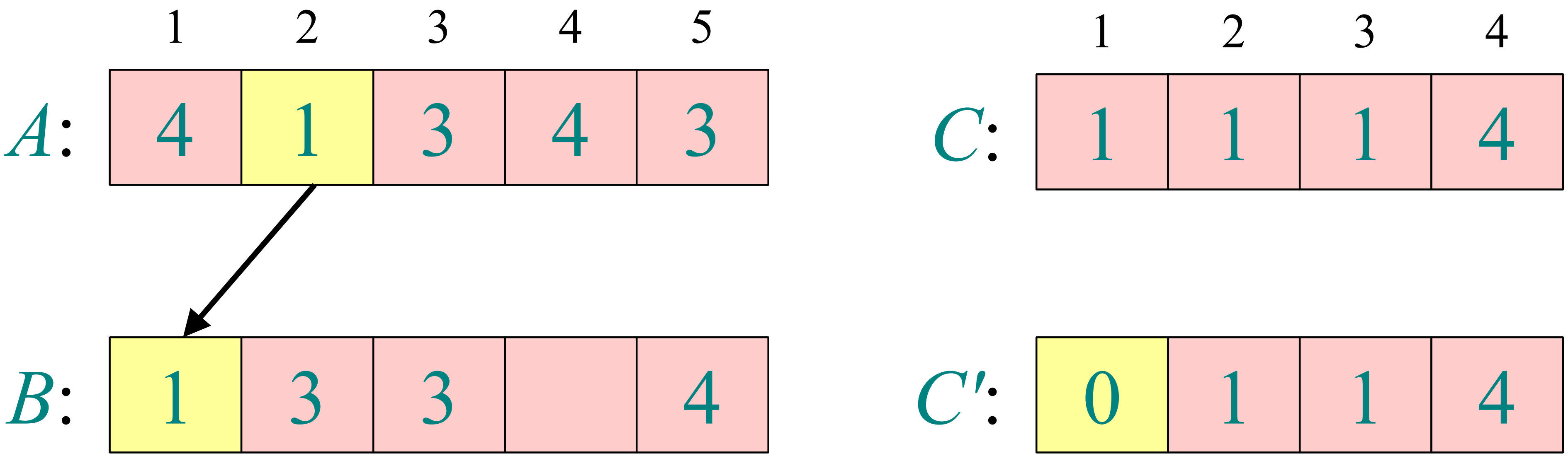


for $j \leftarrow n$ **downto** 1

  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

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 $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

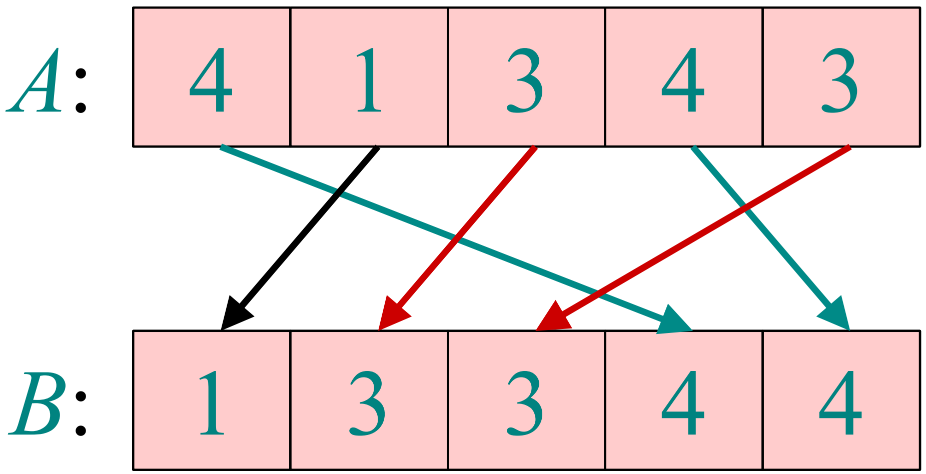
Counting Sort

- **Total time: O(n + k)**
  - Usually, $k = O(n)$
  - Thus counting sort runs in $O(n)$ time
- **But sorting is $\Omega(n \lg n)$!**
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is ***stable***

- 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 ($2^{32} = 4{,}294{,}967{,}296$)

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

- Intuitively, you might sort on the **most significant digit**, then the second msd, etc.
- **Problem:** lots of intermediate piles of cards (read: scratch arrays) to keep track of
- **Key idea**: sort the *least* significant digit first

```
RadixSort(A, d)
  for i=1 to d
    StableSort(A) on digit i
```

  - Example: Fig 9.3

Radix Sort

- ***Can we prove it will work?***
- **Sketch of an inductive argument** (induction on the number of passes):
  - Assume lower-order digits {j: j<i}are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Radix Sort

- ***What sort will we use to sort on digits?***
- **Counting sort is obvious choice:**
  - Sort n numbers on digits that range from $1..k$
  - Time: $O(n + k)$
- **Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$**
  - When d is constant and $k=O(n)$, takes $O(n)$ time
- ***How many bits in a computer word?***

Radix Sort

- **Problem:** sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
- **Compares well with typical O(n lg n) comparison sort**
  - Requires approximate log n = 20 operations per number being sorted
- ***So why would we ever use anything but radix sort?***
- **In general, radix sort based on counting sort is**
  - Fast, Asymptotically fast (i.e., O(n))
  - Simple to code
  - A good choice
- **To think about:** *Can radix sort be used on floating-point numbers?*

Operation of Radix Sort

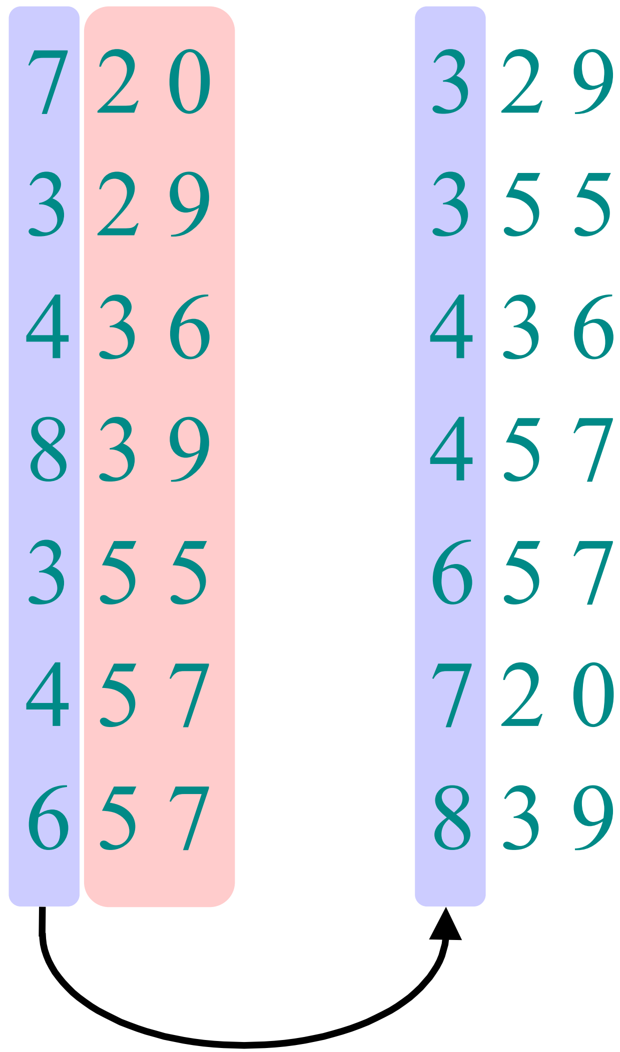
3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Correctness of Radix Sort

Induction on digit position

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

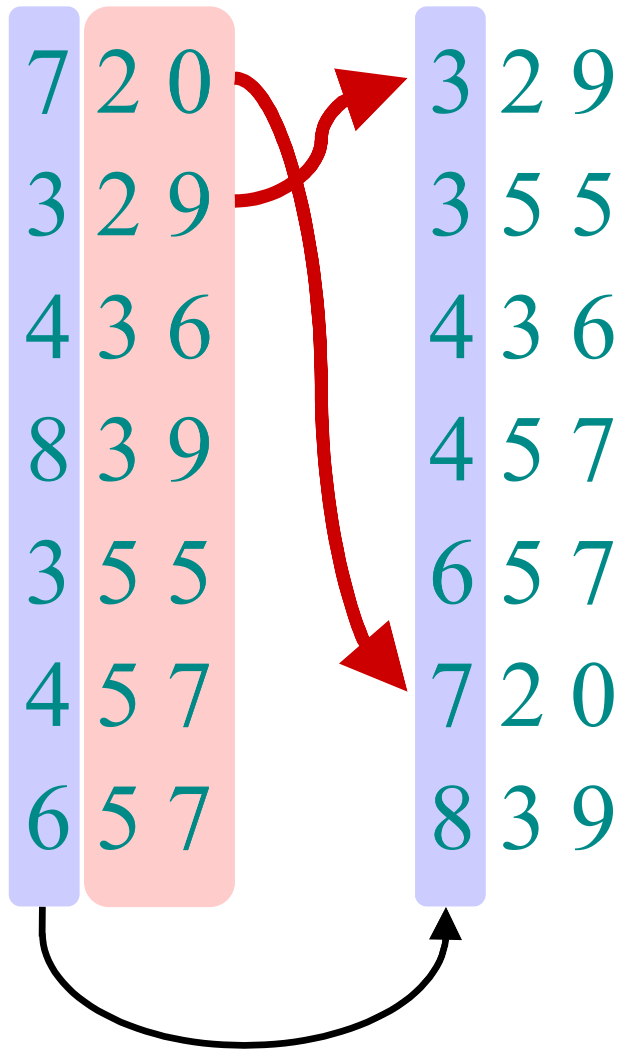
- Sort on digit t



Correctness of Radix Sort

Induction on digit position

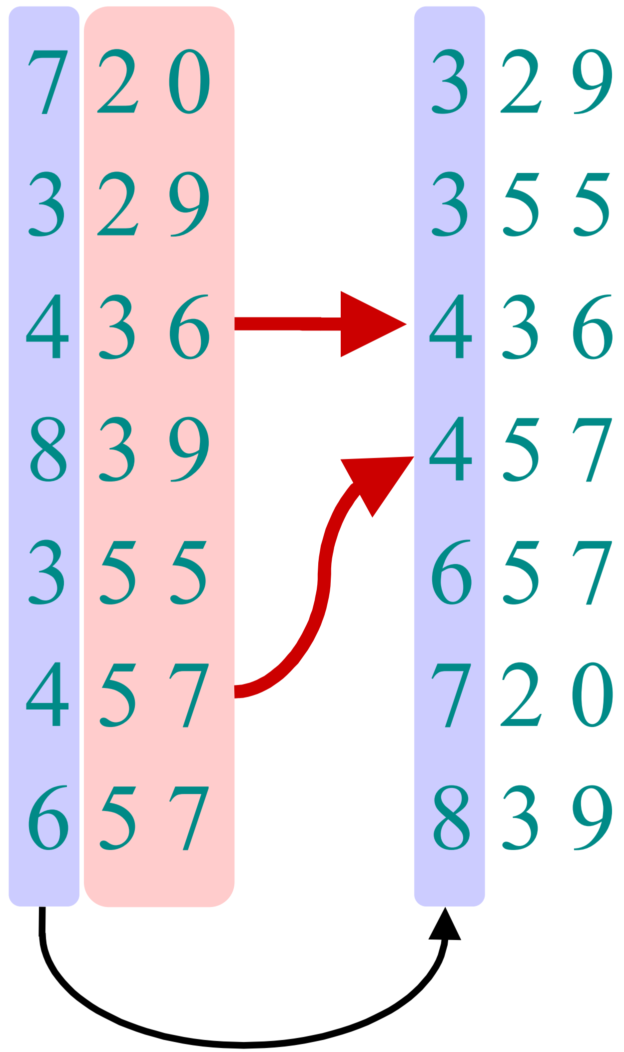
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.



Correctness of Radix Sort

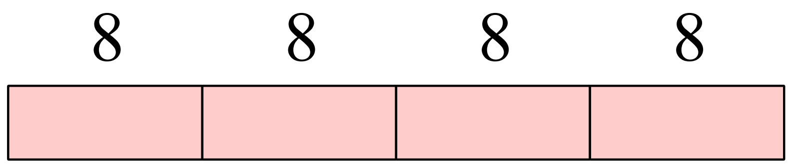
Induction on digit position

- 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   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

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

How many passes should we make?

Analysis of Radix Sort

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

$$\Theta((b/r)n + 2^r)$$

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

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

Choosing r

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

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

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

- 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.

Bucket Sort

- **Assumption: uniform distribution**
  - Input numbers are **uniformly distributed** in [0,1).
  - Suppose input size is n.
- **Idea:**
  - Divide [0,1) into n equal-sized subintervals (buckets).
  - Distribute n numbers into buckets
  - Expect that each bucket contains few numbers.
  - Sort numbers in each bucket (insertion sort as default).
  - Then go through buckets in order, listing elements,

BUCKET-SORT(A)

1. *n* ← length[A]

2. **for** *i* ← 1 to *n*

3. **do** insert A[*i*] into bucket B[⌊*n*A[*i*]⌋]

4. **for** *i* ← 0 to *n*-1

5. **do** sort bucket B[*i*] using insertion sort

6. Concatenate bucket B[0],B[1],…,B[*n*-1]

Example of BUCKET-SORT

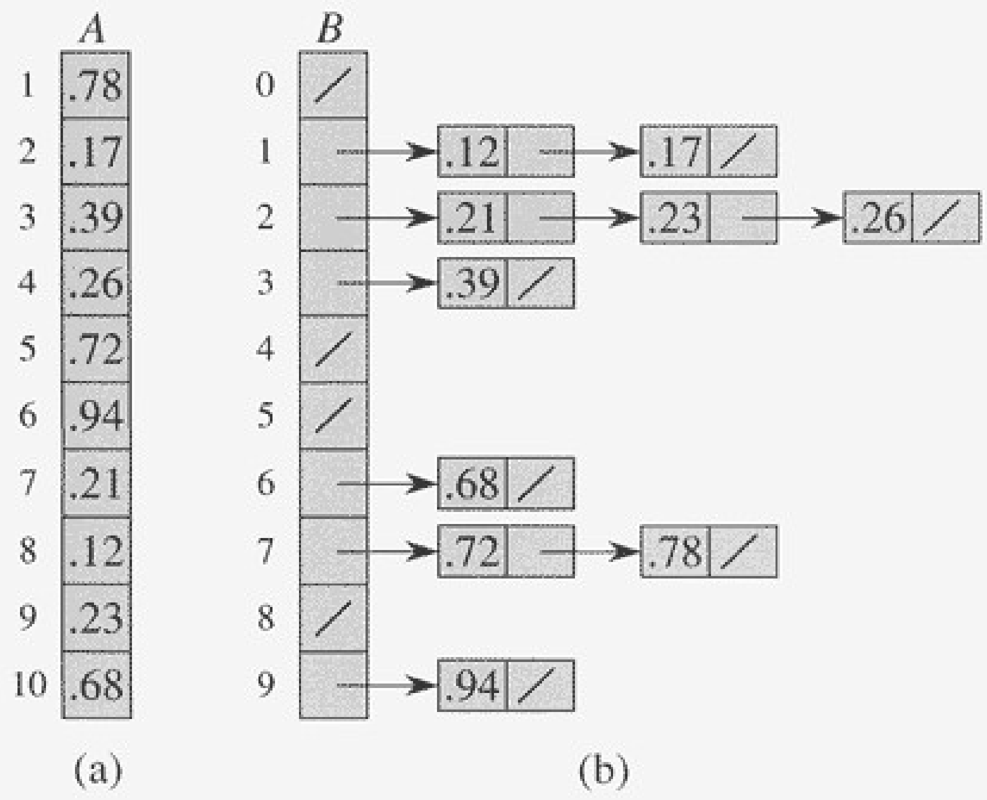


Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Analysis of BUCKET-SORT(A)

1. $n \leftarrow$ length[A] $\Omega(1)$
2. **for** $i \leftarrow 1$ to n $O(n)$
3. **do** insert A[i] into bucket B[$\lfloor n\text{A}[i] \rfloor$] $\Omega(1)$ (i.e. total $O(n)$)
4. **for** $i \leftarrow 0$ to n-1 $O(n)$
5. **do** sort bucket B[i] with insertion sort $O(n_i^2)$ $(\sum_{i=0}^{n-1} O(n_i^2))$
6. Concatenate bucket B[0],B[1],…,B[n-1] $O(n)$

Where n_i is the size of bucket B[i].

Thus $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$

$= \Theta(n) + n\, O(2\text{-}1/n) = \Theta(n)$

Analysis of BUCKET-SORT(A)

Time: $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$ (n_i: number of elements in i^{th} bucket)

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \quad \text{(linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) \quad (\mathrm{E}[aX] = a\mathrm{E}[X])
\end{aligned}
$$

$\mathrm{E}[n_i^2] = 2 - (1/n)$ => $\mathrm{E}[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n)$

$= \Theta(n)$