Design and Analysis of Algorithms

Sorting in linear time

Reference:
CLRS Chapter 8

Topics:

- **Lower bound for sorting**
- **Counting sort**
- **Radix sort**
- **Bucket sort**

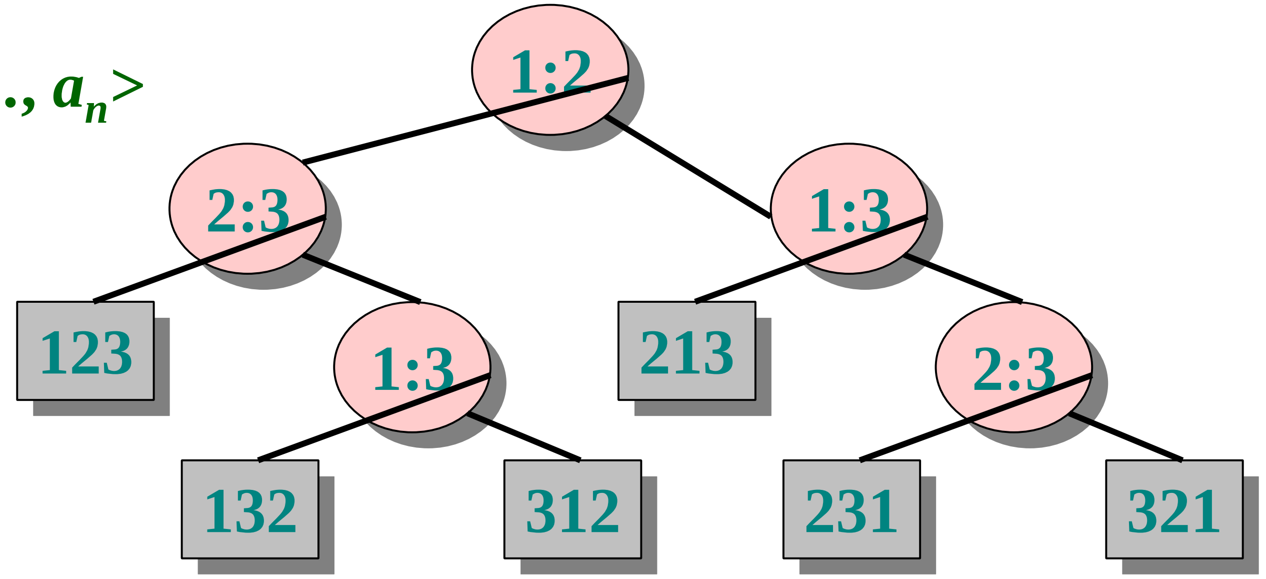
How fast can we sort?

- 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 $O(n \lg n)$.

- Q: Is $O(n \lg n)$ the best we can do?
- A: Yes, as long as we use comparison sorts
- TODAY: Prove any comparison sort has $\Omega(n \lg n)$ worst case running time

Decision-tree example

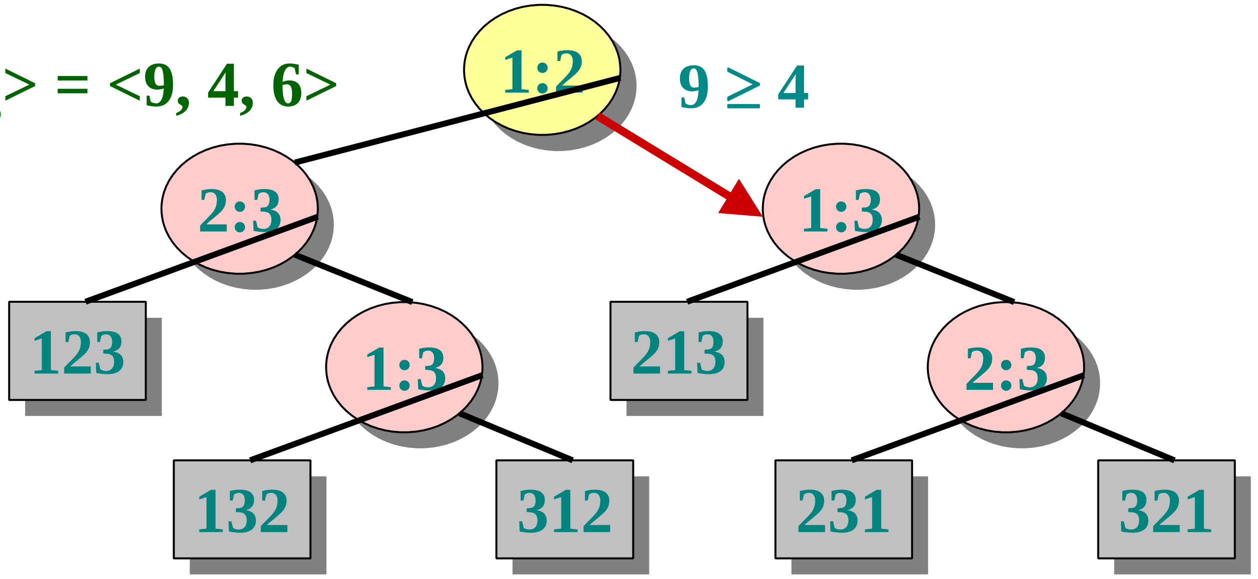
- Sort $\langle a_1, a_2, \ldots, a_n \rangle$



- Each internal node is labelled $i{:}j$ 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

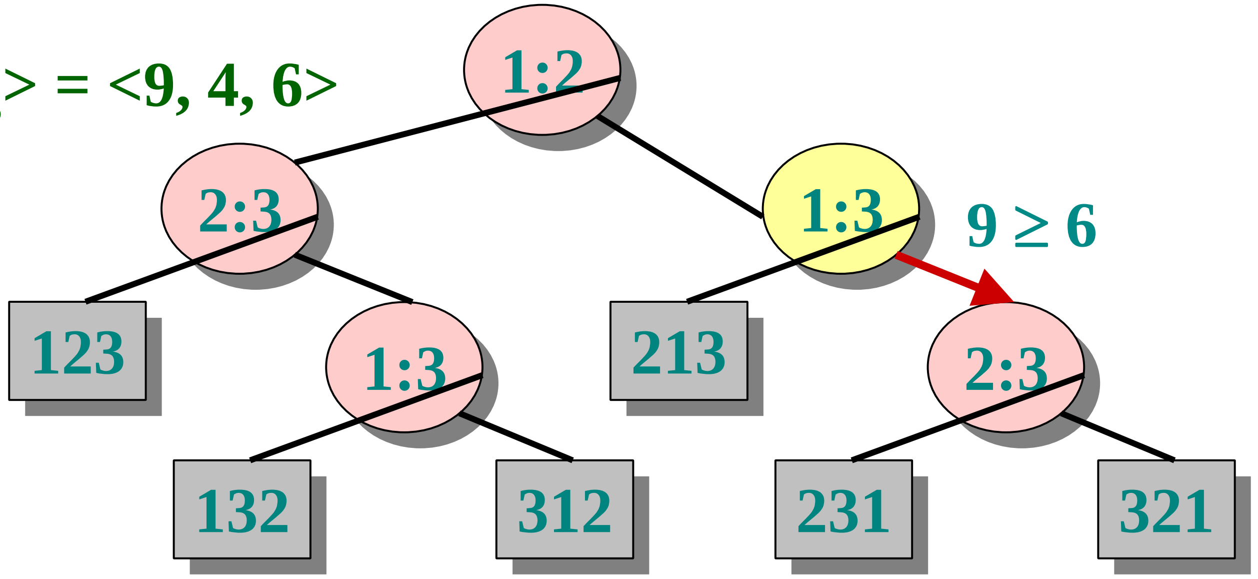
- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$



- Each internal node is labelled $i{:}j$ 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

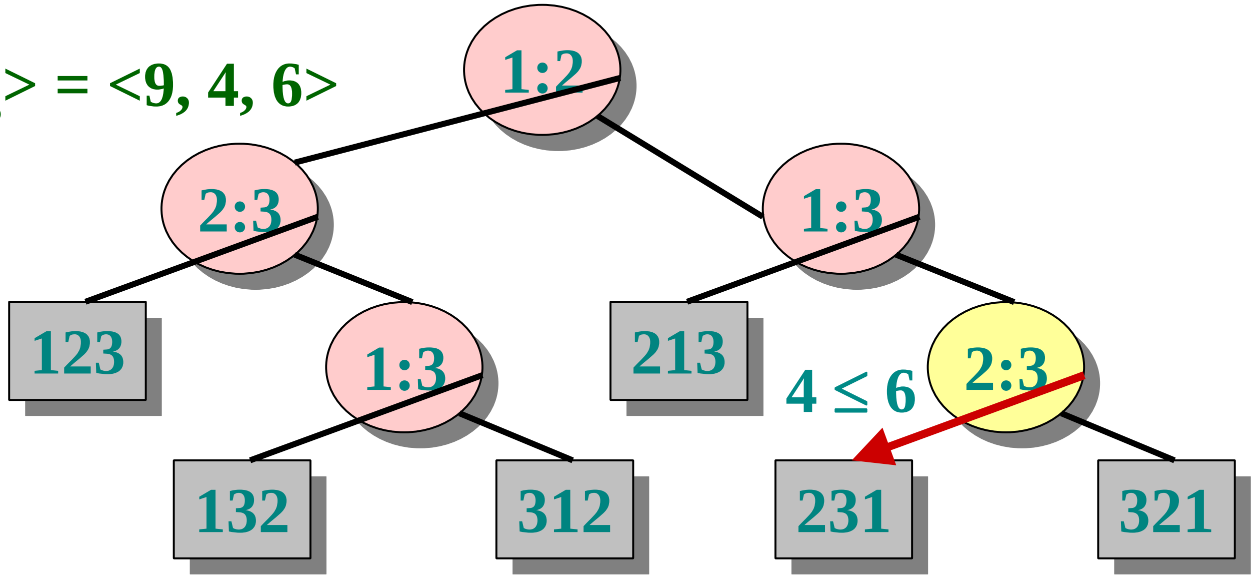
- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$



- Each internal node is labelled $i{:}j$ 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

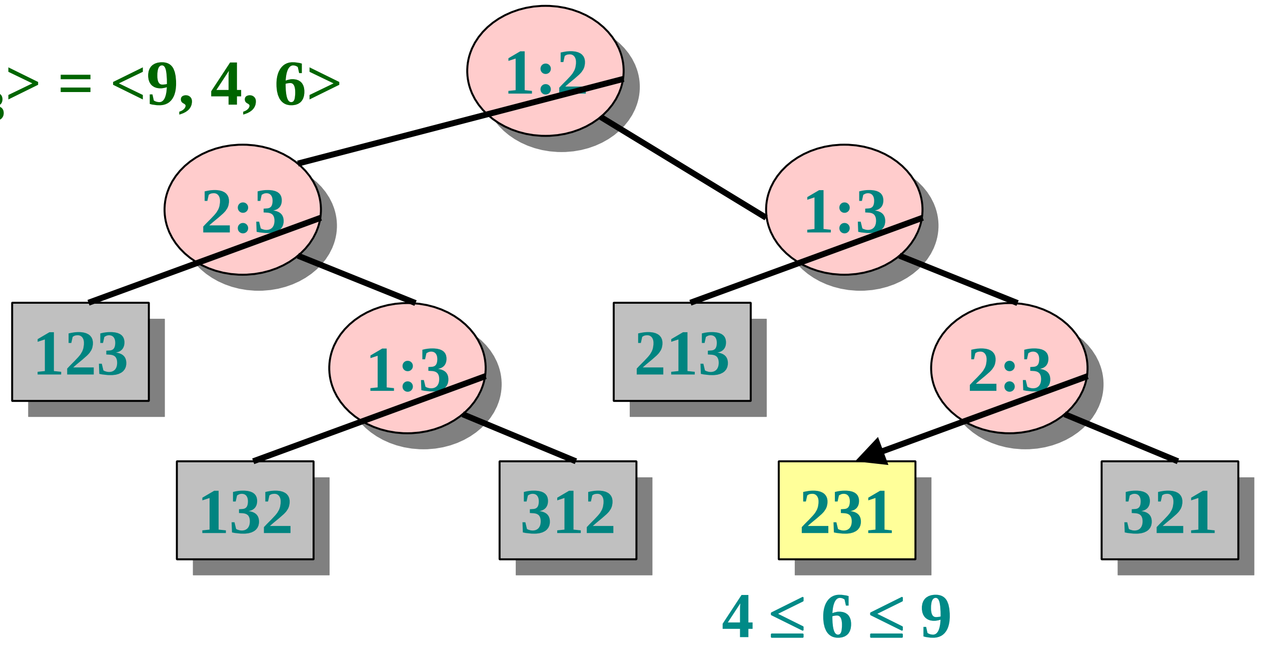
- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$



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

Decision-tree model

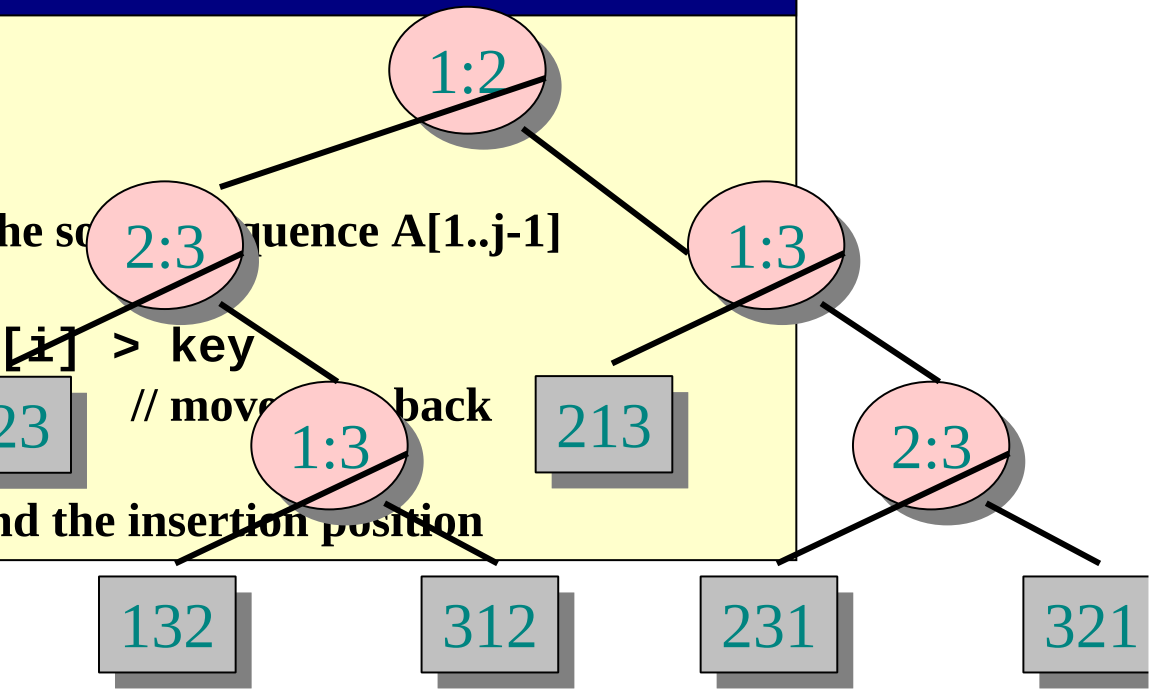
- A decision tree can model the execution of any comparison sort:
  - One tree for each input size *n*.
  - View the algorithm as splitting whenever it compares two elements.
  - The tree contains the comparisons along all possible instruction traces.
  - The running time of the algorithm = the length of the path taken.
  - Worst-case running time = height of tree.

Any comparison sort

- Can be turned into a Decision tree

INSERTION SORT

```
INSERTION-SORT(A)
1 for j ← 2 to length(A)
2     do key ← A[j]
3             // insert A[j] into the sc          quence A[1..j-1]
4         i ← j – 1
5         while i > 0 and A[i] > key
6             do A[i+1] ←          // move        back
7                 i ← i –
8         A[i+1] ← key    //find the insertion position
```



Lower Bound for decision-tree Sorting

- **Theorem.** Any comparison sorting algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

- **Proof.** Worst case dictated by tree height h.
  - $n!$ different orderings.
  - One (or more) leaves corresponding to each ordering.
  - Binary tree with $n!$ leaves must have height

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

Lower Bound for Comparision Sorting

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

Sorting Lower bound

- **Is there a faster algorithm?**
  If different model of computation?

INSERTION SORT

```
INSERTION-SORT(A)
for j ← 2 to length(A)
   do key ← A[j]
          // insert A[j] into the sorted sequence A[1..j-1]
      i ← j – 1
      while i > 0 and A[i] > key
         do A[i+1] ← A[i]   // move item back
            i ← i – 1
      A[i+1] ← key   //find the insertion position
```

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

COUNTING SORT

```
COUNTING-SORT(A, B, k)
1   for i ← 1 to k
2       do C[i] ← 0
3   for j ← 1 to length[A]
4       do C[A[j]] ← C[A[j]]+1
5   //C[i] now contains the number of elements equal to i.
6   for i ← 2 to k
7       do C[i] ← C[i]+ C[i-1]
8   //C[i] now contains the number of elements less than or equal to i.
9   for j ← length[A] downto 1
10      do B[C[A[j]]] ← A[j]
11         C[A[j]] ← C[A[j]]-1
```

Counting-sort example

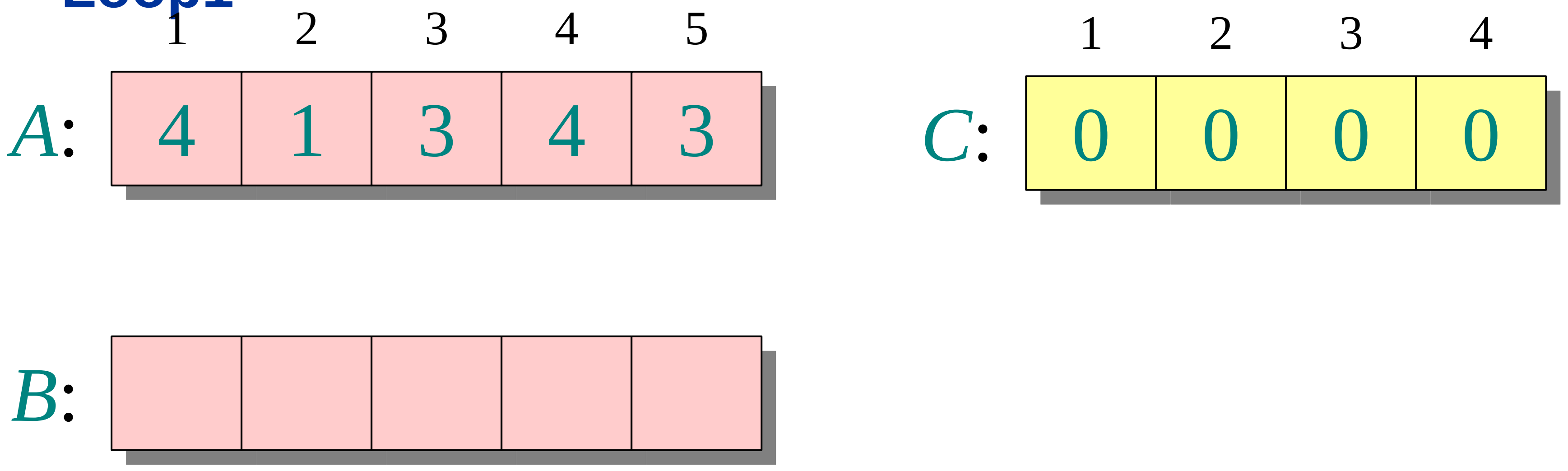
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:				

B:					

Counting-sort example

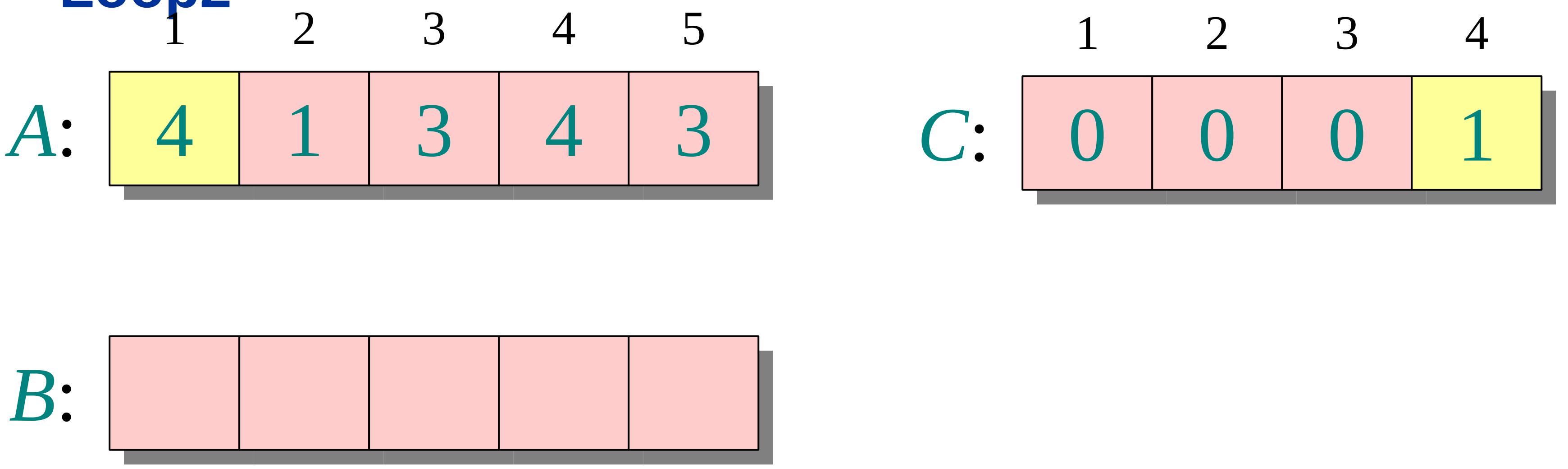
- **Loop1**



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

Counting-sort example

- **Loop2**



```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$        // $C[i] = |\{\text{key} = i\}|$
```

Counting-sort example

- **Loop2**

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

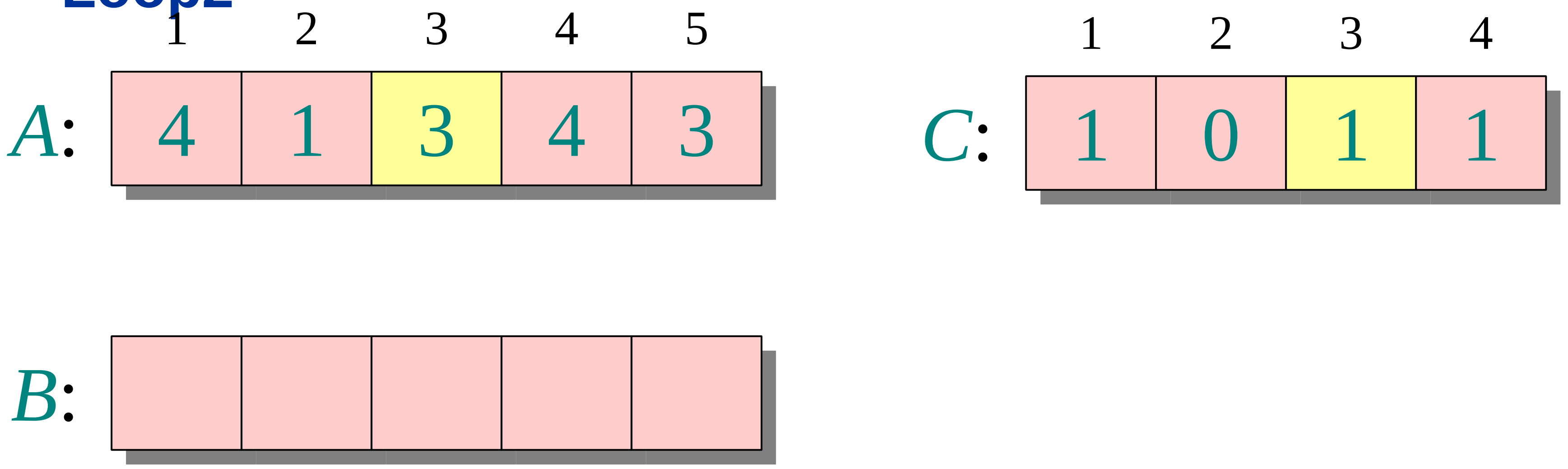
	1	2	3	4
C:	1	0	0	1

B:					

```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$        // $C[i] = |\{\text{key} = i\}|$
```

Counting-sort example

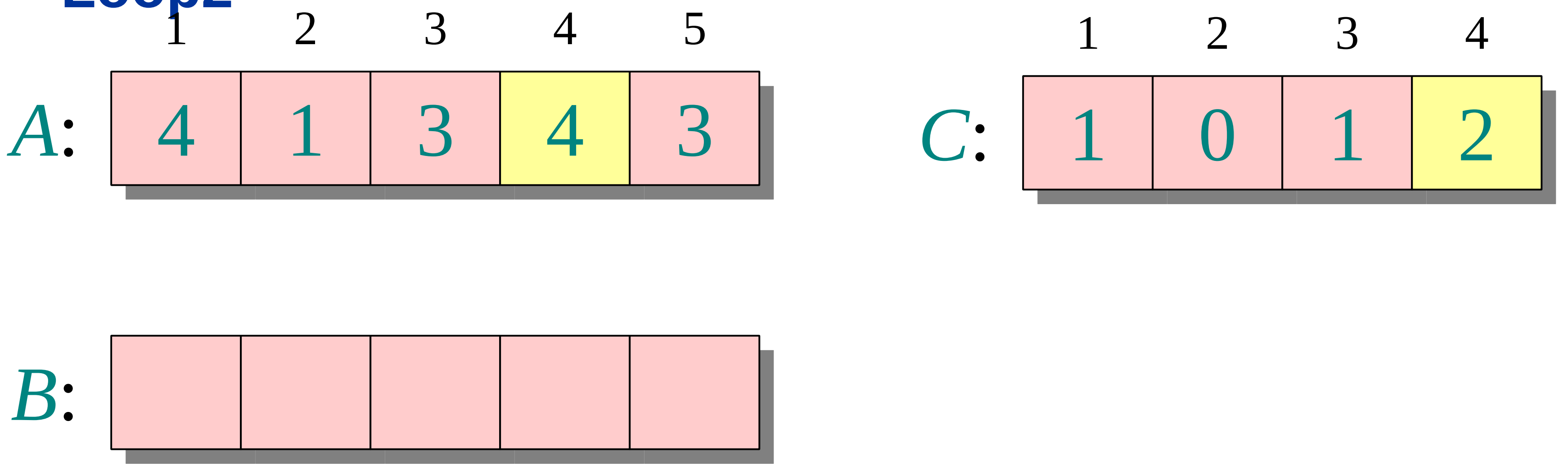
- Loop2



```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$        // $C[i] = |\{\text{key} = i\}|$
```

Counting-sort example

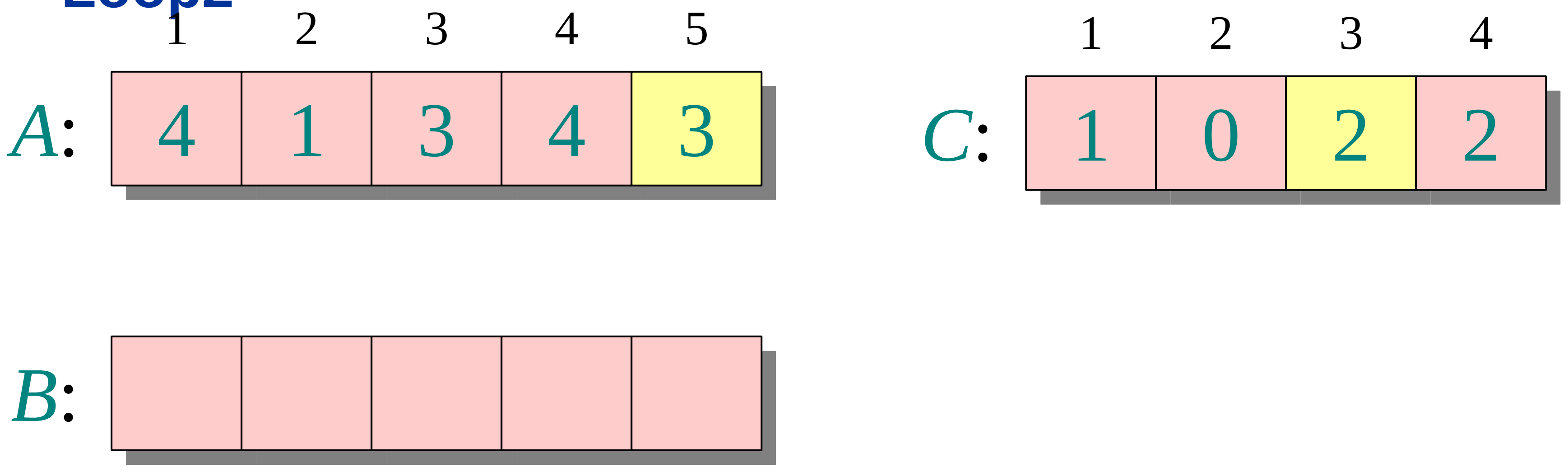
- **Loop2**



```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$        // $C[i] = |\{\text{key} = i\}|$
```

Counting-sort example

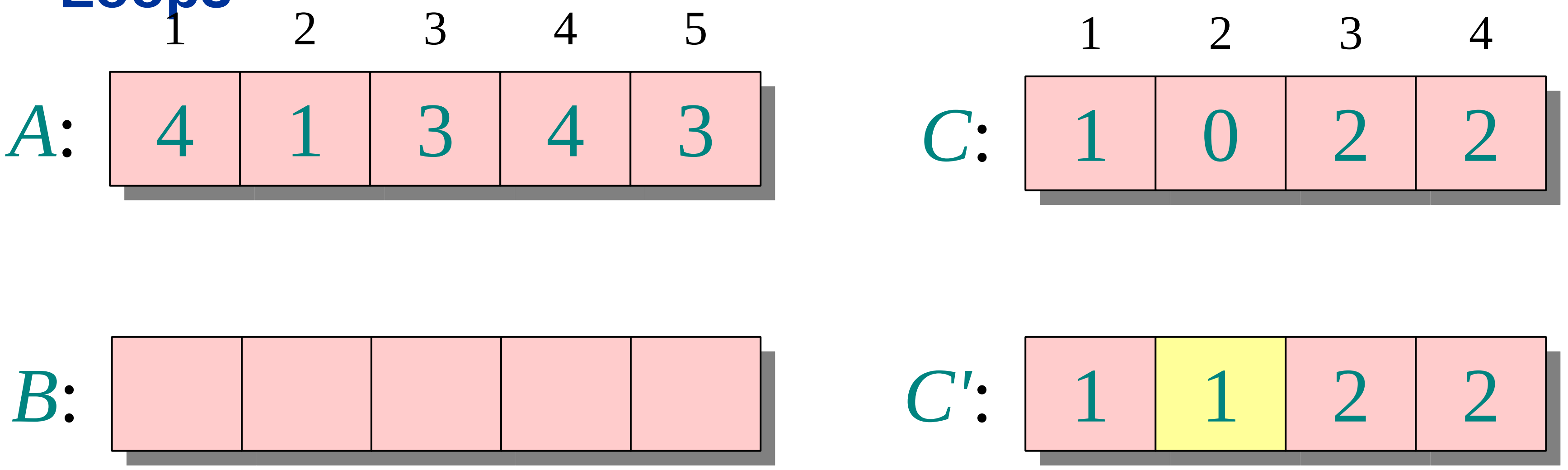
- **Loop2**



```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$        // $C[i] = |\{\text{key} = i\}|$
```

Counting-sort example

- **Loop3**



```
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$          // $C[i] = |\{\text{key} \le i\}|$
```

Counting-sort example

- **Loop3**

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

	1	2	3	4
C:	1	0	2	2

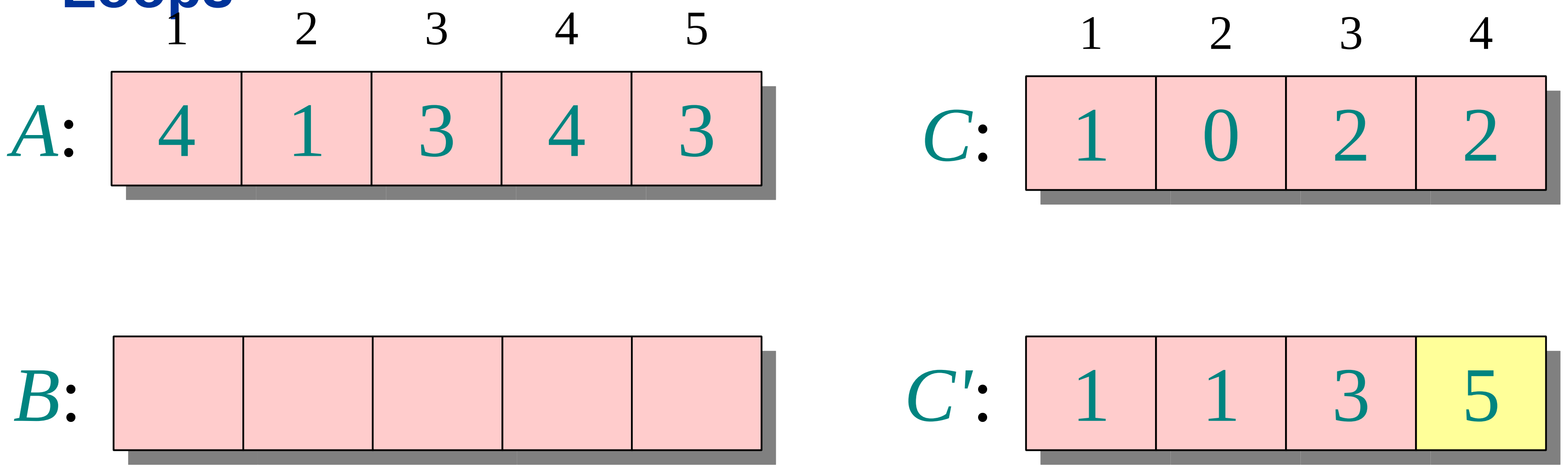
B:					

C':	1	1	3	2

```
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$            // $C[i] = |\{\text{key} \le i\}|$
```

Counting-sort example

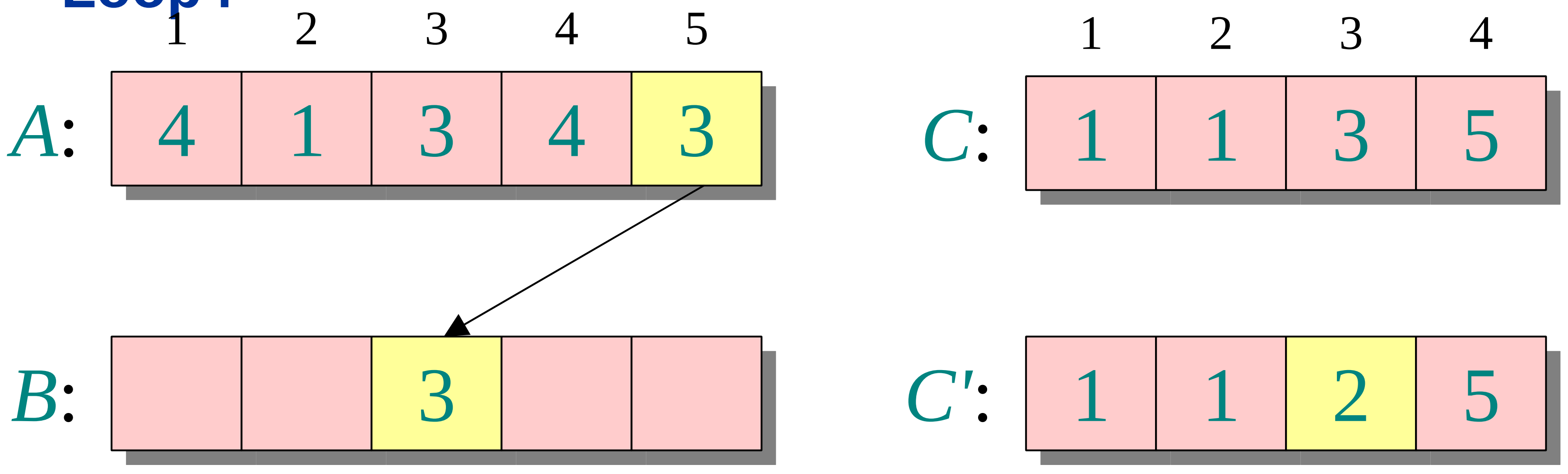
- **Loop3**



```
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$        // $C[i] = |\{\text{key} \le i\}|$
```

Counting-sort example

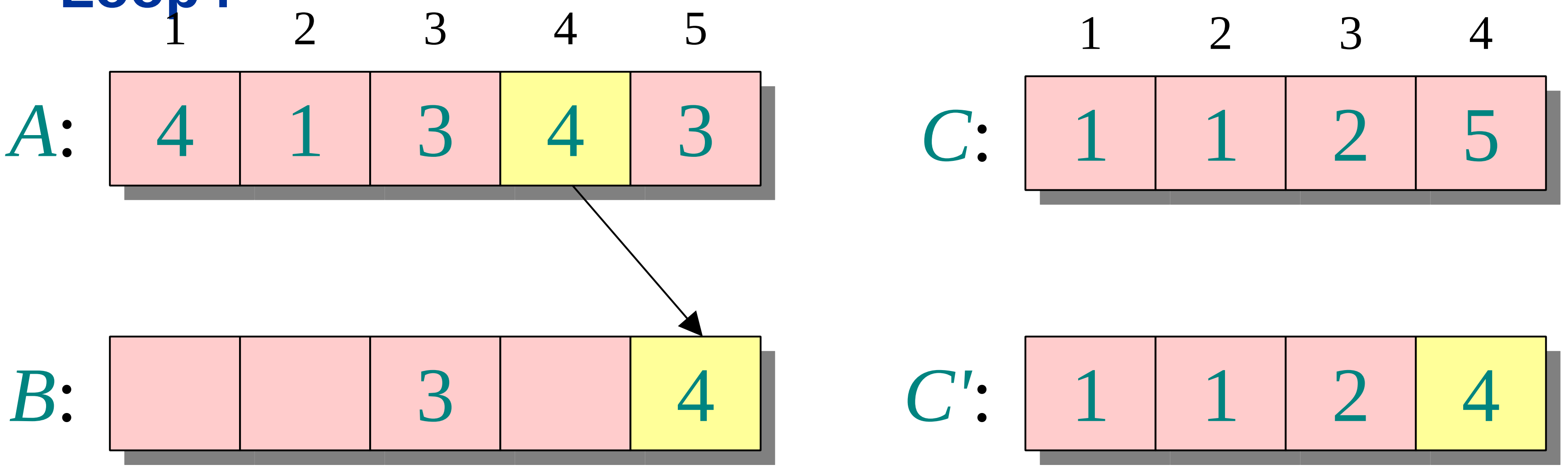
- **Loop4**



```
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

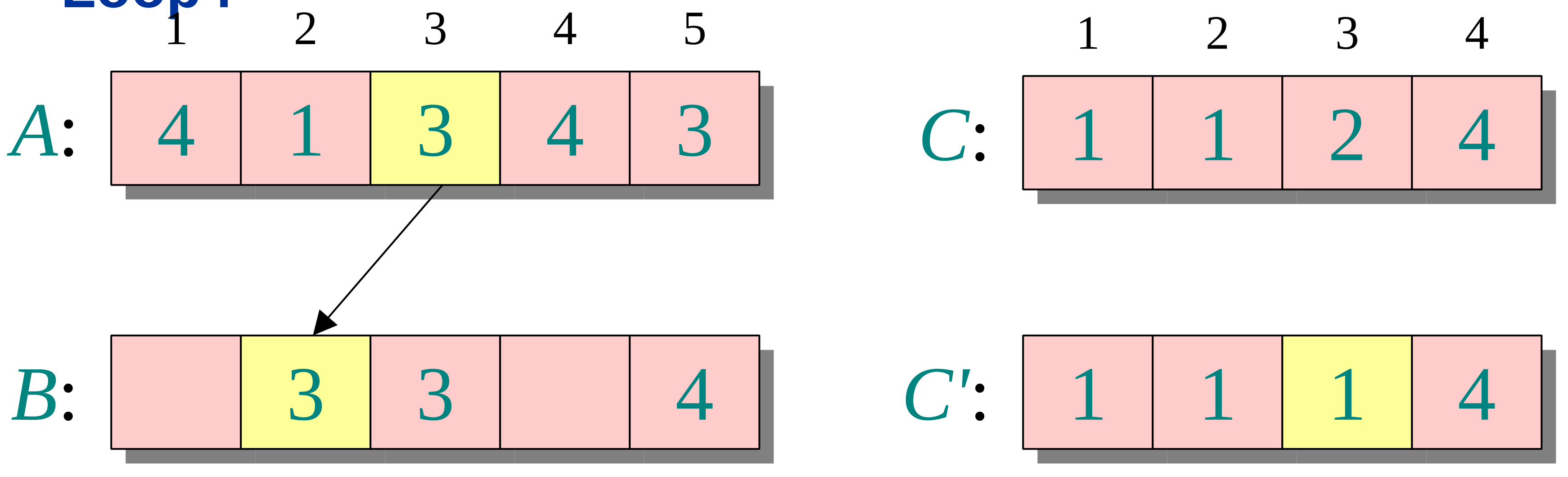
- Loop4



```
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

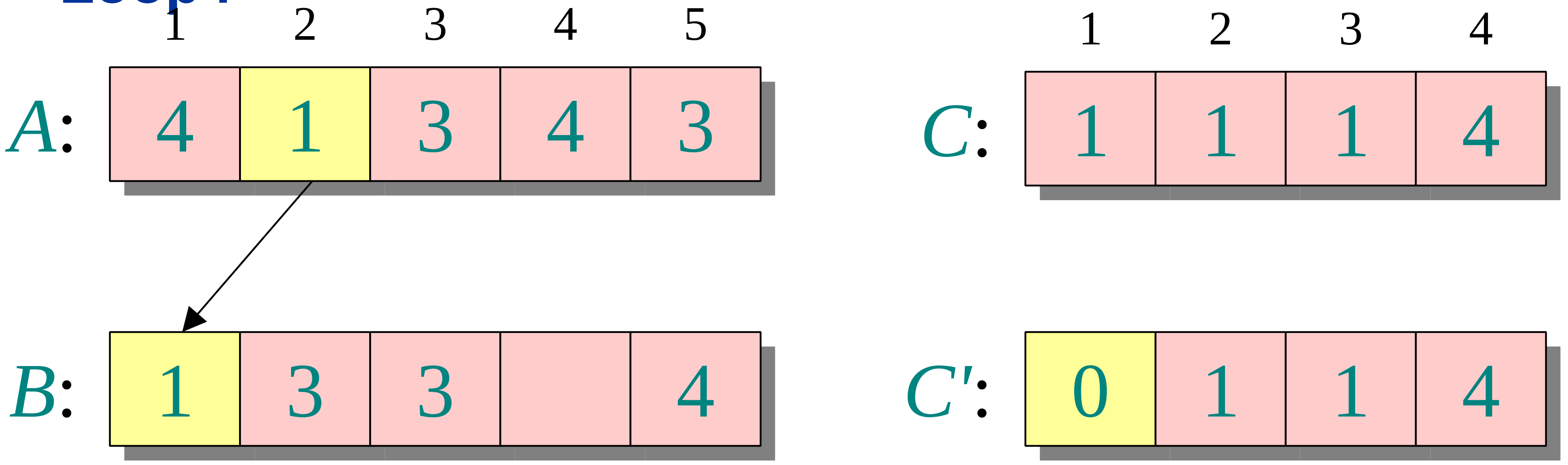
- **Loop4**



```
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

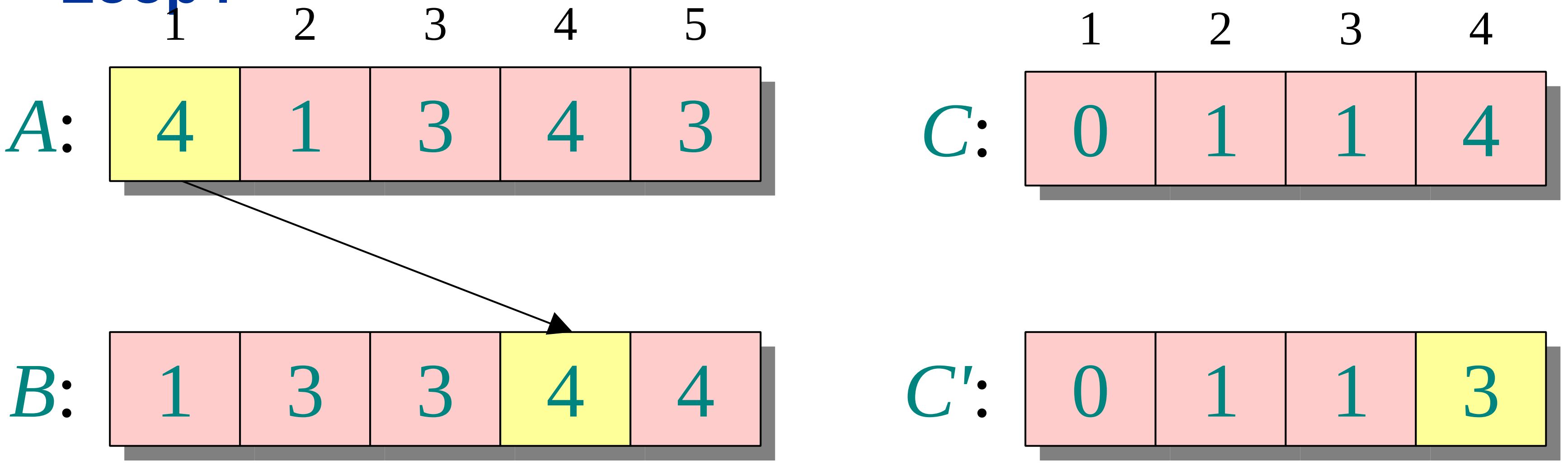
- Loop4



```
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

- **Loop4**



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

Analysis

COUNTING SORT

```
COUNTING-SORT(A, B, k)
1   for i ← 1 to k
2       do C[i] ← 0                          } $\Theta(k)$
3   for j ← 1 to length[A]
4       do C[A[j]] ← C[A[j]]+1               } $\Theta(n)$
5   //C[i] now contains the number of elements equal to i.
6   for i ← 1 to k
7       do C[i] ← C[i]+ C[i-1]               } $\Theta(k)$
8   //C[i] now contains the number of elements less than or equal to i.
9   for j ← length[A] downto 1
10      do B[C[A[j]]] ← A[j]                 } $\Theta(n)$
11         C[A[j]] ← C[A[j]]-1
```

- **The overall time is $\Theta(k+n)$**

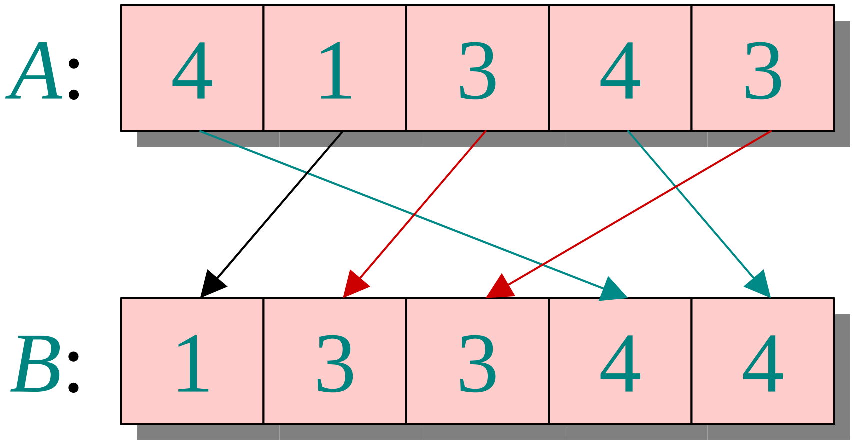
Running time

- 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

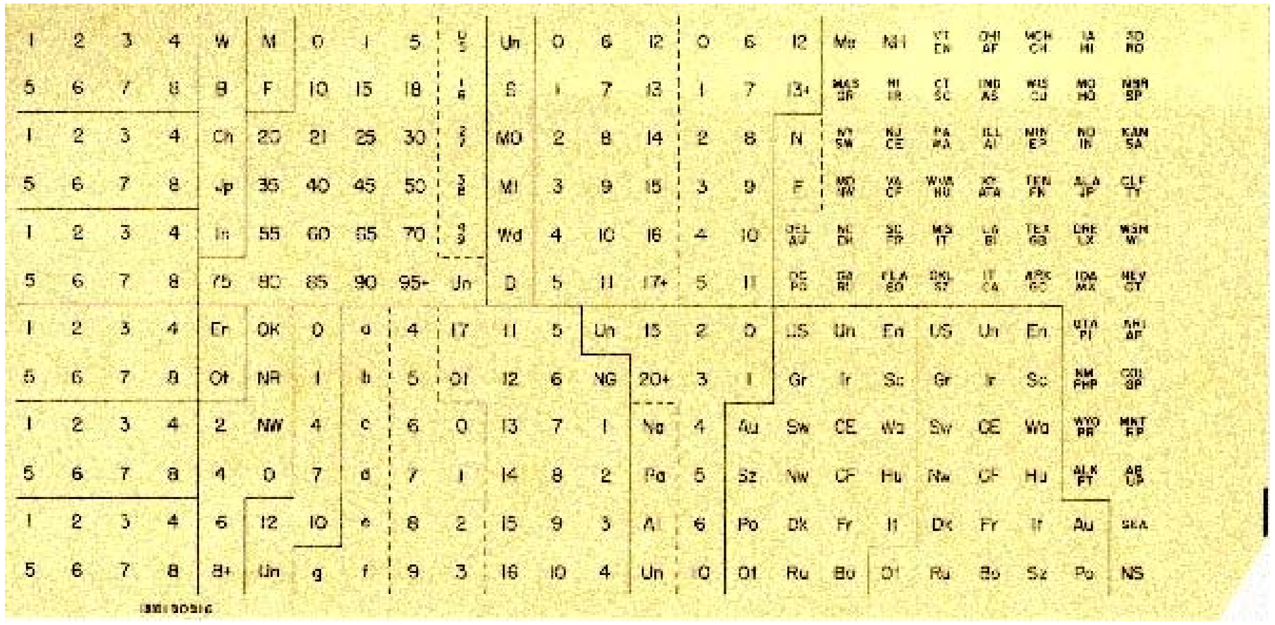
- Counting sort is a **stable** sort: it preserves the input order among equal elements.



- Exercise: What other sorts have this property?

Punched cards

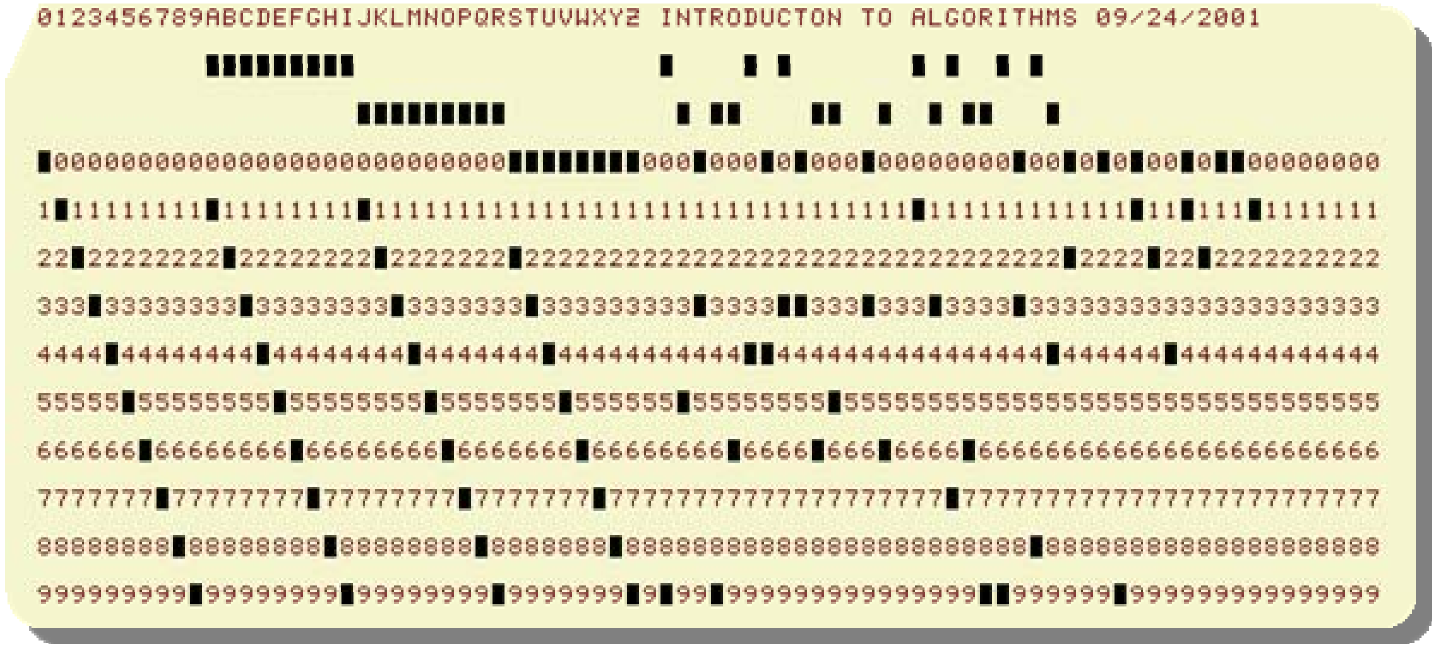
- Punched card = data record.
- Hole = value.
- Algorithm = machine + human operator.



Replica of punch card from the 1900 U.S. census. [Howells 2000]

“Modern” IBM card

- One character per column.


0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ INTRODUCTON TO ALGORITHMS 09/24/2001

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.

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.

Hollerith's tabulating system

- Pantograph card punch
- Hand-press reader
- Dial counters
- Sorting box

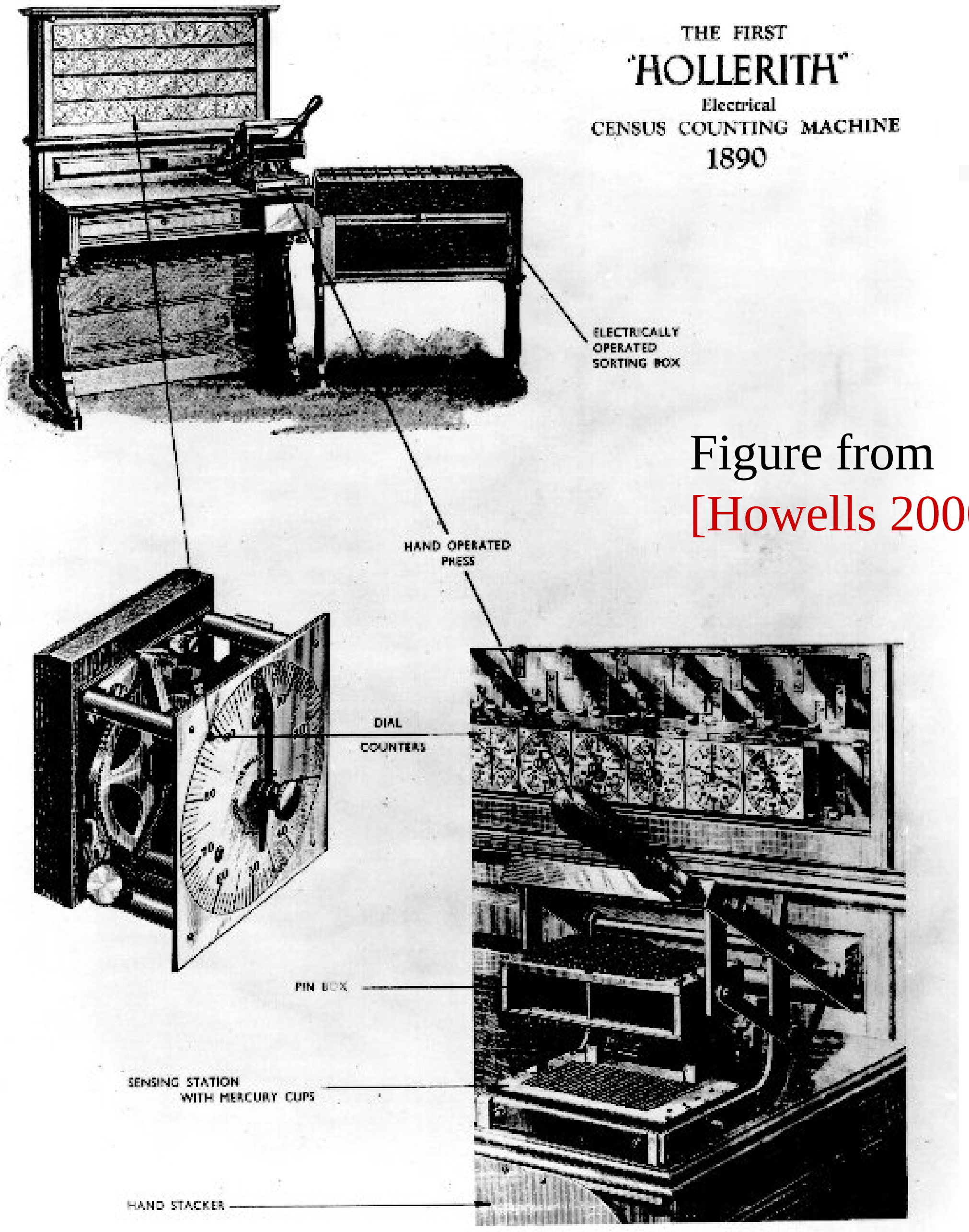
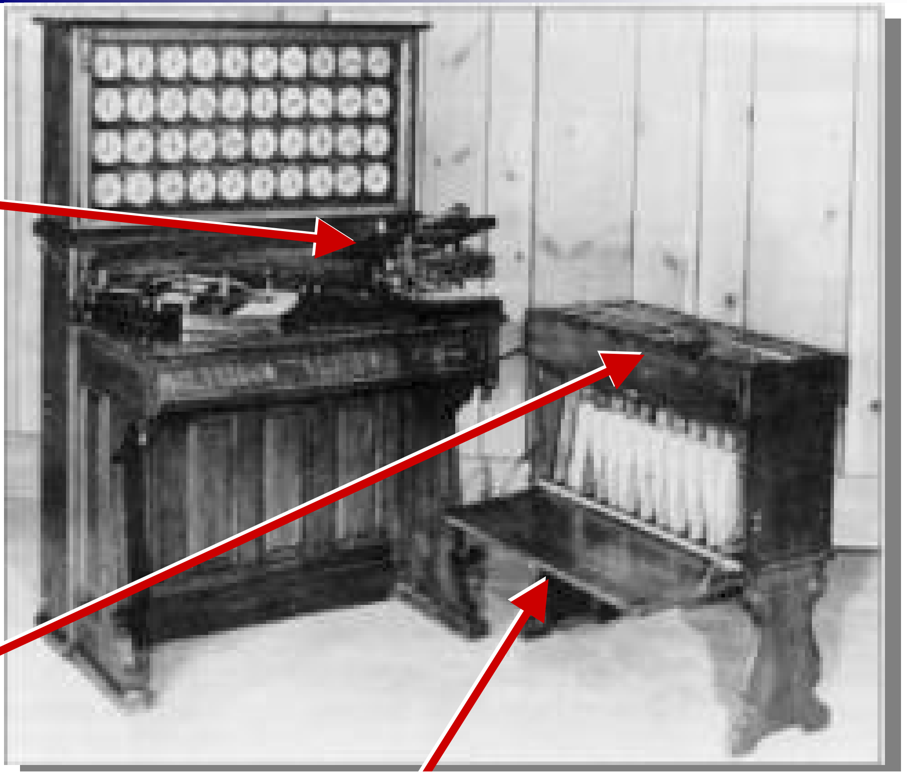


Figure from [Howells 2000].

Operation of the sorter

- An operator inserts a card into the press.
- Pins on the press reach through the punched holes to make electrical contact with mercury-filled cups beneath the card.
- Whenever a particular digit value is punched, the lid of the corresponding sorting bin lifts.
- The operator deposits the card into the bin and closes the lid.


Hollerith Tabulator, Pantograph, Press, and Sorter

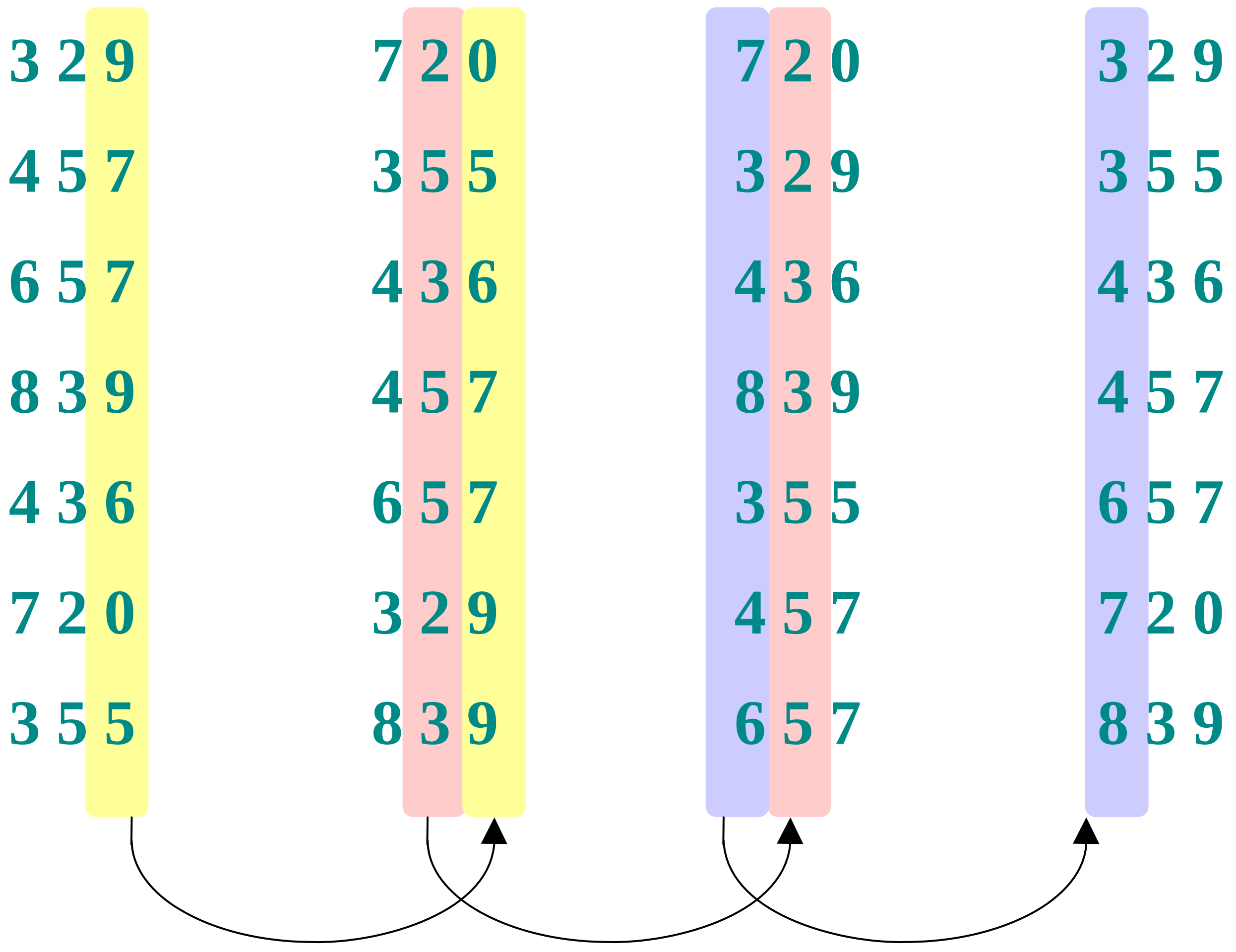
- When all cards have been processed, the front panel is opened, and the cards are collected in order, yielding one pass of a stable sort.

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

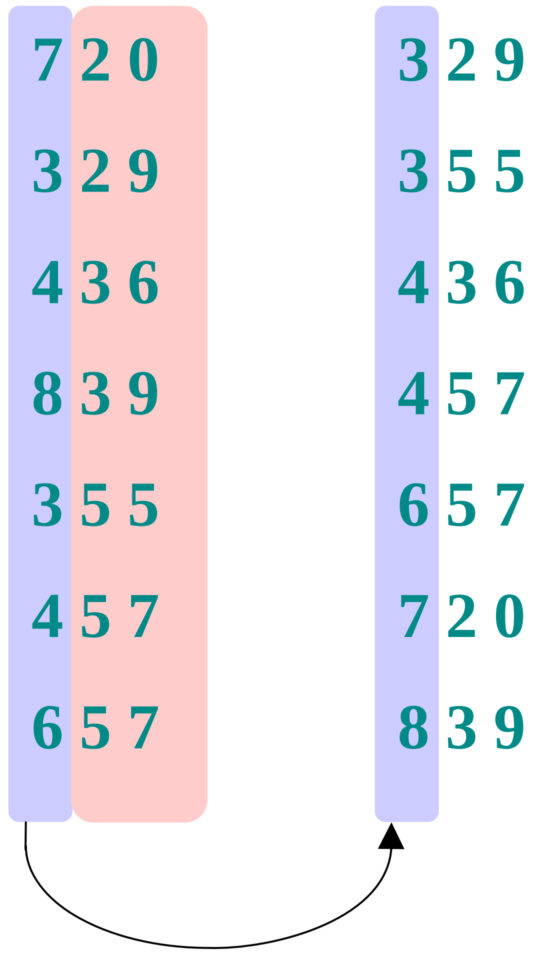


Correctness of radix sort

Induction on digit position

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

- **Sort on digit t**

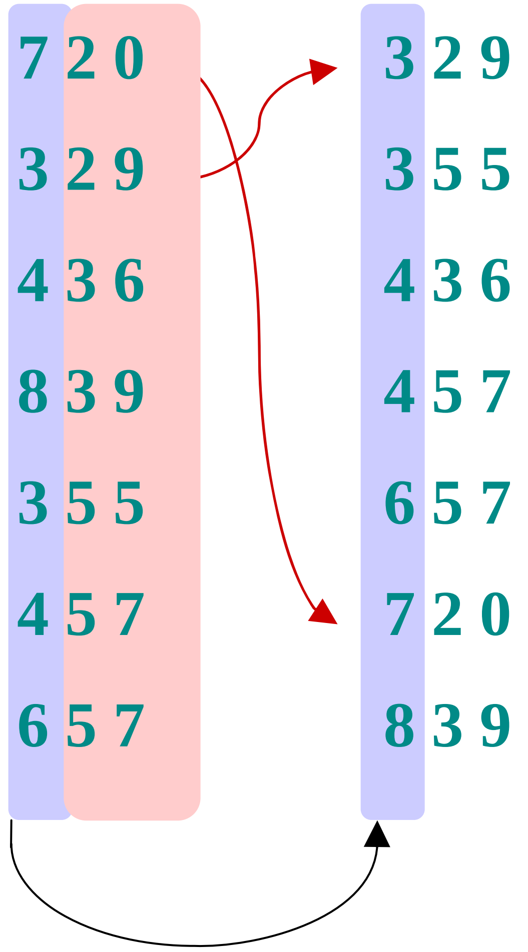


Correctness of radix sort

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.

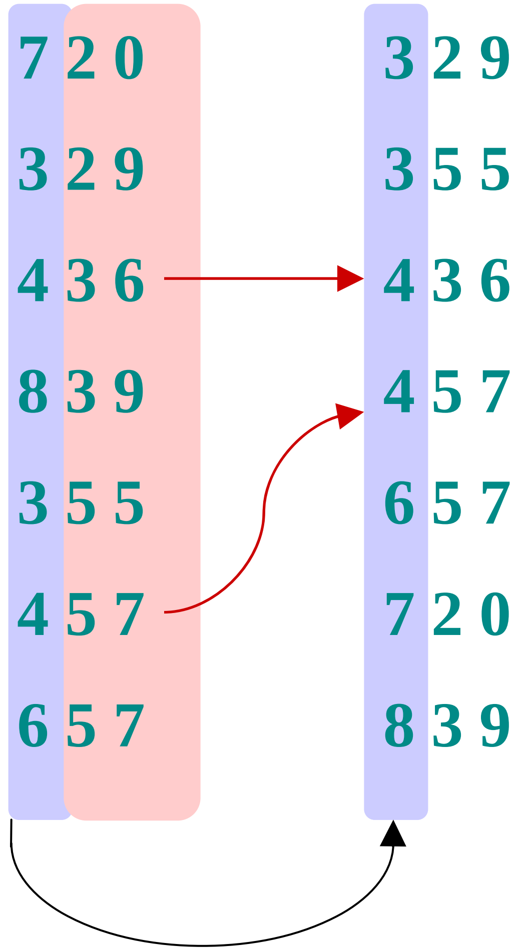


Correctness of radix sort

Induction on digit position

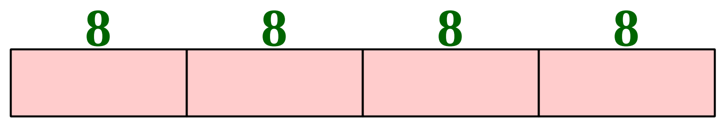
- **Assume that the numbers are sorted by their low-order $t-1$ digits.**

- **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

$$T(n, b) = (b/r)(n + 2^r)$$

- Choose r to minimize $T(n, b)$:
  - Increasing r means fewer passes, but as $r \gg \lg n$, 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.

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: Can’t sort in place using counting sort. Also, Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better sometimes on modern processors, with steep memory hierarchies.

Appendix: Punched-card technology

- Herman Hollerith (1860-1929)
  - 利用凿孔把字母信息在卡片上编码的一种方式(以美国发明人赫尔曼·霍尔瑞斯 **Herman Hollerith** 命名 1860-1929)
- Punched cards
- Hollerith’s tabulating system
- Operation of the sorter
- Origin of radix sort
- “Modern” IBM card
- Web resources on punched-card technology

Web resources on punched-card technology

- Doug Jones's punched card index
- Biography of Herman Hollerith
- The 1890 U.S. Census
- Early history of IBM
- Pictures of Hollerith's inventions
- Hollerith's patent application (borrowed from Gordon Bell's CyberMuseum)
- Impact of punched cards on U.S. history

Bucket Sort

- **Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly.**

- **Idea of Bucket Sort**
  - **Divide the interval $[0,1)$ into n equal-sized subintervals, or bucket, and then distribute the n input number into the buckets.**

Bucket Sort

- Assume that all the input is generated by a random process that distributes elements uniform over the interval $[0,1)$
- Bucket sort
  - ① Allocate a bucket for each value of the key
    - » That is, insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
  - ② For each bucket,
    - » sort list $B[i]$ with insertion sort
  - ③ concatenate the list $B[0], B[1], \ldots, B[n-1]$ together in order

Bucket Sort Example

- A=(0.5, 0.1, 0.3, 0.4, 0.3, 0.2, 0.1, 0.1, 0.5, 0.4, 0.5)

bucket in array	
B[1]	0.1, 0.1, 0.1
B[2]	0.2
B[3]	0.3, 0.3
B[4]	0.4, 0.4
B[5]	0.5, 0.5, 0.5

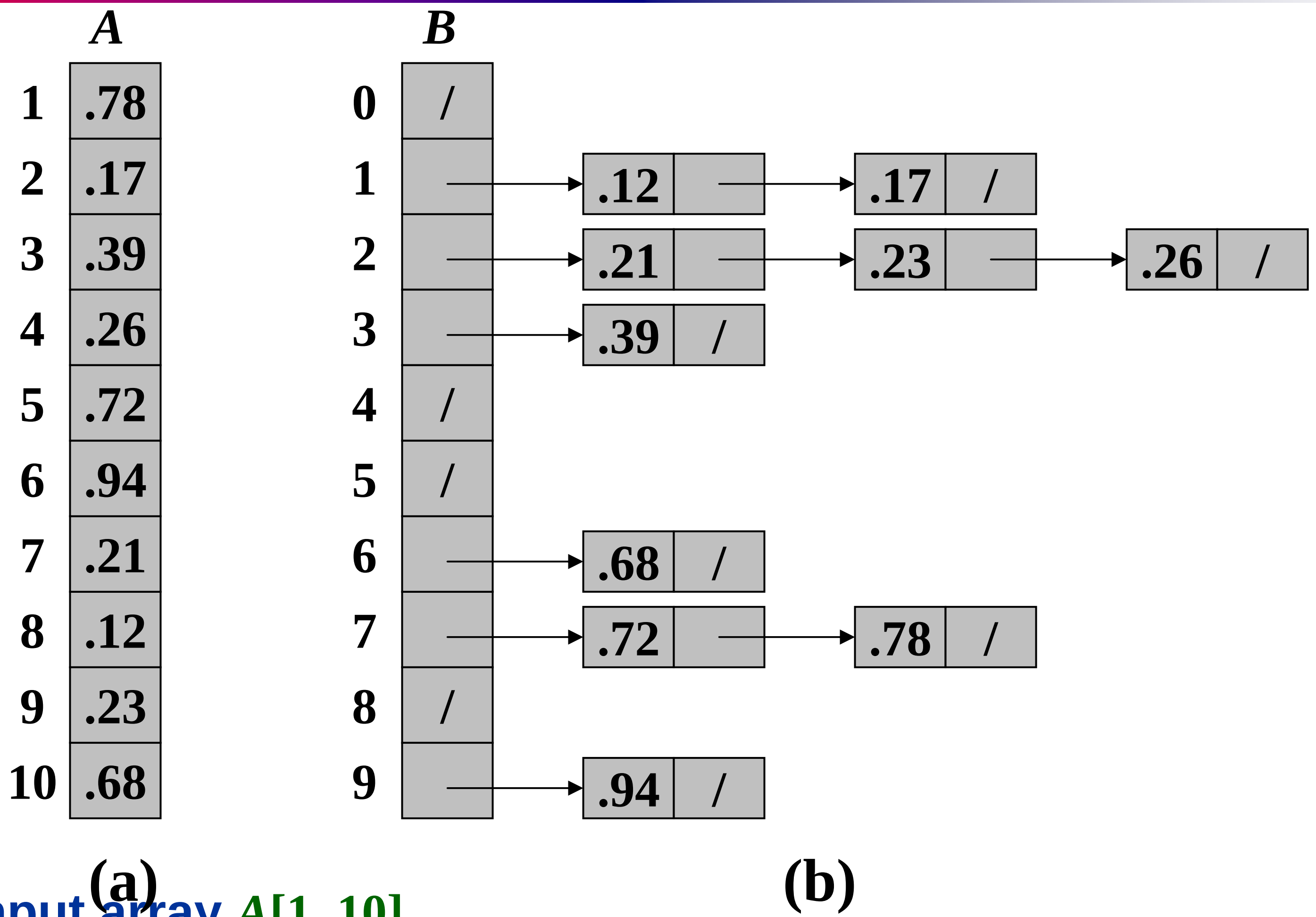
Sorted list:
0.1, 0.1, 0.1, 0.2, 0.3, 0.3,
0.4, 0.4, 0.5, 0.5, 0.5

Bucket Sort Algorithm

BUCKET SORT

```
BUCKET-SORT(A)
1 n ← length[A]
2 for i ← 1 to n
3   do insert A[i] into list B[⌊nA[i]⌋]
4 for i ← 1 to n-1
5   do sort list B[i] with insertion sort
6 concatenate the list B[0],B[1],…,B[n-1] together in order
```

Another Example



(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)

Algorithm Analysis

- All lines except line 5 take $O(n)$ time in the worst case. Total time to examine all buckets in line 5 is $O(n)$, without the sorting time.
- To analyze sorting time, let n_i be a random variable denoting the number of elements placed in bucket $B[i]$. The total time to sort is

$$ET(n) = E(\Theta(n) + \sum_{0 \le i \le n-1} O(n_i^2)) = O(\sum_{0 \le i \le n-1} E[n_i^2])$$

Since, $E[n_i^2] = \mathrm{Var}[n_i] + E^2[n_i]$

$$= np(1 - p) + 1^2 = 1 - (1/n) + 1$$

$$= 2 - 1/n = \Theta(1)$$

- The expected time for bucket sort is $\Theta(n)$.

Summary of Sorting Algorithms

Sorting methods	Worst case	Best case	Average case	Application
Insert Sort	n^2	n	n^2	Very fast when n<50
Bubble Sort	n^2	n	n^2	Very fast when n<50
Merge Sort	$n \lg n$	$n \lg n$	$n \lg n$	Need extra space;good for external sort
Heap Sort	$n \lg n$	$n \lg n$	$n \lg n$	Good for real-time app.
Quick Sort	n^2	$n \lg n$	$n \lg n$	Practical and fast
Bucket Sort	$n \lg n$	n	n	uniform distribution
Radix Sort	$d(n+ k)$	$d(n+ k)$	$d(n+ k)$	Small, fixed range Need extra spaces

- **Which In place?**
- **Which stable?**