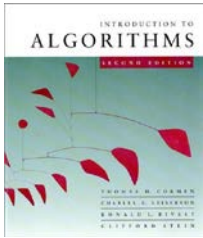


# Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).



# Divide and conquer

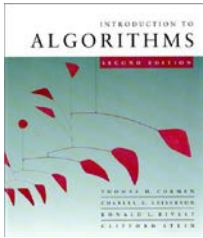
Quicksort an  $n$ -element array:

- 1. Divide:** Partition the array into two subarrays around a **pivot**  $x$  such that elements in lower subarray  $\leq x \leq$  elements in upper subarray.



- 2. Conquer:** Recursively sort the two subarrays.
- 3. Combine:** Trivial.

**Key:** *Linear-time partitioning subroutine.*

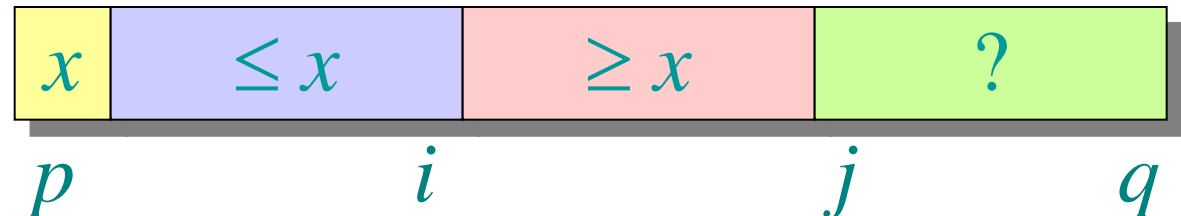


# Partitioning subroutine

```
PARTITION( $A, p, q$ )  $\triangleright A[p \dots q]$   
   $x \leftarrow A[p]$   $\triangleright \text{pivot} = A[p]$   
   $i \leftarrow p$   
  for  $j \leftarrow p + 1$  to  $q$   
    do if  $A[j] \leq x$   
      then  $i \leftarrow i + 1$   
           exchange  $A[i] \leftrightarrow A[j]$   
  exchange  $A[p] \leftrightarrow A[i]$   
  return  $i$ 
```

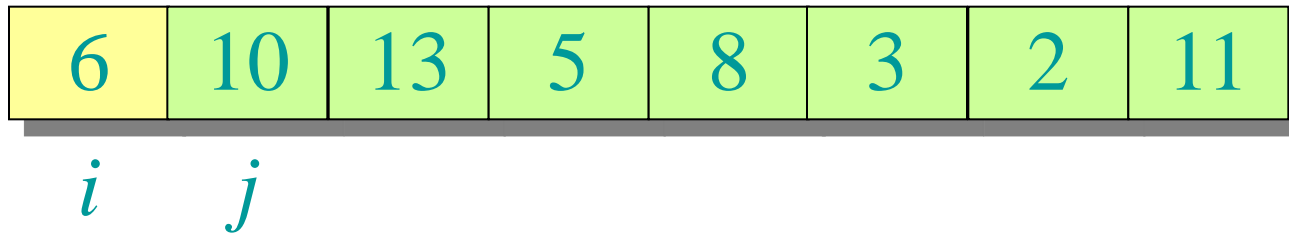
Running time  
=  $O(n)$  for  $n$   
elements.

***Invariant:***



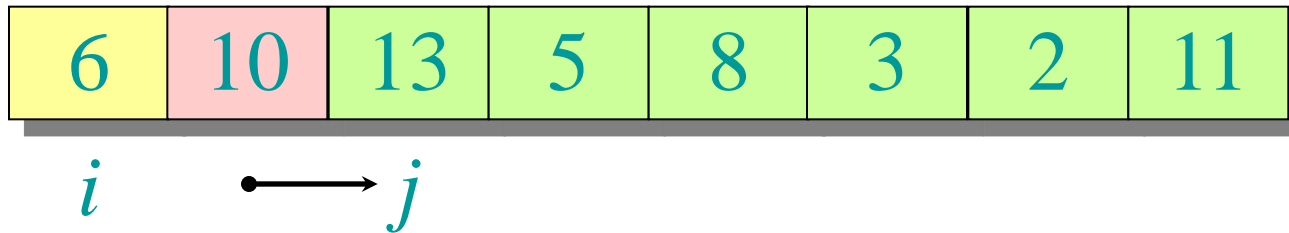


# Example of partitioning



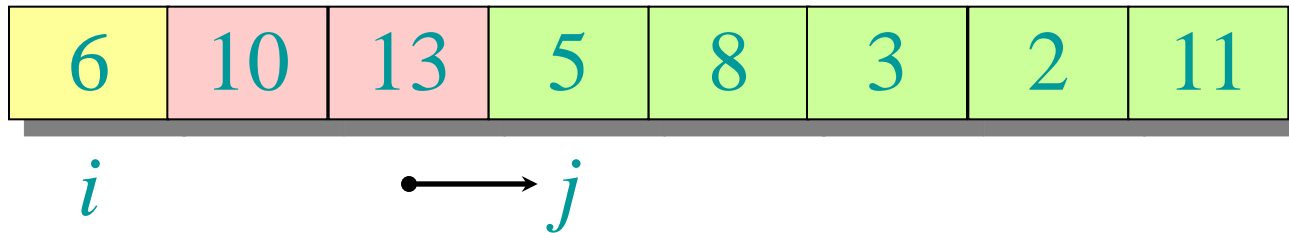


# Example of partitioning



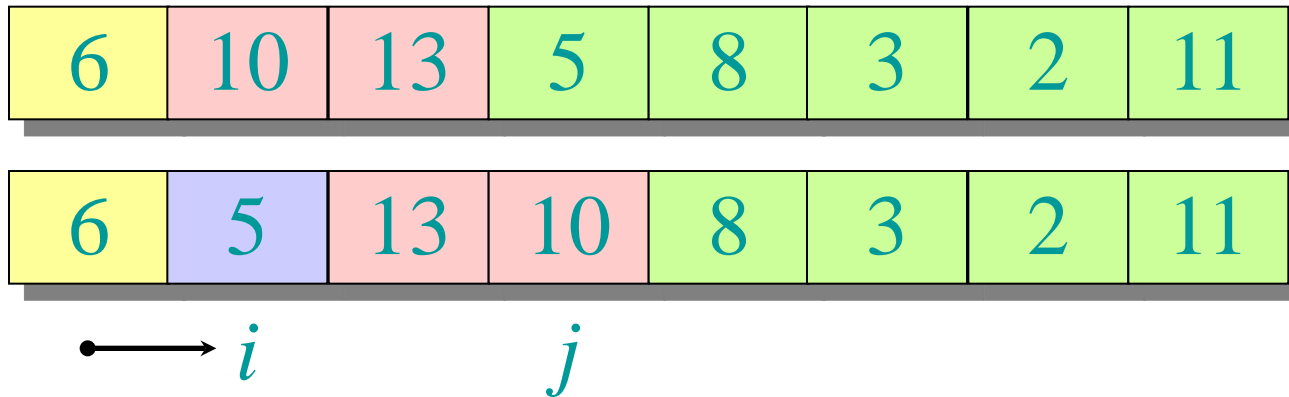


# Example of partitioning



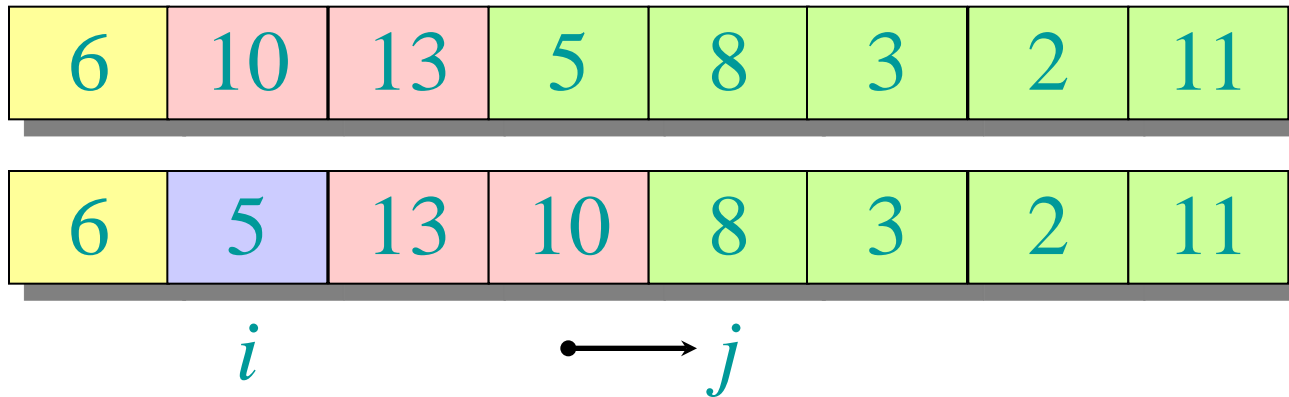


# Example of partitioning





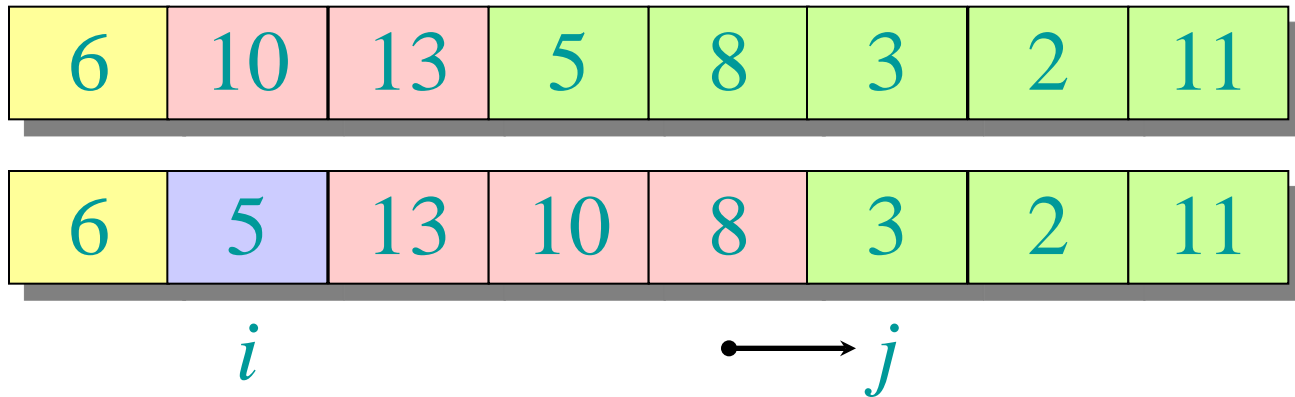
# Example of partitioning

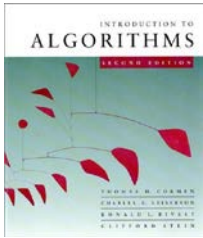




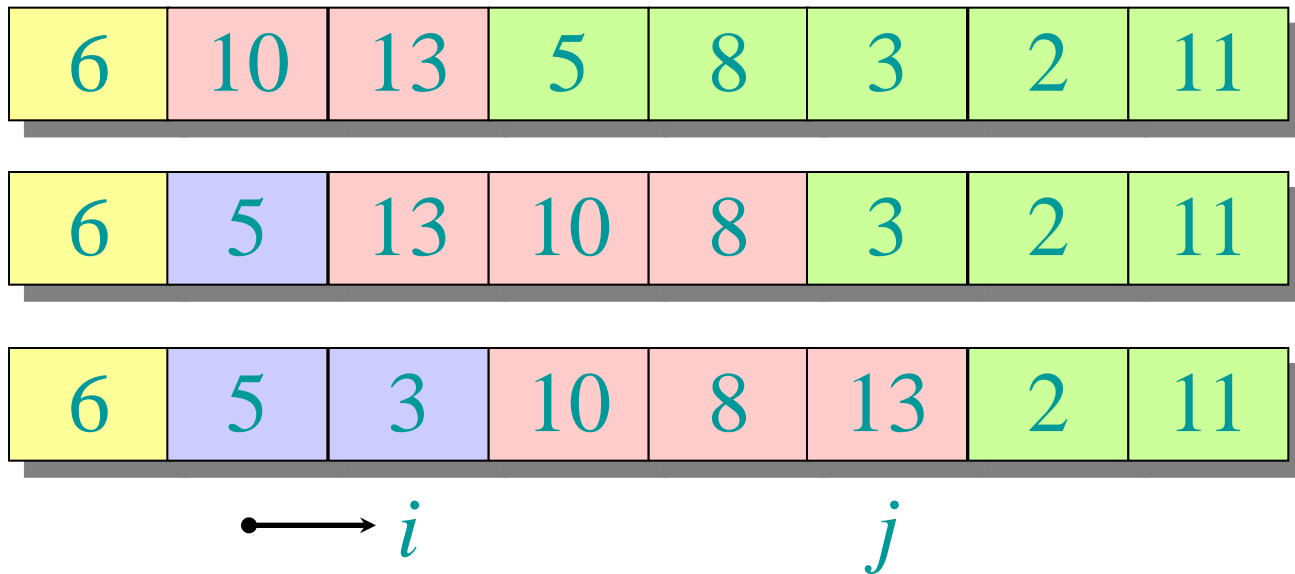


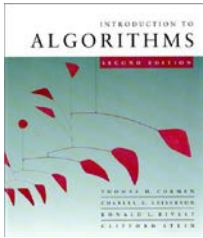
# Example of partitioning



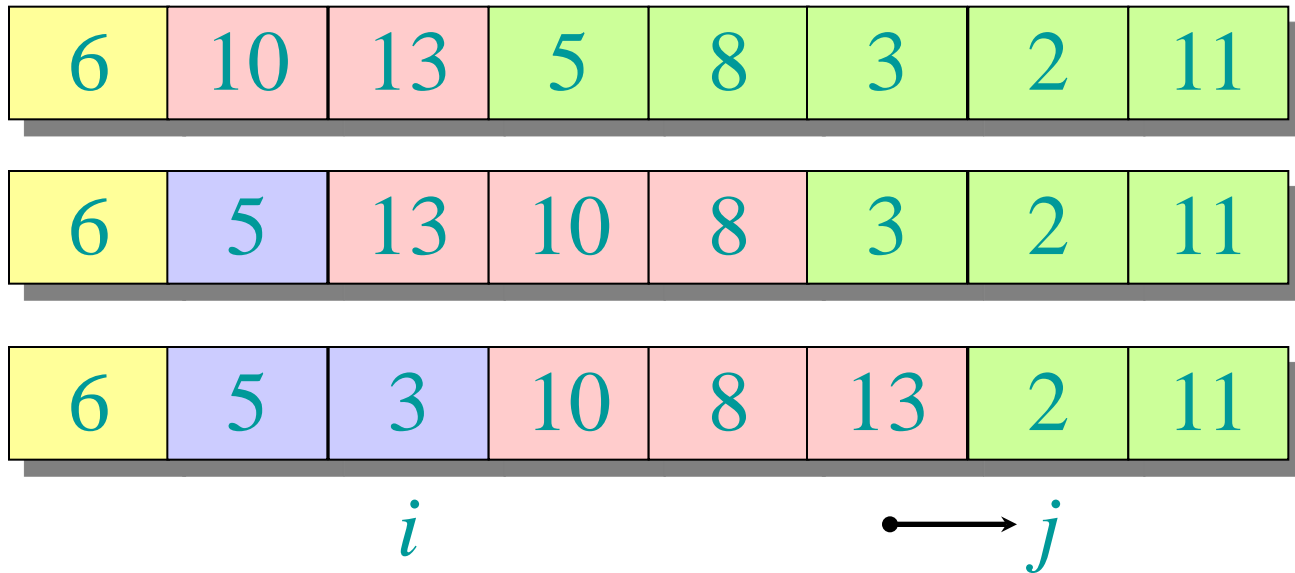


# Example of partitioning



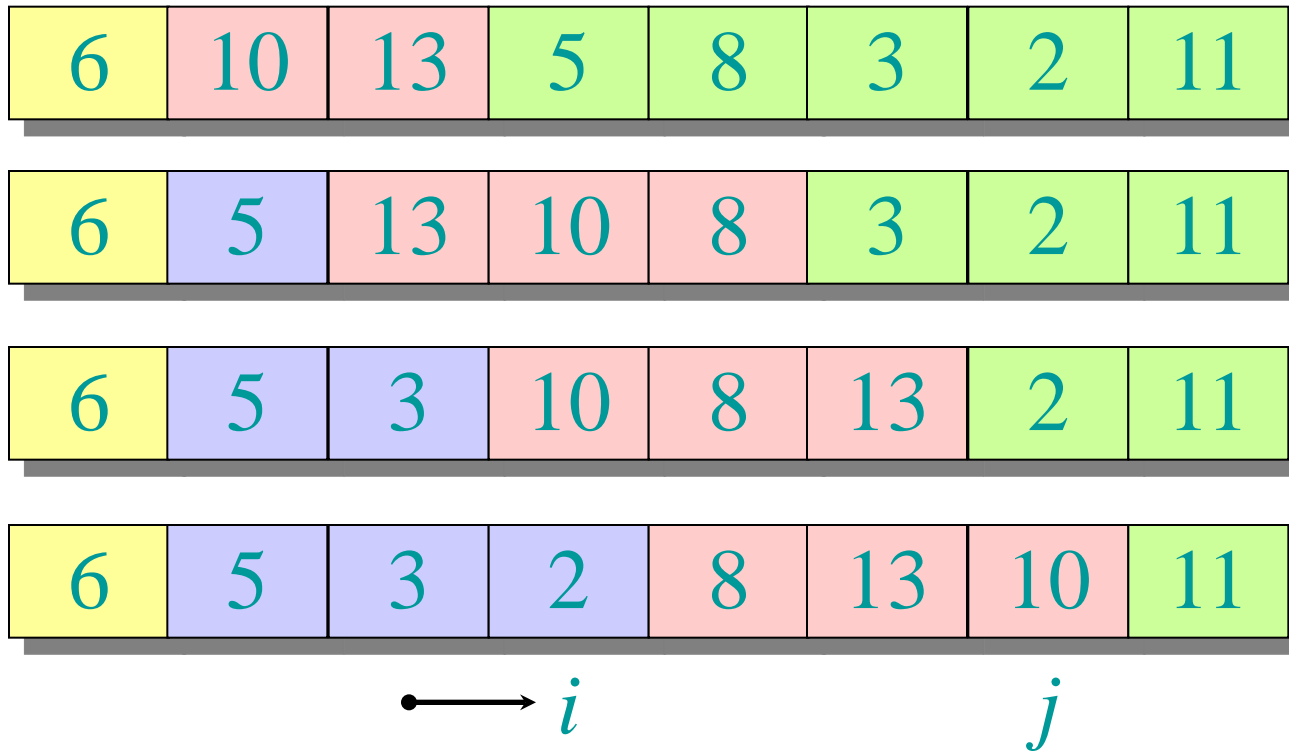


# Example of partitioning



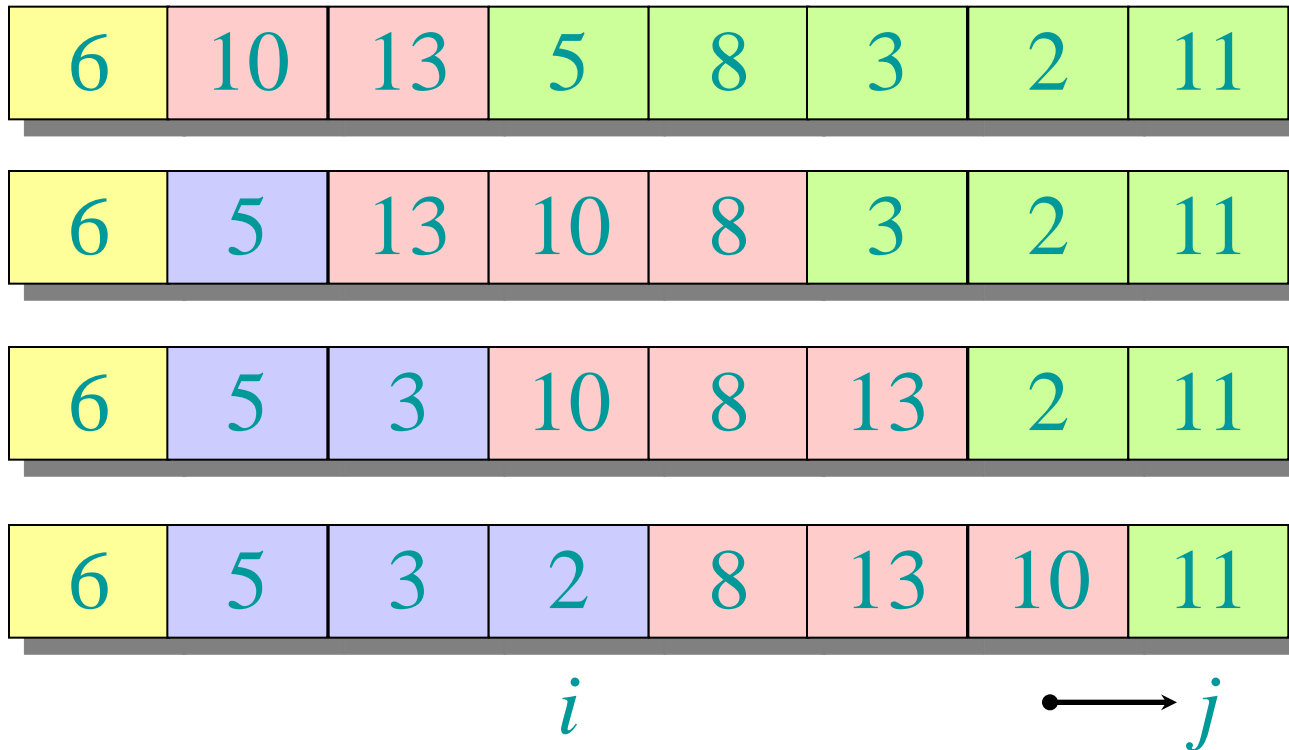


# Example of partitioning



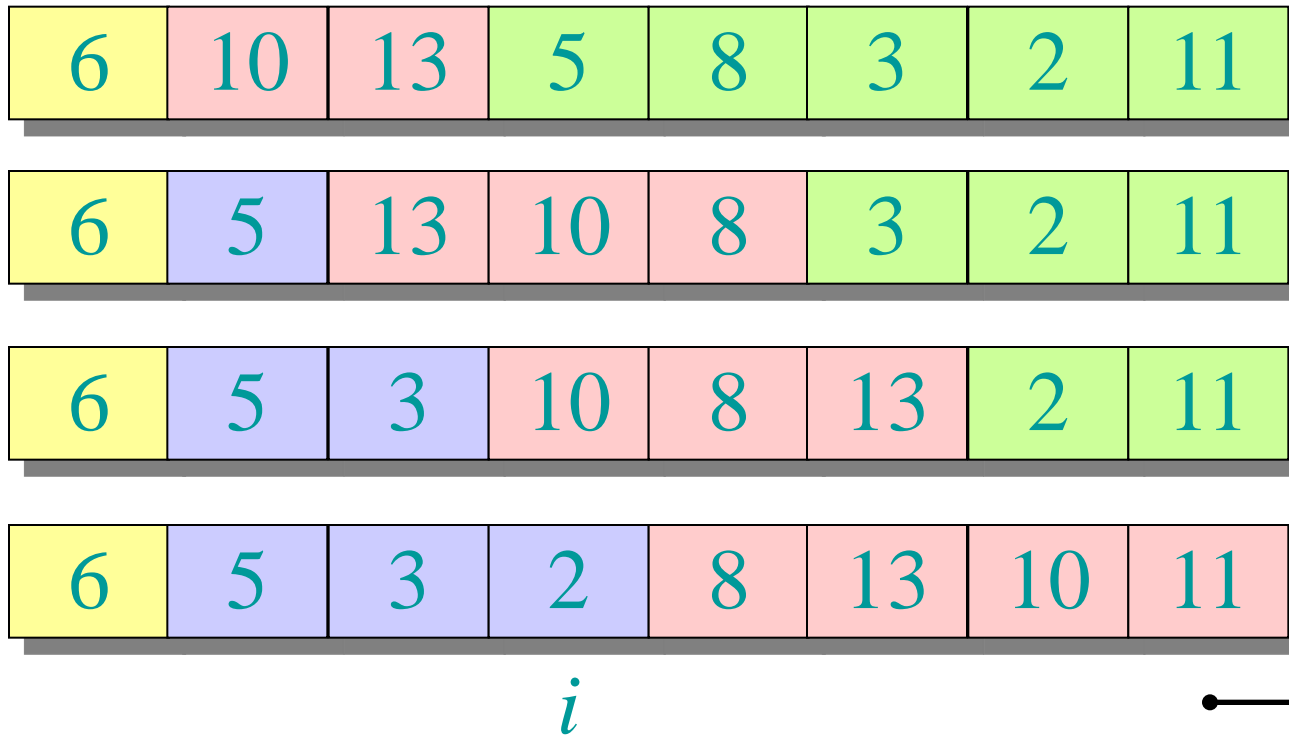


# Example of partitioning



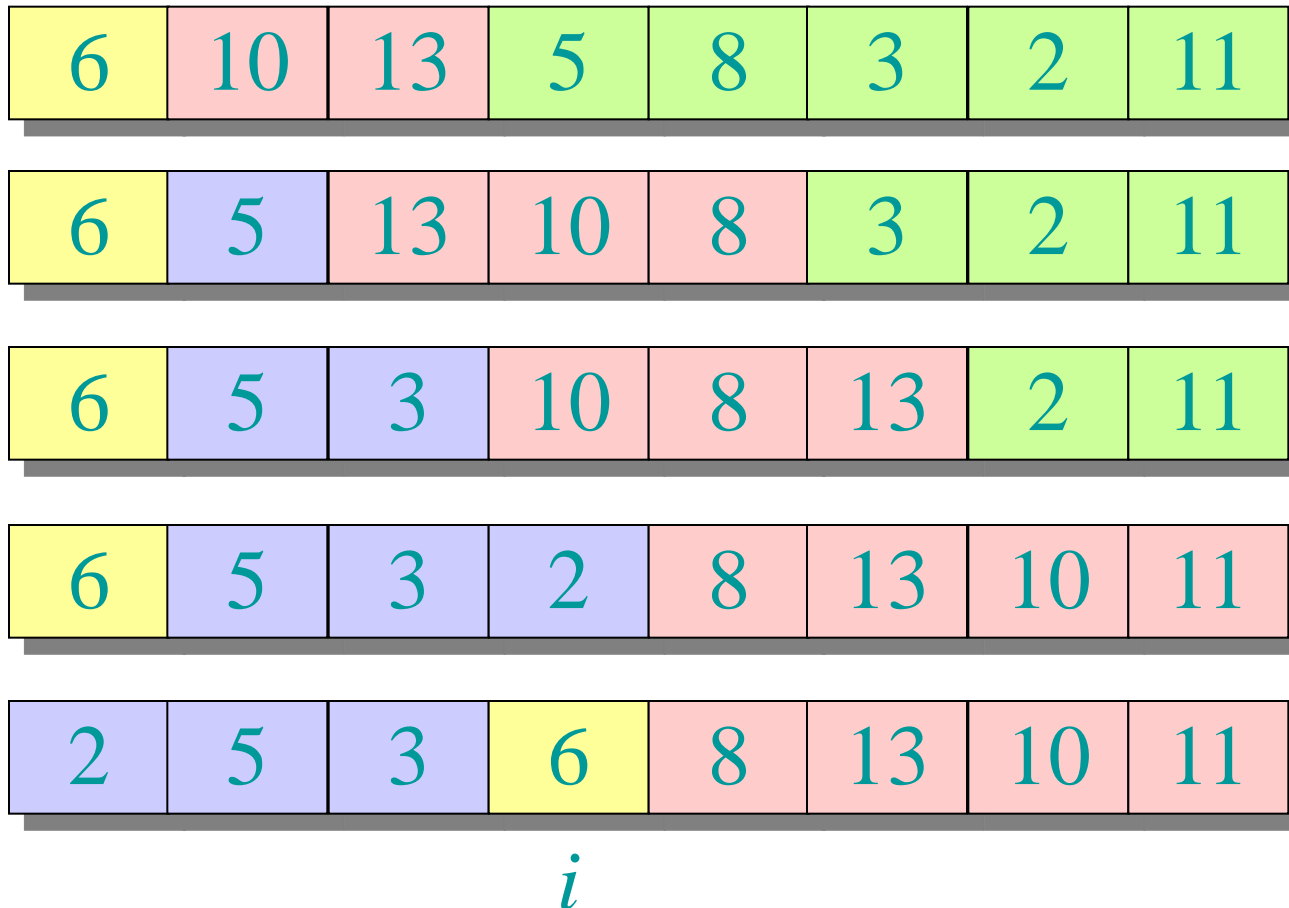


# Example of partitioning





# Example of partitioning





# Pseudocode for quicksort

QUICKSORT( $A, p, r$ )

**if**  $p < r$

**then**  $q \leftarrow \text{PARTITION}(A, p, r)$

      QUICKSORT( $A, p, q-1$ )

      QUICKSORT( $A, q+1, r$ )

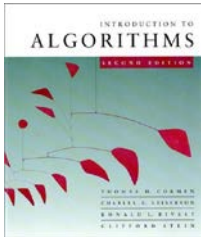
**Initial call:** QUICKSORT( $A, 1, n$ )





# Analysis of quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let  $T(n)$  = worst-case running time on an array of  $n$  elements.



# Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

$$= \Theta(1) + T(n-1) + \Theta(n)$$

$$= T(n-1) + \Theta(n)$$

$$= \Theta(n^2) \quad (\textit{arithmetic series})$$



# Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



# Worst-case recursion tree

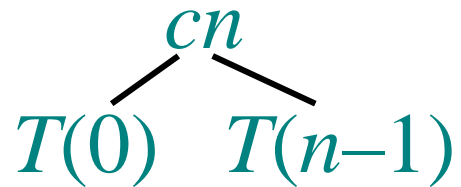
$$T(n) = T(0) + T(n-1) + cn$$

$$T(n)$$



# Worst-case recursion tree

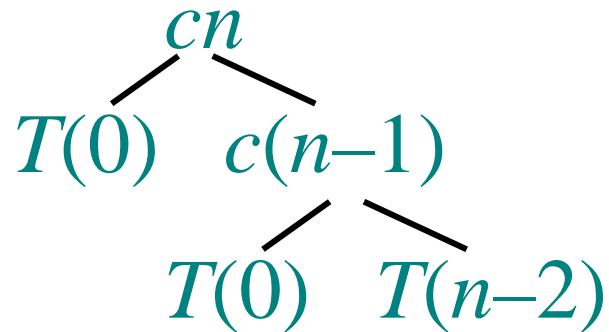
$$T(n) = T(0) + T(n-1) + cn$$





# Worst-case recursion tree

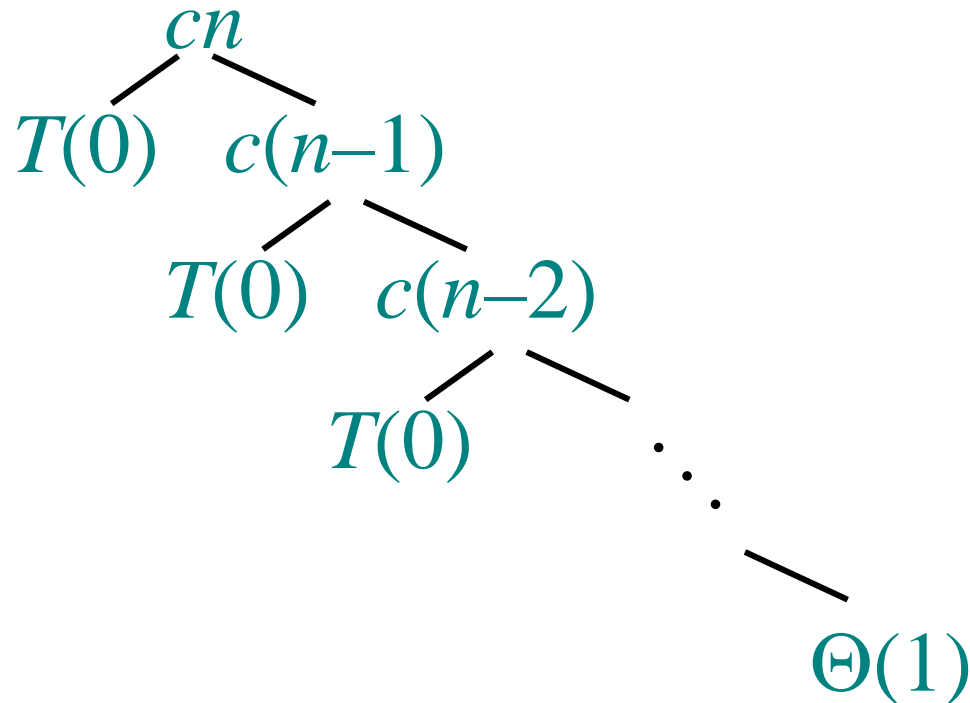
$$T(n) = T(0) + T(n-1) + cn$$





# Worst-case recursion tree

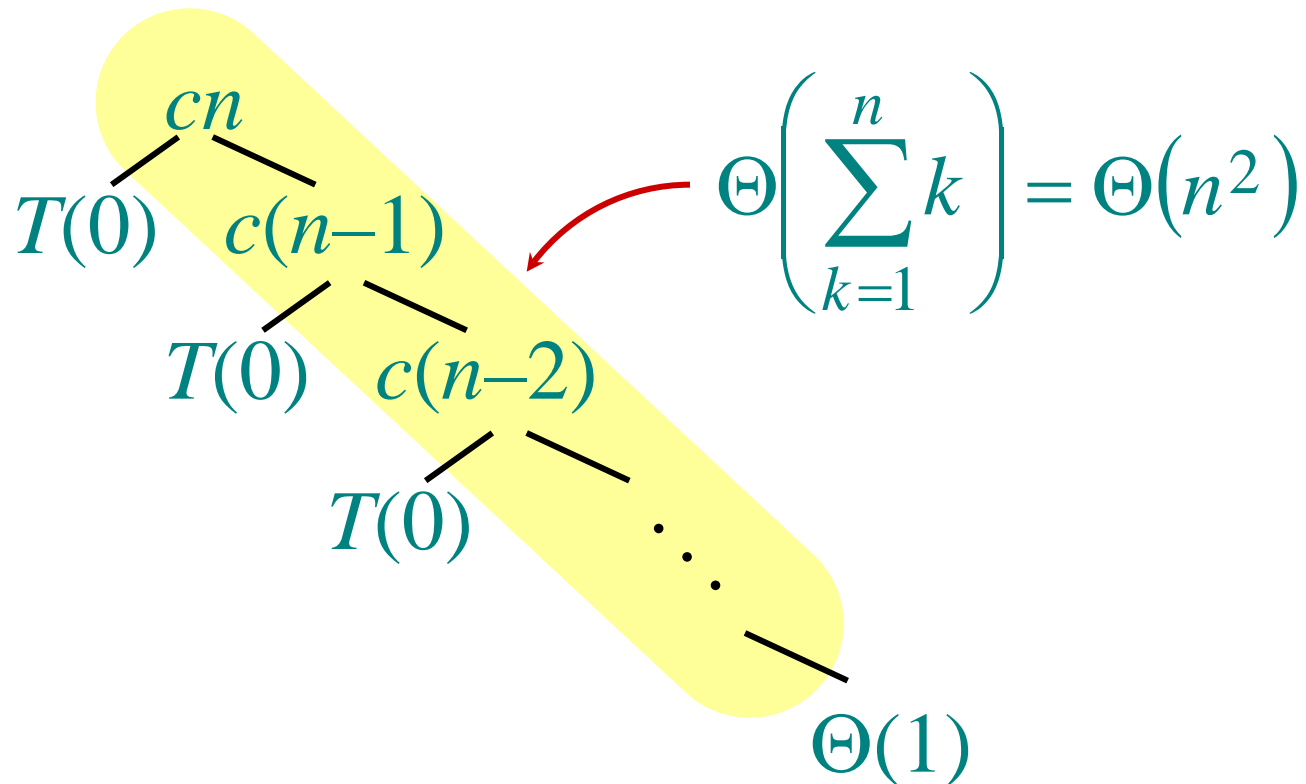
$$T(n) = T(0) + T(n-1) + cn$$



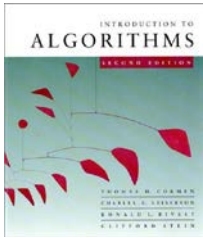


# Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$

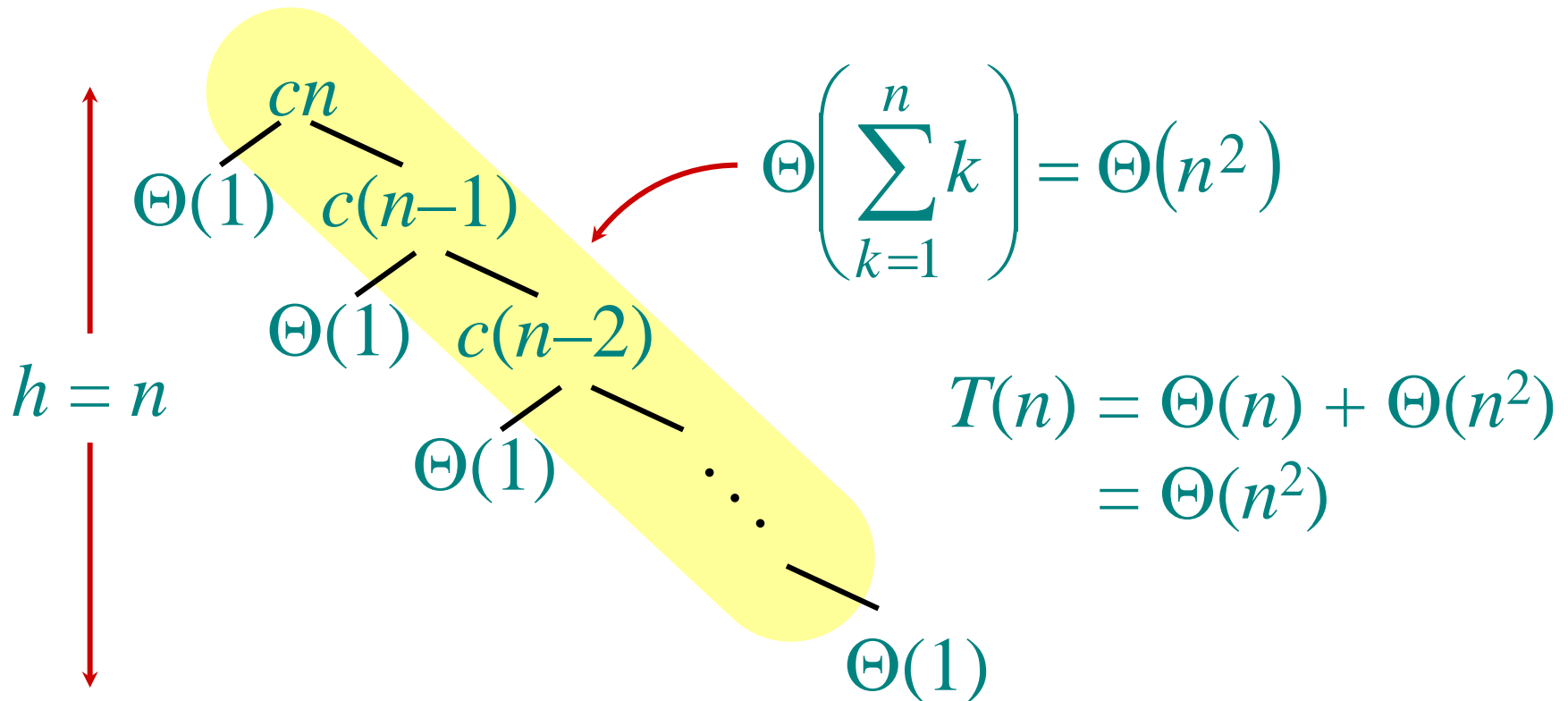






# Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$





# Best-case analysis

*(For intuition only!)*

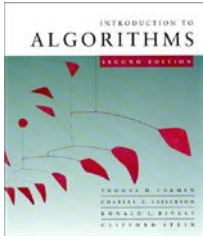
If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \quad (\text{same as merge sort}) \end{aligned}$$

What if the split is always  $\frac{1}{10} : \frac{9}{10}$ ?

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?

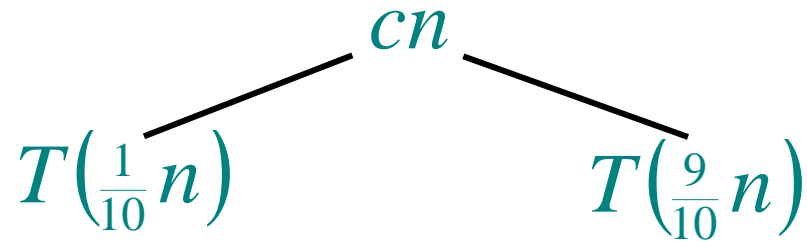


# Analysis of “almost-best” case

$$T(n)$$

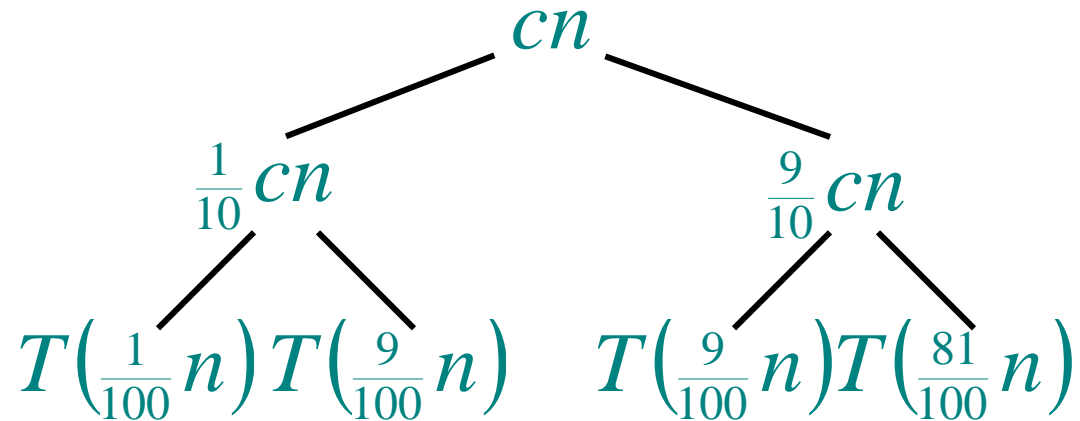


# Analysis of “almost-best” case



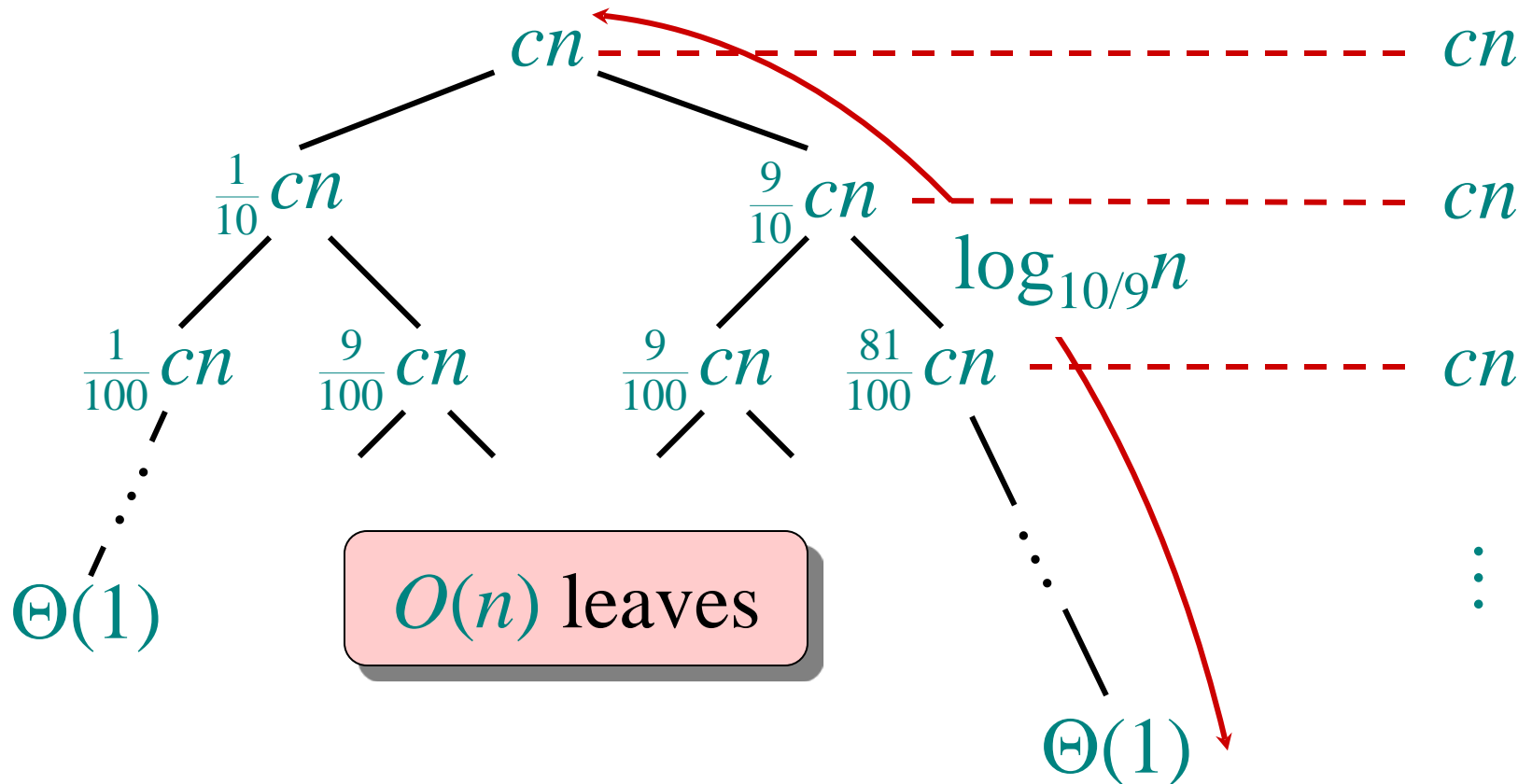


# Analysis of “almost-best” case



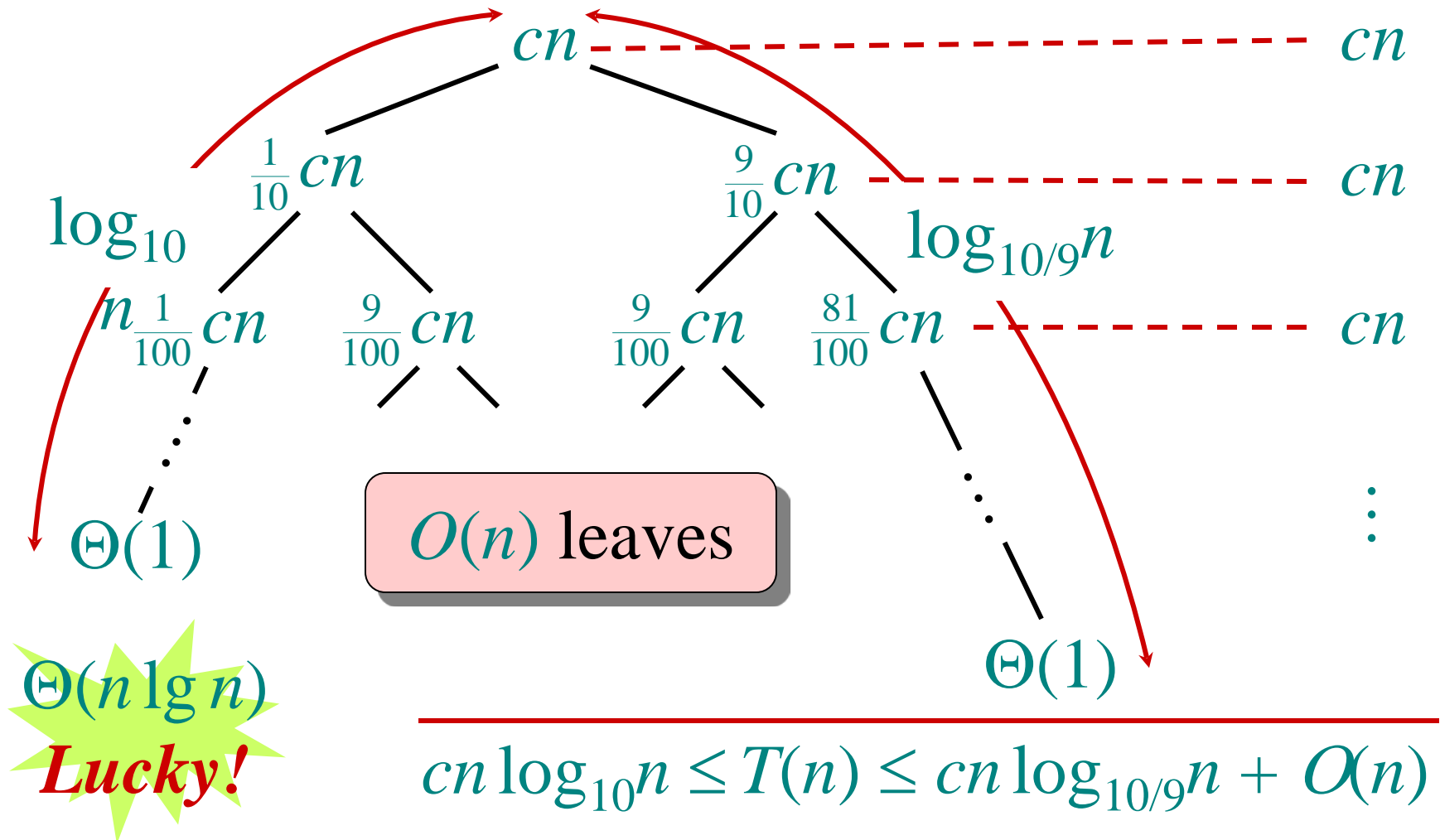


# Analysis of “almost-best” case





# Analysis of “almost-best” case





# More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ....

$$L(n) = 2U(n/2) + \Theta(n) \quad \textit{lucky}$$

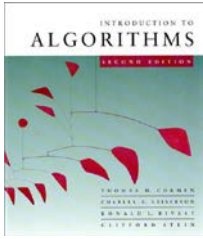
$$U(n) = L(n-1) + \Theta(n) \quad \textit{unlucky}$$

Solving:

$$\begin{aligned} L(n) &= 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2 - 1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned} \quad \textit{Lucky!}$$

How can we make sure we are usually lucky?

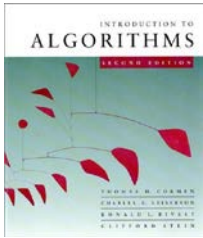




# Sorting in linear time

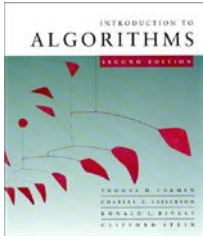
**Counting sort:** No comparisons between elements.

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

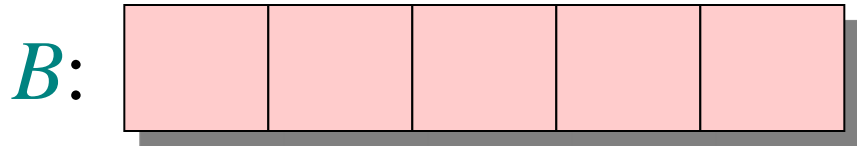
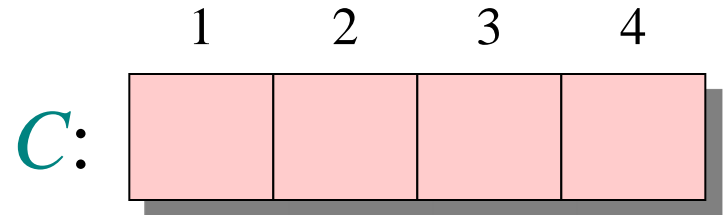
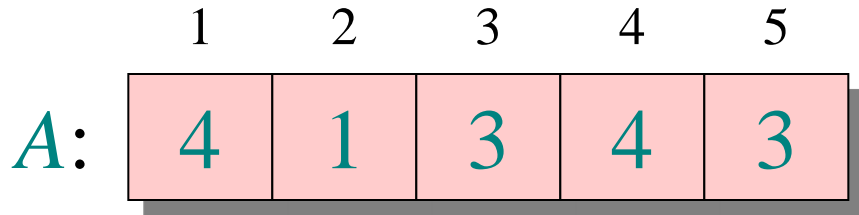


# Counting sort

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



# Counting-sort example





# Loop 1

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

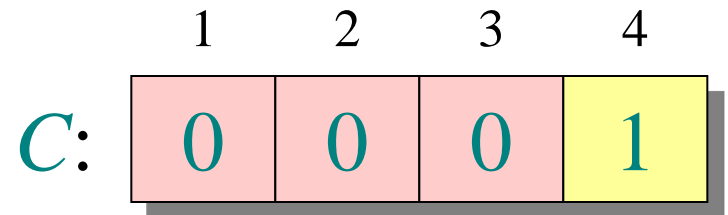
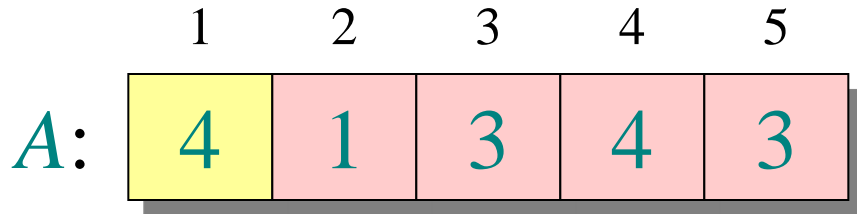
	1	2	3	4
<i>C</i> :	0	0	0	0

<i>B</i> :					
------------	--	--	--	--	--

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



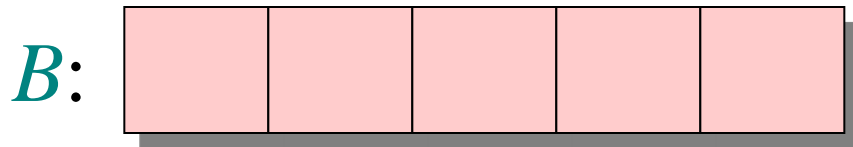
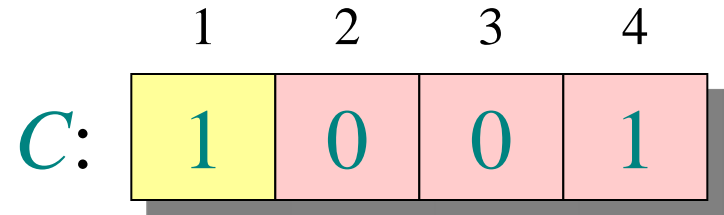
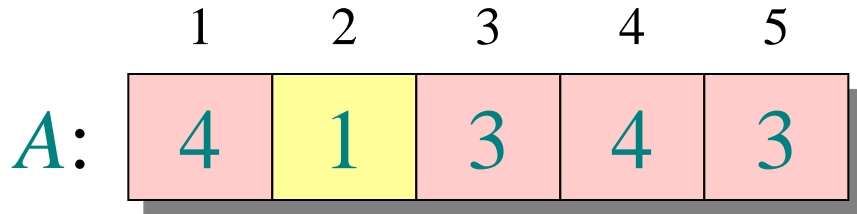
# Loop 2



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



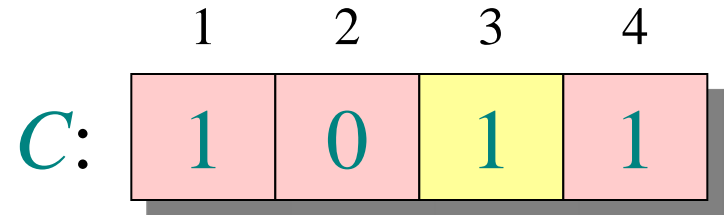
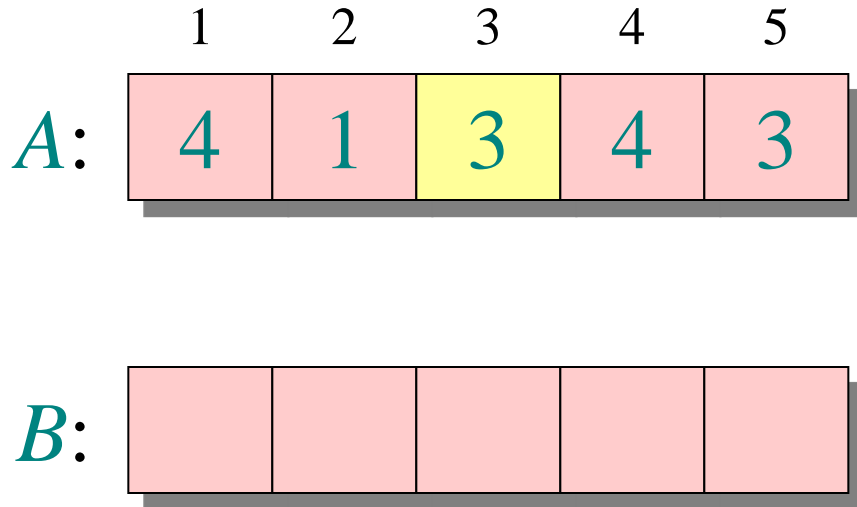
# Loop 2



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



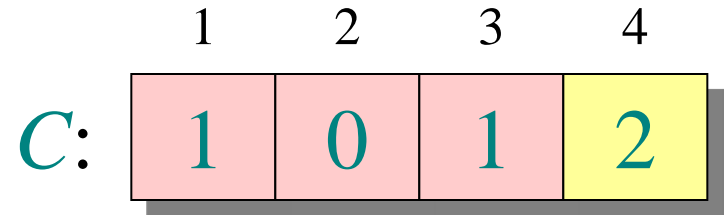
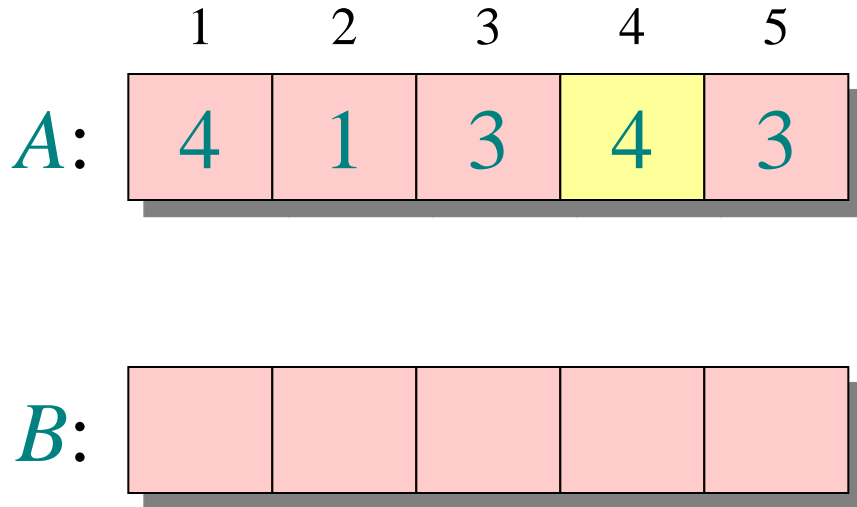
# Loop 2



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



# Loop 2

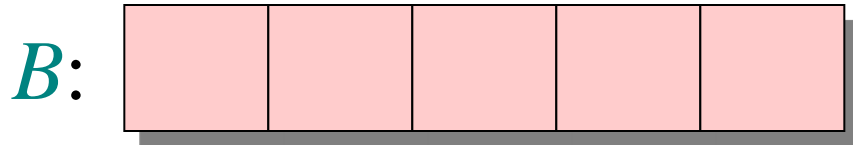
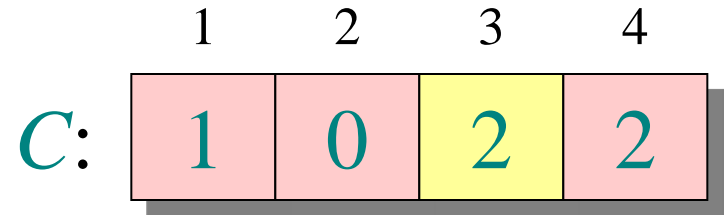
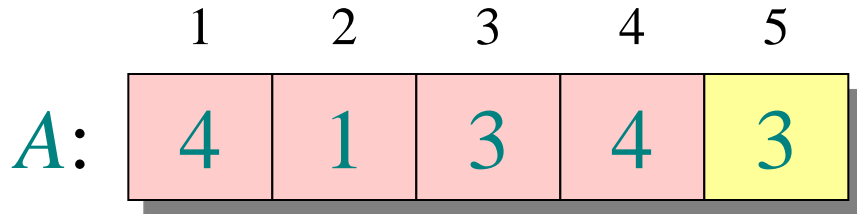


**for**  $j \leftarrow 1$  **to**  $n$   
    **do**  $C[A[j]] \leftarrow C[A[j]] + 1$      $\triangleright C[i] = |\{\text{key} = i\}|$





# Loop 2



**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
<i>A</i> :	4	1	3	4	3

<i>B</i> :					
------------	--	--	--	--	--

	1	2	3	4
<i>C</i> :	1	0	2	2

<i>C'</i> :	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
<i>A</i> :	4	1	3	4	3

<i>B</i> :					
------------	--	--	--	--	--

	1	2	3	4
<i>C</i> :	1	0	2	2

<i>C'</i> :	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

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

<i>B</i> :					
------------	--	--	--	--	--

	1	2	3	4
<i>C</i> :	1	0	2	2

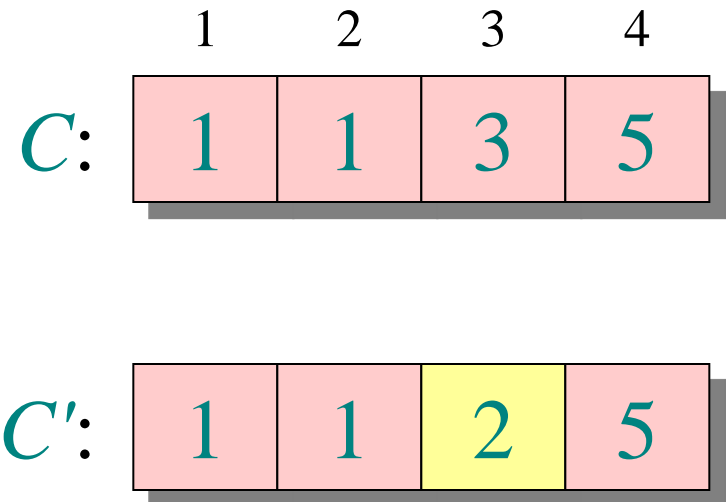
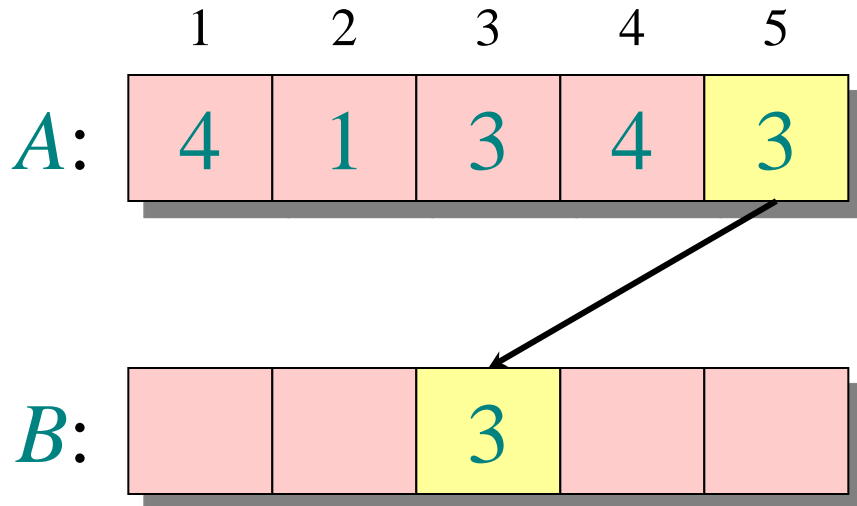
<i>C'</i> :	1	1	3	5
-------------	---	---	---	---

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

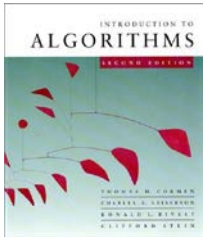
**do**  $C[i] \leftarrow C[i] + C[i-1]$        $\triangleright C[i] = |\{\text{key} \leq i\}|$



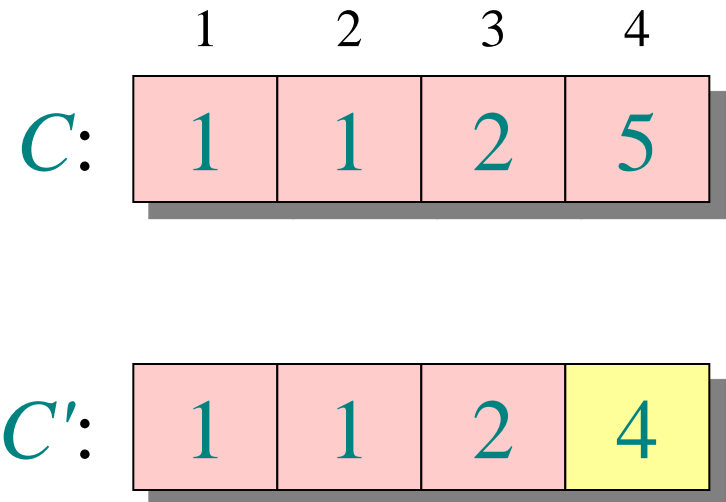
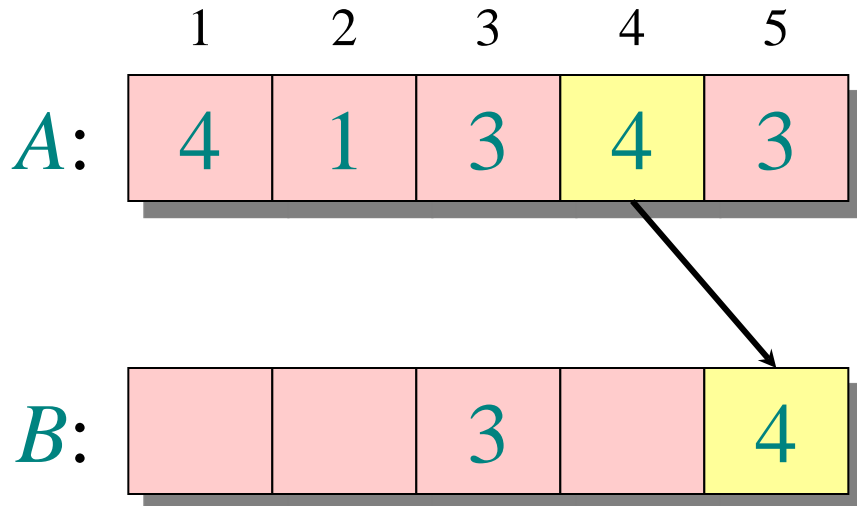
# Loop 4



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



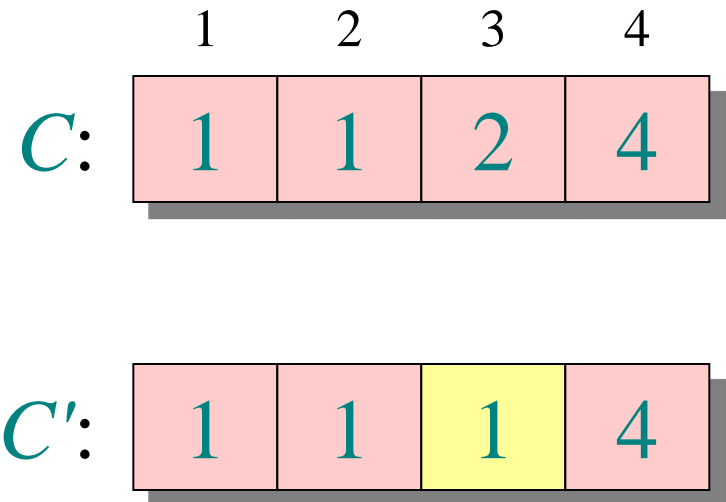
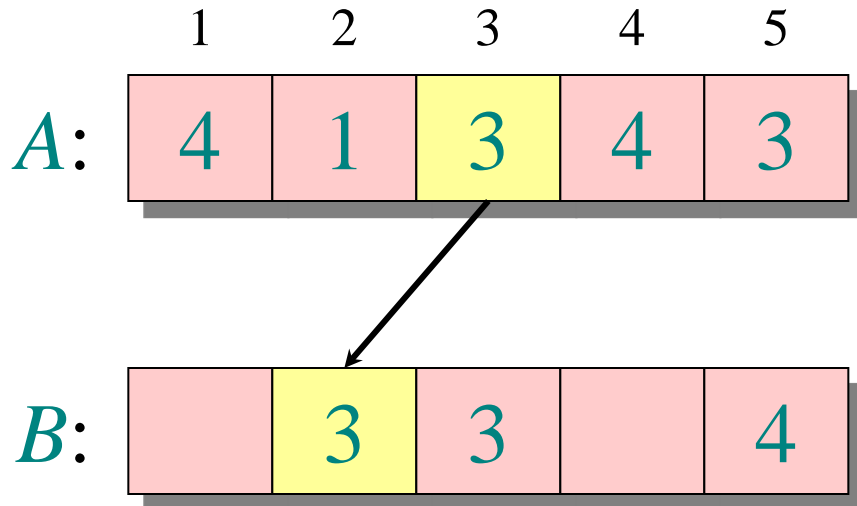
# Loop 4



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



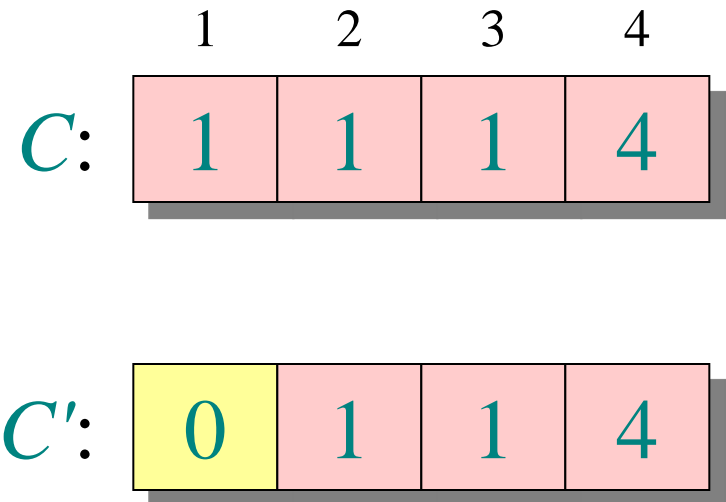
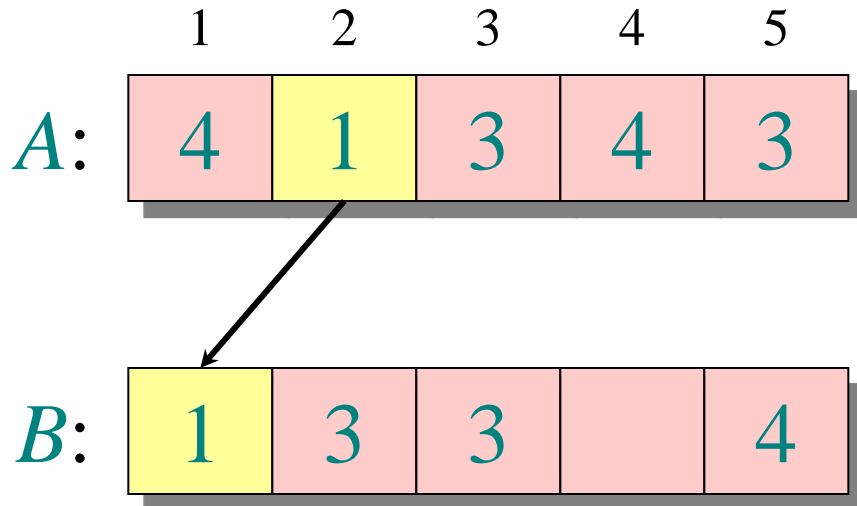
# Loop 4



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



# Loop 4

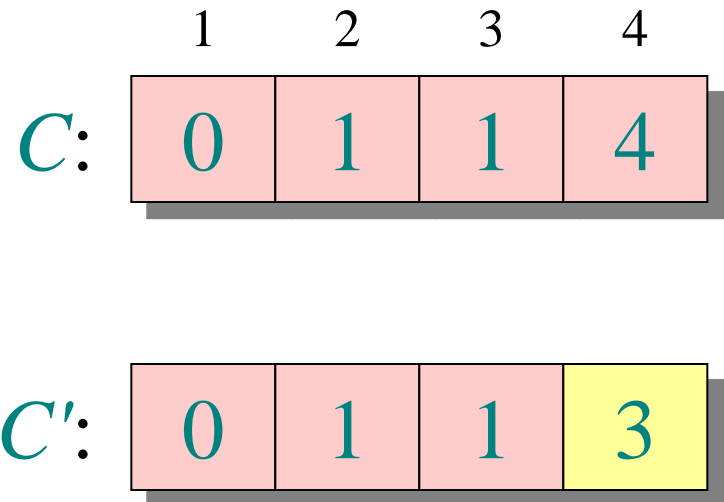
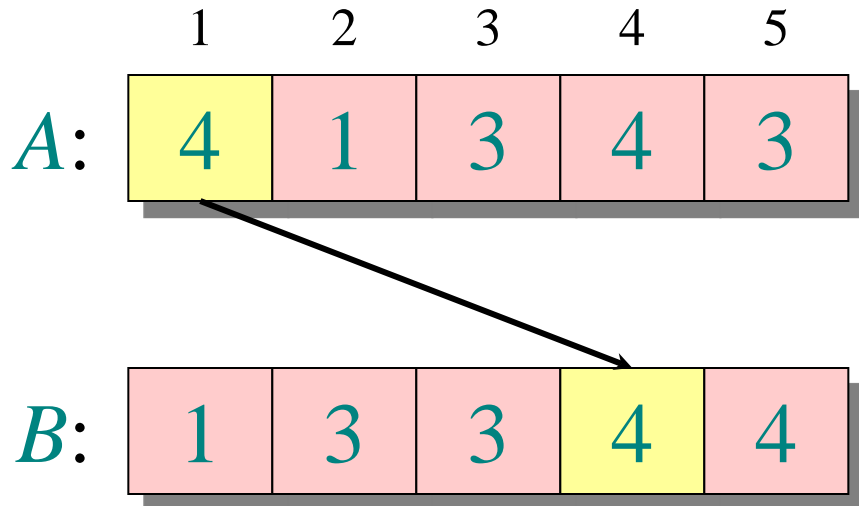


```
for  $j \leftarrow n$  downto 1  
  do  $B[C[A[j]]] \leftarrow A[j]$   
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

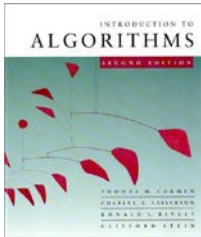




# Loop 4



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



# Analysis

$$\begin{array}{lcl} \Theta(k) & \left\{ \begin{array}{l} \text{for } i \leftarrow 1 \text{ to } k \\ \quad \text{do } C[i] \leftarrow 0 \end{array} \right. & \\ \Theta(n) & \left\{ \begin{array}{l} \text{for } j \leftarrow 1 \text{ to } n \\ \quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1 \end{array} \right. & \\ \Theta(k) & \left\{ \begin{array}{l} \text{for } i \leftarrow 2 \text{ to } k \\ \quad \text{do } C[i] \leftarrow C[i] + C[i-1] \end{array} \right. & \\ \Theta(n) & \left\{ \begin{array}{l} \text{for } j \leftarrow n \text{ downto } 1 \\ \quad \text{do } B[C[A[j]]] \leftarrow A[j] \\ \quad \quad C[A[j]] \leftarrow C[A[j]] - 1 \end{array} \right. & \\ \hline & \Theta(n + k) & \end{array}$$



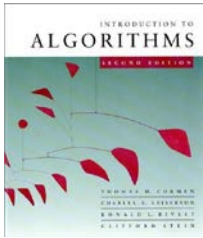
# Running time

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

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

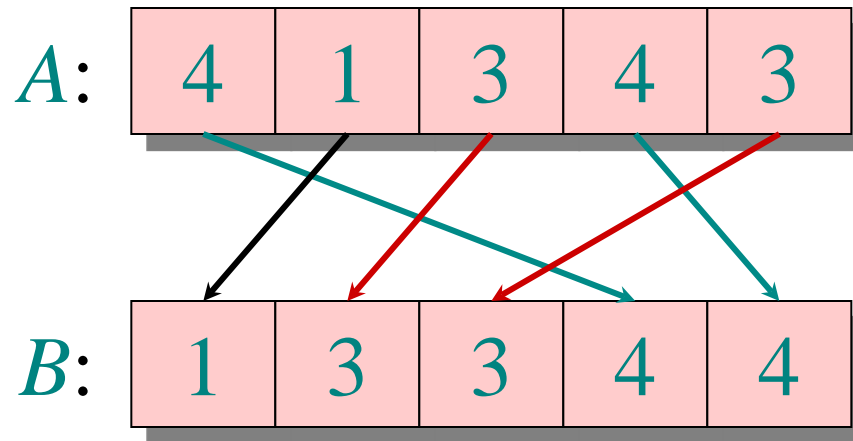
## Answer:

- *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.

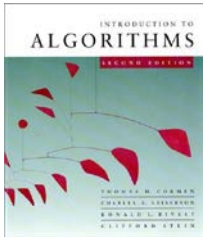


**Exercise:** What other sorts have this property?

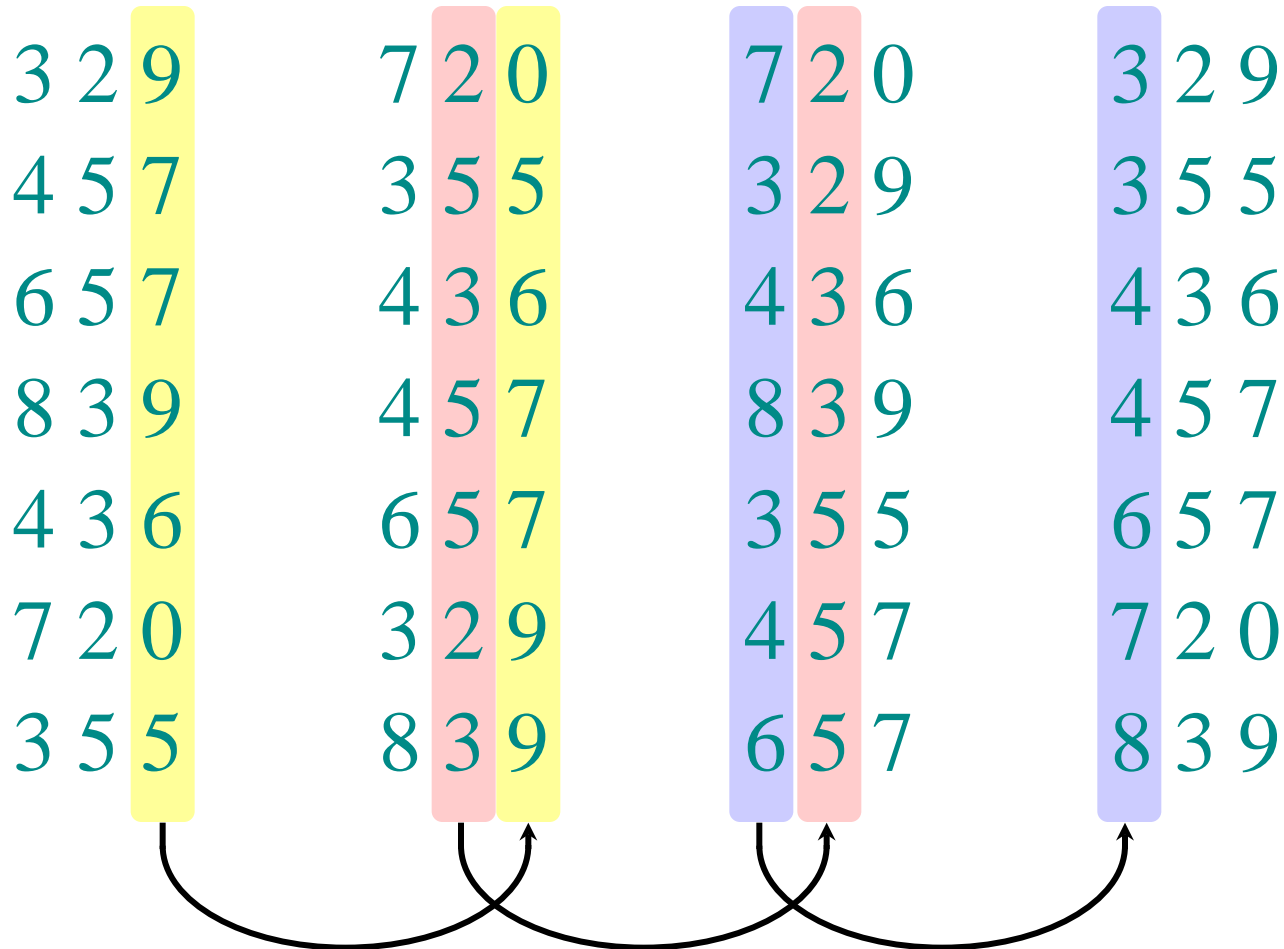


# Radix sort

- *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.



# Operation of radix sort

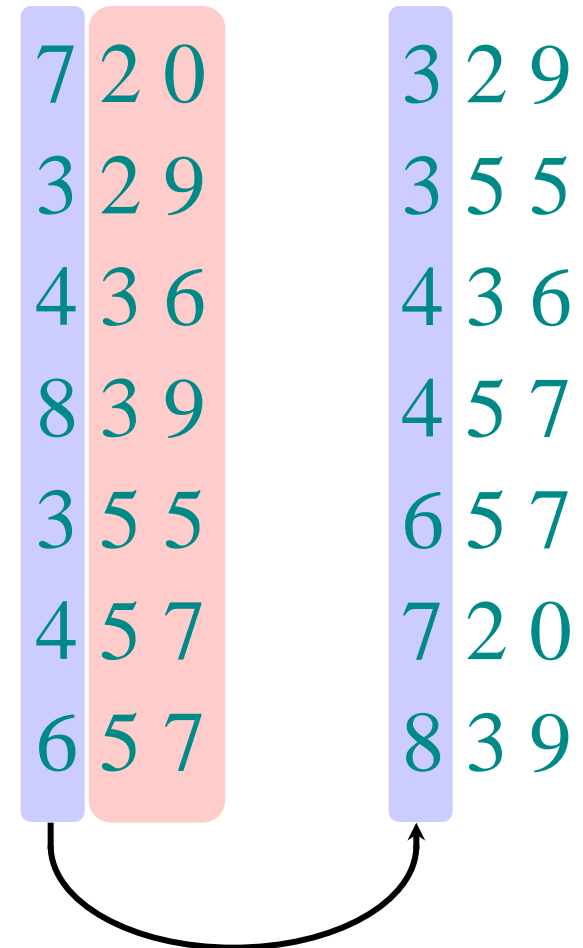




# Correctness of radix sort

*Induction on digit position*

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

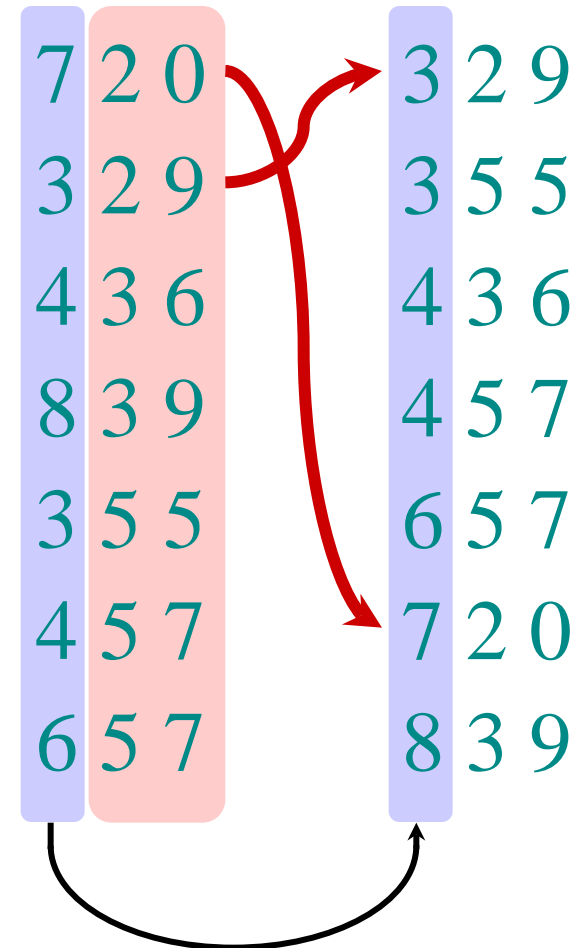




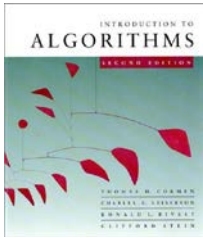
# Correctness of radix sort

*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.



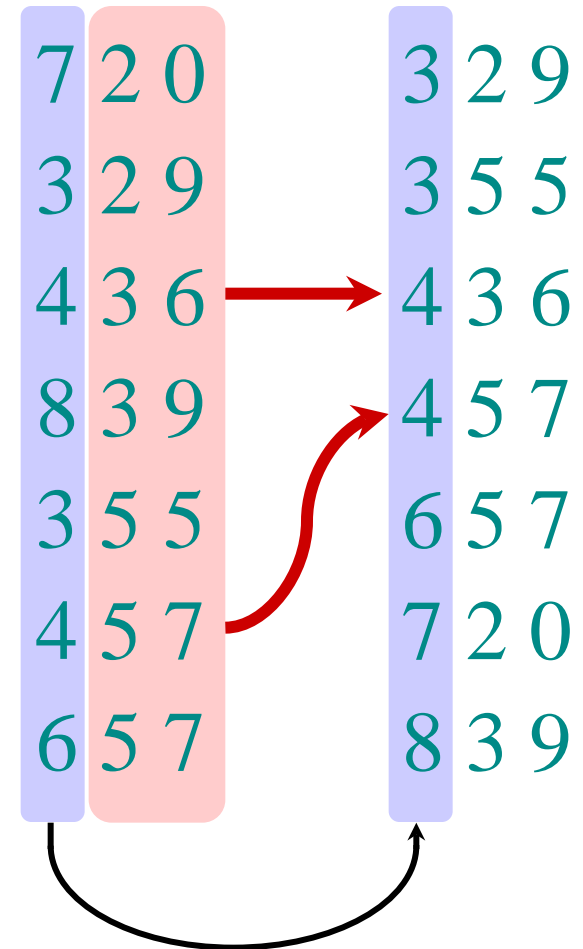


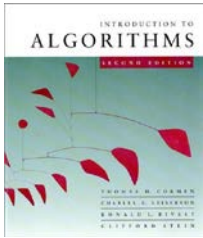


# Correctness of radix sort

*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

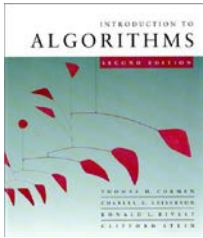
- Assume counting sort is the auxiliary stable sort.
- Sort  $n$  computer words of  $b$  bits each.
- Each word can be viewed as having  $b/r$  base- $2^r$  digits.

**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)

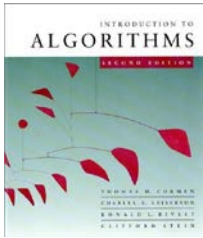
**Recall:** Counting sort takes  $\Theta(n + k)$  time to sort  $n$  numbers in the range from 0 to  $k - 1$ .

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

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

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  $\Theta(dn)$  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  $\geq 2000$  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.