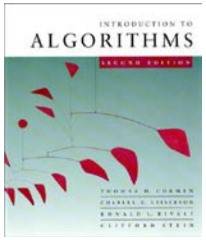


The problem of sorting

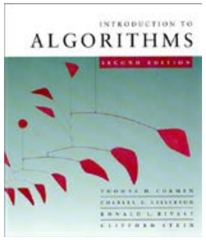
Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9



Insertion sort

"pseudocode"

INSERTION-SORT (A, n) $\rhd$ $A[1 \,.\,.\, n]$

 for $j \leftarrow 2$ **to** n

   do $key \leftarrow A[j]$

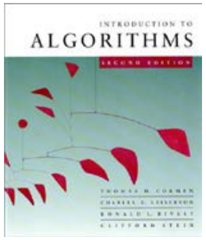
    $i \leftarrow j - 1$

    while $i > 0$ and $A[i] > key$

     do $A[i+1] \leftarrow A[i]$

      $i \leftarrow i - 1$

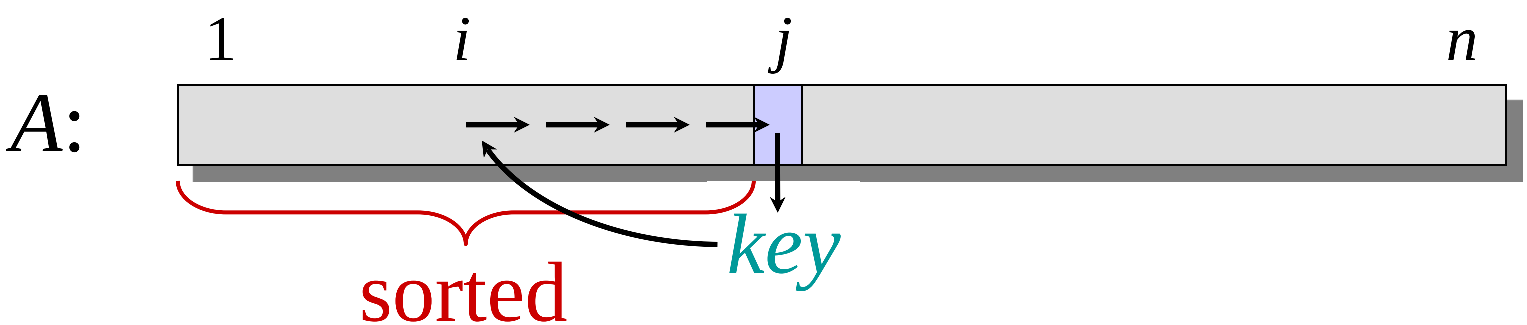
    $A[i+1] = key$

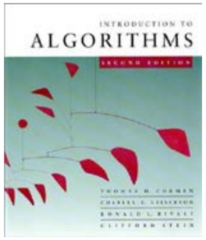


Insertion sort

"pseudocode"

```
INSERTION-SORT (A, n)          ⊳ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



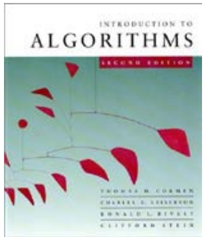


Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

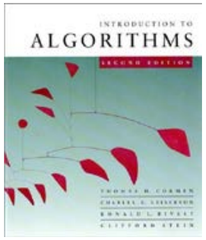
8 2 4 9 3 6



Example of insertion sort

8   2   4   9   3   6

2   8   4   9   3   6



Example of insertion sort

8 2 4 9 3 6

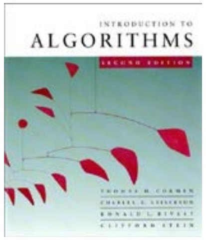
2 8 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

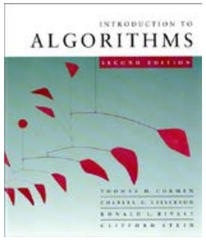


Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6



Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

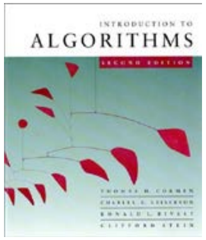
Example of insertion sort

8 **2** 4 9 3 6

2 8 **4** 9 3 6

2 4 8 **9** 3 6

2 4 8 9 **3** 6



Example of insertion sort

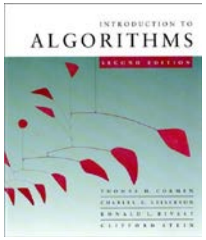
8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6



Example of insertion sort

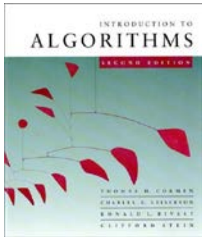
8 **2** 4 9 3 6

2 8 **4** 9 3 6

2 4 8 **9** 3 6

2 4 8 9 **3** 6

2 3 4 8 9 **6**



Example of insertion sort

8 2 4 9 3 6

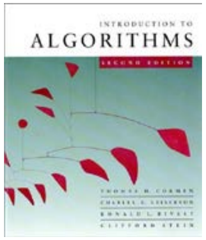
2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

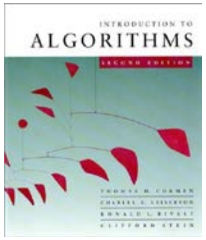
2 3 4 8 9 6

2 3 4 6 8 9 *done*



Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.



Kinds of analyses

Worst-case: (usually)

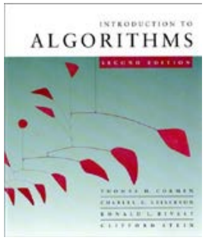
- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (bogus)

- Cheat with a slow algorithm that works fast on *some* input.



Insertion sort analysis

Worst case: Input reverse sorted.

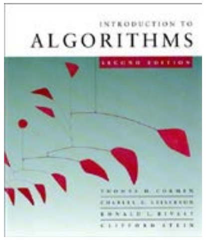
$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta\left(n^2\right) \qquad \text{[arithmetic series]}$$

Average case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta\left(n^2\right)$$

Is insertion sort a fast sorting algorithm?

- Moderately so, for small n.
- Not at all, for large n.

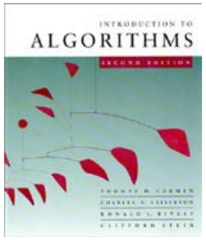


Merge sort

MERGE-SORT $A[1 \,.\,.\, n]$

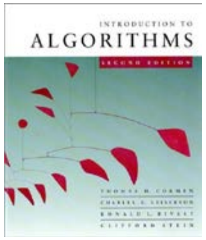
1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
3. "***Merge***" the 2 sorted lists.

Key subroutine: **MERGE**



Merging two sorted arrays

20	12
13	11
7	9
2	1



Merging two sorted arrays

20	12
13	11
7	9
2	1

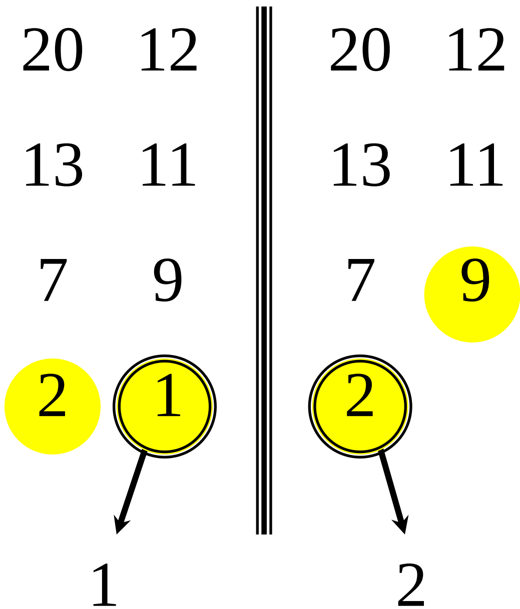
1

Merging two sorted arrays

20	12	20	12
13	11	13	11
7	9	7	9
2	1	2	

1

Merging two sorted arrays



Merging two sorted arrays

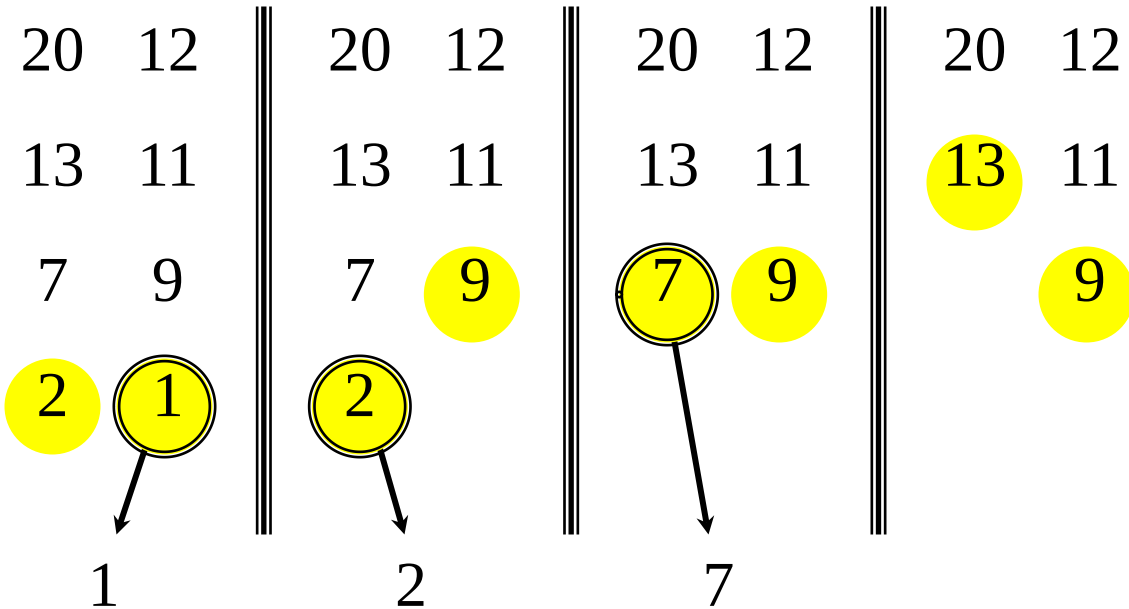
20	12	20	12	20	12
13	11	13	11	13	11
7	9	7	9	7	9
2	1	2			

1 2

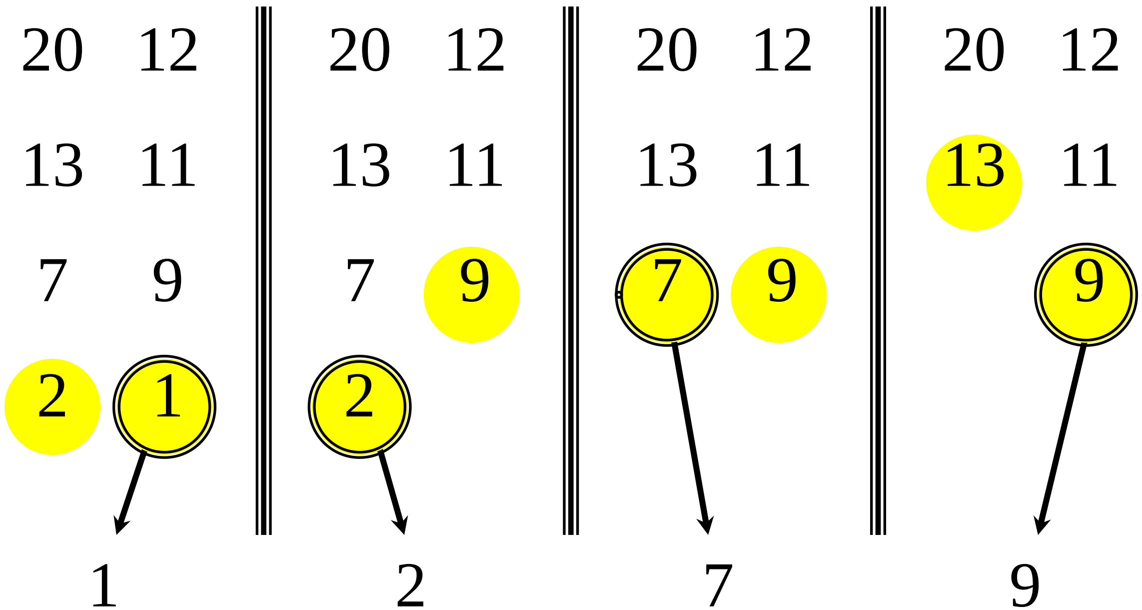
Merging two sorted arrays

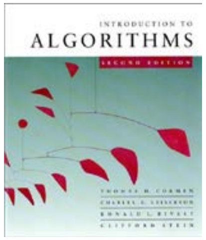
20	12	‖	20	12	‖	20	12
13	11	‖	13	11	‖	13	11
7	9	‖	7	9	‖	7	9
2	1	‖	2		‖		
	1			2			7

Merging two sorted arrays

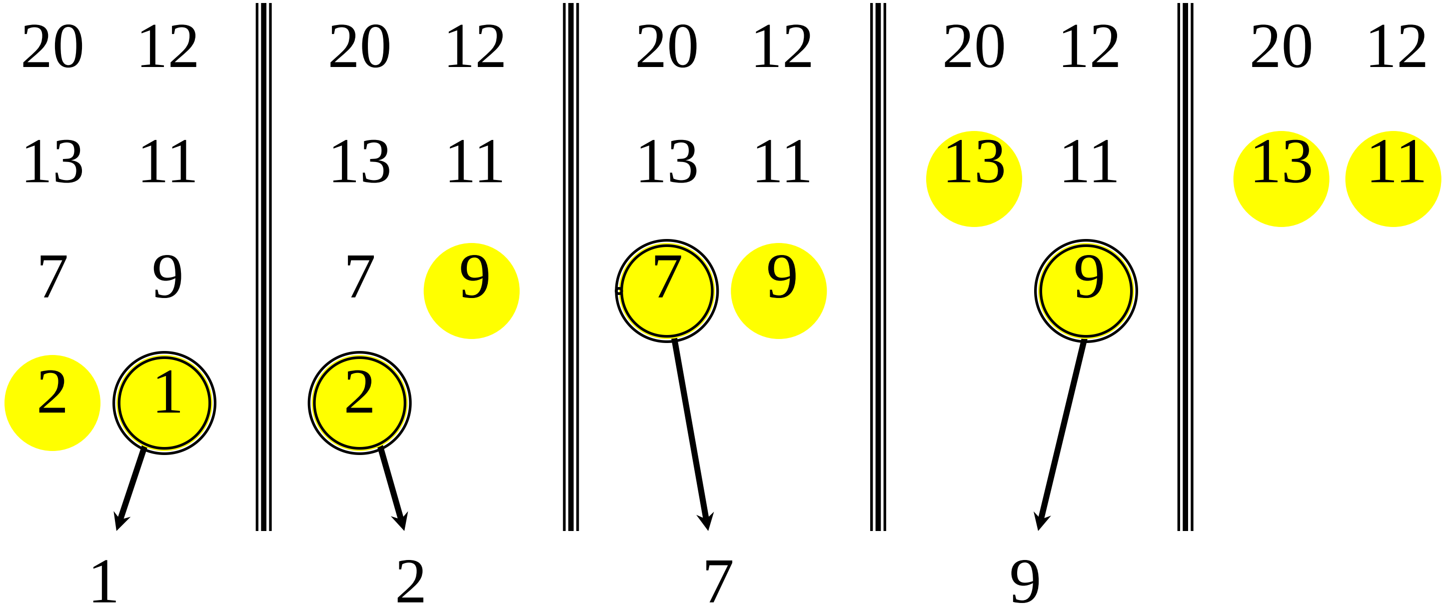


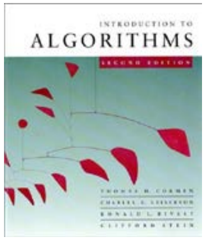
Merging two sorted arrays



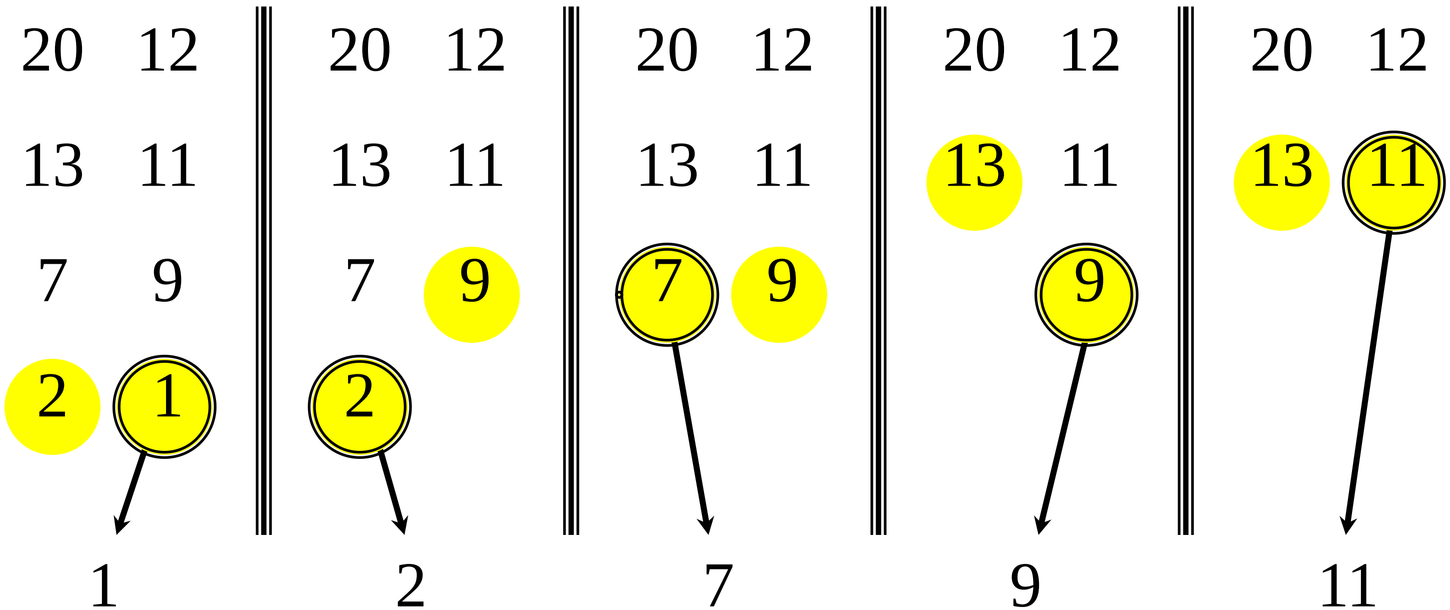


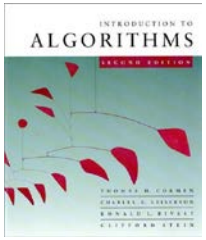
Merging two sorted arrays



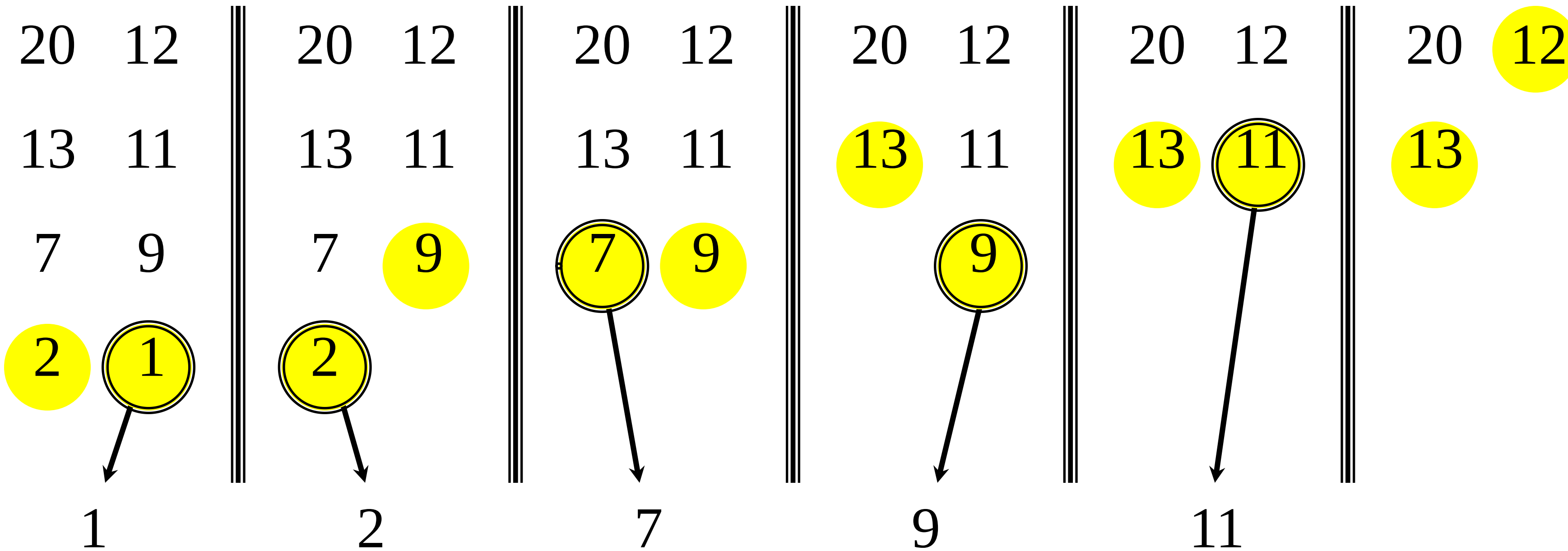


Merging two sorted arrays



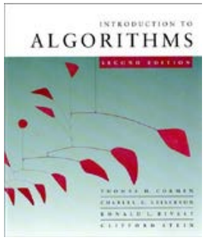


Merging two sorted arrays

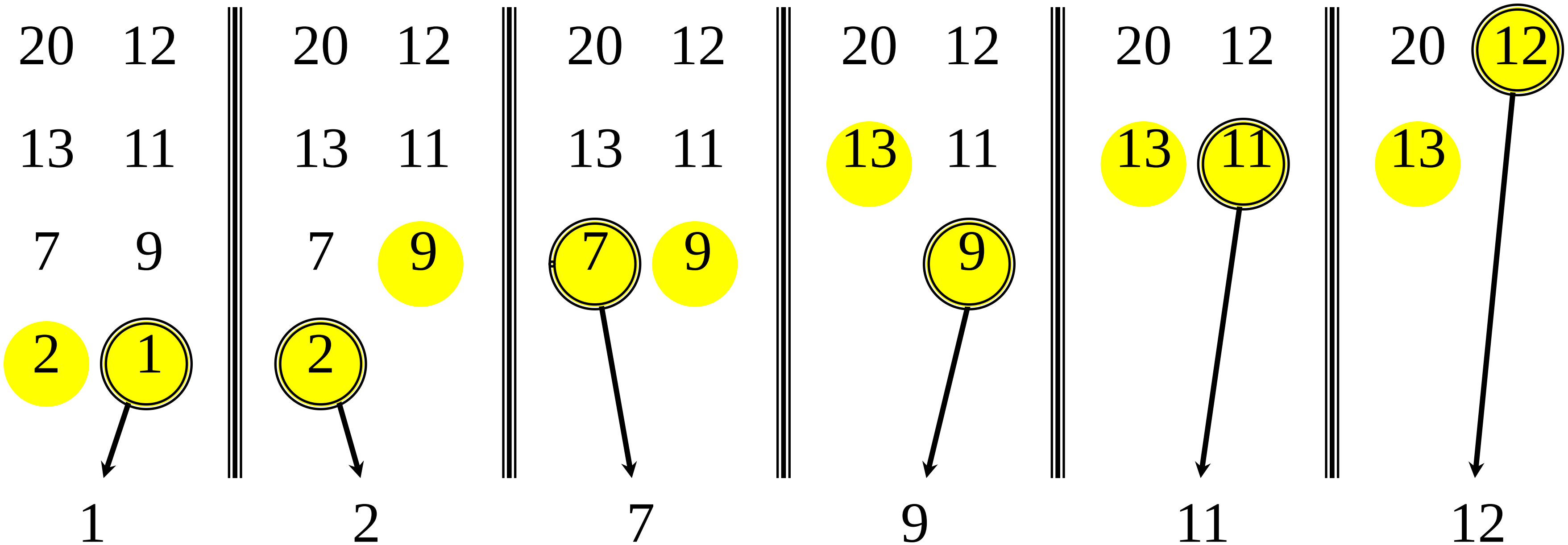


Merging two sorted arrays

20	12	20	12	20	12	20	12	20	12	20	12
13	11	13	11	13	11	13	11	13	11	13	
7	9	7	9	7	9		9				
2	1	2									
1		2		7		9		11		12	



Merging two sorted arrays



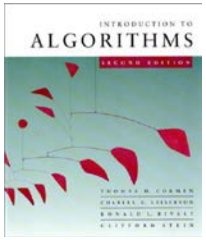
Time = $\Theta(n)$ to merge a total of n elements (linear time).

Analyzing merge sort

$T(n)$	MERGE-SORT $A[1\,..\,n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1\,..\,\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,..\,n\,]$.
$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

Abuse

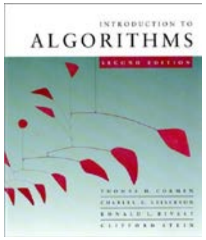
Sloppiness: Should be $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$, but it turns out not to matter asymptotically.



Recurrence for merge sort

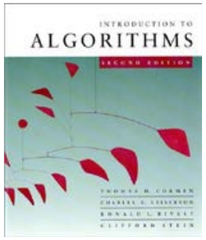
$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n)$.



Recursion tree

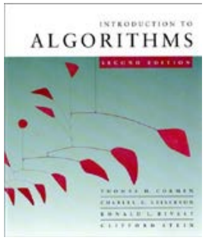
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

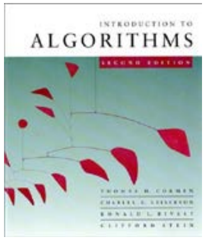


Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$cn$$

$$T(n/2) \qquad\qquad T(n/2)$$



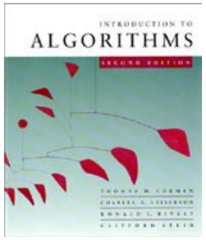
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

cn

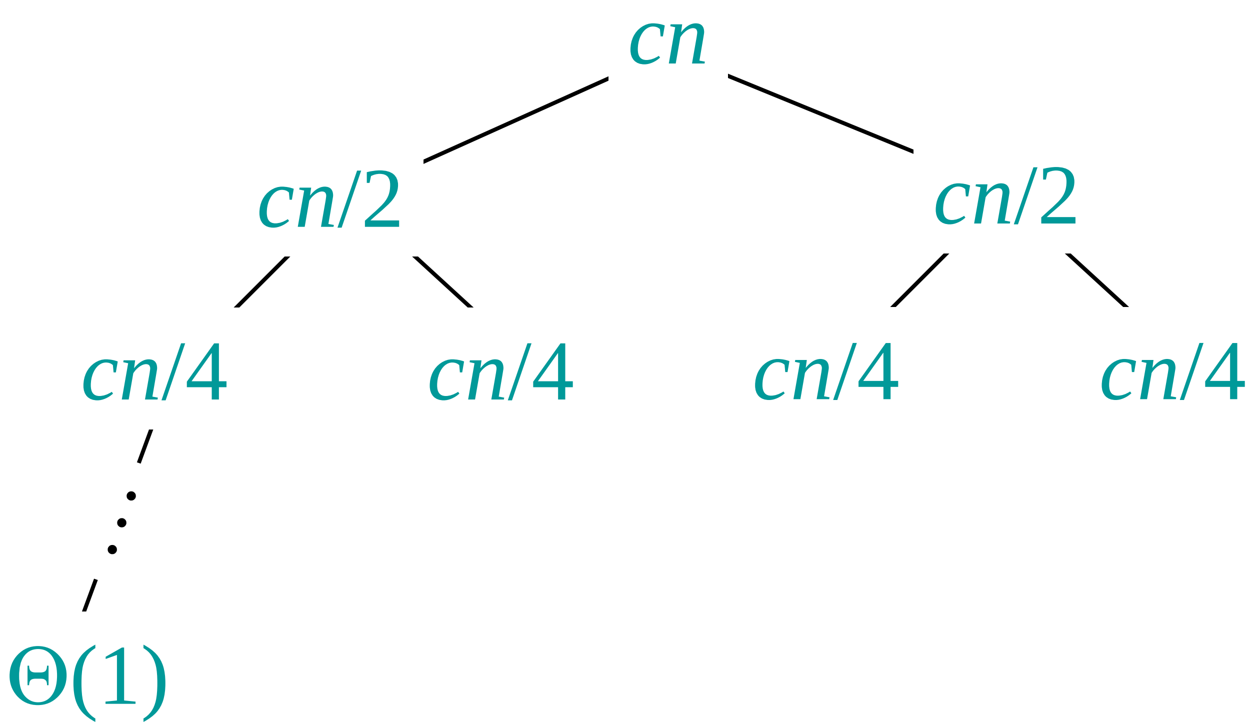
$cn/2$ $\qquad$ $cn/2$

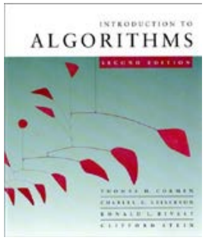
$T(n/4)$ $\quad$ $T(n/4)$ $\quad$ $T(n/4)$ $\quad$ $T(n/4)$



Recursion tree

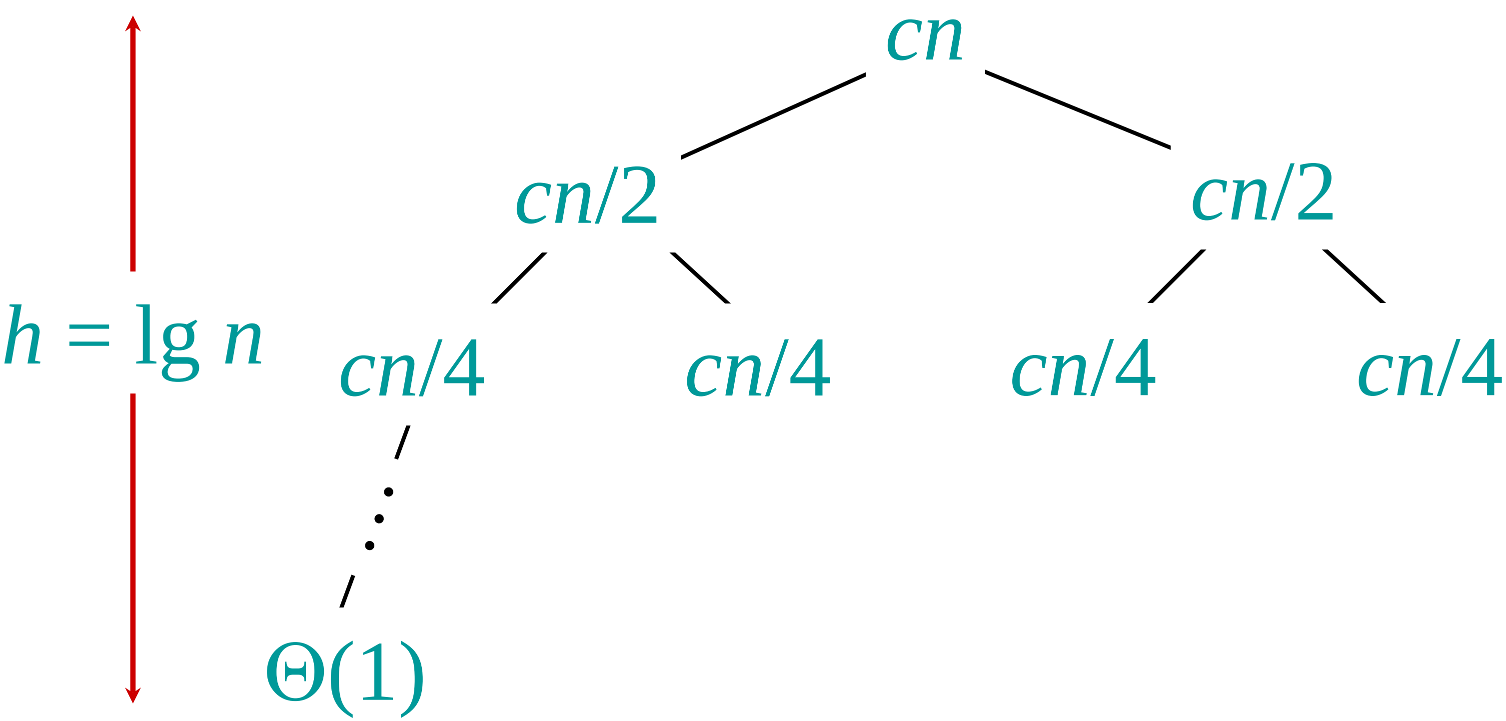
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

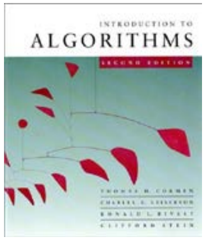




Recursion tree

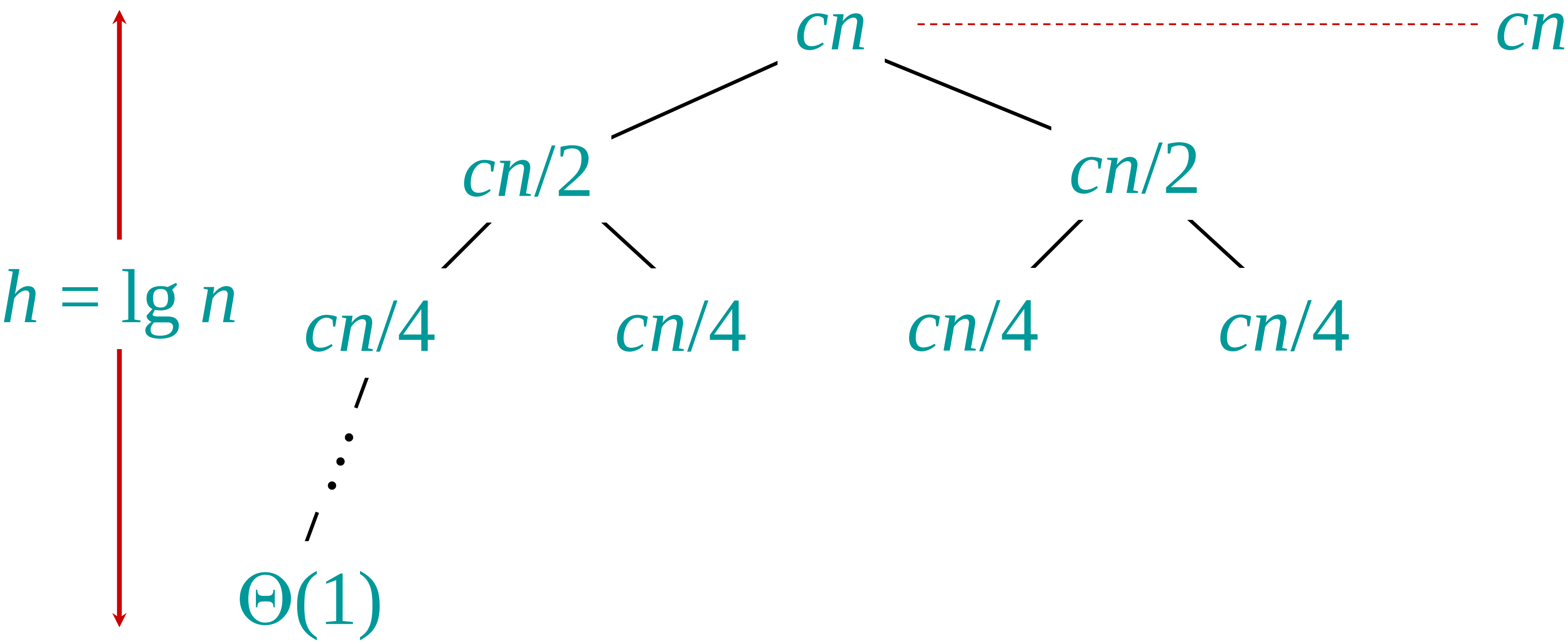
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

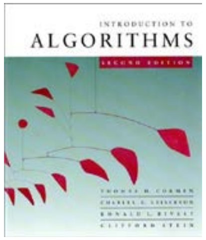




Recursion tree

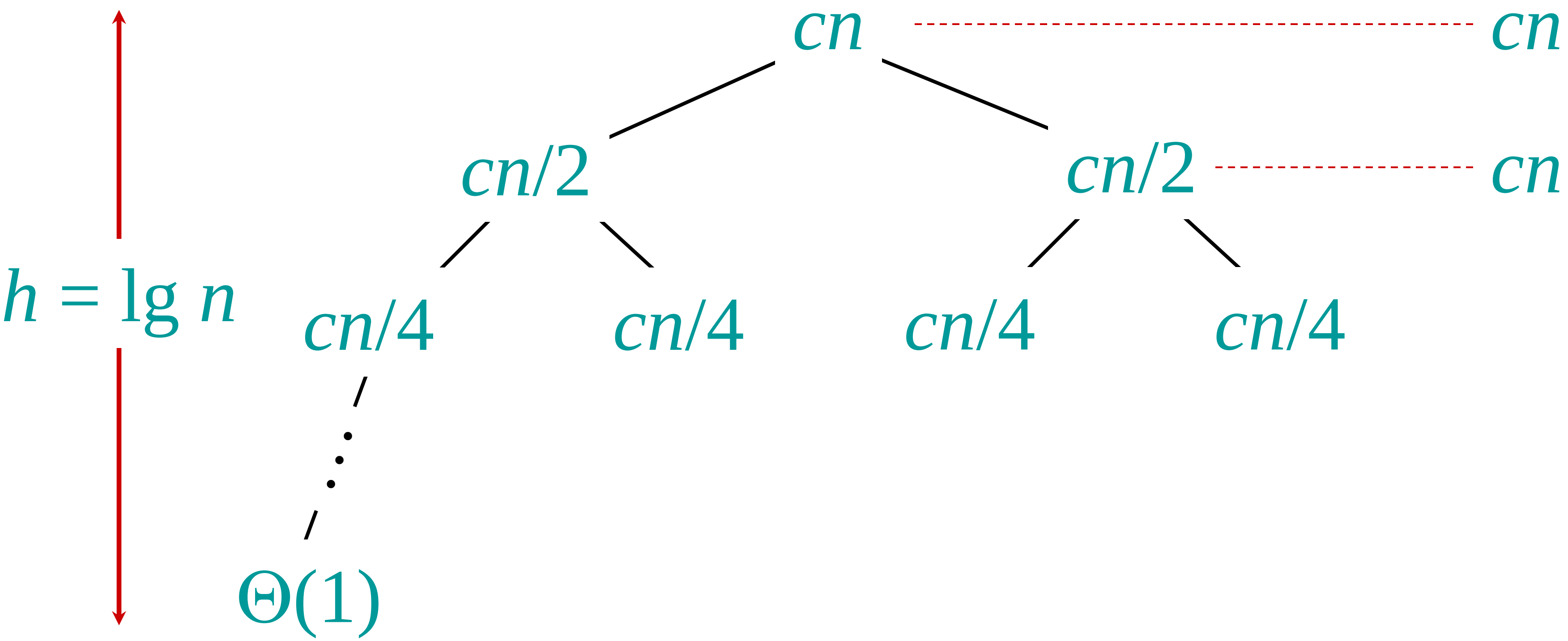
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

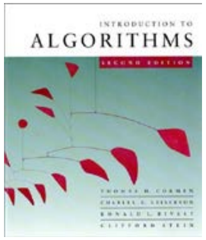




Recursion tree

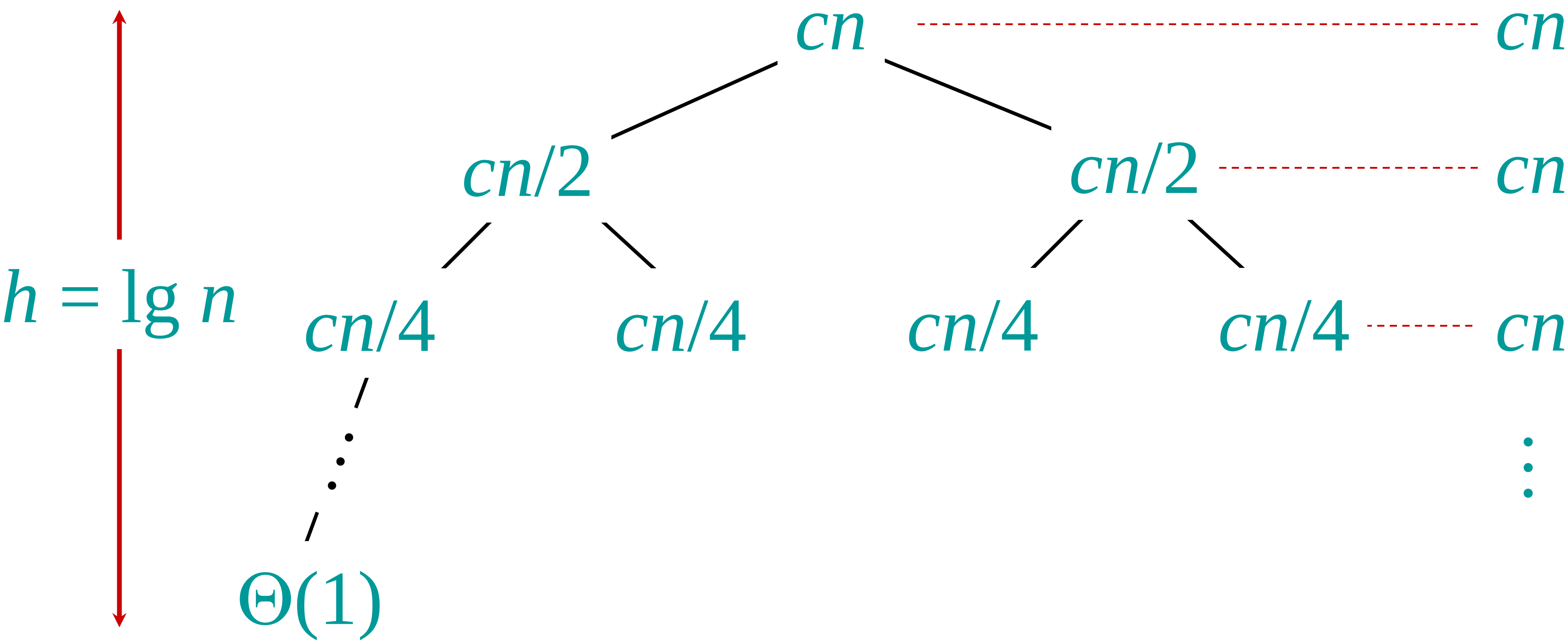
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

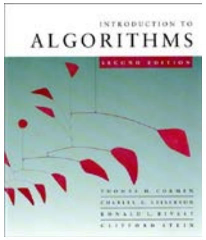




Recursion tree

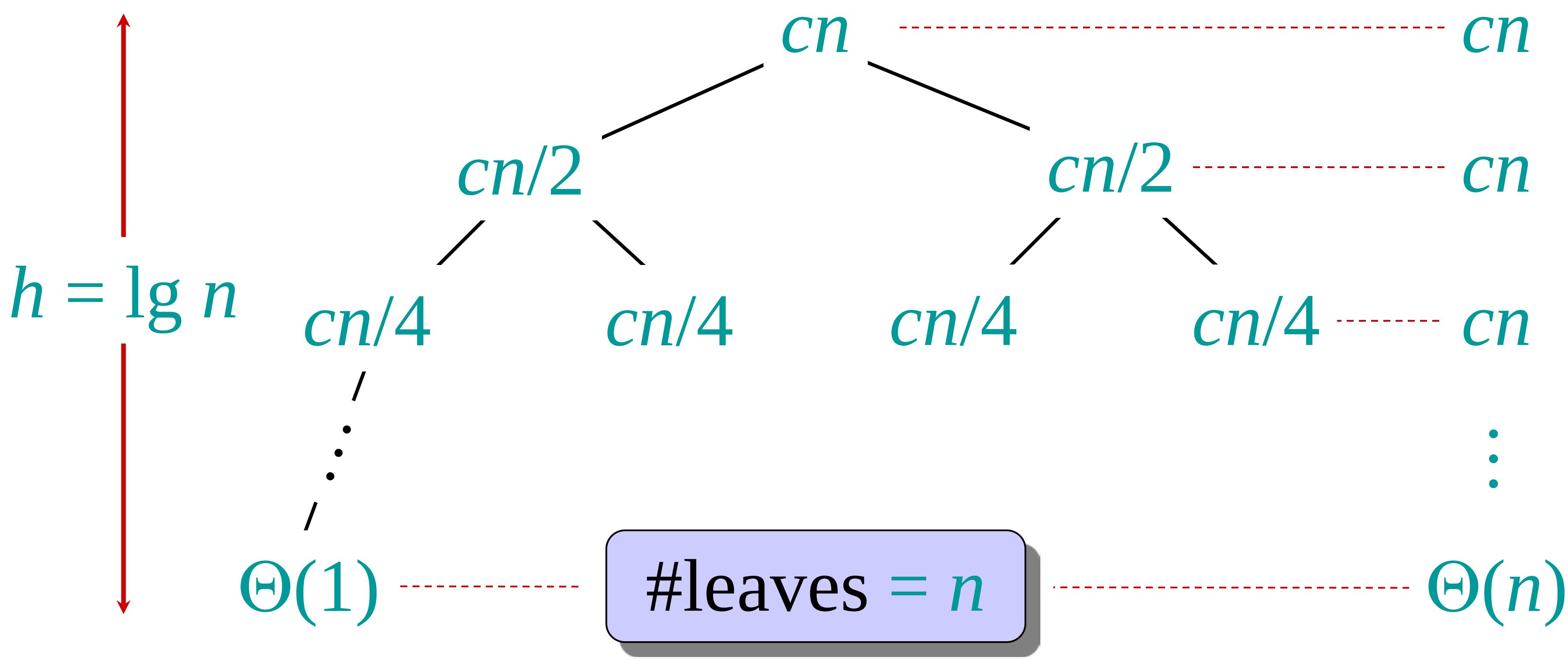
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

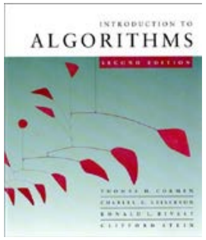




Recursion tree

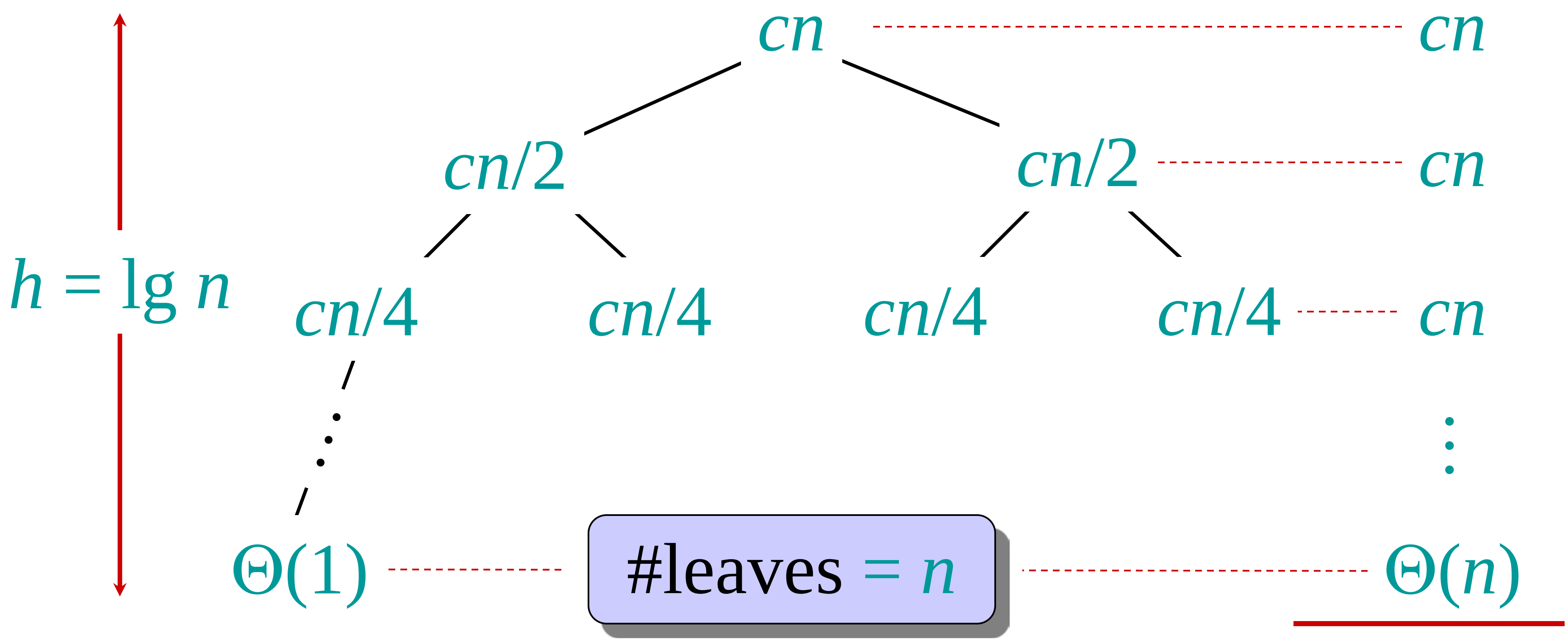
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



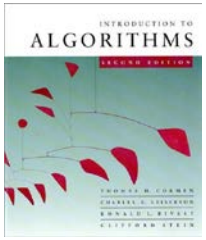


Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

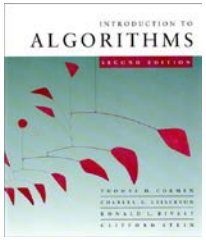


Total $= \Theta(n \lg n)$



Conclusions

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.
- Go test it out for yourself!



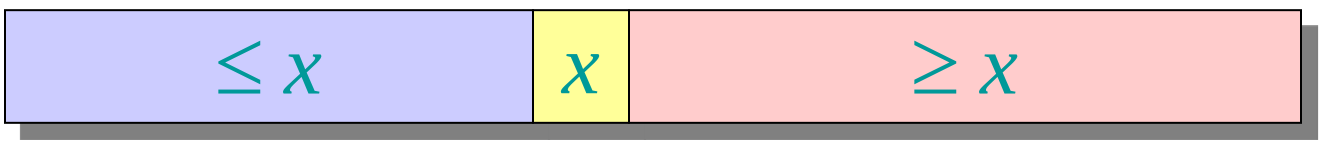
Quicksort

- 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

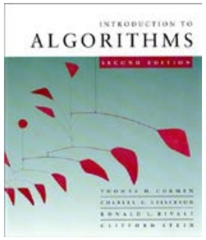
Quicksort an n-element array:

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) ⊳ $A[p\,..\,q]$
    $x \leftarrow A[p]$ ⊳ pivot $= A[p]$
    $i \leftarrow p$
    for $j \leftarrow p + 1$ **to** q
        do if $A[j] \le x$
            then $i \leftarrow i + 1$
                exchange $A[i] \leftrightarrow A[j]$
    exchange $A[p] \leftrightarrow A[i]$
    return i

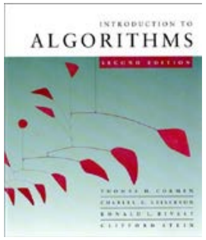
Running time $= O(n)$ for n elements.

Invariant:

x	$\le x$	$\ge x$	?
p	i	j	q

Example of partitioning

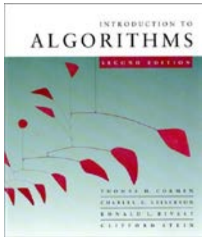
6	10	13	5	8	3	2	11
i	j						



Example of partitioning

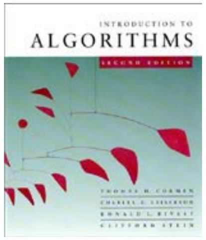
6	10	13	5	8	3	2	11

i $\longrightarrow j$



Example of partitioning

6	10	13	5	8	3	2	11
i			j				

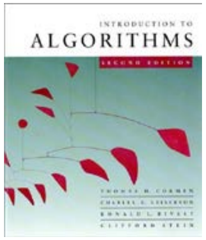


Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

i j

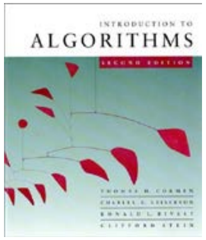


Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

i $\bullet\!\longrightarrow j$



Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11
	i			$\longrightarrow$	j		

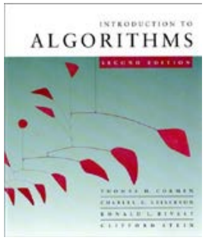
Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

6	5	3	10	8	13	2	11

i $\qquad\qquad\qquad$ j



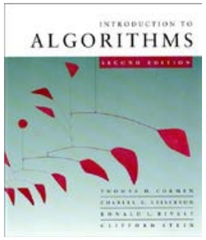
Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

6	5	3	10	8	13	2	11

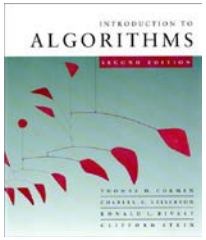
i j



Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11

$\longrightarrow i$ $\qquad\qquad j$



Example of partitioning

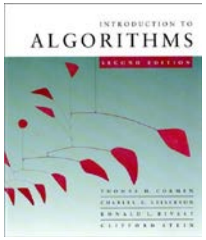
6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

6	5	3	10	8	13	2	11

6	5	3	2	8	13	10	11

i j



Example of partitioning

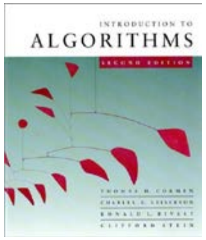
6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

6	5	3	10	8	13	2	11

6	5	3	2	8	13	10	11

i j



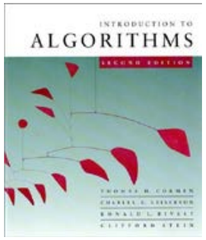
Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11
			i				

Pseudocode for quicksort

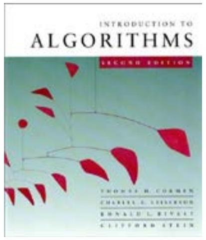
QUICKSORT(A, p, r)
    if $p < r$
        then $q \leftarrow$ PARTITION(A, p, r)
            QUICKSORT$(A, p, q–1)$
            QUICKSORT$(A, q+1, r)$

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



Analysis of quicksort

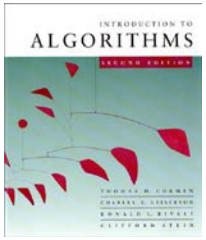
- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n) =$ worst-case running time on an array of n elements.



Worst-case of quicksort

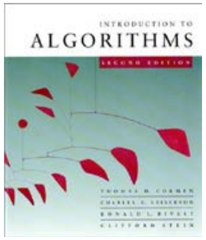
- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$
\begin{aligned}
T(n) &= T(0) + T(n-1) + \Theta(n) \\
&= \Theta(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2) \qquad \textbf{\textit{(arithmetic series)}}
\end{aligned}
$$



Worst-case recursion tree

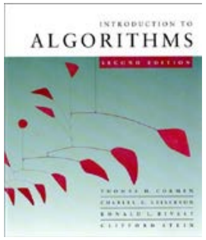
$$T(n) = T(0) + T(n-1) + cn$$



Worst-case recursion tree

$$T(n) = T(0) + T(n–1) + cn$$

$$T(n)$$



Worst-case recursion tree

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

$$cn$$

$$T(0) \qquad T(n-1)$$

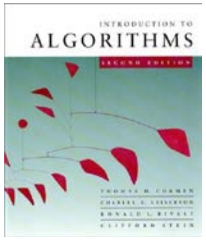
Worst-case recursion tree

$$T(n) = T(0) + T(n\text{–}1) + cn$$

$$cn$$

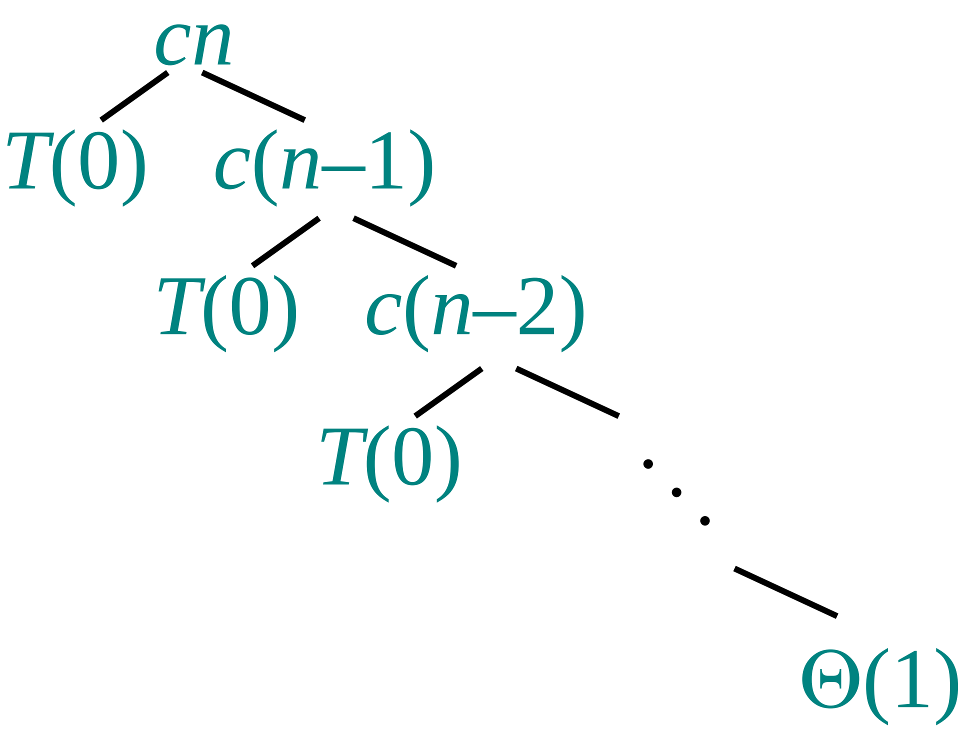
$$T(0) \qquad c(n\text{–}1)$$

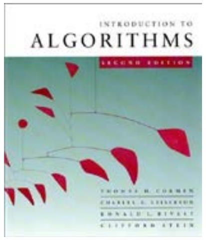
$$T(0) \qquad T(n\text{–}2)$$



Worst-case recursion tree

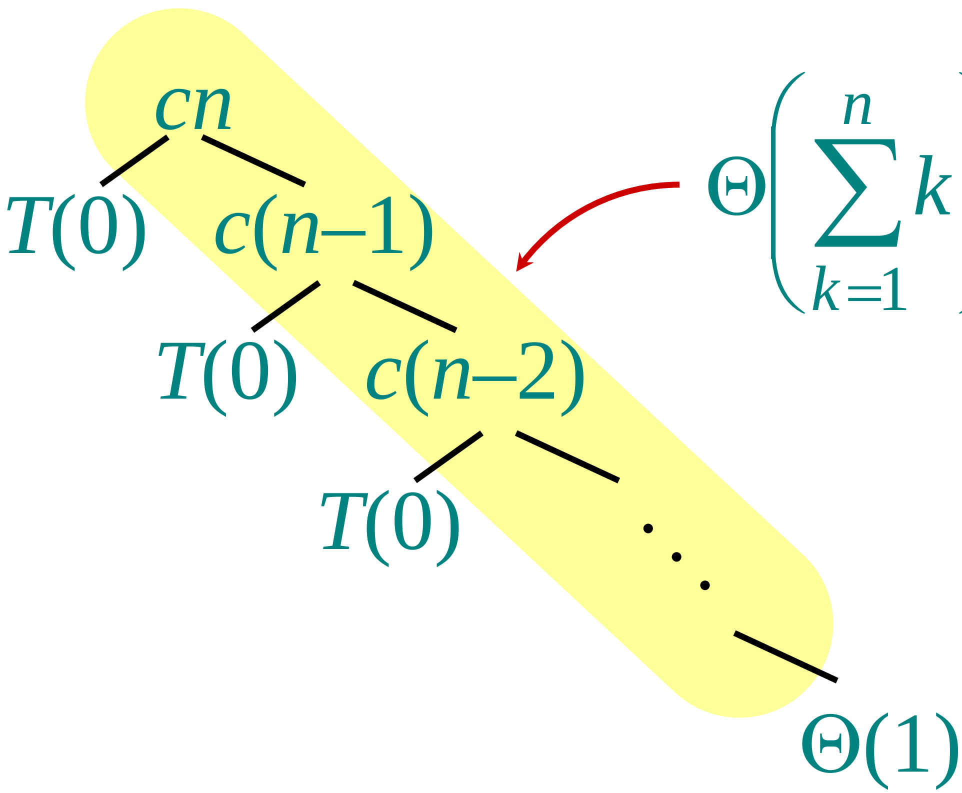
$$T(n) = T(0) + T(n–1) + cn$$



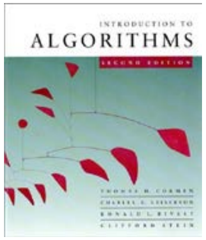


Worst-case recursion tree

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

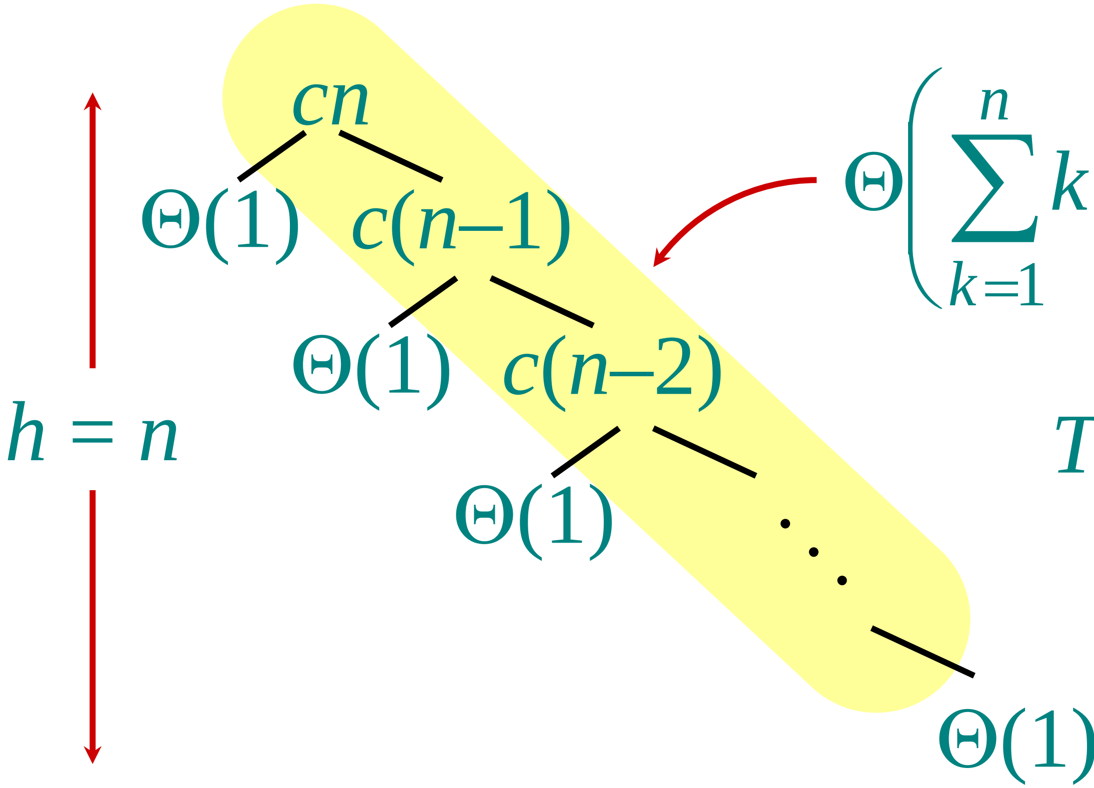


$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$



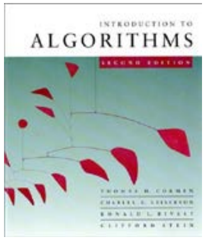
Worst-case recursion tree

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



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$



Best-case analysis

(For intuition only!)

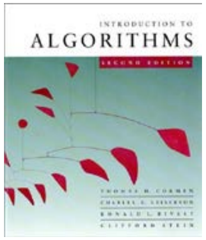
If we're lucky, PARTITION splits the array evenly:

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

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

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

What is the solution to this recurrence?



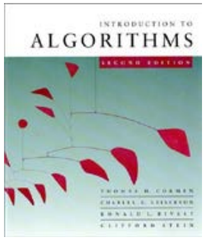
Analysis of “almost-best” case

$$T(n)$$

Analysis of “almost-best” case

$$cn$$

$$T\left(\tfrac{1}{10}n\right) \qquad\qquad T\left(\tfrac{9}{10}n\right)$$

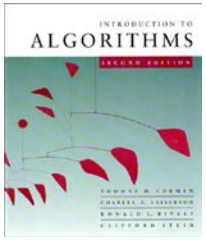


Analysis of "almost-best" case

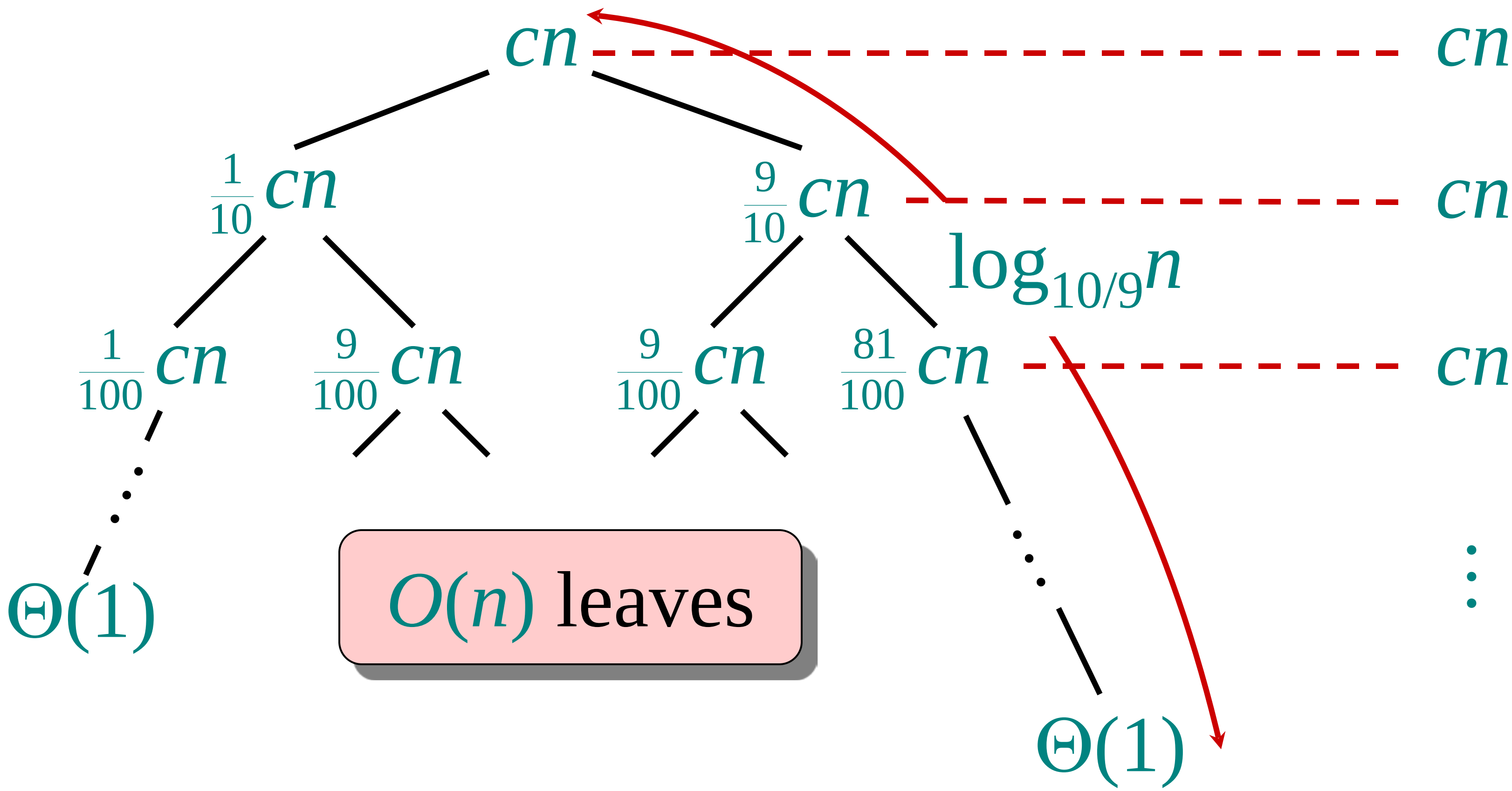
$$cn$$

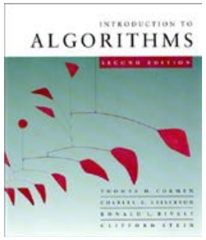
$$\frac{1}{10}cn \qquad\qquad \frac{9}{10}cn$$

$$T\left(\frac{1}{100}n\right)\,T\left(\frac{9}{100}n\right) \qquad T\left(\frac{9}{100}n\right)\,T\left(\frac{81}{100}n\right)$$

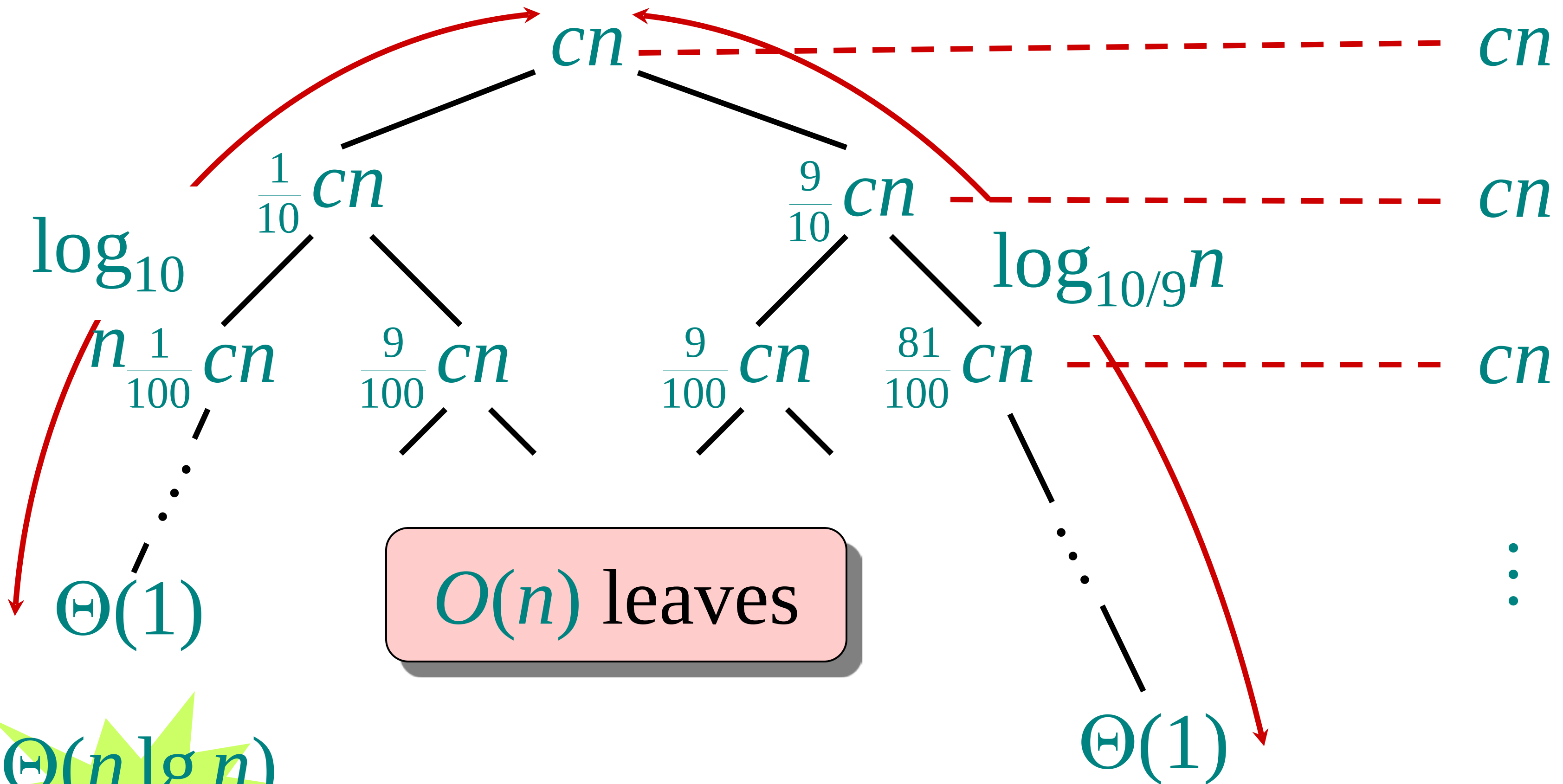


Analysis of “almost-best” case



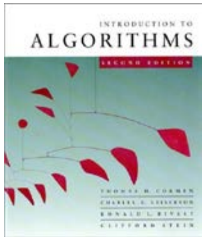


Analysis of "almost-best" case



$\Theta(n \lg n)$
Lucky!

$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$



More intuition

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

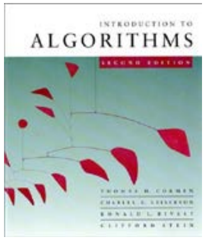
$$L(n) = 2U(n/2) + \Theta(n) \quad \textit{lucky}$$
$$U(n) = L(n-1) + \Theta(n) \quad \textit{unlucky}$$

Solving:

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

Lucky!

How can we make sure we are usually lucky?



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

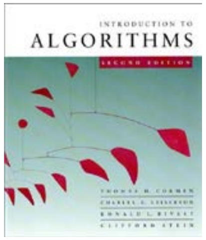
for $i \leftarrow 2$ **to** k

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

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

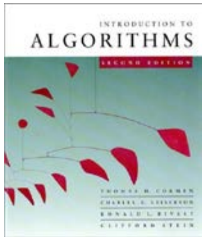
A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4

B:



Loop 1

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
0	0	0	0

B:

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

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

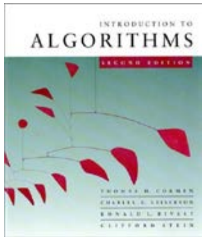
C:

1	2	3	4
0	0	0	1

B:

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

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

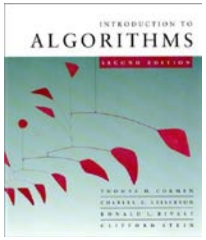
C:

1	2	3	4
1	0	0	1

B:

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

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



Loop 2

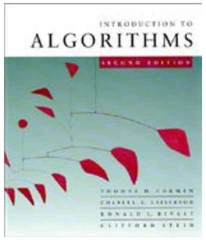
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	1

B:					

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

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

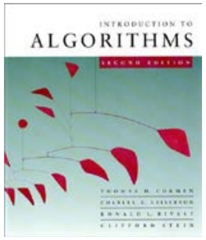
C:

1	2	3	4
1	0	1	2

B:

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

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



Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	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 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	2	2

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

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

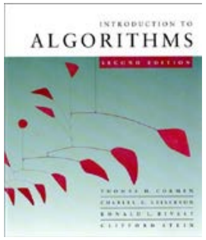
Loop 3

	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	2

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

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



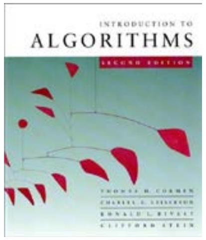
Loop 3

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

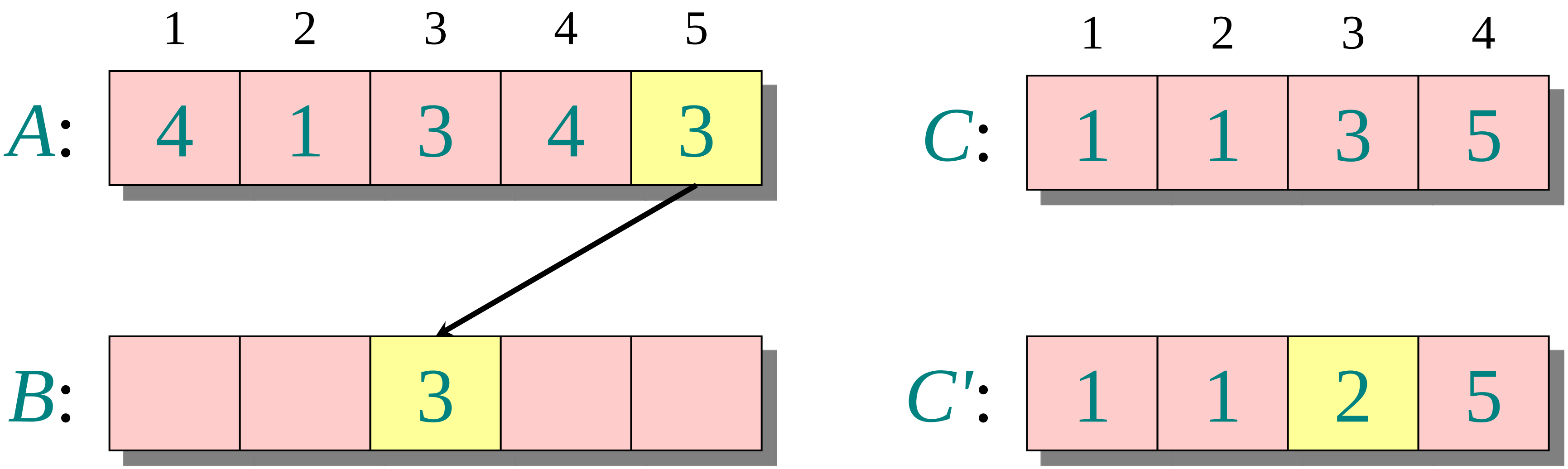
	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

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

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



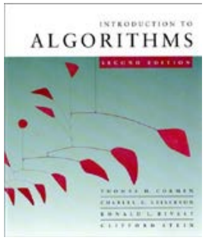
Loop 4



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

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

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



Loop 4

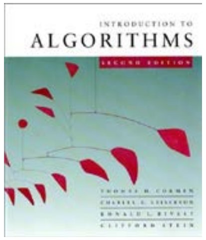
	1	2	3	4	5
A:	4	1	3	4	3
B:			3		4

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

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

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

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



Loop 4

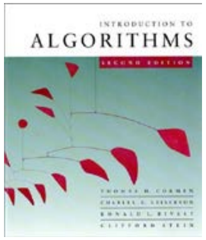
	1	2	3	4	5
A:	4	1	3	4	3
B:		3	3		4

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

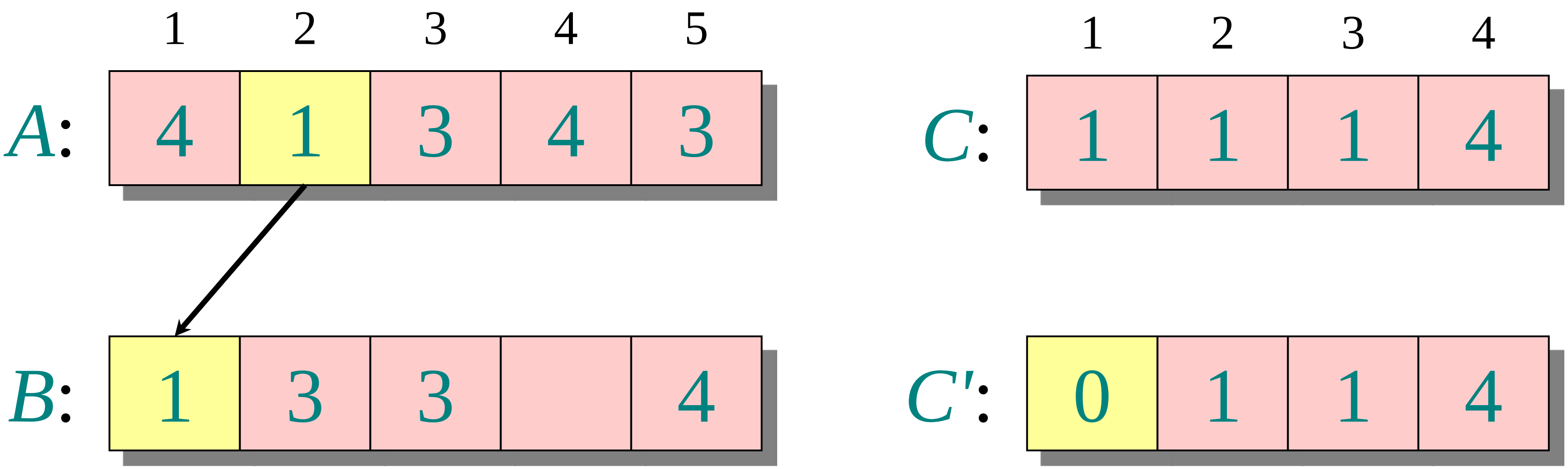
for $j \leftarrow n$ **downto** 1

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

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



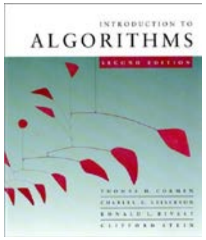
Loop 4



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

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

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



Loop 4

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

	1	2	3	4
C:	0	1	1	4

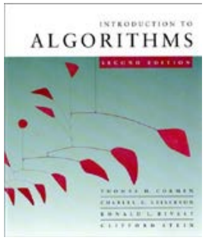
B:	1	3	3	4	4

C':	0	1	1	3

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

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

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



Analysis

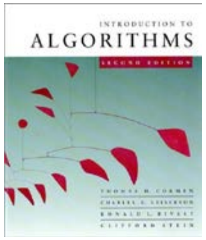
$\Theta(k)$
for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$

$\Theta(n)$
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 $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$



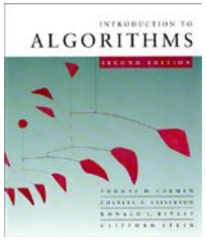
Running time

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

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

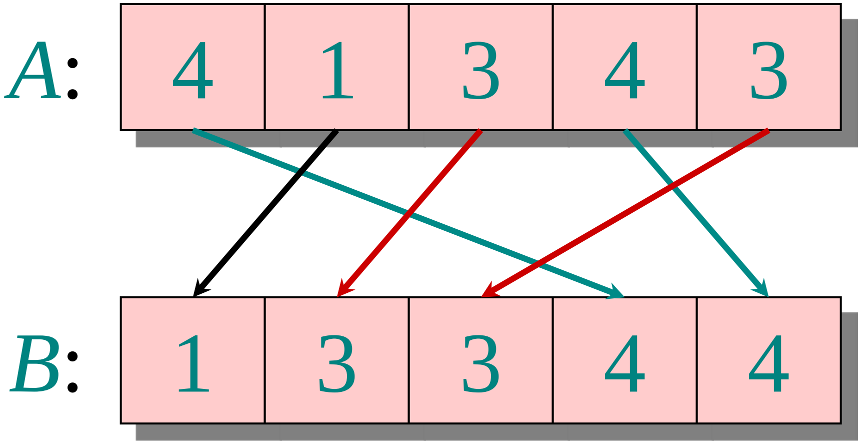
Answer:

- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

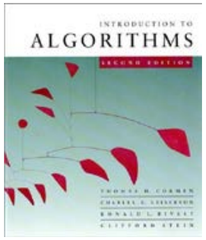


Stable sorting

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

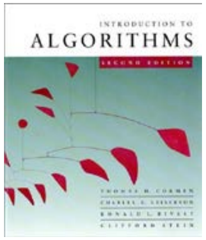


Exercise: What other sorts have this property?



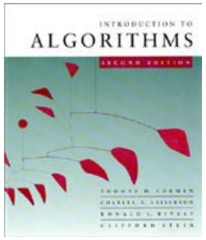
Radix sort

- ***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

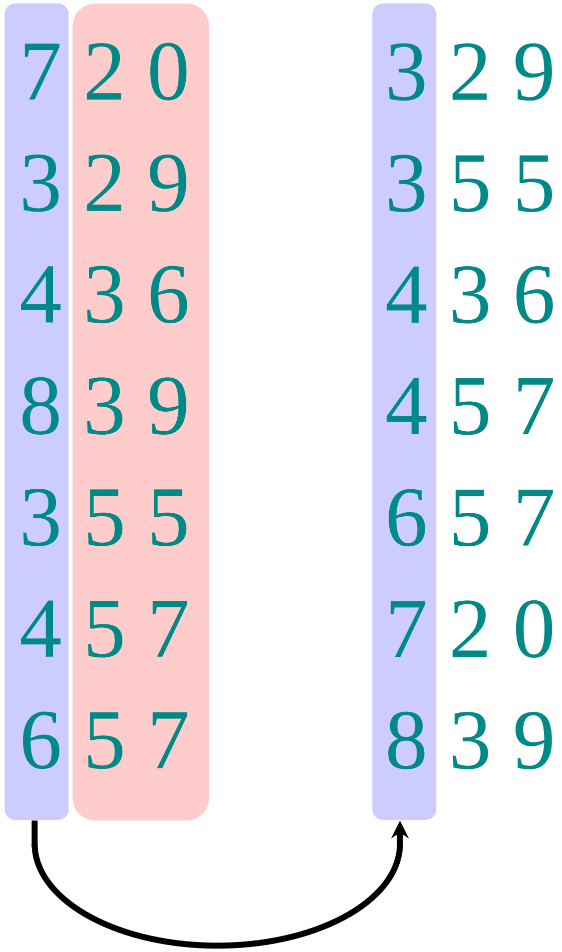
3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

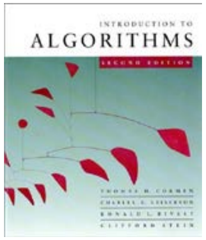


Correctness of radix sort

Induction on digit position

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



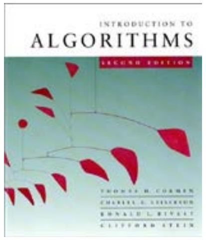


Correctness of radix sort

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.

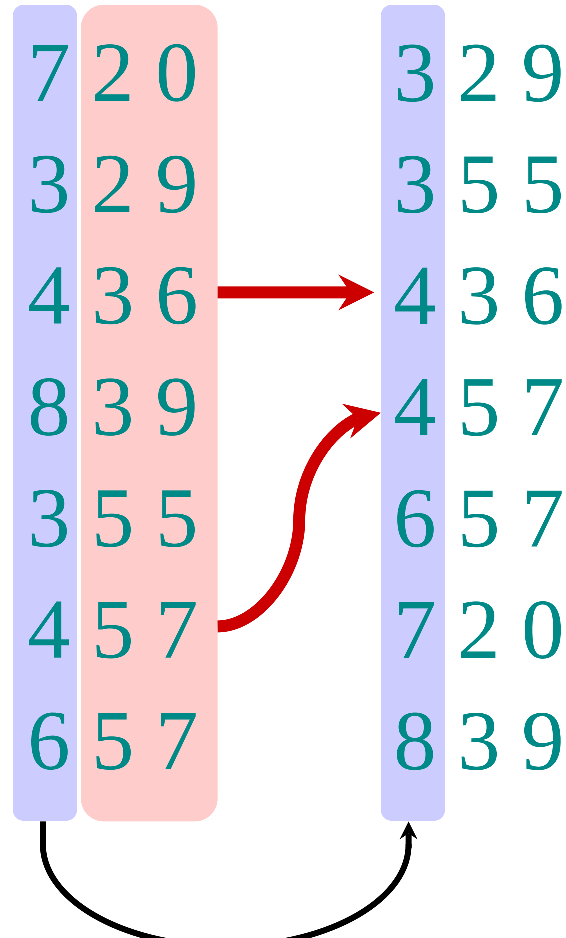
7 2 0	3 2 9
3 2 9	3 5 5
4 3 6	4 3 6
8 3 9	4 5 7
3 5 5	6 5 7
4 5 7	7 2 0
6 5 7	8 3 9

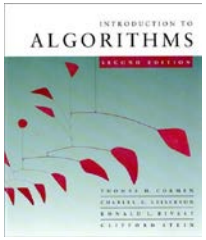


Correctness of radix sort

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.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.

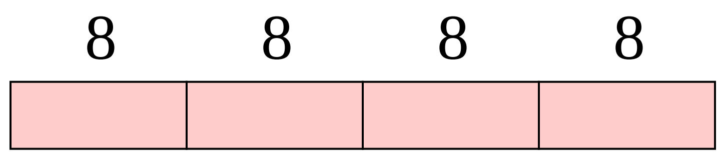




Analysis of radix sort

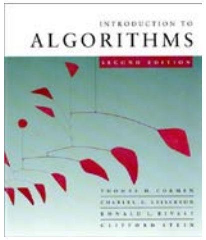
- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

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*

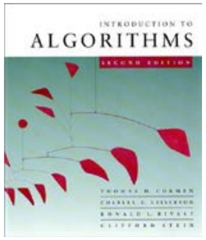
$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right)$$

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.