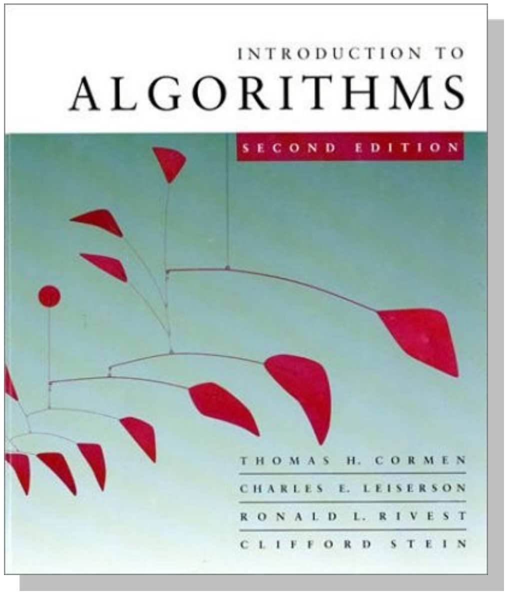
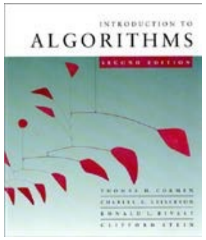
Introduction to Algorithms
6.046J/18.401J



LECTURE 1
Analysis of Algorithms
- Bubble sort
- Asymptotic analysis
- Merge sort
- Recurrences

Prof. Charles E. Leiserson

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



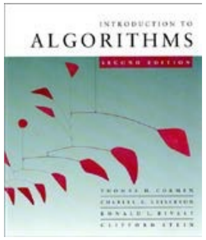
Analysis of algorithms

The theoretical study of computer-program performance and resource usage.

What's more important than performance?

- modularity
- correctness
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability

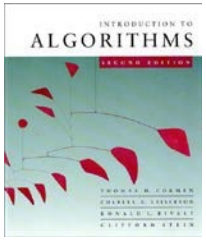
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Why study algorithms and performance?

- Algorithms help us to understand ***scalability***.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** for talking about program behavior.
- Performance is the ***currency*** of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



The problem of sorting

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

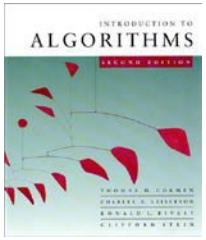
Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

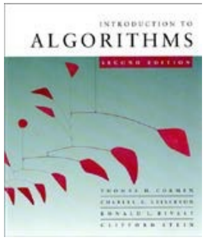
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Bubble Sort

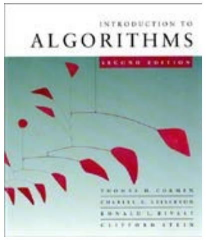


"Bubbling Up" the Largest Element

Traverse a collection of elements

- **Move from the front to the end**
- **"Bubble" the largest value to the end using pair-wise comparisons and swapping**

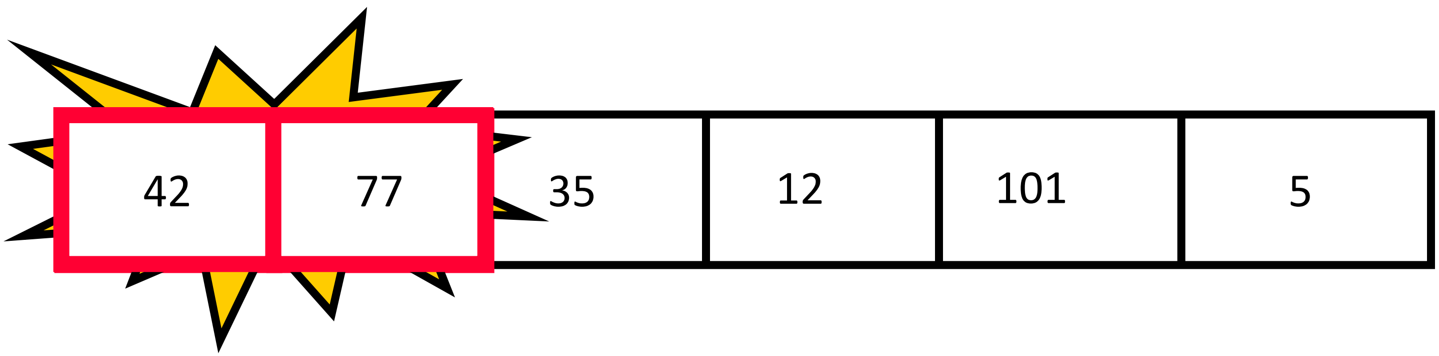
77	42	35	12	101	5

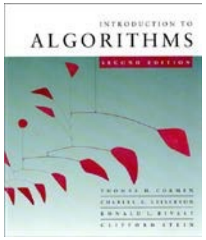


"Bubbling Up" the Largest Element

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

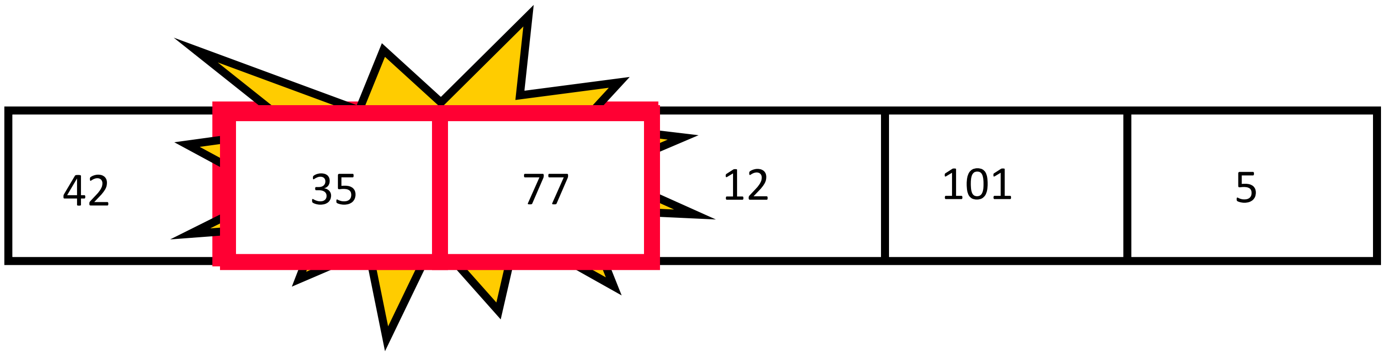


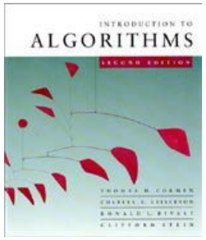


"Bubbling Up" the Largest Element

Traverse a collection of elements

- **Move from the front to the end**
- **"Bubble" the largest value to the end using pair-wise comparisons and swapping**

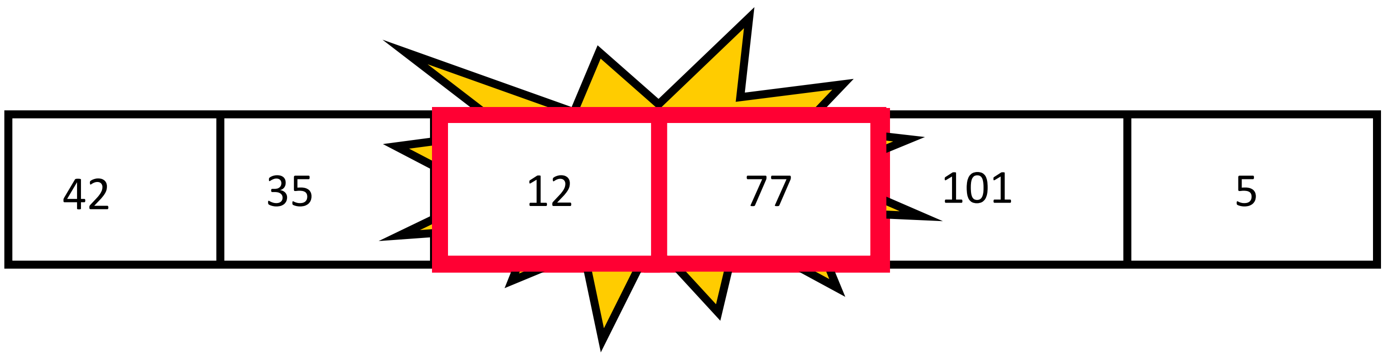


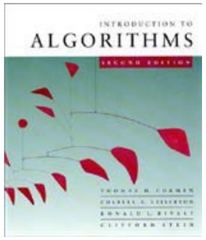


"Bubbling Up" the Largest Element

Traverse a collection of elements

- **Move from the front to the end**
- **"Bubble" the largest value to the end using pair-wise comparisons and swapping**





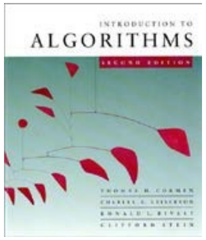
"Bubbling Up" the Largest Element

Traverse a collection of elements

- **Move from the front to the end**
- **"Bubble" the largest value to the end using pair-wise comparisons and swapping**

42	35	12	77	101	5

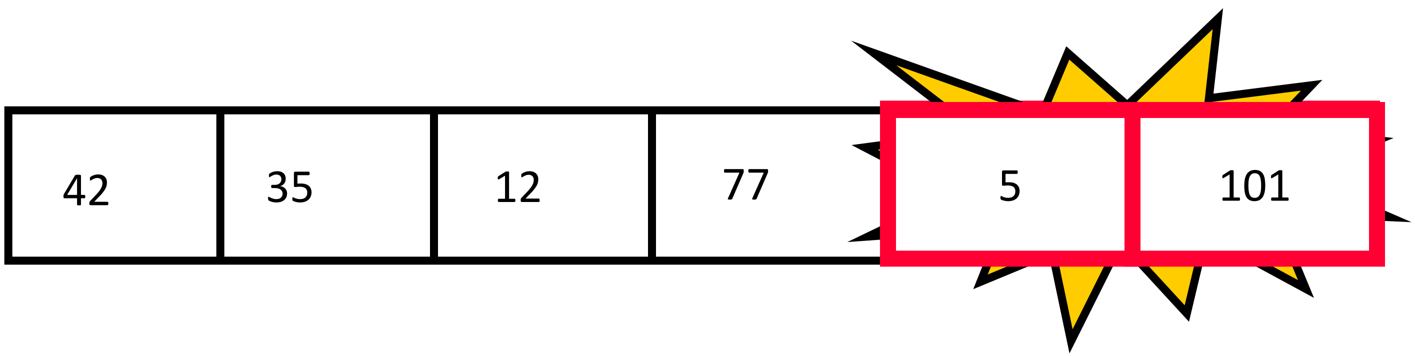
No need to swap

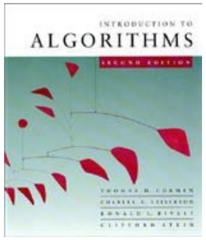


"Bubbling Up" the Largest Element

Traverse a collection of elements

- **Move from the front to the end**
- **"Bubble" the largest value to the end using pair-wise comparisons and swapping**





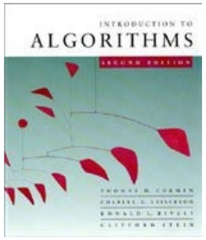
"Bubbling Up" the Largest Element

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

42	35	12	77	5	101

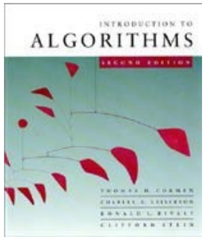
Largest value correctly placed



The “Bubble Up” Algorithm

```
index <- 1
last_compare_at <- n – 1

loop
  exitif(index > last_compare_at)
  if(A[index] > A[index + 1]) then
    Swap(A[index], A[index + 1])
  endif
  index <- index + 1
endloop
```

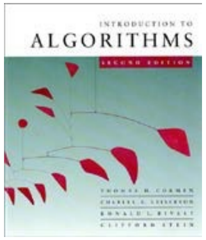


Items of Interest

- **Notice that only the largest value is correctly placed**
- **All other values are still out of order**
- **So we need to repeat this process**

42	35	12	77	5	101

Largest value correctly placed



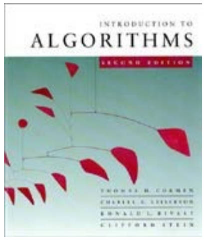
Repeat "Bubble Up" How Many Times?

- **If we have N elements…**

- **And if each time we bubble an element, we place it in its correct location…**

- **Then we repeat the "bubble up" process N – 1 times.**

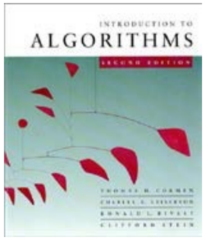
- **This guarantees we'll correctly place all N elements.**



“Bubbling” All the Elements

N - 1

42	35	12	77	5	101
35	12	42	5	77	101
12	35	5	42	77	101
12	5	35	42	77	101
5	12	35	42	77	101



Reducing the Number of Comparisons

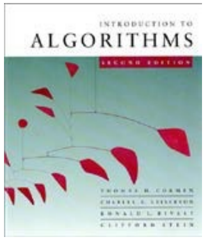
77	42	35	12	101	5

42	35	12	77	5	101

35	12	42	5	77	101

12	35	5	42	77	101

12	5	35	42	77	101

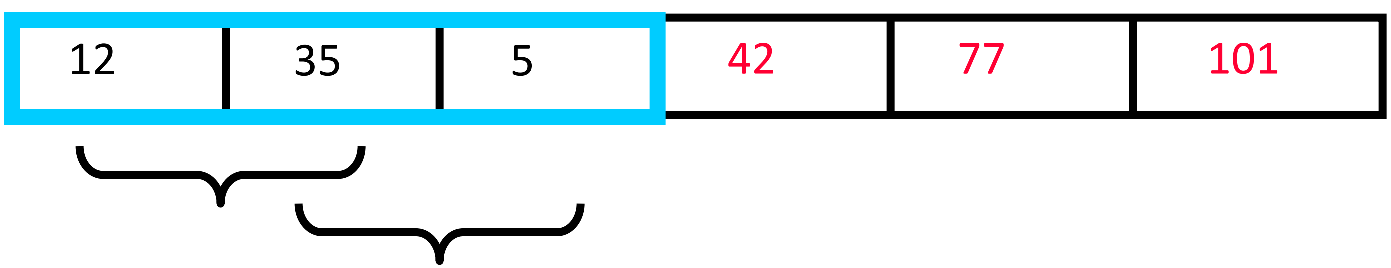


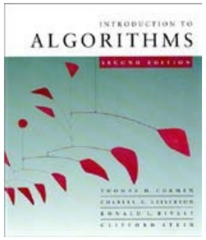
Reducing the Number of Comparisons

On the Nth "bubble up", we only need to do MAX-N comparisons.

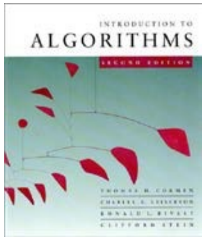
For example:

- This is the 4th "bubble up"
- MAX is 6
- Thus we have 2 comparisons to do





Putting It All Together

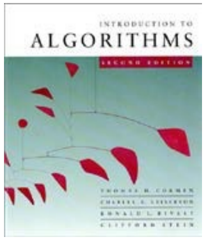



```
procedure Bubblesort(A)
  to_do, index isoftype Num
  to_do <- N - 1

  loop
    exitif(to_do = 0)
    index <- 1
    loop
      exitif(index > to_do)
      if(A[index] > A[index + 1]) then
        Swap(A[index], A[index + 1])
      endif
      index <- index + 1
    endloop
    to_do <- to_do - 1
  endloop
endprocedure // Bubblesort
```

Inner loop

Outer loop

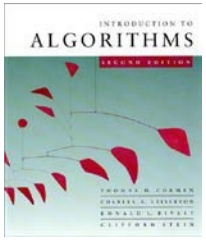


Already Sorted Collections?

- **What if the collection was already sorted?**
- **What if only a few elements were out of place and after a couple of "bubble ups," the collection was sorted?**

- **We want to be able to detect this and "stop early"!**

5	12	35	42	77	101

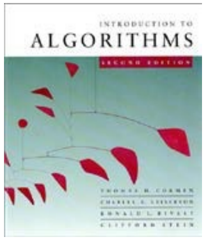


Using a Boolean "Flag"

- We can use a boolean variable to determine if any swapping occurred during the "bubble up."

- If no swapping occurred, then we know that the collection is already sorted!

- This boolean "flag" needs to be reset after each "bubble up."



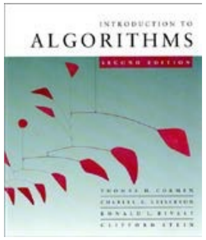

```
did_swap: Boolean
did_swap <- true

loop
  exitif ((to_do = 0) OR NOT(did_swap))
  index <- 1
  did_swap <- false
  loop
    exitif(index > to_do)
    if(A[index] > A[index + 1]) then
      Swap(A[index], A[index + 1])
      did_swap <- true
    endif
    index <- index + 1
  endloop
  to_do <- to_do - 1
endloop
```

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.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Kinds of analyses

Worst-case: (usually)

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

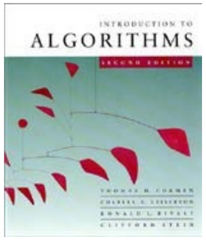
Average-case: (sometimes)

- $T(n)$ = 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.

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Machine-independent time

What is bubble sort's worst-case time?

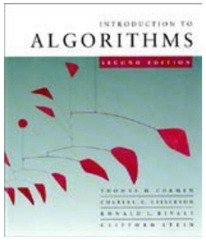
- It depends on the speed of our computer:
  - relative speed (on the same machine),
  - absolute speed (on different machines).

BIG IDEA:

- Ignore machine-dependent constants.
- Look at ***growth*** of $T(n)$ as $n \to \infty$.

"Asymptotic Analysis"

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Θ-notation

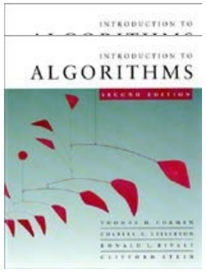
Math:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

Engineering:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



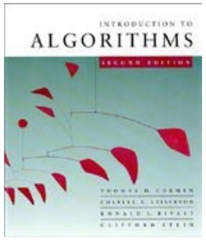
Other notations

$\mathrm{O}(g(n)) = \{ f(n)$: there exist positive constants c_2, and n_0 such that $0 \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c_1, and n_0 such that $0 \le c_2 g(n) \le f(n)$ for all $n \ge n_0 \}$

$\mathrm{o}(g(n)) = \{ f(n)$: for any $\varepsilon \ge 0$, there exist n_0 such that $0 \le g(n) \le \varepsilon f(n)$ for all $n \ge n_0 \}$

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$



Examples

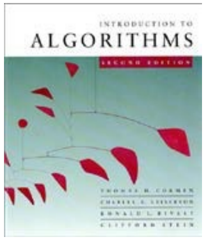
$$\sum_{i=1}^{n} i =$$

$$\sum_{i=1}^{n} i^2 =$$

$$\sum_{i=1}^{n} a^i \cdot c =$$

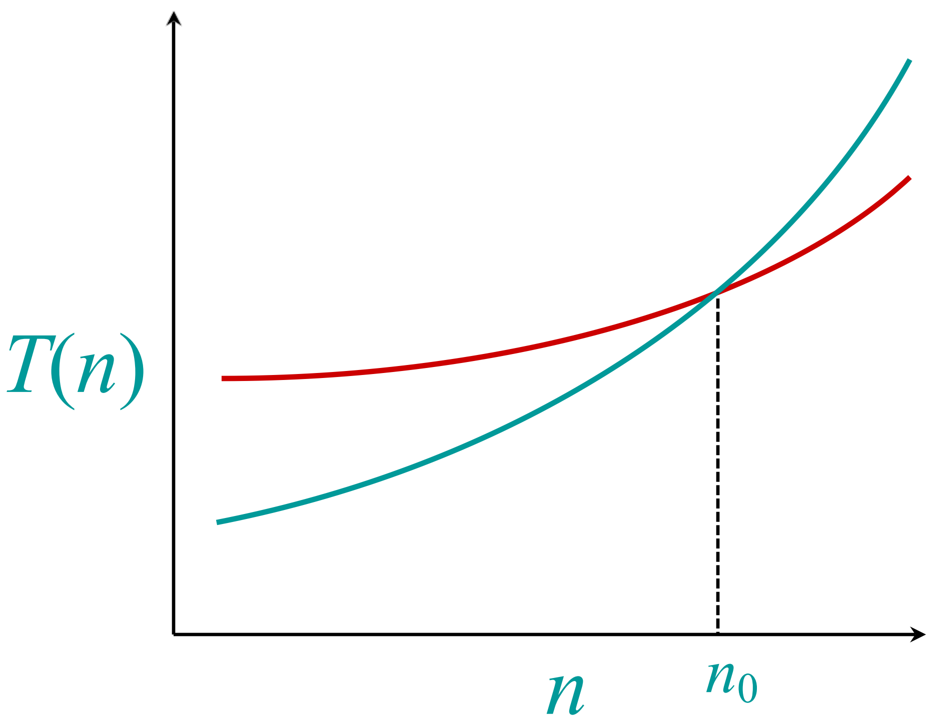
$$\sum_{i=1}^{n} \frac{1}{i} =$$

$$\sum_{i=1}^{n} \frac{1}{i^2} =$$



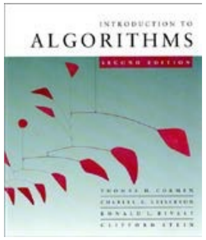
Asymptotic performance

When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Insertion sort analysis

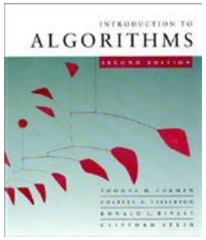
Worst case: Input reverse sorted.

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

Properties of Bubble Sort

- Bubble sort is a **stable** sorting algorithm.
- Bubble sort is an **in-place** sorting algorithm.
- Number of swaps in bubble sort = Number of inversion pairs present in the given array.
- Bubble sort is beneficial when array elements are less and the array is nearly sorted.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



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.

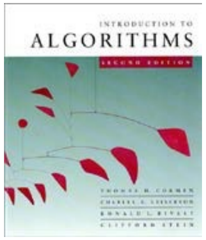
Key <u>subroutine</u>: MERGE

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Merging two sorted arrays

20	12
13	11
7	9
2	1

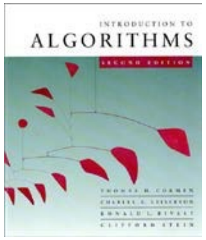
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



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

20	12
13	11
7	9
2	1

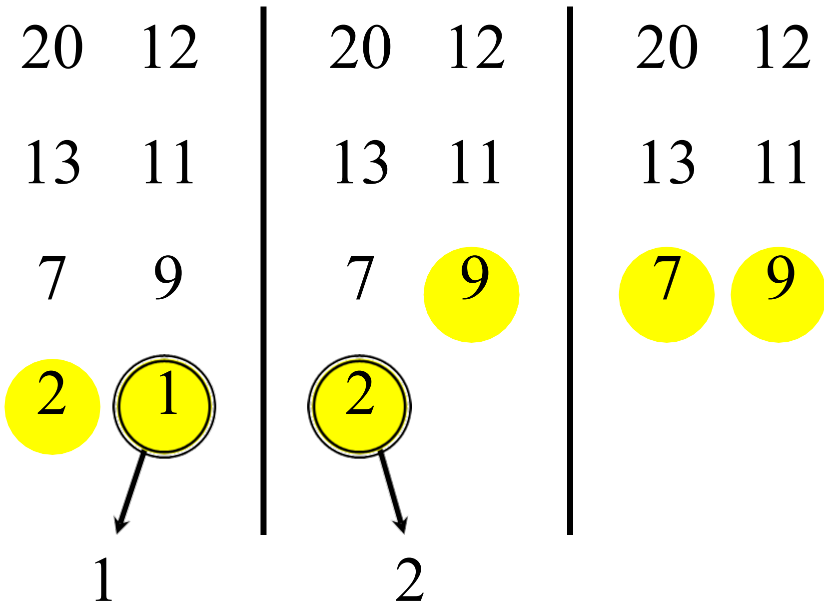
Output: 1

20	12
13	11
7	9
2	

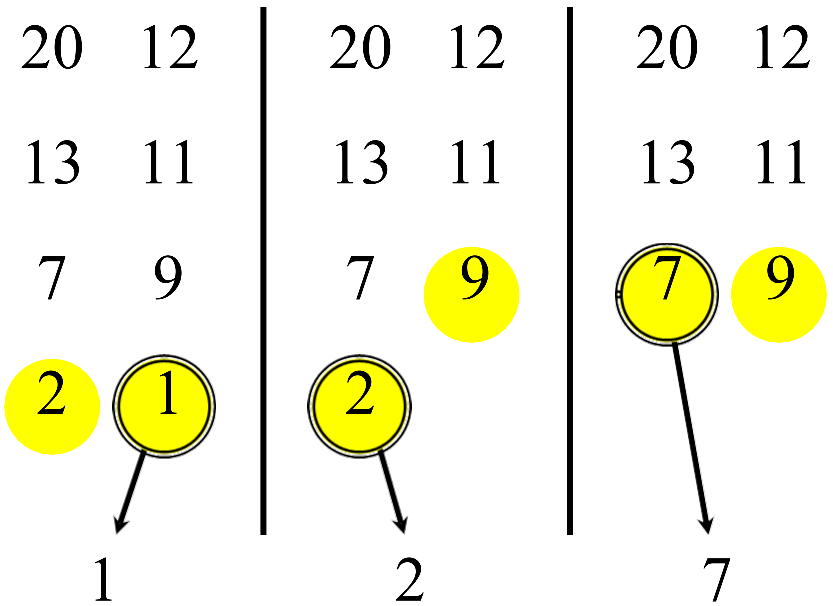
Output: 2

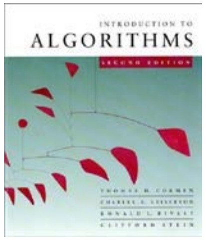
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Merging two sorted arrays

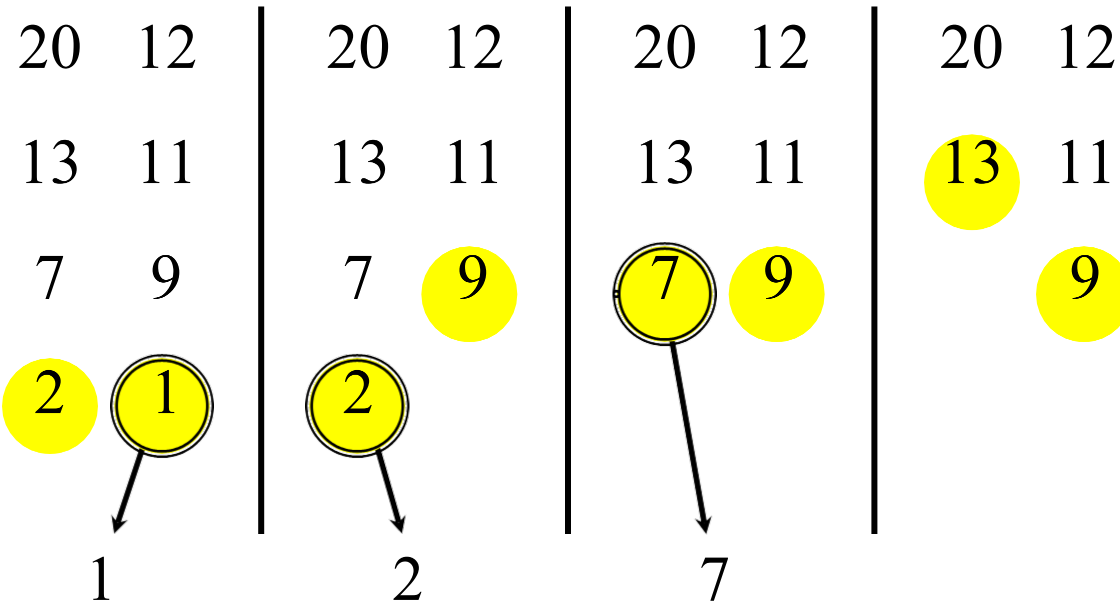


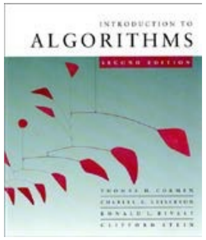
Merging two sorted arrays



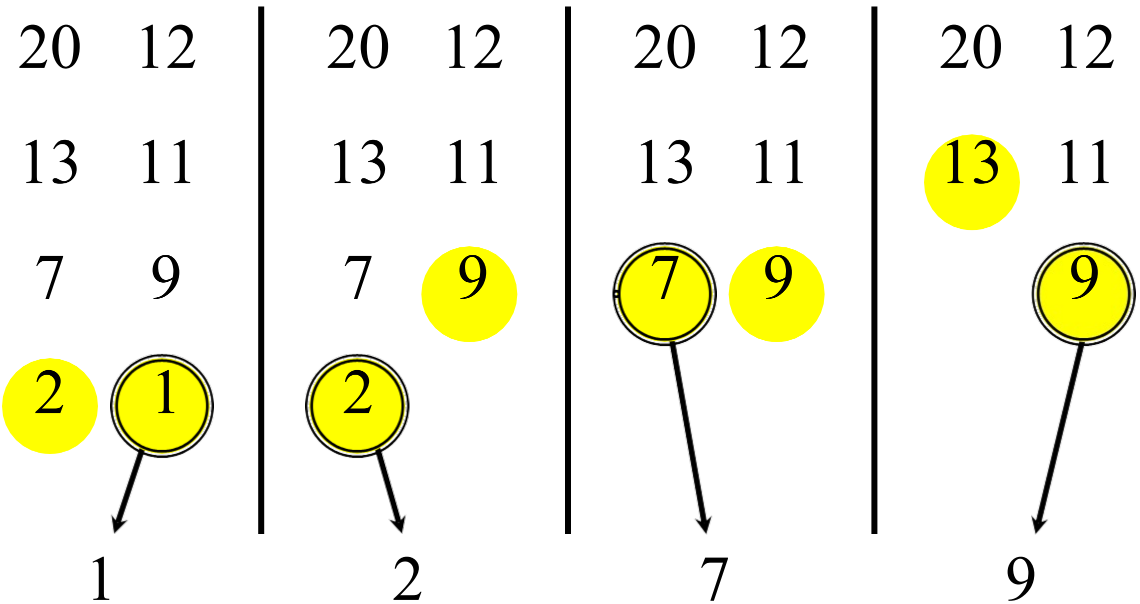


Merging two sorted arrays

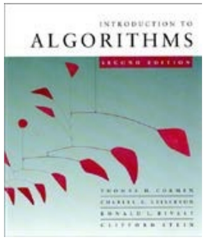




Merging two sorted arrays



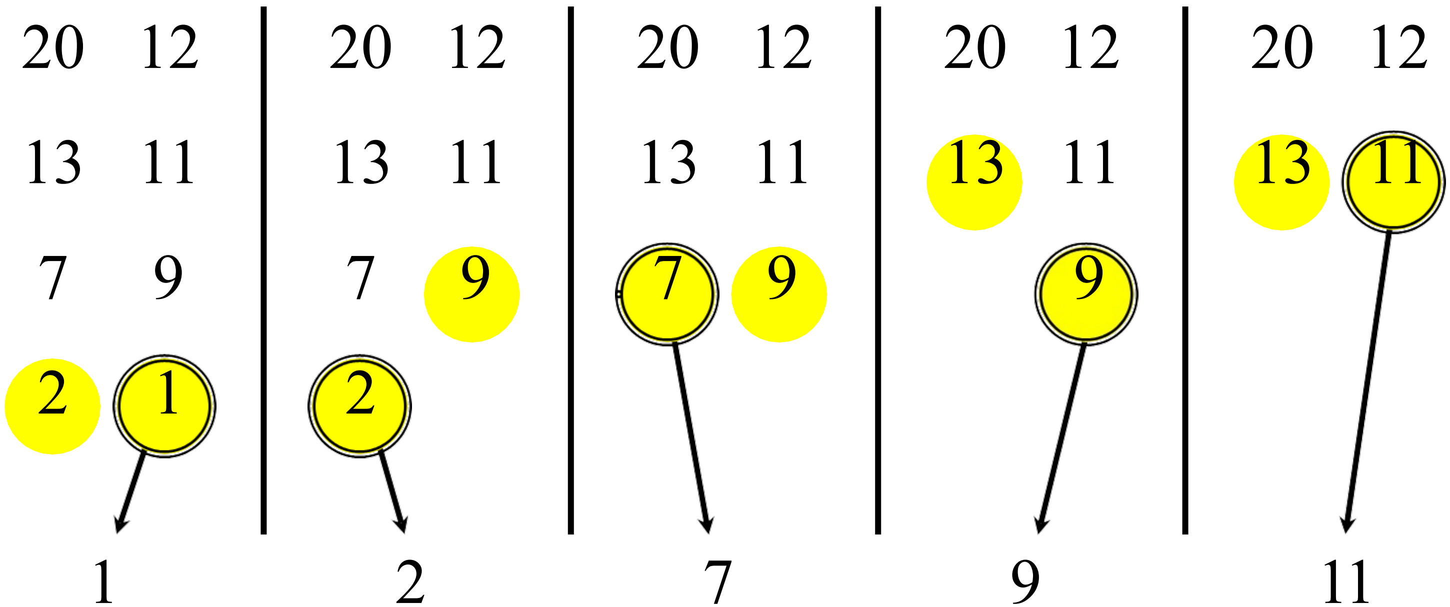
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



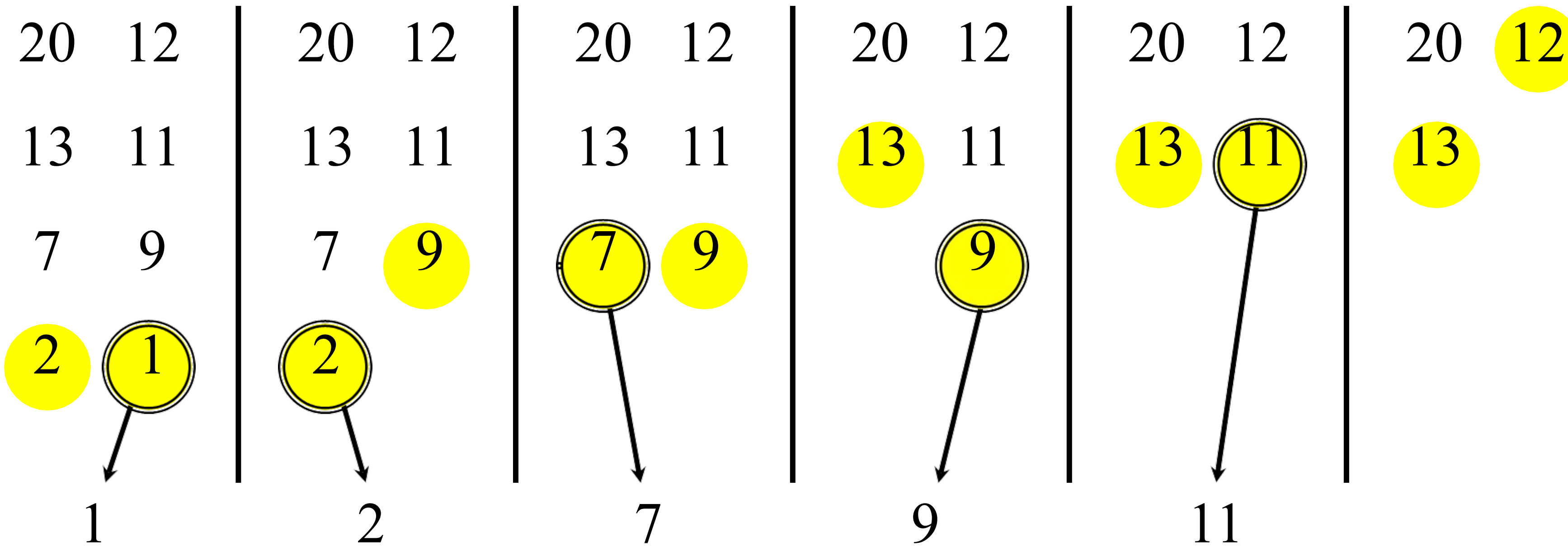
Merging two sorted arrays

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

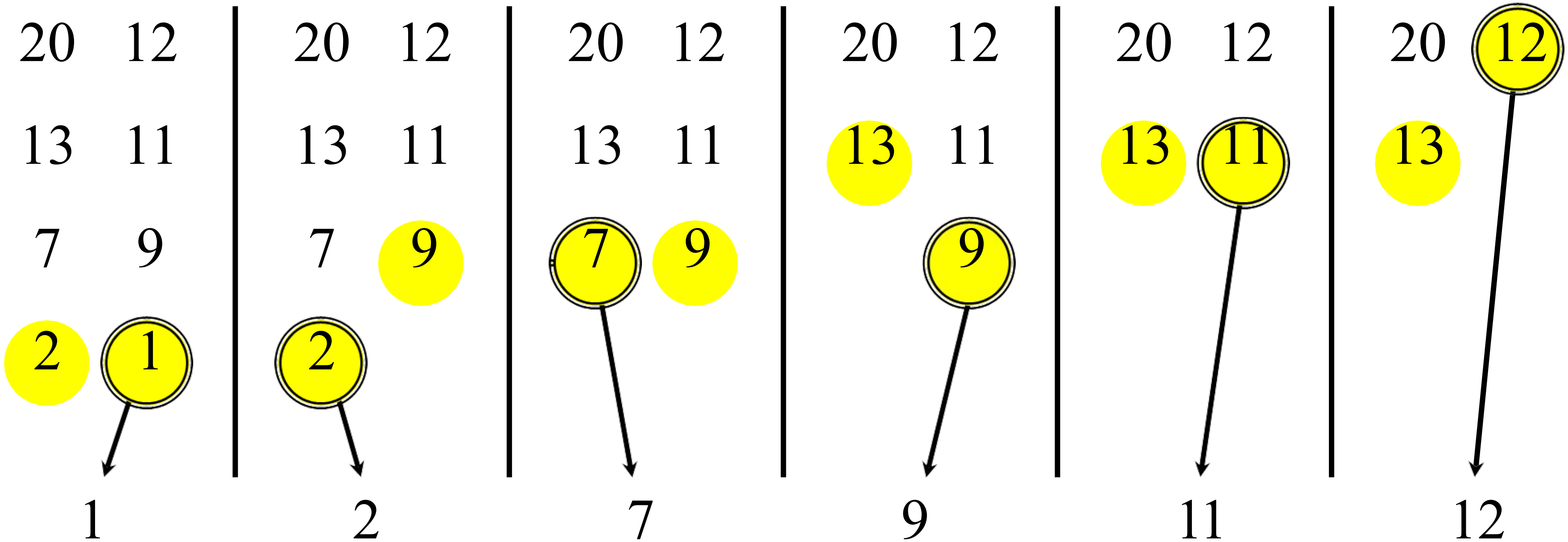
Merging two sorted arrays



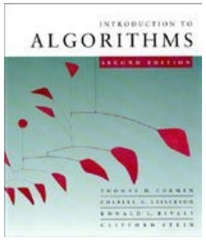
Merging two sorted arrays



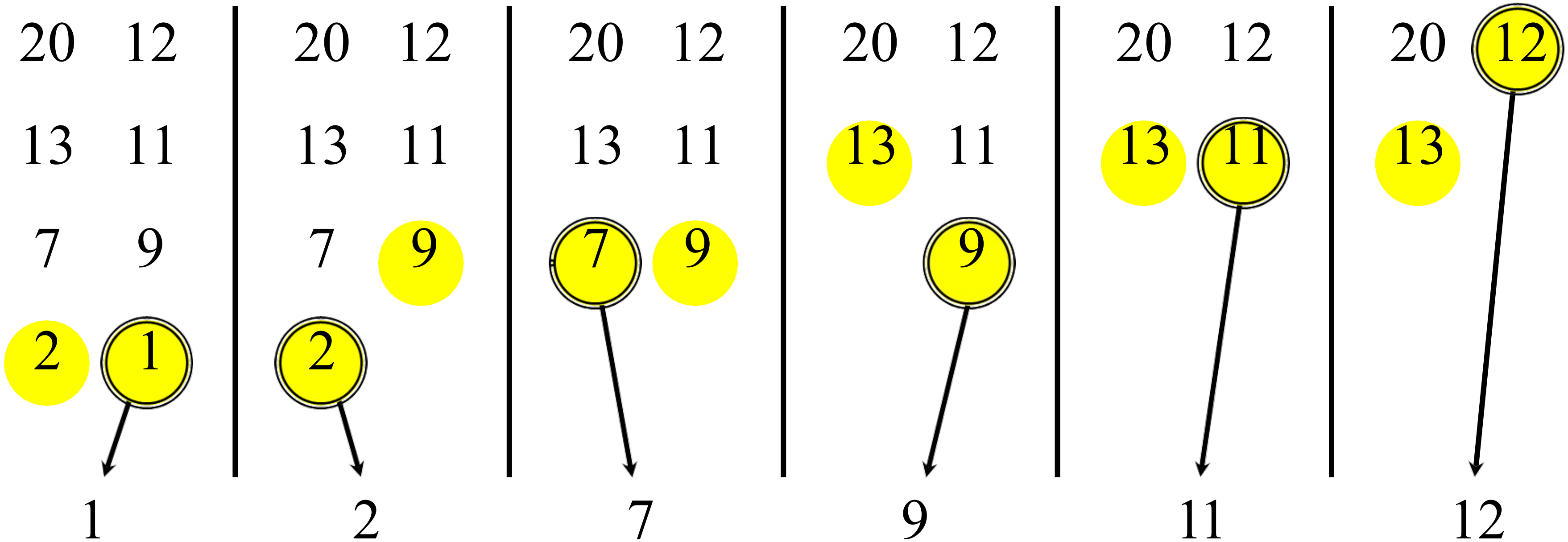
Merging two sorted arrays



Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

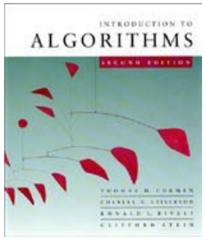


Merging two sorted arrays



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

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



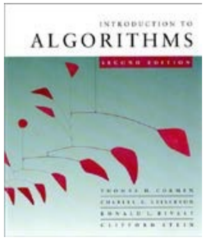
Analyzing merge sort

$T(n)$	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.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Recurrence for merge sort

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

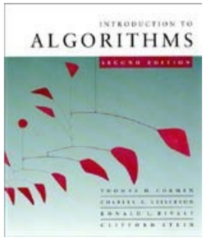
- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n)$.

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Recursion tree

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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Recursion tree

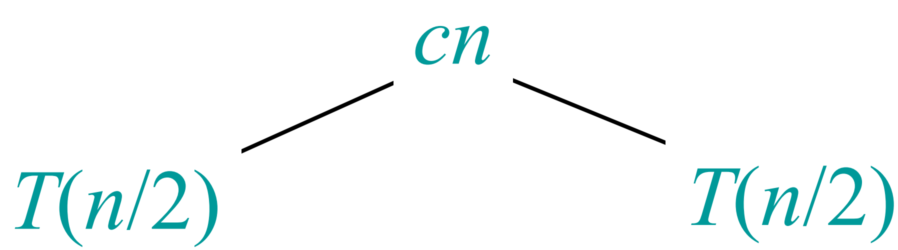
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Recursion tree

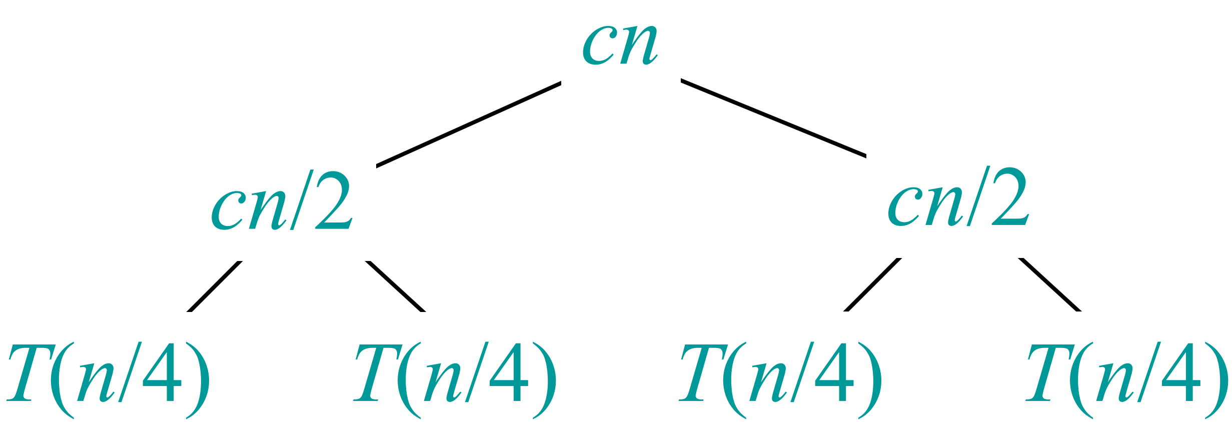
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



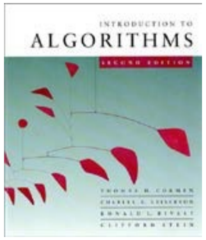
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Recursion tree

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

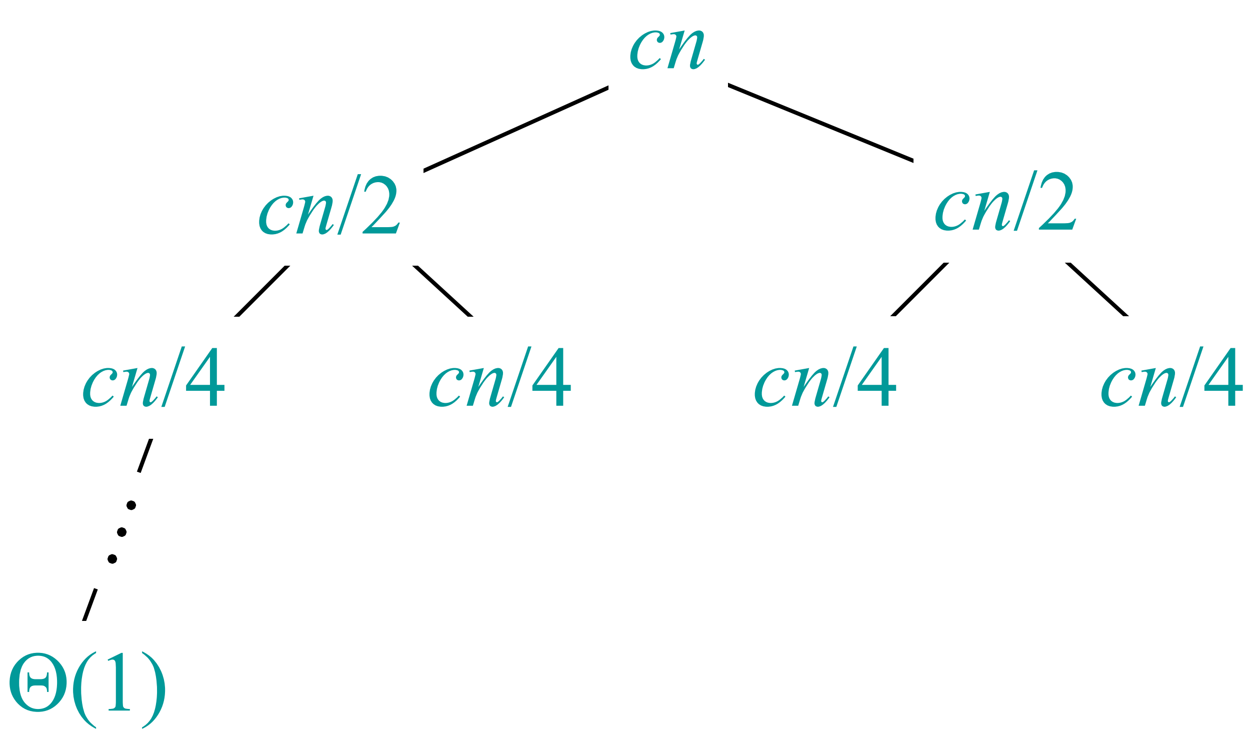


 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

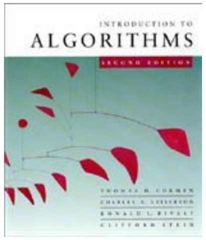


Recursion tree

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

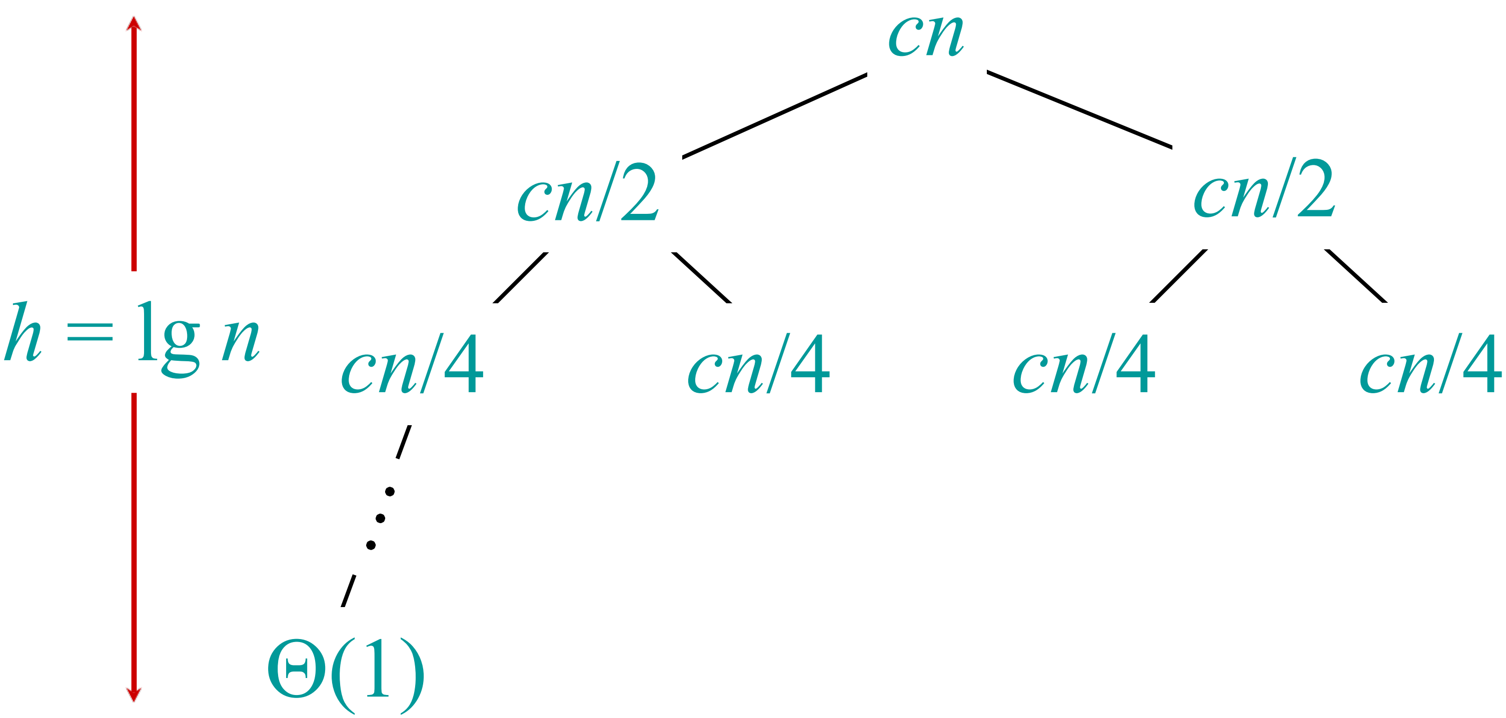


Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

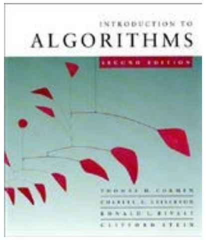


Recursion tree

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

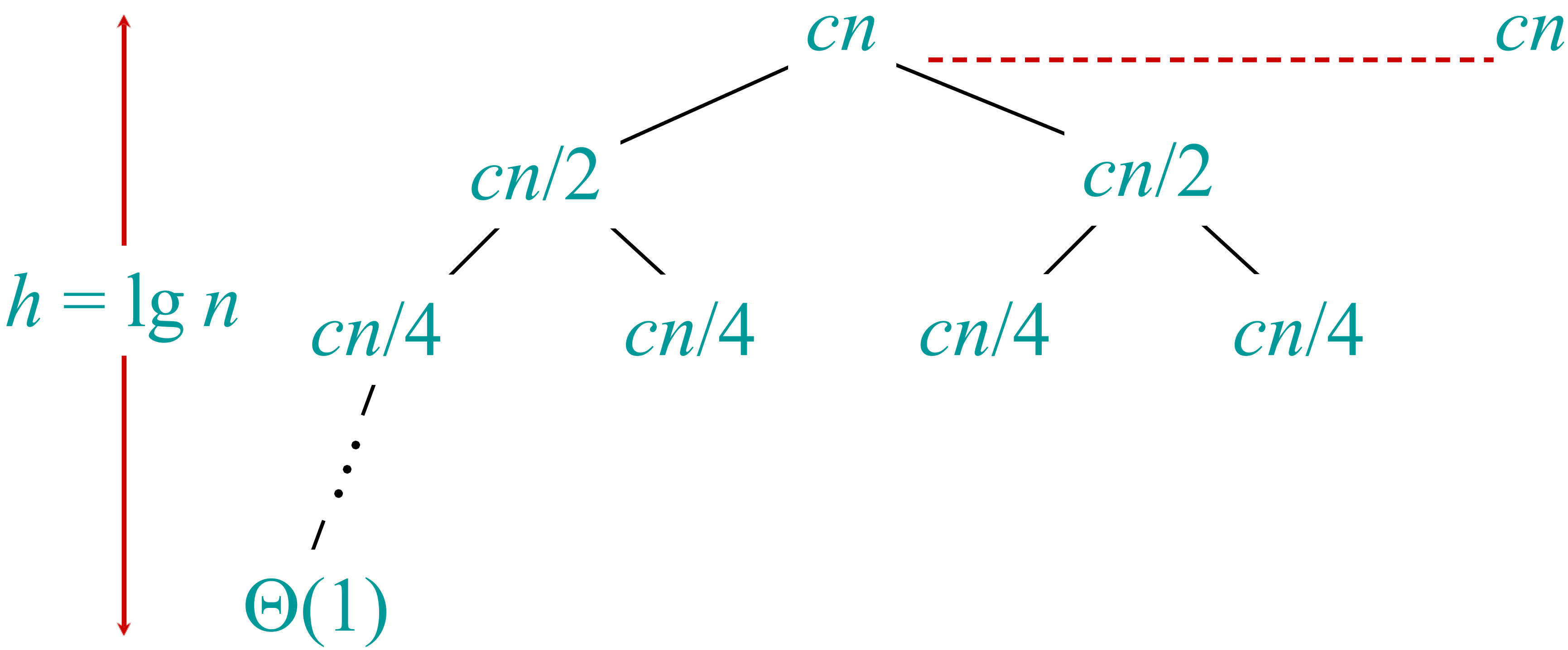


Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

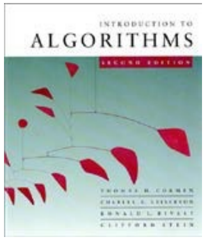


Recursion tree

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

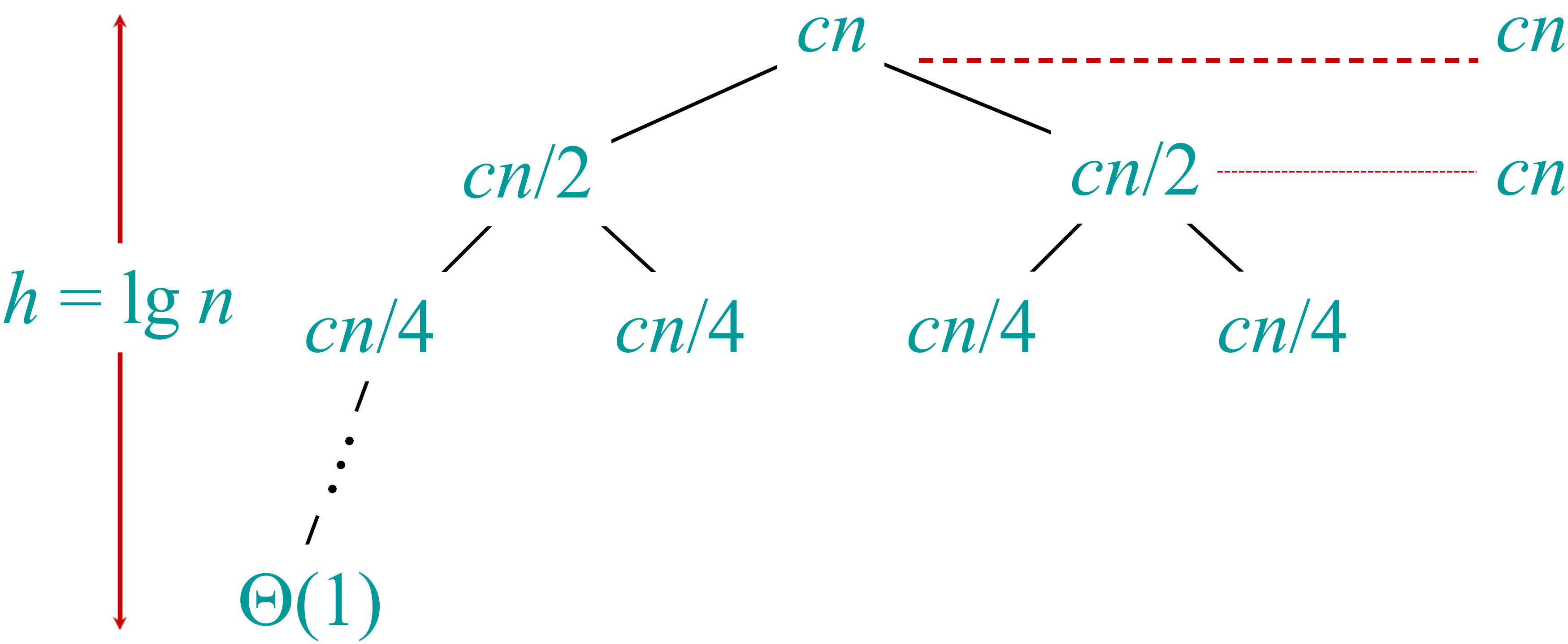


Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

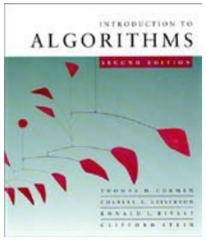


Recursion tree

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

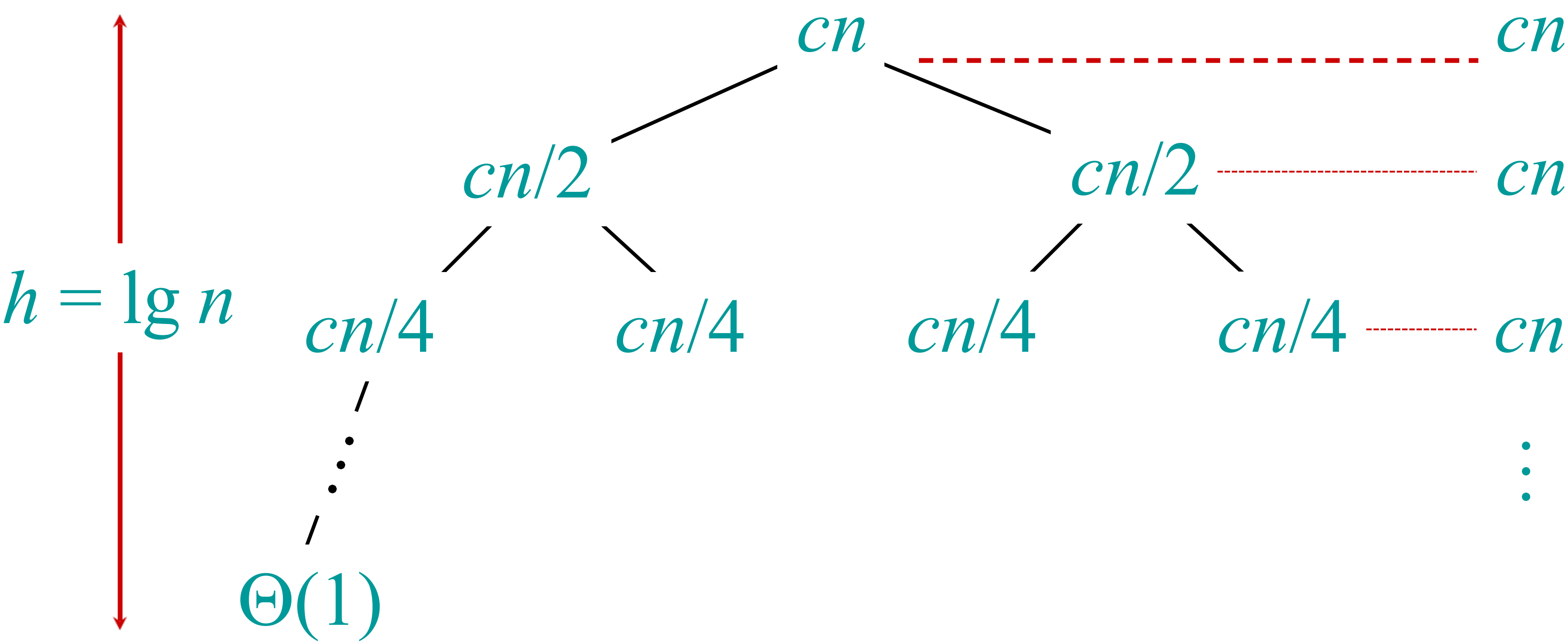


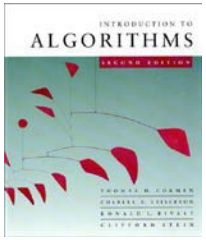
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Recursion tree

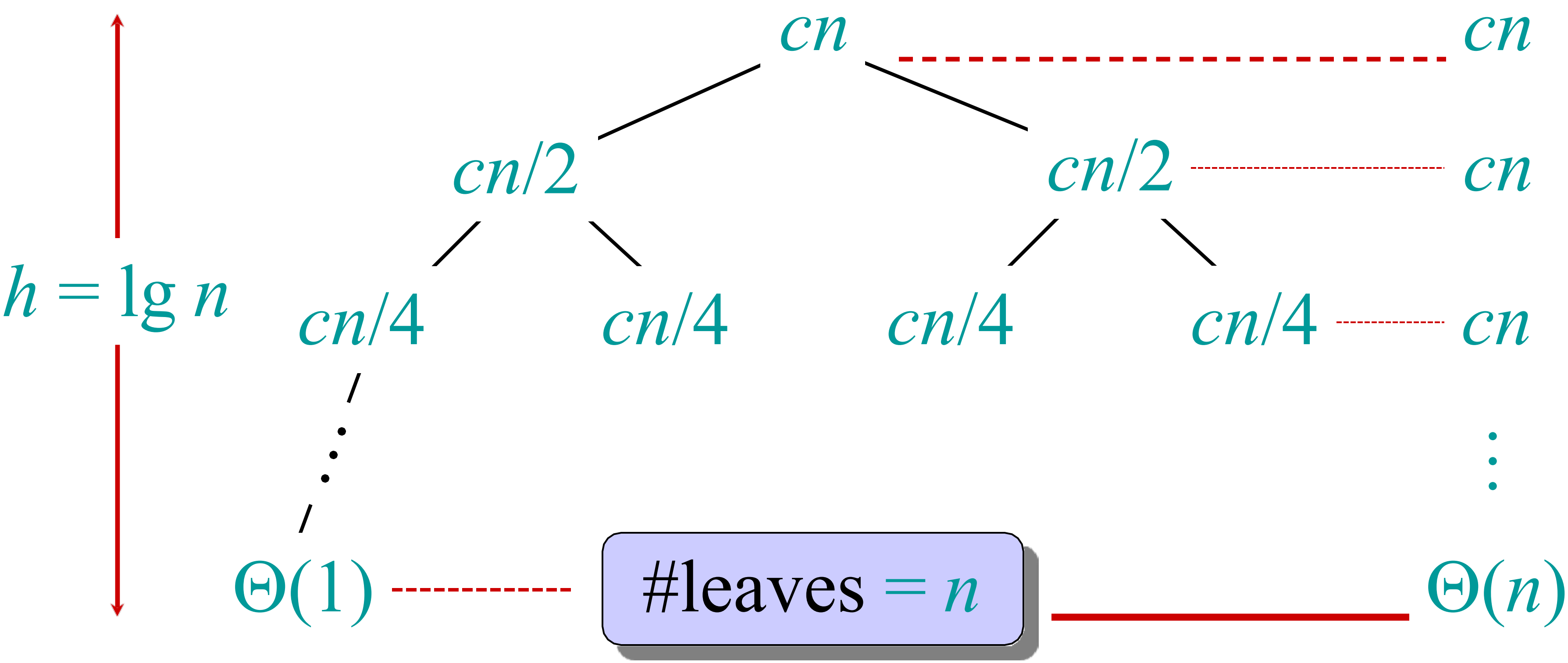
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



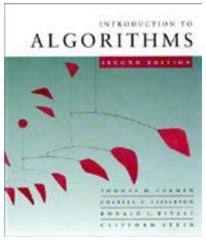


Recursion tree

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

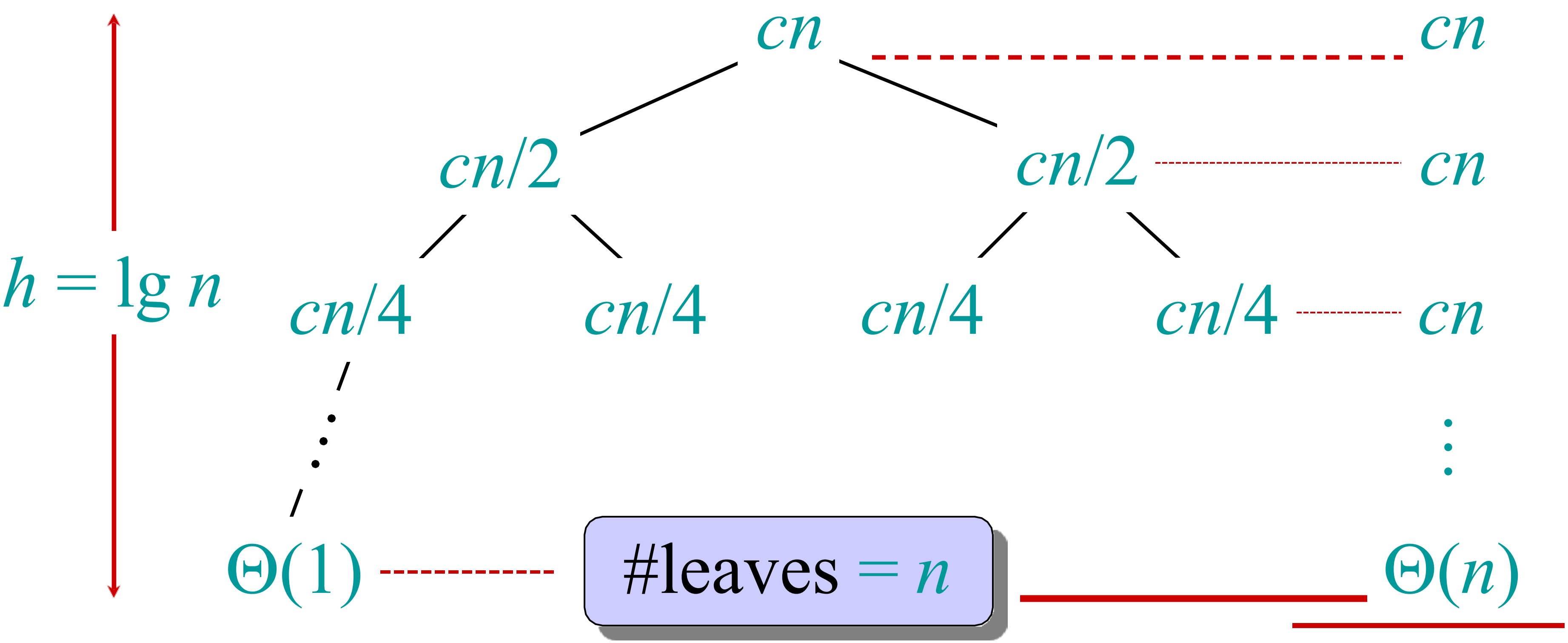


Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



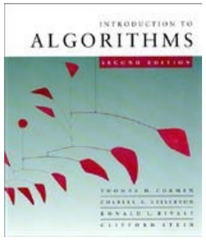
Recursion tree

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



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

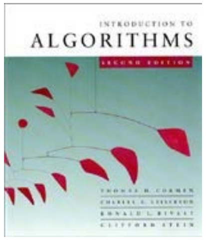
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



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!

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



References

- Big-O Cheat Sheet: https://www.bigocheatsheet.com