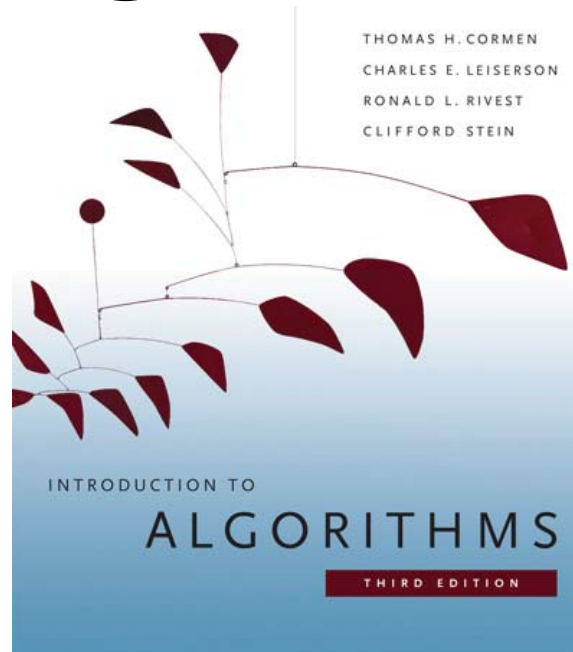


# 6.006- *Introduction to Algorithms*



Courtesy of MIT Press. Used with permission.

## *Lecture 3*

# Menu

- Sorting!
  - Insertion Sort
  - Merge Sort
- Solving Recurrences

# The problem of sorting

***Input:*** array  $A[1..n]$  of numbers.

***Output:*** permutation  $B[1..n]$  of  $A$  such that  $B[1] \leq B[2] \leq \dots \leq B[n]$ .

e.g.  $A = [7, 2, 5, 5, 9.6] \rightarrow B = [2, 5, 5, 7, 9.6]$

How can we do it efficiently ?

# Why Sorting?

- Obvious applications
  - Organize an MP3 library
  - Maintain a telephone directory
- Problems that become easy once items are in sorted order
  - Find a median, or find closest pairs
  - Binary search, identify statistical outliers
- Non-obvious applications
  - Data compression: sorting finds duplicates
  - Computer graphics: rendering scenes front to back

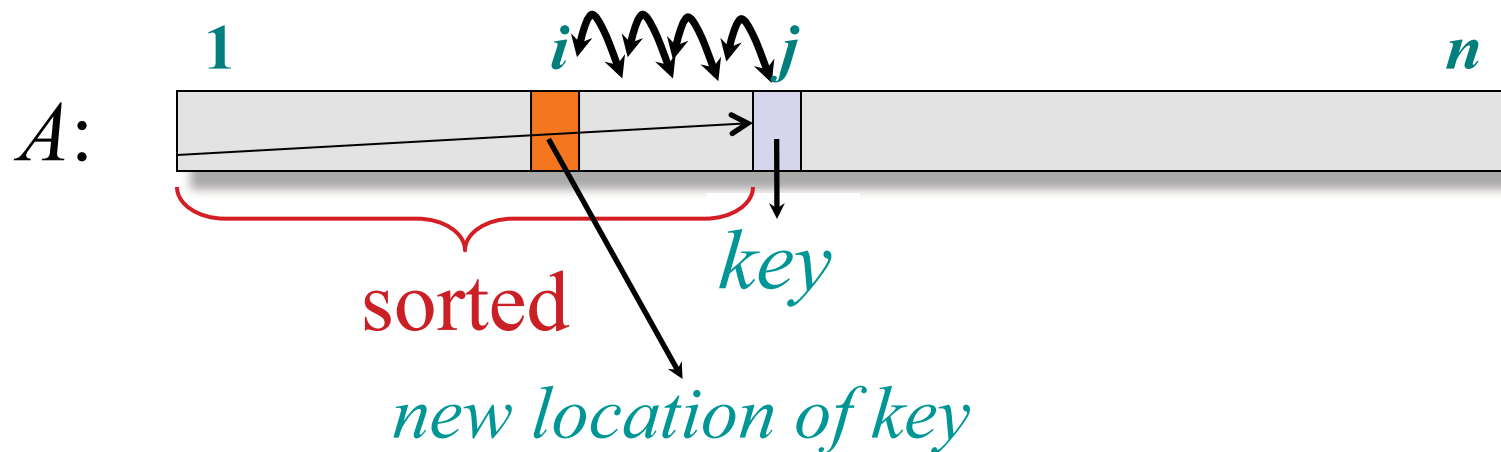
# Insertion sort

**INSERTION-SORT** ( $A, n$ )  $\triangleright A[1 \dots n]$

for  $j \leftarrow 2$  to  $n$

insert key  $A[j]$  into the (already sorted) sub-array  $A[1 \dots j-1]$ .  
by pairwise key-swaps down to its right position

**Illustration of iteration  $j$**



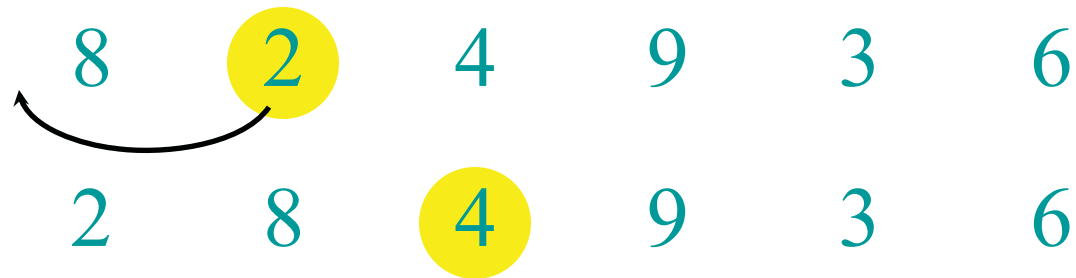
# Example of insertion sort

8 2 4 9 3 6

# Example of insertion sort



# Example of insertion sort

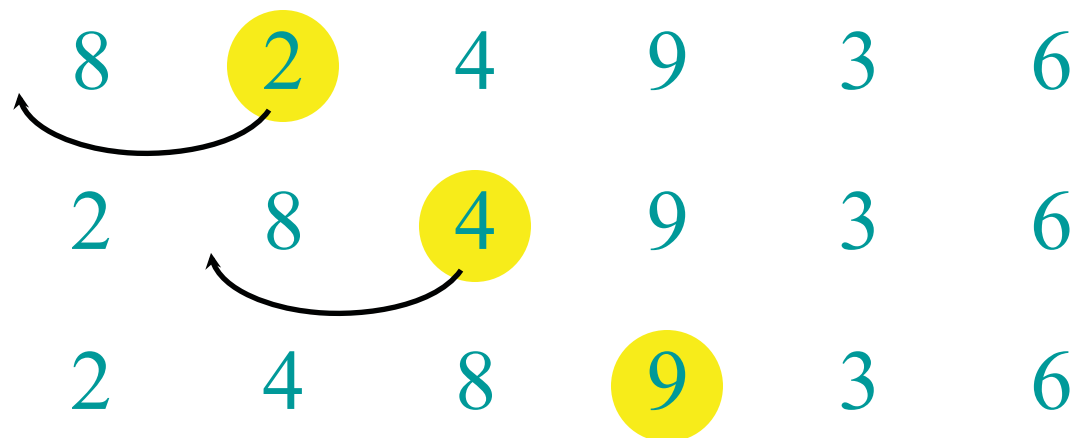




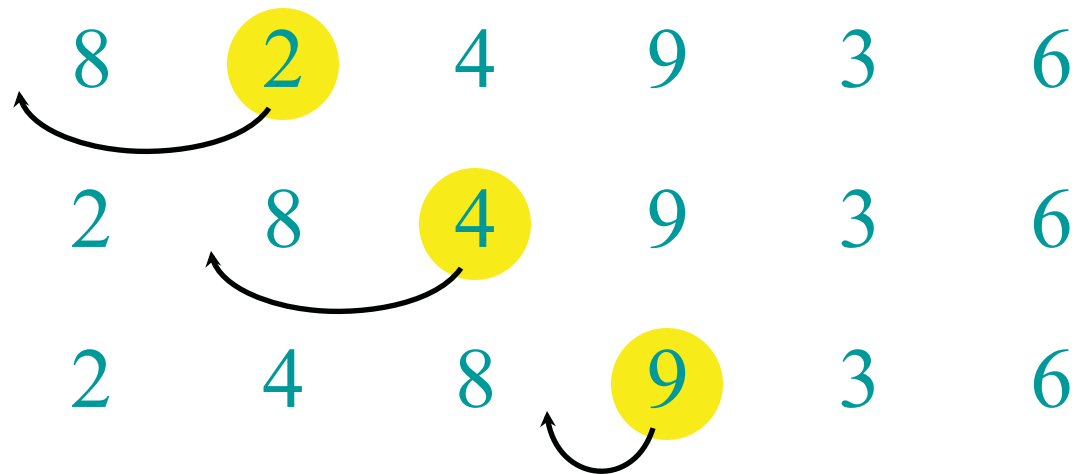
# Example of insertion sort



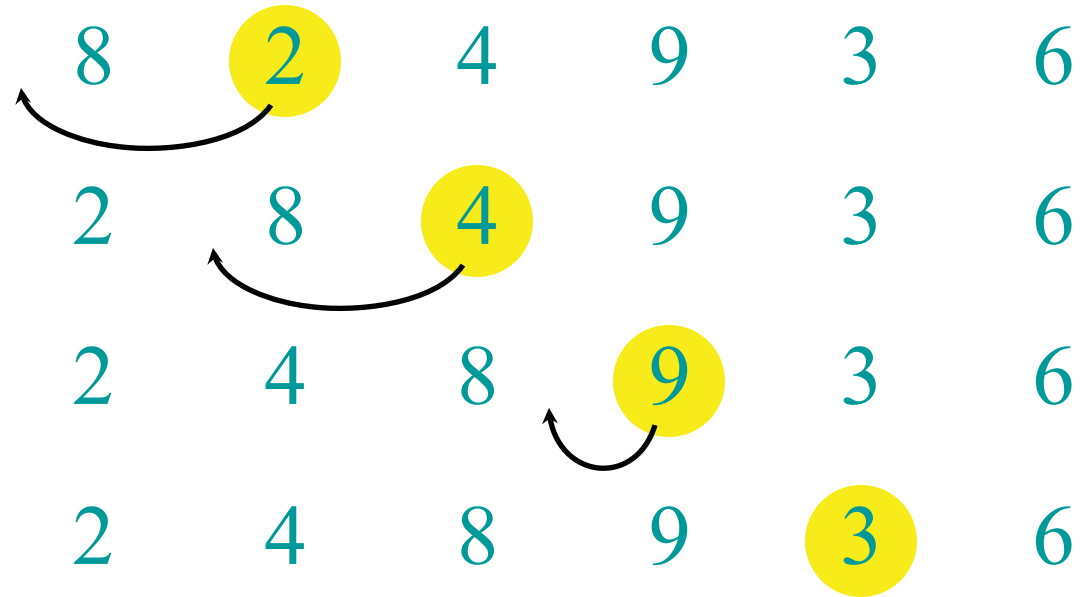
# Example of insertion sort



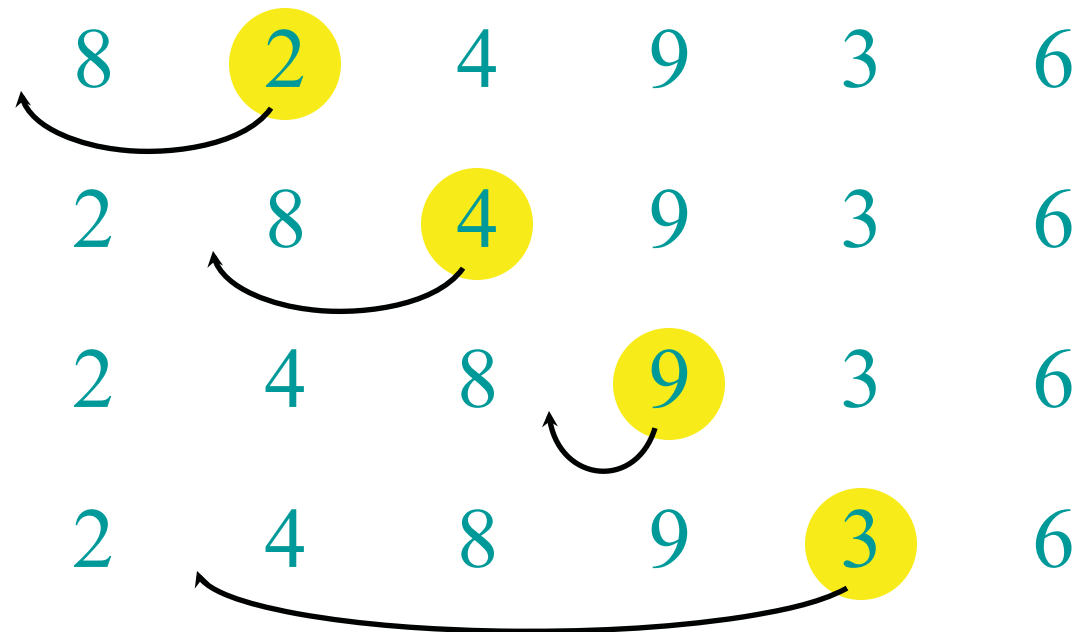
# Example of insertion sort



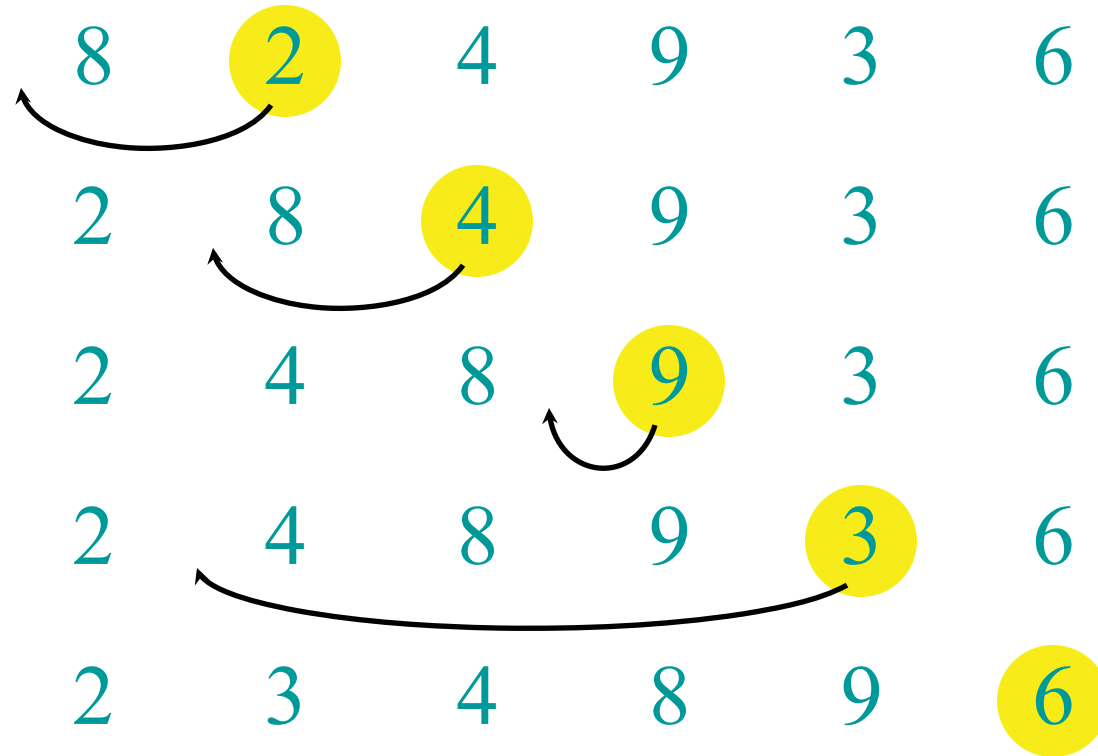
# Example of insertion sort



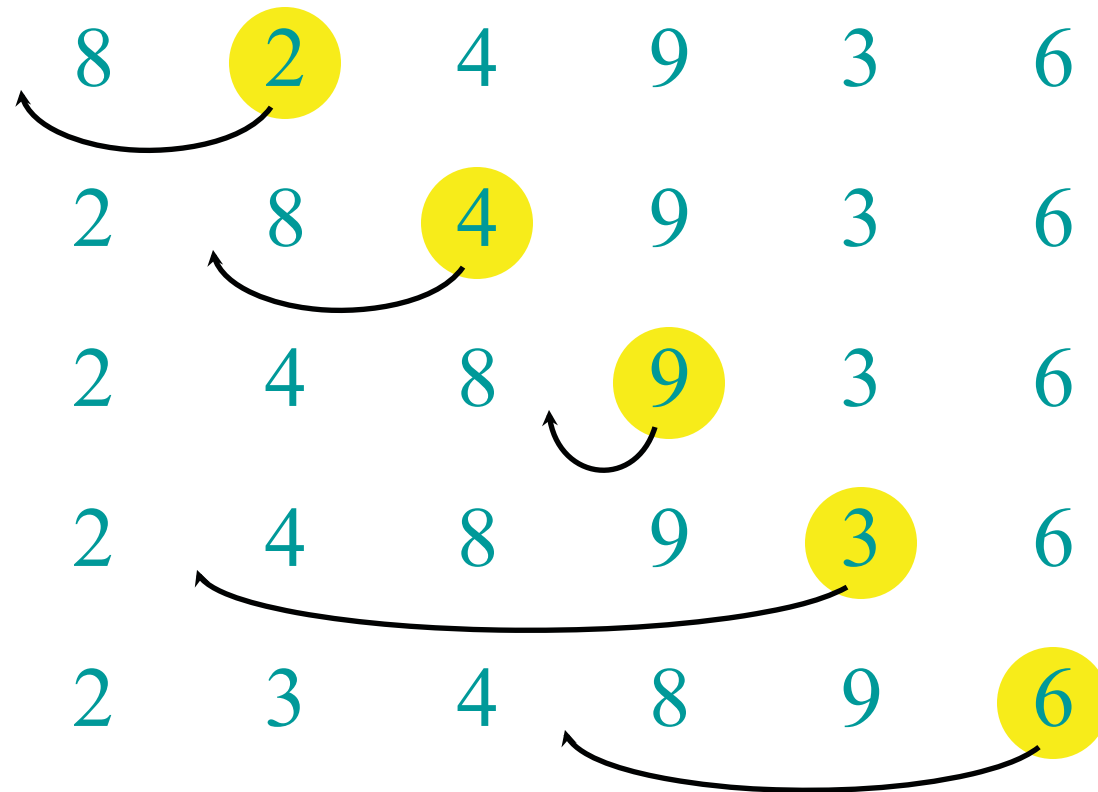
# Example of insertion sort



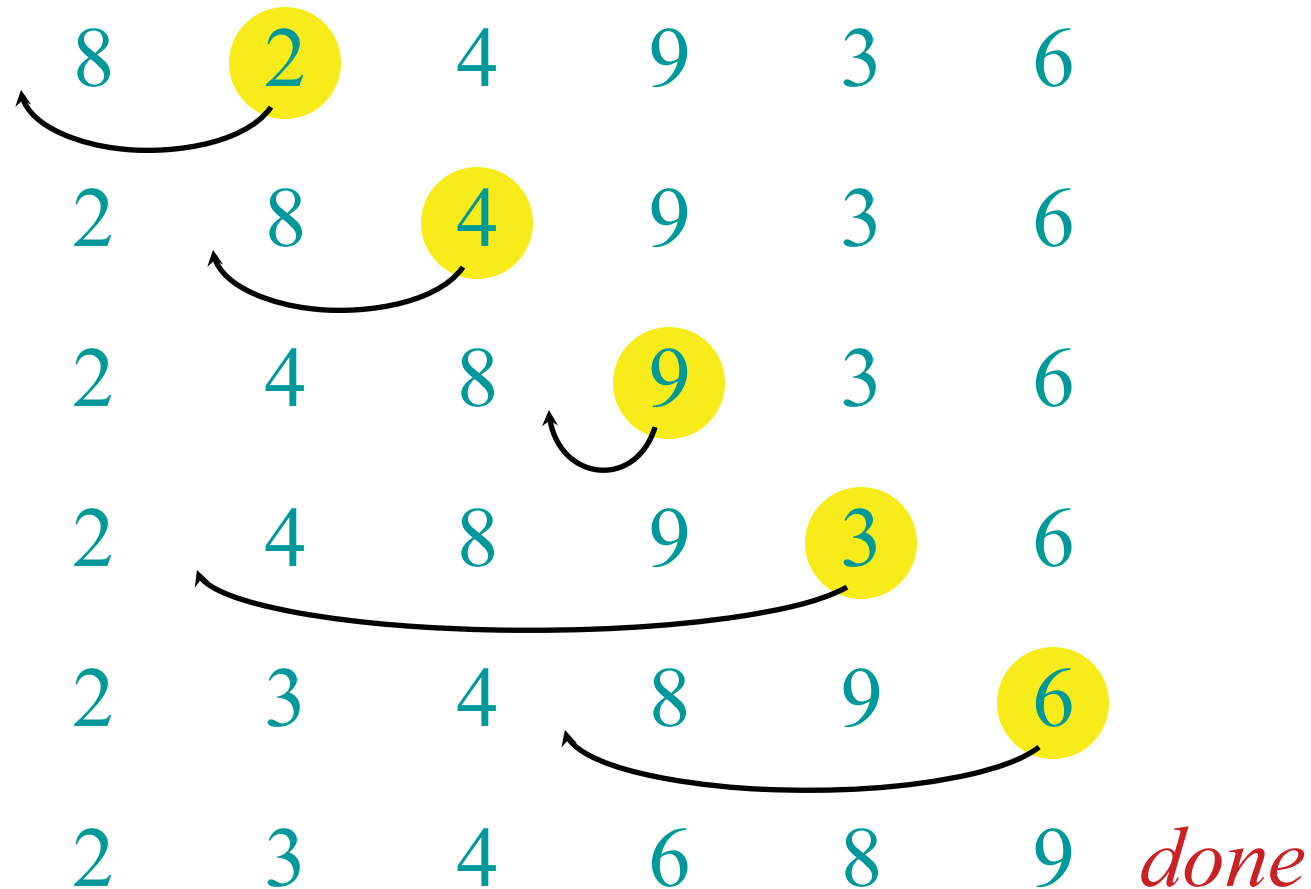
# Example of insertion sort



# Example of insertion sort



# Example of insertion sort



Running time?  $\Theta(n^2)$  because  $\Theta(n^2)$  compares and  $\Theta(n^2)$  swaps  
e.g. when input is  $A = [n, n - 1, n - 2, \dots, 2, 1]$



# Binary Insertion sort

**BINARY-INSERTION-SORT** ( $A, n$ )       $\triangleright A[1 \dots n]$   
  for  $j \leftarrow 2$  to  $n$   
    insert key  $A[j]$  into the (already sorted) sub-array  $A[1 \dots j-1]$ .  
    Use binary search to find the right position

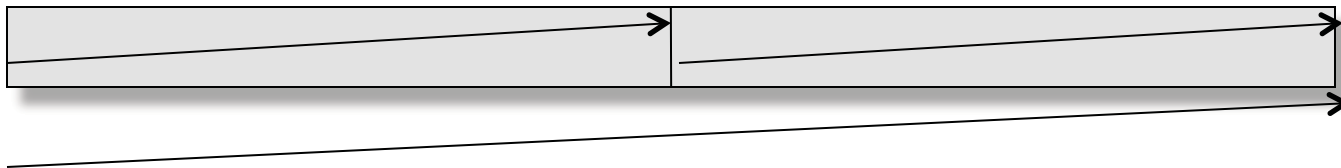
Binary search with take  $\Theta(\log n)$  time.

However, shifting the elements after insertion will still take  $\Theta(n)$  time.

Complexity:  $\Theta(n \log n)$  comparisons  
               $(n^2)$  swaps

# Meet Merge Sort

- divide and conquer
- MERGE-SORT**  $A[1 \dots n]$
1. If  $n = 1$ , done (nothing to sort).
  2. Otherwise, recursively sort  $A[1 \dots n/2]$  and  $A[n/2+1 \dots n]$ .
  3. “*Merge*” the two sorted sub-arrays.



*Key subroutine:* **MERGE**

# Merging two sorted arrays

20 12

13 11

7 9

2 1

# Merging two sorted arrays

20 12

13 11

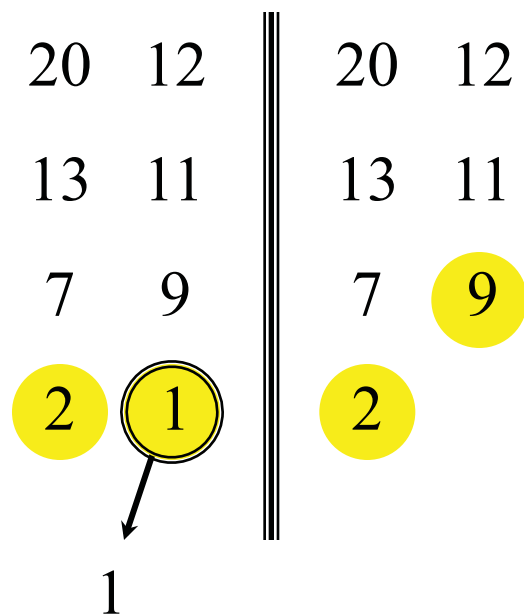
7 9

2 1

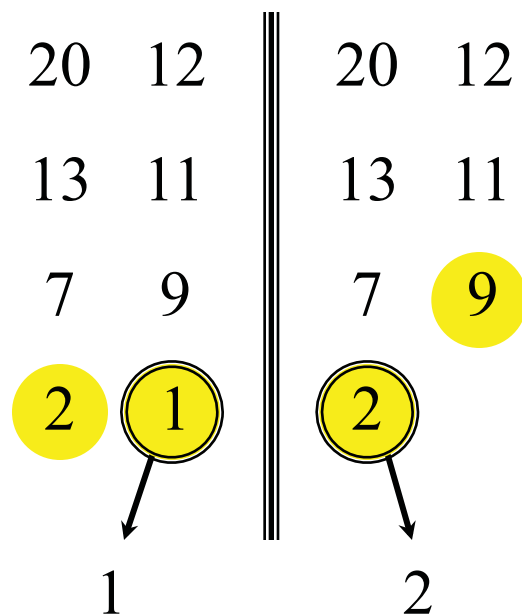
1



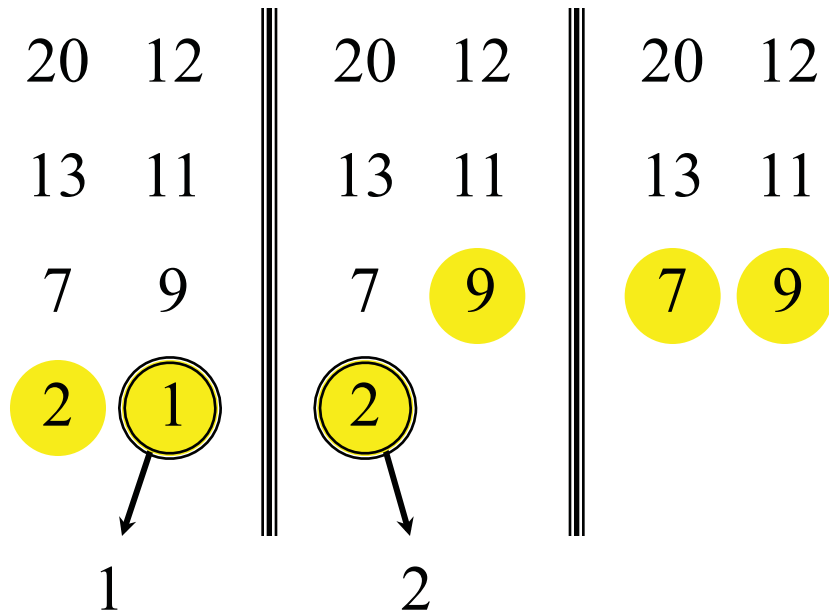
# Merging two sorted arrays



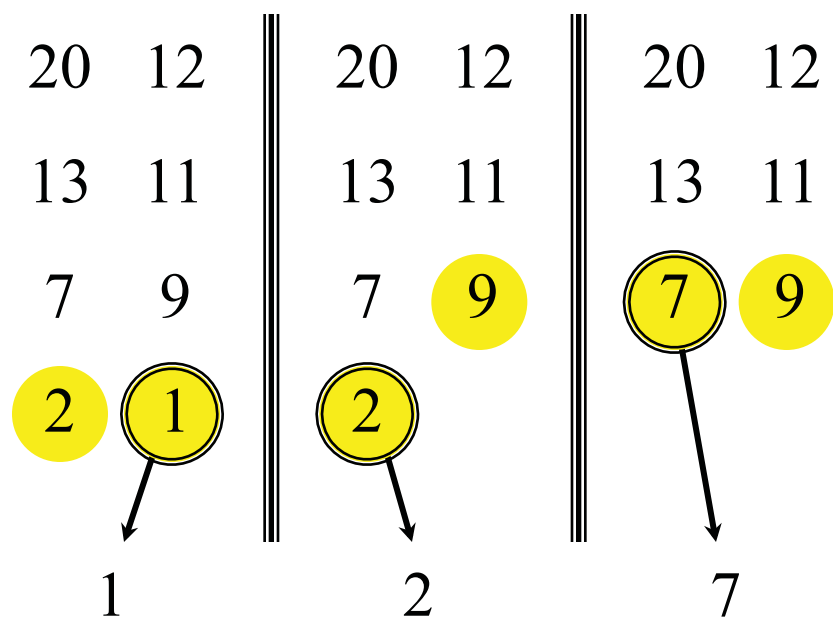
# Merging two sorted arrays



# Merging two sorted arrays

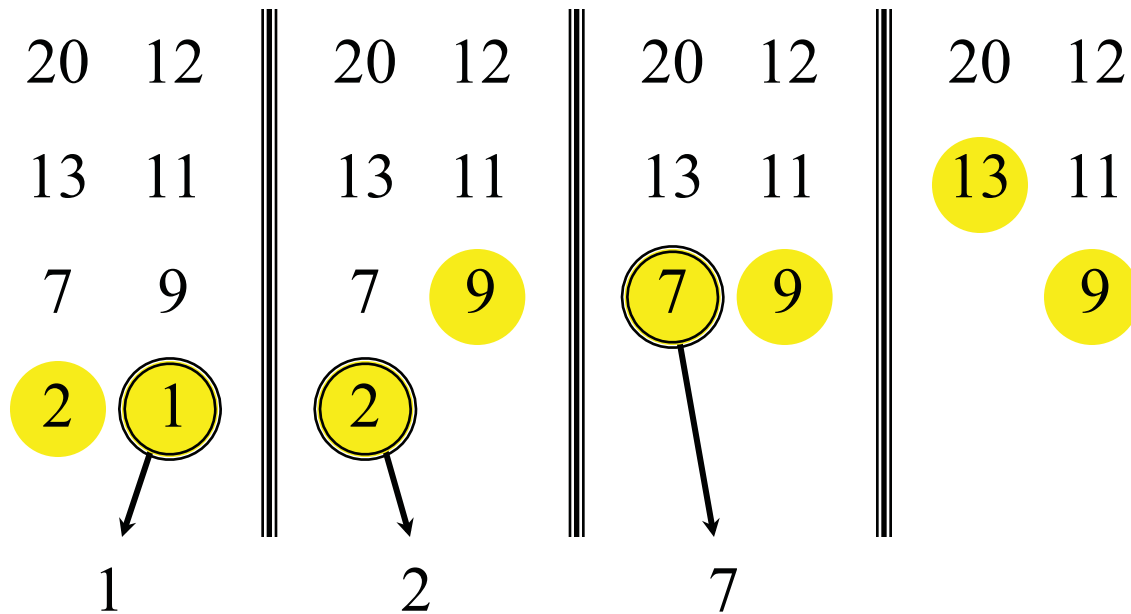


# Merging two sorted arrays

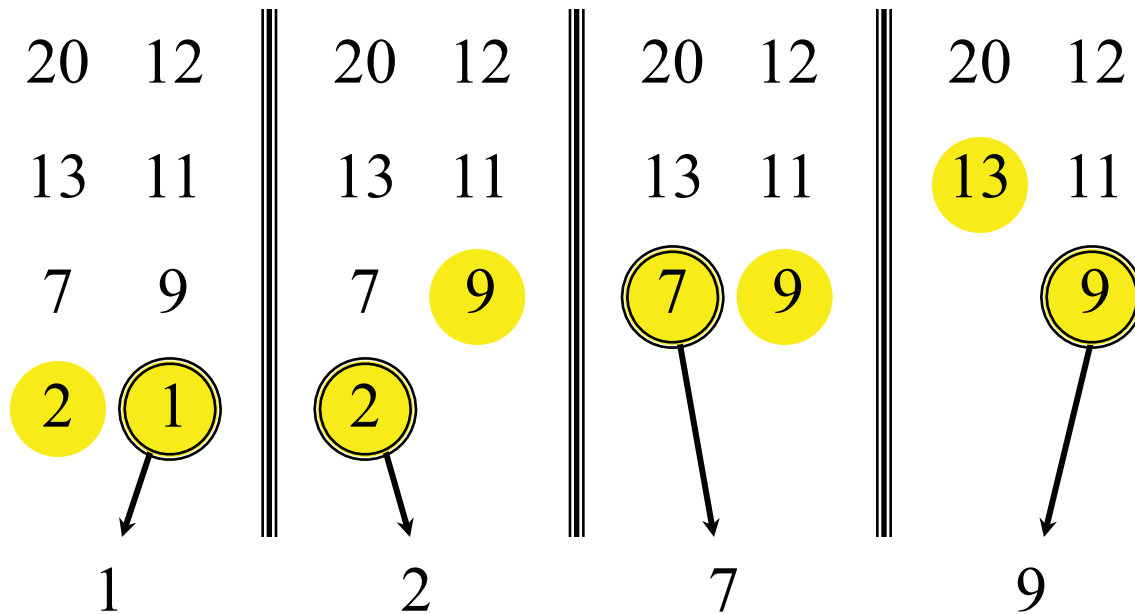




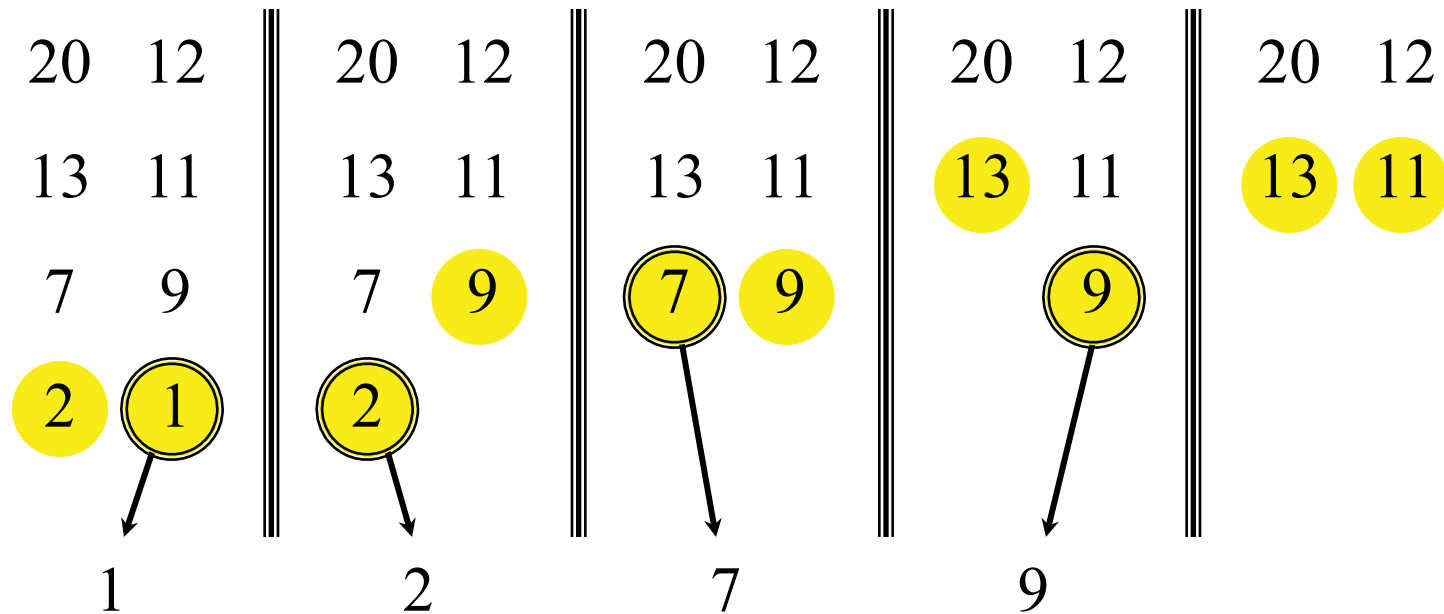
# Merging two sorted arrays



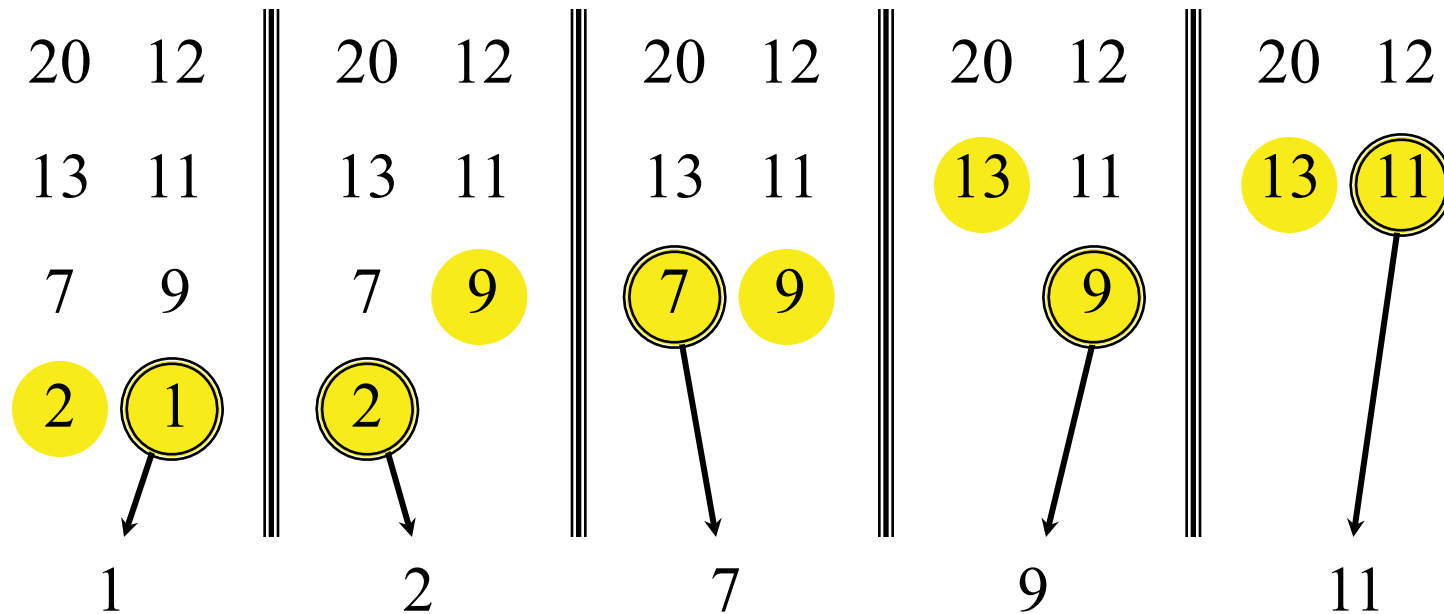
# Merging two sorted arrays



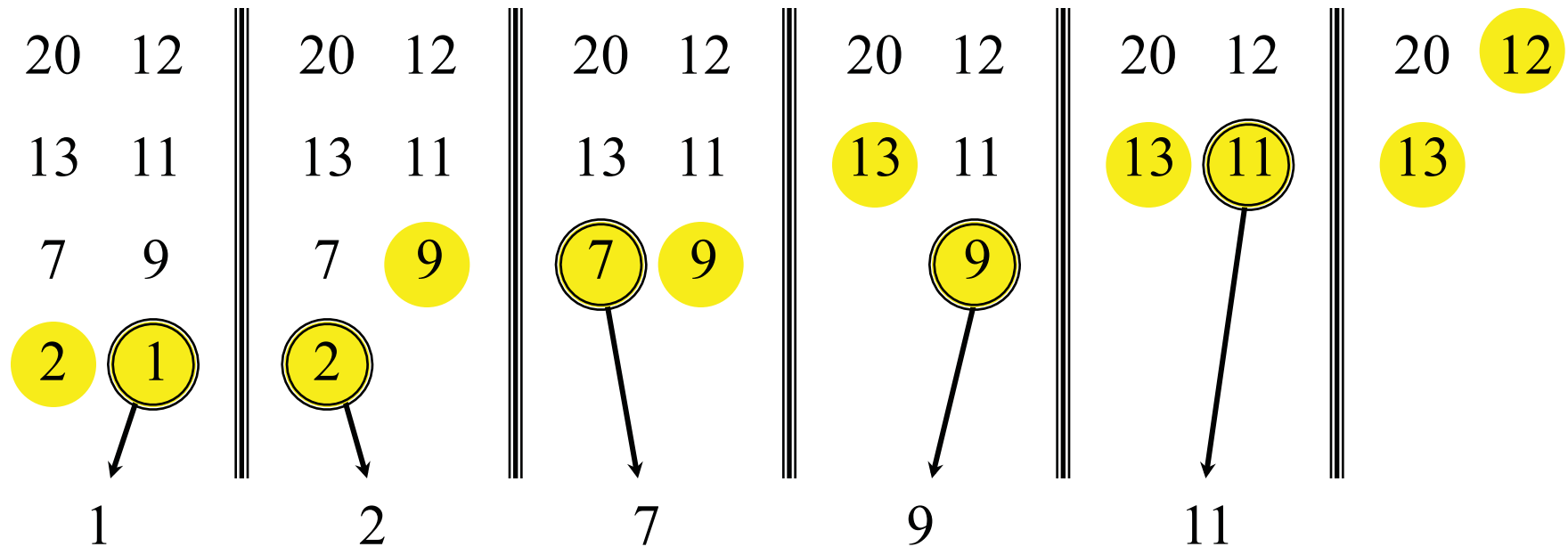
# Merging two sorted arrays



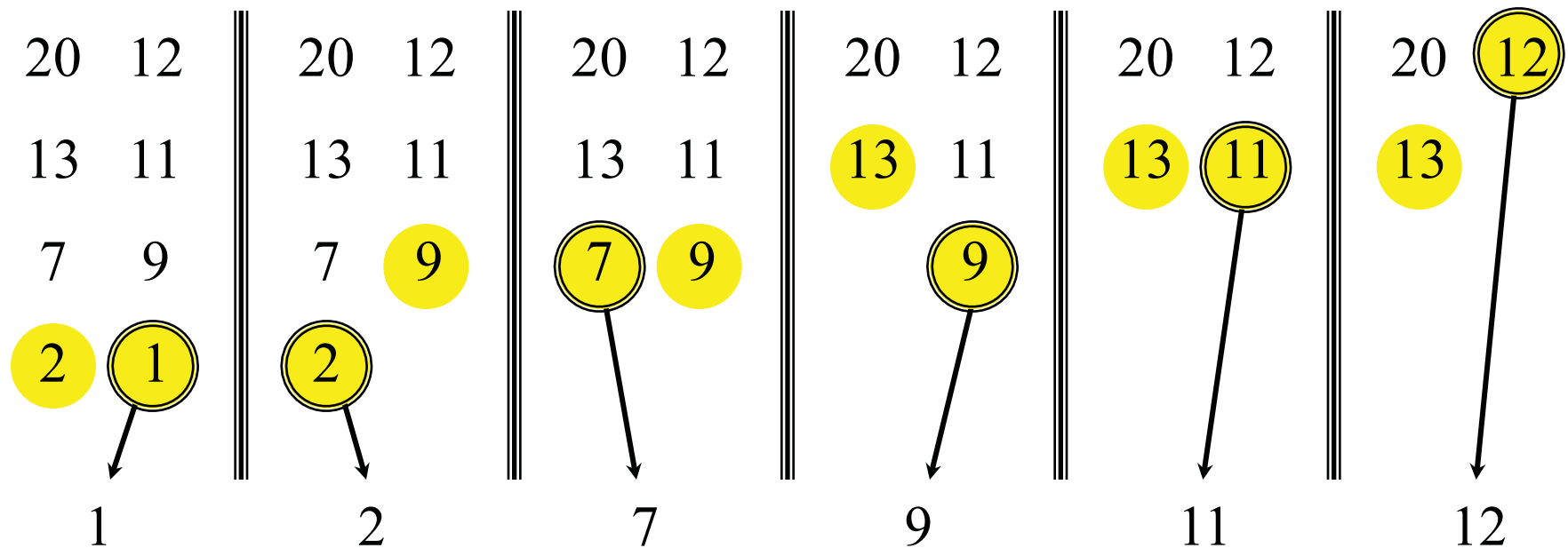
# Merging two sorted arrays



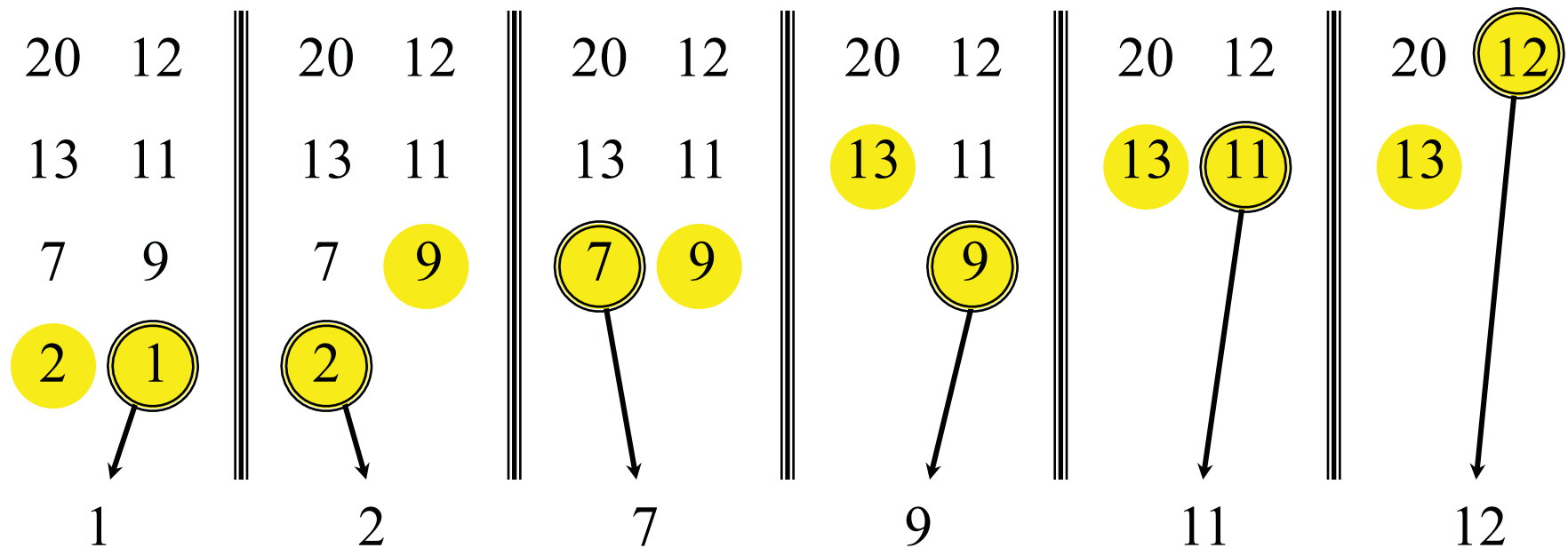
# Merging two sorted arrays



# Merging two sorted arrays



# Merging two sorted arrays



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

# Analyzing merge sort

**MERGE-SORT**  $A[1 \dots n]$

1. If  $n = 1$ , done.

2. Recursively sort  $A[1 \dots \lceil n/2 \rceil]$   
and  $A[\lceil n/2 \rceil + 1 \dots n]$ .

3. *“Merge”* the two sorted lists

$T(n)$

$\Theta(1)$

$2T(n/2)$

$\Theta(n)$

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

$$T(n) = ?$$



# Recurrence solving

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

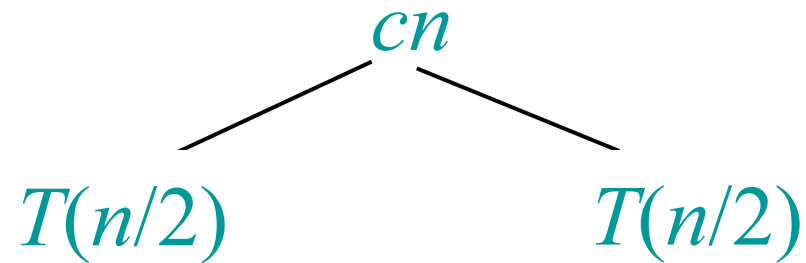
# Recursion tree

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

$$T(n)$$

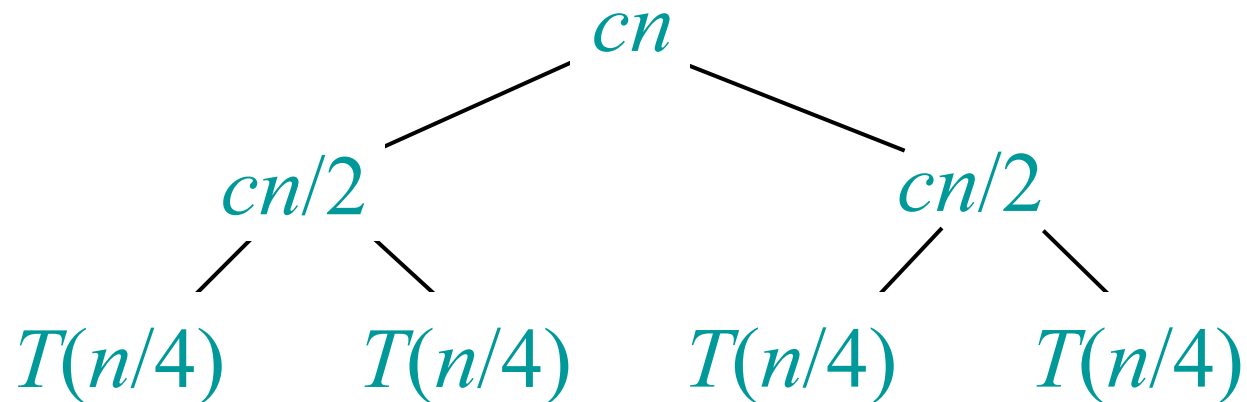
# Recursion tree

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



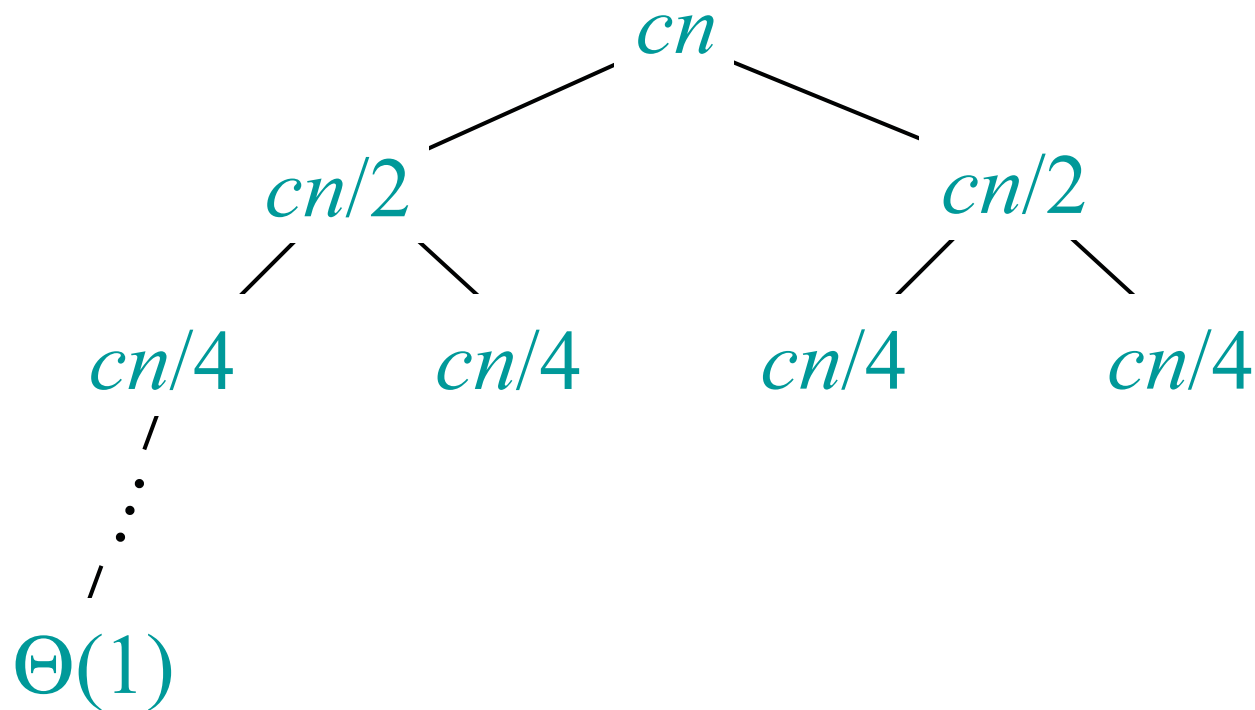
# Recursion tree

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



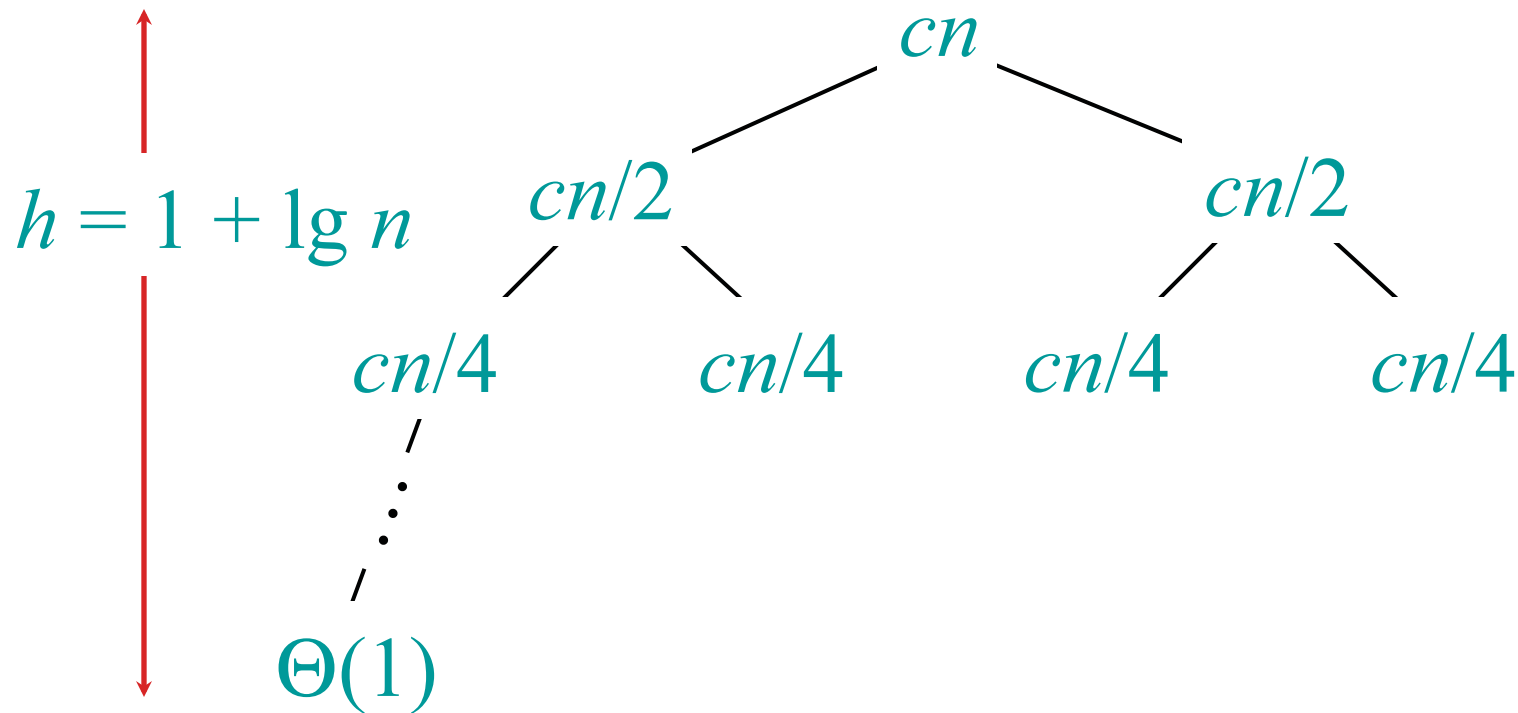
# Recursion tree

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



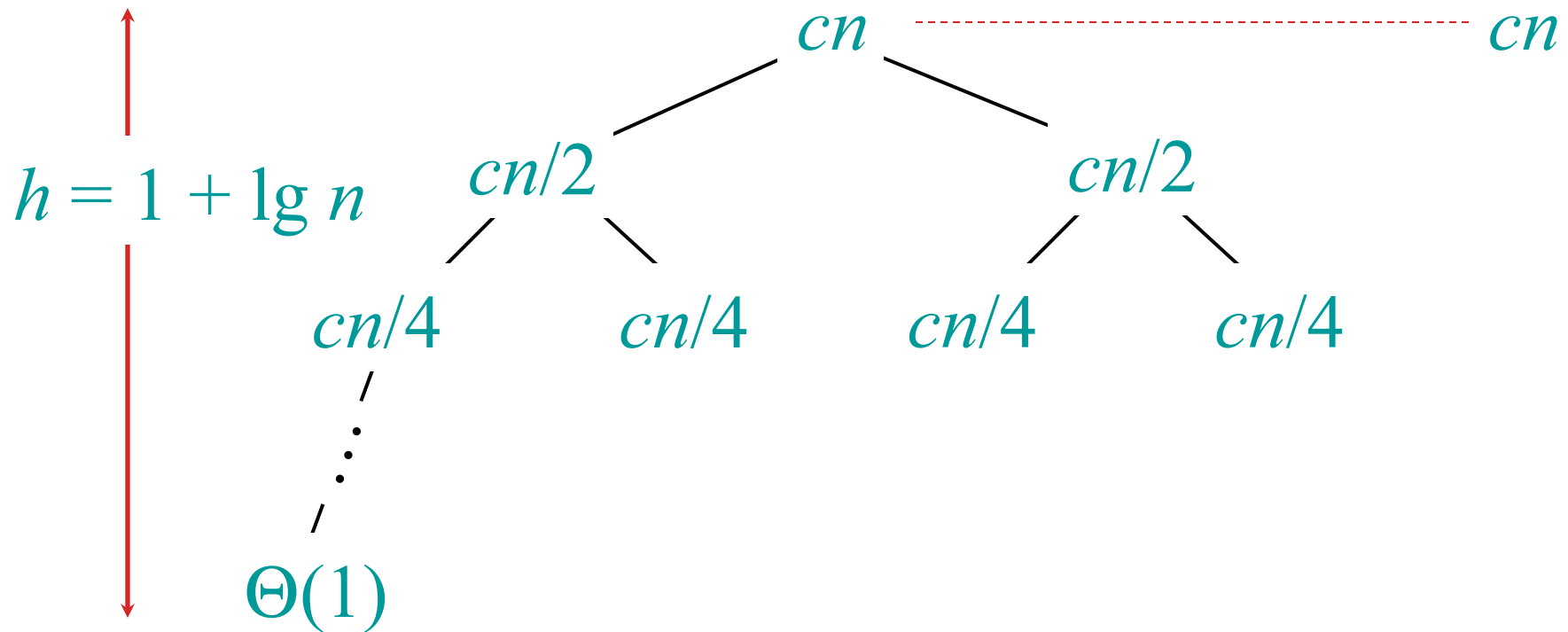
# Recursion tree

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



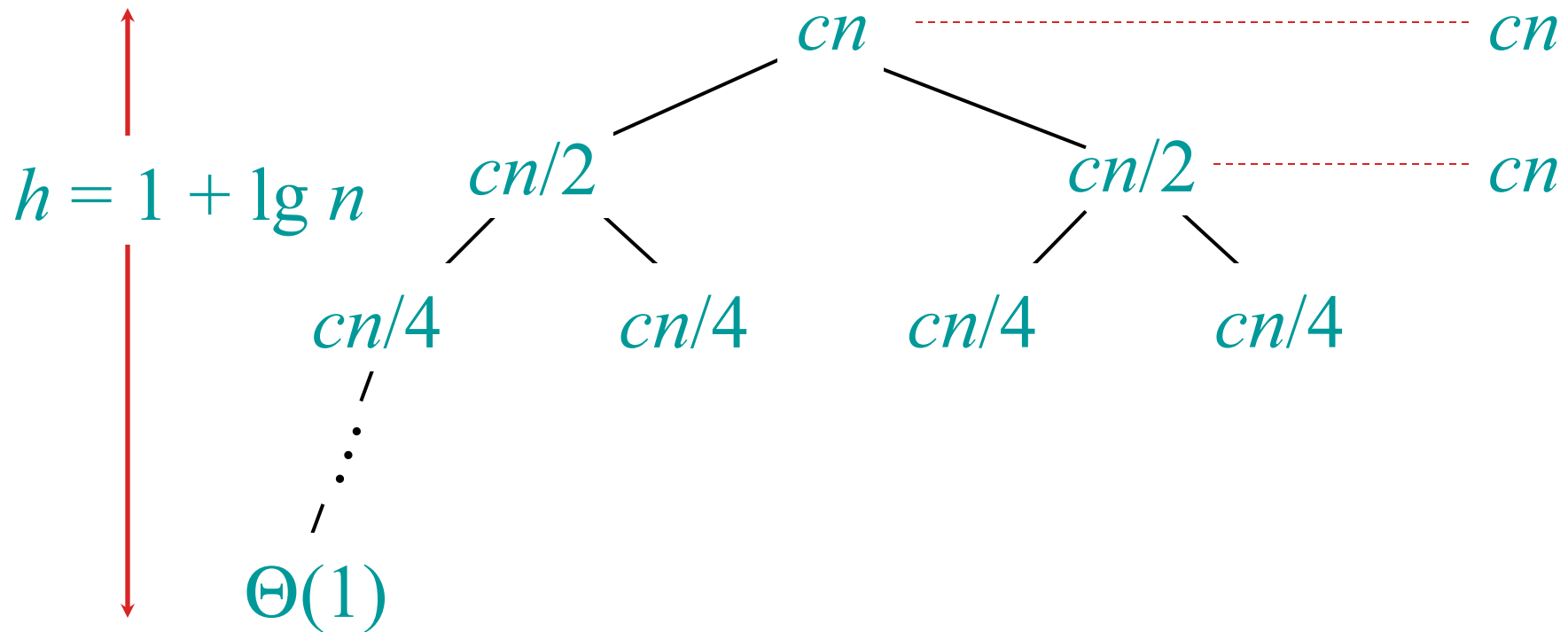
# Recursion tree

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



# Recursion tree

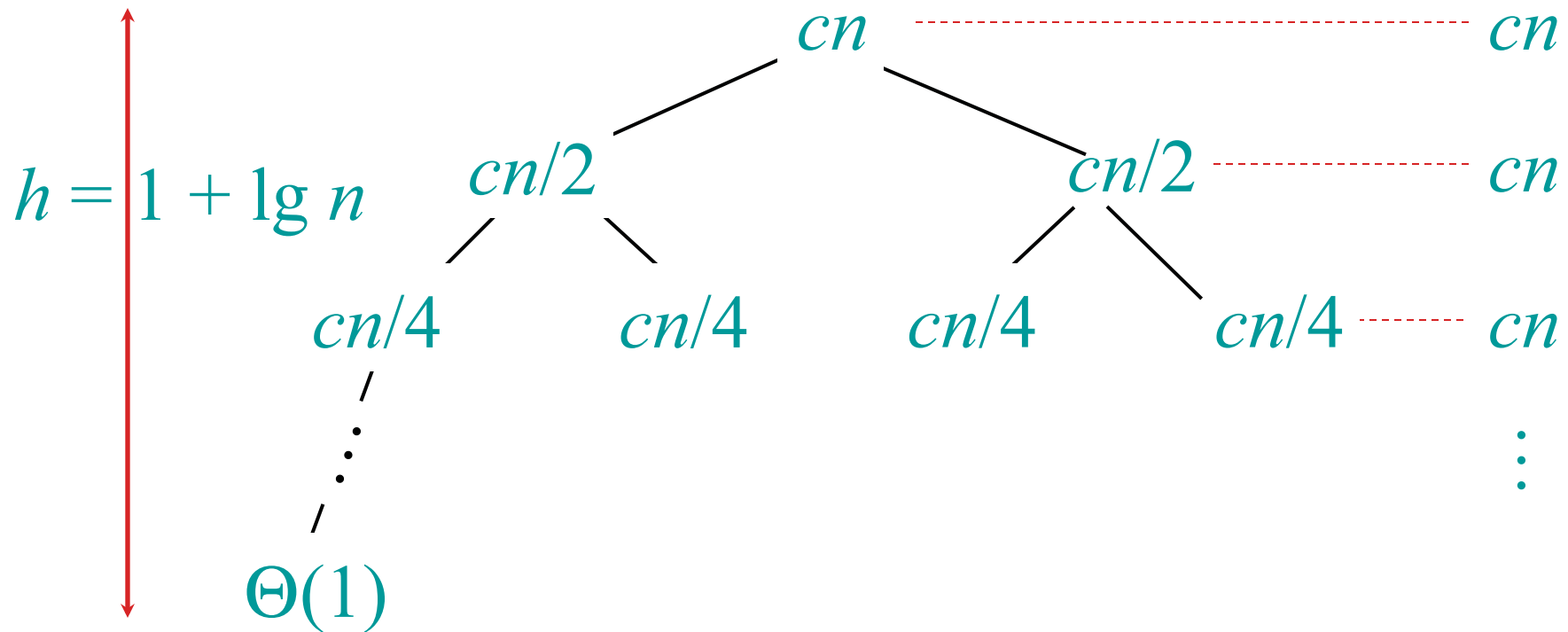
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.





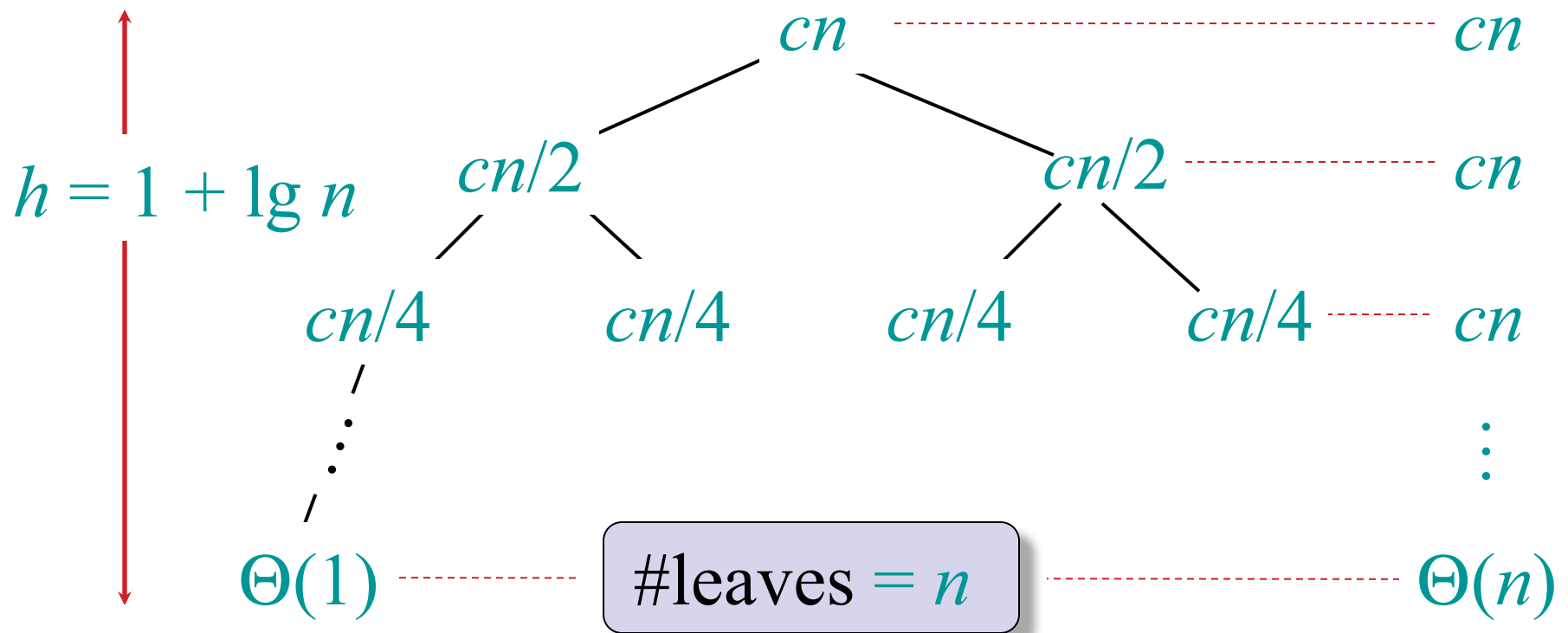
# Recursion tree

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



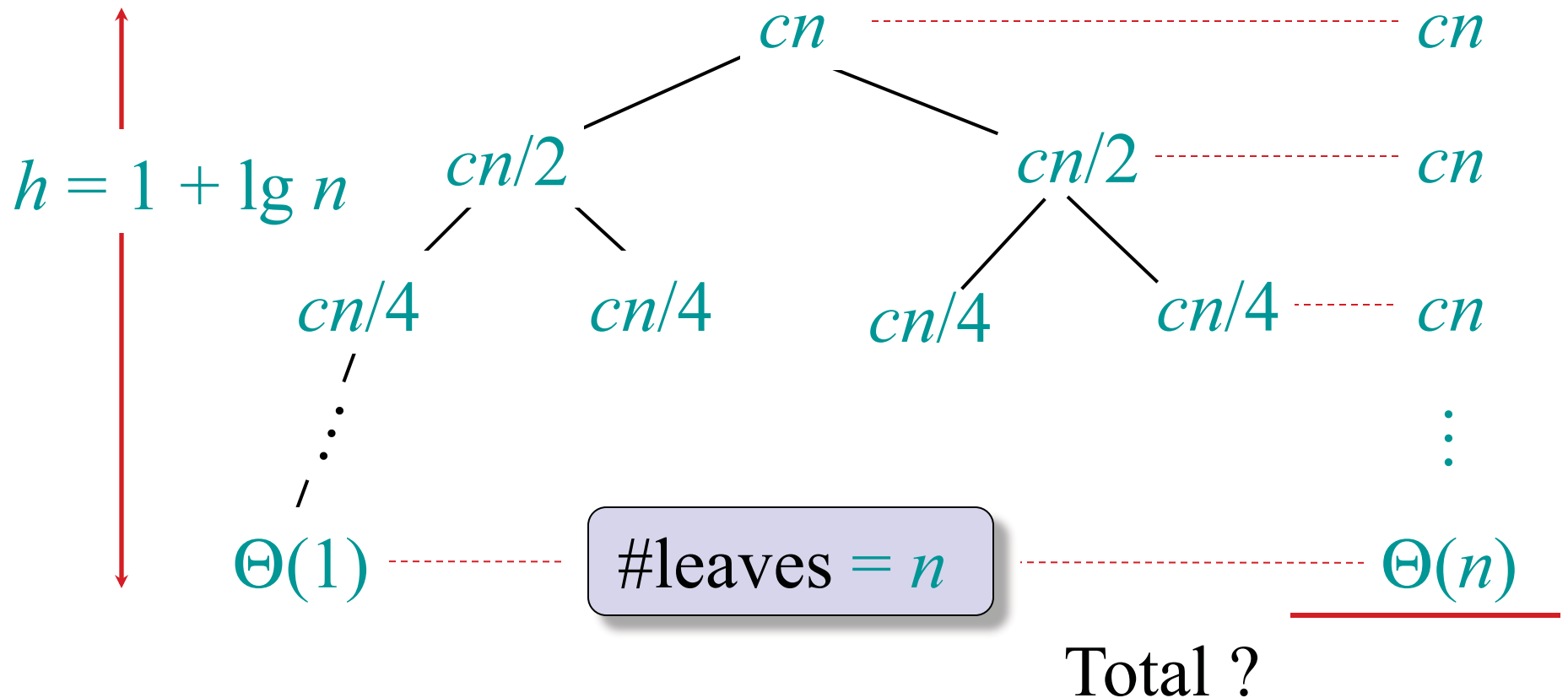
# Recursion tree

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



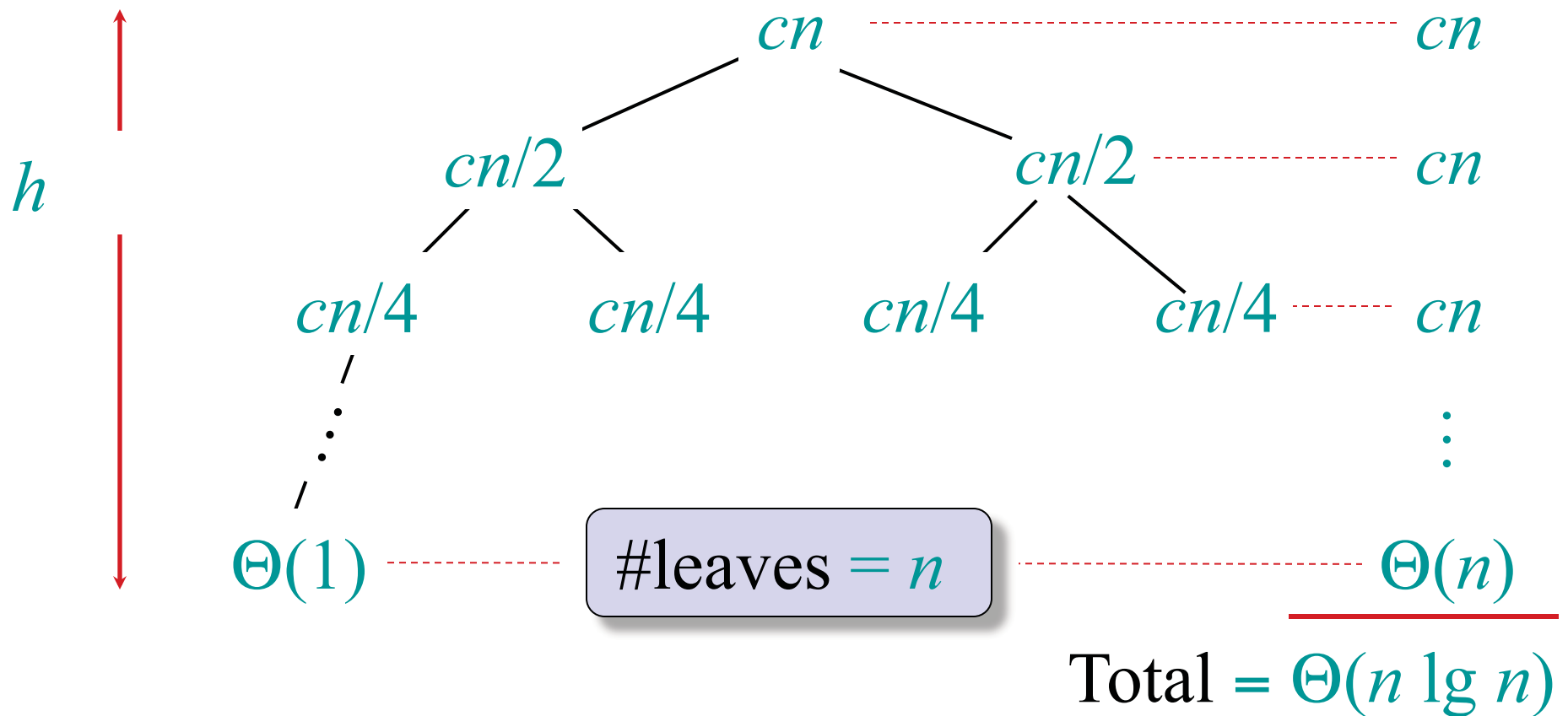
# Recursion tree

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



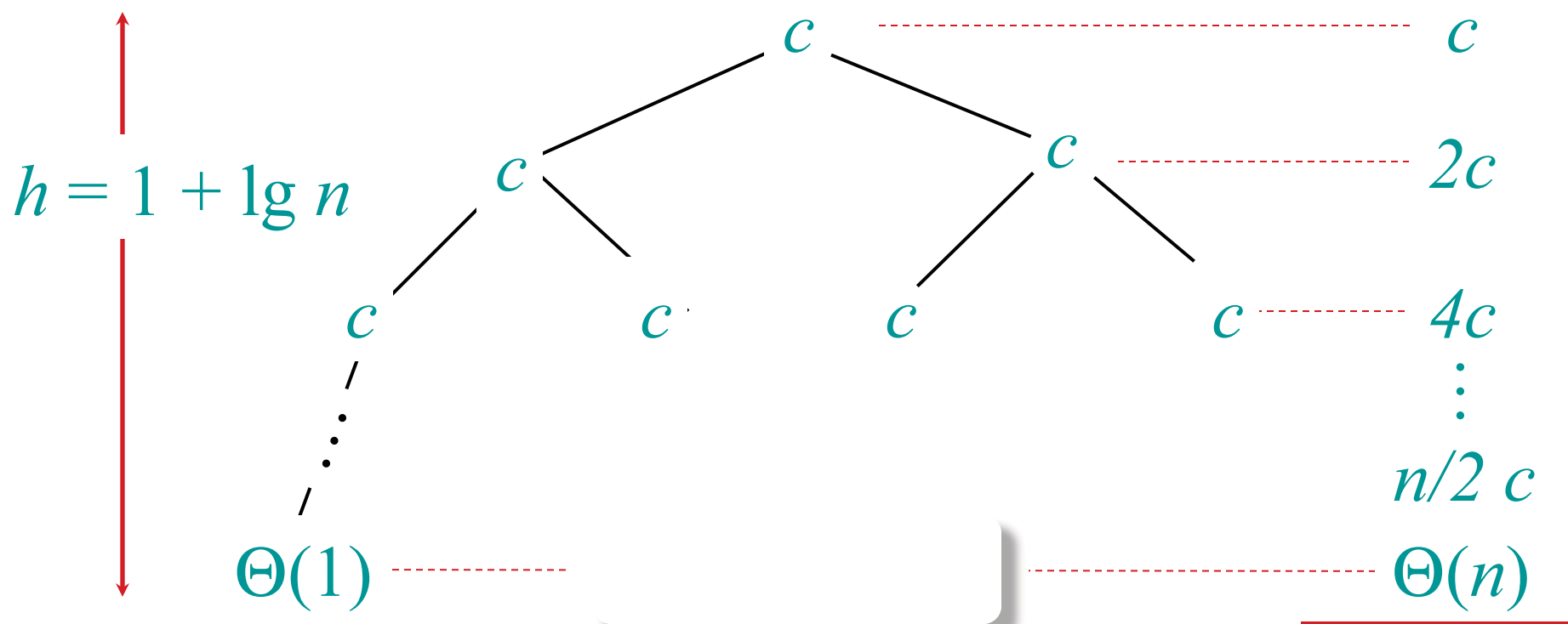
# Recursion tree

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



# Tree for different recurrence

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



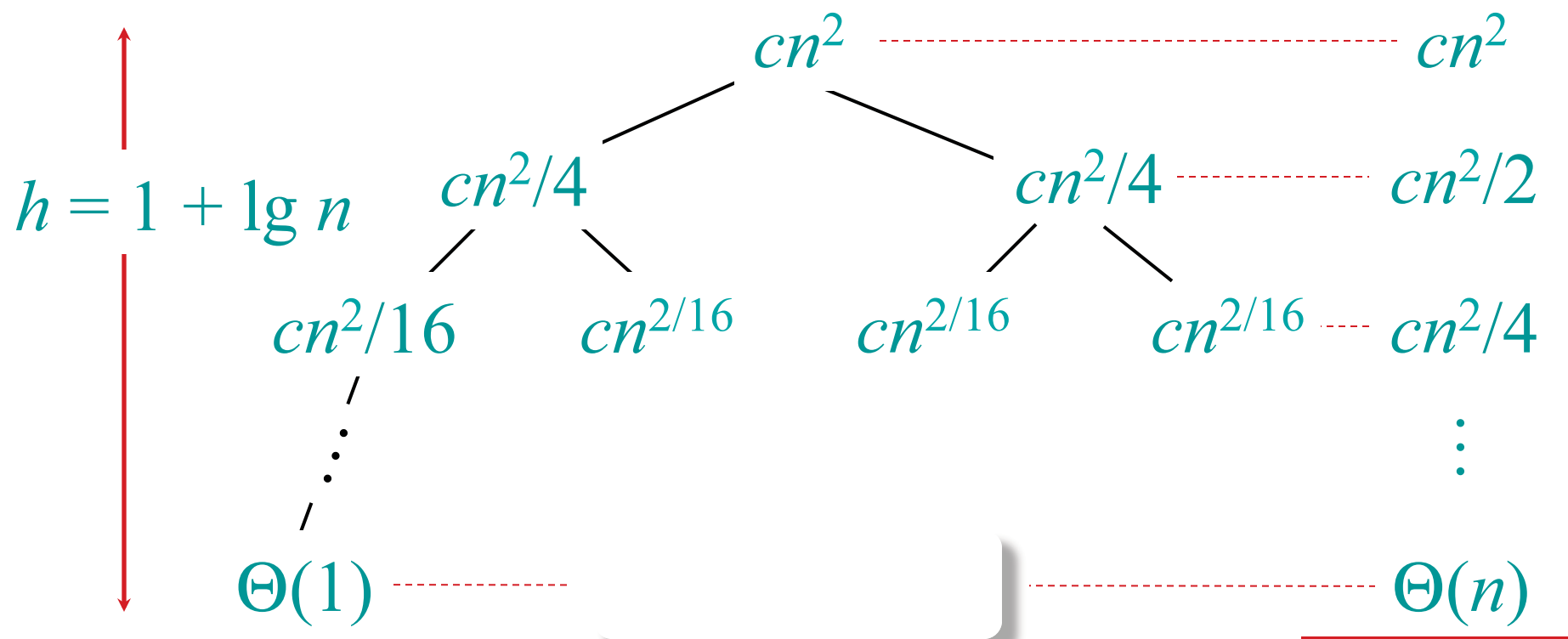
Note that  $1 + \frac{1}{2} + \frac{1}{4} + \dots < 2$

All the work done at the leaves

Total =  $\Theta(n)$

# Tree for yet another recurrence

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



Note that  $1 + \frac{1}{2} + \frac{1}{4} + \dots < 2$

All the work done at the root

Total =  $\Theta(n^2)$

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.006 Introduction to Algorithms  
Fall 2011

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.