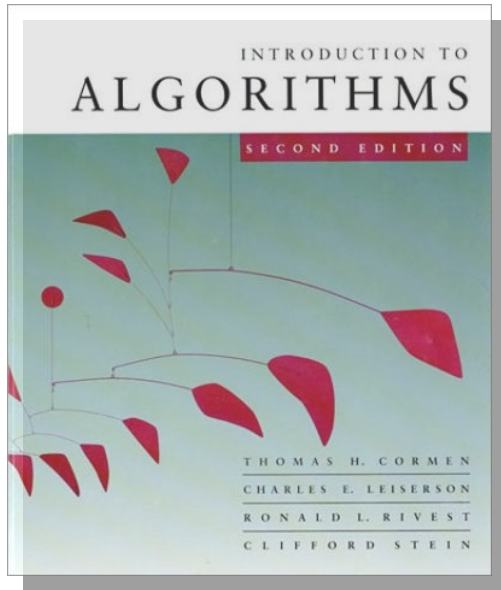


# *Introduction to Algorithms*

**6.046J/18.401J**

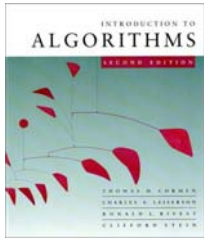


## **LECTURE 1**

### **Analysis of Algorithms**

- Insertion sort
- Asymptotic analysis
- Merge sort
- Recurrences

**Prof. Charles E. Leiserson**



# Course information

- 1. Staff**
- 2. Distance learning**
- 3. Prerequisites**
- 4. Lectures**
- 5. Recitations**
- 6. Handouts**
- 7. Textbook**
- 8. Course website**
- 9. Extra help**
- 10. Registration**
- 11. Problem sets**
- 12. Describing algorithms**
- 13. Grading policy**
- 14. Collaboration policy**



# Analysis of algorithms

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

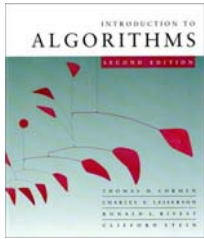
What's more important than performance?

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



# 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!



# The problem of sorting

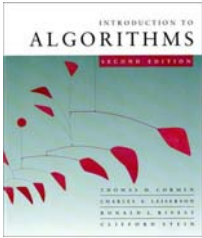
**Input:** sequence  $\langle a_1, a_2, \dots, a_n \rangle$  of numbers.

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

**Example:**

**Input:** 8 2 4 9 3 6

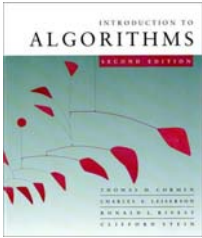
**Output:** 2 3 4 6 8 9



# Insertion sort

“pseudocode”

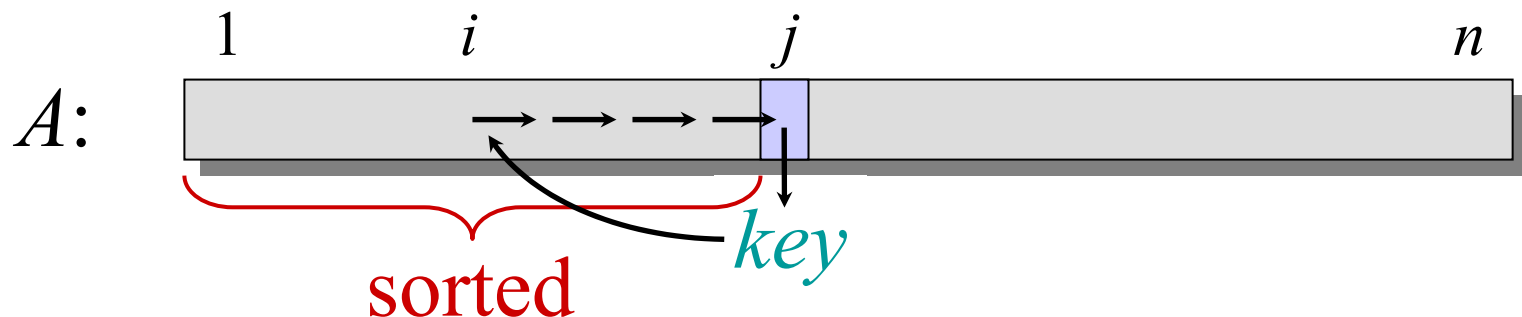
```
INSERTION-SORT ( $A, n$ )    ▷  $A[1 \dots n]$   
  for  $j \leftarrow 2$  to  $n$   
    do  $key \leftarrow A[j]$   
       $i \leftarrow j - 1$   
      while  $i > 0$  and  $A[i] > key$   
        do  $A[i+1] \leftarrow A[i]$   
           $i \leftarrow i - 1$   
       $A[i+1] = key$ 
```



# Insertion sort

“pseudocode”

```
INSERTION-SORT ( $A, n$ )    ▷  $A[1 \dots n]$   
  for  $j \leftarrow 2$  to  $n$   
    do  $key \leftarrow A[j]$   
       $i \leftarrow j - 1$   
      while  $i > 0$  and  $A[i] > key$   
        do  $A[i+1] \leftarrow A[i]$   
           $i \leftarrow i - 1$   
       $A[i+1] = key$ 
```





# Example of insertion sort

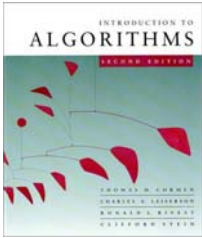
8 2 4 9 3 6





# Example of insertion sort



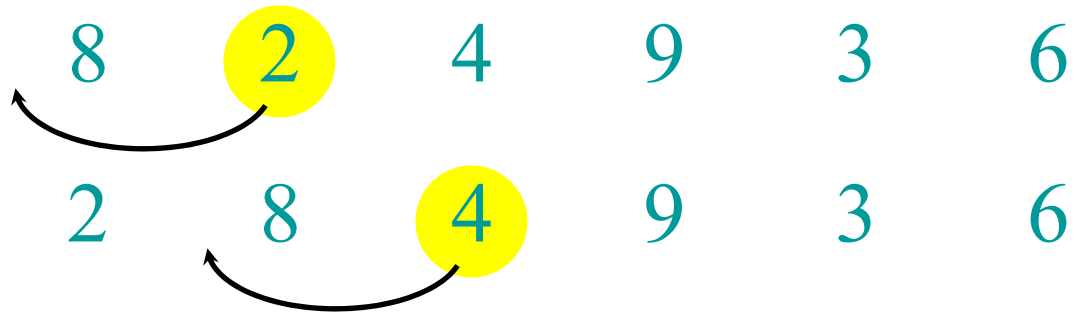


# Example of insertion sort



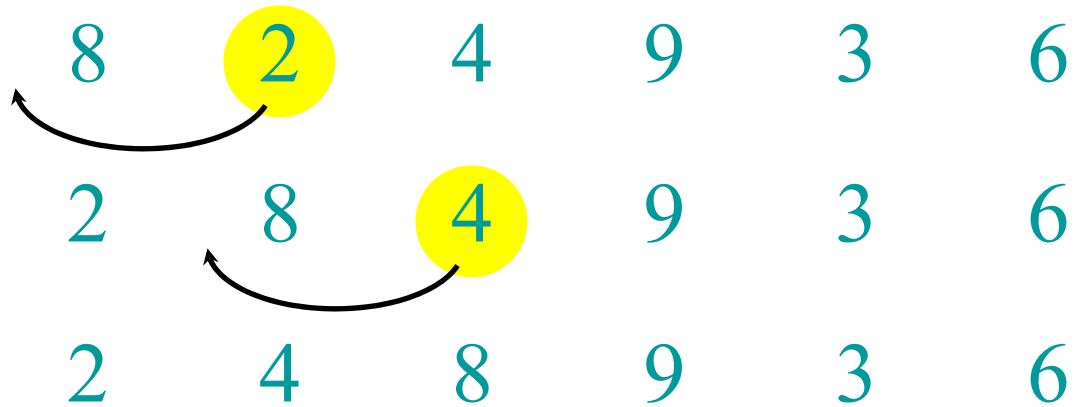


# Example of insertion sort



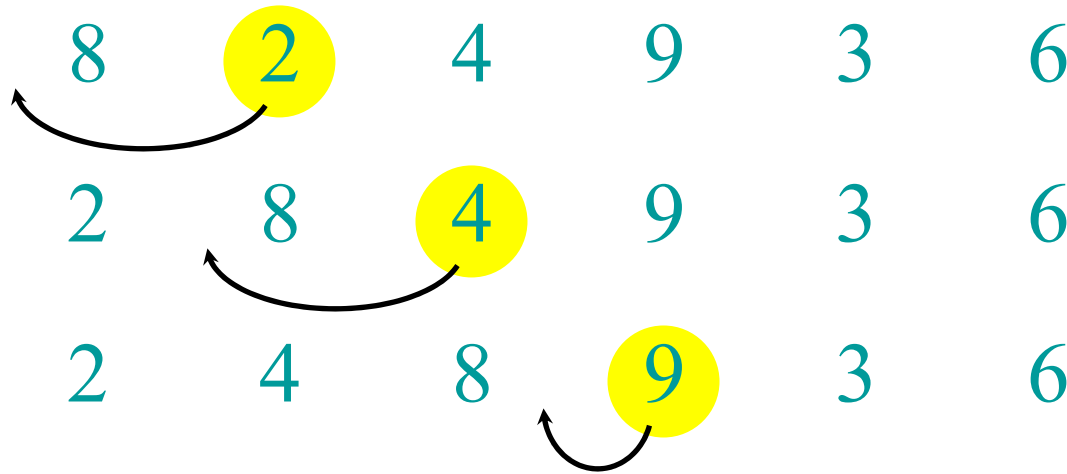


# Example of insertion sort



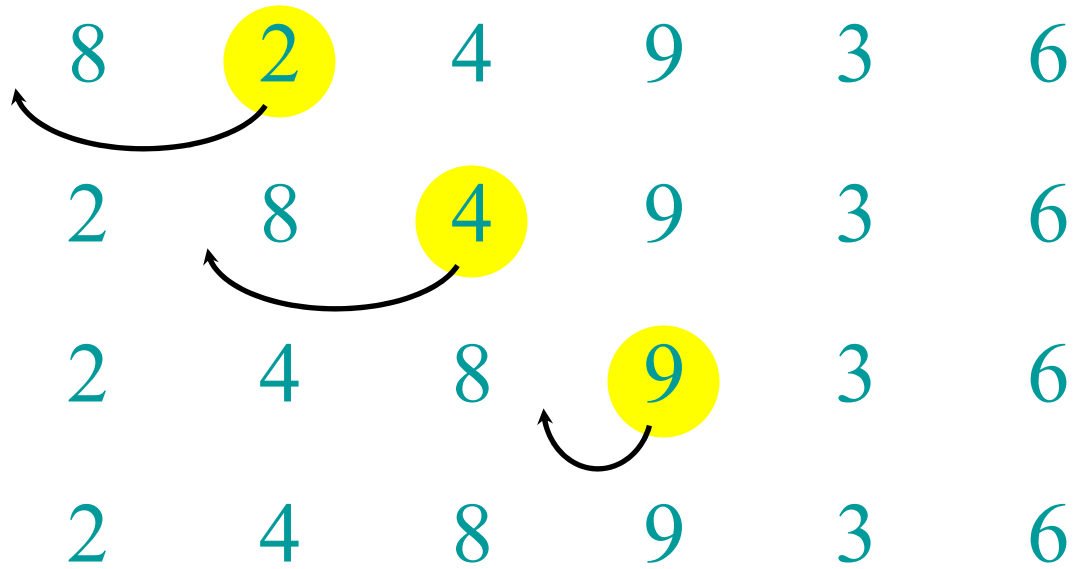


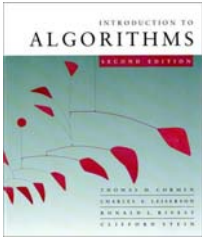
# Example of insertion sort



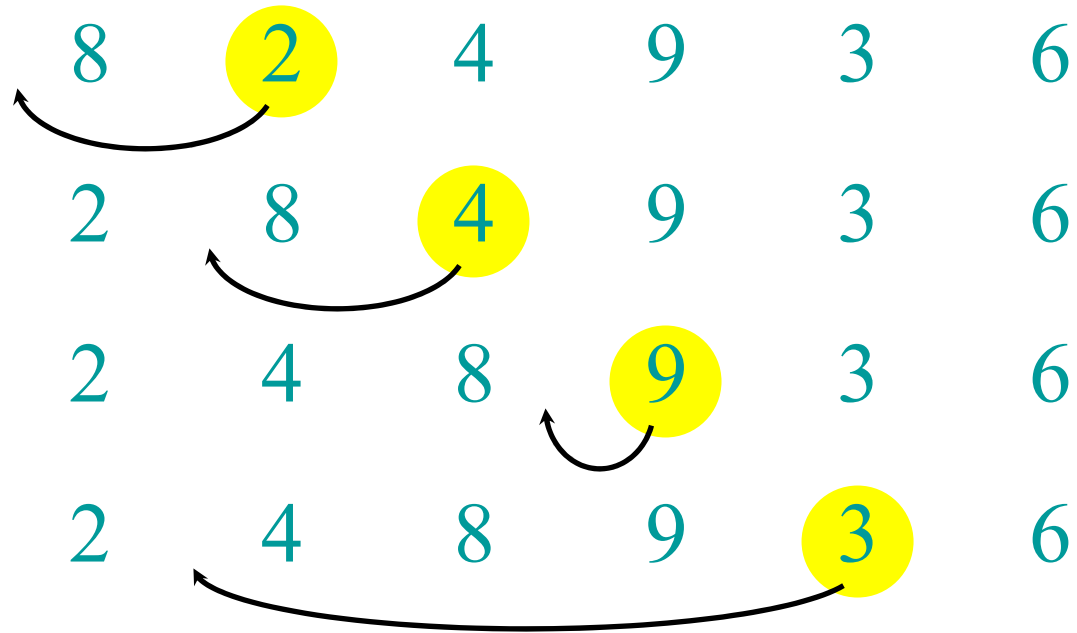


# Example of insertion sort



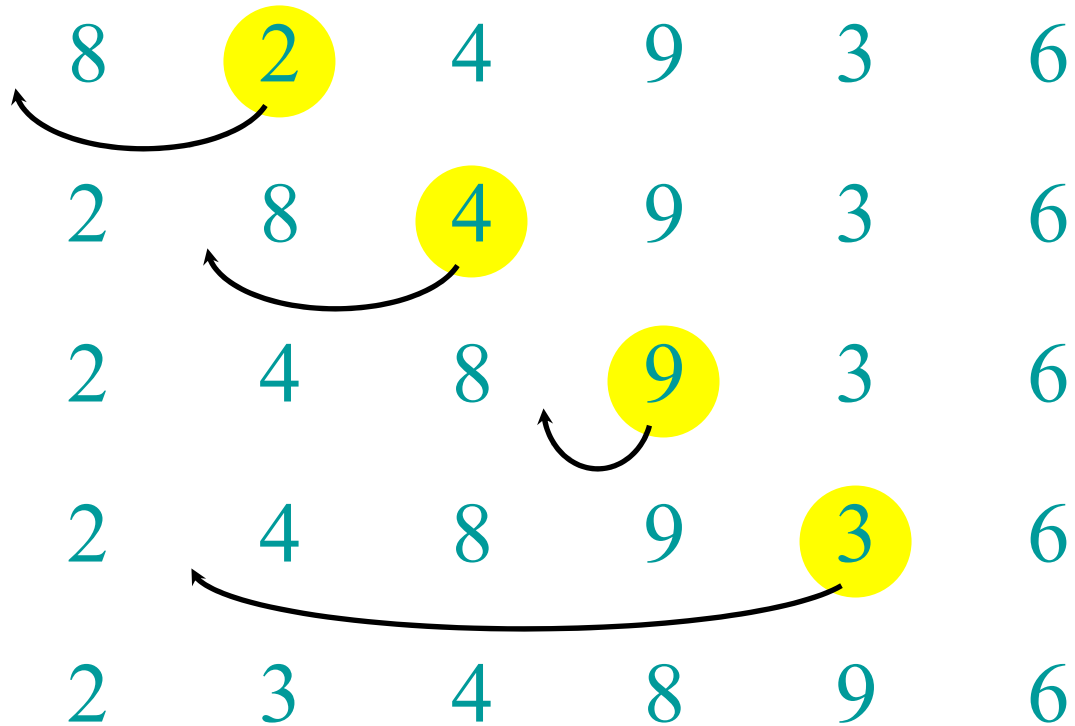


# Example of insertion sort





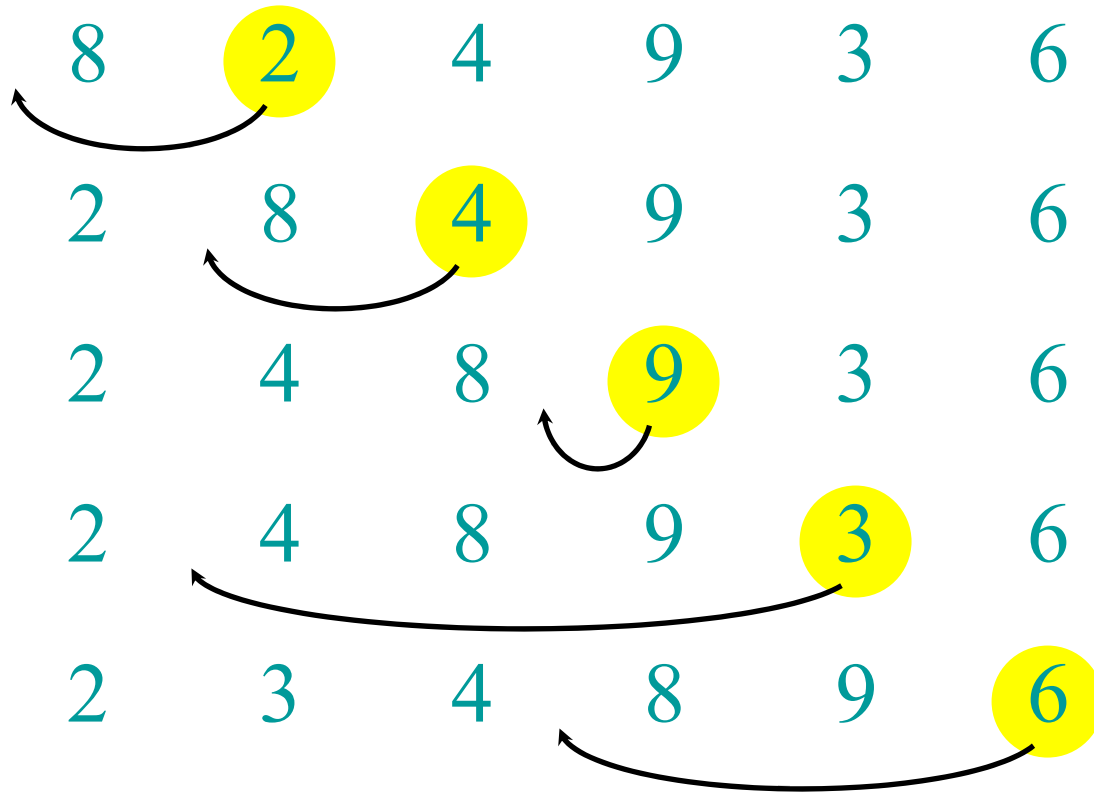
# Example of insertion sort





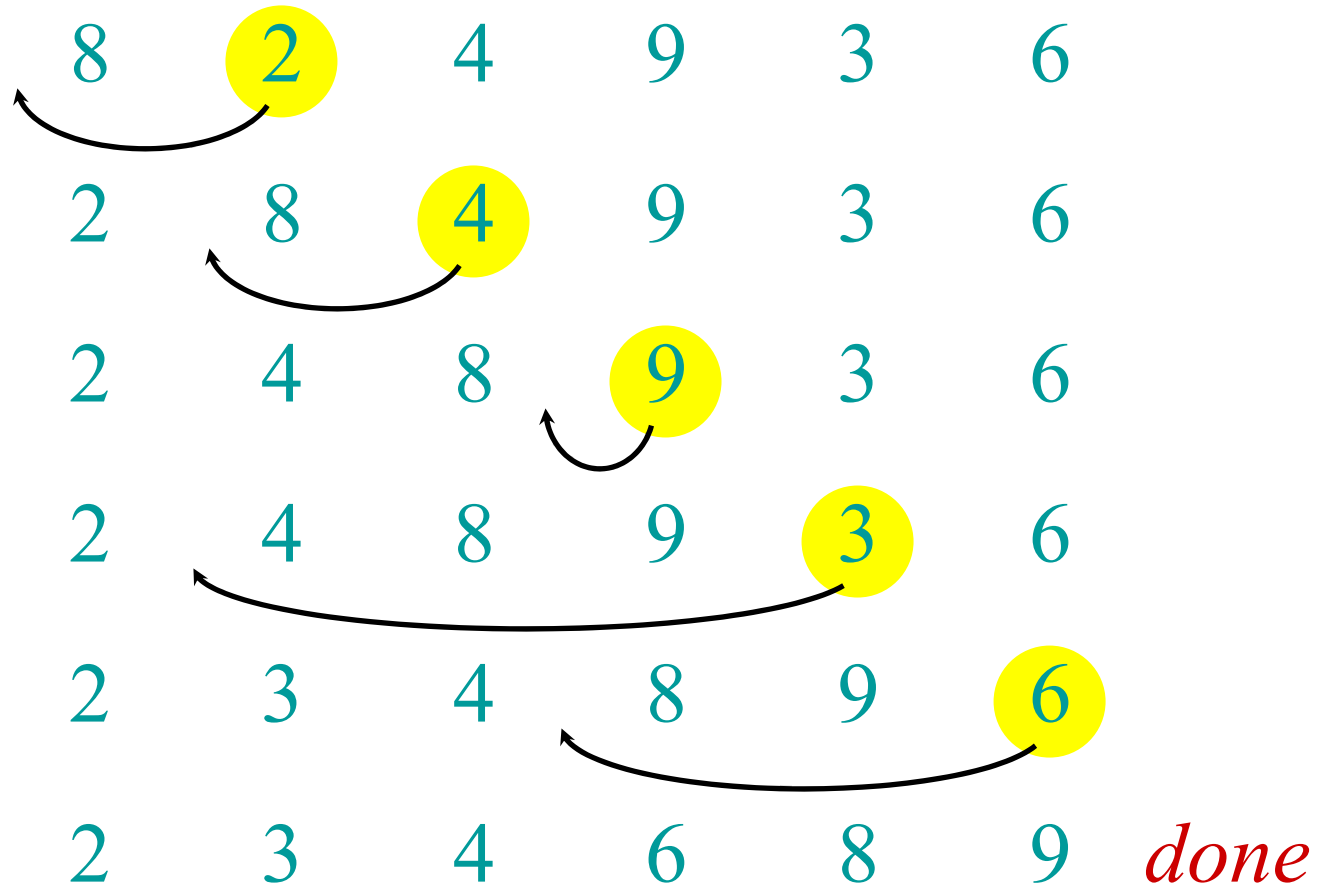


# Example of insertion sort





# Example of insertion sort





# Running time

- 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)

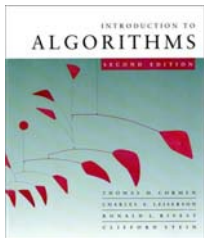
- $T(n)$  = maximum time of algorithm on any input of size  $n$ .

**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.



# Machine-independent time

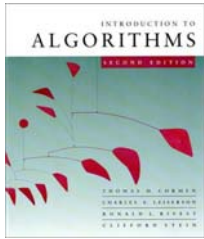
*What is insertion sort's worst-case time?*

- 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  $n \rightarrow \infty$ .

**“Asymptotic Analysis”**



# $\Theta$ -notation

## *Math:*

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

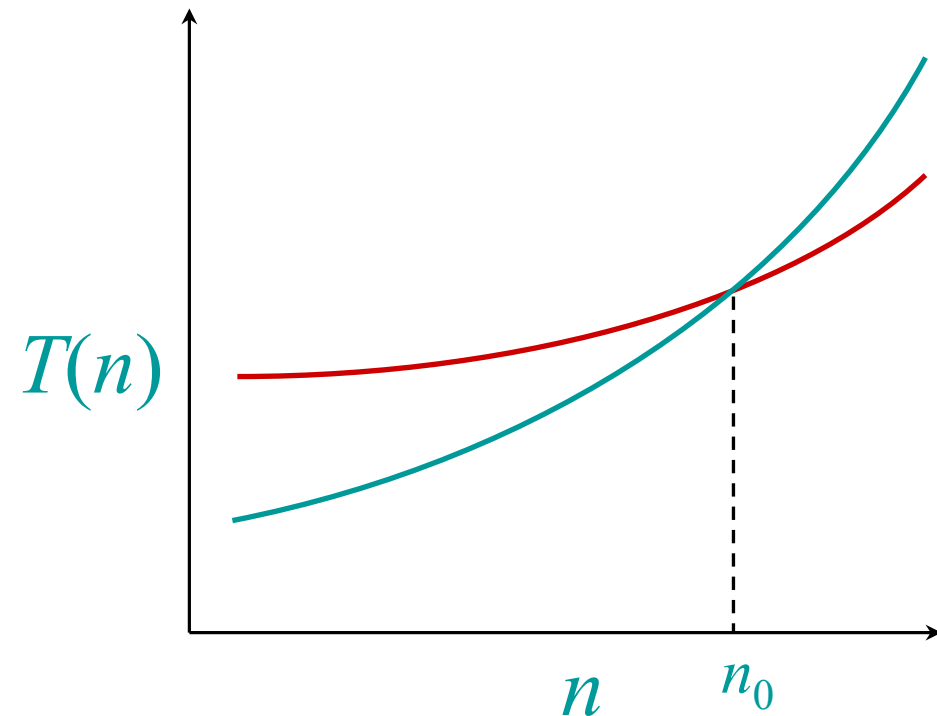
## *Engineering:*

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

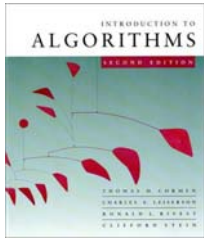


# Asymptotic performance

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.



# Insertion sort analysis

***Worst case:*** Input reverse sorted.

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

☐ ***Average case:*** All permutations equally likely.

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

☐ *Is insertion sort a fast sorting algorithm?*

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



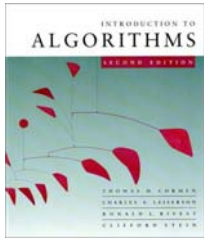


# Merge sort

## 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 2 sorted lists.

*Key subroutine:* MERGE



# Merging two sorted arrays

20 12

13 11

7 9

2 1

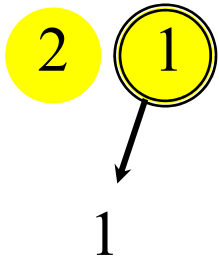


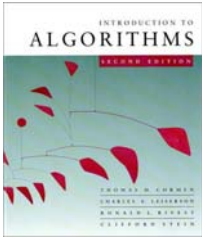
# Merging two sorted arrays

20 12

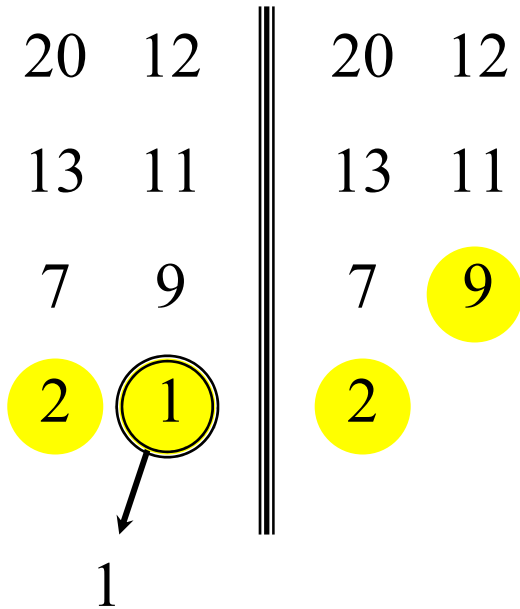
13 11

7 9



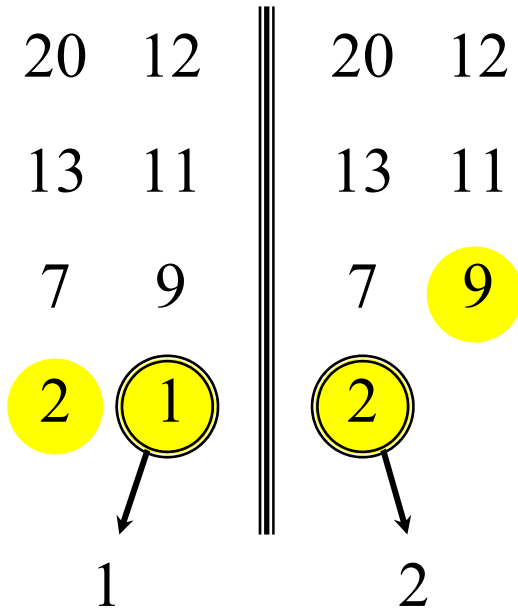


# Merging two sorted arrays



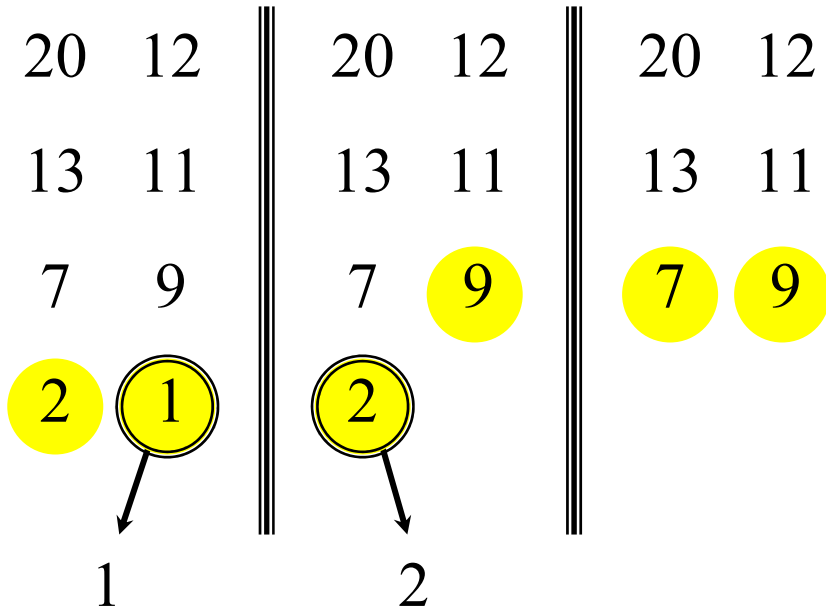


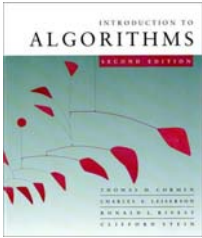
# Merging two sorted arrays



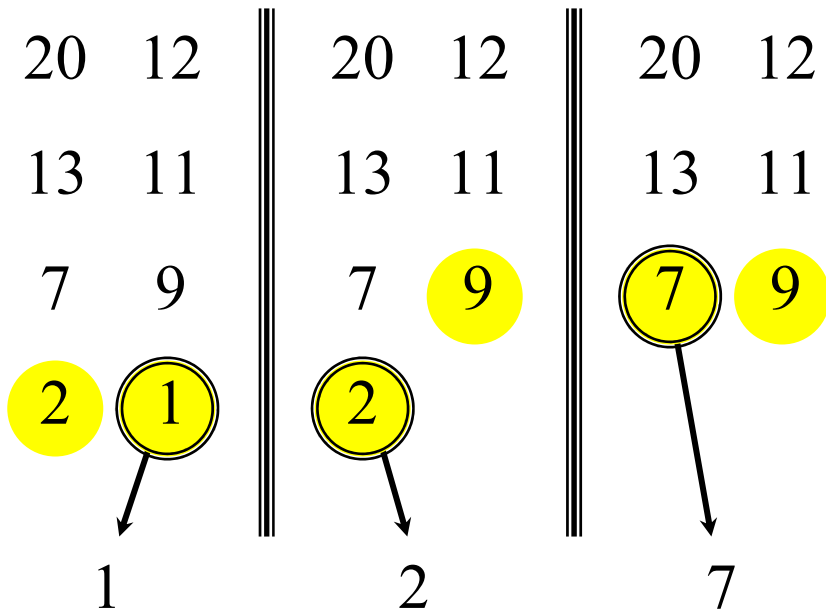


# Merging two sorted arrays



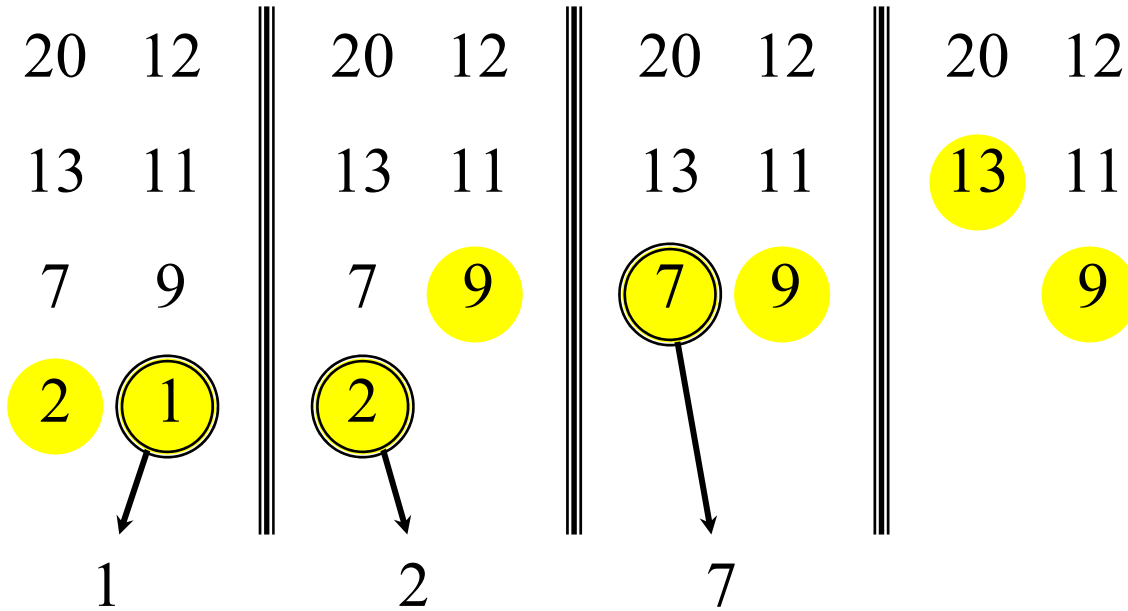


# Merging two sorted arrays





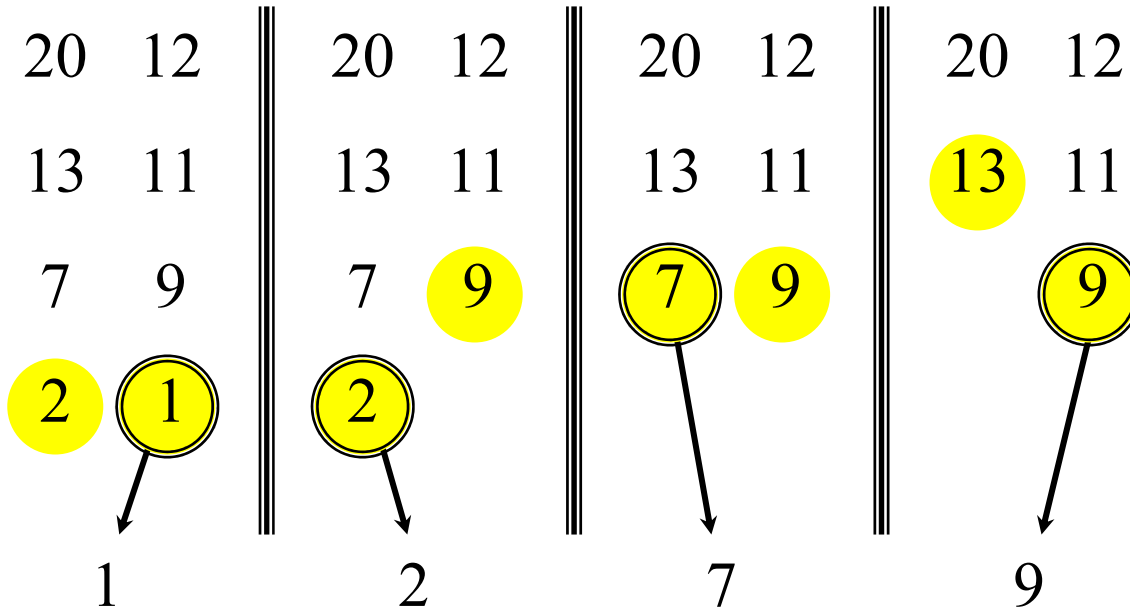
# Merging two sorted arrays

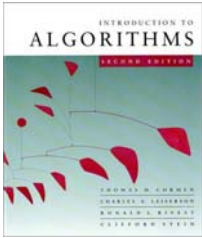




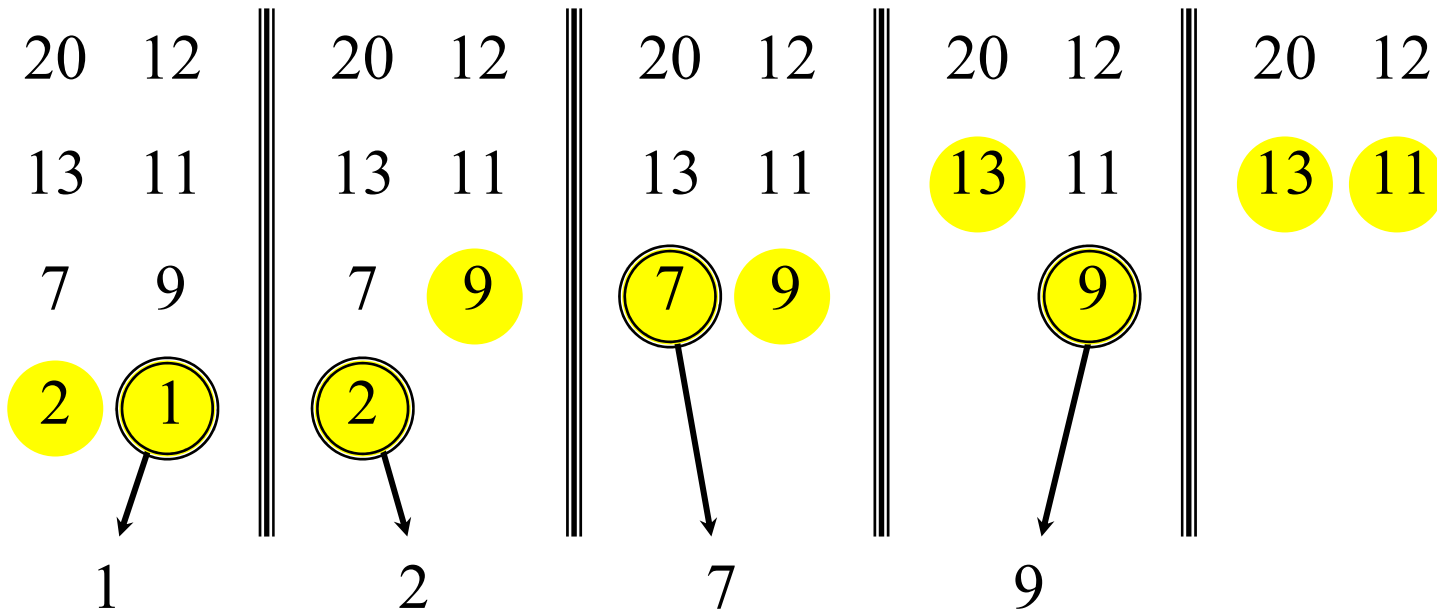


# Merging two sorted arrays



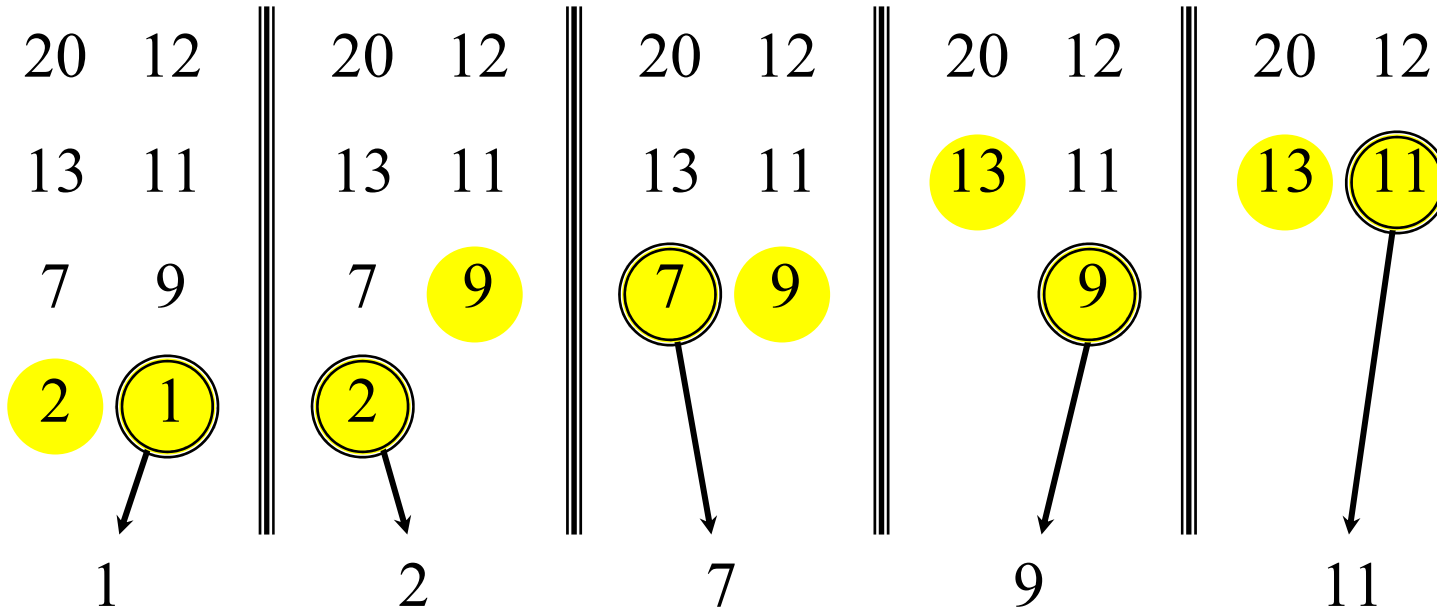


# Merging two sorted arrays



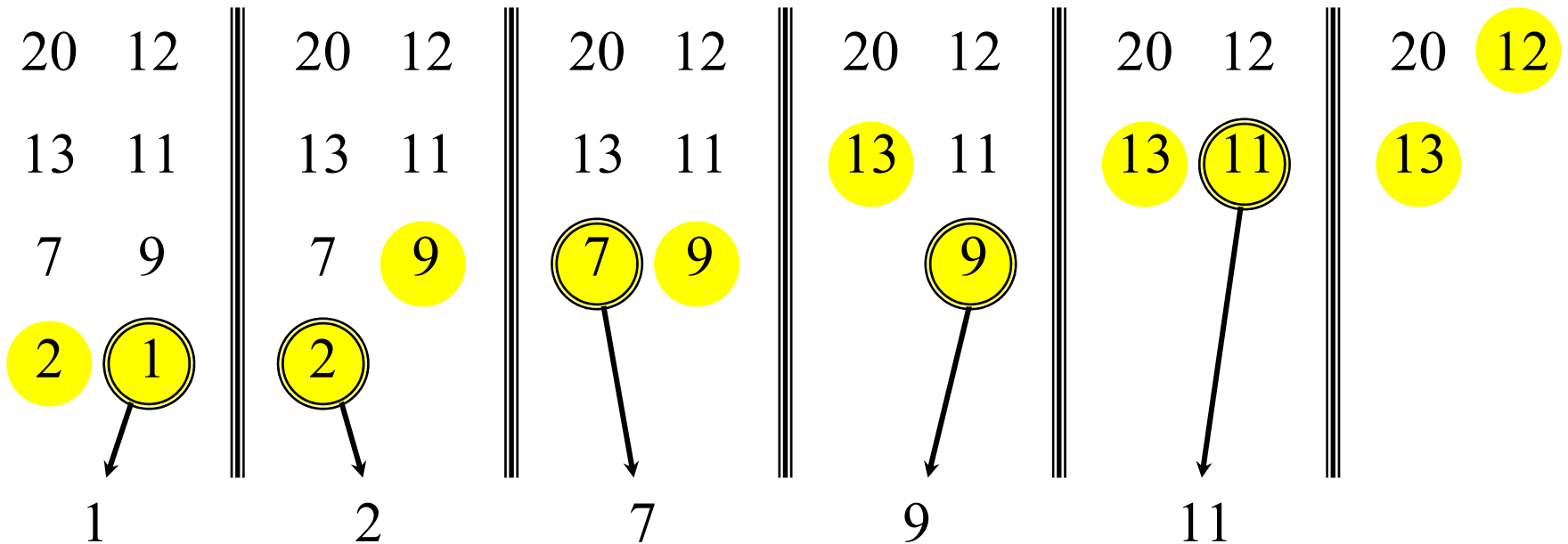


# Merging two sorted arrays



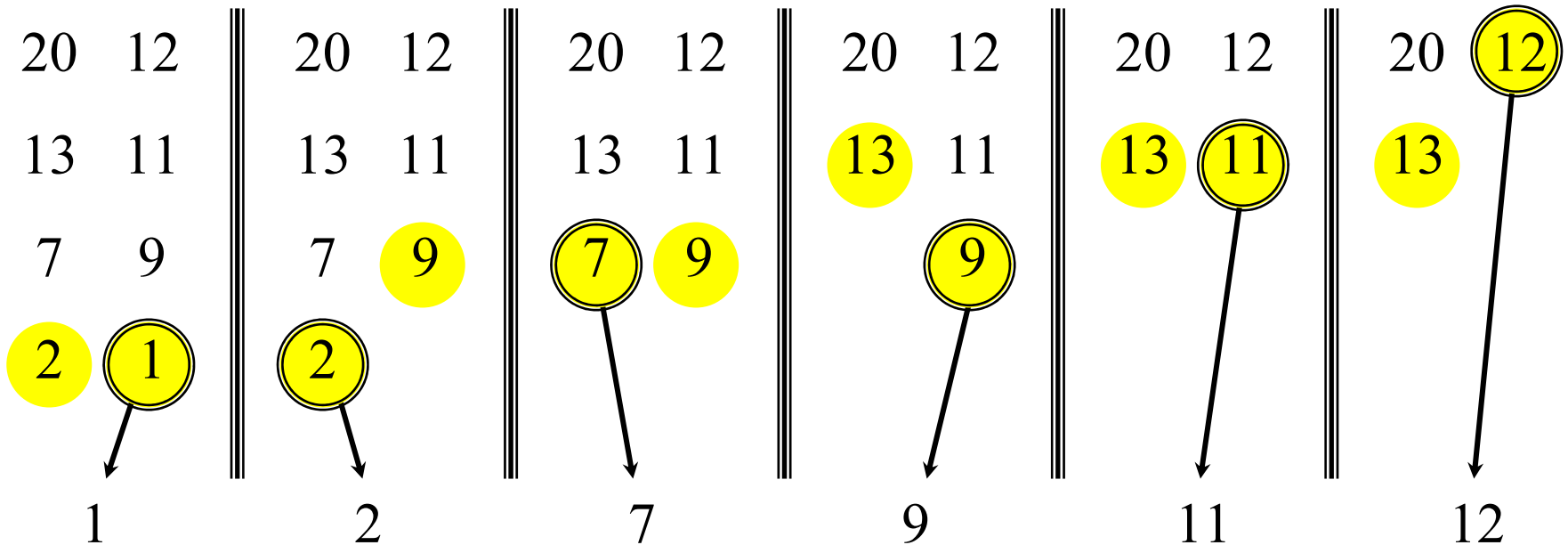


# Merging two sorted arrays



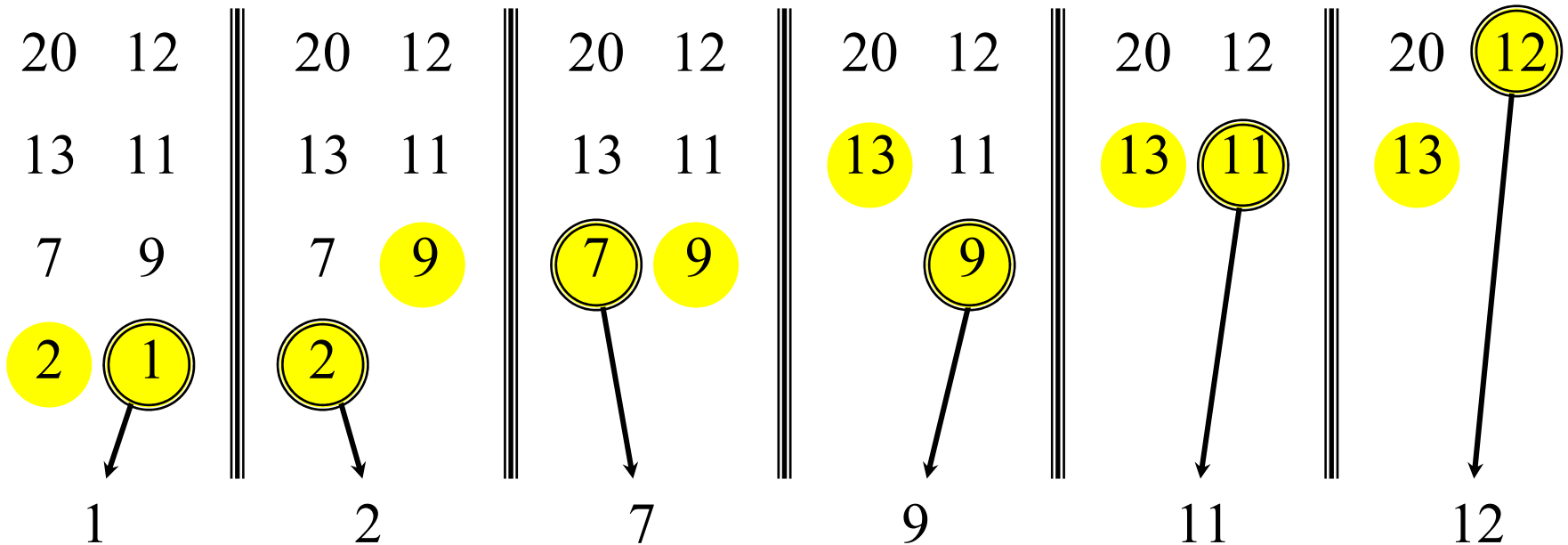


# Merging two sorted arrays

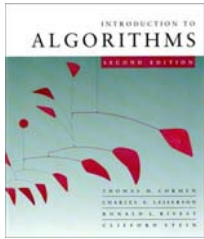




# Merging two sorted arrays



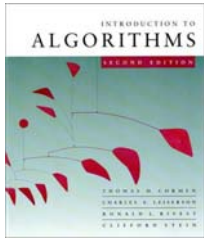
Time =  $\Theta(n)$  to merge a total of  $n$  elements (linear time).



# Analyzing merge sort

	$T(n)$		<b>MERGE-SORT</b> $A[1 \dots n]$
	$\Theta(1)$		1. If $n = 1$ , done.
<i>Abuse</i> ↗	$2T(n/2)$		2. Recursively sort $A[1 \dots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \dots n]$ .
	$\Theta(n)$		3. “ <i>Merge</i> ” the 2 sorted lists

**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

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

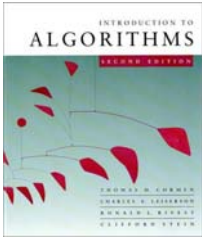
- We shall usually omit stating the base case when  $T(n) = \Theta(1)$  for sufficiently small  $n$ , but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on  $T(n)$ .





# Recursion tree

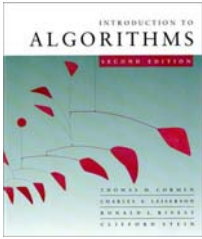
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.



# Recursion tree

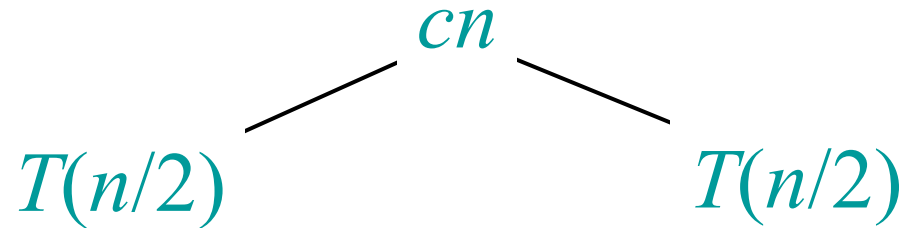
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

$$T(n)$$



# Recursion tree

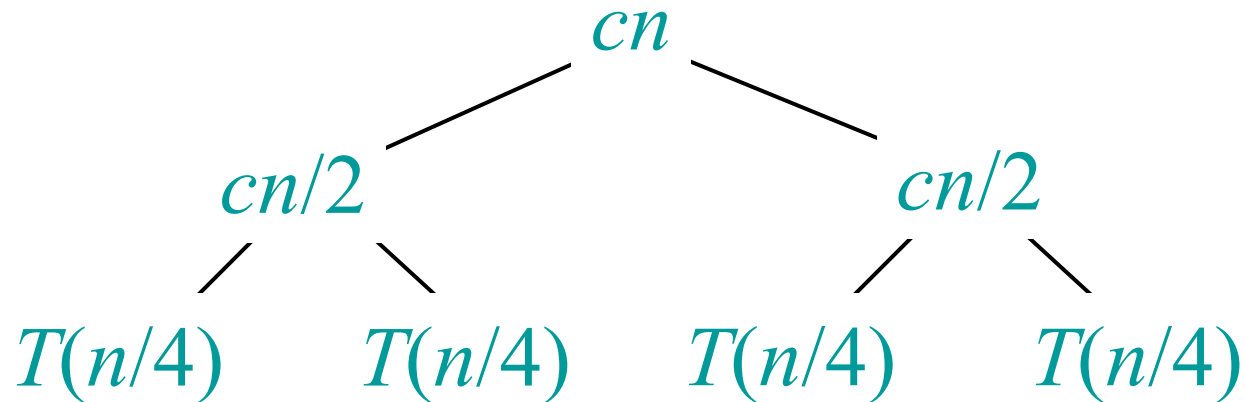
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.





# Recursion tree

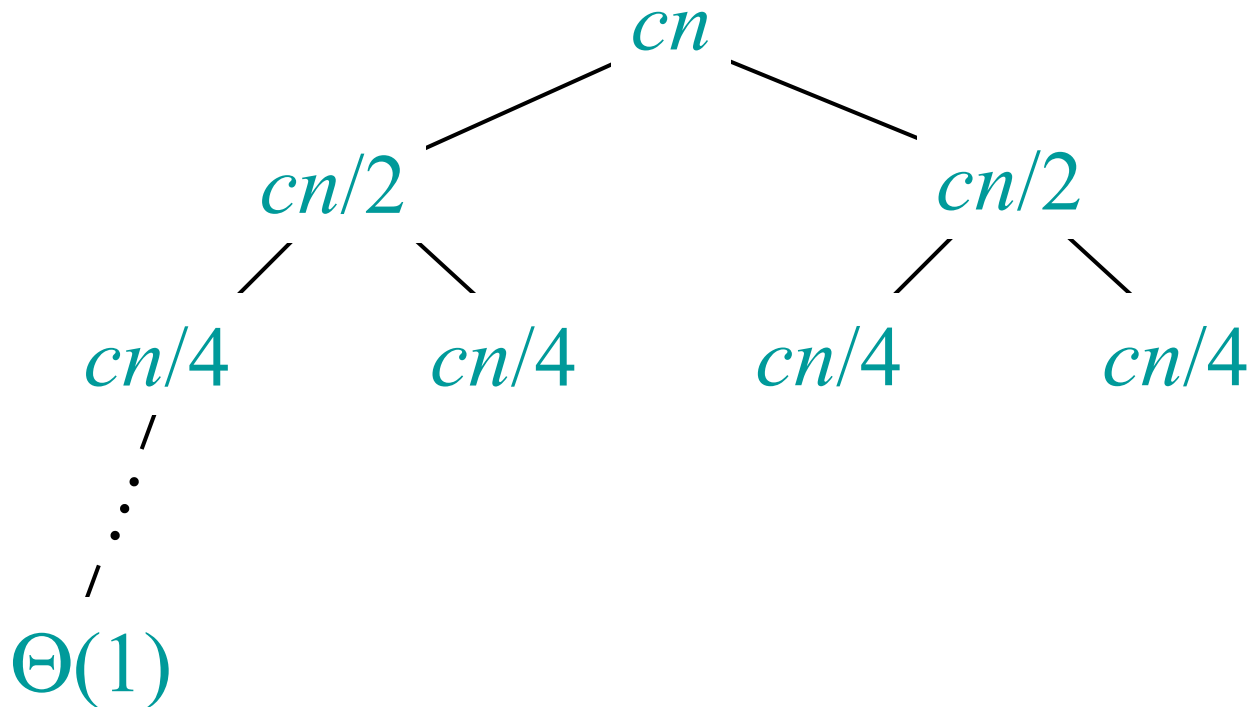
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.





# Recursion tree

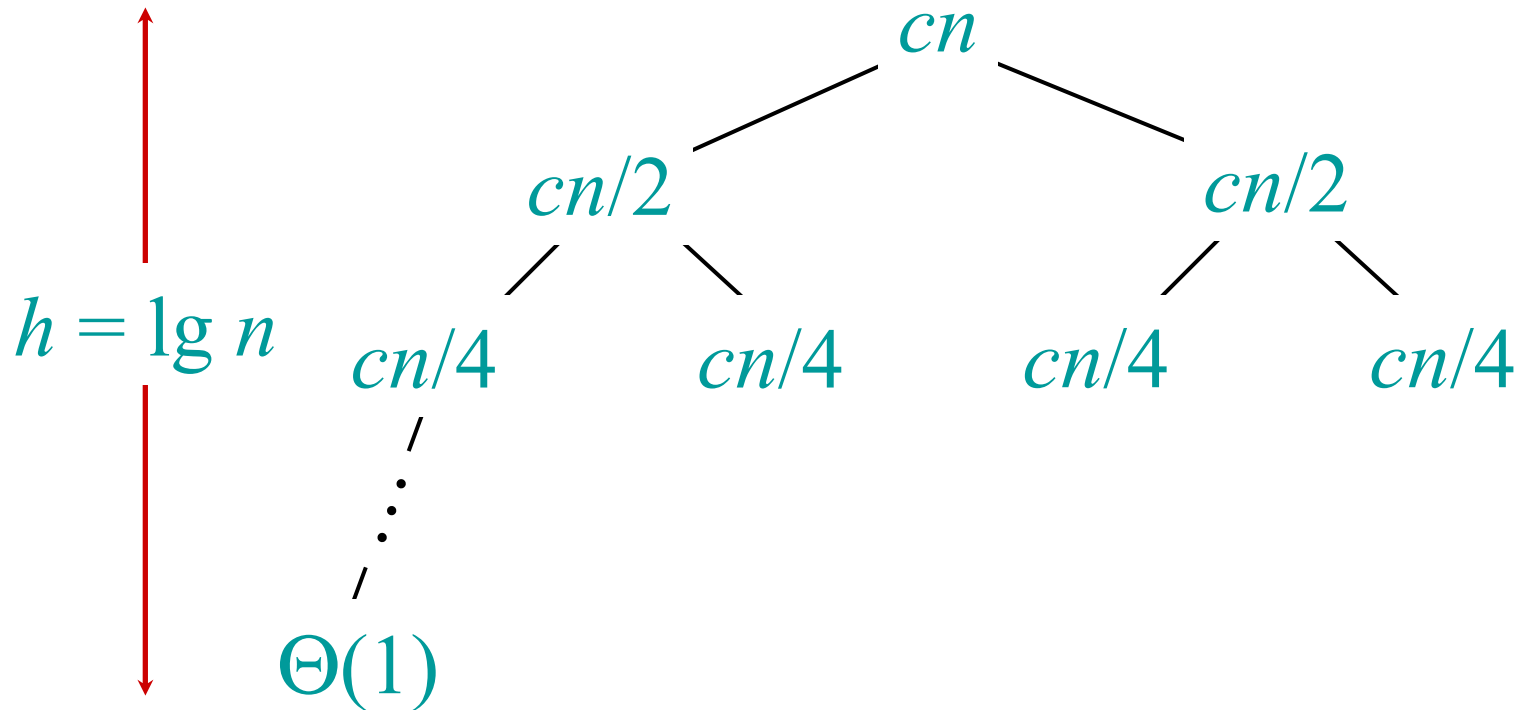
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.





# Recursion tree

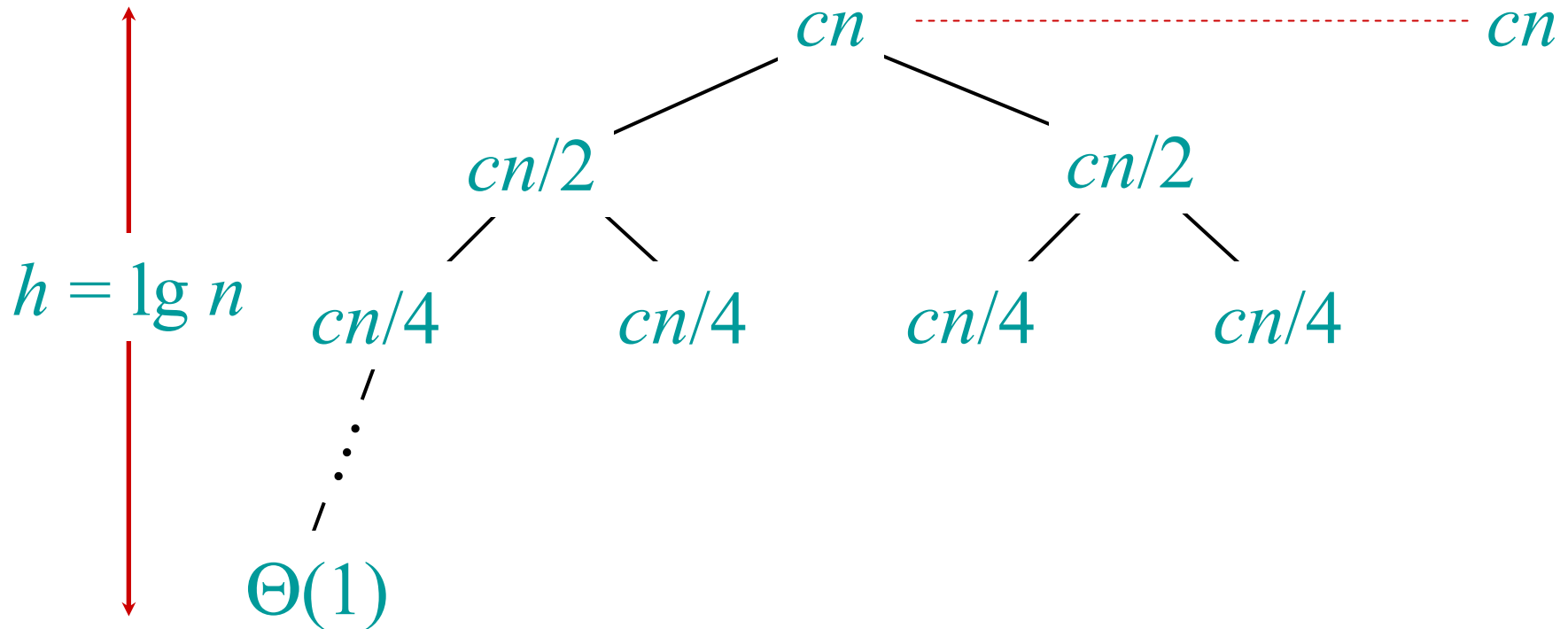
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.





# Recursion tree

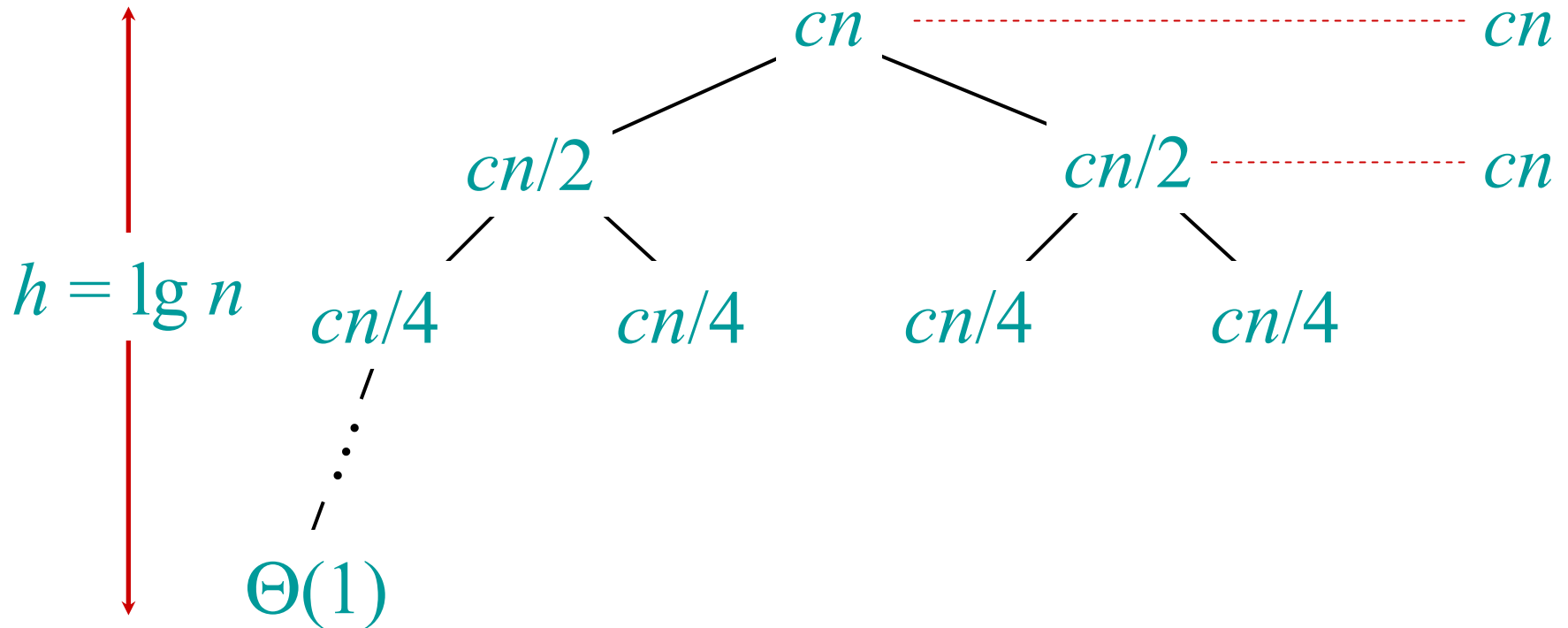
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.





# Recursion tree

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

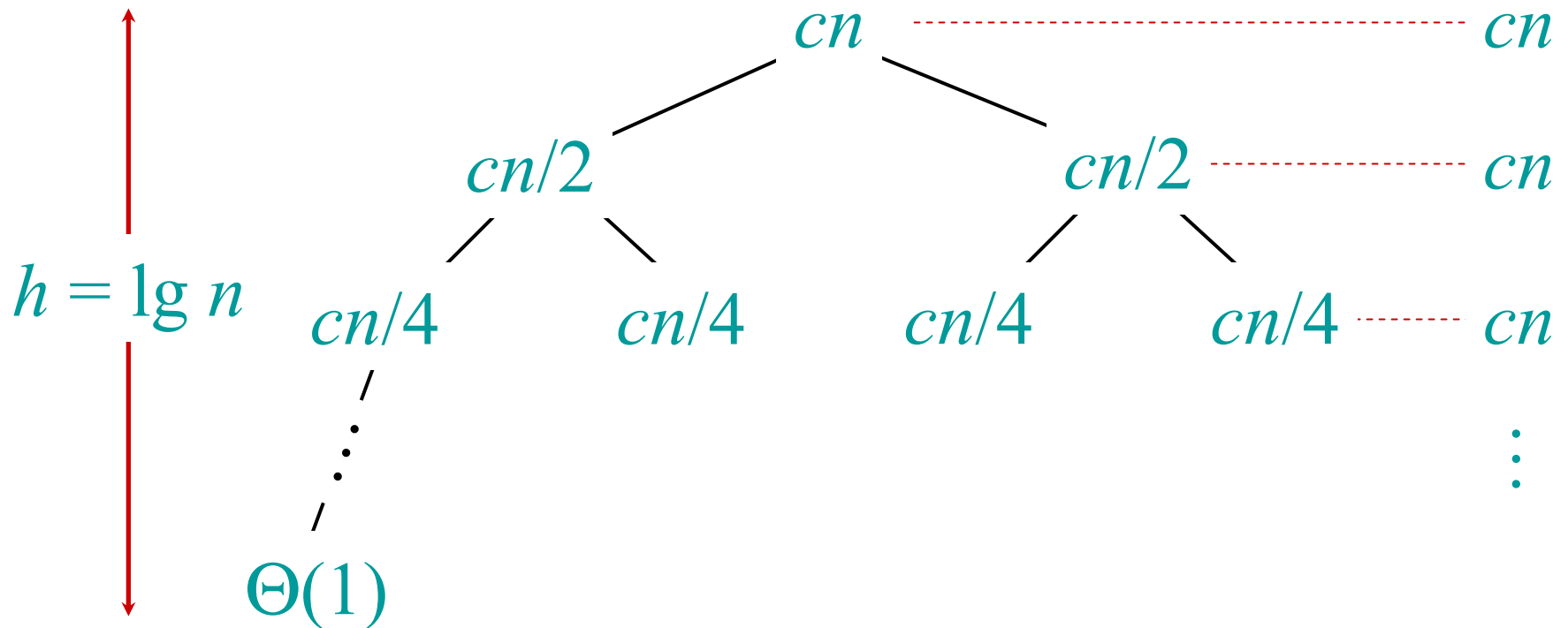






# Recursion tree

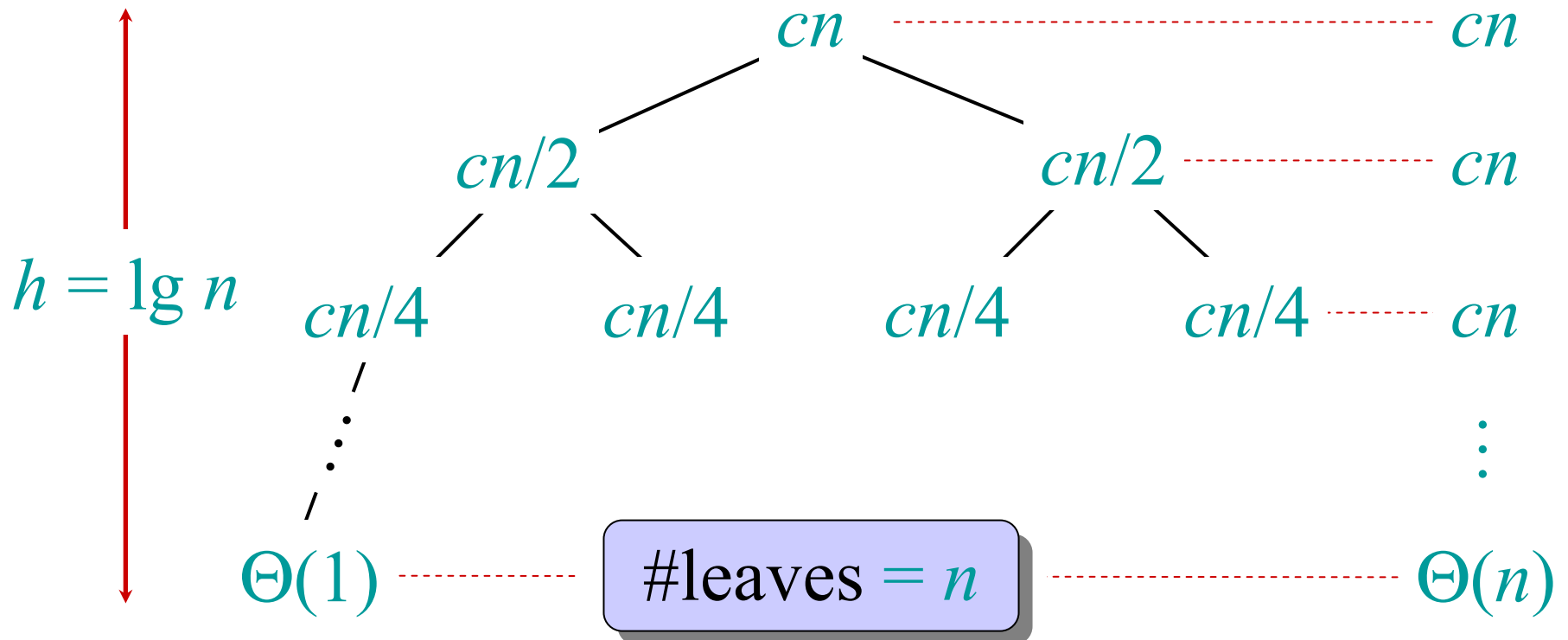
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.





# Recursion tree

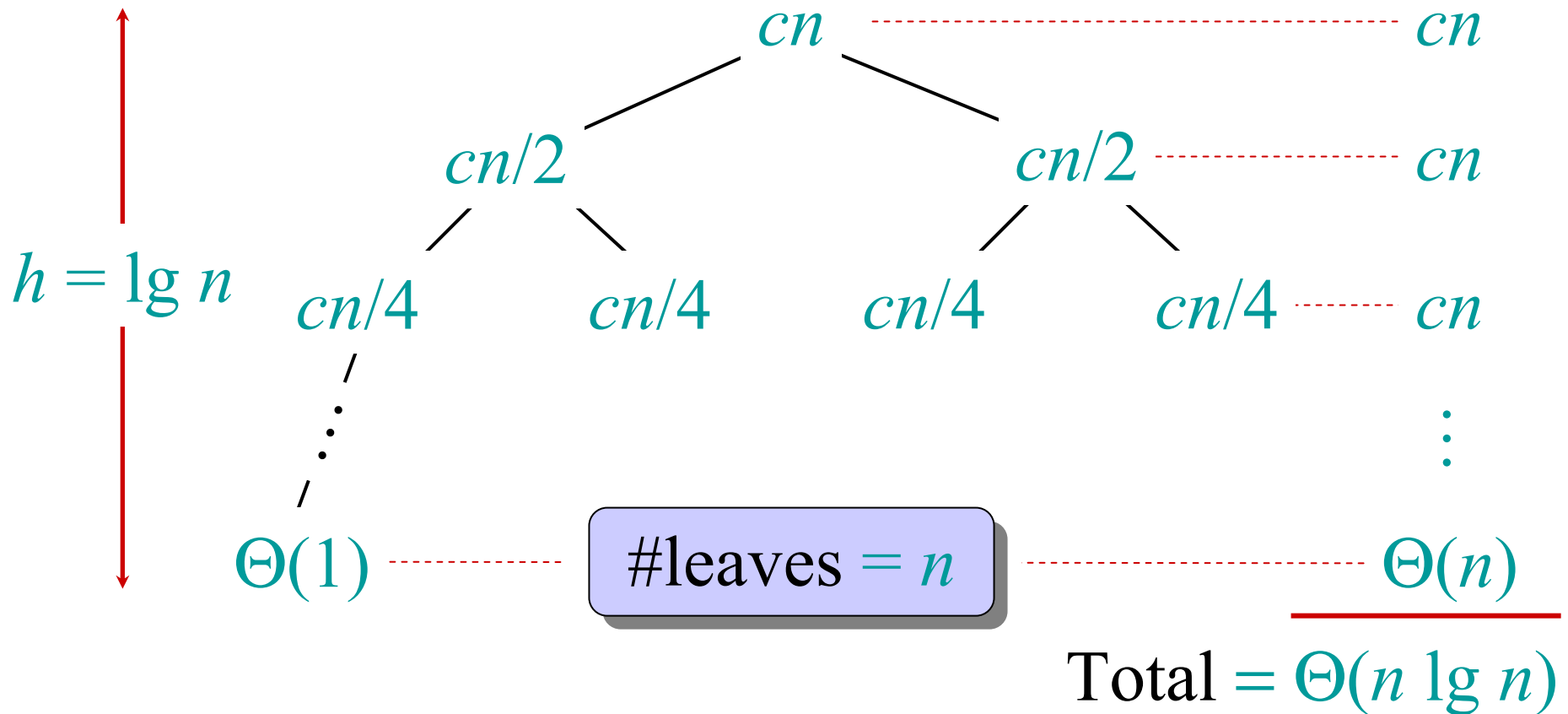
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.





# Recursion tree

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





# 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!