CS 325 – Analysis of Sorting

Week 1 – Part 1

Getting Started

What is this class about? The theoretical study of design and analysis of computer algorithms

Basic goals for an algorithm:

- always correct
- always terminates
- This class: performance
  - Performance often draws the line between what is possible and what is impossible.

Design and Analysis of Algorithms

- ***Analysis:*** predict the cost of an algorithm in terms of resources and performance. *Theory*

- ***Design:*** design algorithms which minimize the cost

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

Importance of Sorting

- Maintain a directory of names, phone book, sort by grades of students, …
- Find the median
- Binary Search assumes array is sorted.
- Greedy Algorithms

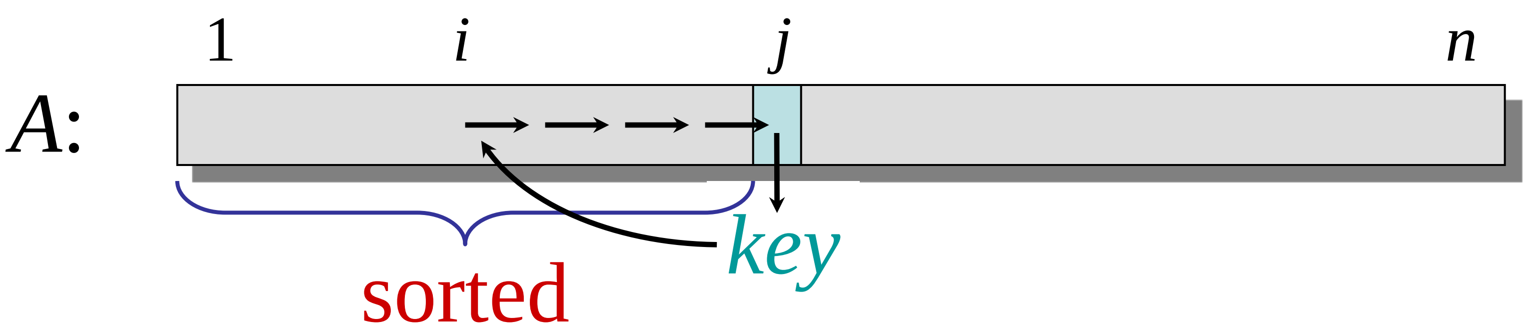
Problem vs Algorithm

- Many algorithms exist to solve the sorting problem.
- Running time is associated with an algorithms.
- Bounds on running times may also be associated with the problem.

Algorithm 1: 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.
- Major Simplifying Convention: Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.

  $T_A(n) =$ time of A on length n inputs
- Generally, we seek upper bounds on the running time, to have a guarantee of performance.

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: (NEVER)

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

Insertion sort analysis

Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} j = \mathrm{O}\left(n^2\right)$$ [arithmetic series]

Is insertion sort a fast sorting algorithm?

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

Insertion sort analysis

Average case: All permutations equally likely.

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

Is insertion sort a fast sorting algorithm?

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

Insertion sort analysis

Best Case: Already sorted. *Nearly Sorted??*

O(n)

Can we sort better?

Insertion upper bound $O(n^2)$

Are there other ways to sort??

Merge Sort

Sorting Problem: Sort a sequence of *n* elements into non-decreasing order.

- ***Divide:*** Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each

- ***Conquer:*** Sort the two subsequences recursively using merge sort.

- ***Combine:*** Merge the two sorted subsequences to produce the sorted answer.

Merge sort – Pseudo code

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

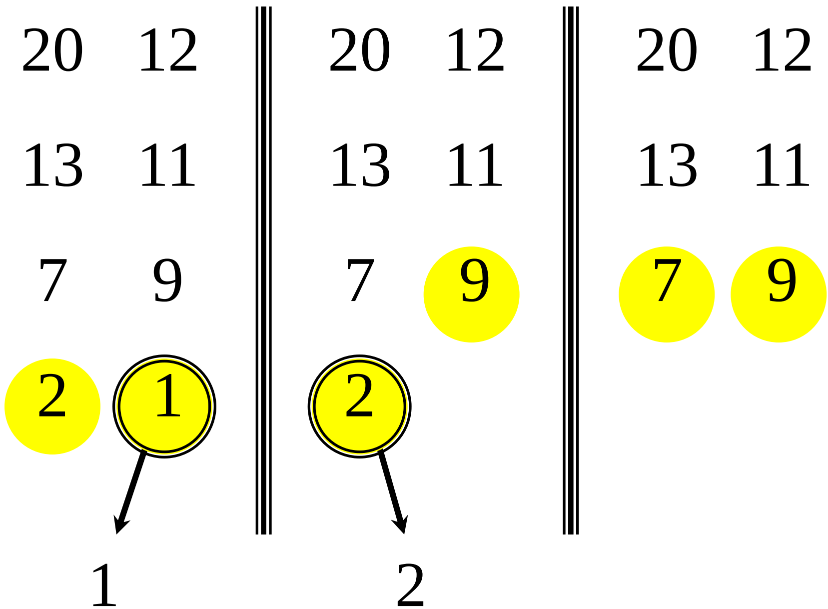
20	12
13	11
7	9
2	1

1

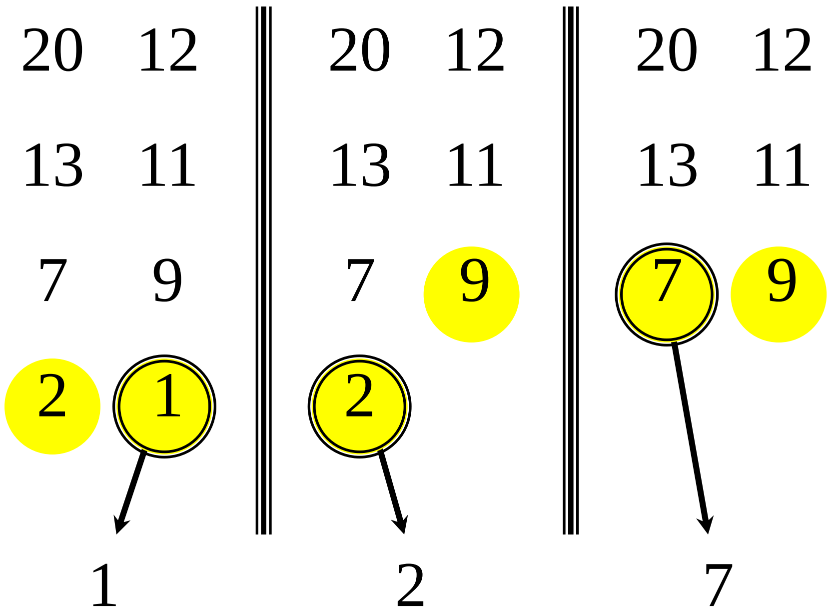
20	12
13	11
7	9
2	

2

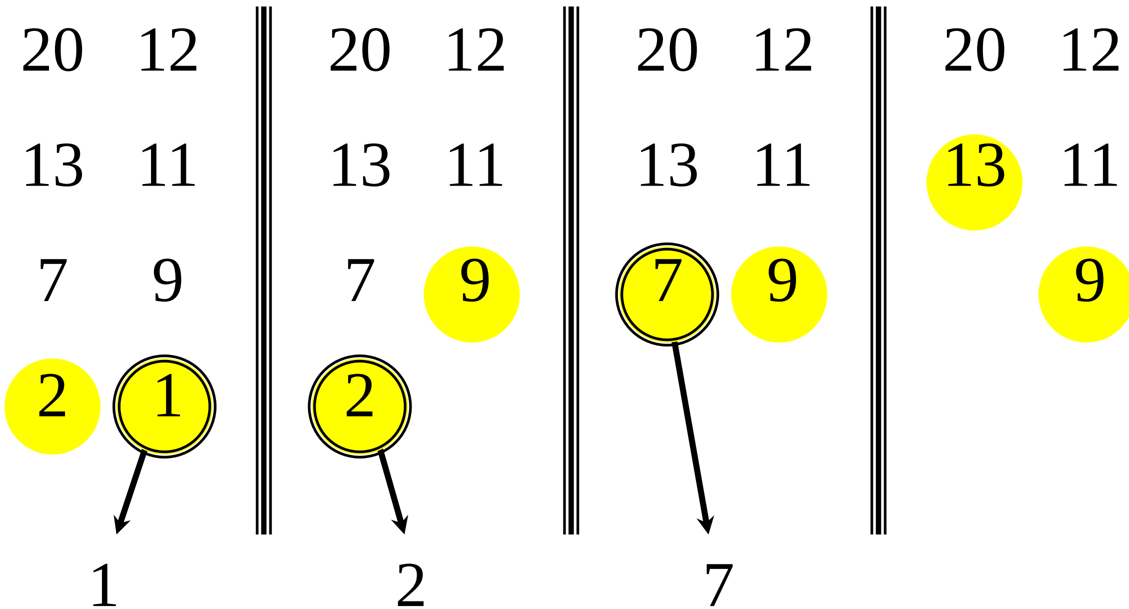
Merging two sorted arrays



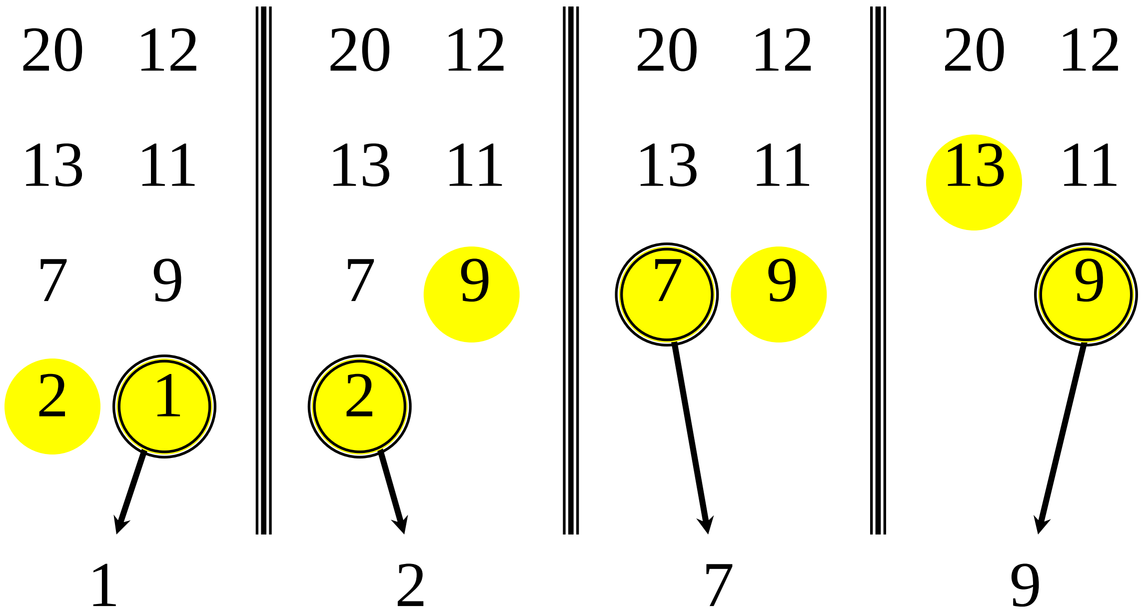
Merging two sorted arrays



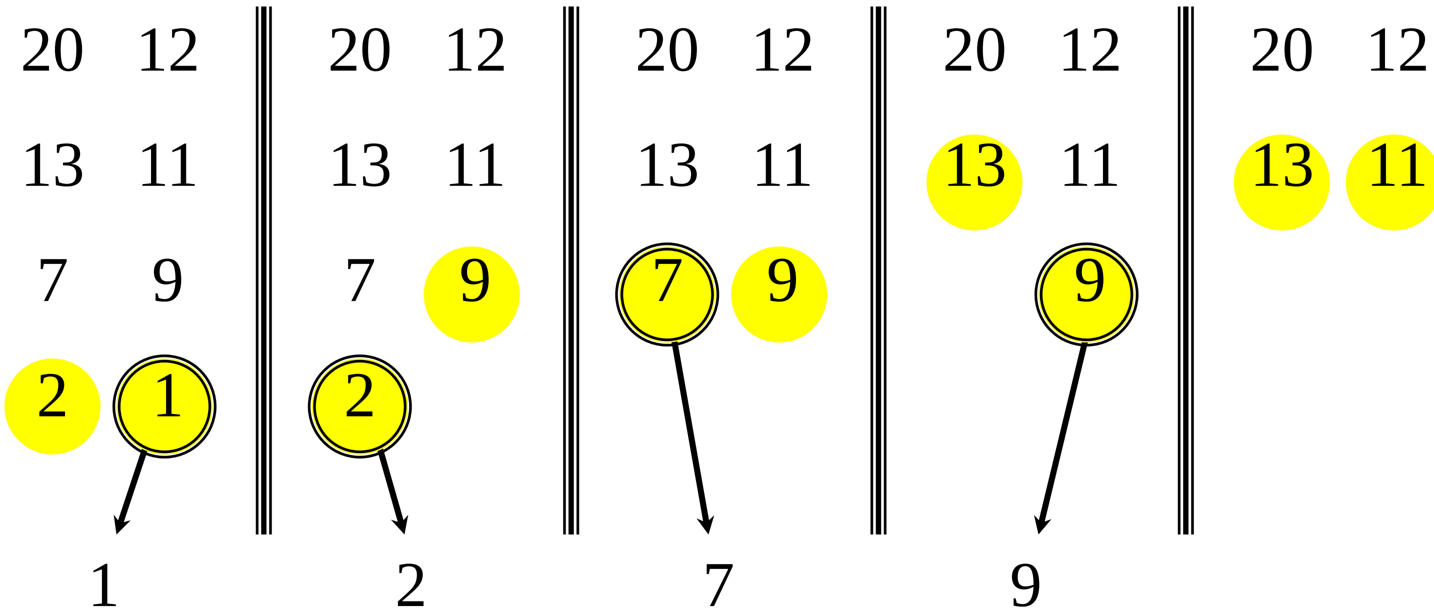
Merging two sorted arrays



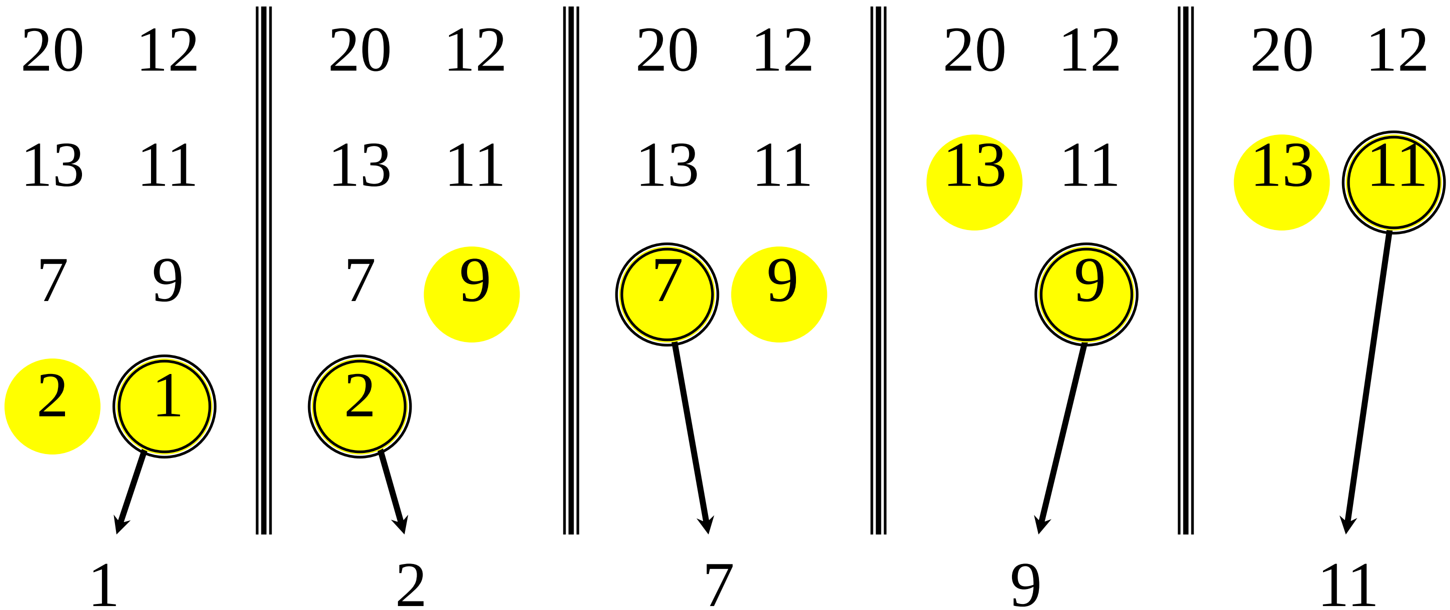
Merging two sorted arrays



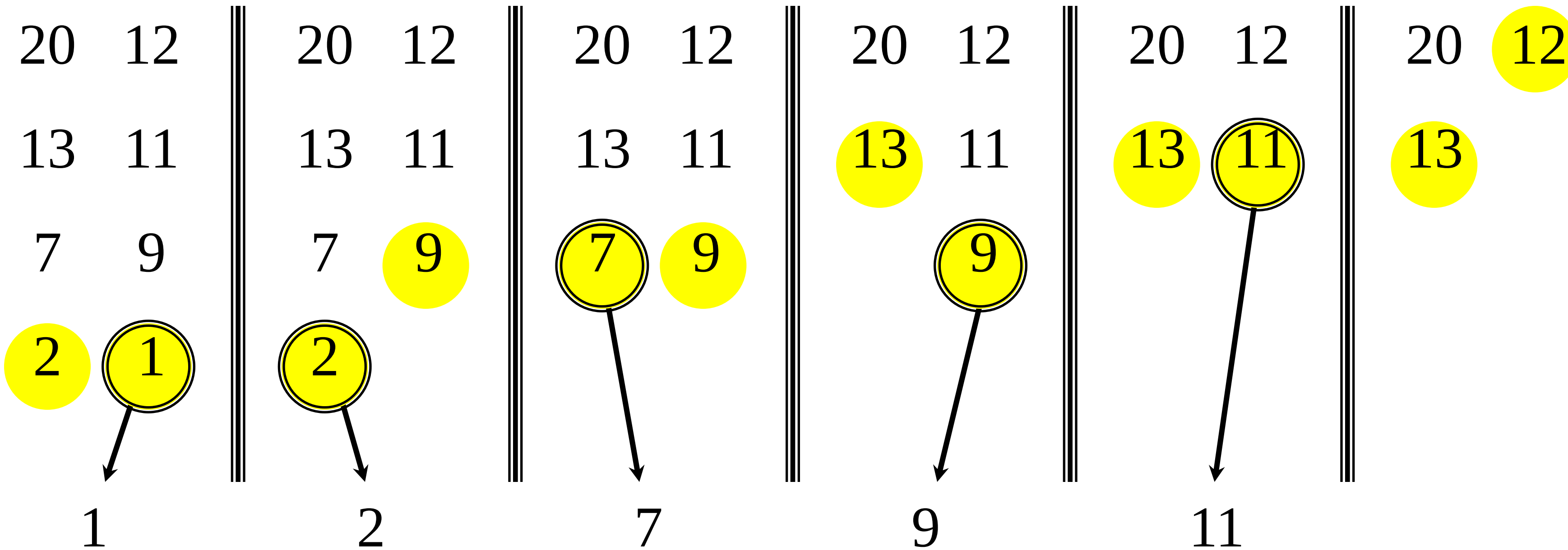
Merging two sorted arrays



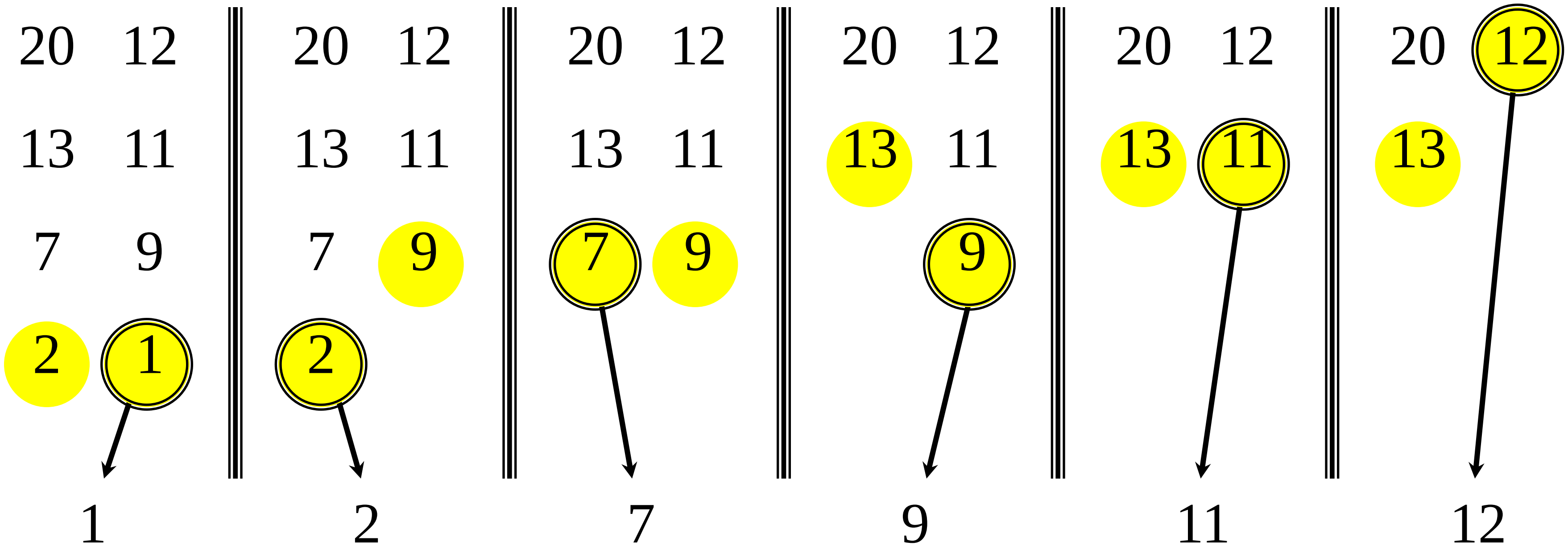
Merging two sorted arrays



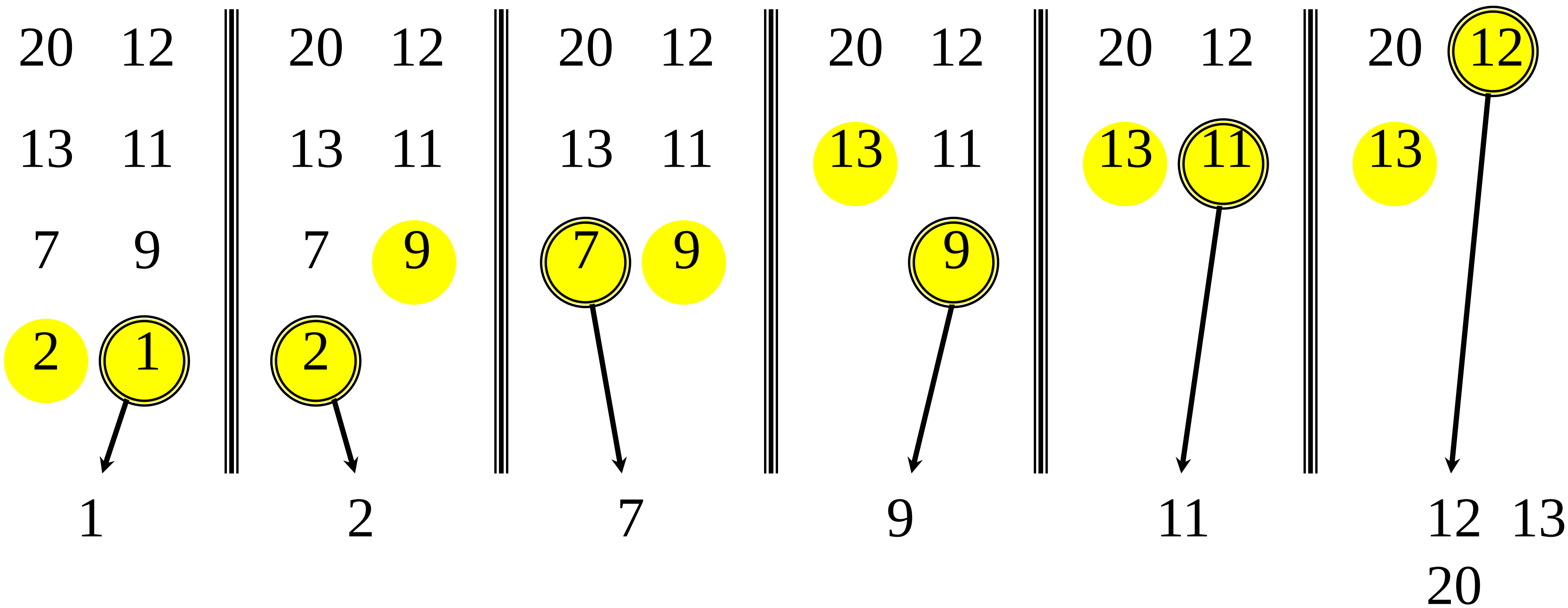
Merging two sorted arrays



Merging two sorted arrays

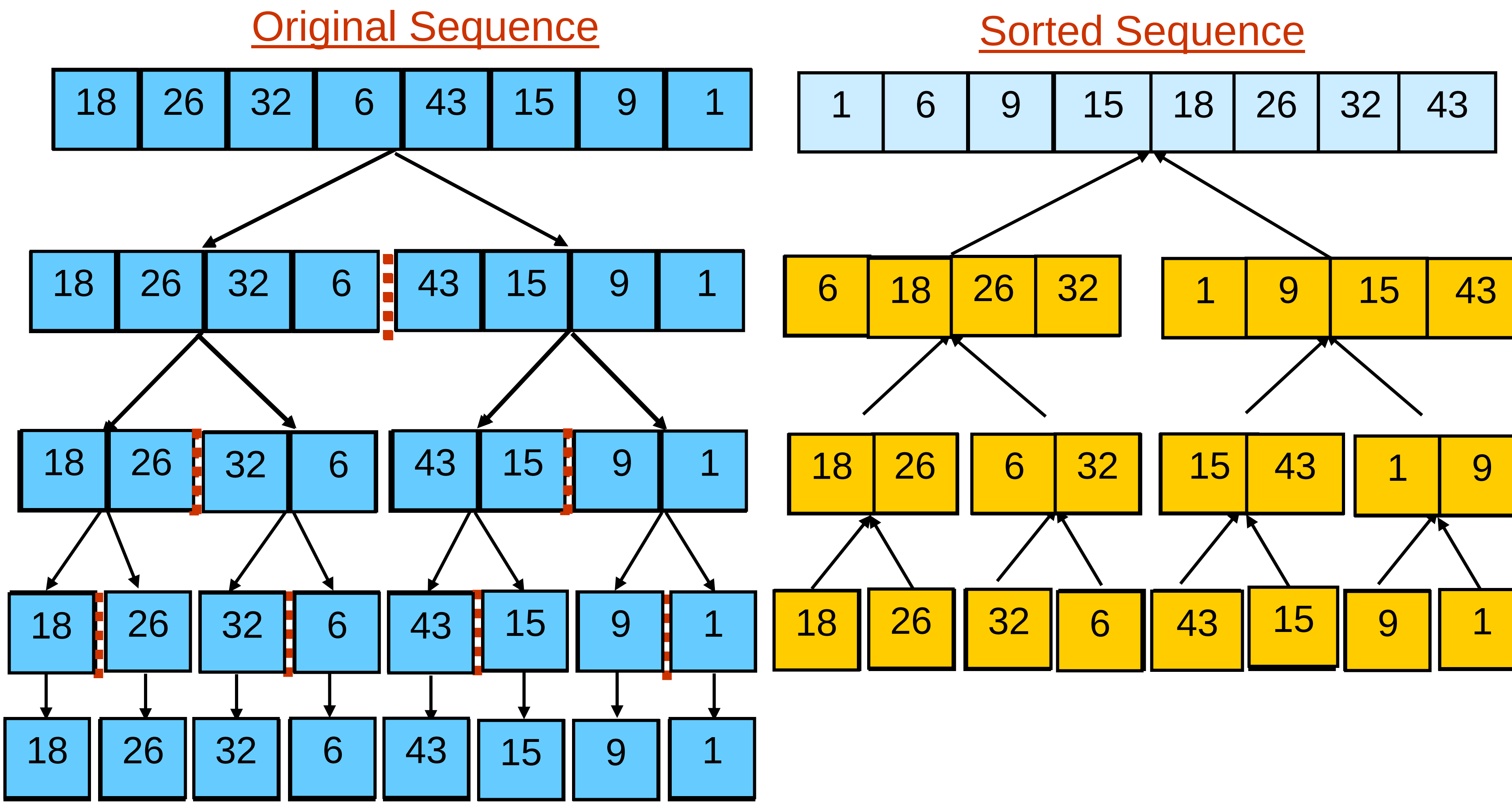


Merging two sorted arrays



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

Merge Sort – Example



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 $A[\,\lceil n/2\rceil+1\,..\,n\,]$.
$O(n)$	3. ***"Merge"*** 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

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

- 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.
- Week 2 provides 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.

$$cn$$

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

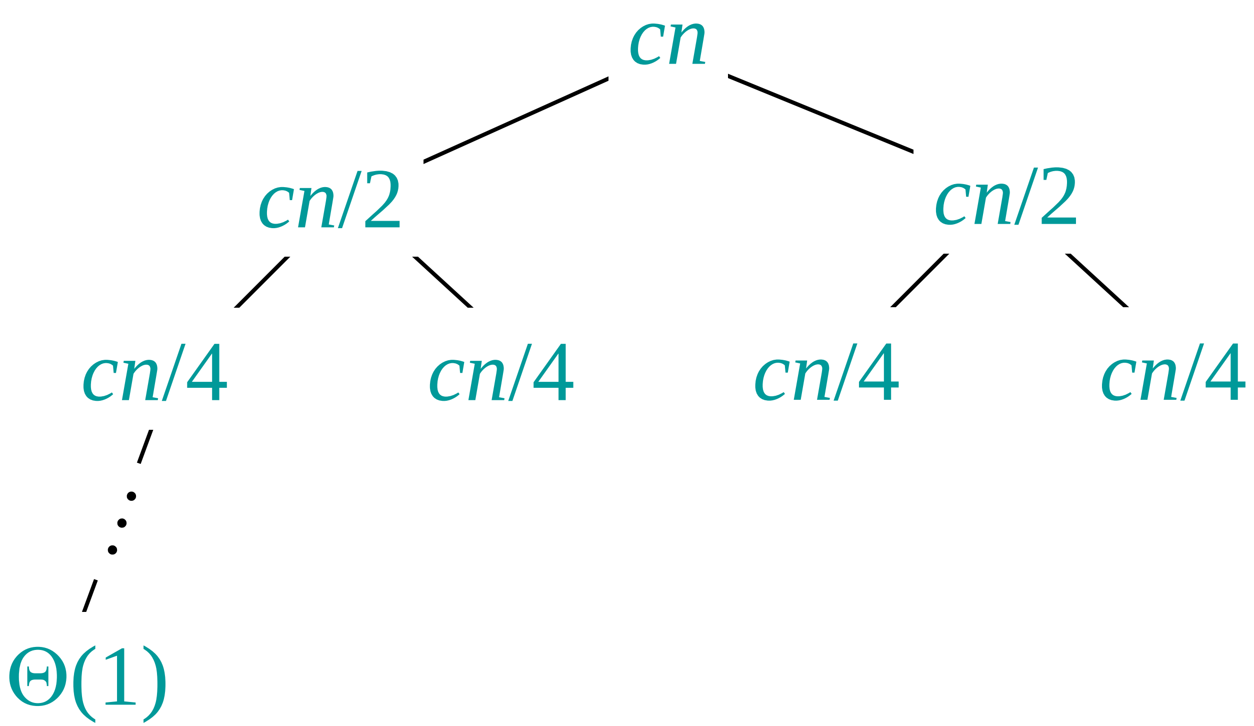
Recursion tree

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

$$
\begin{array}{cccc}
 & cn & & \\
cn/2 & & & cn/2 \\
T(n/4) & T(n/4) & T(n/4) & T(n/4)
\end{array}
$$

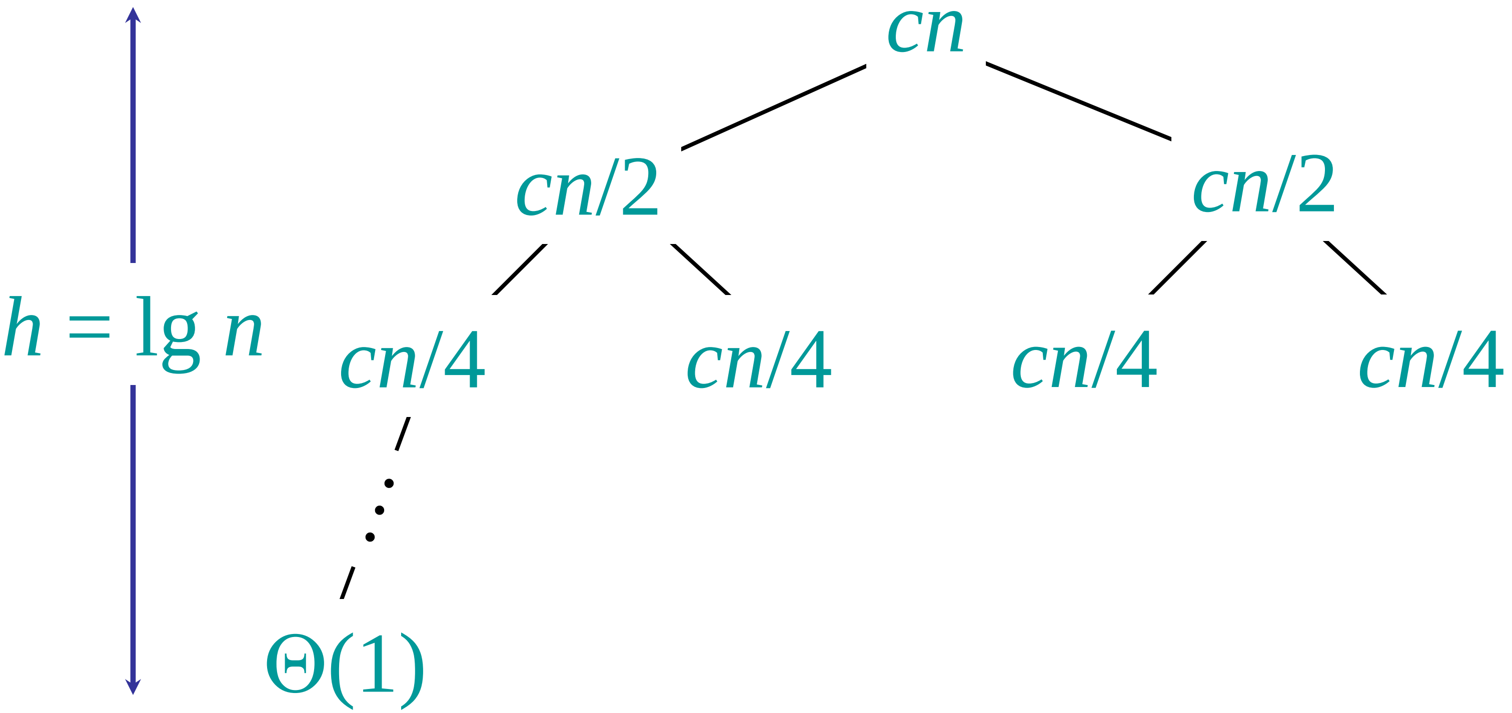
Recursion tree

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



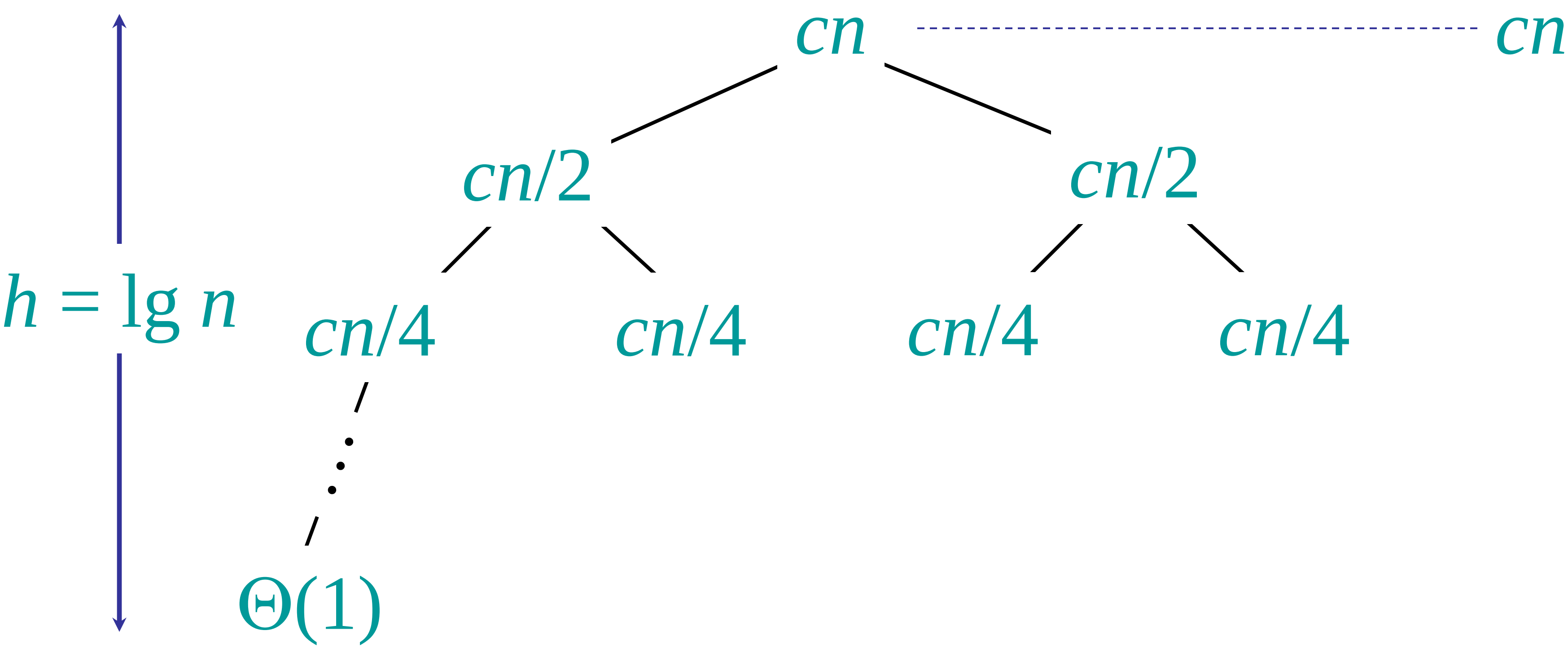
Recursion tree

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



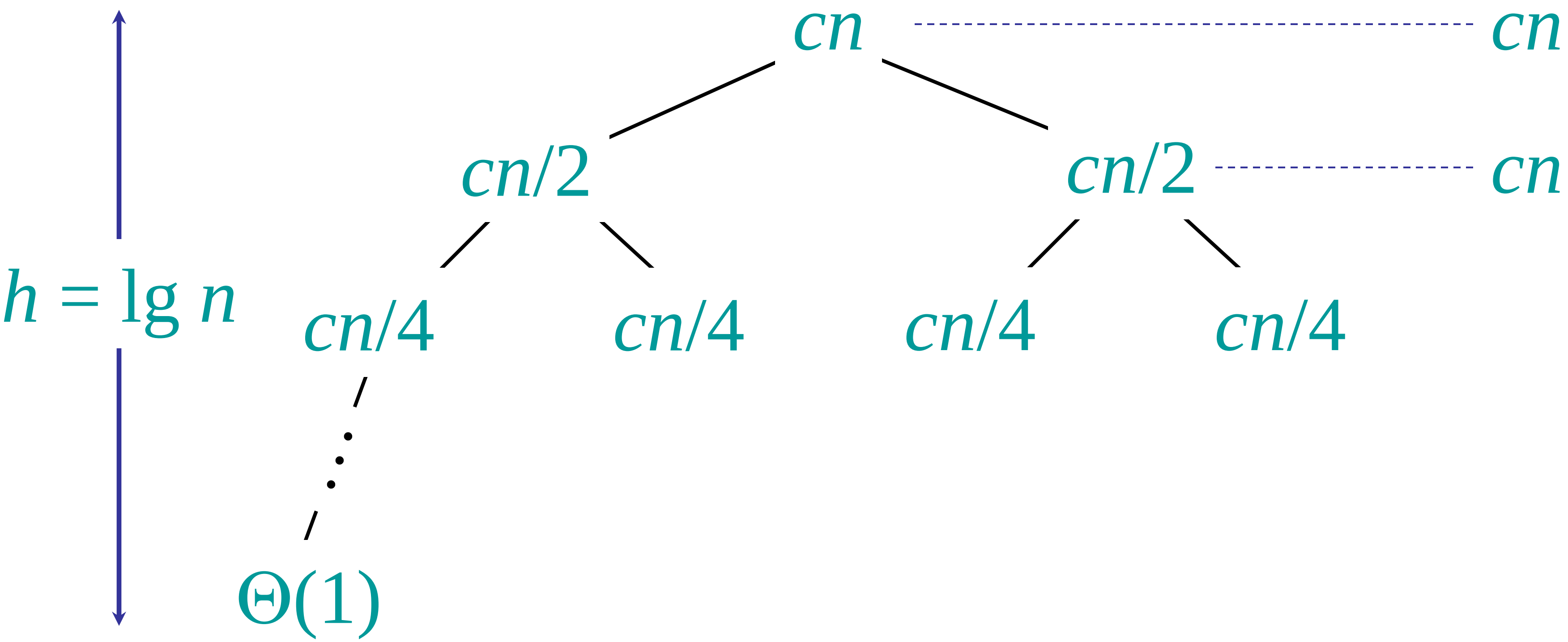
Recursion tree

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



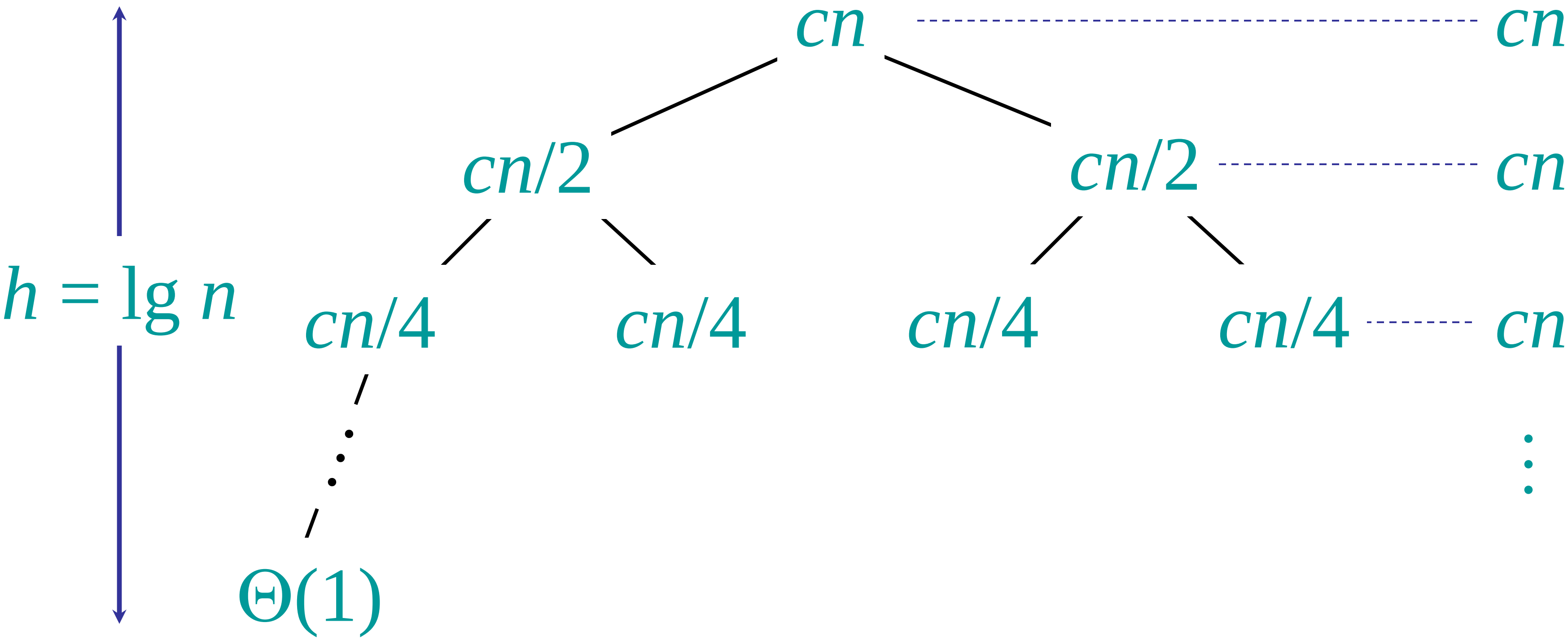
Recursion tree

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



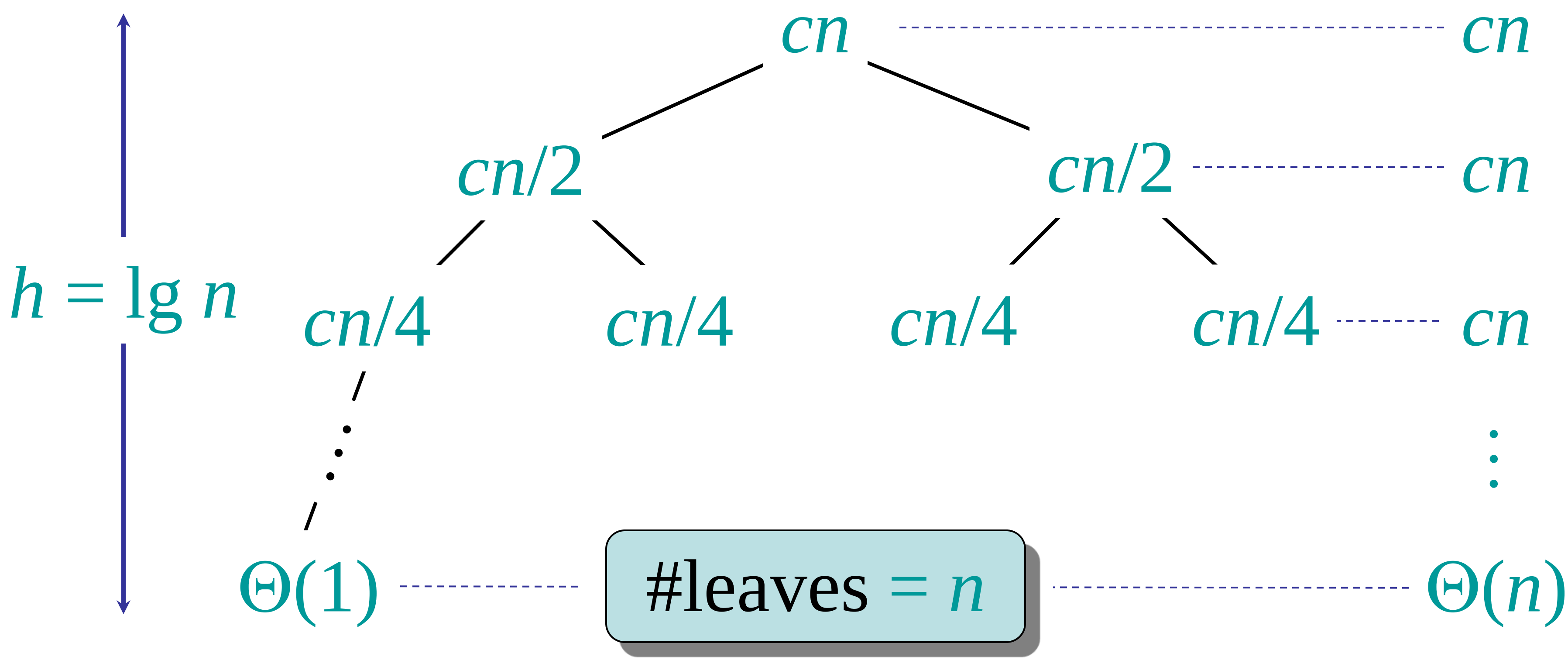
Recursion tree

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



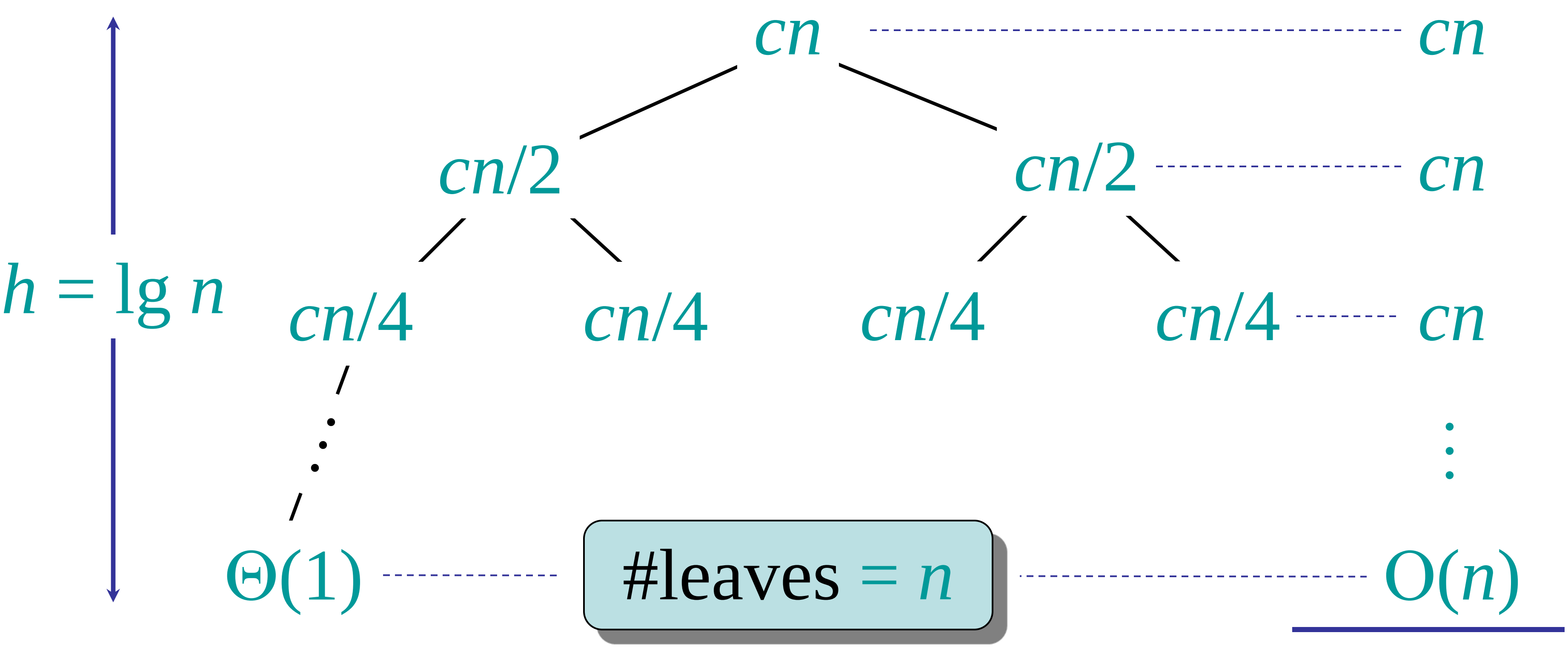
Recursion tree

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 $= O(n \lg n)$

Conclusions

- $O(n \lg n)$ grows more slowly than $O(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.

- *Nearly Sorted??*

A tighter bound

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.

More about Theta Θ in the next lecture.