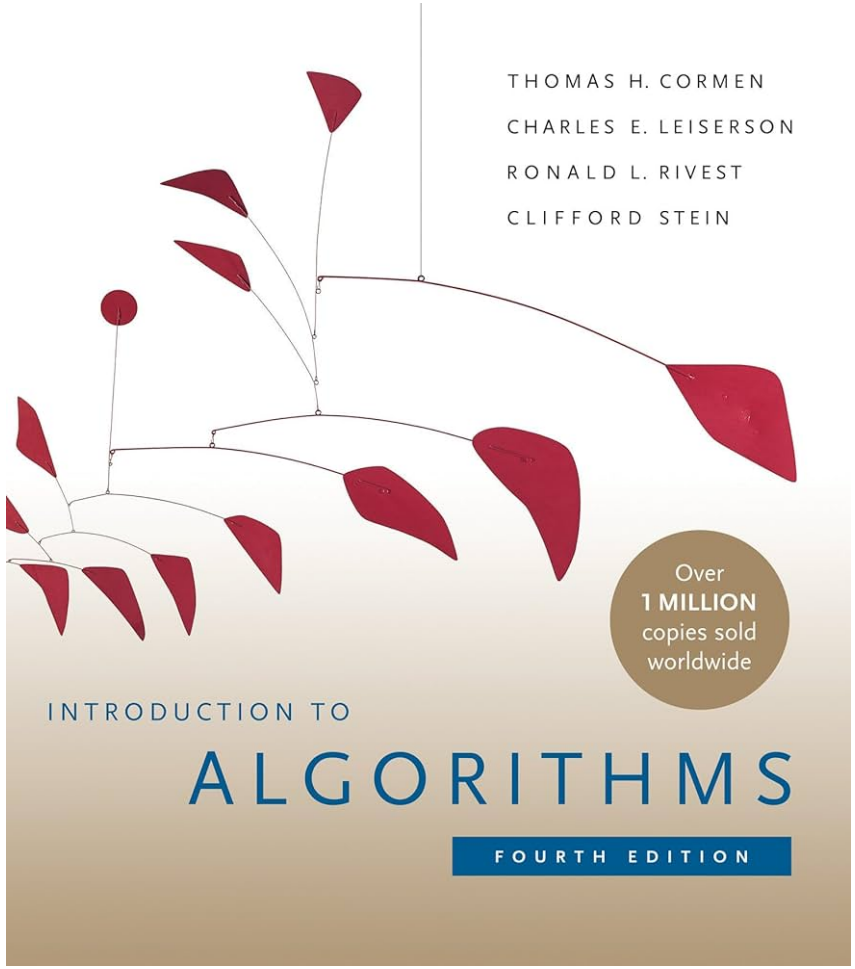
Introduction to Algorithms
Lecture 1: Analysis of Algorithms

Prof. Charles E. Leiserson and Prof. Erik Demaine
Massachusetts Institute of Technology

February 11, 2025

Introduction to Algorithms



Content has been extracted from *Introduction to Algorithms*, Fourth Edition, by Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.
Visit `https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/`.
Original slides from *Introduction to Algorithms 6.046J/18.401J*, Fall 2005 Class by Prof. Charles Leiserson and Prof. Erik Demaine. MIT OpenCourseWare Initiative available at `https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/`.

Plan

Analysis of Algorithms

Insertion-sort

Asymptotic Analysis

Merge-sort

Recurrences

Analysis of Algorithms

- The theoretical study of computer-program performance and resource usage.
- But... What's more important than performance?

Analysis of Algorithms

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

Plan

The problem of sorting

Input: sequence $<a_1, a_2, \cdots, a_n>$ of numbers.
Output: permutation $<a'_1, a'_2, \cdots, a'_n>$ 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

Algorithm 1 Insertion Sort

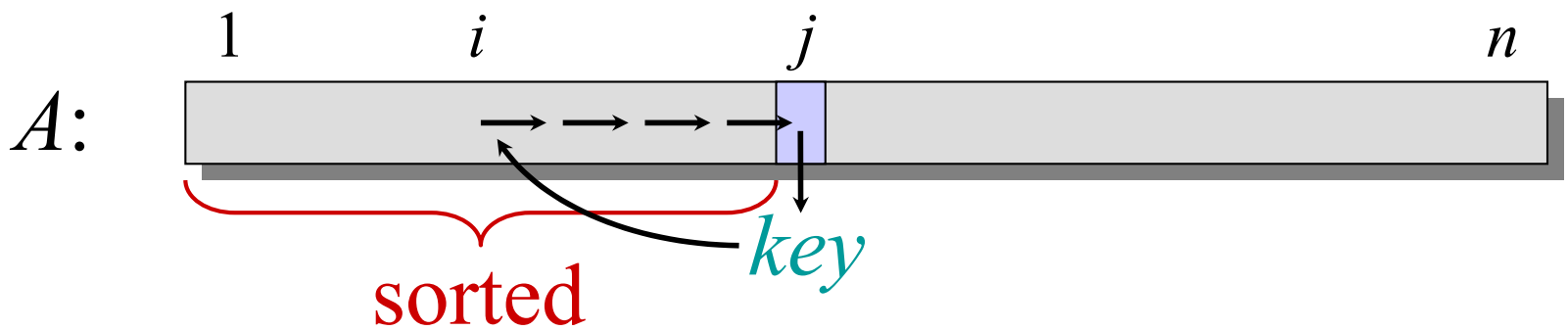
```
1: INSERTION-SORT(A, n)        ▷ A[1 . . . n]
2:     for j ← 2 to n do
3:         key ← A[j]
4:         i ← j − 1
5:         while i > 0 and A[i] > key do
6:             A[i + 1] ← A[i]
7:             i ← i − 1
8:         A[i + 1] ← key
```

"pseudocode"

Insertion Sort

Algorithm 2 Insertion Sort

```
1: INSERTION-SORT(A, n)        ▷ A[1 . . . n]
2:     for j ← 2 to n do
3:         key ← A[j]
4:         i ← j − 1
5:         while i > 0 and A[i] > key do
6:             A[i + 1] ← A[i]
7:             i ← i − 1
8:         A[i + 1] ← key
```



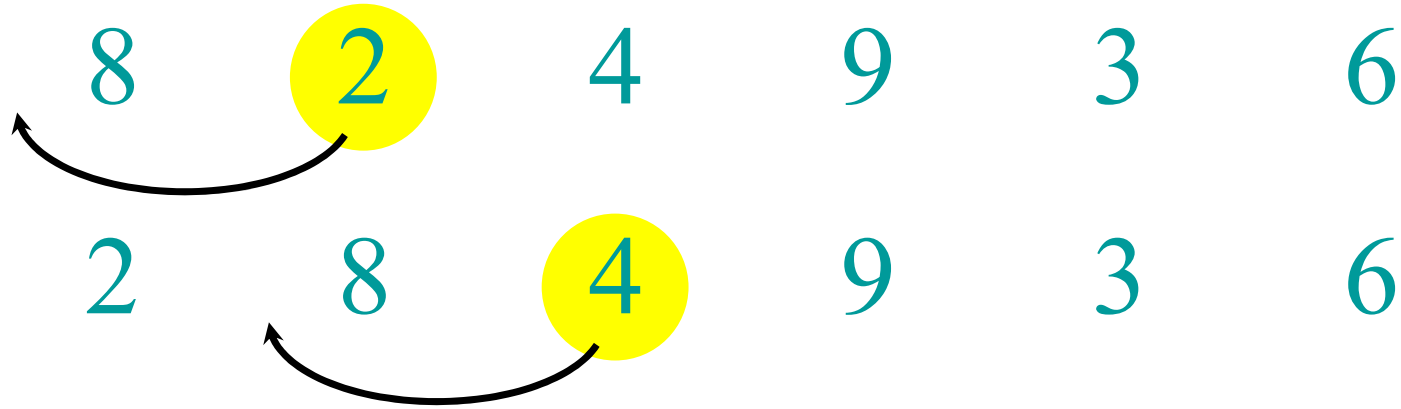
Example of insertion sort

8 2 4 9 3 6

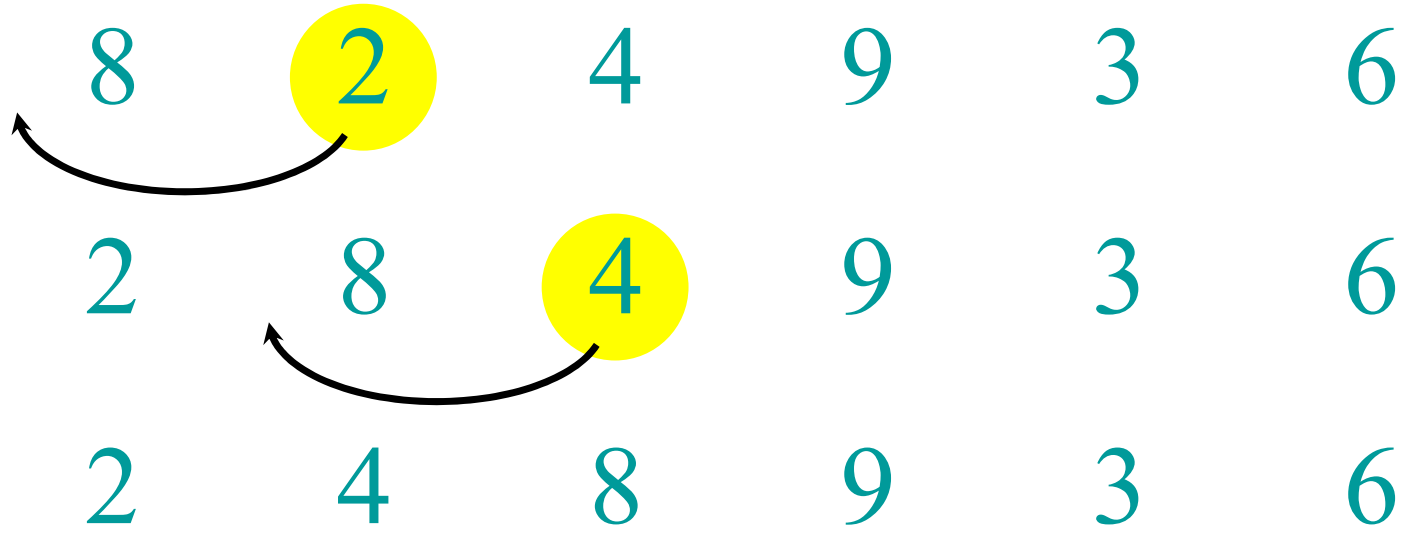
Example of insertion sort

Example of insertion sort

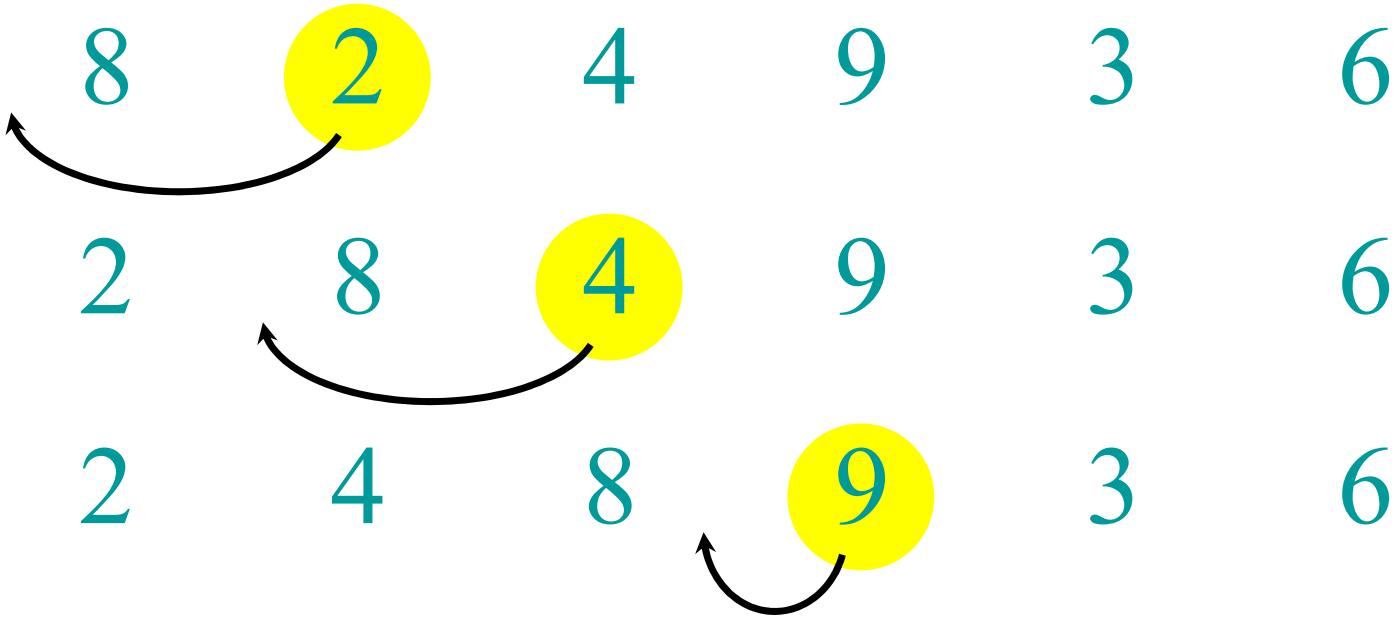
Example of insertion sort



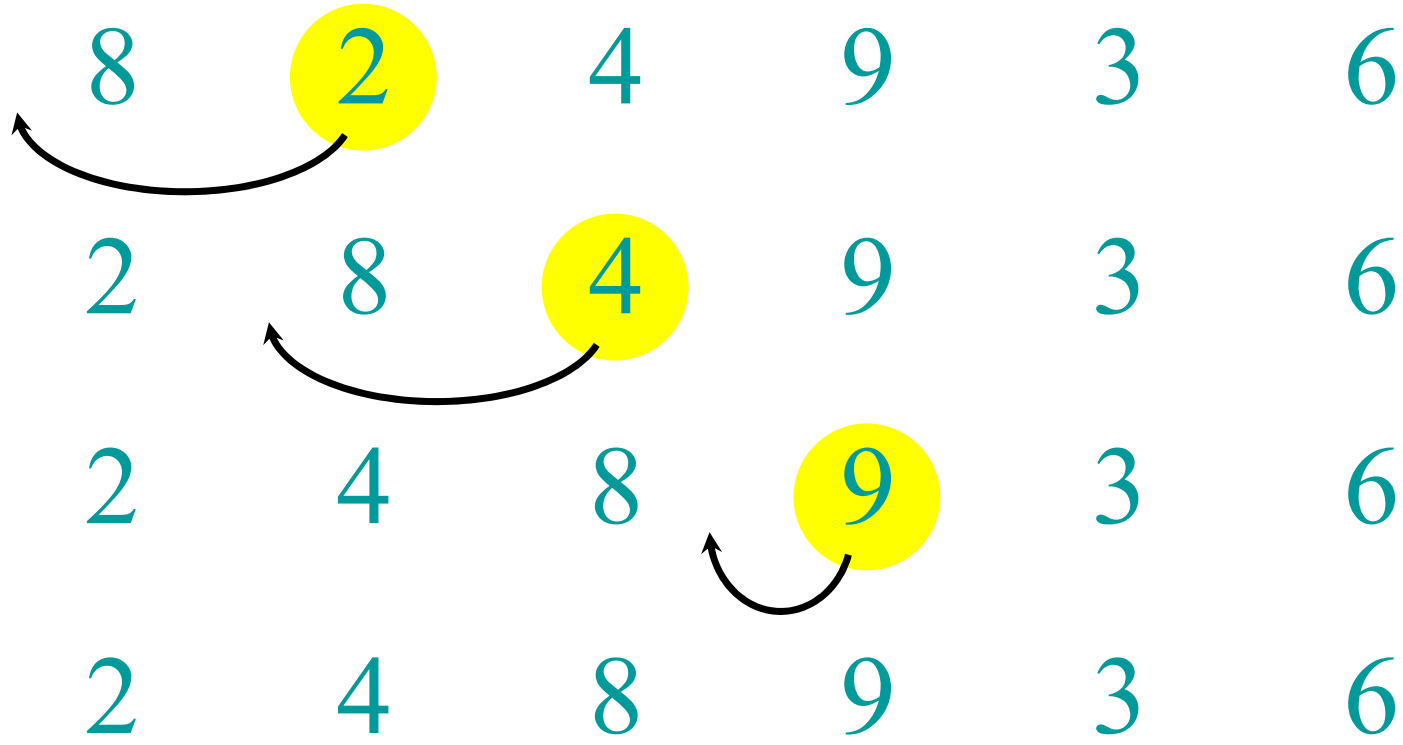
Example of insertion sort



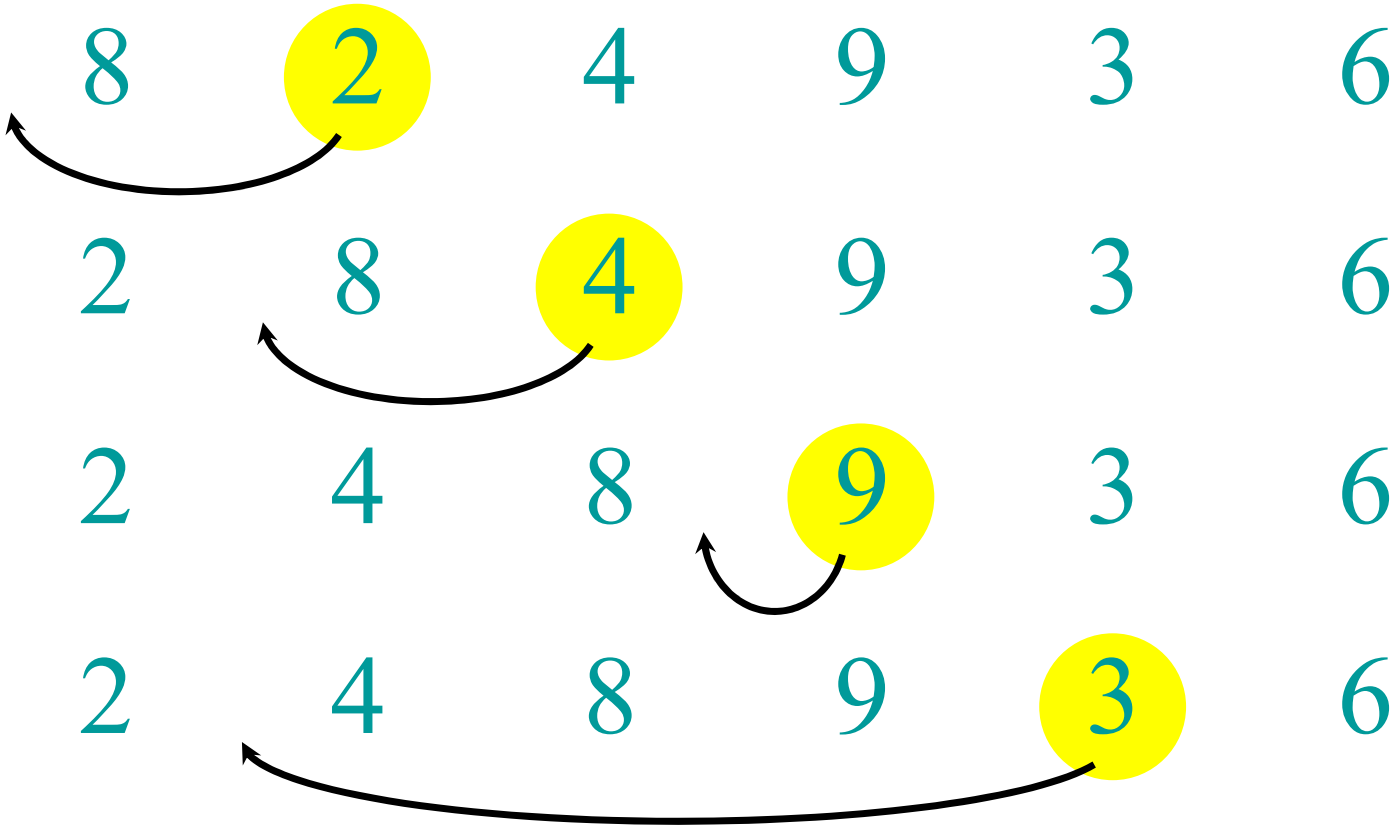
Example of insertion sort



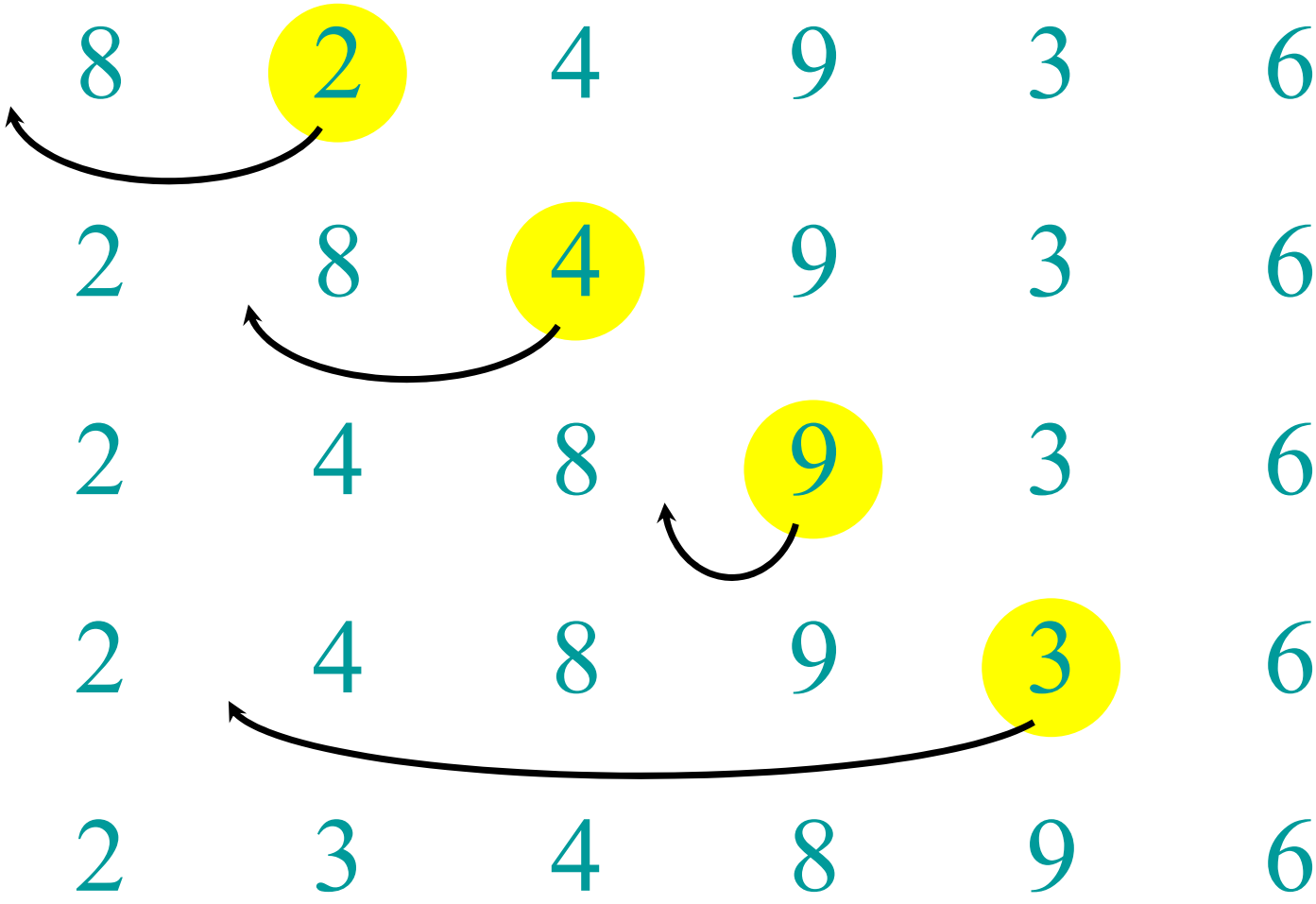
Example of insertion sort



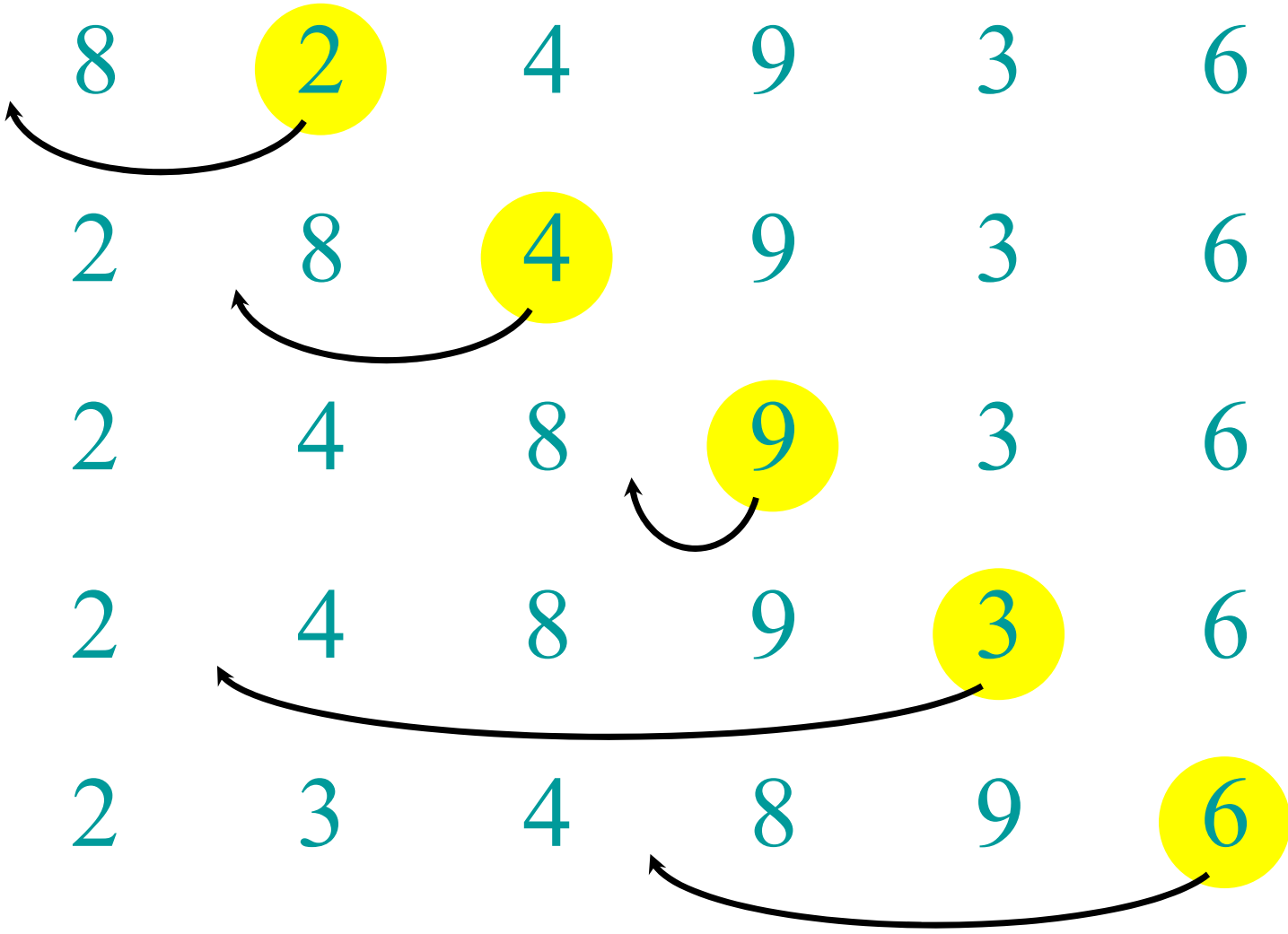
Example of insertion sort



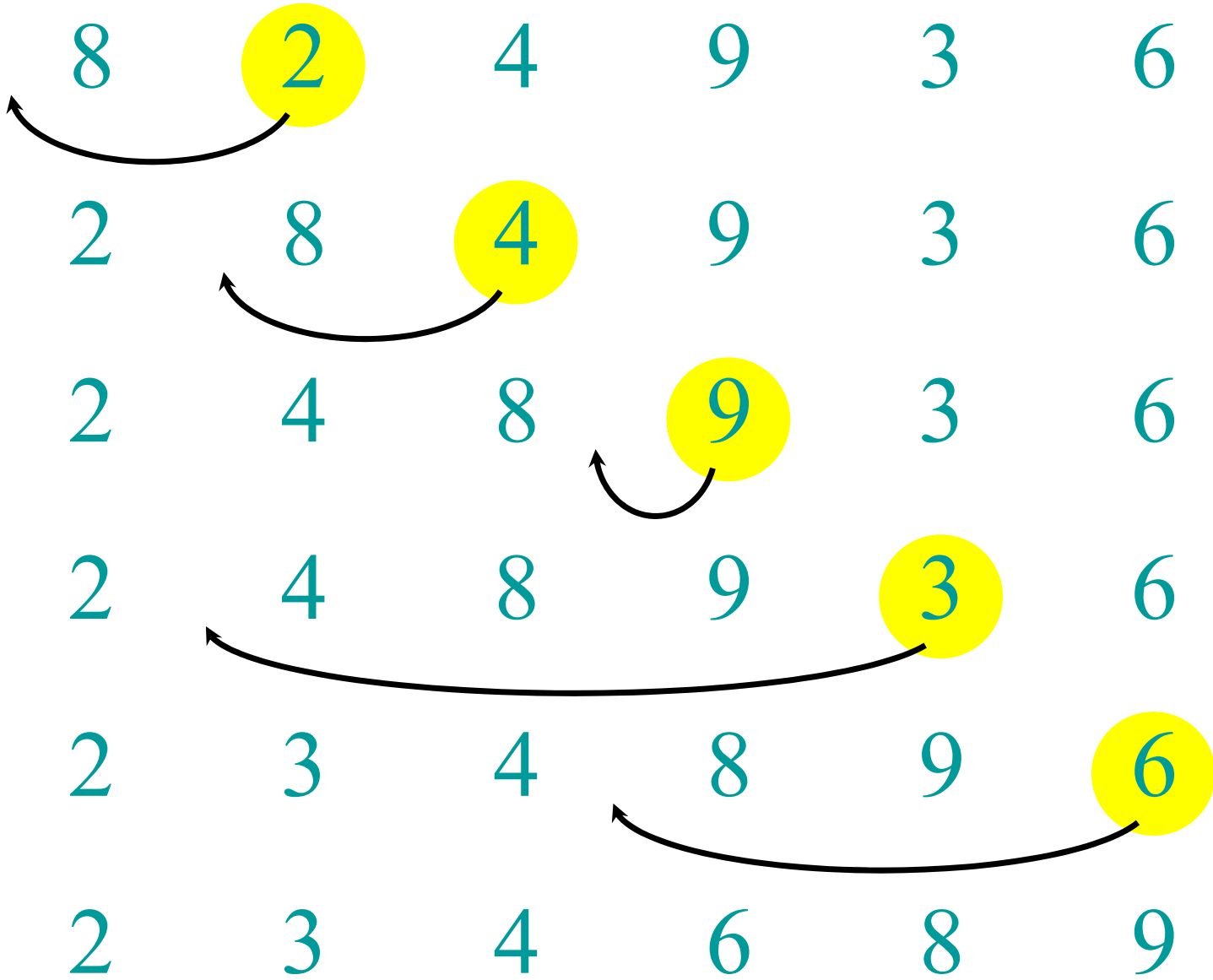
Example of insertion sort



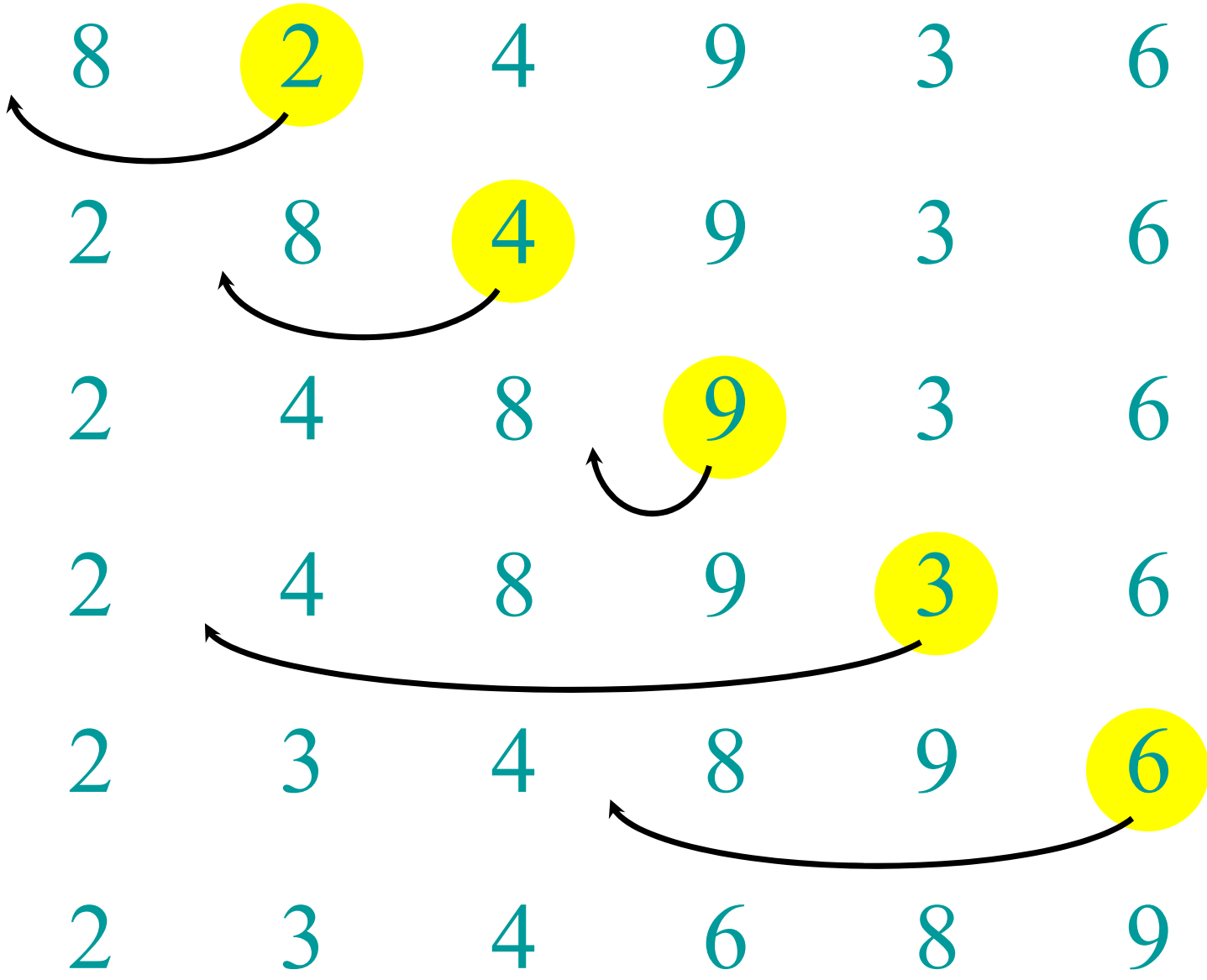
Example of insertion sort



Example of insertion sort



Example of insertion sort



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.

Plan

Analysis of Algorithms

Insertion-sort

Asymptotic Analysis

Merge-sort

Recurrences

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 $n \to \infty$

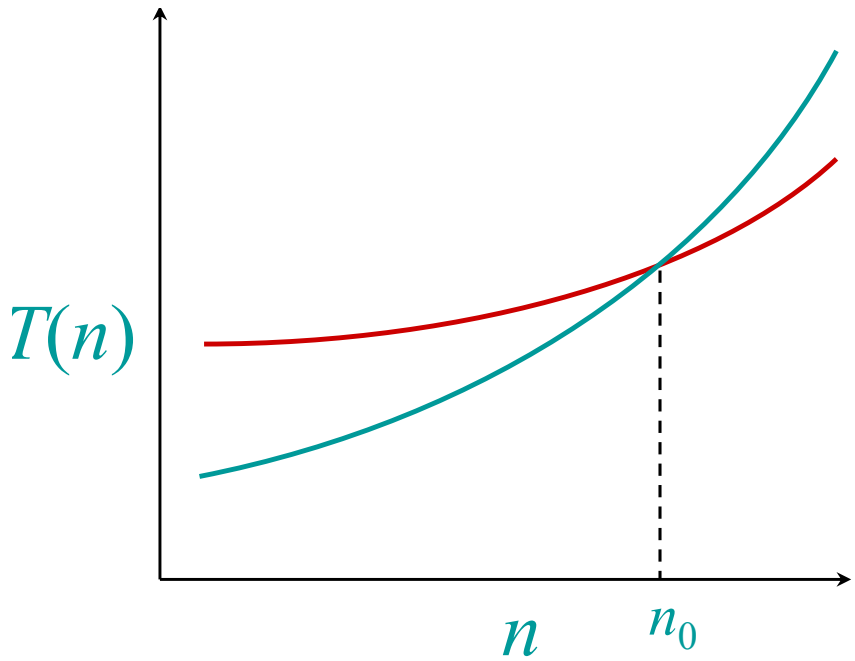
"Asymptotic Analysis"

Θ-notation

- Math:
  - $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$
- Engineering:
  - Drop low-order terms; ignore leading constants.
  - For instance: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Asymptotic Performance

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



- We should not 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.
  - $T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$.
  - Arithmetic series!
- **Average case:**
  - All permutations equally likely.
  - $T(n) = \sum_{j=2}^{n} \Theta(\frac{j}{2}) = \Theta(n^2)$.
- *¿Is insertion sort a fast sorting algorithm?*
  - Moderately so, for small n.
  - Not at all, for large n.

Plan

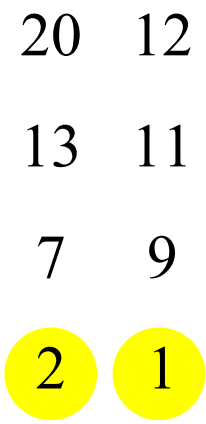
Merge-sort

Algorithm 3 Insertion Sort

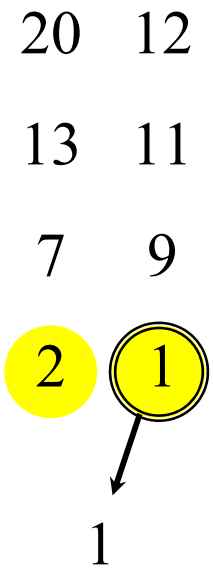
1: MERGE-SORT(A, n) $\quad \triangleright A[1 \ldots n]$
2: **if** $n = 1$, done.
3: Recursively sort $A[1 \ldots \lceil \frac{n}{2} \rceil]$ and $A[\lceil \frac{n}{2} \rceil + 1 \ldots n]$
4: MERGE the 2 sorted list.

Key subroutine: MERGE

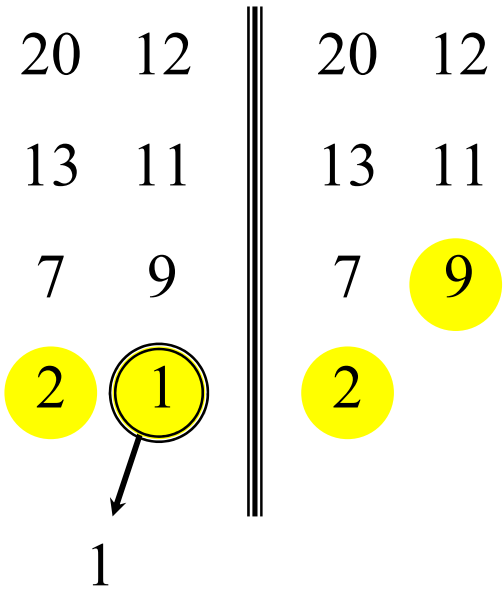
Merging two sorted arrays



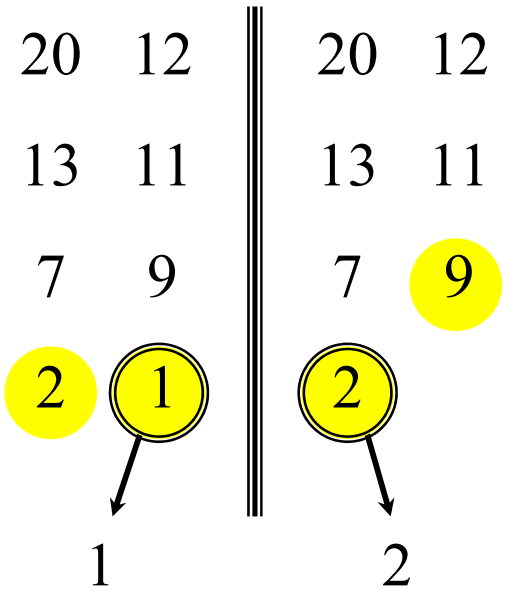
Merging two sorted arrays



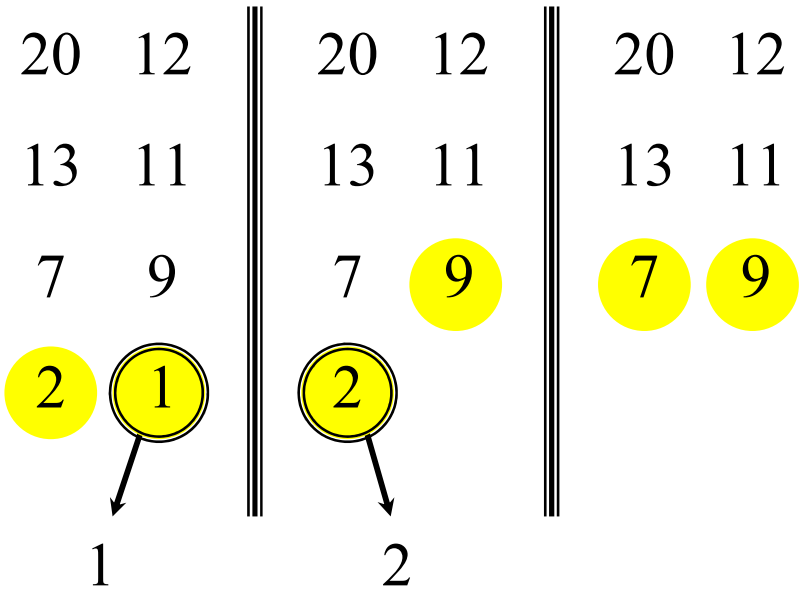
Merging two sorted arrays



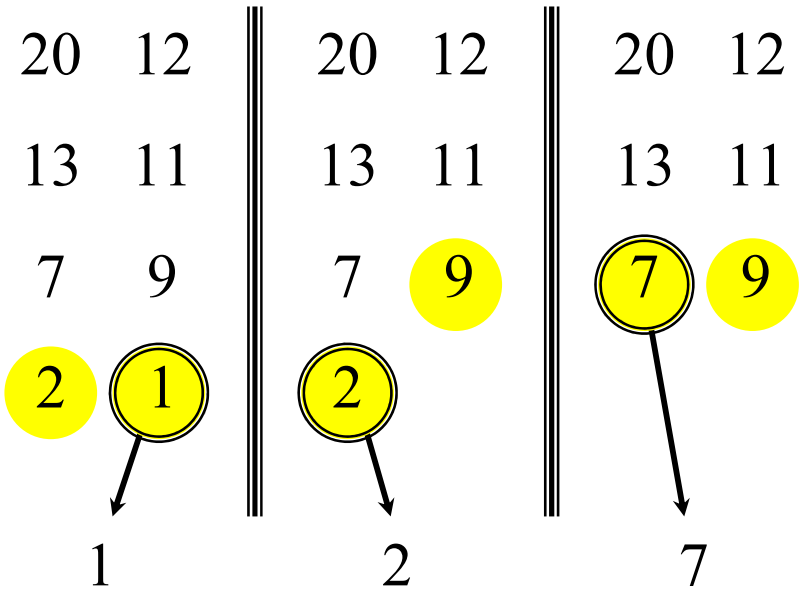
Merging two sorted arrays



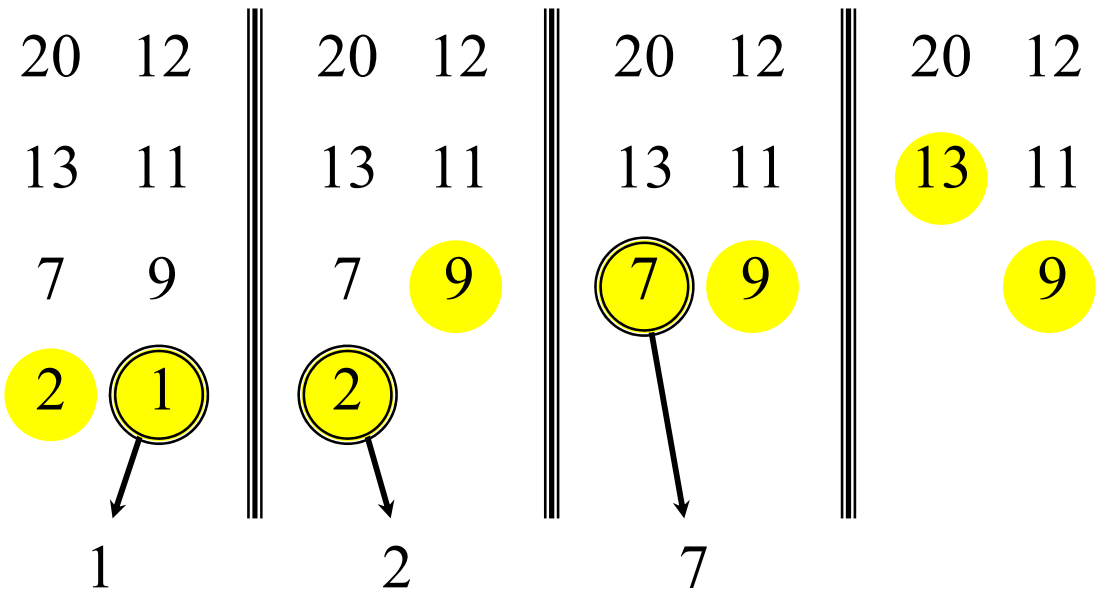
Merging two sorted arrays



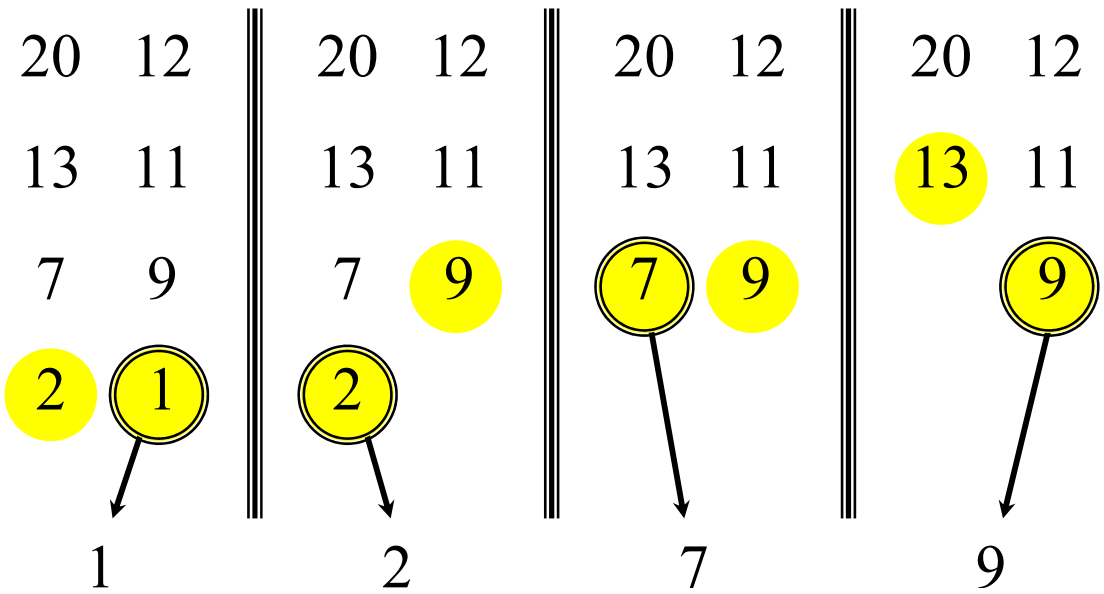
Merging two sorted arrays



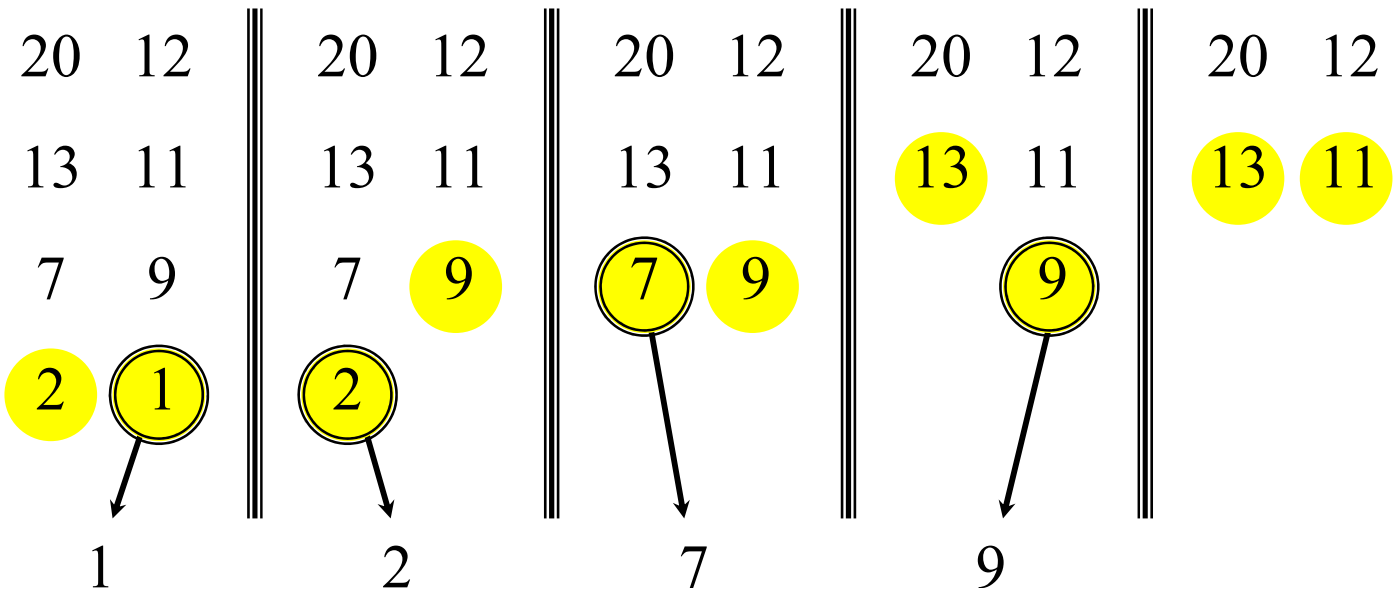
Merging two sorted arrays



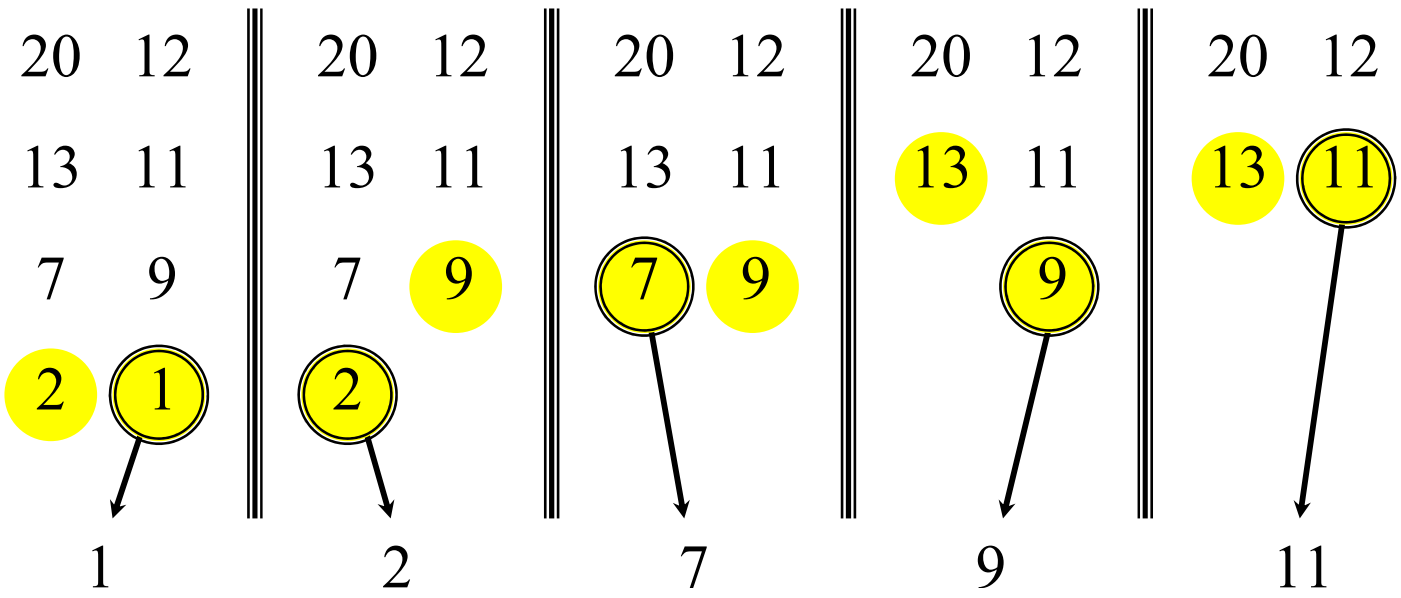
Merging two sorted arrays



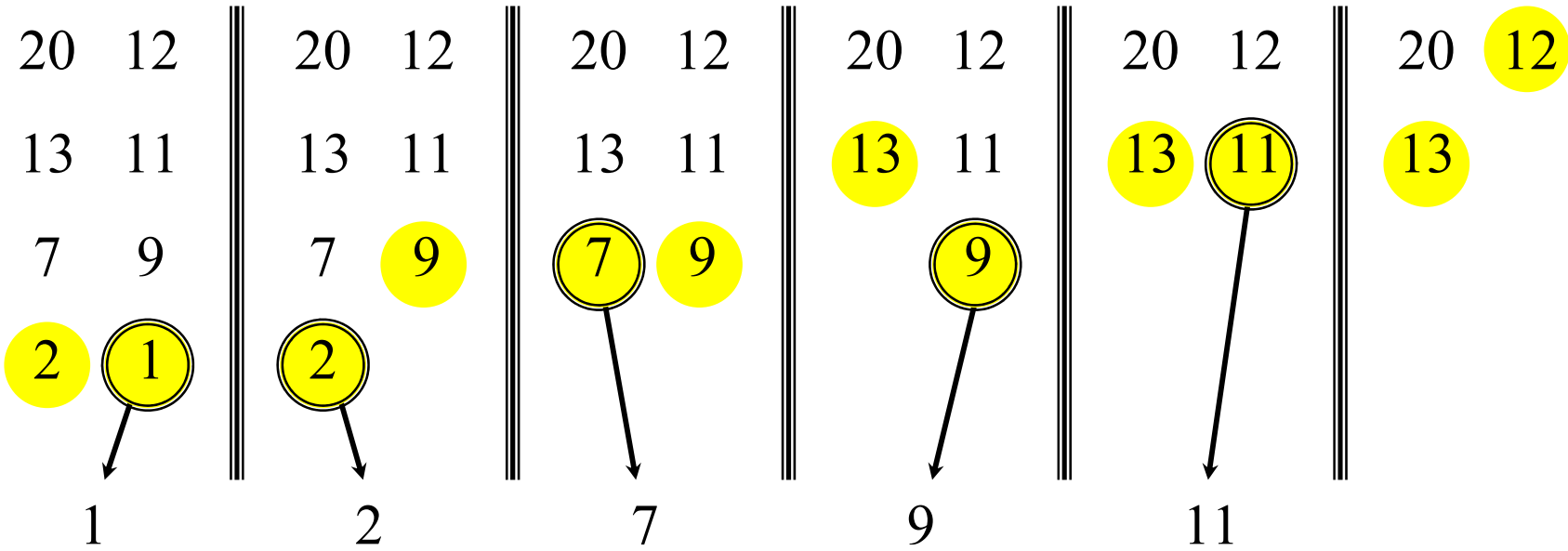
Merging two sorted arrays



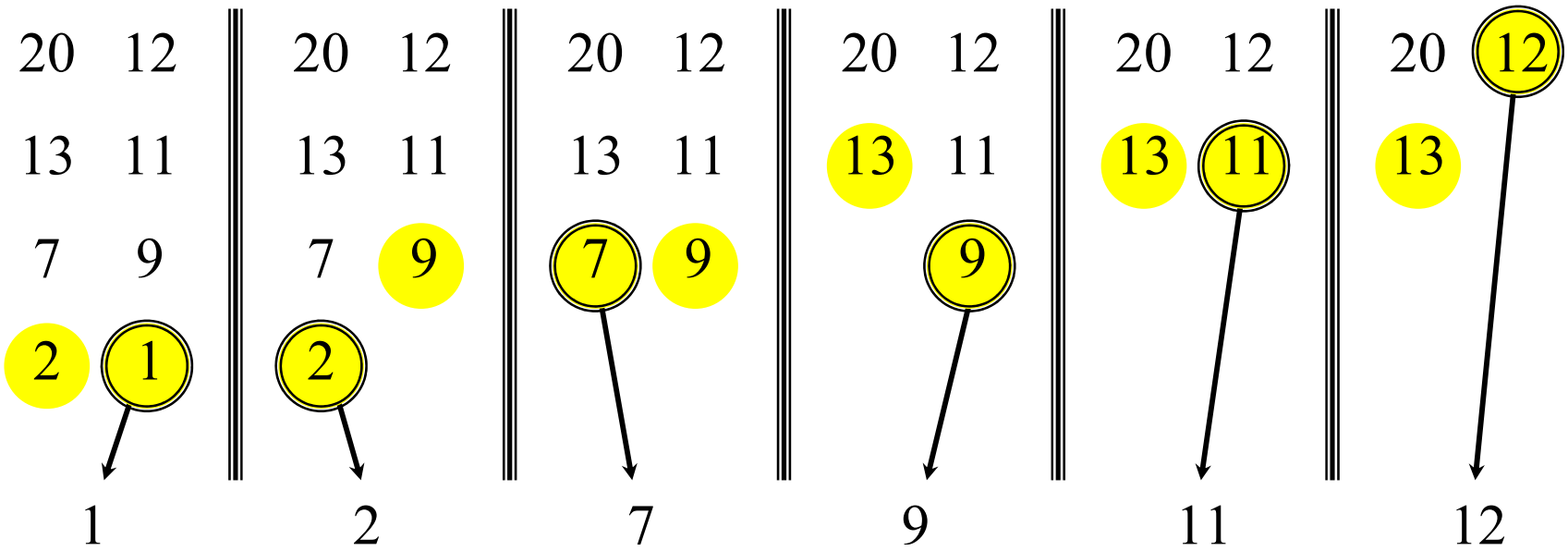
Merging two sorted arrays



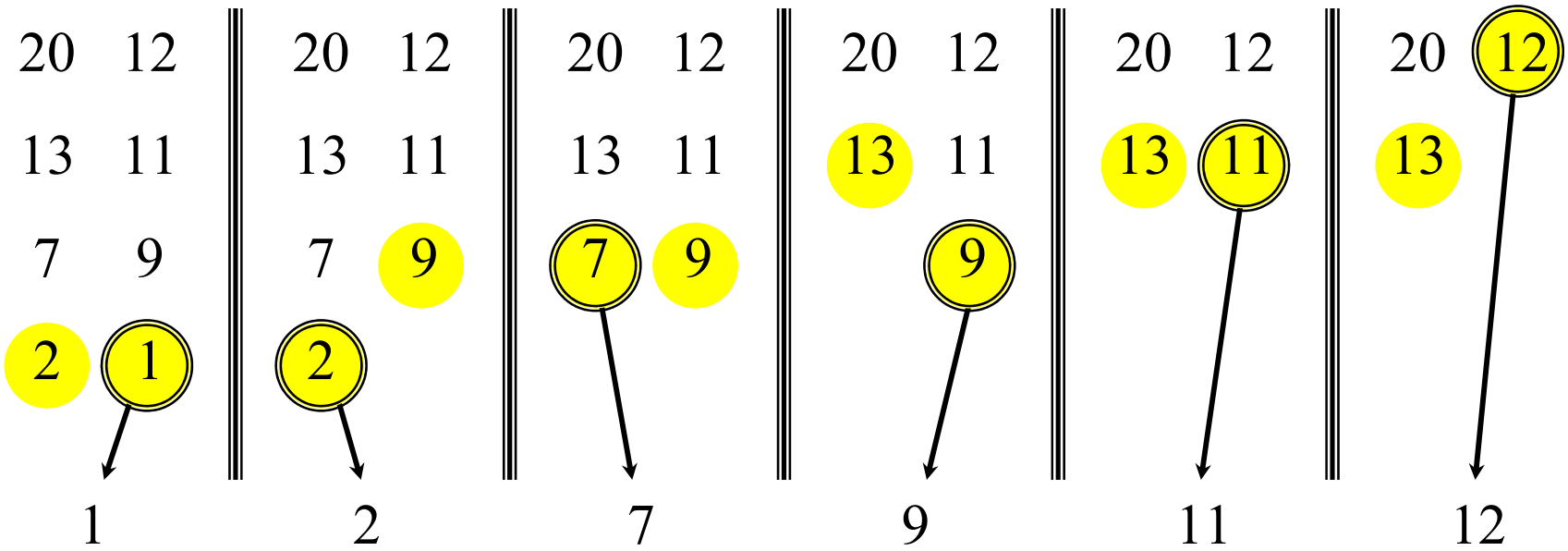
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



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

Analyzing merge-sort

Abuse!

$T(n)$	MERGE-SORT $A[1 \ldots n]$
$\Theta(1)$	1. **if** $n = 1$, done!
$2T(\frac{n}{2})$	2. Recursively sort $A[1 \ldots \lceil \frac{n}{2} \rceil]$ and $A[\lceil \frac{n}{2} \rceil + 1 \ldots n]$.
$\Theta(n)$	3. MERGE the 2 sorted lists.

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

Plan

Recurrence for merge-sort

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

- 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[1] and Lecture 2 provide several ways to find a good upper bound on $T(n)$.

[1] The book

Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.

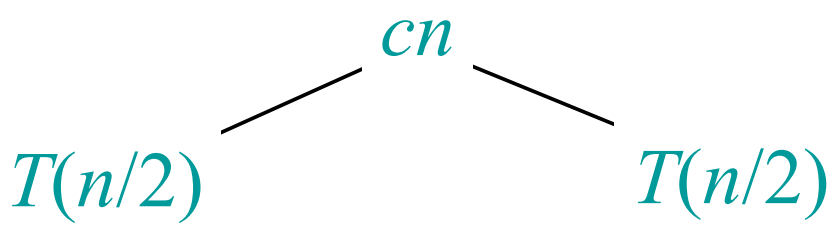
Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.

$T(n)$

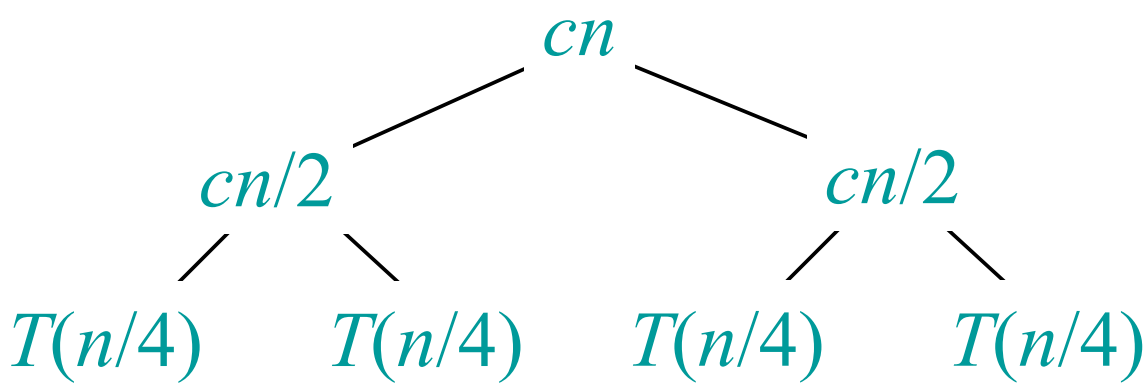
Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.



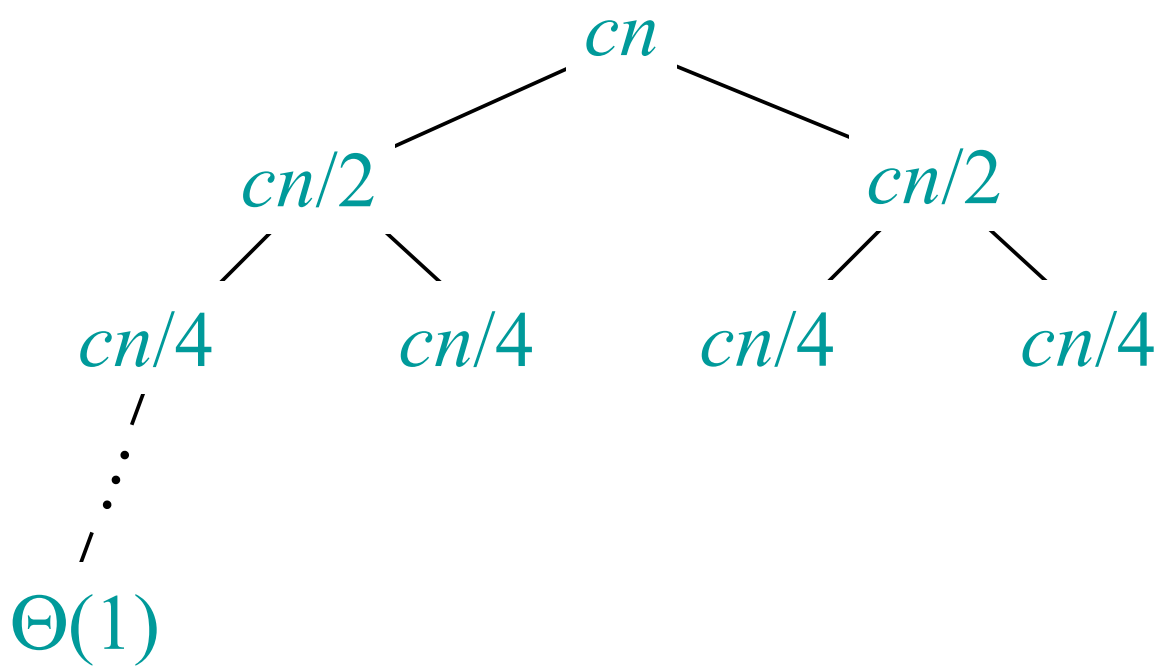
Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.



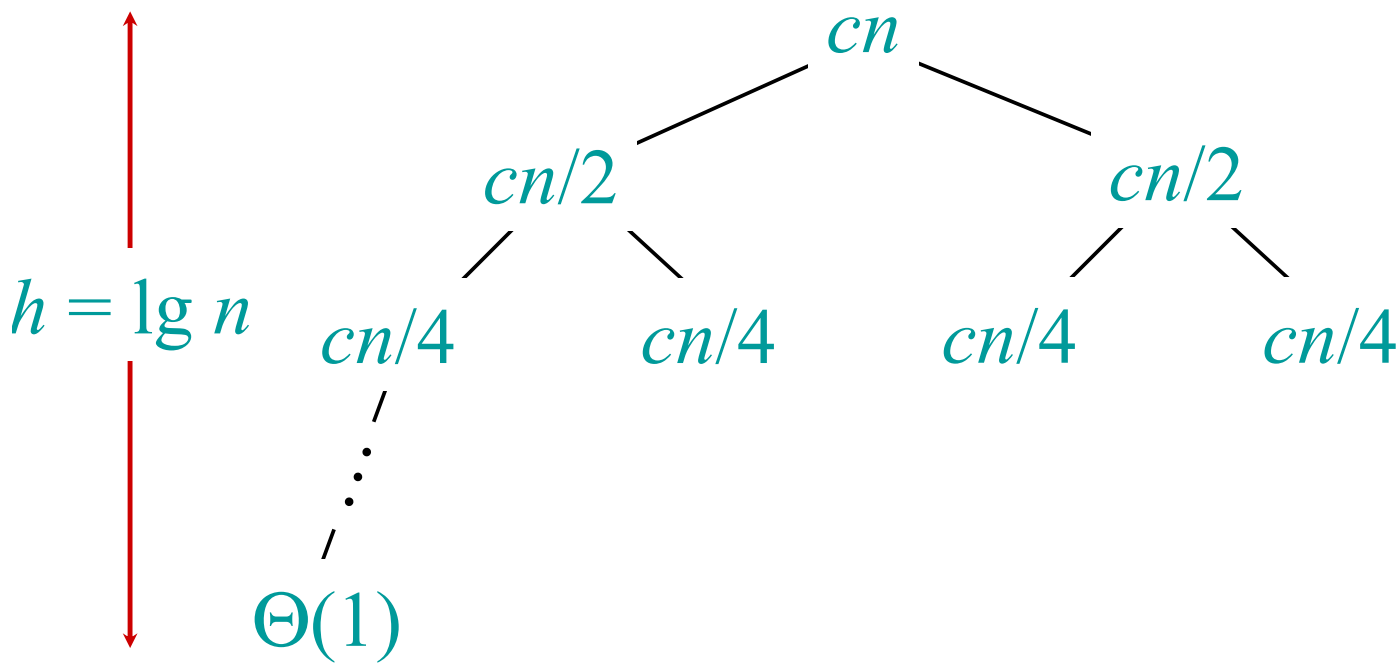
Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.



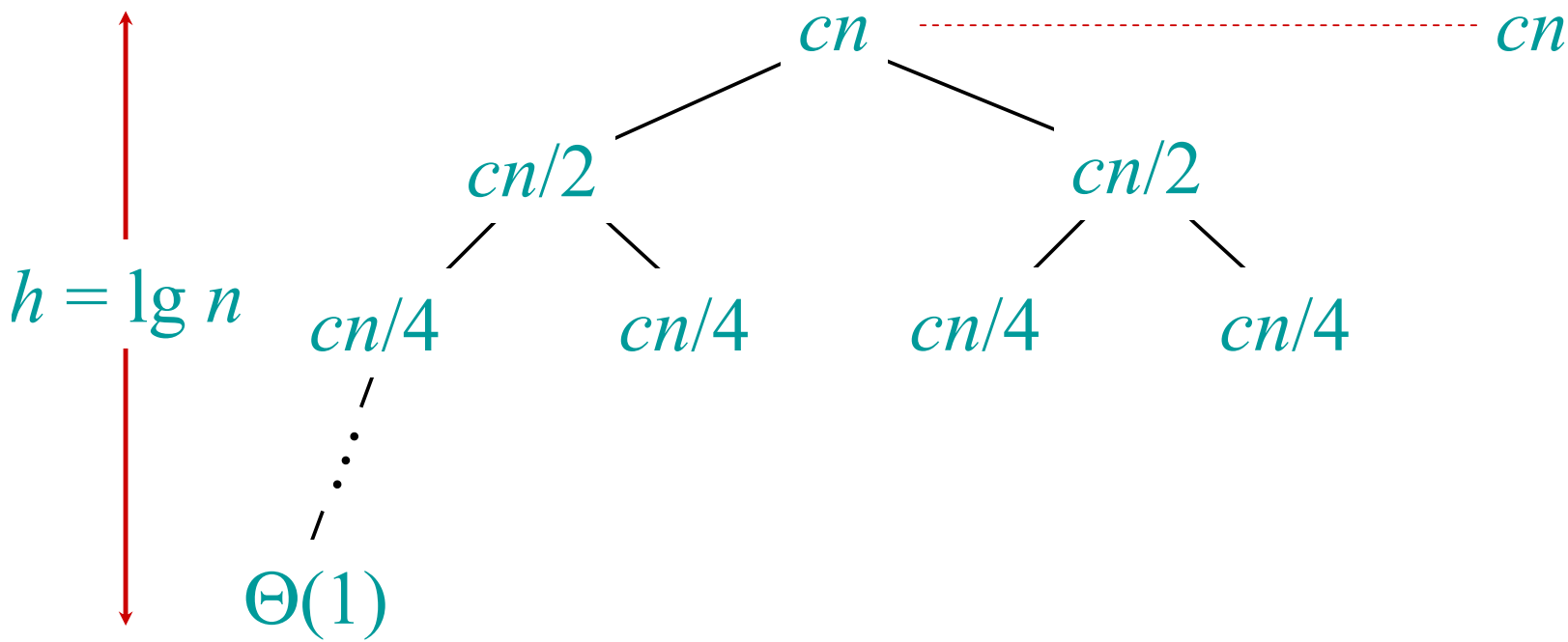
Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.



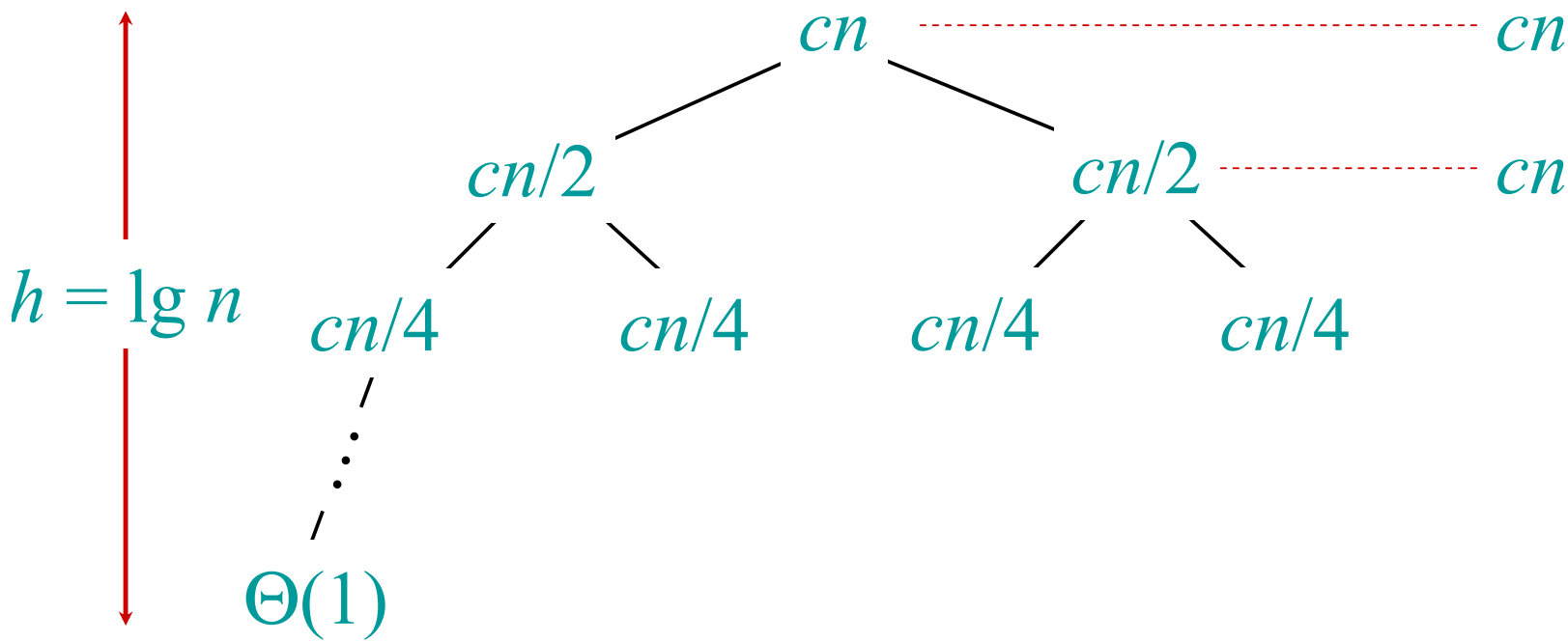
Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.



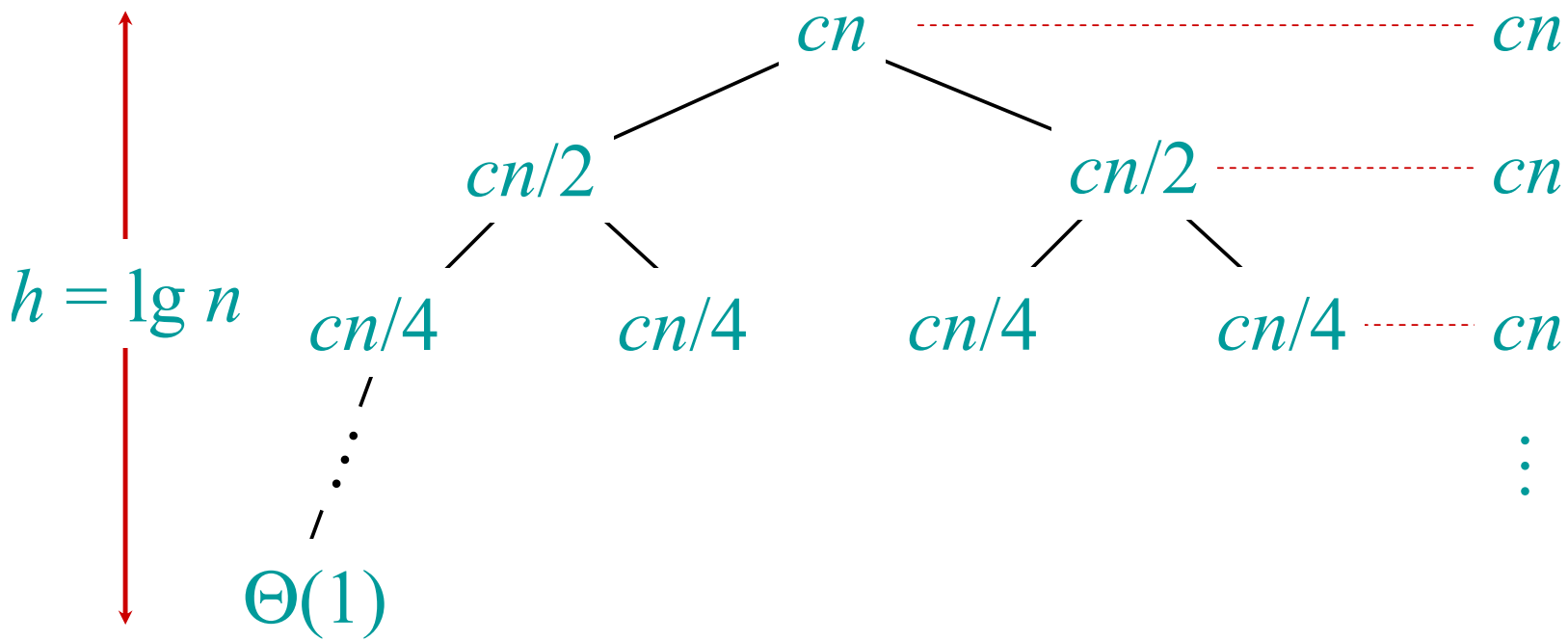
Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.



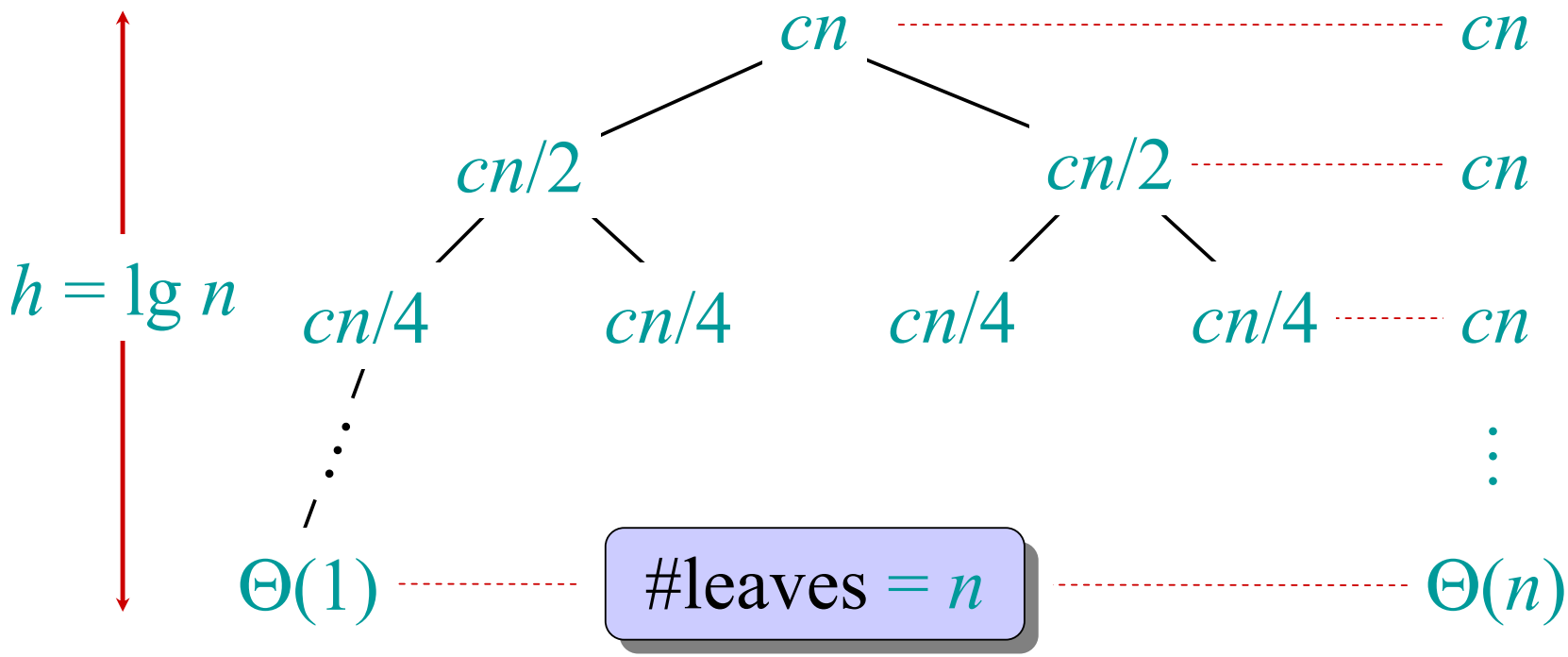
Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.



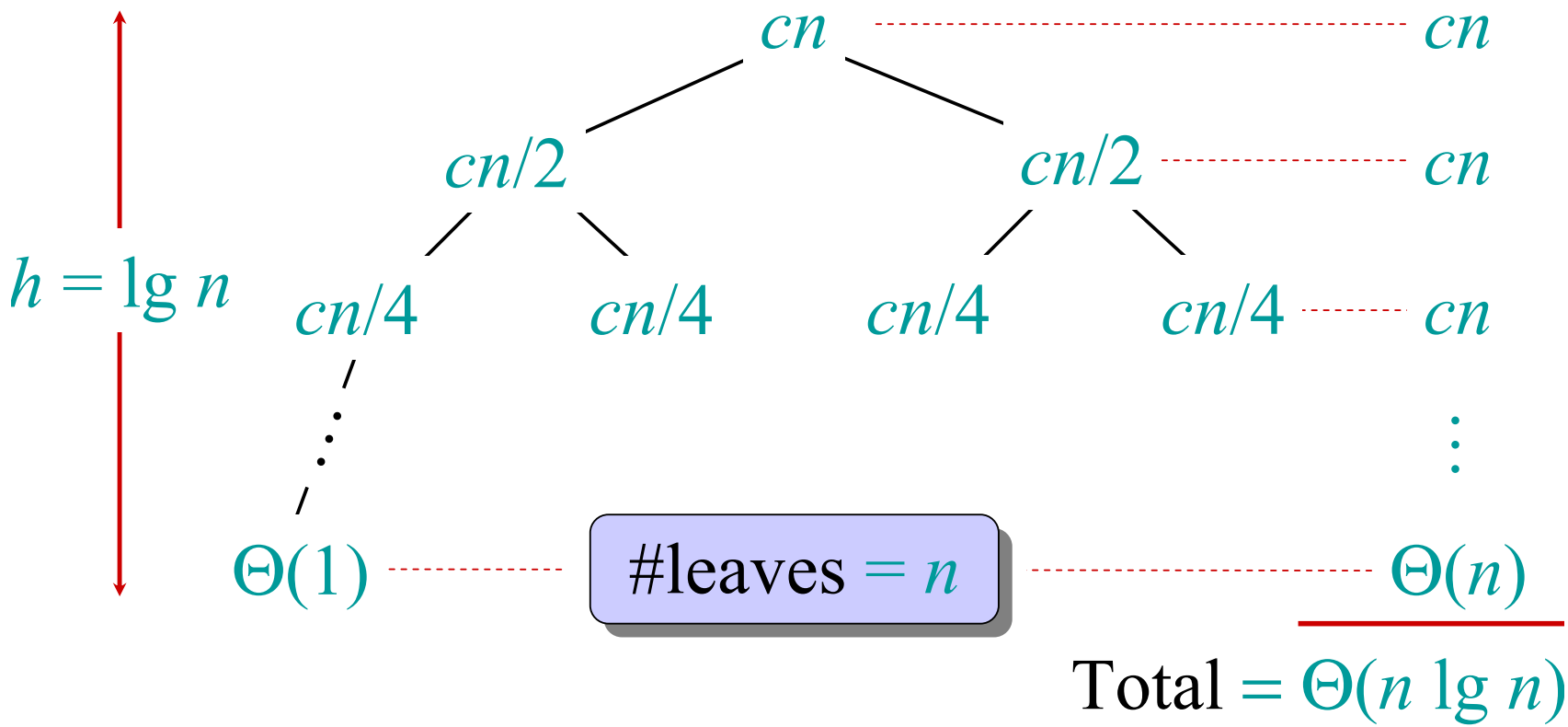
Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.



Recursion Tree

Solve $T(n) = 2T(\frac{n}{2}) + cn$, where $c > 0$ is constant.



Conclusions

- $\Theta(nlogn)$ grows more slowly that $\Theta(n^2)$.
- 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!