CS 421: Design and Analysis of Algorithms

Chapter 2: Introduction to Design & Analysis of Algorithms

Dr. Gaby Dagher
Department of Computer Science
Boise State University

January 11, 2022

BOISE STATE UNIVERSITY

Content of this Chapter

- **Algorithm: Insertion Sort**
- Analyzing Algorithms
- Designing Algorithms

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

Sorting Example: Insertion Sort

8 2 4 9 3 6

Sorting Example: Insertion Sort

8 2 4 9 3 6

Sorting Example: Insertion Sort

8 2 4 9 3 6

2 8 4 9 3 6

Sorting Example: Insertion Sort

8 2 4 9 3 6

2 8 4 9 3 6

Sorting Example: Insertion Sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Sorting Example: Insertion Sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Sorting Example: 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

Sorting Example: 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

Sorting Example: 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

Sorting Example: 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

Sorting Example: 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*

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
```

Insertion Sort

"pseudocode"

INSERTION-SORT (A, n) $\triangleleft A[1 \mathinner{.\,.} n]$

    for $j \leftarrow 2$ **to** n

        do $key \leftarrow A[j]$

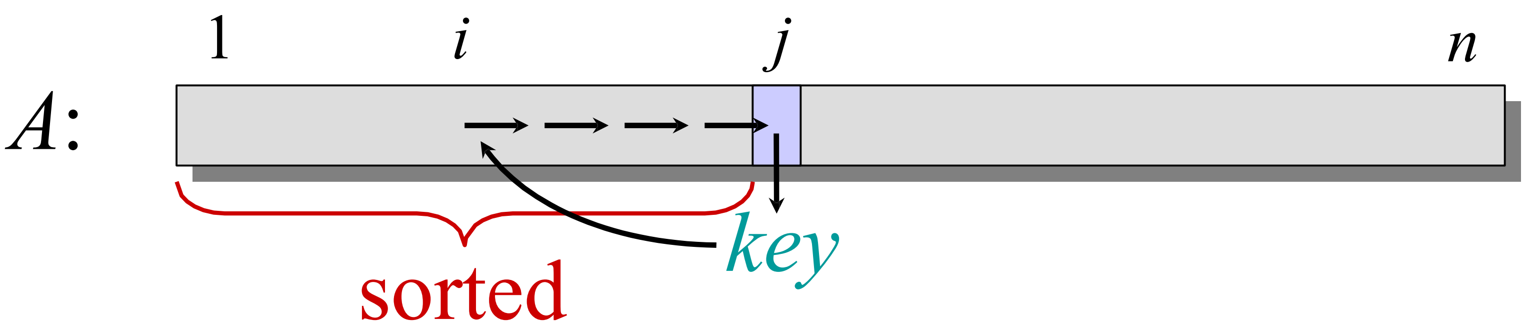
            $i \leftarrow j - 1$

            while $i > 0$ and $A[i] > key$

                do $A[i+1] \leftarrow A[i]$

                    $i \leftarrow i - 1$

            $A[i+1] = key$



Content of this Chapter

- Algorithm: Insertion Sort
- **Analyzing Algorithms**
- Designing Algorithms

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.

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 $n \to \infty$.

"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 $n \geq n_0 \}$

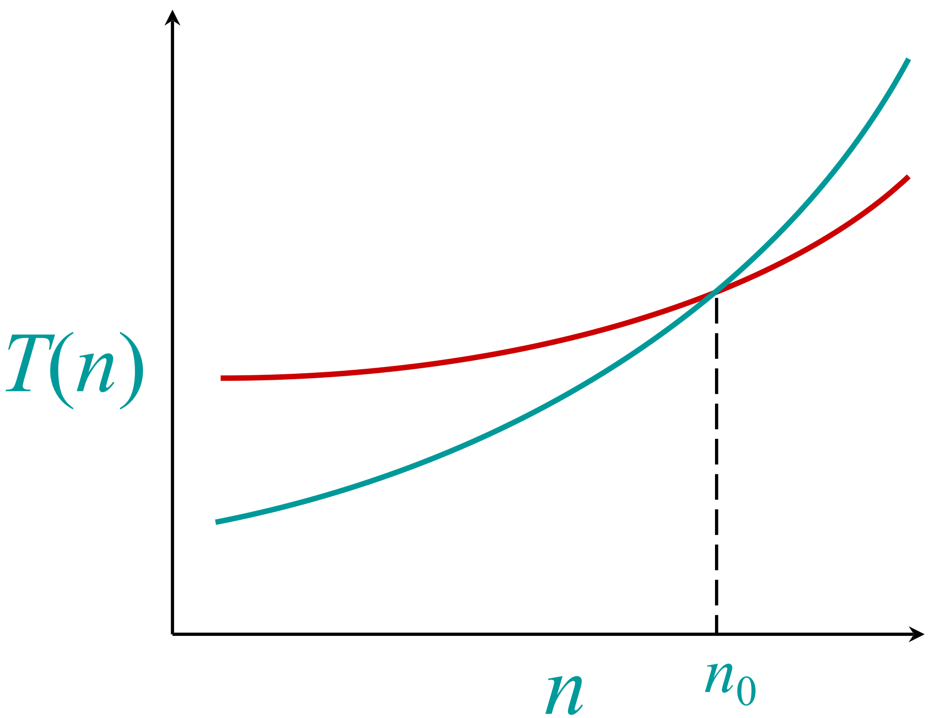
Engineering:

- Drop low-order terms
- Ignore leading constants.

- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

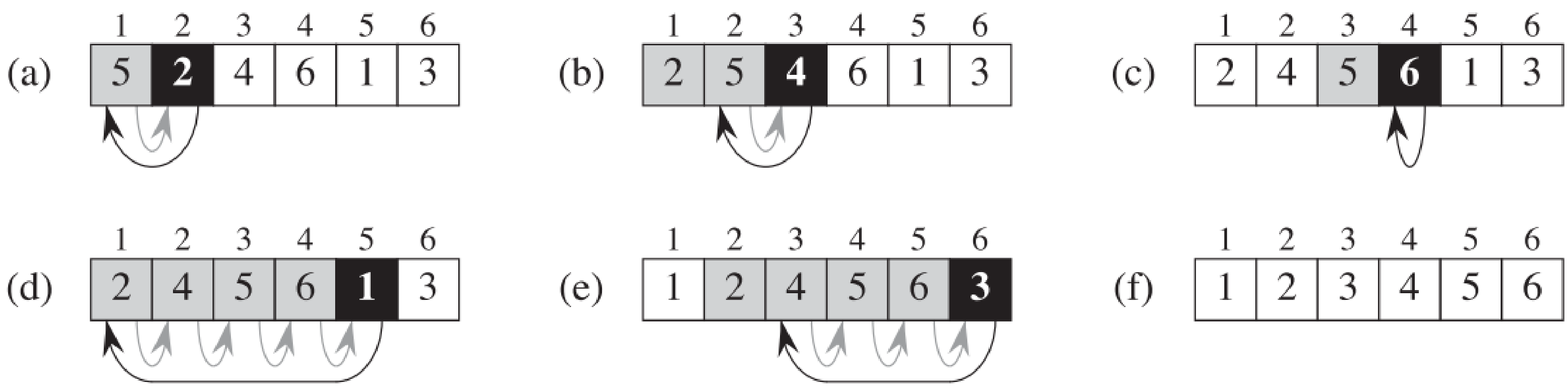
Asymptotic performance

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: another example



Insertion sort analysis

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$\quad i = j - 1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\quad\quad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\quad\quad i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Insertion sort analysis

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best case: Input already sorted.

→ $t_j = 1$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

→ $T(n) = \Theta(n)$

Insertion sort analysis

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Worst case: Input reverse sorted.

$\rightarrow t_j = j$

$$\boxed{\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1}$$

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

$\rightarrow T(n) = \Theta\left(n^2\right)$

Insertion sort analysis

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Average case: All permutations equally likely.

→ $t_j = j/2$

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

Loop Invariant

- Loop invariant is a statement that helps show an algorithm is correct.

- Three things must be shown about a loop invariant:
  - **Initialization**: It is true prior to the *first* iteration of the loop.
  - **Maintenance**: If it is true before *an* iteration of the loop, it remains true after *the* iteration and before the next iteration.
  - **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Correctness of Insertion Sort

- **Loop Invariant:**

  At the start of each iteration of the **for** loop of lines 1–8, the subarray A[1, …, j-1] consists of the elements originally in A[1, …, j-1] but in sorted order.

- **Proof:**

- **Initialization.** Since the first iteration of the loop starts at j=2, we are concerned with subarray A[1, j-1] = A[1,1]:
  - Subarray A[1, 1] consists of just the single element A[1], which is the original element in A[1, 1].
  - Subarray A[1,1] is already sorted (A[1] is the only element in the subarray).

Correctness of Insertion Sort (continue)

- **Maintenance.** When the next iteration of the **for** loop is $j = k$, where $2 <= k <= n$, then the **while** loop (lines 4–7) keeps moving A[k] one position to the left until it finds the proper position for it, at which point it inserts the value of A[k] (line 8). <u>At the start of iteration k+1 of the **for** loop</u>:
  - The subarray A[1, k] consists of the elements originally in A[1, k], and
  - The elements in the subarray A[1, k] are already sorted.
- **Termination.** The **for** loop terminates when $j = n+1$. After iteration n completes and before we start evaluating the condition for iteration n+1:
  - The subarray A[1, n] consists of the elements originally in A[1, n], and
  - The elements in the subarray A[1, n] are already sorted.

  Since the subarray A[1, n] is the entire array, we conclude that the entire array is sorted.

Content of this Chapter

- Algorithm: Insertion Sort
- Analyzing Algorithms
- ➢ **Designing Algorithms**

Insertion sort efficiency

Is insertion sort a fast sorting algorithm?

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

In *insertion sort*, we used ***incremental*** algorithm design technique.

Divide-and-conquer Technique

Why divide-and-conquer?

- Many useful algorithms are ***recursive*** in structure.
- Recursive algorithms typically follow a divide-and-conquer approach: they (1) break the problem into several *subproblems* that are *similar* to the original problem but *smaller* in size, (2) solve the subproblems *recursively*, and (3) combine these solutions to create a solution to the original problem.

Divide-and-conquer Technique

The divide-and-conquer approach involves *three steps* at each level of the recursion:

- **Divide** the problem into a number of subproblems.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems into the solution for the original problem.

Analyzing divide-and-conquer algorithms

- We can often describe the runtime of a recursive function by a *recurrence equation*.
- A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm: *divide*, *conquer*, and *combine*.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \,. \end{cases}$$

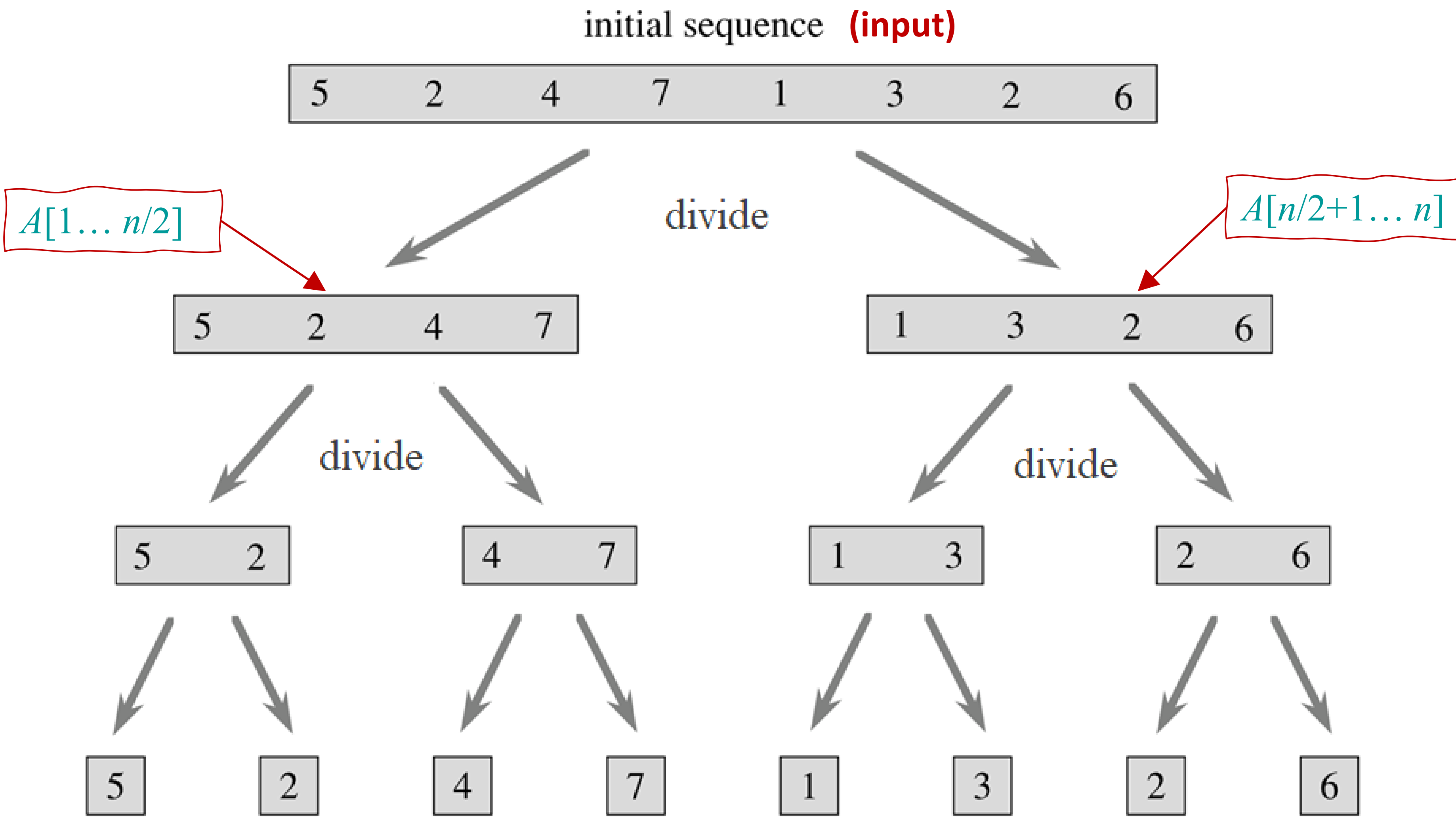
Base case

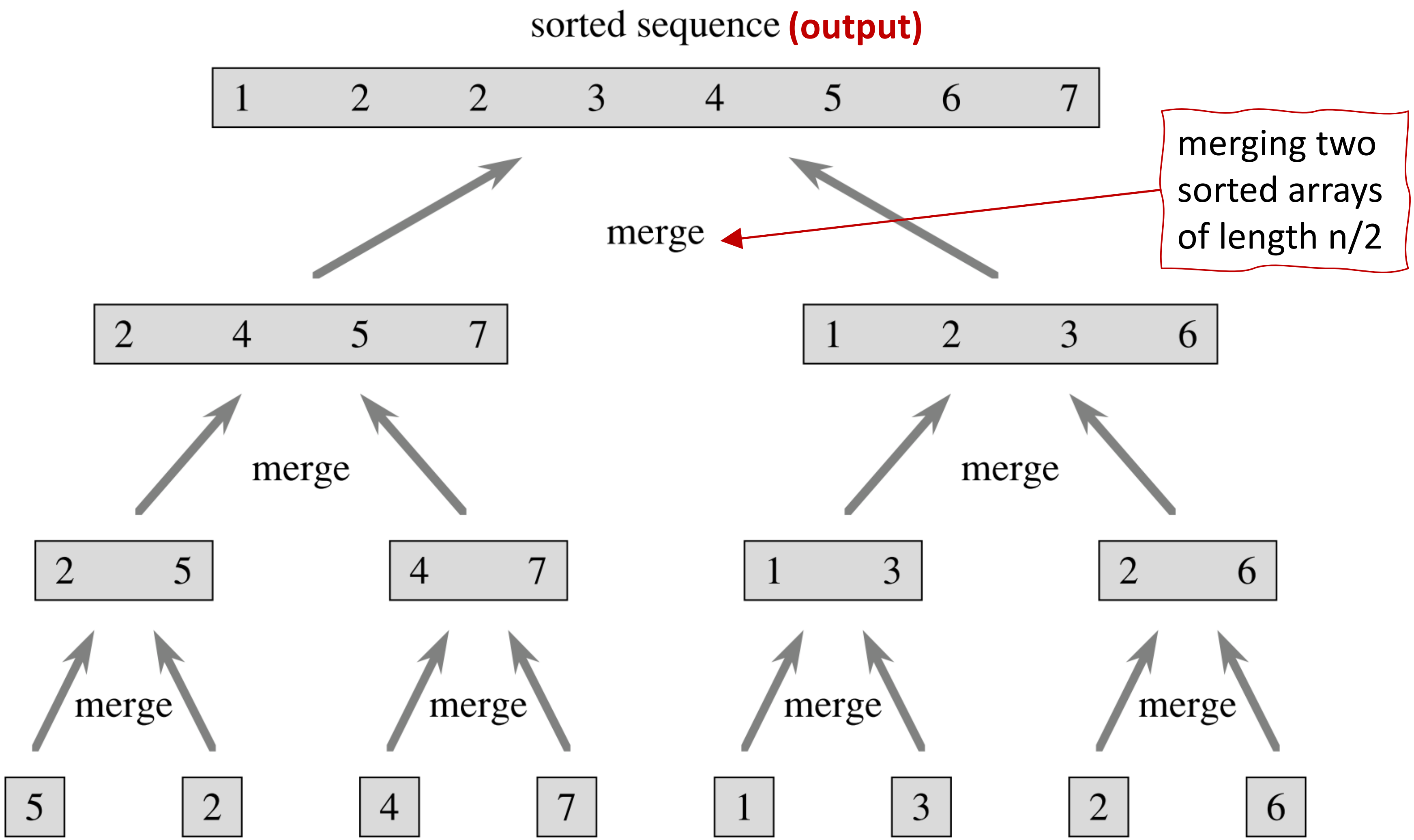
Recursive case

Divide-and-conquer Algorithm: Merge sort

MERGE-SORT $A[1 \,.\,.\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[1\ldots n/2]$ and $A[n/2+1\ldots n]$.
3. "***Merge***" the 2 sorted lists.


initial sequence (input)
5 2 4 7 1 3 2 6
divide
A[1… n/2]
A[n/2+1… n]
5 2 4 7
1 3 2 6
divide
divide
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6


sorted sequence (output)
1 2 2 3 4 5 6 7
merge
merging two sorted arrays of length n/2
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6

Merge sort

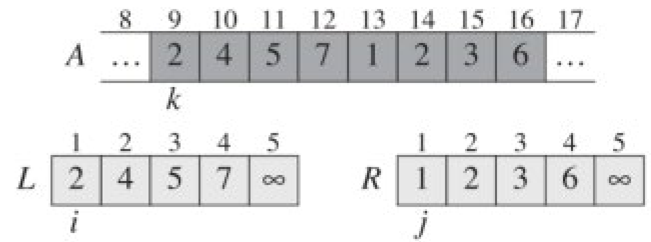
MERGE-SORT $A[1\ .\ .\ n]$

1. If $n = 1$, done.
2. Recursively sort $A[1 \ldots n/2]$ and $A[n/2+1 \ldots n]$.
3. "***Merge***" the 2 sorted lists.

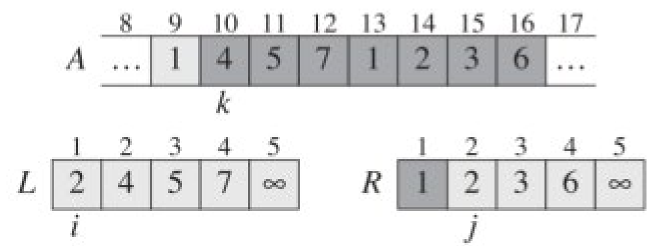
Key function: Merge

Merge function

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```



(a)



(b)

Merge Function: It merges <u>two sorted arrays</u> of size n/2

20	12
13	11
7	9
2	1

Merge Function: **It merges <u>two sorted arrays</u> of size n/2**

20	12
13	11
7	9
2	1

1

Merge Function: It merges **two sorted arrays** of size n/2

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

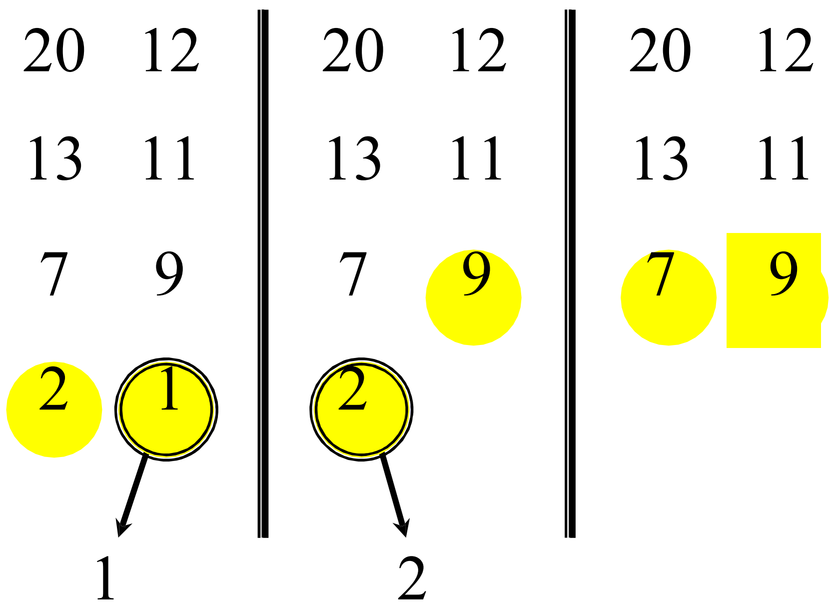
1

Merge Function: It merges **two sorted arrays** of size n/2

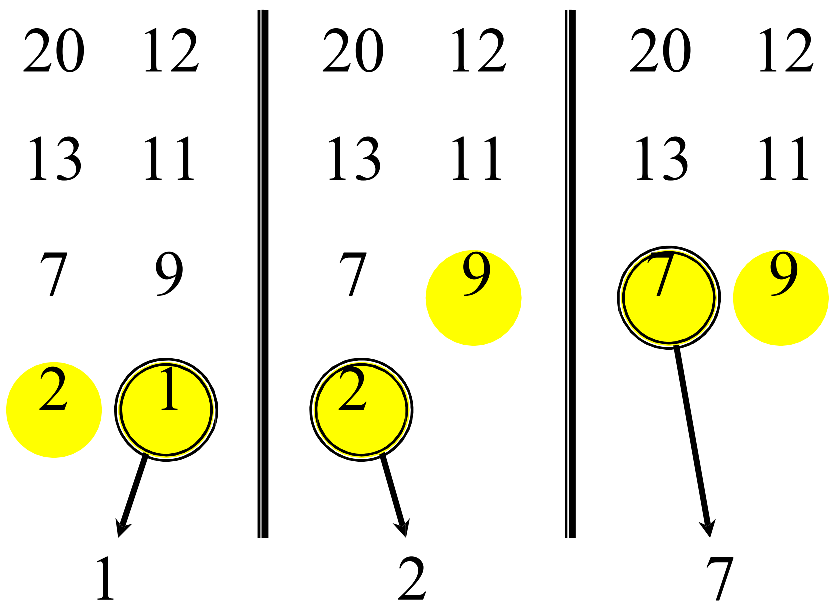
20	12		20	12
13	11		13	11
7	9		7	9
2	1		2	

1 2

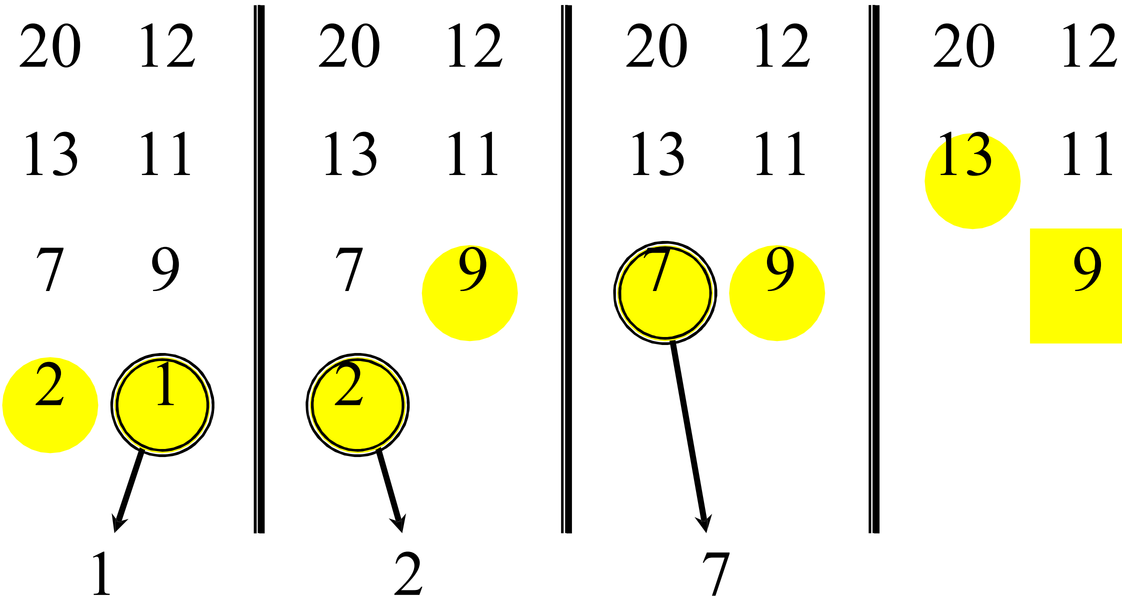
Merge Function: It merges <u>two sorted arrays</u> of size n/2



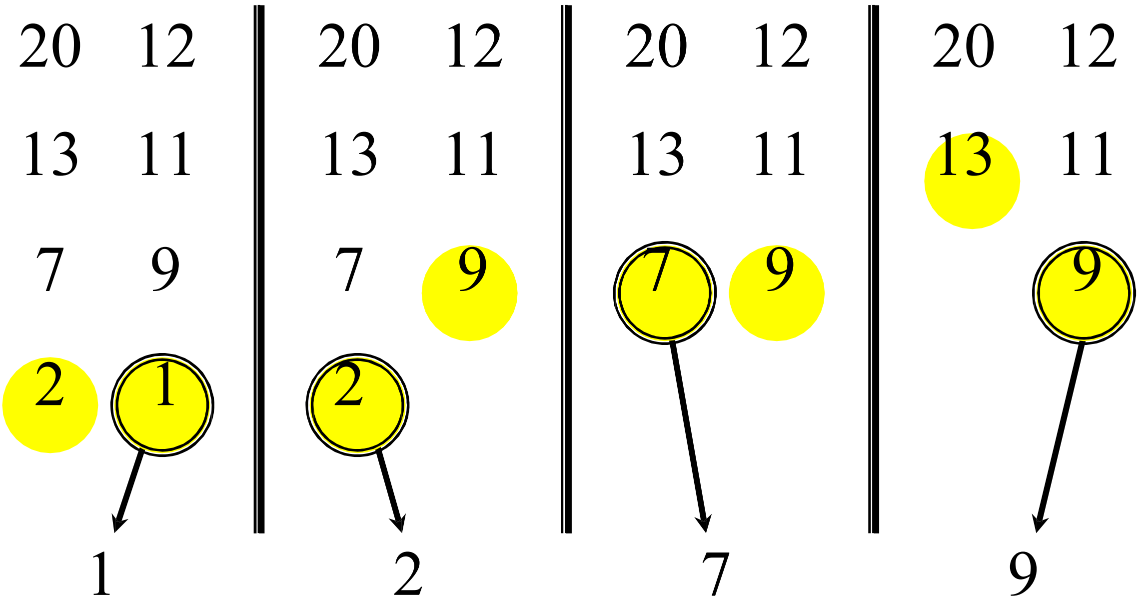
Merge Function: It merges **two sorted arrays** of size n/2



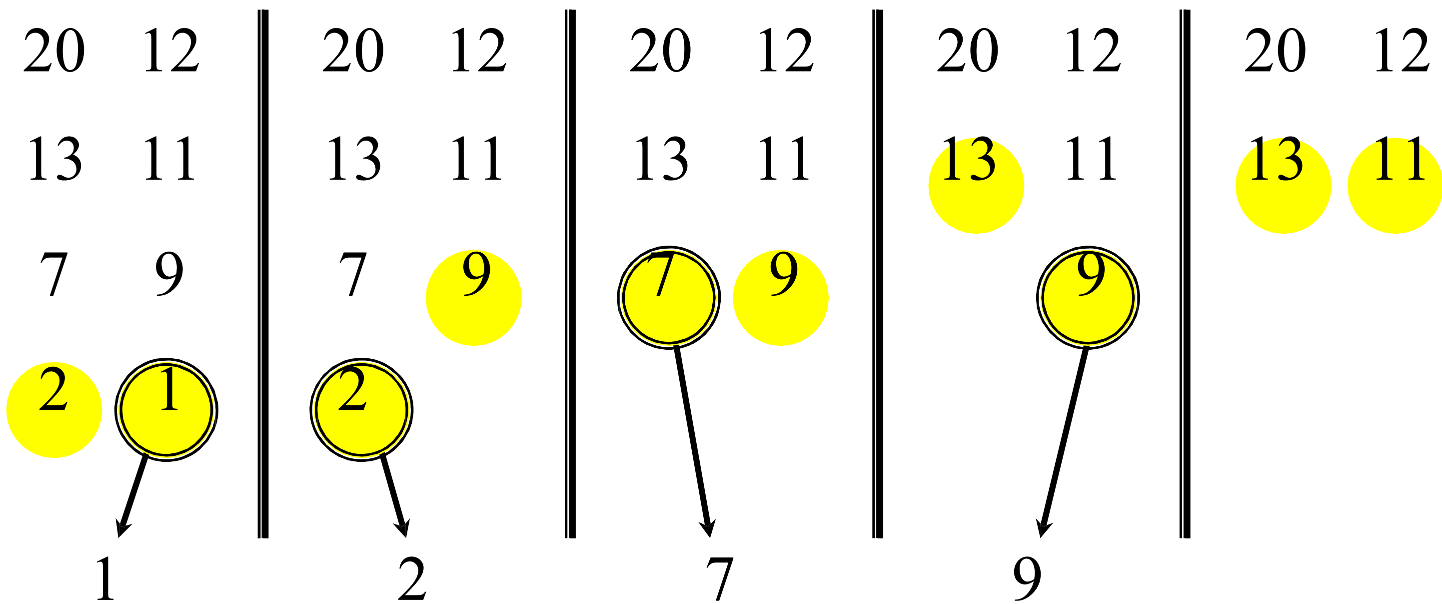
Merge Function: It merges two sorted arrays of size n/2



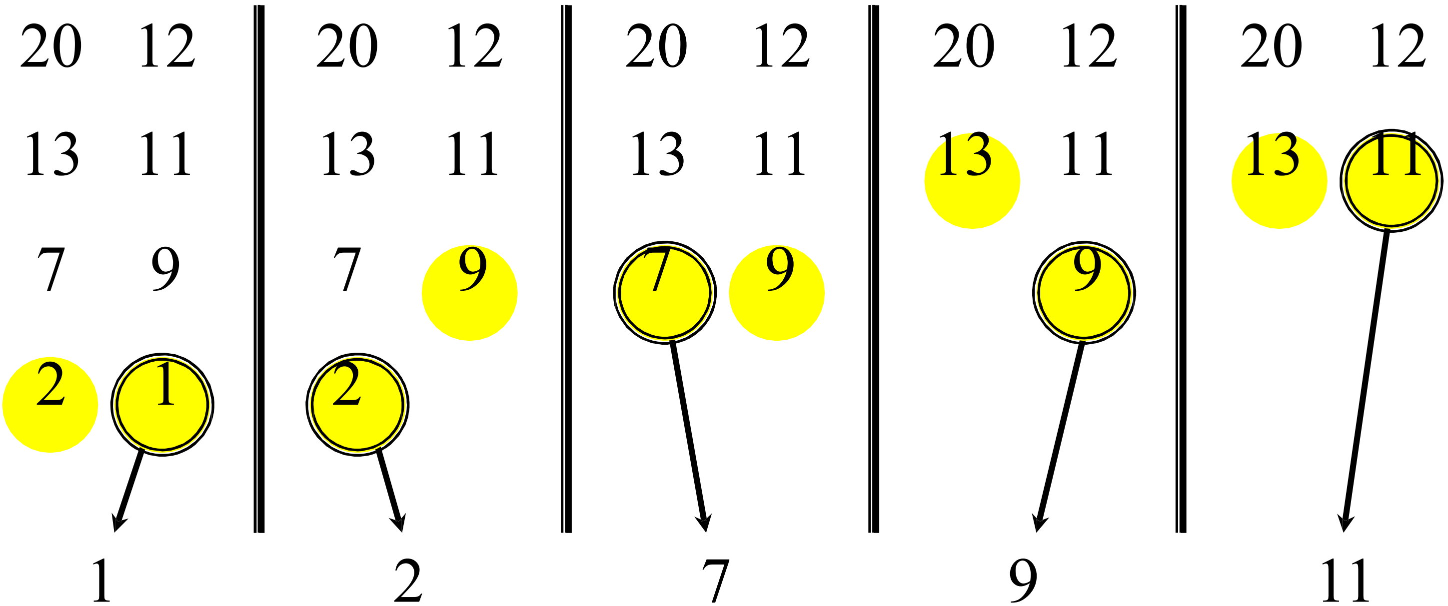
Merge Function: It merges **two sorted arrays** of size n/2



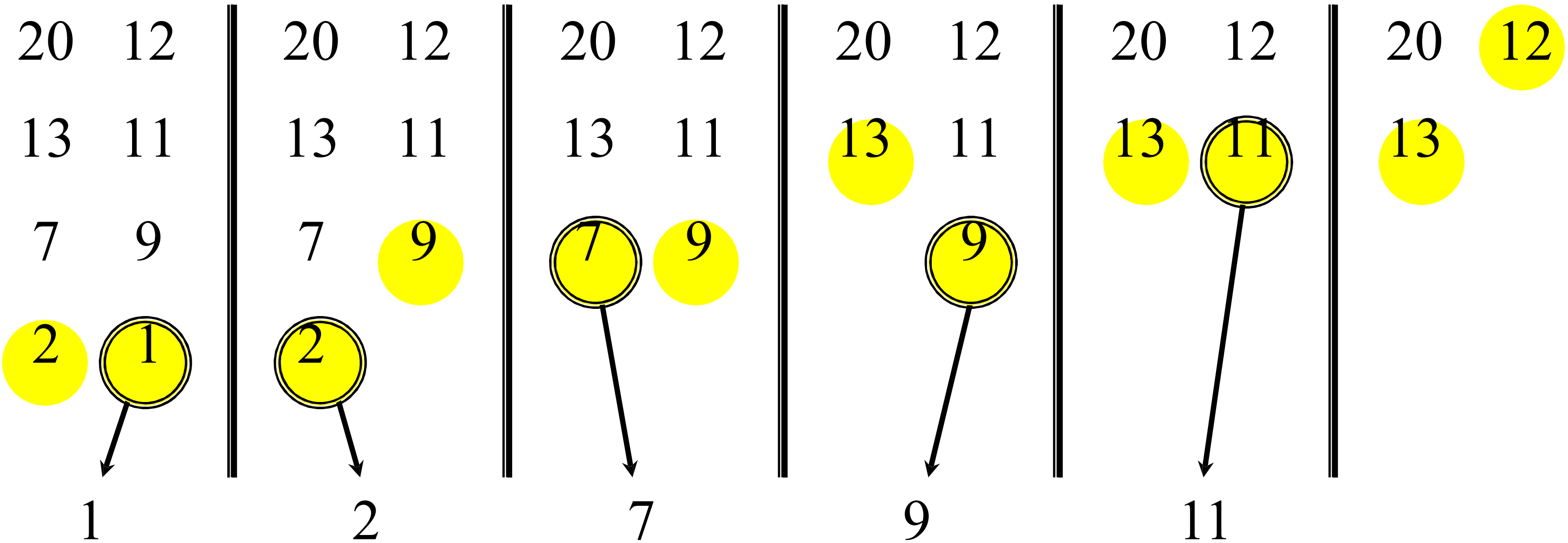
Merge Function: It merges **two sorted arrays** of size n/2



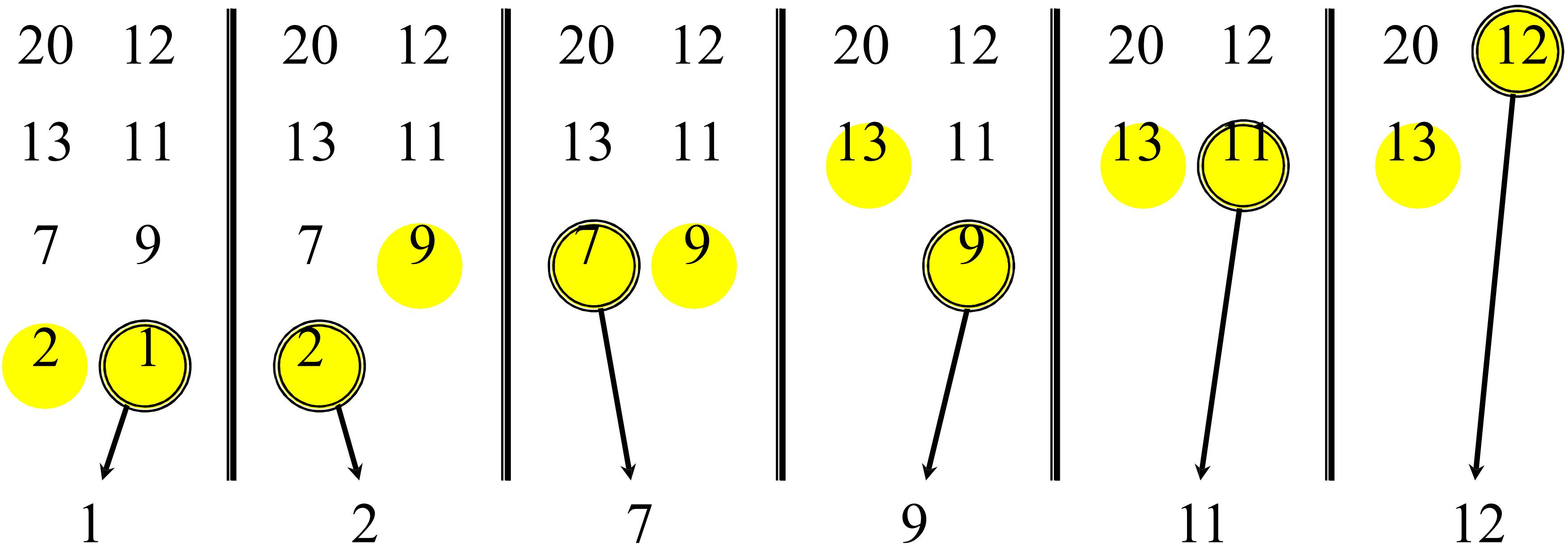
Merge Function: It merges <u>two sorted arrays</u> of size n/2



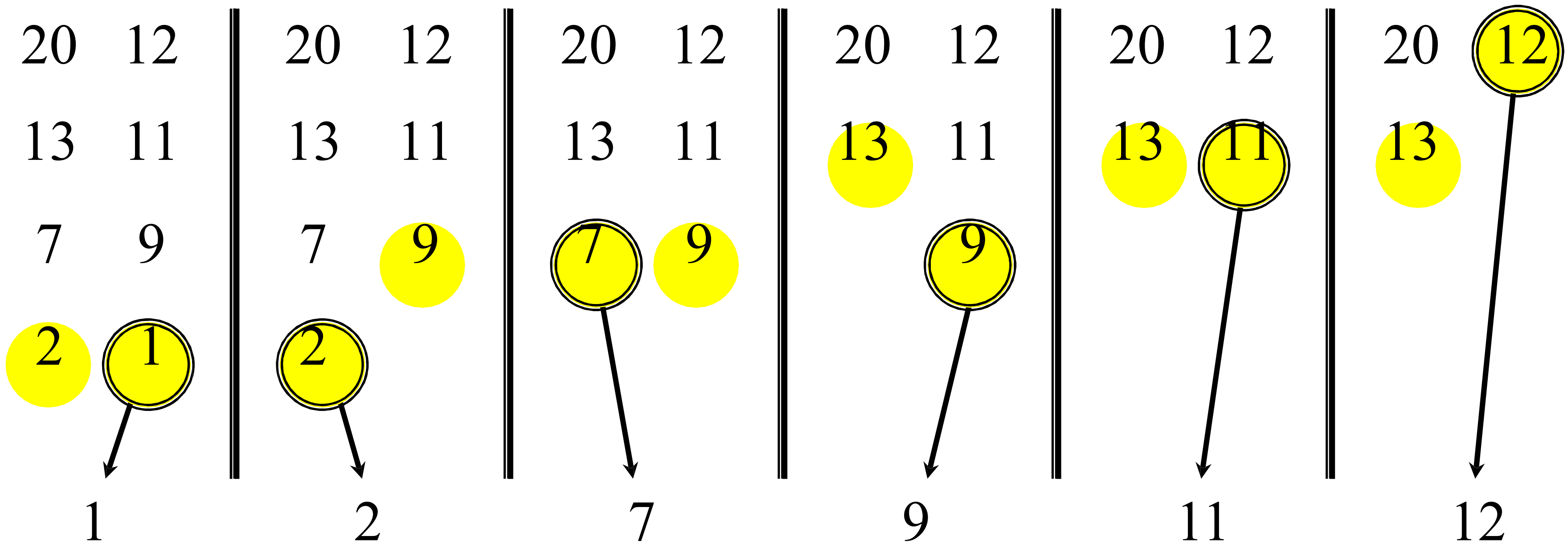
Merge Function: It merges two sorted arrays of size n/2



Merge Function: It merges <u>two sorted</u> <u>arrays</u> of size n/2



Merge Function: It merges <u>two sorted arrays</u> of size n/2



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

Analyzing Merge sort

$T(n)$	MERGE-SORT $A[1\,.\,.\,n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1\,.\,.\,n/2]$ and $A[\,n/2+1\,.\,.\,n\,]$.
$\Theta(n)$	***3.*** ***"Merge"*** the 2 sorted lists

Sloppiness: Should be $T(\,n/2\,) + T(\,n/2\,)$, 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.
- In coming lectures, we will explore several ways for finding a good upper bound (worst-case) on $T(n)$.

Recursion tree for Merge sort

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

Recursion tree for Merge sort

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

$$T(n)$$

Recursion tree for **Merge sort**

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

$$
\begin{array}{ccc}
 & cn & \\
\swarrow & & \searrow \\
T(n/2) & & T(n/2)
\end{array}
$$

Recursion tree for Merge sort

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

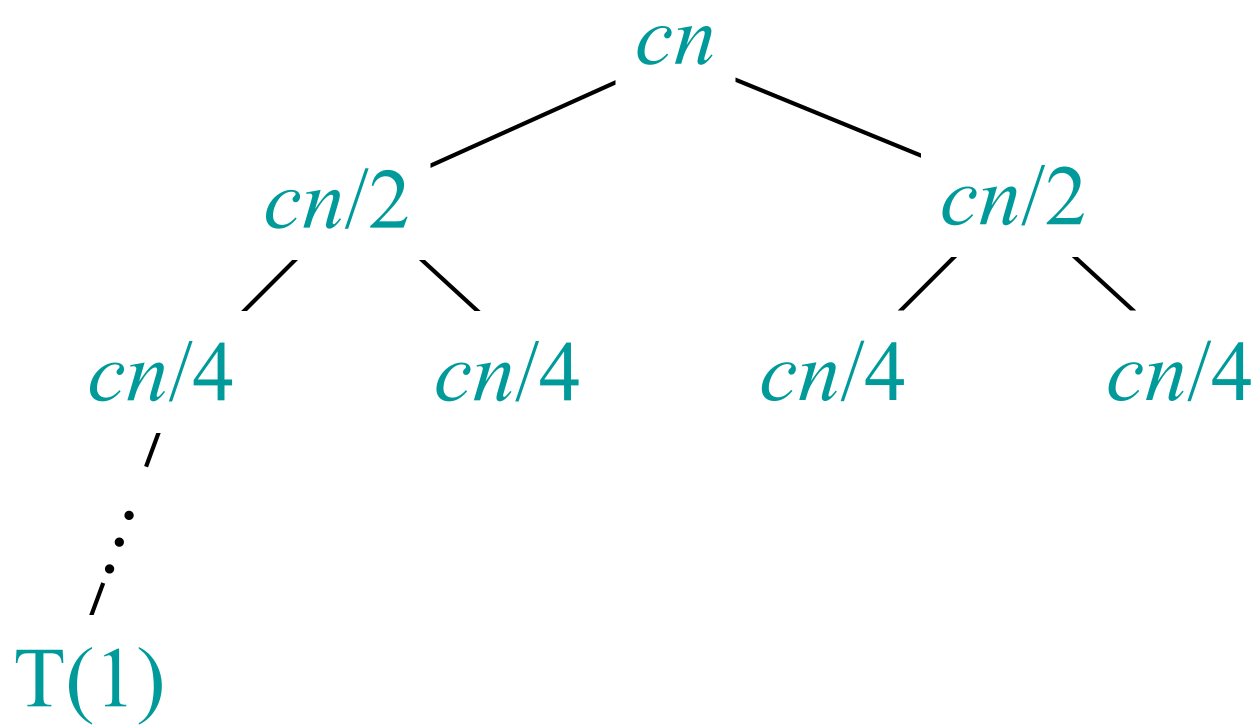
$$cn$$

$$cn/2 \qquad\qquad cn/2$$

$$T(n/4) \qquad T(n/4) \qquad T(n/4) \qquad T(n/4)$$

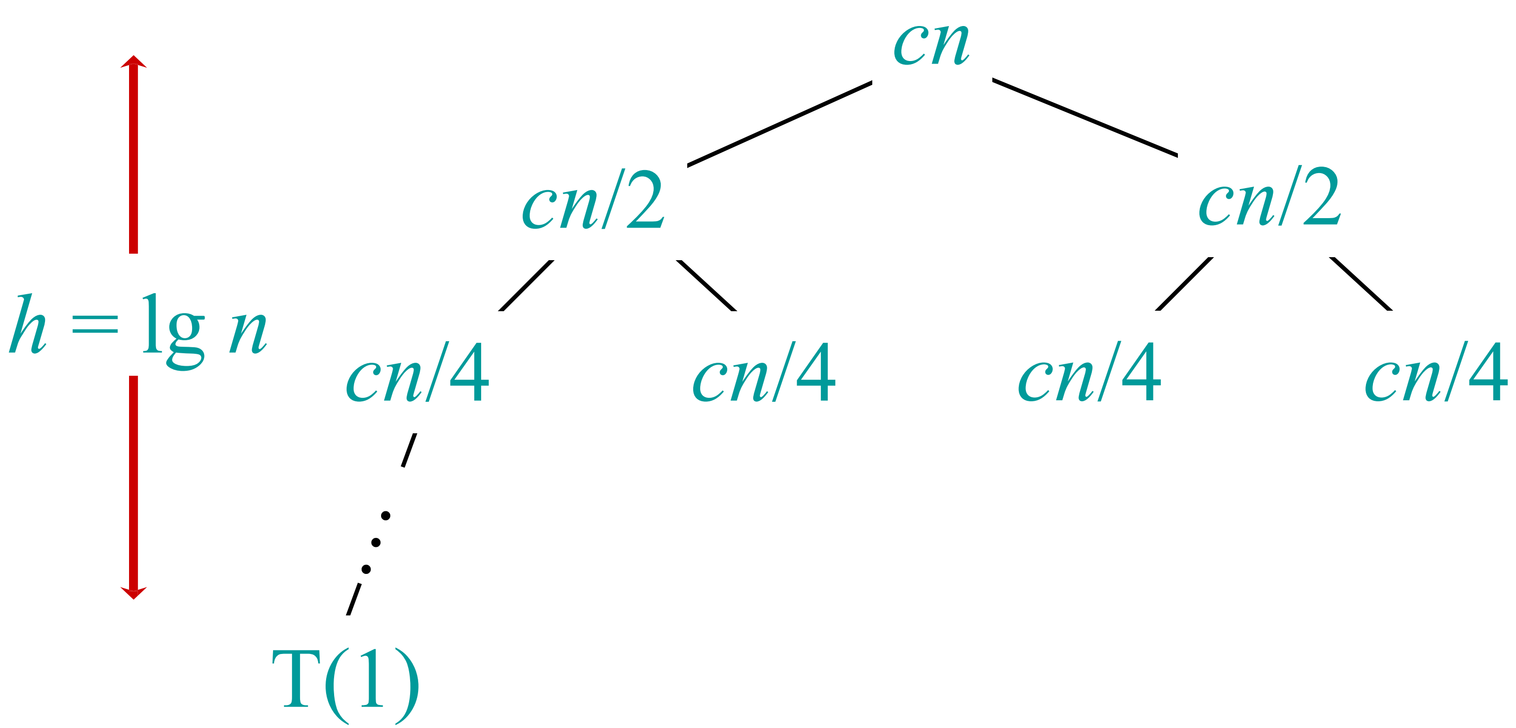
Recursion tree for Merge sort

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



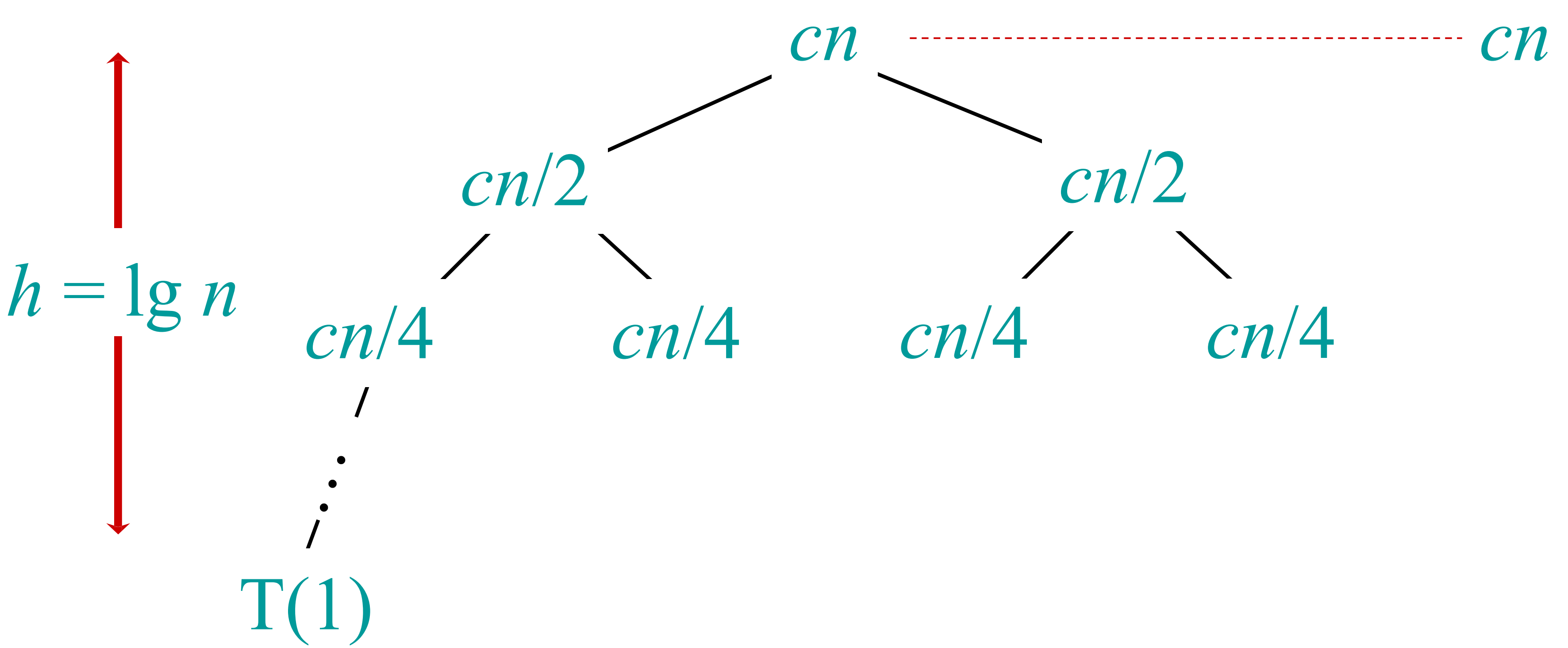
Recursion tree for Merge sort

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



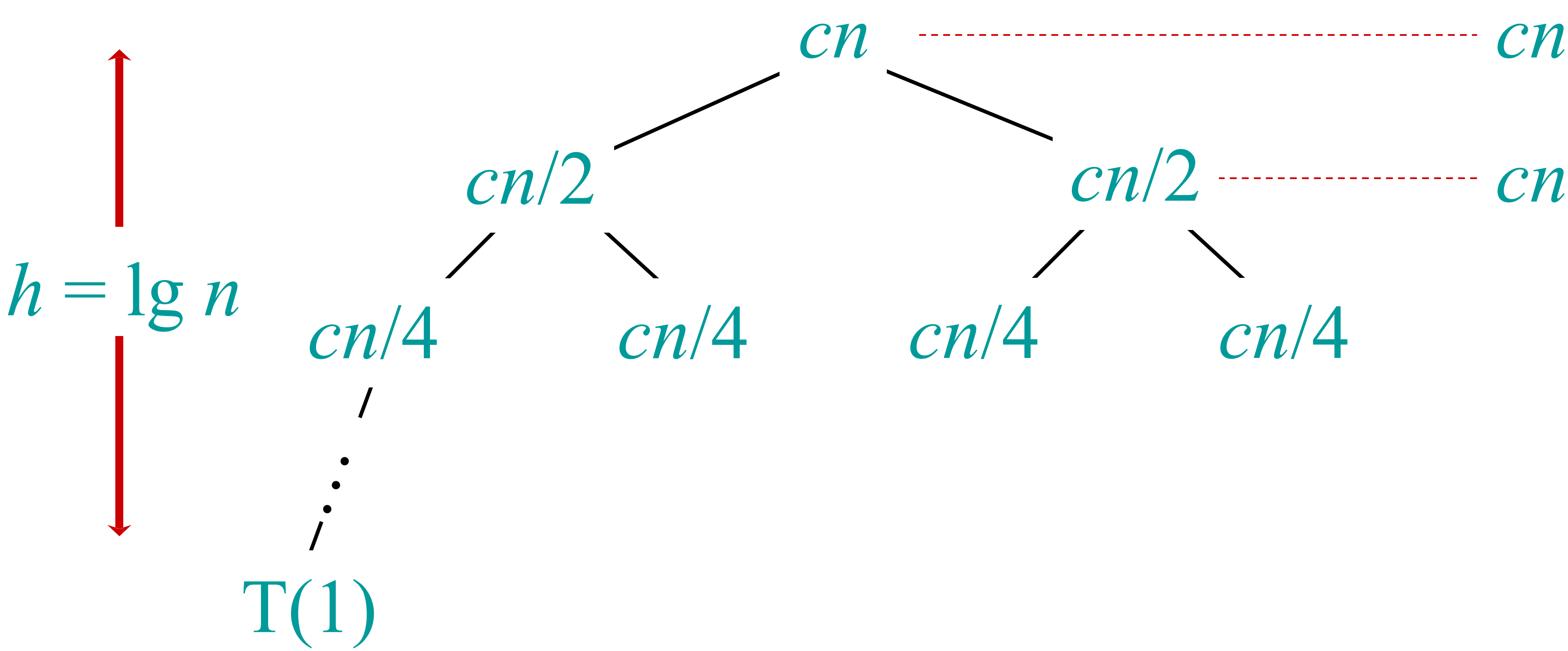
Recursion tree for Merge sort

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



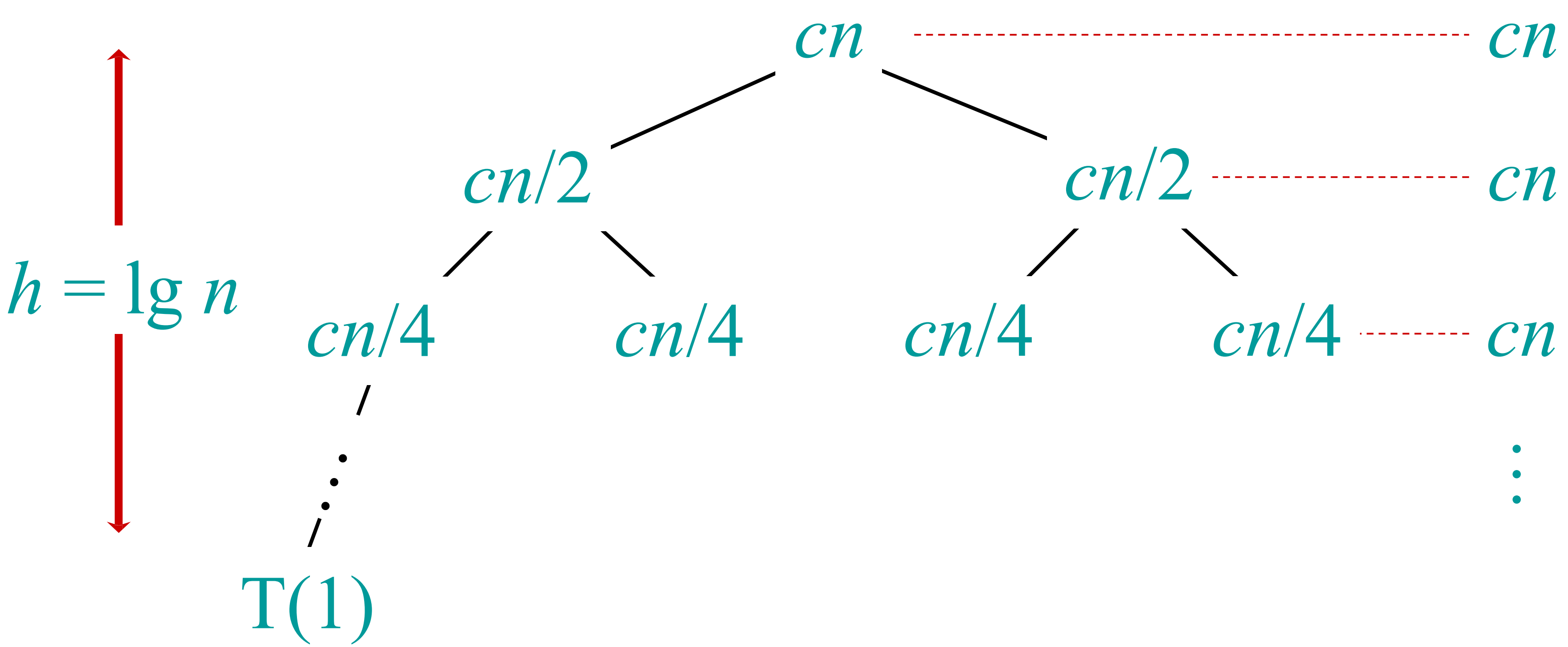
Recursion tree for Merge sort

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



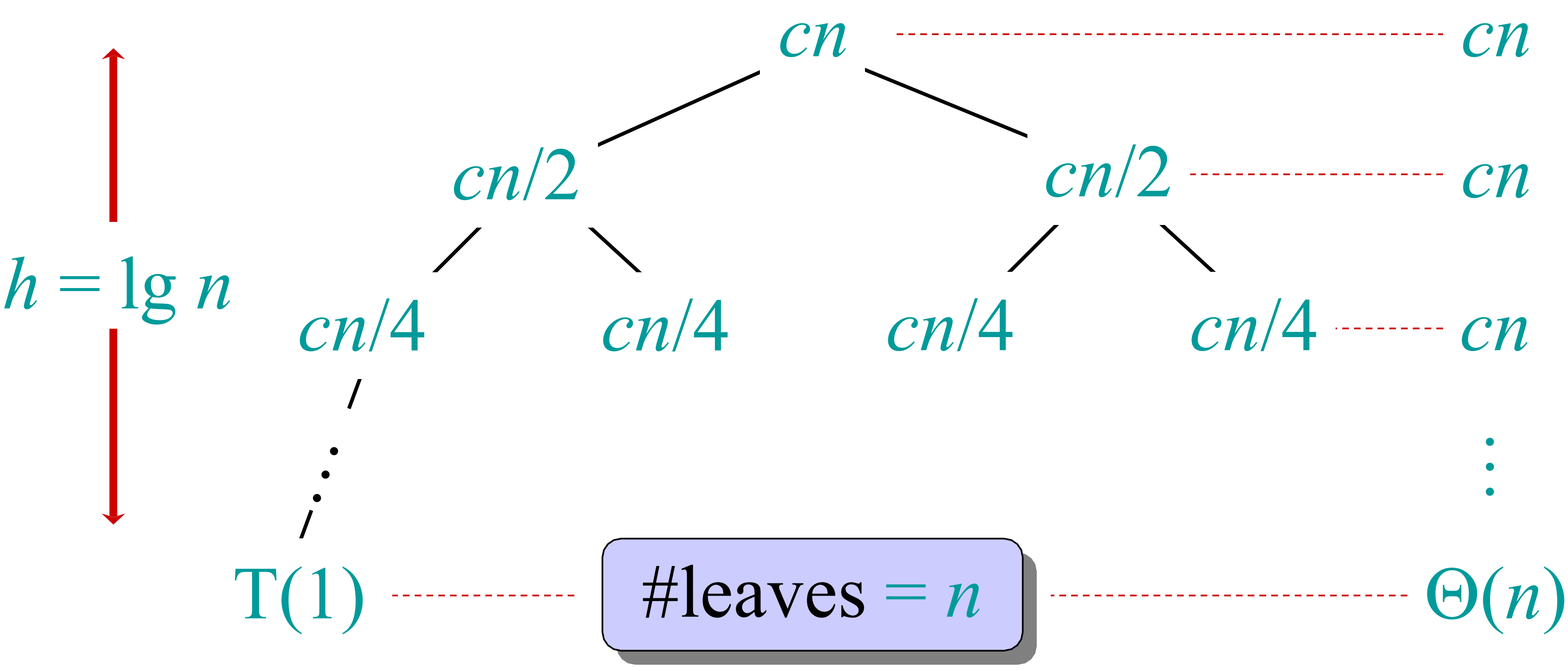
Recursion tree for Merge sort

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



Recursion tree for Merge sort

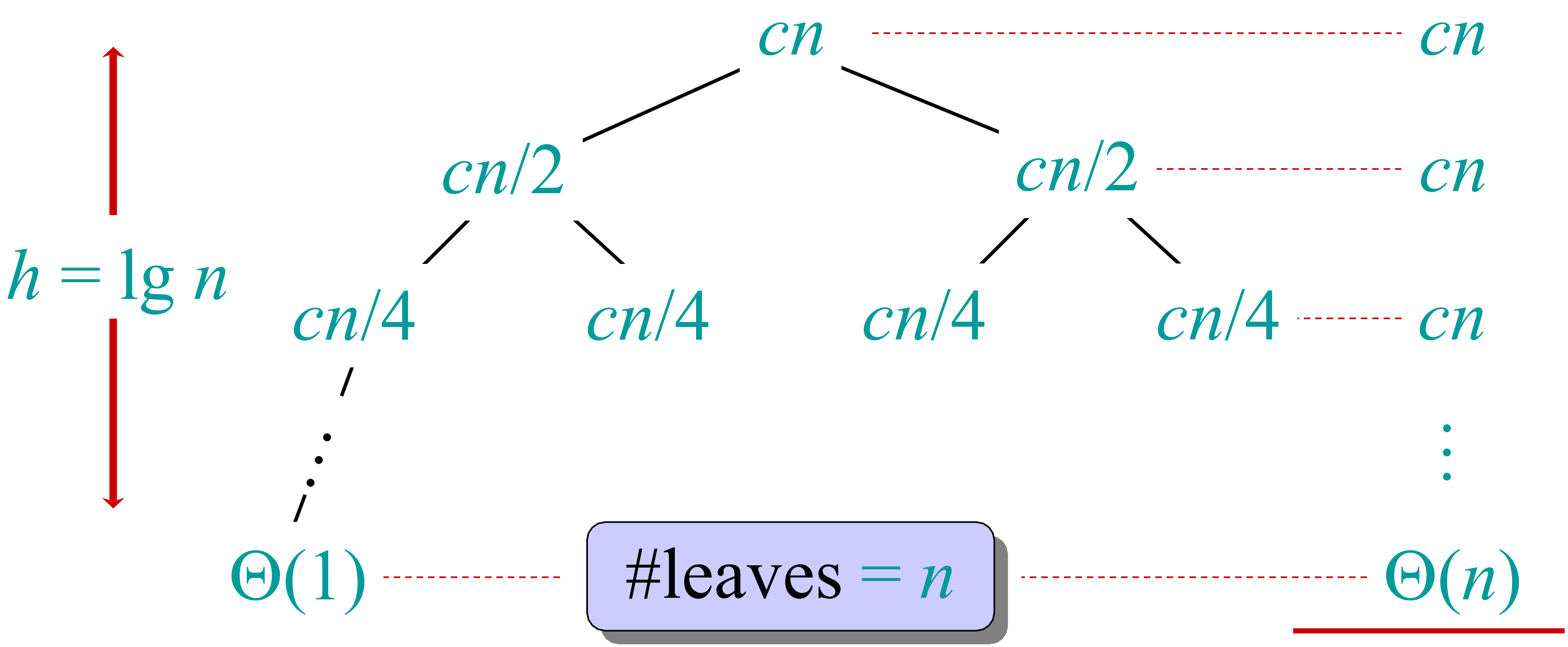
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recall that: $T(1) = \Theta(1)$ if $n = 1$

Recursion tree for Merge sort

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



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

Conclusions

- $\Theta(n \lg n)$ (merge sort) grows more slowly than $\Theta(n^2)$ (insertion sort).
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.