CENG 218
Design and Analysis of Algorithms

Izmir Institute of Technology

Lecture 3:
Divide-and-conquer approach

Slides were mostly prepared using the material provided by Prof. Charles E. Leiserson and Prof. Erik Demaine from MIT

Divide-and-conquer approach

- Many useful algorithms are recursive in structure.
- To solve a problem they call themselves recursively to deal with smaller size problems.
- They follow a divide-and-conquer approach:

Divide the problem into several subproblems.

Conquer the subproblems by solving them recursively.

Combine the solutions of subproblems to obtain the solution of the original problem.

Example: Merge sort

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 two sorted lists.

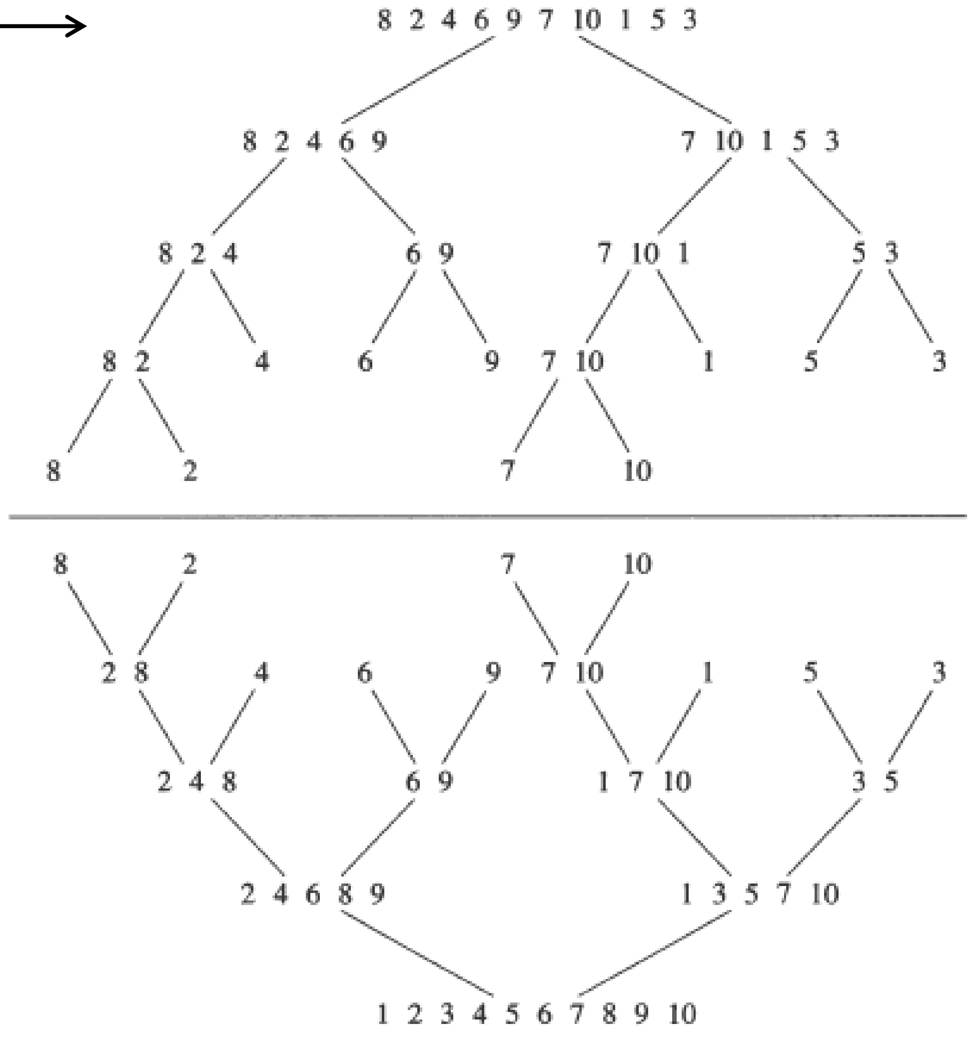
Key subroutine: **MERGE**

A visualization for merge sort

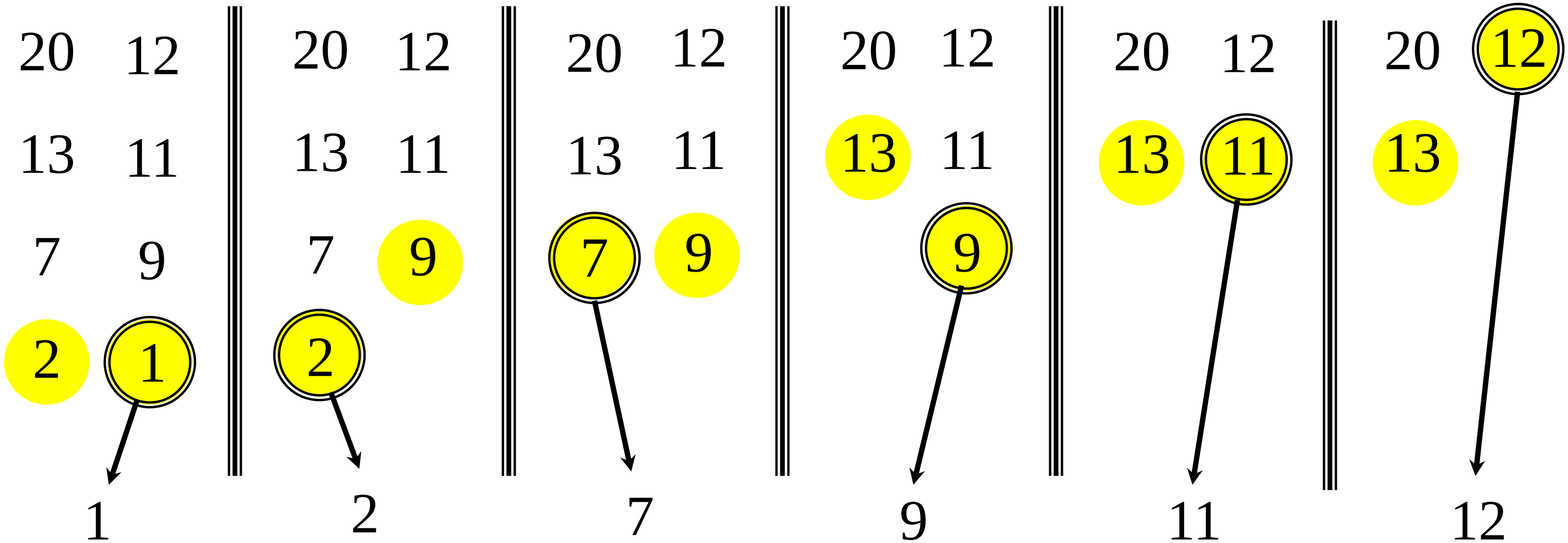
Merge sort of
[8 2 4 6 9 7 10 1 5 3].

The list is divided into two sub-lists recursively.

Sub-lists are merged by comparing their smallest elements.



Merging two sorted arrays



$\mathrm{T}(n) = \Theta(n)$ ..time to merge a total of *n* elements

Procedure MERGE-SORT

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

if $n > 1$ **then**

begin

$m := \lfloor n/2 \rfloor$ {this is roughly halfway}

$A :=$ MERGE (MERGE-SORT $(A[1\,..\,m])$,

MERGE-SORT $(A[m+1\,..\,n])$)

end

Procedure MERGE

MERGE (A, B: sorted lists)
L := empty list
k := 1
while A and B are non-empty
begin
m := smaller of the first elements of A and B
remove m from the list it is in {A or B}
$L[k] := m$
$k := k+1$
end {*L is merged elements in increasing order*}

Takes $\Theta(|A|+|B|) = \Theta(n)$ time.

Analyzing merge sort

$T(n) =$
constant time $\rightarrow \Theta(1) +$
$2T(n/2) +$
linear time $\rightarrow \Theta(n)$

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

Actually this is $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$,
but it does not matter asymptotically.

Analyzing recursive algorithms

Recursive algorithms can be described by recurrences. I.e. The running time of problem with size n is written in terms of smaller inputs.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \text{ is minimal size} \\ aT(n/b) + \mathrm{D}(n) + \mathrm{C}(n) & \text{if } n > 1. \end{cases}$$

- ‘Divide’ step yields a subproblems of size 1/b each. (For merge-sort a=b=2)
- D(n) is the required time to divide the problem.
- C(n) is the required time to combine the solutions.

Recurrence for merge sort

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

- In this example, dividing the problem into two is a computation of the middle index (position) in the list. So, takes a constant time, $D(n) = \Theta(1)$.
- $C(n) = \Theta(n)$. Done by Procedure MERGE.
- Thus, $D(n)+C(n) = \Theta(n)$.

Recurrence for merge sort

- For now, we assume that the original problem size is a power of 2. Later, we'll see that this assumption does not affect the solution.
- $\lg n = \log_2 n$
- Let us construct a tree to solve the problem.
- Let c represent a constant time to solve a problem with $\Theta(1)$.

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

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.

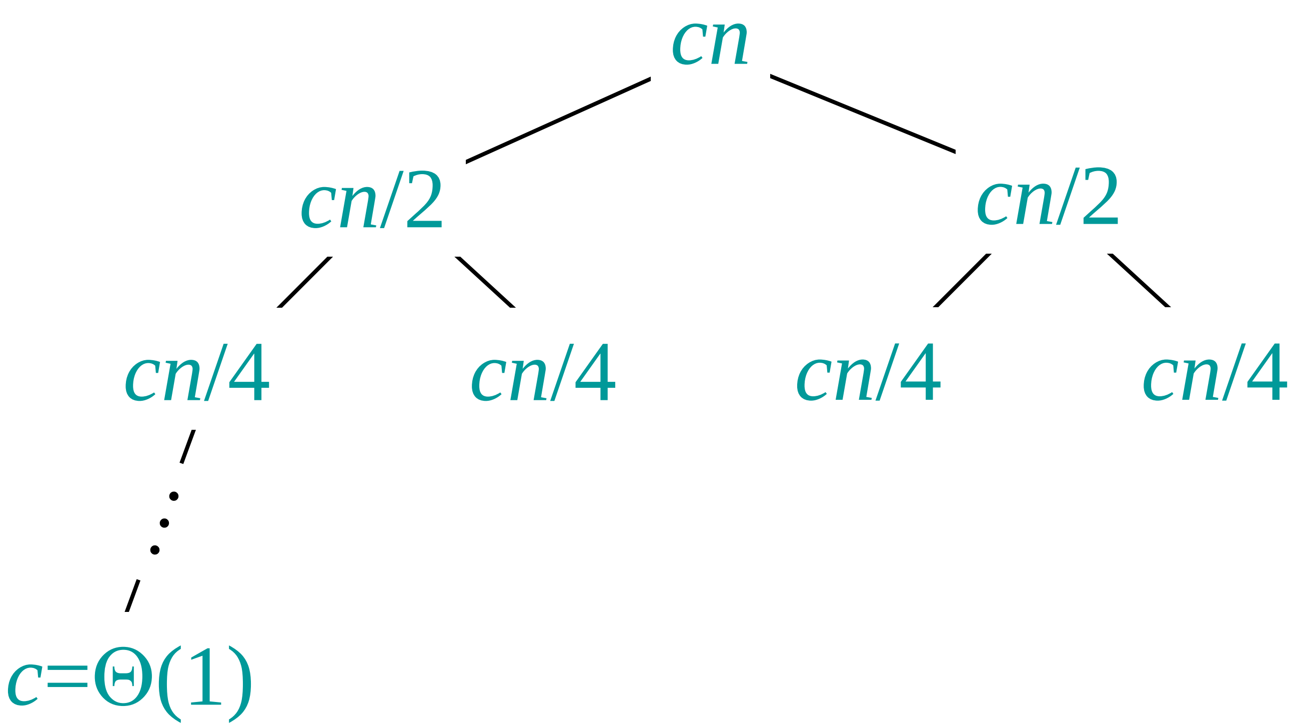
$$cn$$

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

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

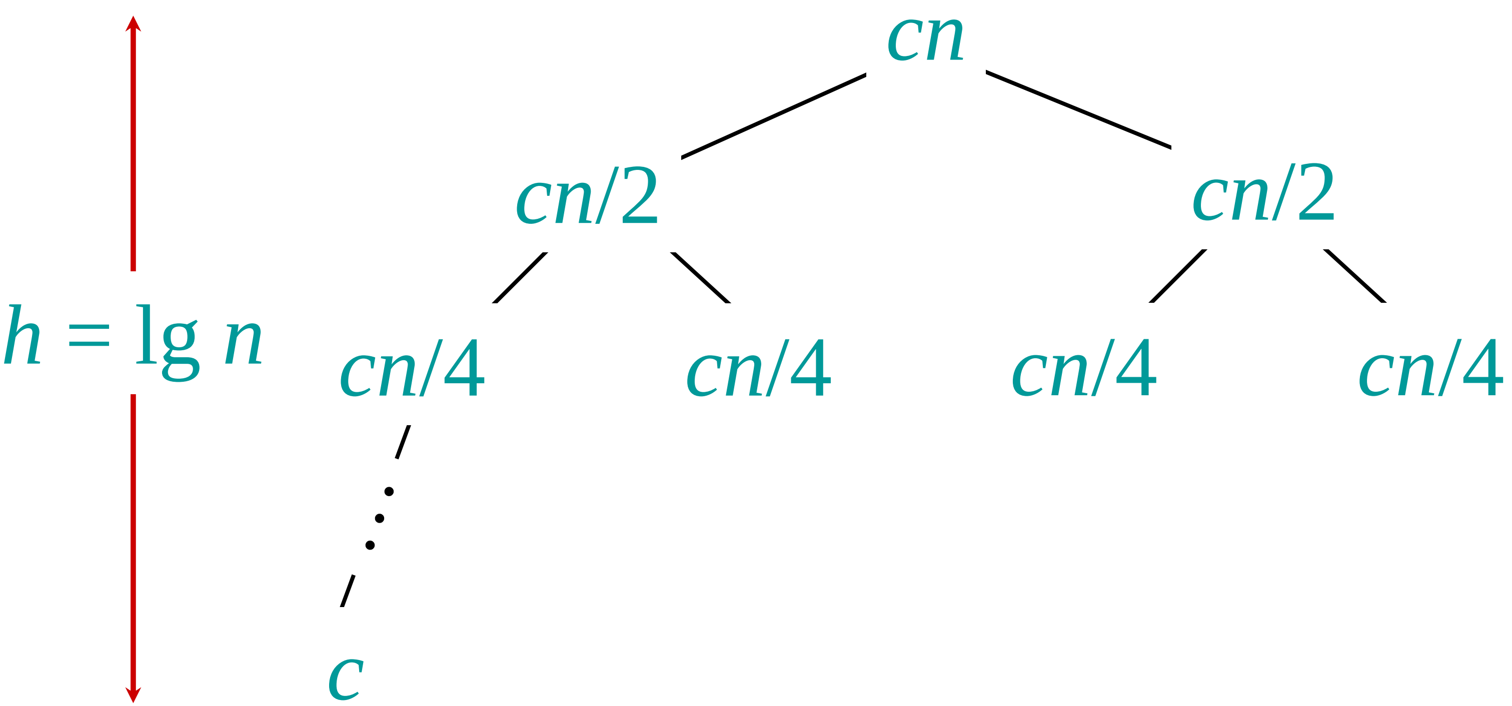
Recursion tree

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



Recursion tree

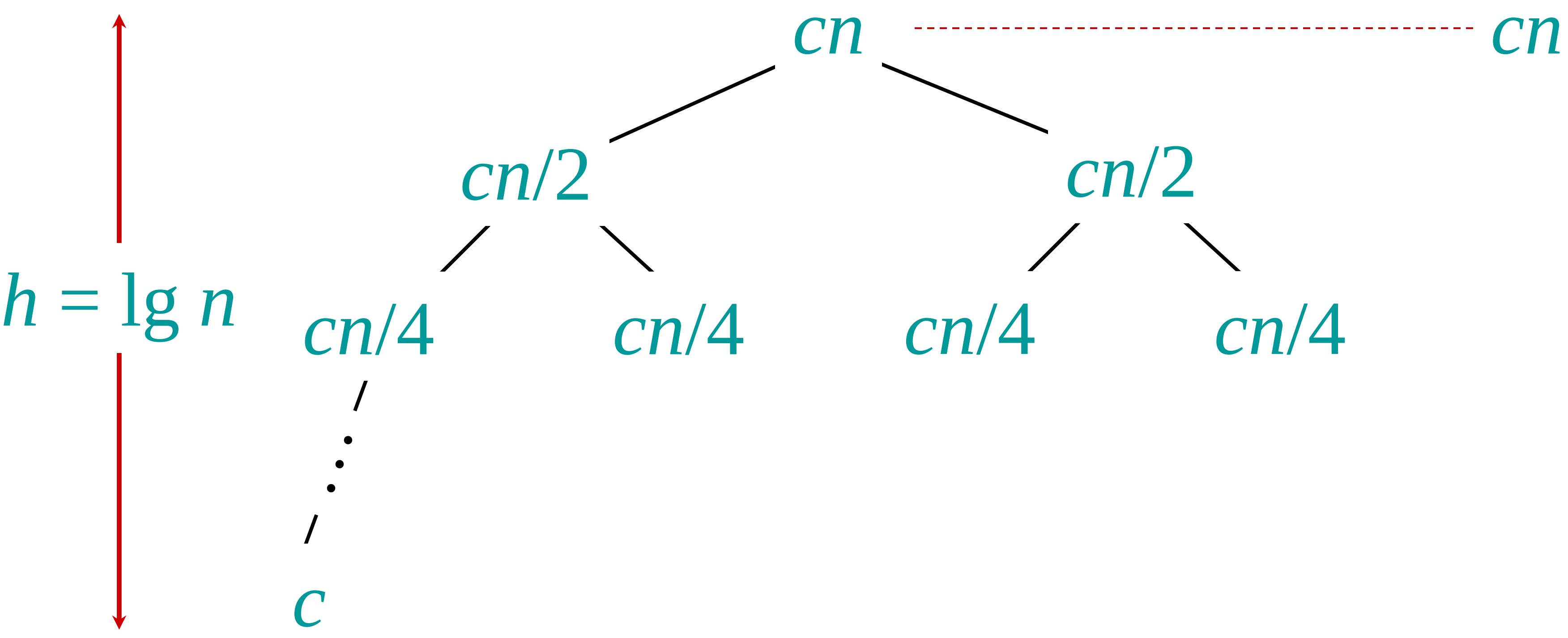
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Let $n=2^k$. It takes k steps to reach the bottom. $k=\log_2 n$

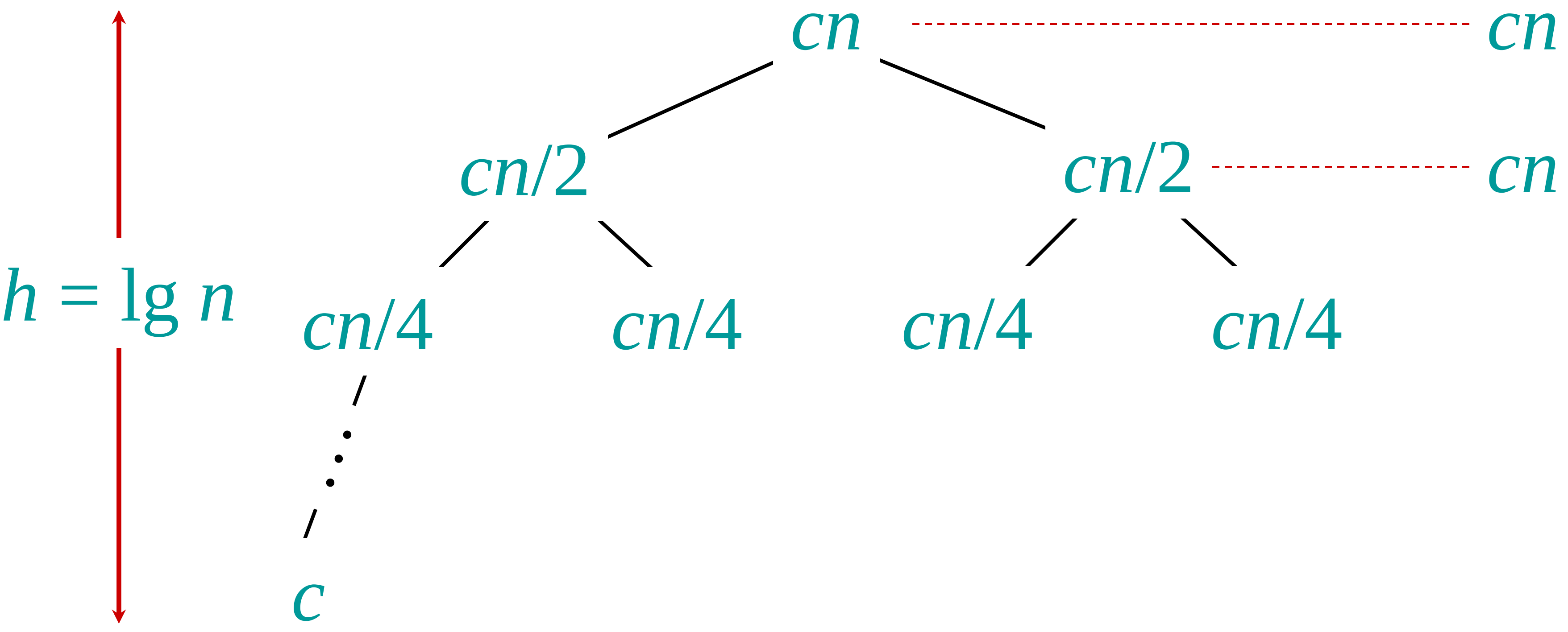
Recursion tree

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



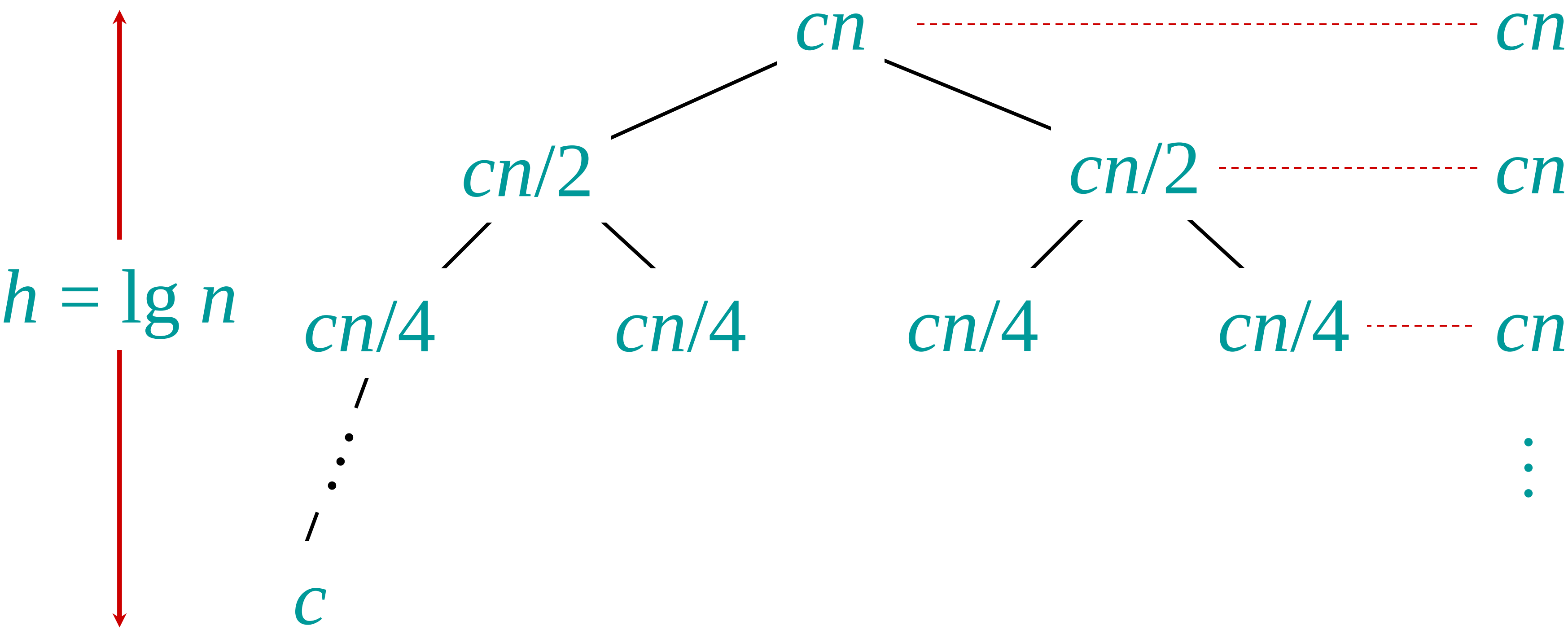
Recursion tree

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



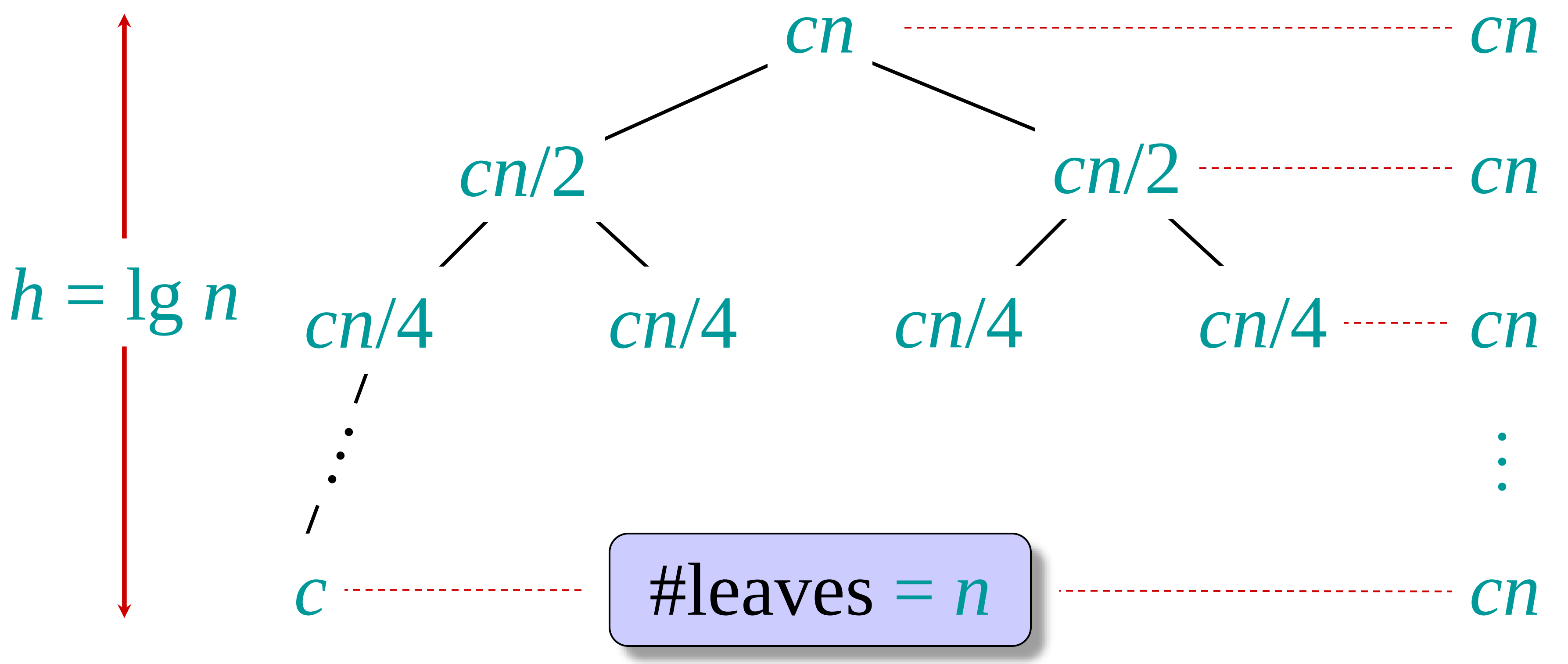
Recursion tree

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



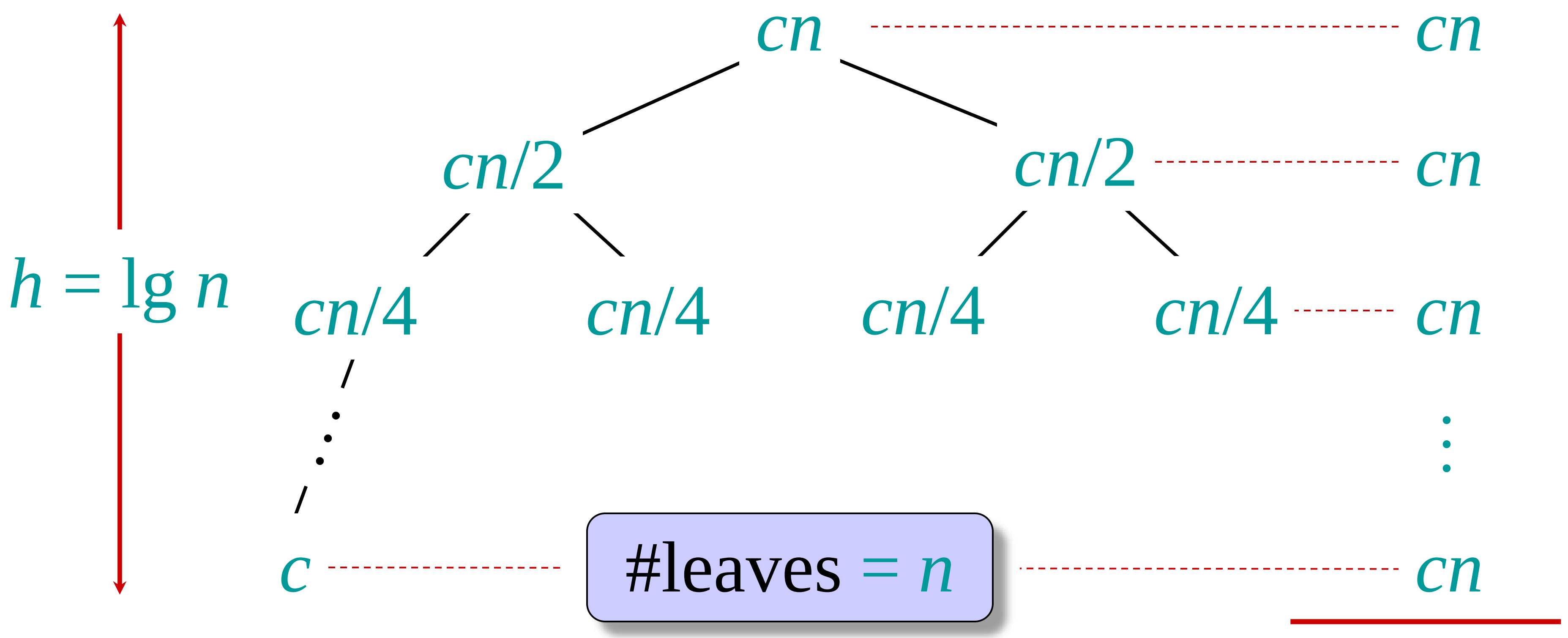
Recursion tree

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.



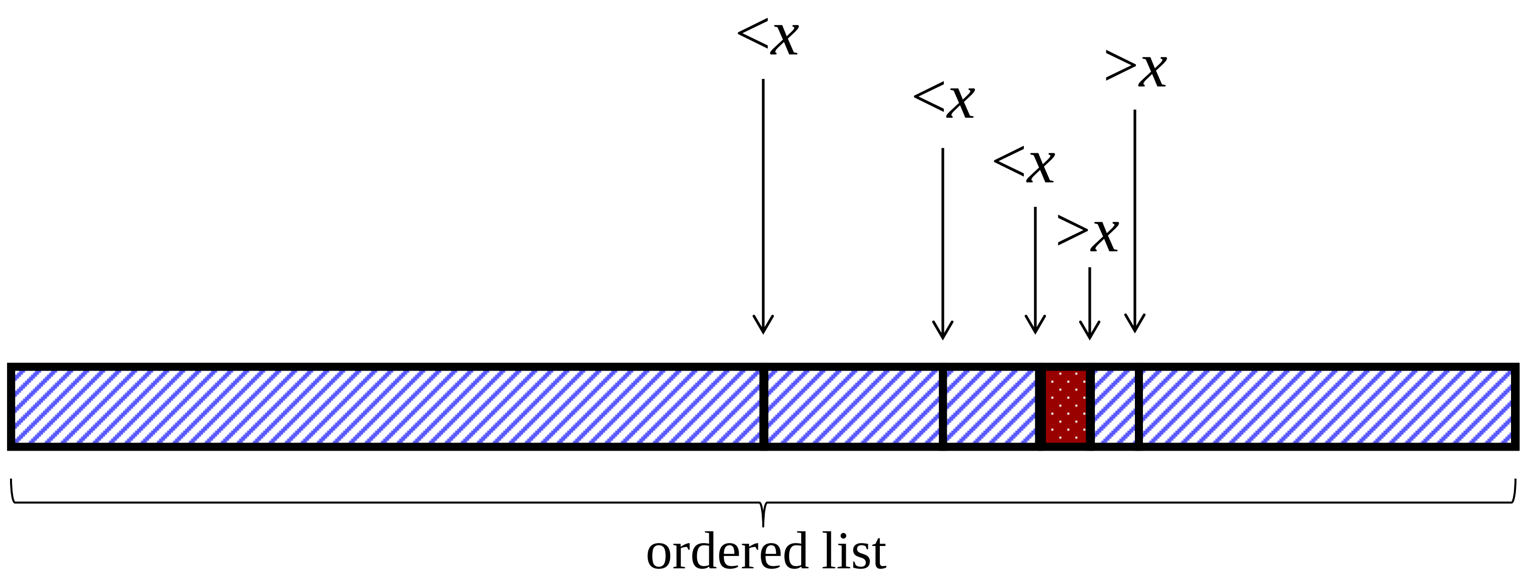
Total: $cn \lg n = \Theta(n \lg n)$

Comparison

- $\Theta(n \lg n)$ 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.

Example 2: Binary Search

- Search a particular element (x) in a given sorted list of n elements.
- Basic idea: On each step, look at the *middle term* of the remaining list to eliminate half of it.



Recursive Binary Search

RECURSIVE-BINARY-SEARCH (A, x, low, $high$)

{low and $high$ are the left and right endpoints of the search interval, x is the element searched.}

if $low > high$

return NIL {x is not found in the list}

$mid := \lfloor (low+high)/2 \rfloor$ {midpoint}

if $x = A[mid]$

return mid {x is found in position mid}

elseif $x > A[mid]$

return RECURSIVE-BINARY-SEARCH (A, x, $mid+1$, $high$)

else return RECURSIVE-BINARY-SEARCH (A, x, low, $mid-1$)

Note: There is also an iterative version of binary search. Please check Exercise 2.3-5.

Recursive Binary Search

- The procedure terminate the search when the range is empty (*low* > *high*) or when x is found.
- Otherwise, the search continues with the range halved.
- The recurrence is therefore

  $$T(n) = T(n/2) + c$$

  where c is constant time.
- Total time = $c \lg n = \Theta(\lg n)$

  (Let $n=2^k$. It takes $k=\log_2 n$ steps to reach the bottom)

Binary search analysis with recursion tree

c ---------- *complete list* ---------- c

c ---------- *half-list* ---------- c

$h = \lg n$

c ---------- *quarter-list* ---------- c

$\vdots$ $\vdots$

c ---------- *one element remained* ---------- c

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

The End

- Solve exercises in Chapter 2.2 and 2.3
- Solve problems of Chapter 2.