Design and Analysis of Algorithms

Merge Sort & Recurrence Relation

Spring 2022

National University of Computer and Emerging Sciences,
Islamabad

Merge sort

- The ***merge sort*** algorithm closely follows the **divide-and-conquer paradigm**. Intuitively,
- it operates as follows.
- **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 (Divide and Conquer)

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

Key Operation

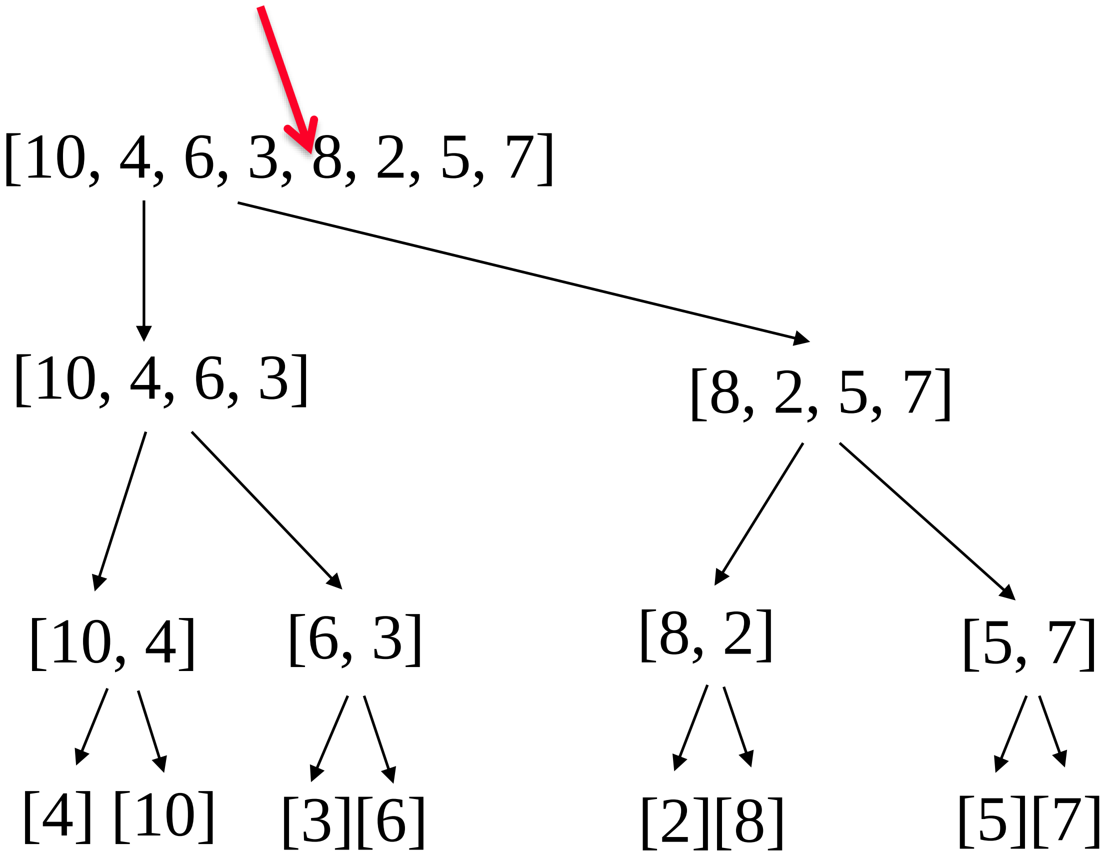
- The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step.
- We merge by calling an auxiliary procedure
- MERGE (A, p, q, r), where A is an array and p, q, and r are indices into the array such that $p \le q < r$.
- The procedure assumes that the subarrays A[p..q] and A[q+1.. r] are in sorted order.
- It ***merges*** them to form a single sorted subarray that replaces the current subarray A[p..r]

Visualization MergeSort

- https://opendsa-server.cs.vt.edu/embed/mergesortAV

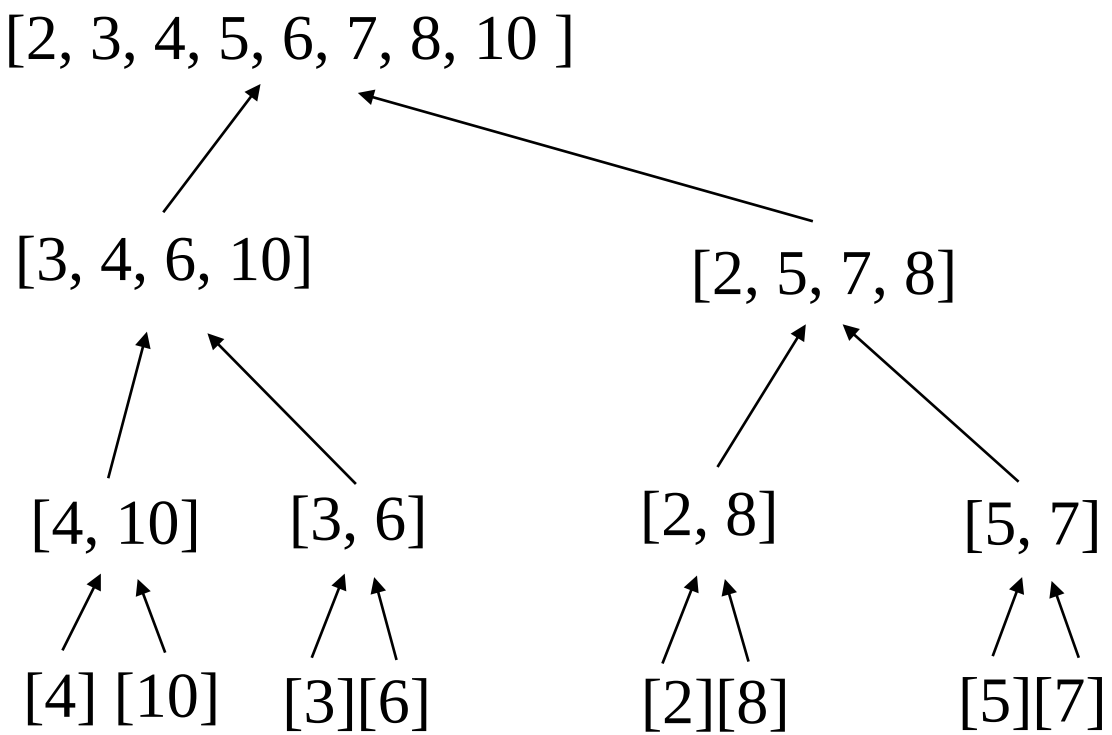
Example

- **Partition into lists of size n/2**



Example Cont'd

- **Merge**



```
MERGE(A, p, q, r)
n_1 = q - p + 1
n_2 = r - q
let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
for i = 1 to n_1
        L[i] = A[p + i - 1]
for j = 1 to n_2
        R[j] = A[q + j]
L[n_1 + 1] = ∞
R[n_2 + 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
```

Reference: Pg#31
Introduction to algorithms

Merge-Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Analysis of Merge Sort

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

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

Worst-case running time T(n) of Merge sort:

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

- Merge sort, with its θ (n lg n) running time, outperforms insertion sort, whose running time is θ (n^2), in the worst case.

Analysis of Merge Sort

?

Recurrences

- **The expression:**

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\dfrac{n}{2}\right) + cn & n > 1 \end{cases}$$

is a *recurrence*.

  - **Recurrence: an equation that describes a function in terms of its value on smaller functions**

Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

Example

Function(int number)

```
if number<=1
  then return;
else
     Function(number/2)
     Function(number/2)
   for(i=1 to number )
          Display i;
```

It follows that

$$T(n) = 2\,T(n/2) + cn$$

work done dividing and combining

number of sub-problems

Sub-problem size

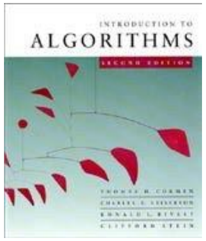
Recursion tree

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

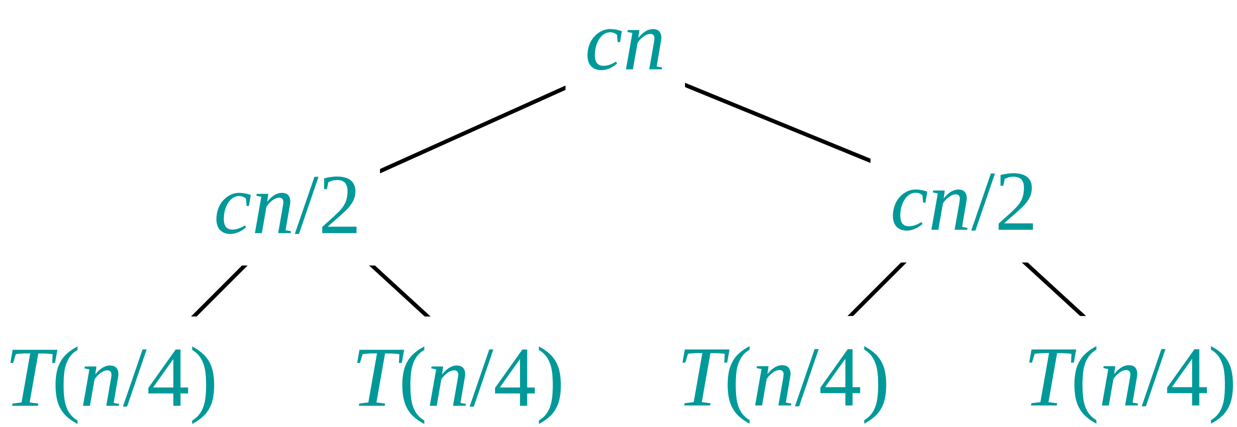
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

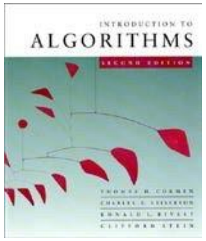
$$cn$$

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

Recursion tree

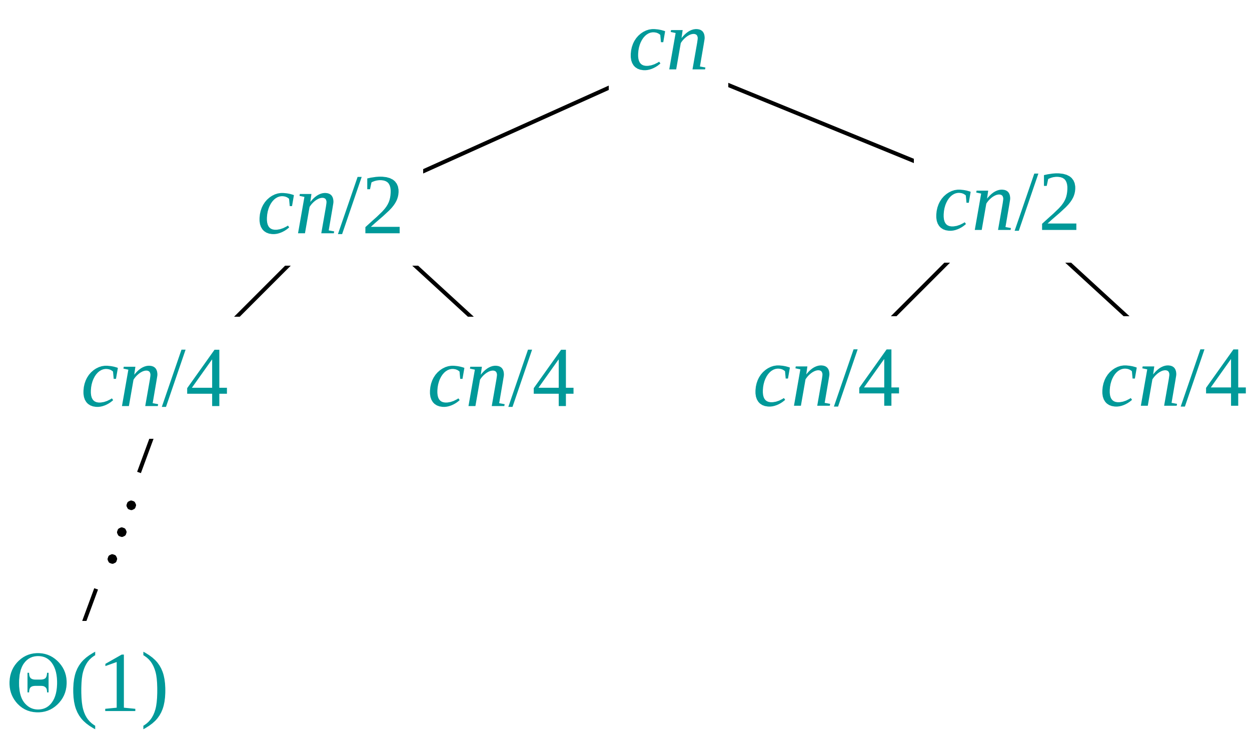
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

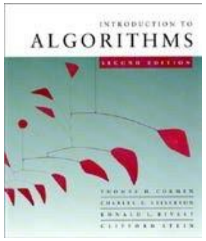




Recursion tree

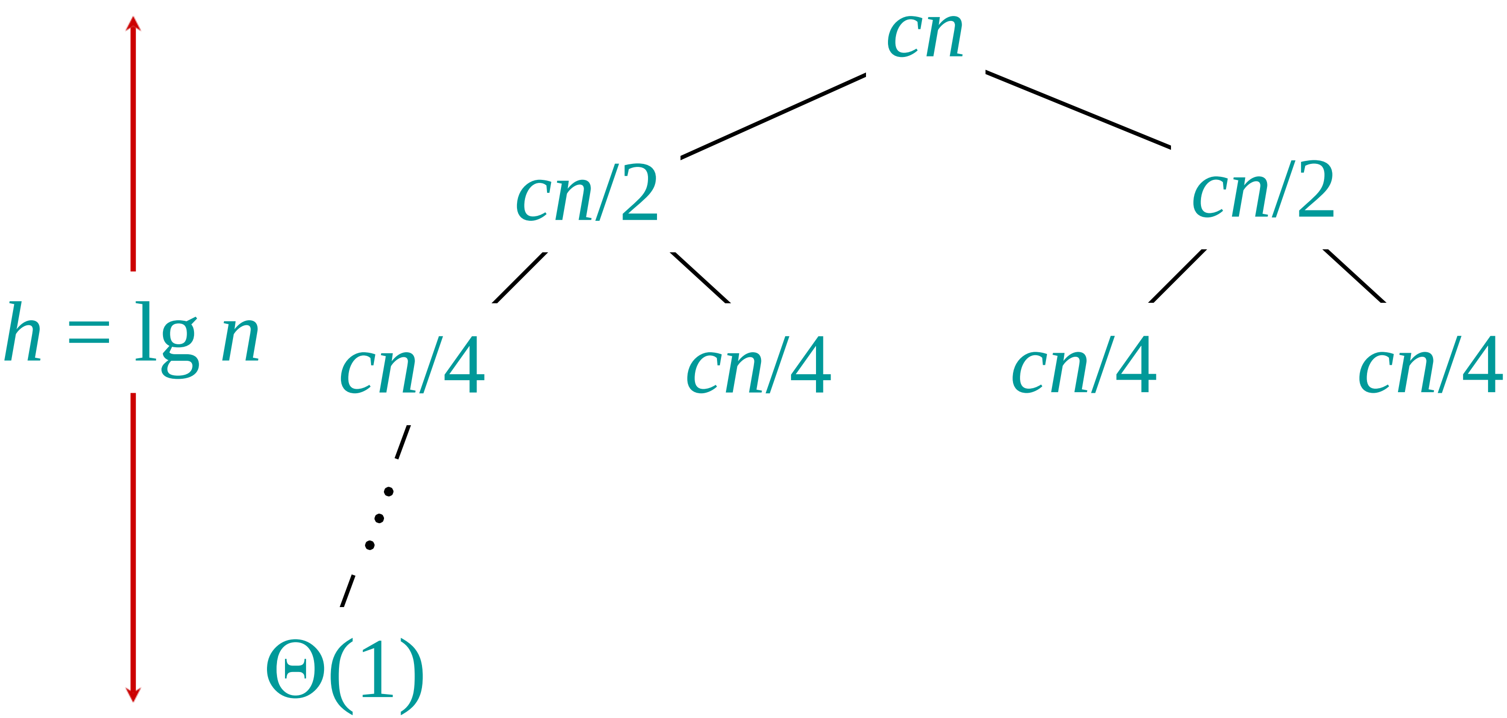
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

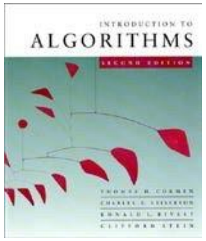




Recursion tree

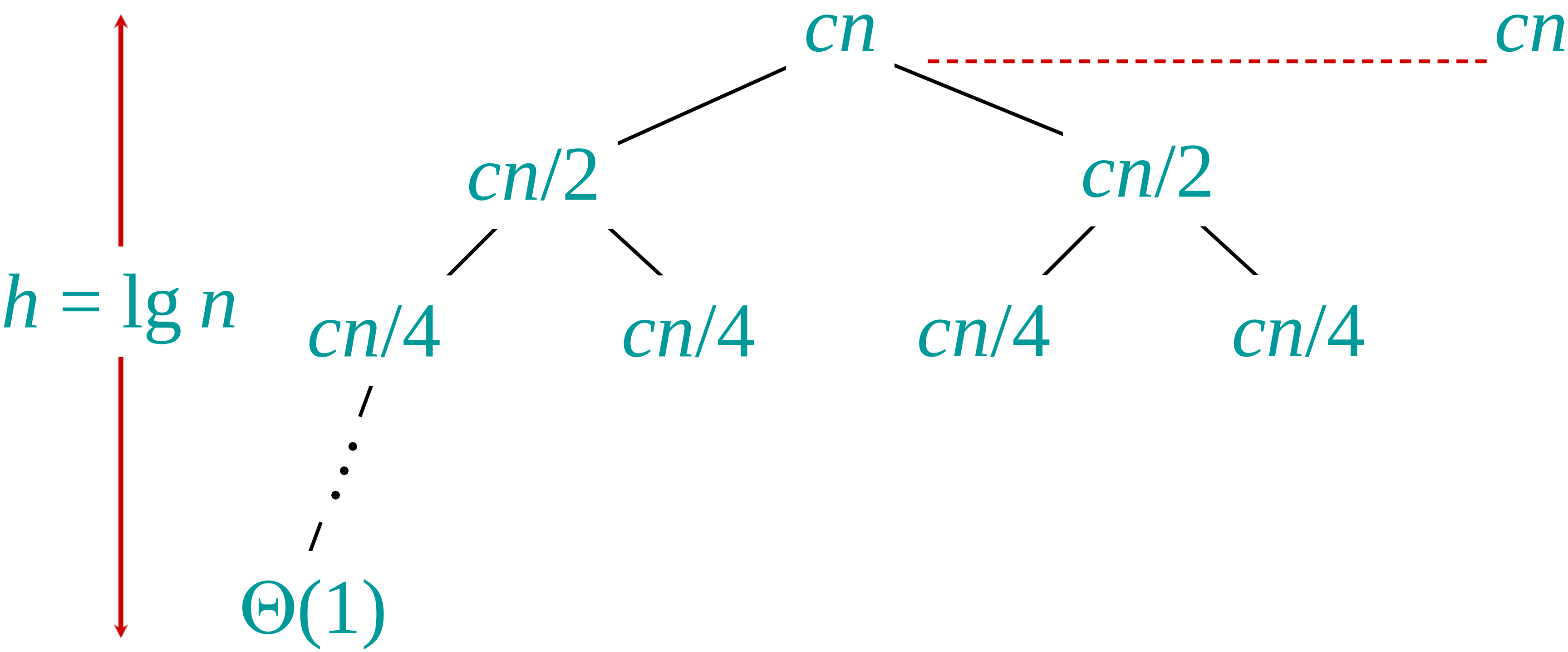
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

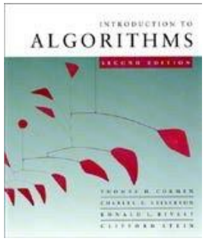




Recursion tree

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.

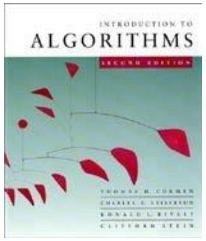
cn $\quad$ cn

$cn/2$ $\quad$ $cn/2$ $\quad$ cn

$h = \lg n$ $\quad$ $cn/4$ $\quad$ $cn/4$ $\quad$ $cn/4$ $\quad$ $cn/4$

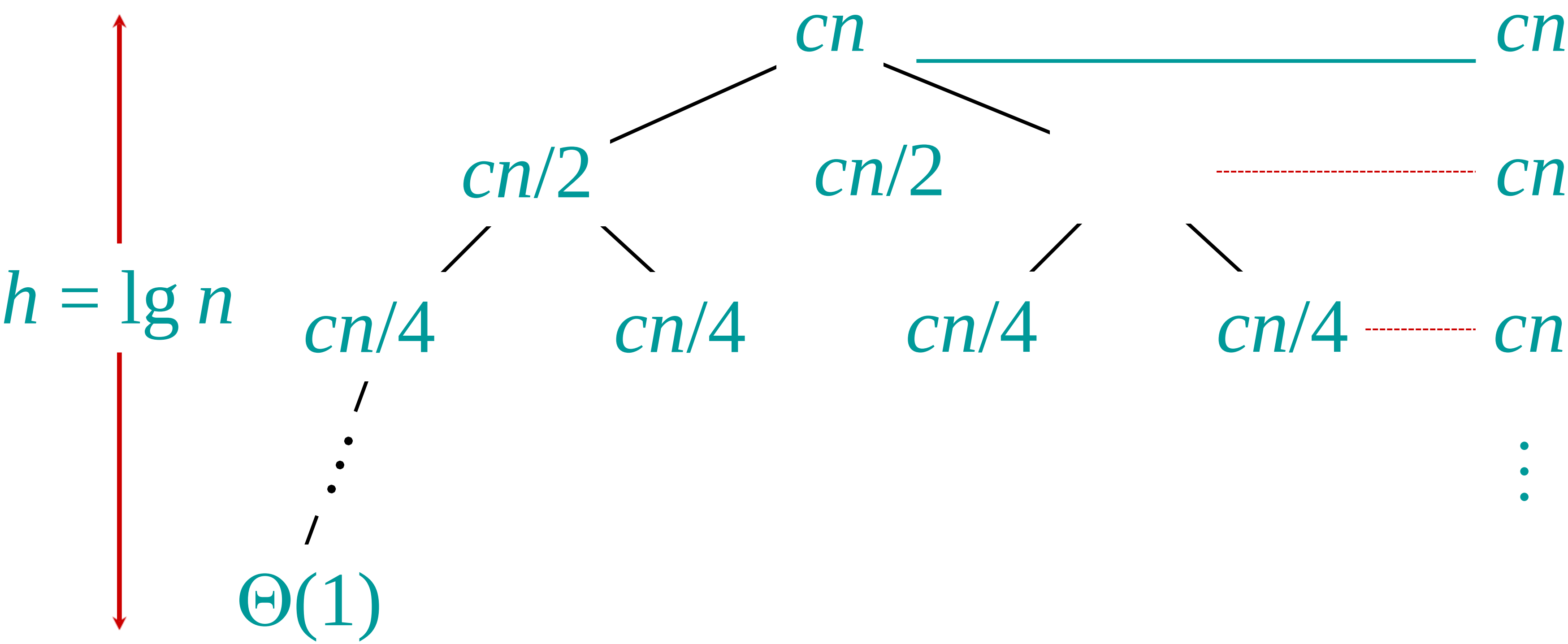
$\vdots$

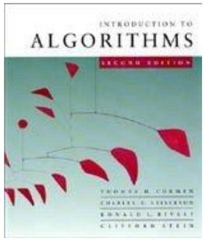
$\Theta(1)$



Recursion tree

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

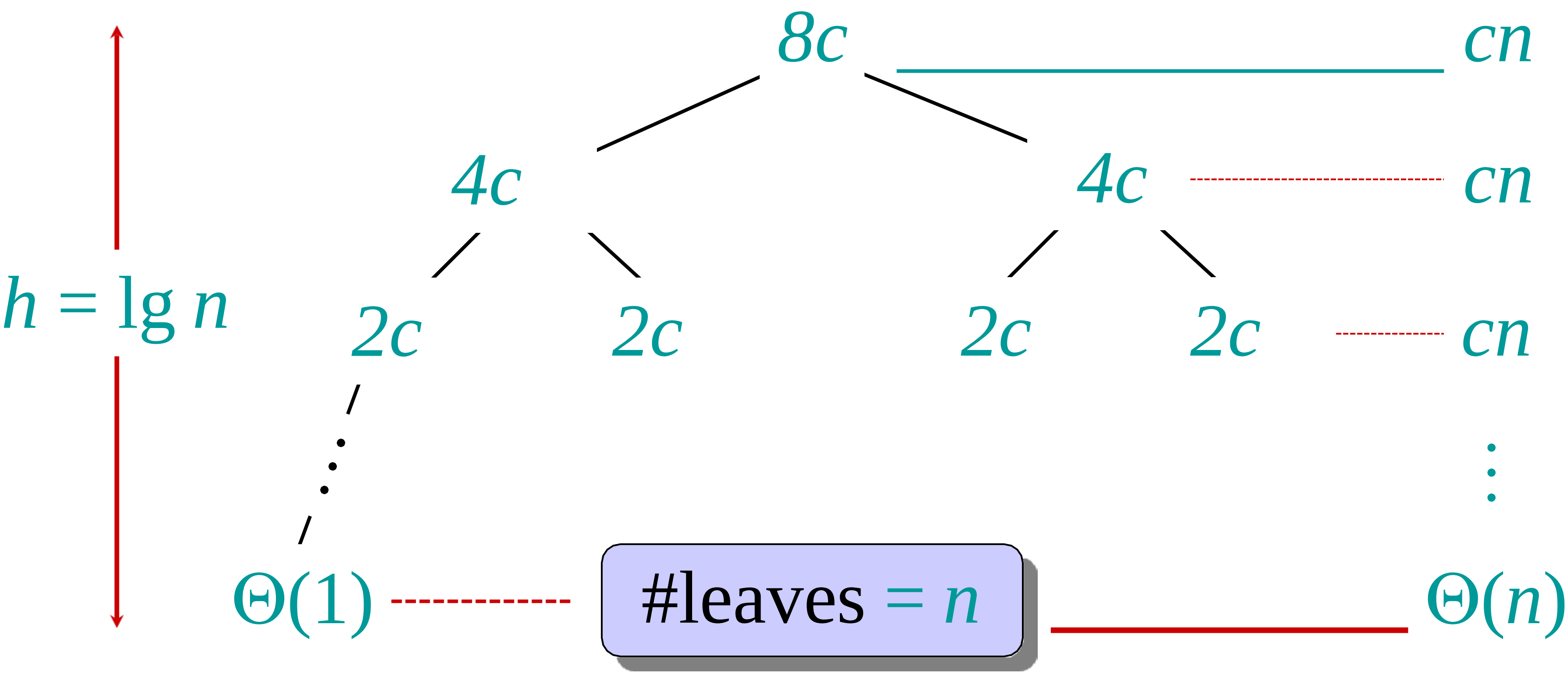


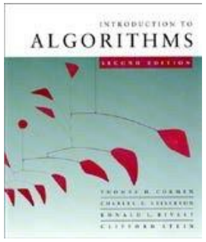


Recursion tree

n=8

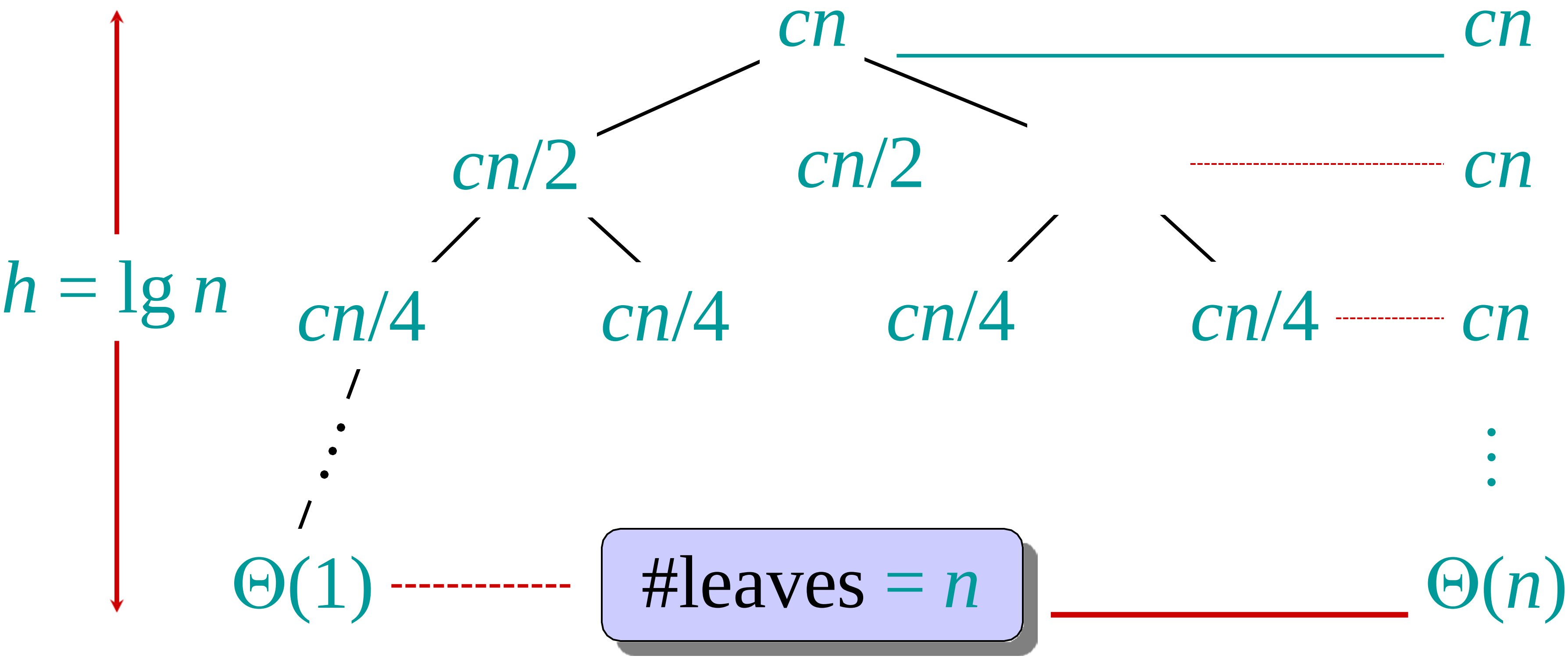
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.





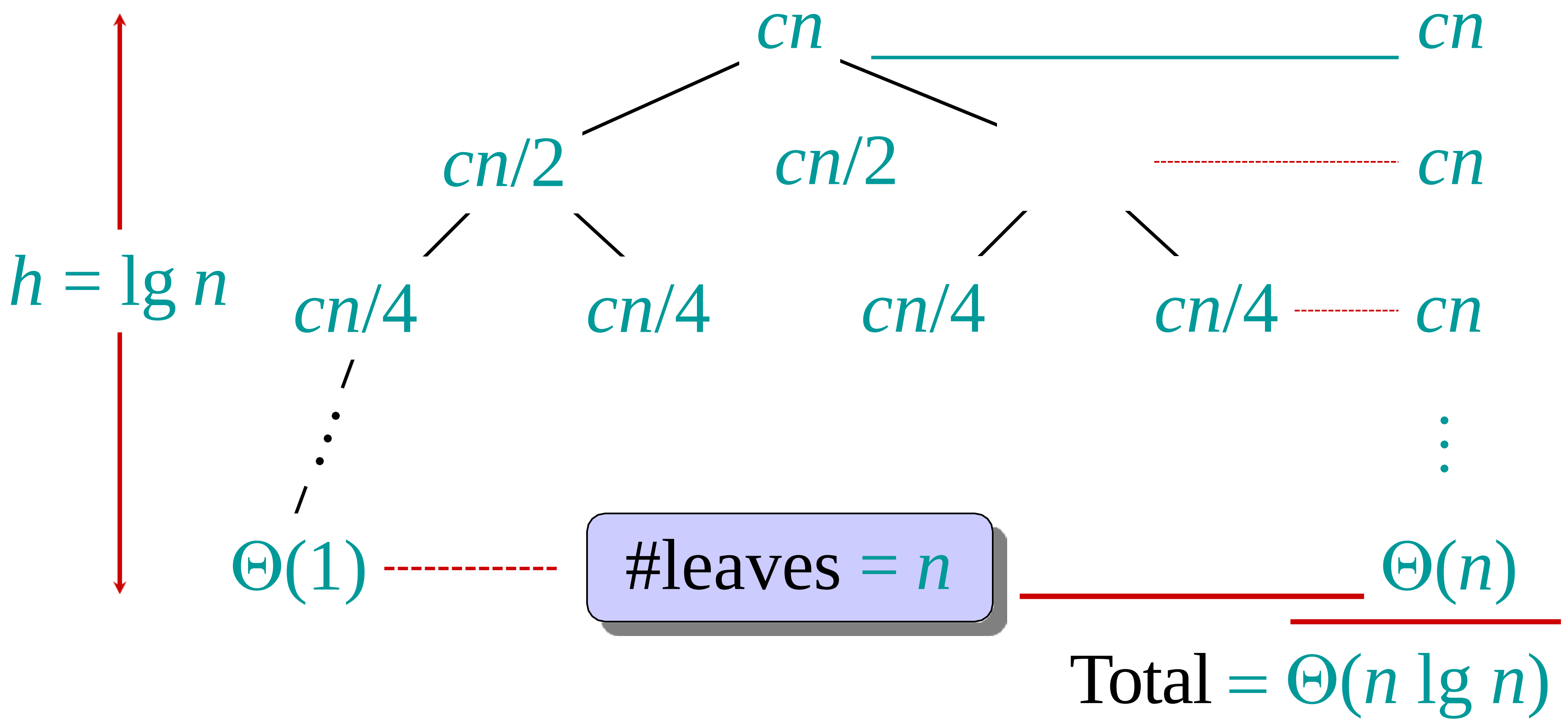
Recursion tree

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

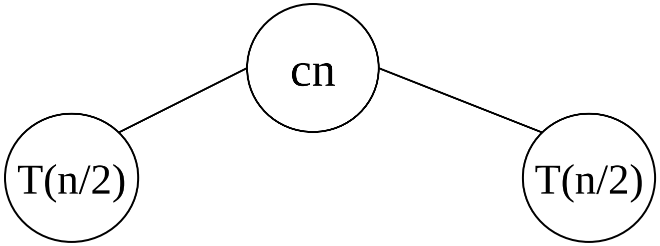


Recursion tree

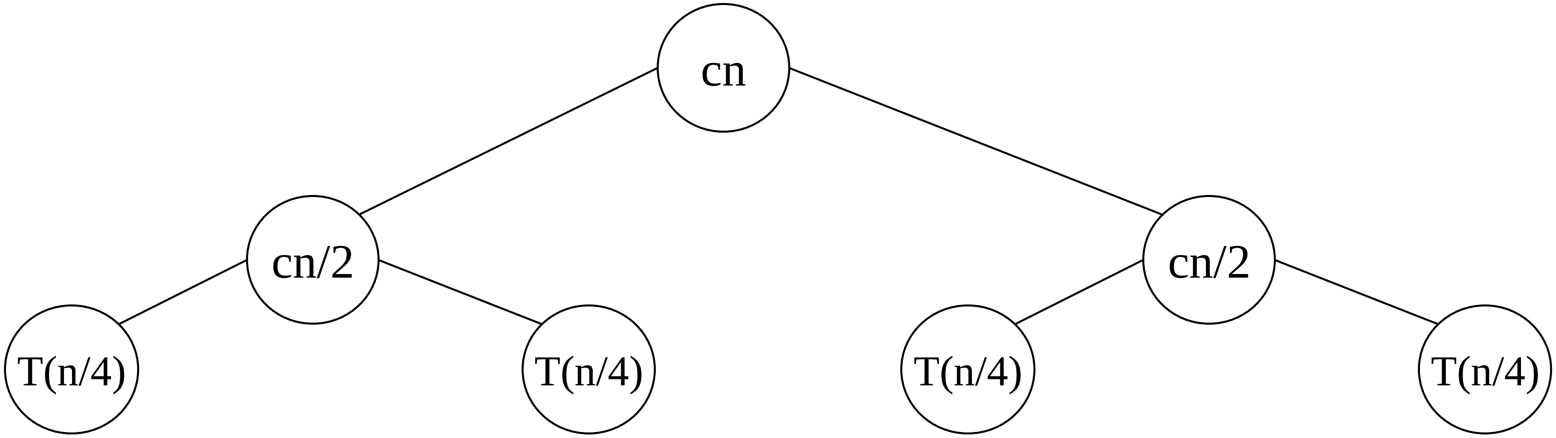
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



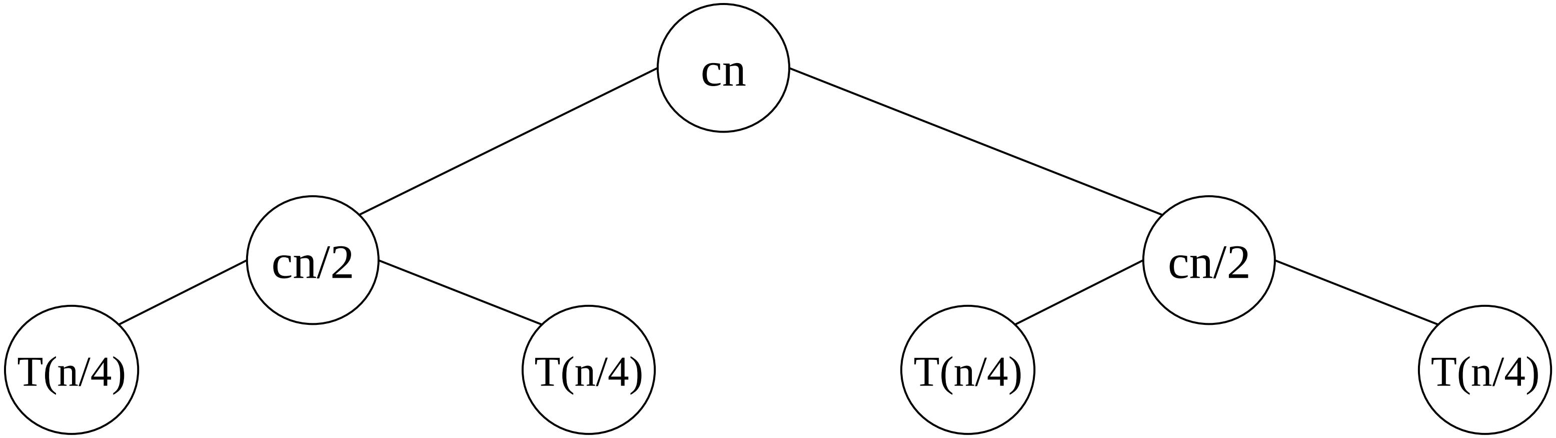
Recursion Tree for Algorithm



Recursion Tree for Algorithm

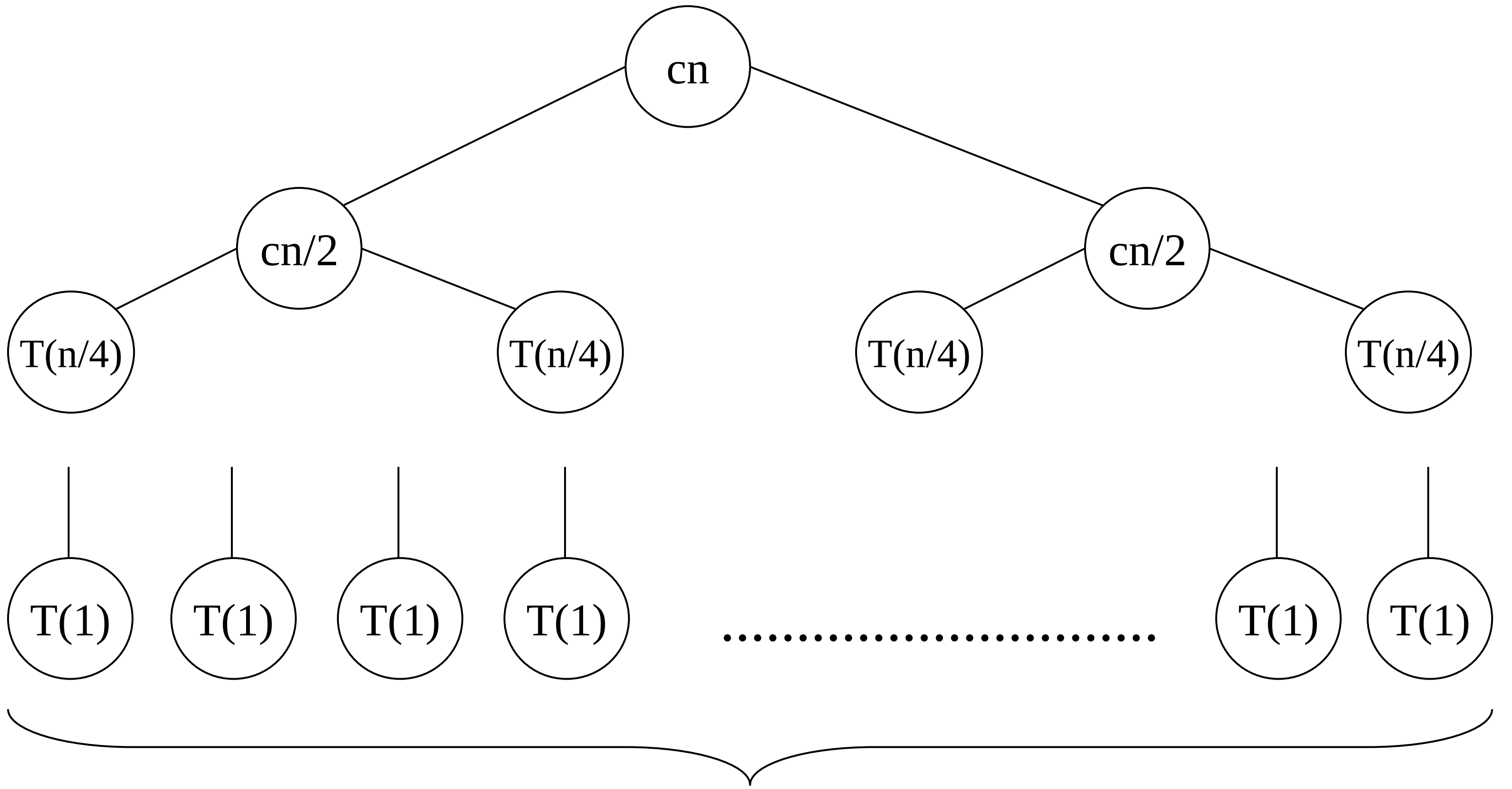


Recursion Tree for Algorithm



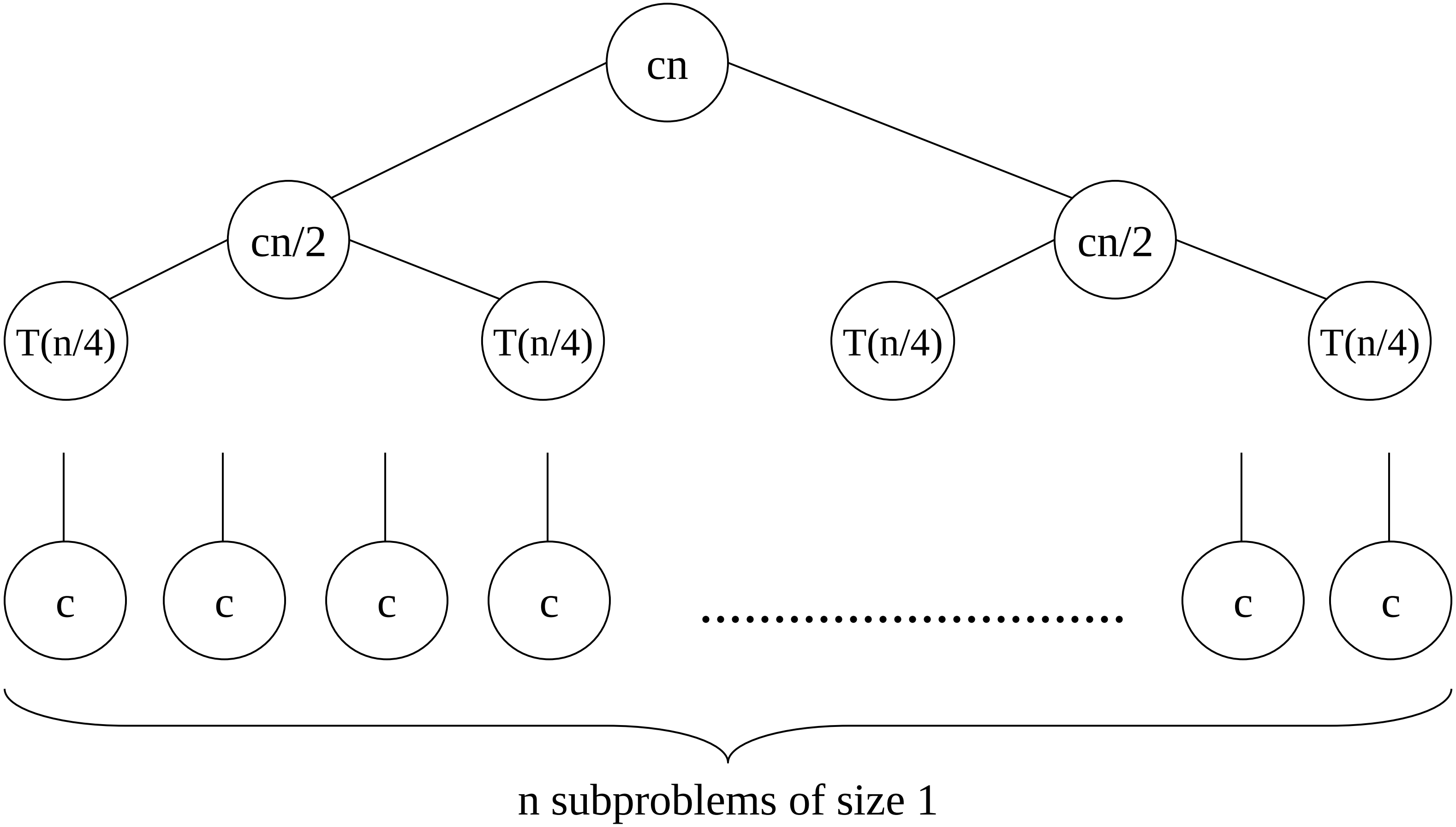
Eventually, the input size (the argument of T) goes to 1, so...

Recursion Tree for Algorithm



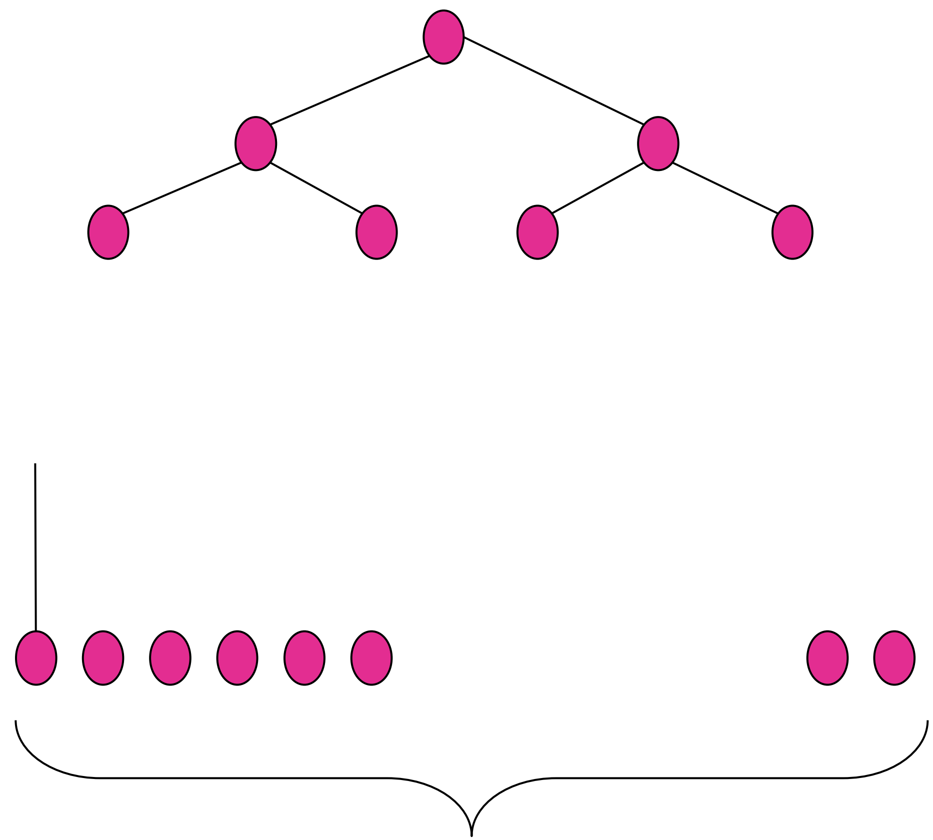
n sub problems of size 1, but T(1) = c
by boundary condition

Recursion Tree for Algorithm



Recursion Tree for Algorithm

level	nodes/ level	cost/ level
0	$2^0 = 1$	cn
1	$2^1 = 2$	cn
2	$2^2 = 4$	cn
.	.	
.	.	
.	.	
N-1	$2^{N-1}=n$	cn



n nodes at level N-1

Since $2^{N-1} = n$,

$N-1 = \lg(n)$

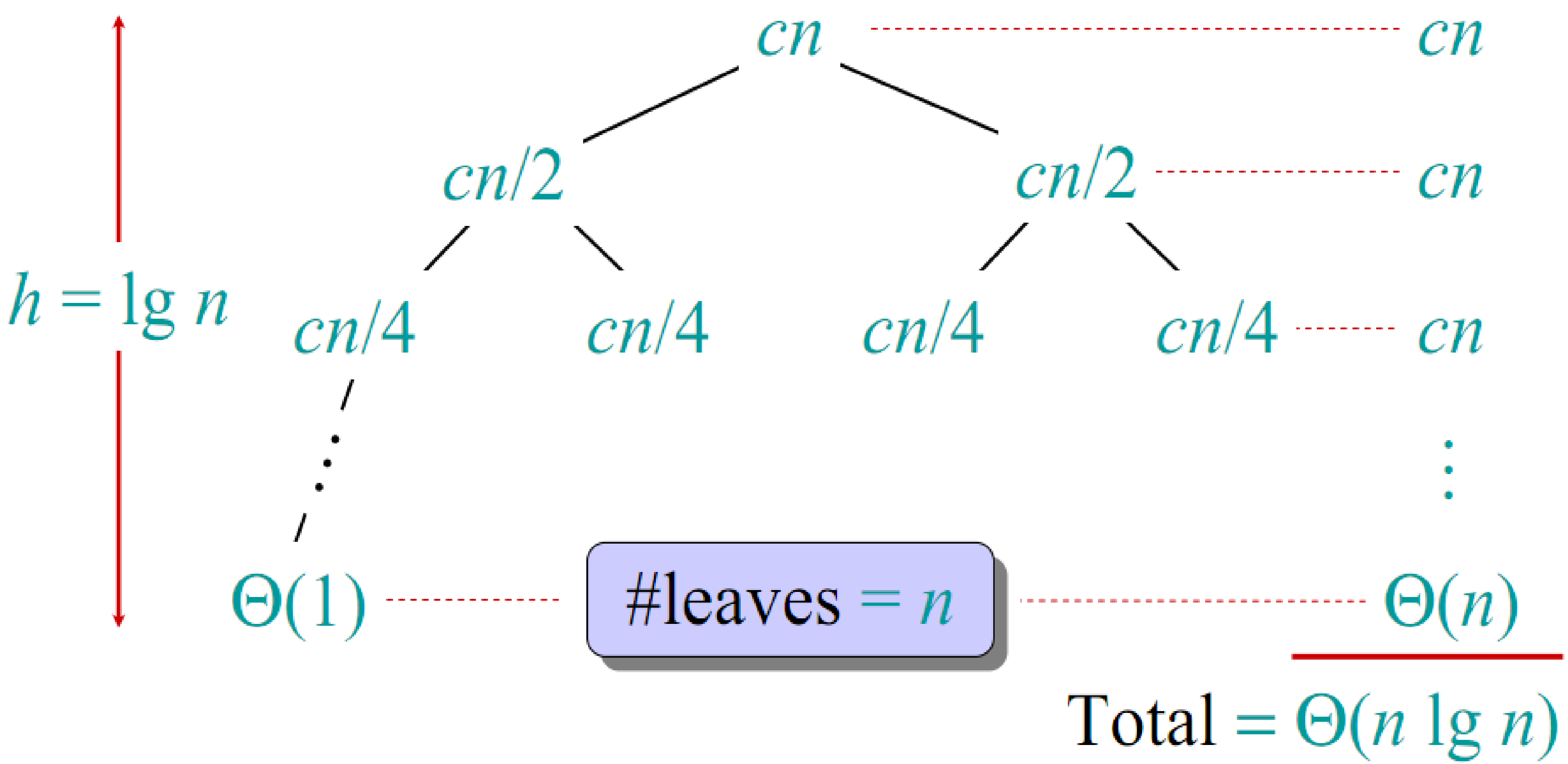
levels = $N = 1+\lg(n)$

T(n) = total cost = (levels)(cost/level)

$T(n) = cn\,[1+\lg(n)] = O(n \lg(n))$

Visual Representation of the Recurrence for Merge Sort

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



Time Complexity (Using Master Theorem)

- **Recurrence Relation**

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

Using Master Theorem applying case 2:

$$\Theta\left(n^{\log_b a} \log n\right)$$

So time complexity is O(nlogn)

- $\Theta(n\lg n)$ 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 >=3

Sorting algorithms

- **Selection and bubble sort have quadratic best/average/worst-case performance**
- **Insertion sort has quadratic average-case and worst-case performance**
- **The faster comparison based algorithm ?**

  O(nlogn)

- **Mergesort and Quicksort**

Solving Recurrences

- **The substitution method**
  - **A.k.a. "making a good guess method"**
  - **Guess the form of the answer, then use mathematical induction to find the constants and show that the solution works**
  - **Example:**
    - **$T(n) = 2T(n/2) + n$ □ $T(n) = O(n \lg n)$**

- **https://opendsa-server.cs.vt.edu/embed/mergesortAV**
- **https://www.youtube.com/watch?v=4V30R3I1vLI&ab_channel=AbdulBari**