Design and Analysis of Algorithms

Recursion

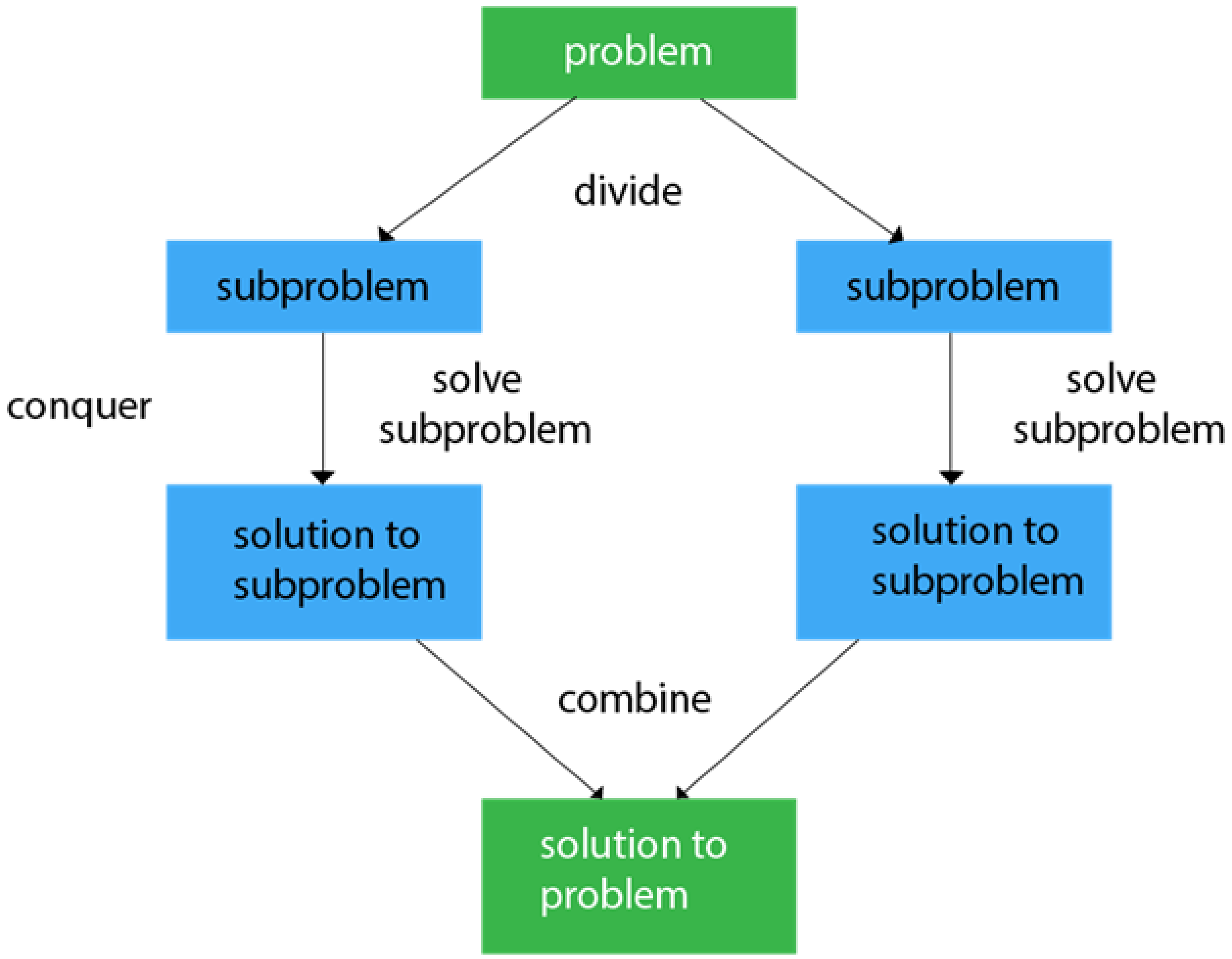
Fall 2022

National University of Computer and Emerging Sciences,
Islamabad

Divide-and-Conquer

- **Divide** the problem into a number of subproblems that are smaller instances of the
- same problem.
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

Divide and Conquer



Recursion (2)

- **To solve problem recursively**

```
1. Define the base case(s)
2. Define the recursive case(s)
    a) Divide the problem into smaller sub-problems
    b) Solve the sub-problems
    c) Combine results to get answer
```

Sub-problems solved as a recursive call to the same function

- **Sub-problem must be smaller than the original problem**
  - **Otherwise recursion never terminates**

Recursion

Recursion occurs when a function/procedure calls itself.

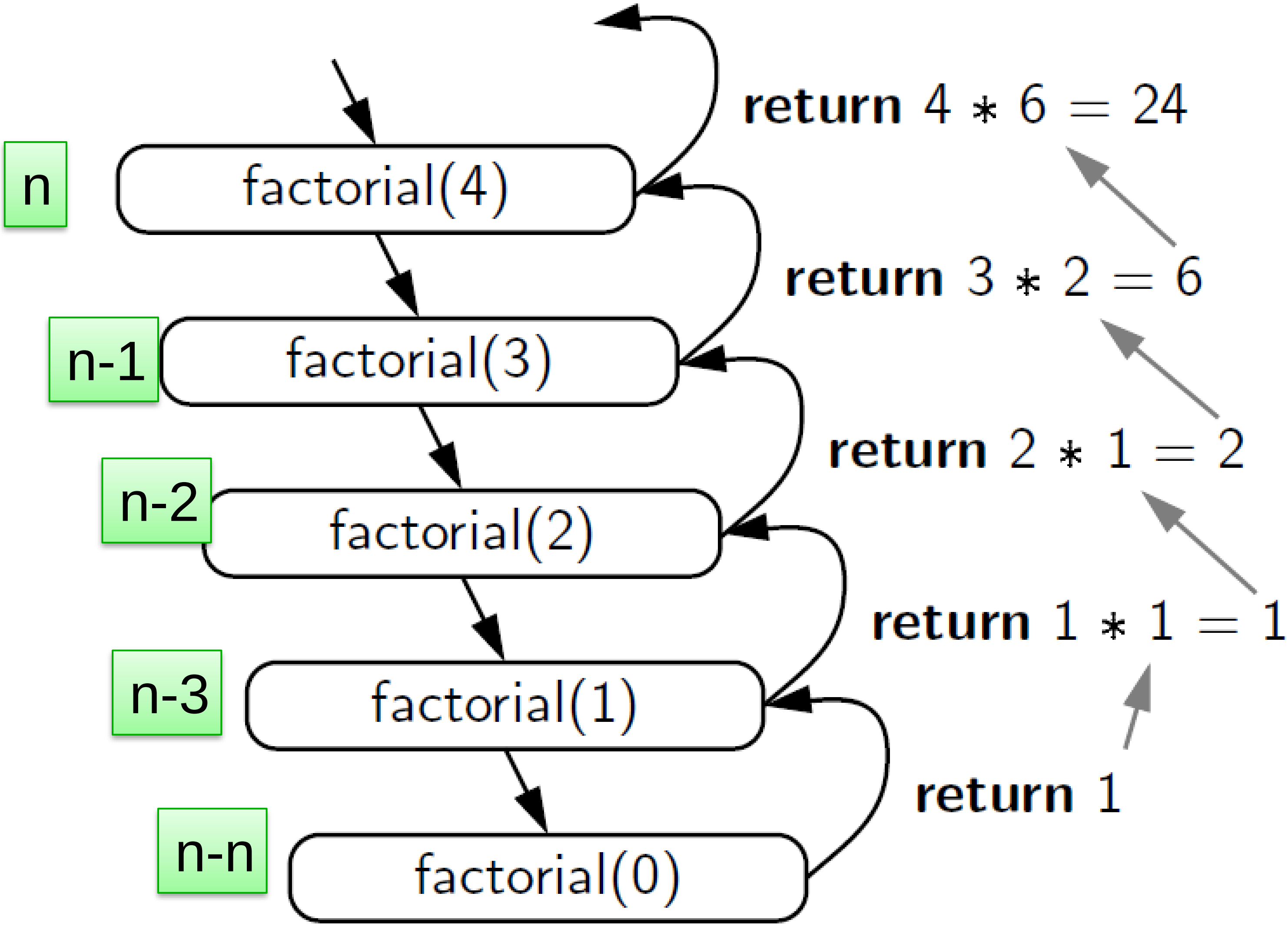
Many algorithms can be best described in terms of recursion.

Example: Factorial function

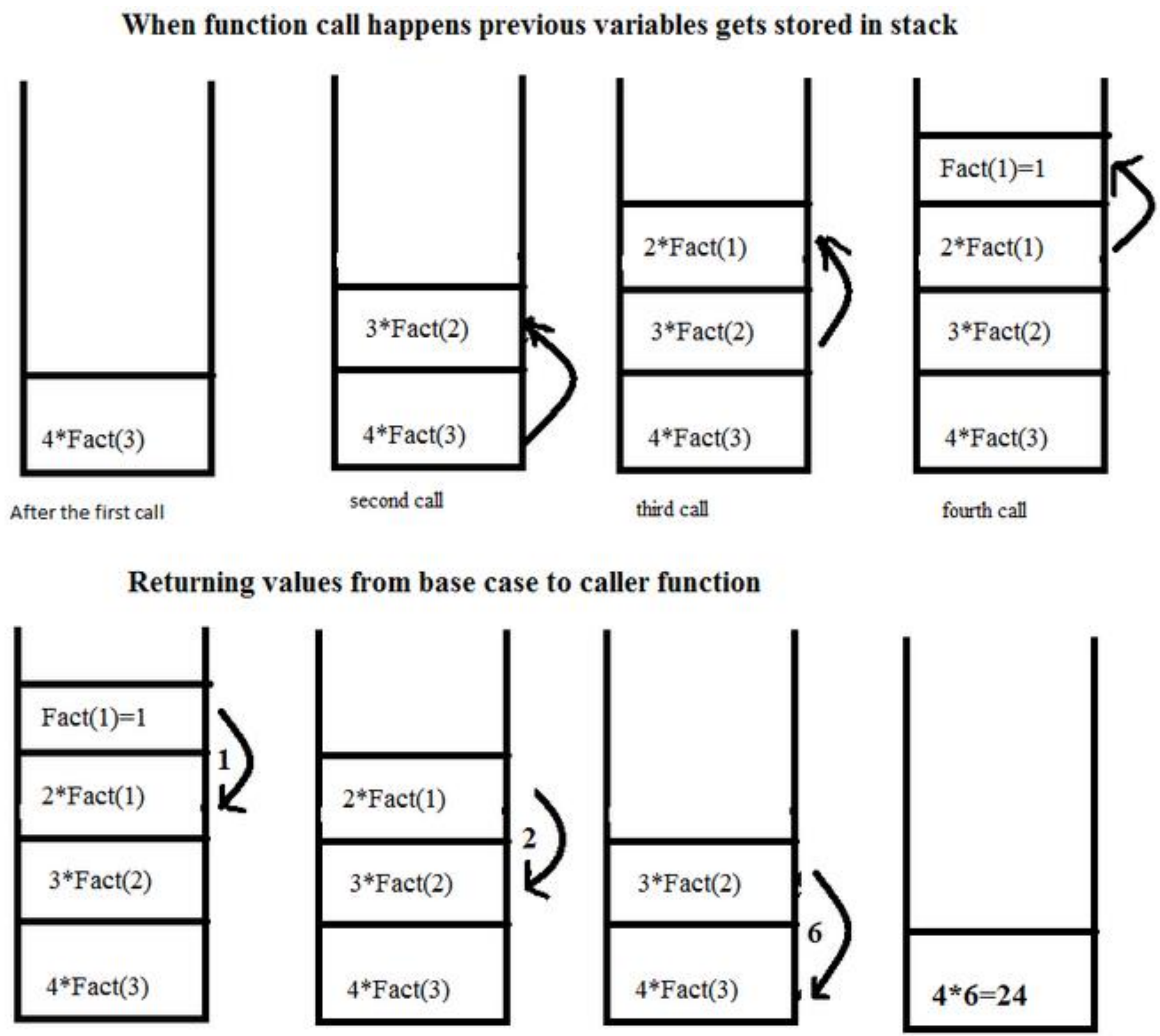
The product of the positive integers from *1* to *n* inclusive is called "**n factorial**", usually denoted by n!:

$$n! = 1 * 2 * 3 .... (n-2) * (n-1) * n$$

Factorial function



Recursive Definition of the Factorial Function



Stack Overflow!

- **Recursive functions cannot use statically allocated local variables**
  - **Each instance of the function needs its own copies of local variables**

- **Most modern languages allocate local variables for functions on the run-time stack**

- **Calling a recursive function many times or with large arguments may result in stack overflow**

Recursive Definition of the Factorial Function

Recursive Definition of the Factorial Function

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$$

$5! = 5 * 4!$	$= 5 * 24 = 120$
$4! = 4 * 3!$	$= 4 * 3! = 4 * 6 = 24$
$3! = 3 * 2!$	$= 3 * 2! = 3 * 2 = 6$
$2! = 2 * 1!$	$= 2 * 1! = 2 * 1 = 2$
$1! = 1 * 0!$	$= 1 * 0! = 1$

Recursive Definition of the Fibonacci Numbers

The Fibonacci Sequence is the series of numbers:
1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The Fibonacci numbers are a series of numbers as follows:

Recursive Definition of the Fibonacci Numbers

The Fibonacci Sequence is the series of numbers:
1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The Fibonacci numbers are a series of numbers as follows:

fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
...

$$\text{fib}(n) = \begin{cases} 1, & n <= 2 \\ \text{fib}(n-1) + \text{fib}(n-2), & n > 2 \end{cases}$$

fib(3) = 1 + 1 = 2
fib(4) = 2 + 1 = 3
fib(5) = 2 + 3 = 5

How do I write a recursive function?

- Determine the size factor
- Determine the base case(s)
  (the one for which you know the answer)
- Determine the general case(s)
  (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm
  (use the "Three-Question-Method")

Using Recursion Properly

For correct recursion we need two parts:

1. One (ore more) **base cases** that are not recursive, i.e. we can directly give a solution:
```
if (n==1)
    return 1;
```

2. One (or more) **recursive cases** that operate on smaller problems that get closer to the base case(s)
```
return n * factorial(n-1);
```

The base case(s) should always be checked before the recursive calls.

Counting Digits

- **19865 (5 Digit Number)**
- **386(3 Digit Number)**

Counting Digits

- **Recursive definition**

digits(n) = 1 **if (–9 <= n <= 9)** Base Case

1 + digits(n/10) **otherwise** Recursive case

- **Example**

digits(321) =

1 + digits(321/10) = 1 +digits(32) =

1 + [1 + digits(32/10)] = 1 + [1 + digits(3)] =

1 + [1 + (1)] =

3

Counting Digits in C++

```
int numberofDigits(int n)
{
  if ((-10 < n) && (n < 10))
    return 1;
  else
    return 1 +
  numberofDigits(n/10);
}
```

Base Case

Recursive case

Evaluating Exponents Recursively

```
5 ^ 4 =?
power(5, 4)
```

Evaluating Exponents Recursively

```
int power(int k, int n) {
  // raise k to the power n
  if (n == 0)
    return 1;
  else
    return k * power(k, n - 1);
}
```

5 ^ 4 =?

power(5, 4)

625

5 * power(5, 3) = 5 ***5 ^ 3**

125

5 * power(5, 2) = 5 ***5 ^ 2**

25

5 * power(5, 1) = 5 ***5 ^ 1**

5

5 * power(5, 0) = 5*1

1

1

Evaluating Exponents Recursively

```
int power(int k, int n) {
   // raise k to the power n
   if (n == 0)
      return 1;
   else
      return k * power(k, n - 1);
}
```

Divide and Conquer

- Using this method each recursive subproblem is about one-half the size of the original problem
- If we could define `power` so that each subproblem was based on computing $k^{n/2}$ instead of k^{n-1} we could use the divide and conquer principle
- Recursive divide and conquer algorithms are often more efficient than iterative algorithms

Evaluating Exponents Using Divide and Conquer

```
int power(int k, int n) {
  // raise k to the power n
  if (n == 0)
    return 1;
  else{
    int t = power(k, n/2);
    if ((n % 2) == 0)
      return t * t;
    else
      return k * t * t;
}
```

Evaluating Exponents Using Divide and Conquer

```
int power(int k, int n) {
  // raise k to the power n
  if (n == 0)
    return 1;
  else{
    int t = power(k, n/2);
    if ((n % 2) == 0)
      return t * t;
    else
      return k * t * t;
}
```

5 ^ 4 =?

power(5, 4)

n=4,
k=5

power(5, **4**/2)
= power(5, 2)

n=4,
t=25

25*25=625

n=2,
k=5

power(5, **2**/2)
= power(5, 1)

5*5=25

n=2,
t=5

n=1,
k=5

power(5, **1**/2)
=power(5, **0)**

5*1*1

n=1,
t=1

n=0,
k=5

return 1

1

Disadvantages

- **May run slower.**
  - **Compilers**
  - **Inefficient Code**

- **May use more space.**

Advantages

- **More natural.**
- **Easier to prove correct.**
- **Easier to analyze.**
- **More flexible.**

Recurrence Examples

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

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

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

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

Methods to solve recurrence

- **Iteration method**
- **Substitution method**
- **Recursion tree method**
- **Master Theorem**

Solving Recurrences "iteration method"

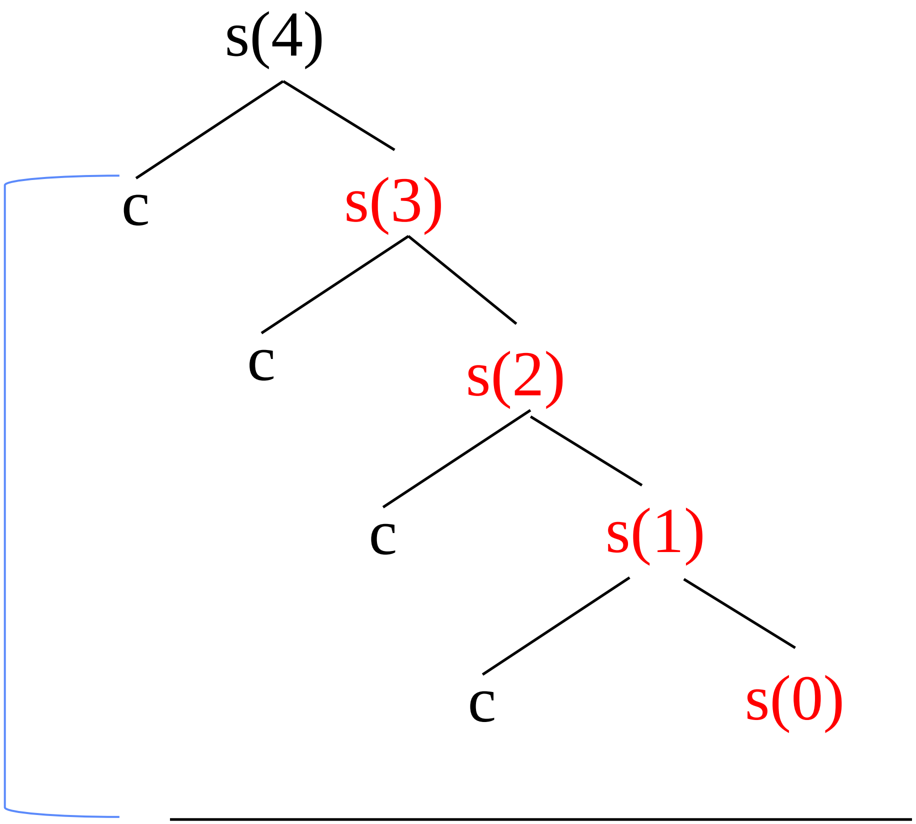
- **Work some algebra to express as a summation**
- **Evaluate the summation**

Example 1

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

$\boxed{s(n-1) = c + s(n-2)}$

$s(n-2) = c + s(n-3)$



=4c+s(0)

=4c+0

=4c

=nc = n

n=4

Example 1

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

$\boxed{s(n-1) = c + s(n-2)}$

s(n-2) = c + s(n-3)

- **s(4) =**

  c + s(4-1)

  c + c + s(4-2)

  2c + s(4-2)

  2c + c + s(4-3)

  3c + s(4-3)

  4c + s(4-4)

  = 4c + s(0)

=4c+0 [Stops here]

=nc

Example 1

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

- **s(n) =**

  c + s(n-1)

s(n-1) = c + s(n-2)

Example 1

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

- **s(n) =**

  c + s(n-1)

  c + c + s(n-2)

s(n-1) = c + s(n-2)

Example 1

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

- **s(n) =**

  c + s(n-1)

  c + c + s(n-2)

  2c + s(n-2)

s(n-1) = c + s(n-2)

s(n-2) = c + s(n-3)

Example 1

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

- **s(n) =**

  c + s(n-1)

  c + c + s(n-2)

  2c + s(n-2)

  2c + c + s(n-3)

$\boxed{s(n-1) = c + s(n-2)}$

s(n-2) = c + s(n-3)

Example 1

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

- **s(n) =**

  c + s(n-1)

  c + c + s(n-2)

  2c + s(n-2)

  2c + c + s(n-3)

  3c + s(n-3)

$\boxed{s(n-1) = c + s(n-2)}$

$\boxed{s(n-2) = c + s(n-3)}$

Example 1

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

- **s(n) =**

  c + s(n-1)

  c + c + s(n-2)

  2c + s(n-2)

  2c + c + s(n-3)

  3c + s(n-3)

  …

  kc + s(n-k) = ck + s(n-k)

s(n-1) = c + s(n-2)

s(n-2) = c + s(n-3)

Example 1

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

- **s(n) =**

  c + s(n-1)

  $\boxed{s(n-1) = c + s(n-2)}$

  2c + s(n-2)

  s(n-2) = c + s(n-3)

  3c + s(n-3)

  …

  kc + s(n-k) = ck + s(n-k)

Example 1

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

- s(n) =

  c + s(n-1)

  $\boxed{s(n-1) = c + s(n-2)}$

  2c + s(n-2)

  $\boxed{s(n-2) = c + s(n-3)}$

  3c + s(n-3)

  4c + s(n-4)

  …

  kc + s(n-k) = ck + s(n-k)

Example 1

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

- So far for **n >= k** we have
  - **s(n) = ck + s(n-k)**
- What if k = n?
  - **s(n) = cn + s(0) = cn**

Example 1

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

- So far for n >= k we have
  - s(n) = ck + s(n-k)
- What if k = n?
  - s(n) = cn + s(0) = cn
- So

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

- Thus in general
  - s(n) = cn

Example 2

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

- **s(n)**

Example 2

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

- **s(n)**

= n + s(n-1)

Example 2

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

- **s(n)**

= n + s(n-1)

= n + n-1 + s(n-2)

Example 2

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

- s(n)

= n + s(n-1)

= n + n-1 + s(n-2)

= n + n-1 + n-2 + s(n-3)

Example 2

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

- **s(n)**

= n + s(n-1)

= n + n-1 + s(n-2)

= n + n-1 + n-2 + s(n-3)

= n + n-1 + n-2 + n-3 + s(n-4)

Example 2

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

- s(n)

= n + s(n-1)

= n + n-1 + s(n-2)

= n + n-1 + n-2 + s(n-3)

= n + n-1 + n-2 + n-3 + s(n-4)

= …

= n + n-1 + n-2 + n-3 + … + n-(k-1) + s(n-k)

Example 2

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

- **s(5)**

= 5 + s(5-1) **=5+S(4)**

= 5 + 4 + s(5-2) **=5+4+S(3)**

= 5 + 4 + 3 + s(5-3) **= 5+4+3+S(2)**

= 5 + 4 + 3 + 2 + s(5-4) **=5+4+3+2+S(1)**

= 5 + 4 + 3 + 2 +1+ s(5-5) **=5+4+3+2+1+S(0)**

Example 2

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

- s(n)

= n + s(n-1)

= n + n-1 + s(n-2)

= n + n-1 + n-2 + s(n-3)

= n + n-1 + n-2 + n-3 + s(n-4)

= …

= n + n-1 + n-2 + n-3 + … + n-(k-1) + s(n-k)

Example 2

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

- **s(n)**

= n + s(n-1)

= n + n-1 + s(n-2)

= n + n-1 + n-2 + s(n-3)

= n + n-1 + n-2 + n-3 + s(n-4)

= …

= n + n-1 + n-2 + n-3 + … + n-(k-1) + s(n-k)

=

$$\sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

Example 2

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

- **So far for n >= k we have**

$$\sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

Example 2

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

- **So far for n >= k we have**

$$\sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

- **What if k = n?**

S(5)=5 + 4 + 3 + 2 +1+ s(5-5)
=5+4+3+2+1+S(0)

Example 2

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

- **So far for n >= k we have**

$$\sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

- **What if k = n?**

$$\sum_{i=1}^{n} i \quad + \quad s(0) \quad = \quad \sum_{i=1}^{n} i \quad + \quad 0 \quad = \quad n\frac{n+1}{2}$$

Example 2

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

- **So far for n >= k we have**

$$\sum_{i=n-k+1}^{n} i \quad + \quad s(n-k)$$

- **What if k = n?**

$$\sum_{i=1}^{n} i \quad + \quad s(0) \quad = \quad \sum_{i=1}^{n} i \quad + \quad 0 \quad = \quad n\frac{n+1}{2}$$

- **Thus in general**

$$s(n) \quad = \quad n\frac{n+1}{2}$$

Example 3

Home Activity

- *Solve the following Recurrence using iteration Method*

- $T(1) = 1$
  $T(n) = 2\,T(n/2) + n$

Note: Before looking at the solution, do it by yourself

Example 3

Home Activity

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

$$= 2(2T(n/4) + n/2) + n$$
$$= 4T(n/4) + n + n$$
$$= 4(2T(n/8) + n/4) + n + n$$
$$= 8T(n/8) + n + n + n$$
$$= nT(n/n) + n + \ldots + n + n + n$$
$$= n + n + \ldots + n + n + n$$

Example 3

Home Activity

- $T(n) = 2T(n/2) + n$
  $= 2(2T(n/4) + n/2) + n$
  $= 4T(n/4) + n + n$
  $= 4(2T(n/8) + n/4) + n + n$
  $= 8T(n/8) + n + n + n$
  $= nT(n/n) + n + ... + n + n + n$
  $= n + n + ... + n + n + n$

Counting the number of repetitions of n in the sum at the end, we see that there are $\lg n + 1$ of them. Thus the running time is $n(\lg n + 1) = n \lg n + n$. We observe that $n \lg n + n < n \lg n + n \lg n = 2n \lg n$ for $n>0$, so the running time is $O(n \lg n)$.

Reference: https://www.cs.cornell.edu/courses/cs3110/2014sp/recitations/21/solving-recurrences.html

Divide and Conquer

The divide-and-conquer paradigm

What is a paradigm? One that serves as a pattern or model.

- Divide **the problem into a number of subproblems**
- Conquer **the subproblems by solving them recursively. If small enough, just solve directly without recursion**
- Combine **the solutions of the subproblems into the solution for the original problem**

Analyzing Divide-and-Conquer Algorithms

Let T(n) be the running time on a problem of size n.

If problem is small enough, then solution takes constant time (this is a boundary condition, which will be assumed for all analyses)

It takes time to divide the problem into sub-problems at the beginning. Denote this work by D(n).

Analyzing Divide-and-Conquer Algorithms

It takes time to combine the solutions at the end. Denote this by C(n).

If we divide the problem into 'a' problems, each of which is '1/b' times the original size (n), and since the time to solve a problem with input size 'n/b' is T(n/b), and this is done 'a' times, the total time is:

$$T(n) = \Theta(1),\ n \text{ small (or } n \le c\text{)}$$
$$T(n) = aT(n/b) + D(n) + C(n)$$

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

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

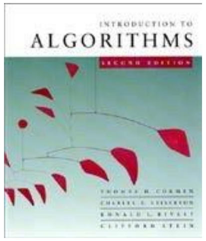
number of sub-problems

Sub-problem size

work done dividing and combining

Recursion tree

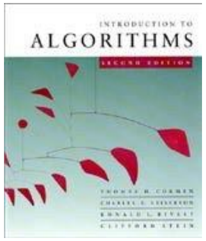
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



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.

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

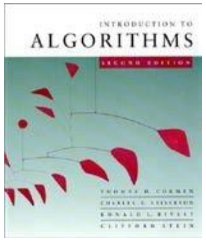
Recursion tree

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

$$cn$$

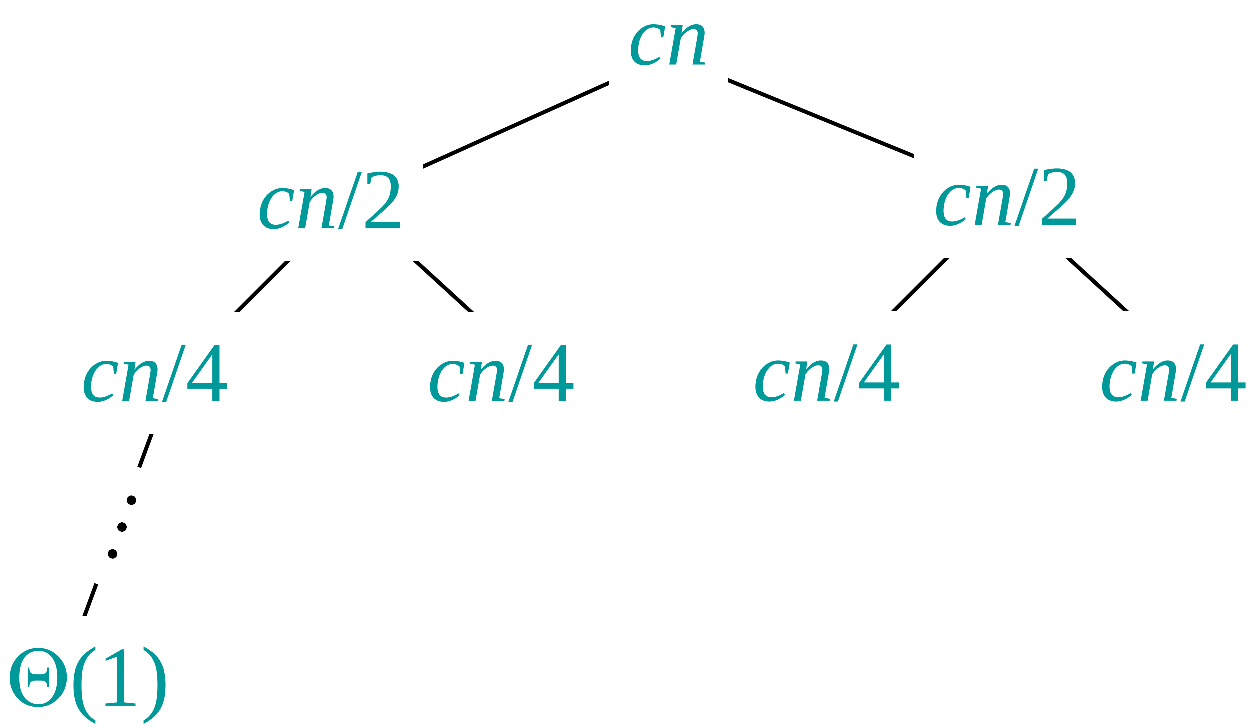
$$cn/2 \qquad\qquad cn/2$$

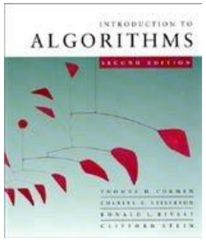
$$T(n/4) \qquad T(n/4) \qquad T(n/4) \qquad T(n/4)$$



Recursion tree

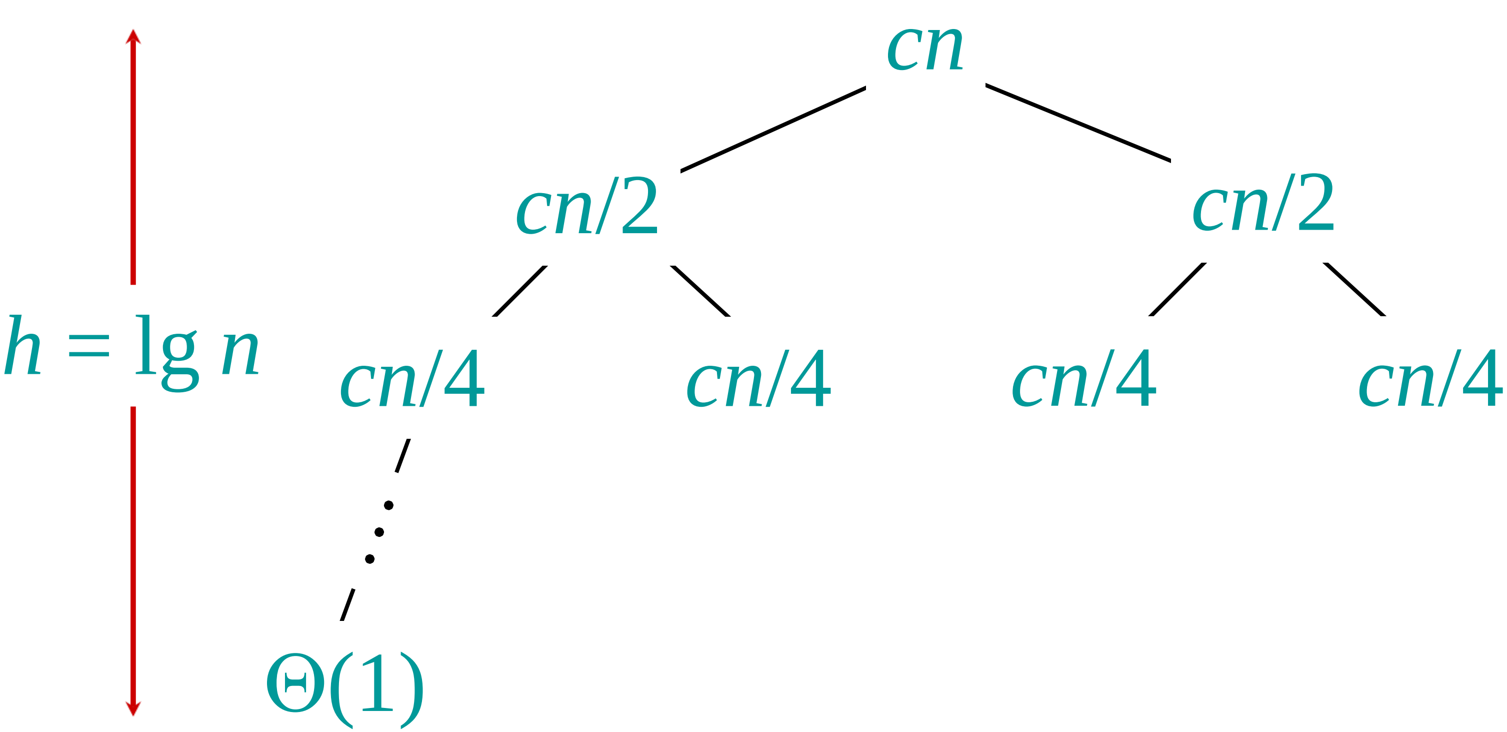
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

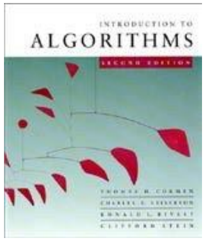




Recursion tree

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 $\quad\quad$ cn

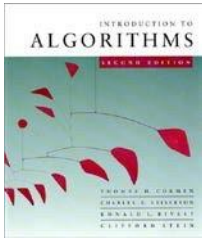
$cn/2 \quad\quad cn/2$

$cn/4 \quad cn/4 \quad cn/4 \quad cn/4$

$h = \lg n$

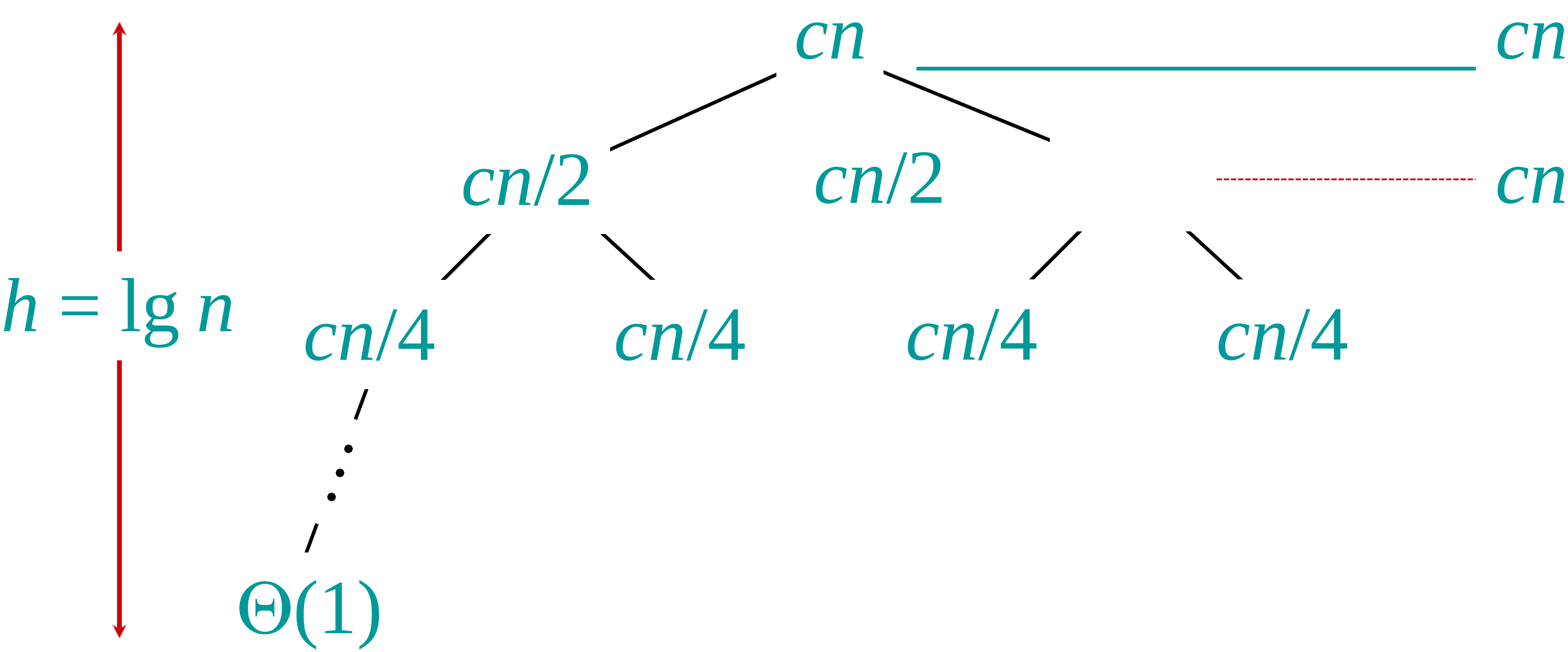
$\vdots$

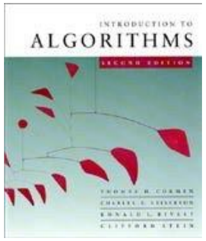
$\Theta(1)$



Recursion tree

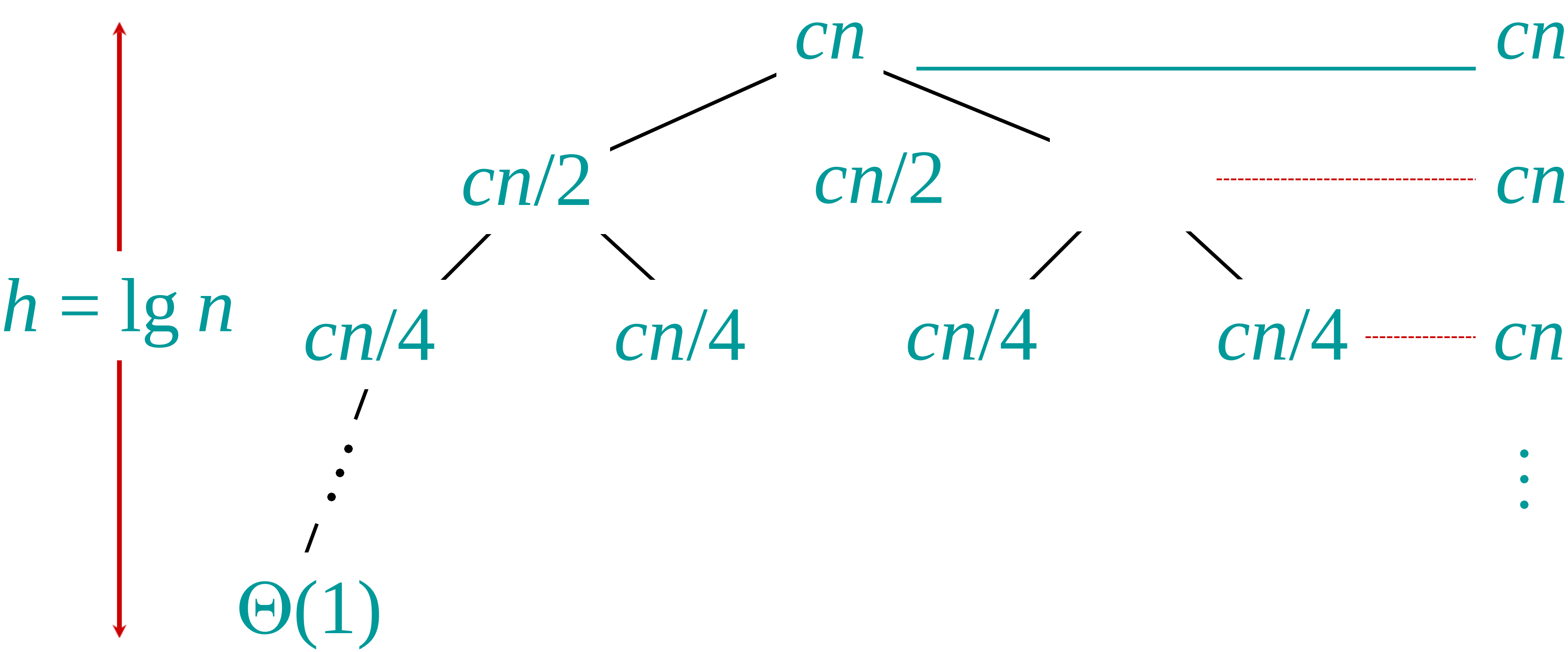
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

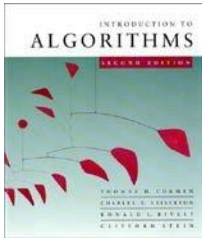




Recursion tree

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





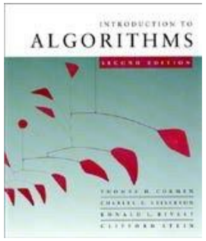
Recursion tree

$n=8$

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

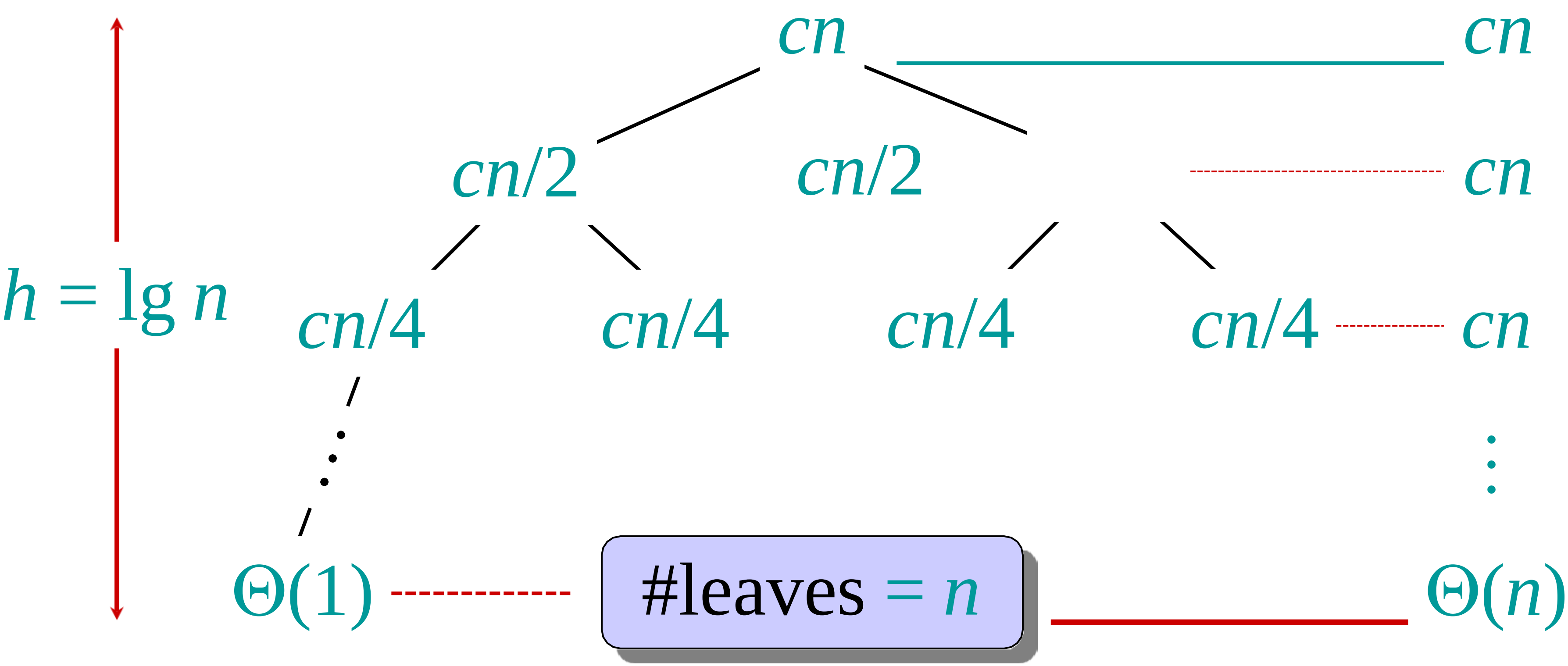
$$
\begin{array}{ccccccc}
 & & & 8c & & & cn \\
 & 4c & & & & 4c & cn \\
2c & & 2c & & 2c & 2c & cn \\
\vdots & & & & & & \vdots \\
\Theta(1) & & & \#\text{leaves} = n & & & \Theta(n)
\end{array}
$$

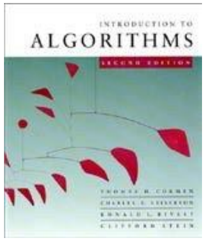
$h = \lg n$



Recursion tree

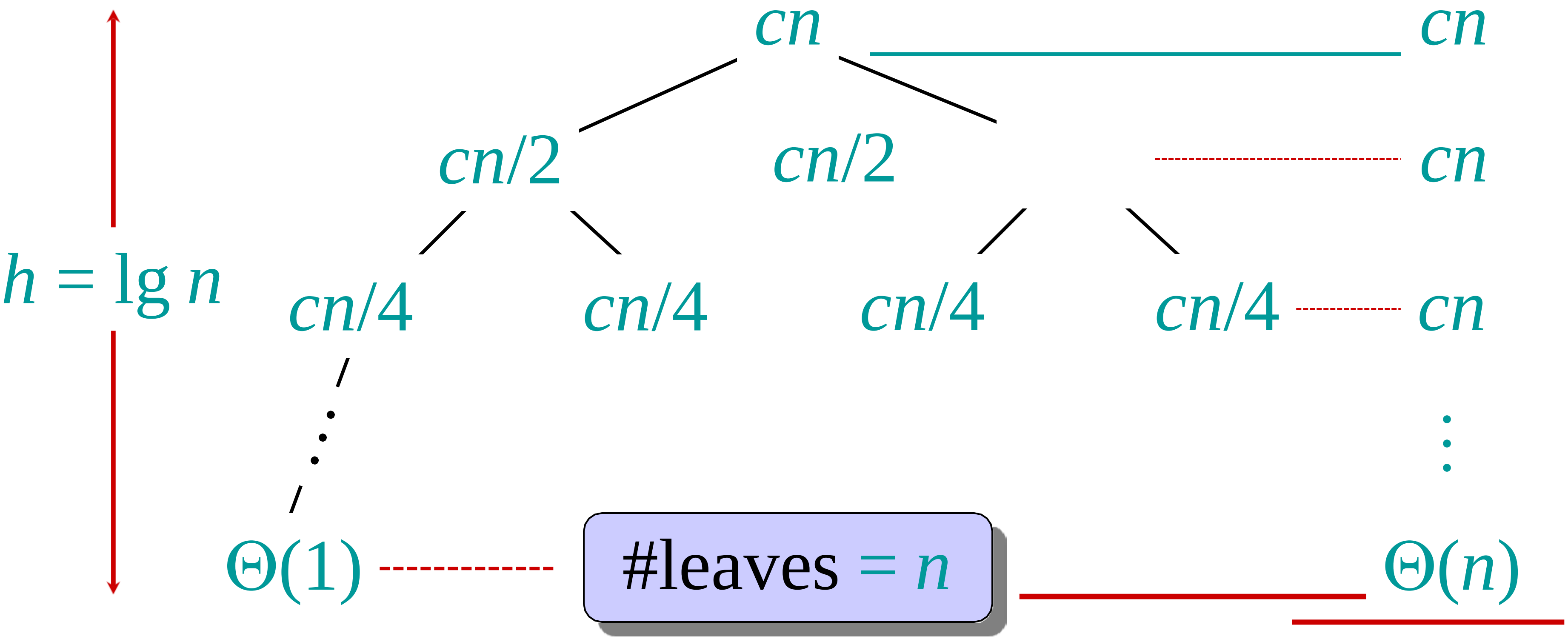
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.





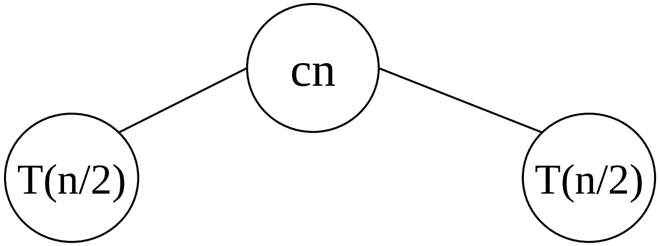
Recursion tree

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

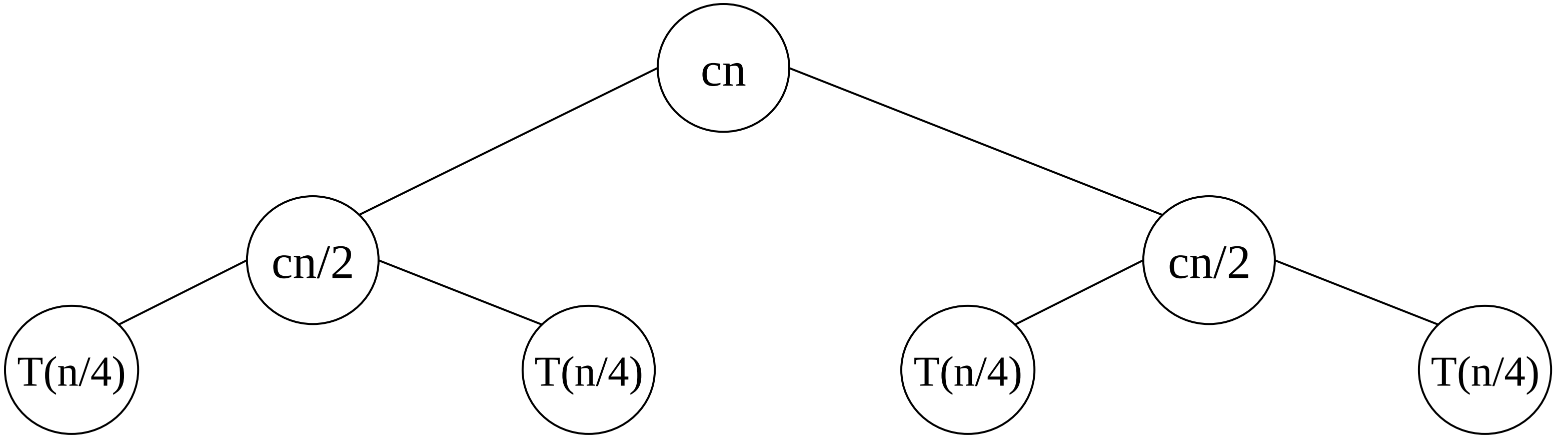


Total $= \Theta(n \lg n)$

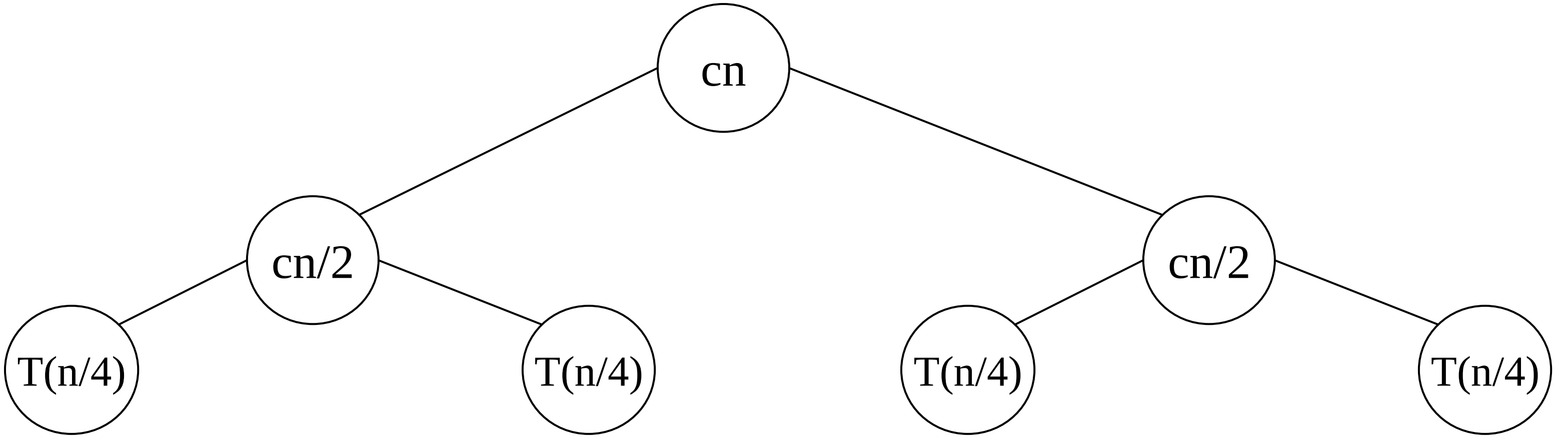
Recursion Tree for Algorithm



Recursion Tree for Algorithm

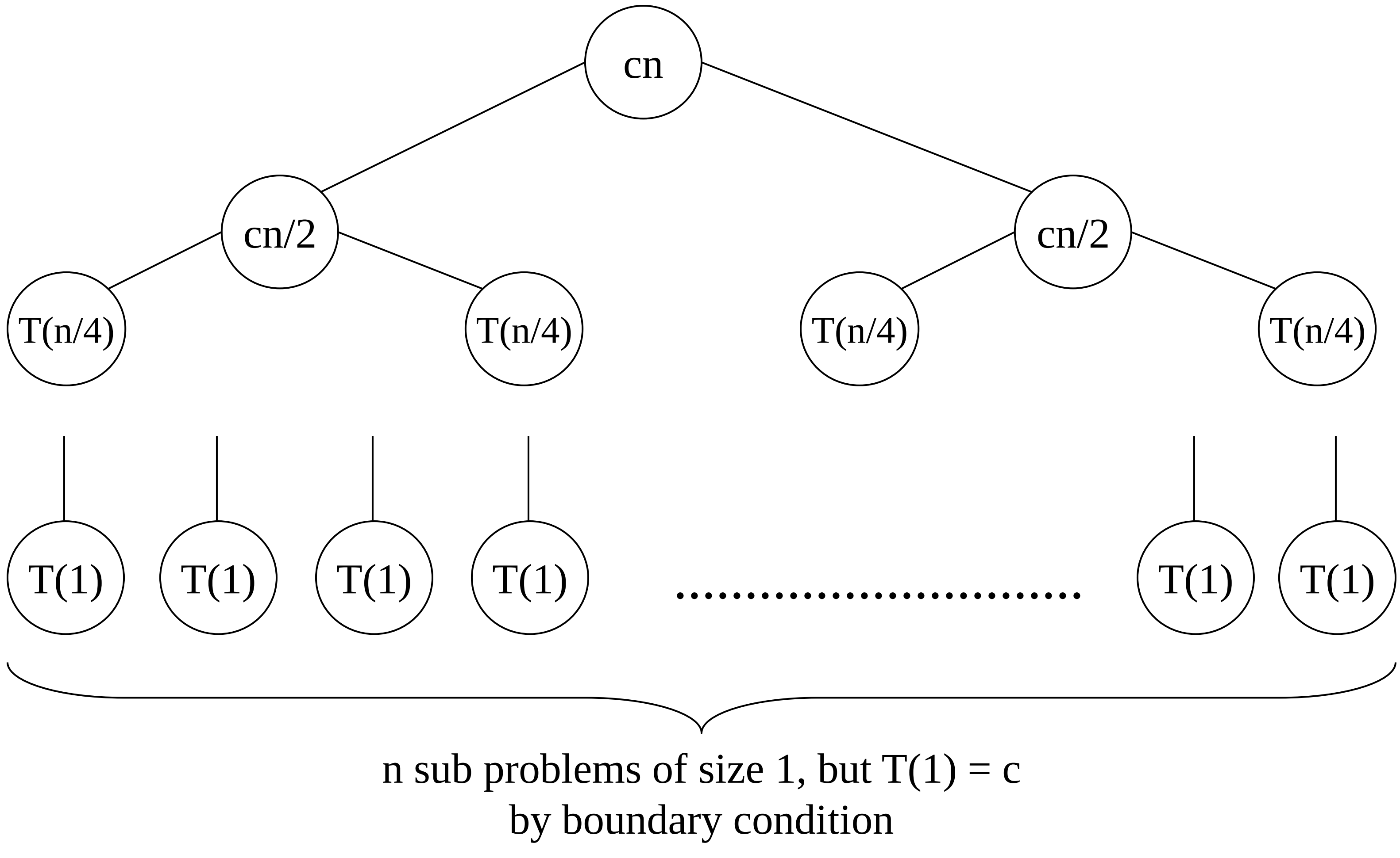


Recursion Tree for Algorithm

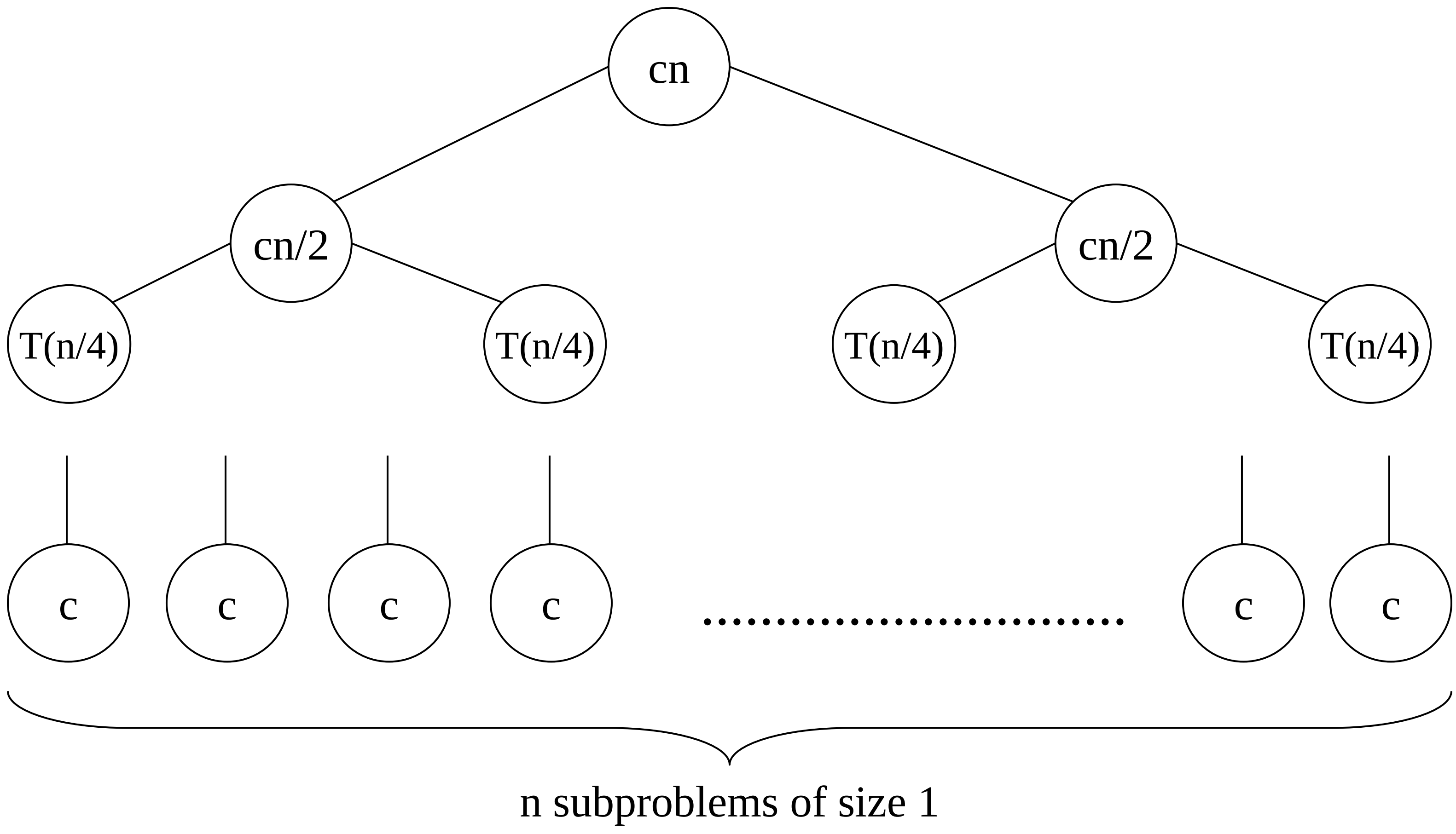


Eventually, the input size (the argument of T) goes to 1, so...

Recursion Tree for Algorithm

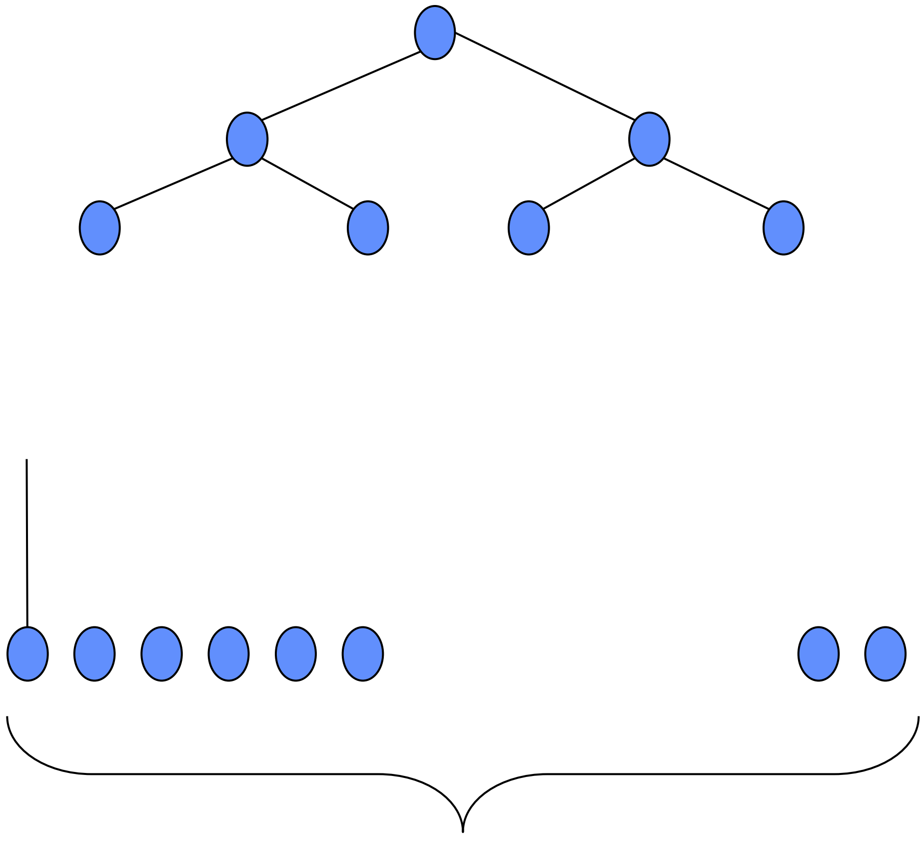


Recursion Tree for Algorithm



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



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

$N-1 = \lg(n)$

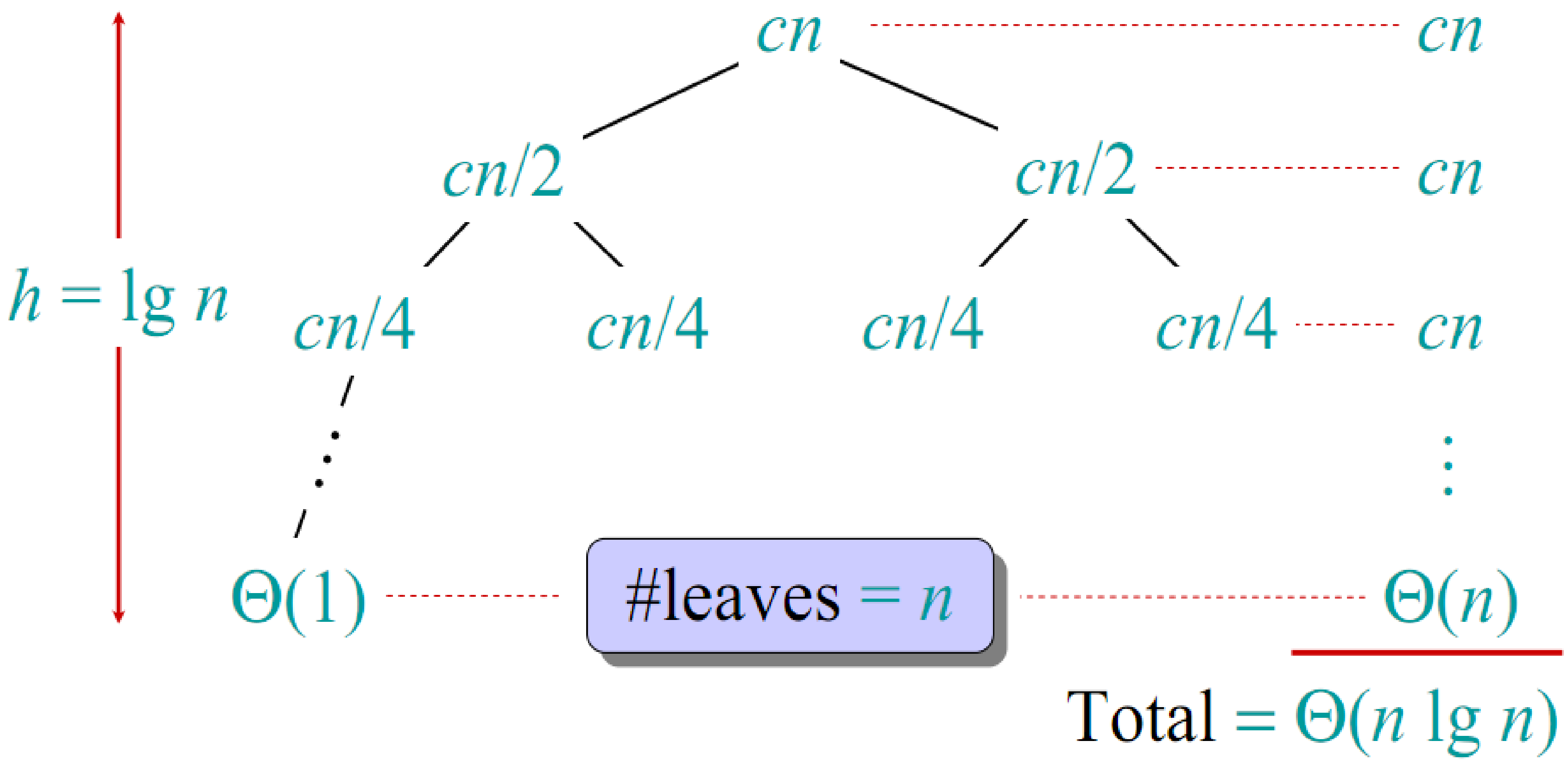
levels = N = 1+lg(n)

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:**
    - **$T(n) = 2T(n/2) + n$ $\boxed{?}$ $T(n) = O(n \lg n)$**

- https://opendsa-server.cs.vt.edu/embed/mergesortAV
- https://www.youtube.com/watch?v=4V30R3l1vLI&ab_channel=AbdulBari

Reference

- **Introduction to Algorithms**
- **Chapter # 4**
  - **Thomas H. Cormen**
  - **3rd Edition**