Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Methods for solving recurrences

There are three main methods for solving recurrences:

1. Substitution Method
2. Recursion Tree
3. Master Method

Substitution method

- Substitution method has two basic steps

1. Substitute a guess for $T(n)$
2. Prove your guess using **strong induction**

Substitution method

- Substitution method has two basic steps

1. Substitute a guess for $T(n)$
2. Prove your guess using **strong induction**

Example: $T(n) = 4\,T\left(\frac{n}{4}\right) + n$, for simplicty say $T(1) = 0$

Guess: $T(n) = O(n \log n)\ \rightarrow T(n) \leq\ c\, n \log n$, prove it.

Substitution method

- Substitution method has two basic steps

1. Substitute a guess for $T(n)$
2. Prove your guess using **strong induction**

Example: $T(n) = 4\,T\left(\frac{n}{4}\right) + n$, for simplicty say $T(1) = 0$

Guess: $T(n) = O(n \log n) \rightarrow T(n) \leq c\, n \log n$, prove it.

How to show that $T(n)$ is $\Theta(n \log n)$? We should also prove $T(n) = \Omega(n \log n)$

Substitution method

- Unfortunately, to show that $T(n) = \Omega(n \log n\ )$ showing the following doesn't work

$$T(n) \geq cn \log n$$

Substitution method

- Unfortunately, to show that $T(n) = \Omega(n \log n\ )$ showing the following doesn't work

$$T(n) \geq cn \log n$$

- We have to show the exact form which is

$$T(n) \geq c_1 n \log n + c_2 n$$

- This is one of the drawbacks with this method

Substitution method

- Also, note that you can never use asymptotic notations in an inductive proof.

Substitution method

- Also, note that you can never use asymptotic notations in an inductive proof.
- Say we want to prove that $\boldsymbol{n = O(1)}$ which is obviously incorrect.

- For $n = 1$, we have $1 = O(1)$
- If for $n = k$, we $k = O(1)$
- Then for $n = k + 1 = O(1) + 1 = O(1)$

Substitution method

- This happens because **we can only use big-O for a constant number of times**.

- But if we try to use the big-O notation many times (non-constant) then the hidden constants inside of big-O don't remain constant anymore when they add up.

Substitution method

- The first disadvantage is that you have to make a **good guess**

Substitution method

- The first disadvantage is that you have to make a **good guess**
- Some tips in making a good guess
  - Come up with a trivial guess and gradually improve it
  - Try to get some ideas from the next two methods :)

Substitution method

- The first disadvantage is that you have to make a **good guess**
- Some tips in making a good guess
  - Come up with a trivial guess and gradually improve it
  - Try to get some ideas from the next two methods :)
- Another disadvantage is that there are many subtleties and pitfalls for this method (you can see them on CLRS page 84-87, these pages are optional)

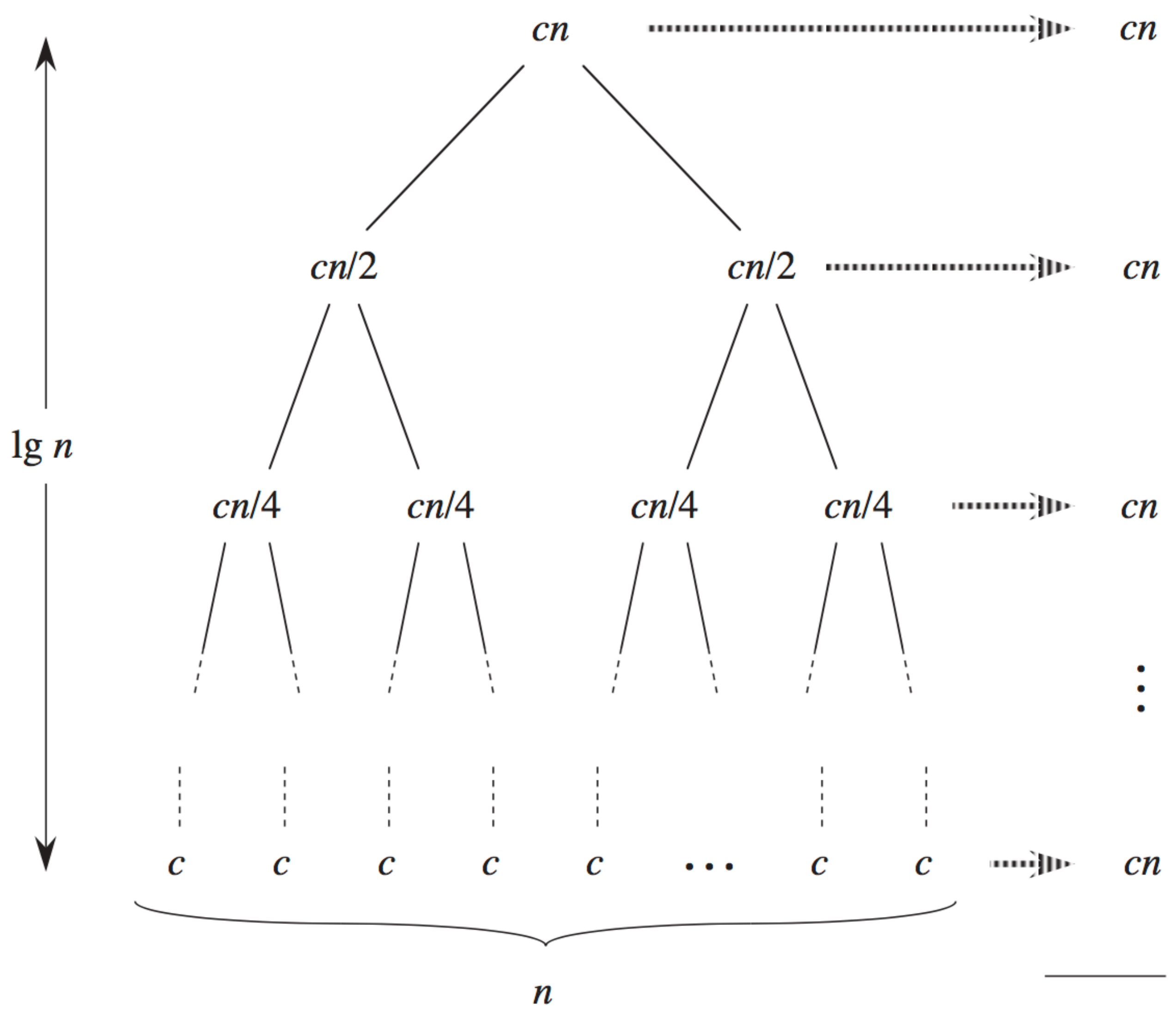
Substitution method

- The first disadvantage is that you have to make a **good guess**
- Some tips in making a good guess
  - Come up with a trivial guess and gradually improve it
  - Try to get some ideas from the next two methods :)
- Another disadvantage is that there are many subtleties and pitfalls for this method (you can see them on CLRS page 84-87, these pages are optional)
- **Note:** If I ask you to solve a recurrence using this method I will ask you simple ones and provide you with the exact form.

Recursion tree method

- It's the **most intuitive** way to solve a recurrence

- We want to show the cost of the recursive algorithm as a tree
- The cost of each recursive call is reflected in a node



How to make a recursion tree?

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

I can write $\boldsymbol{\Theta}(\boldsymbol{n})$ as $\boldsymbol{cn}$

$T(n)$

How to make a recursion tree?

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

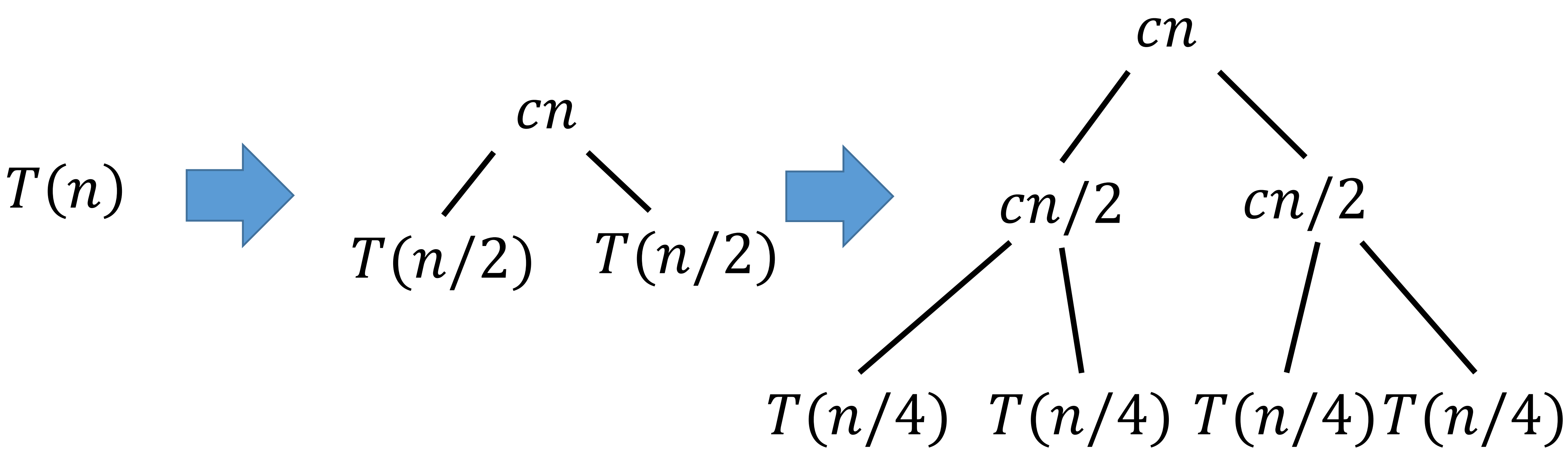
I can write $\boldsymbol{\Theta(n)}$ as $\boldsymbol{cn}$

$$T(n) \quad \Rightarrow \quad \begin{array}{c} cn \\ \diagup \qquad \diagdown \\ T(n/2) \quad T(n/2) \end{array}$$

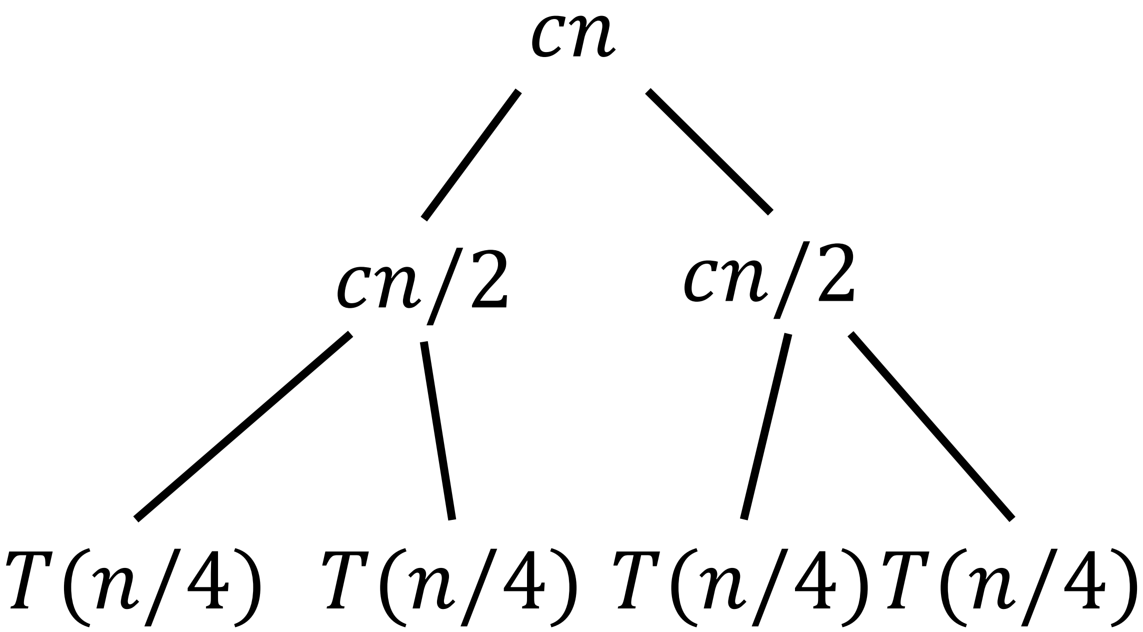
How to make a recursion tree?

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

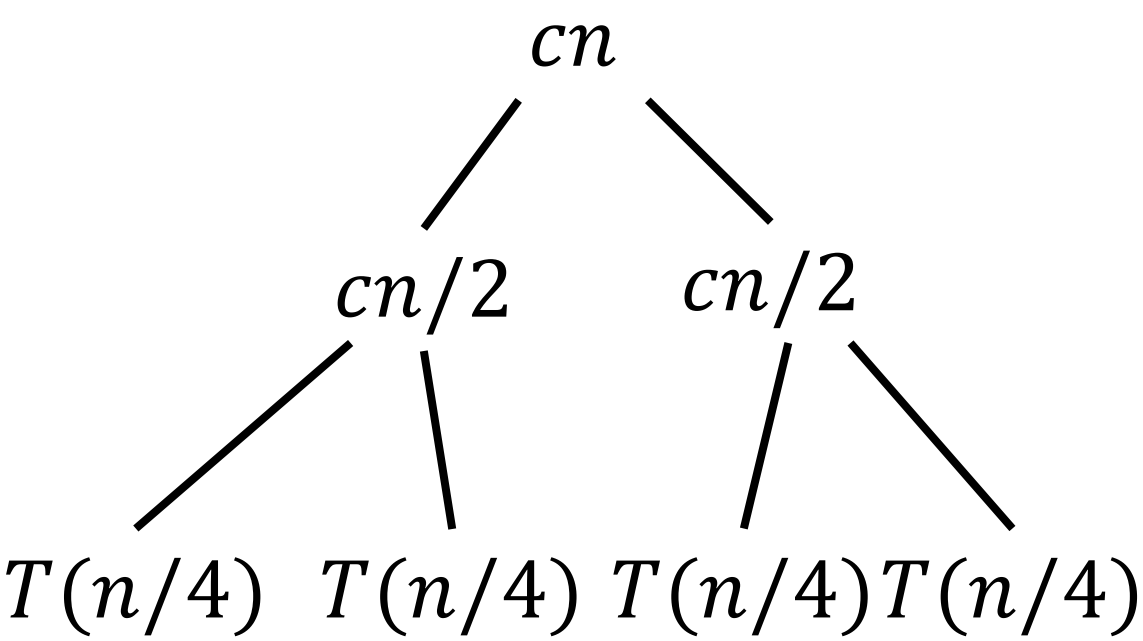
I can write $\boldsymbol{\Theta(n)}$ as $\boldsymbol{cn}$



How to make a recursion tree?

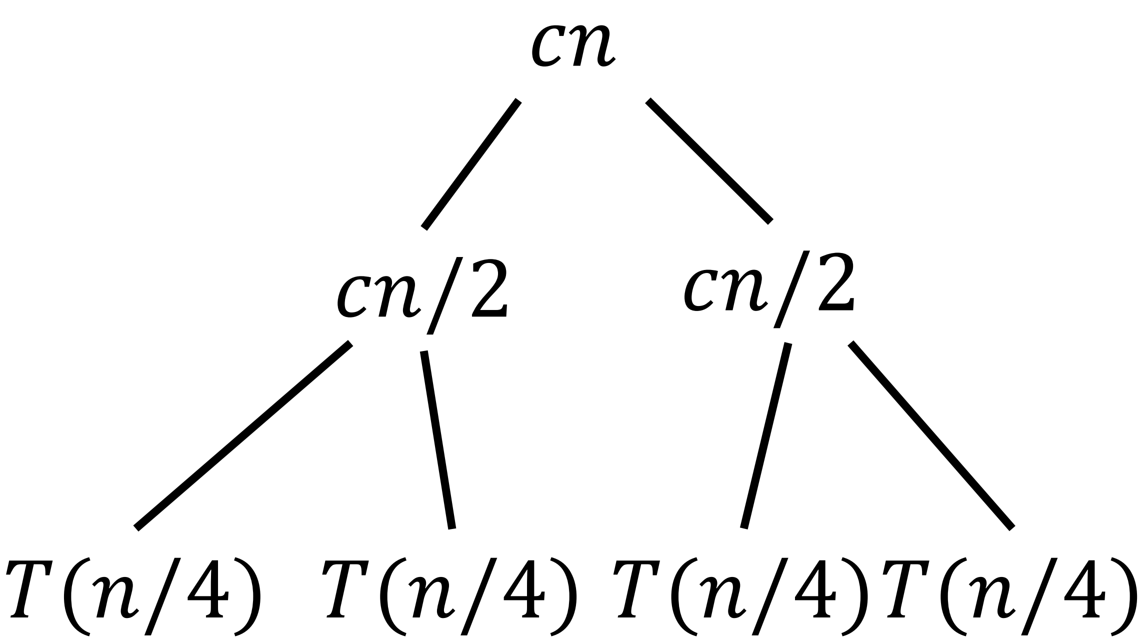


How to make a recursion tree?



Question: Why do I have to put the exact number? For example, why don't I simply put $\Theta(n)$ instead of cn or cn/2 in the nodes?

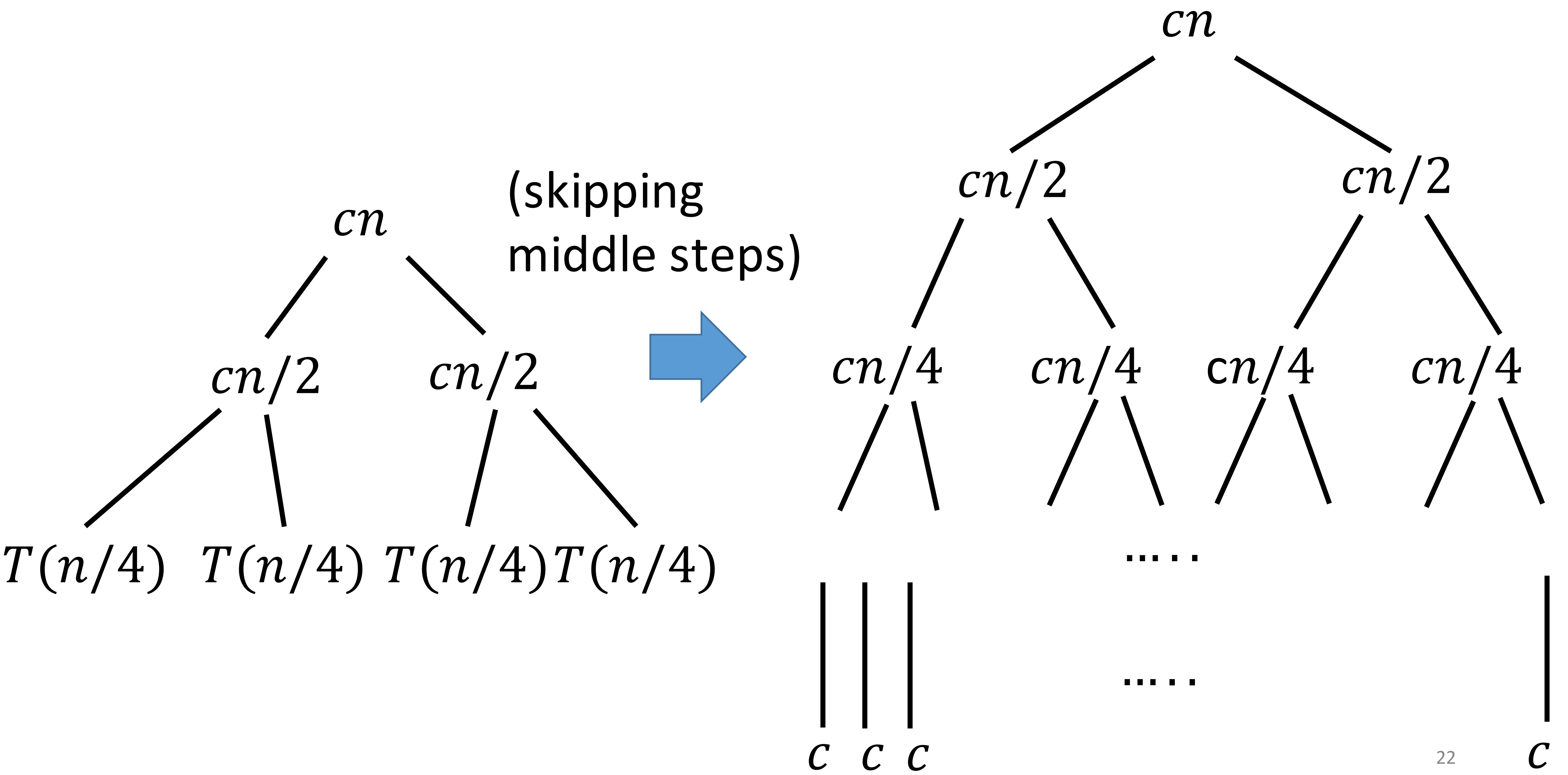
How to make a recursion tree?

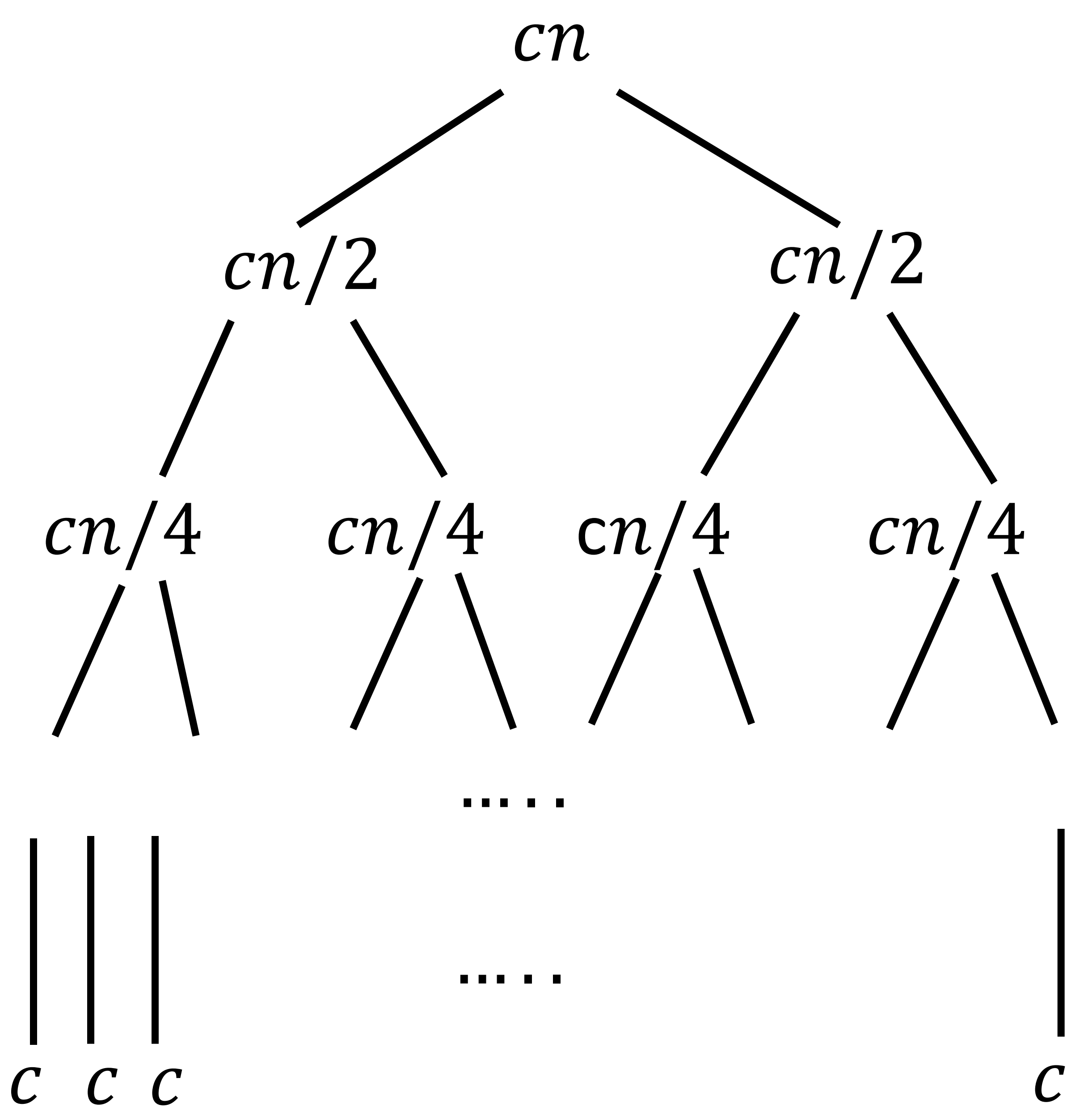


Question: Why do I have to put the exact number? For example, why don't I simply put $\Theta(n)$ instead of cn or cn/2 in the nodes?

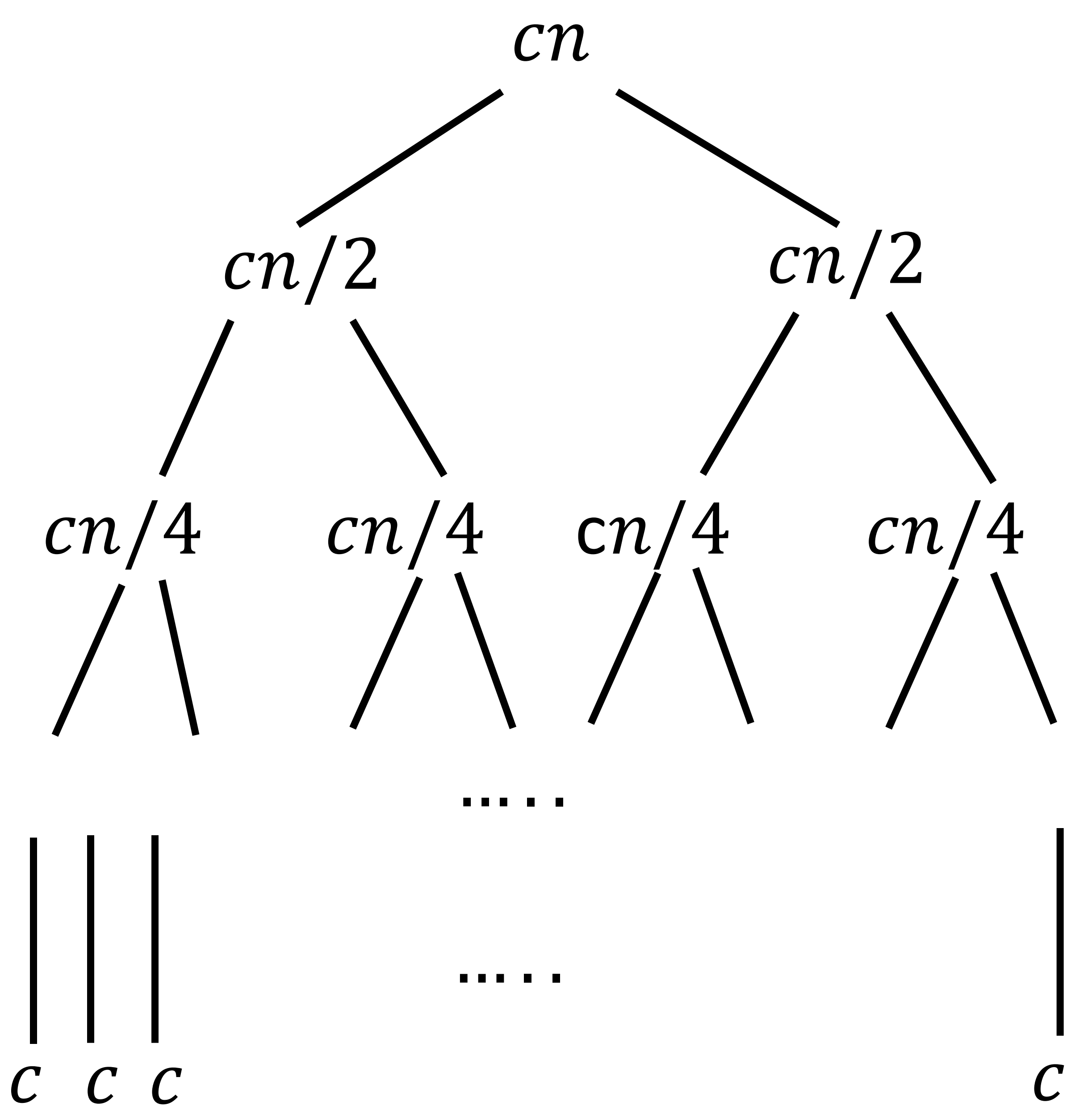
Answer: Because we can't use asymptotic notations for a non-constant number of times. And the exact constants will matter in the end.

How to make a recursion tree?

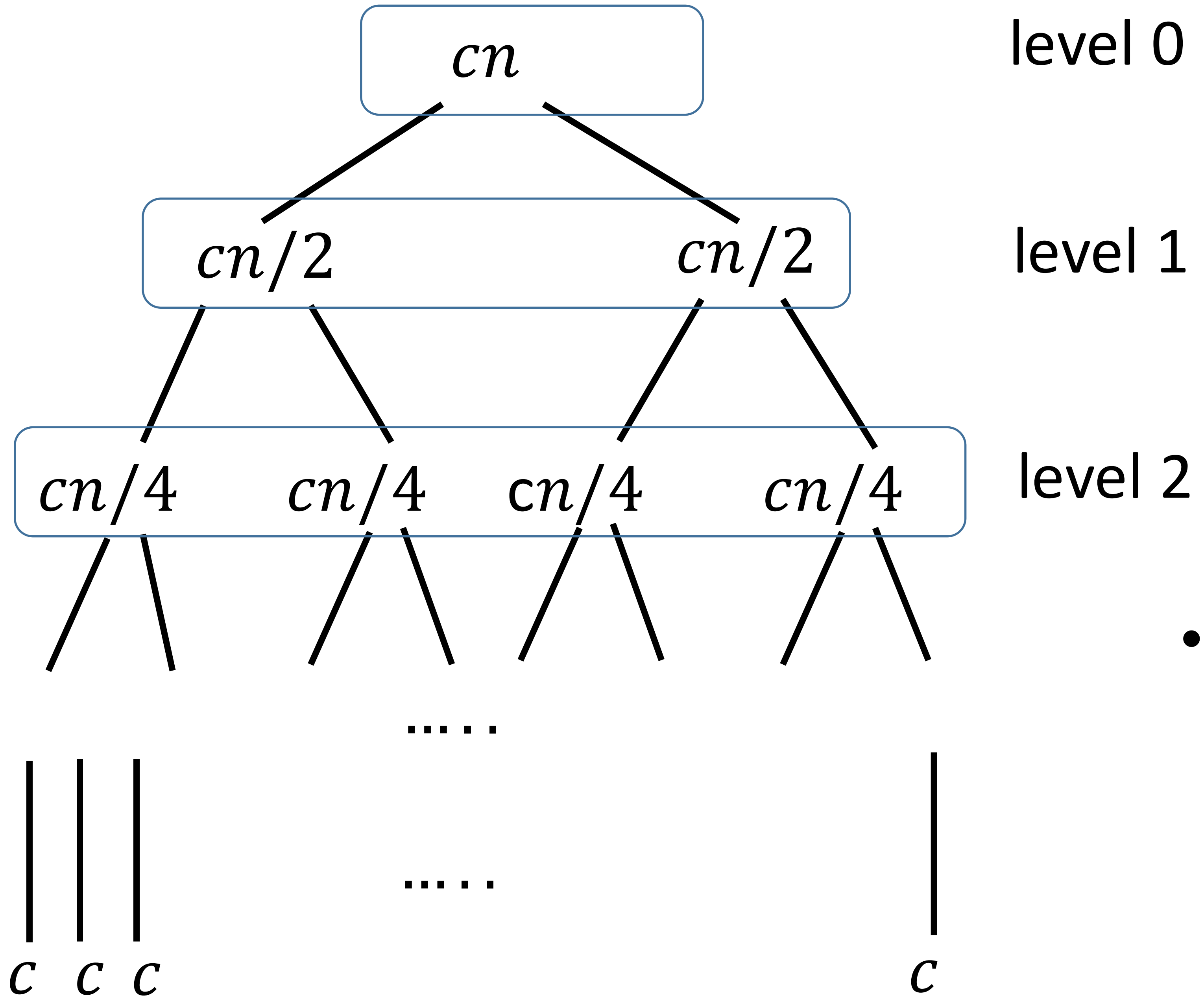




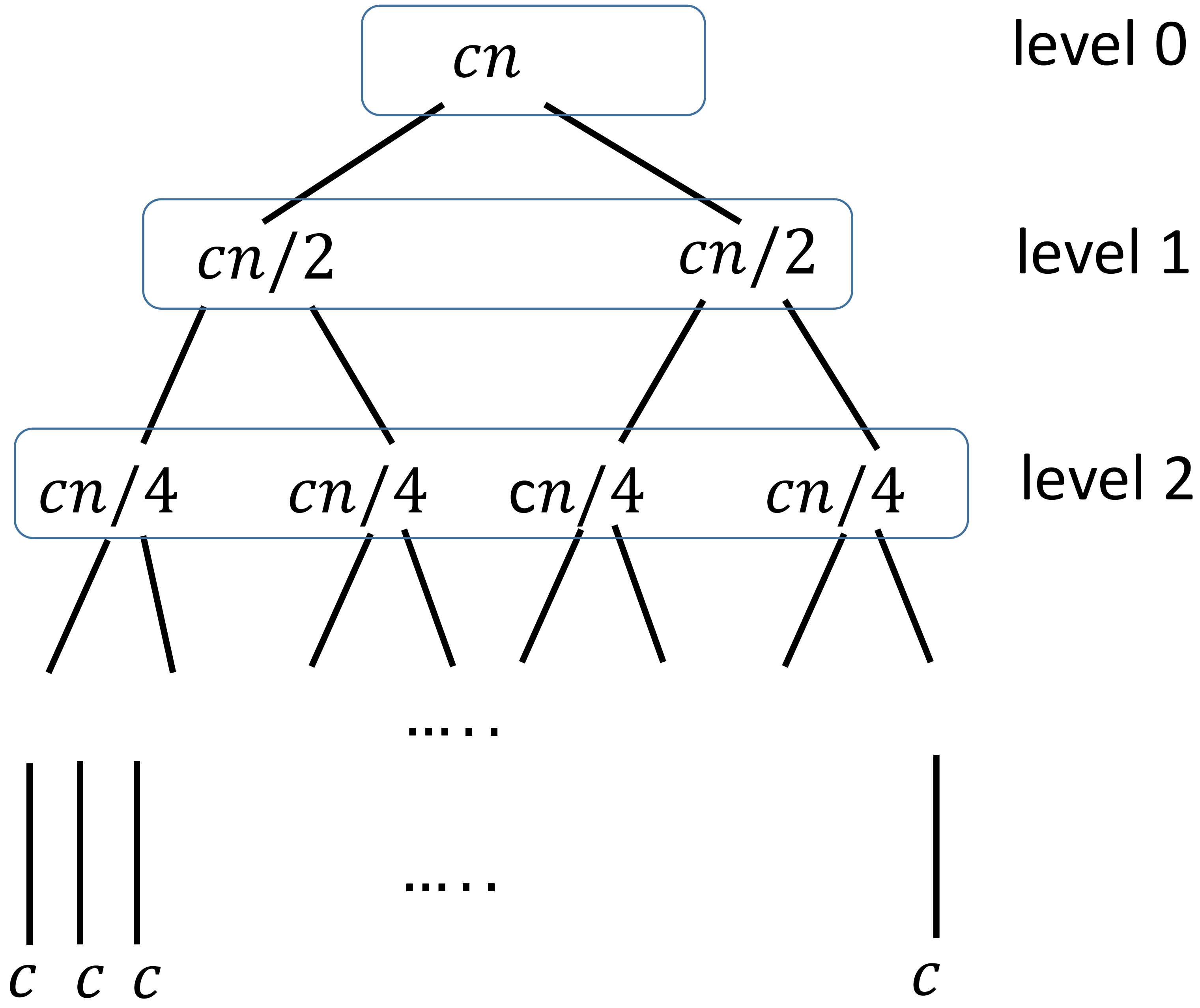
cn
$cn/2$ $cn/2$
$cn/4$ $cn/4$ $cn/4$ $cn/4$
.....
.....
c c c c



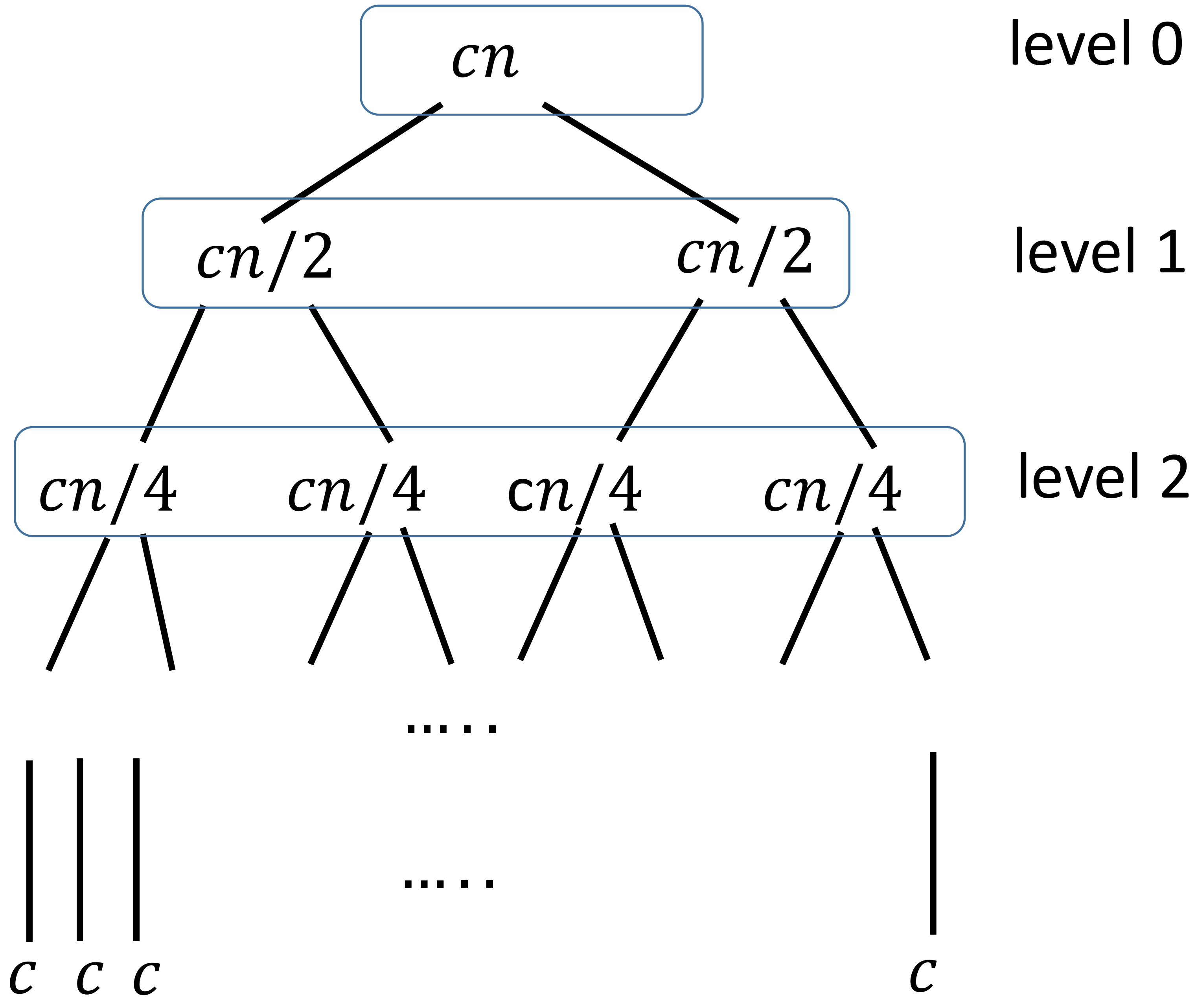
- Such a structure is called a **binary tree**
- It’s called binary because each node has 2 nodes below it



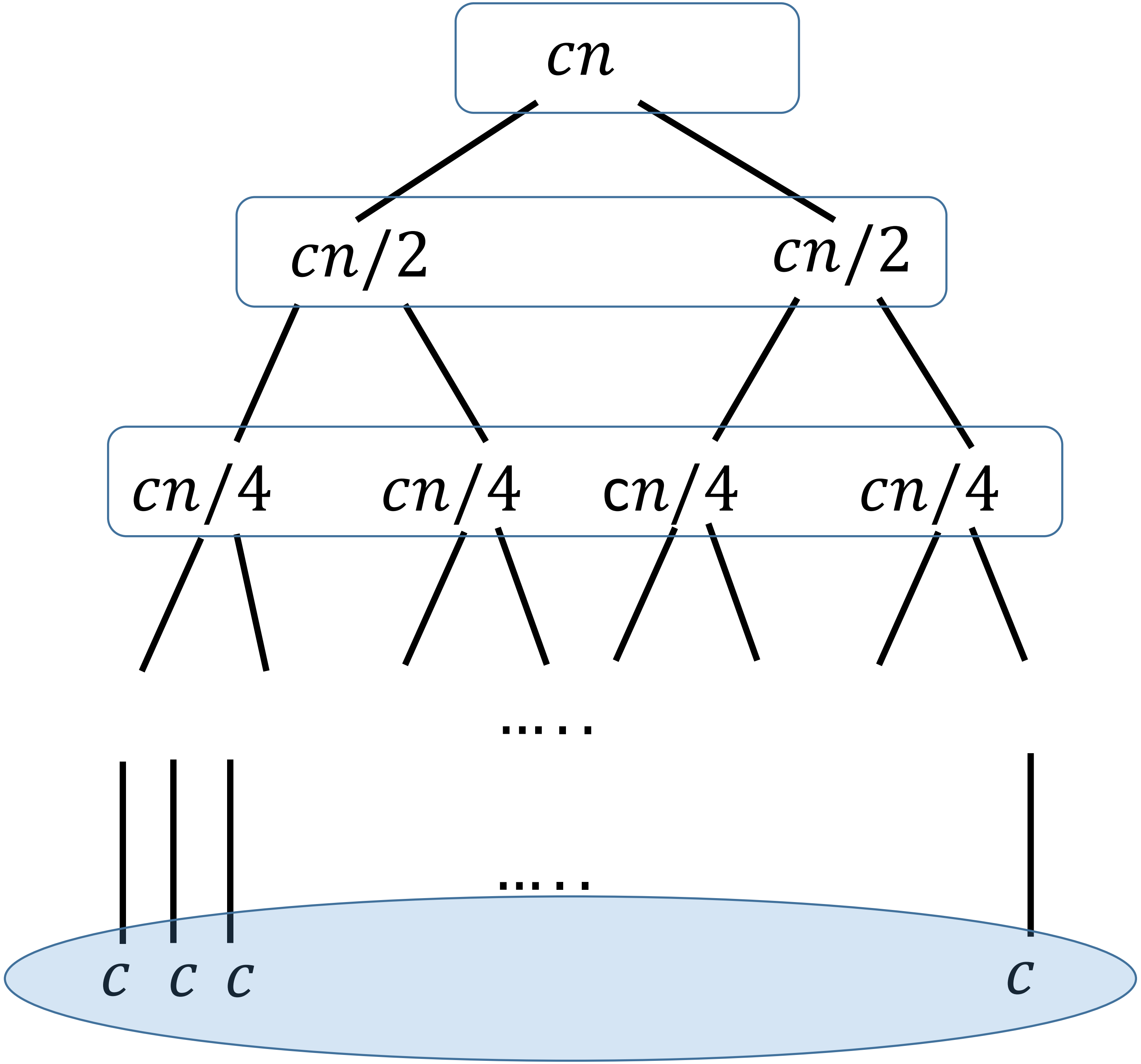
- Each row of nodes is called a **level**



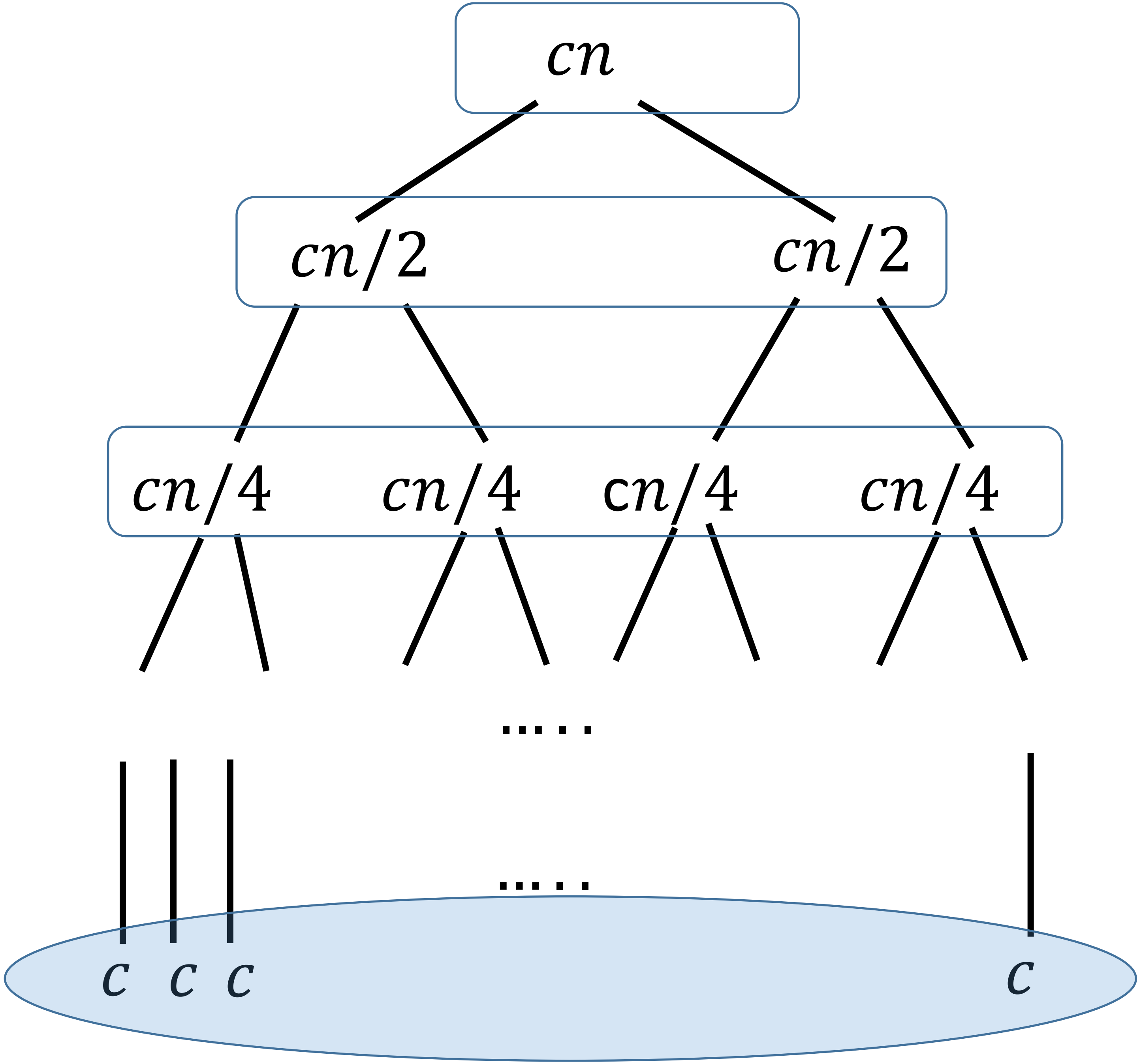
- **Question:** How many levels does this tree have?



- **Question:** How many levels does this tree have?
- **Answer:** $\log n$ (base 2) because each time we divide n by 2 until we get to 1



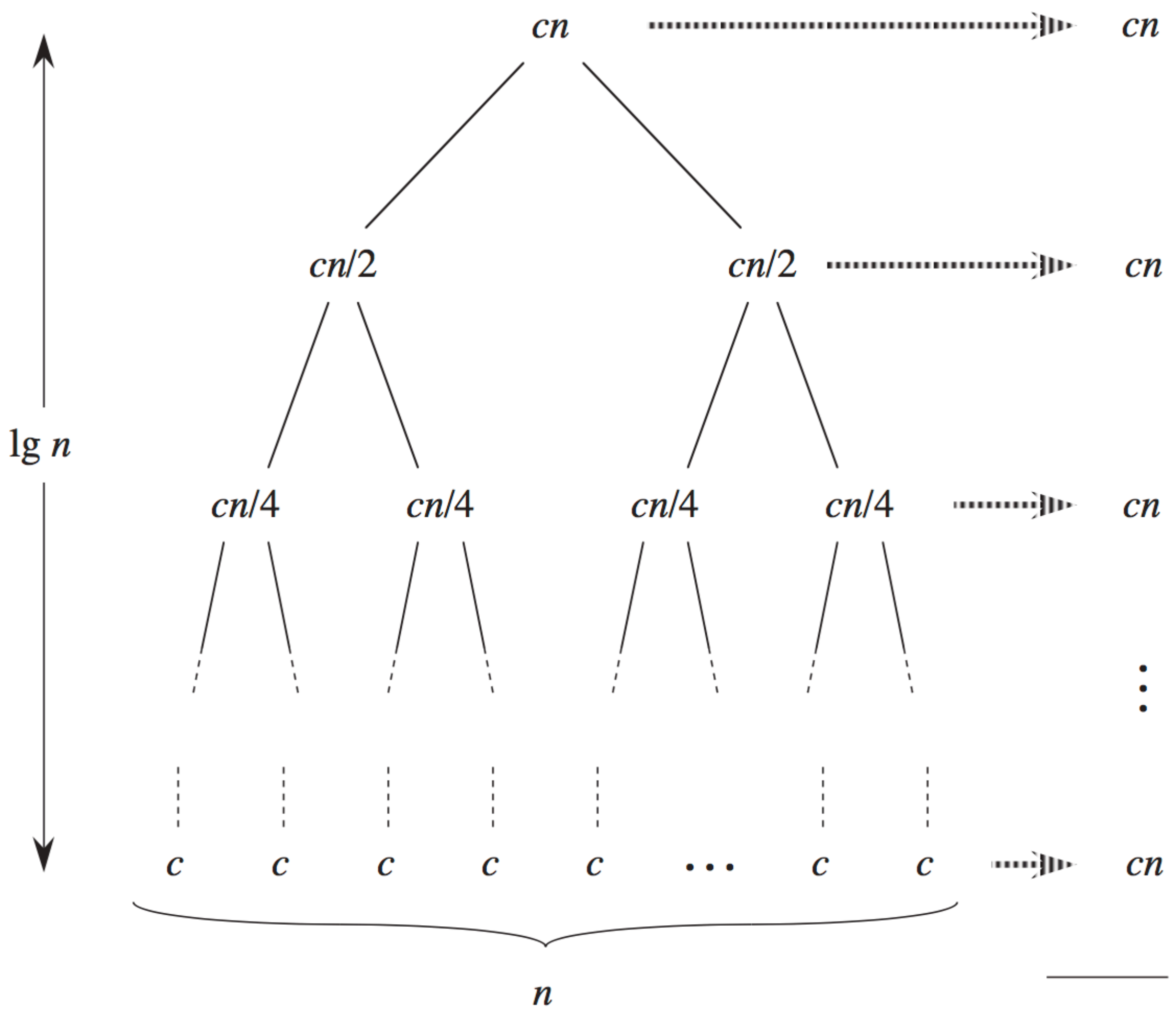
- **Question:** How many nodes are there at the lowest level?



- **Question:** How many nodes are there at the lowest level?
- **Answer:** $\mathbf{2}^{log\ n} = \boldsymbol{n}$**,** since we have 2^i nodes at level i

How to make a recursion tree?

- We have $\log n$ levels
- We take cn time at each level
- So, the overall time is $T(n) = cn \log n = \Theta(n \log n)$



total is $\Theta(n \log n)$

How to make a recursion tree?

- Start with one node which is $T(n)$
- Then, at each step replace a node with **the cost of operations on that node** and make **new nodes for the recursive parts**
- Stop when n reaches the base case, in this example $n = 1$. At this point there is no recursive call so you cannot grow the tree anymore
- At the bottom of the tree there are nodes that take $\Theta(1)$ or just a constant c number of operations

How to compute $T(n)$

- Each node of the recursion tree tells you how much time was spent on that node
- So, to get the running time of the whole recursive algorithm, we have to sum all values on all nodes
- To do this, it's easier to first compute the cost at each level first
- Then, determine how many levels we will have, and compute **number of levels** $\times$ **cost at each level**

A simple example

$$T(n) = \begin{cases} O(1) & n \le 1 \\ T(n-1) + O(1) & n > 1 \end{cases}$$ we can write $\boldsymbol{O(1)}$ as $\boldsymbol{c}$

$$T(n)$$

A simple example

$$T(n) = \begin{cases} O(1) & n \le 1 \\ T(n-1) + O(1) & n > 1 \end{cases}$$ we can write $\boldsymbol{O(1)}$ as $\boldsymbol{c}$

$T(n)$ ⇒

$$\begin{array}{c} c \\ | \\ T(n-1) \end{array}$$

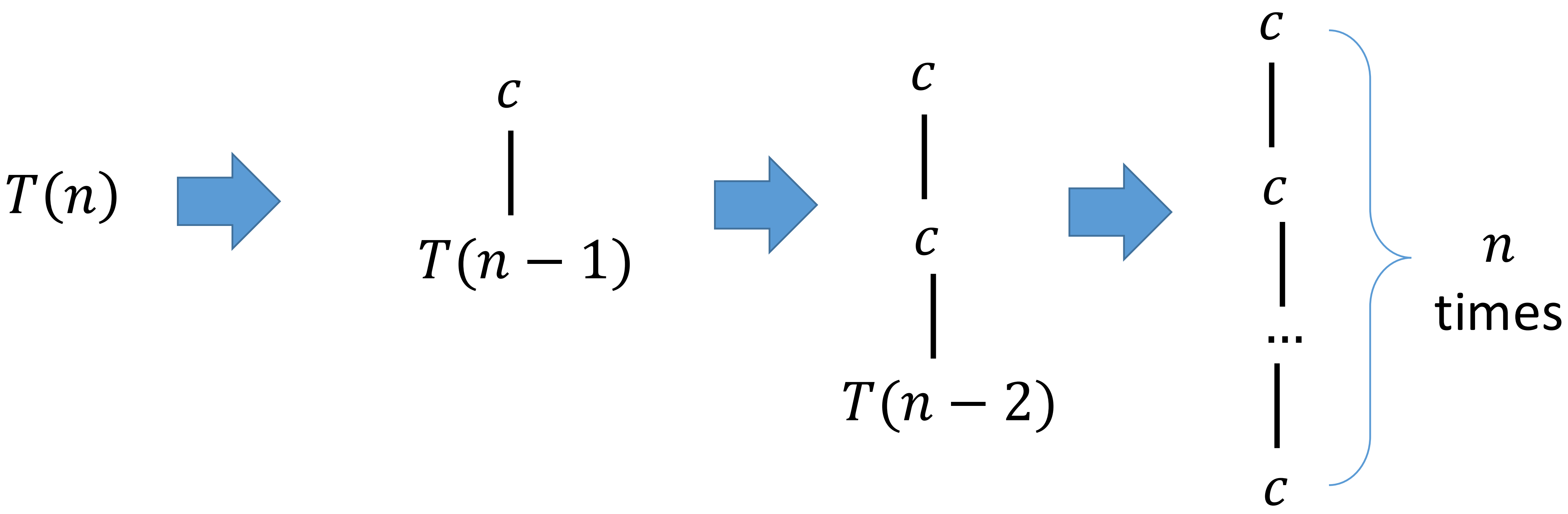
A simple example

$$T(n) = \begin{cases} O(1) & n \le 1 \\ T(n-1) + O(1) & n > 1 \end{cases}$$ we can write $\boldsymbol{O(1)}$ as $\boldsymbol{c}$

$$T(n) \Rightarrow \begin{array}{c} c \\ | \\ T(n-1) \end{array} \Rightarrow \begin{array}{c} c \\ | \\ c \\ | \\ T(n-2) \end{array}$$

A simple example

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ T(n-1) + O(1) & n > 1 \end{cases}$$ we can write $\boldsymbol{O(1)}$ as $\boldsymbol{c}$



A simple example

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ T(n-1) + O(1) & n > 1 \end{cases}$$

we can write $\boldsymbol{O(1)}$ as $\boldsymbol{c}$

$$T(n) \Rightarrow \begin{array}{c} c \\ | \\ T(n-1) \end{array} \Rightarrow \begin{array}{c} c \\ | \\ c \\ | \\ T(n-2) \end{array} \Rightarrow \left.\begin{array}{c} c \\ | \\ c \\ | \\ \cdots \\ | \\ c \end{array}\right\} n \text{ times}$$

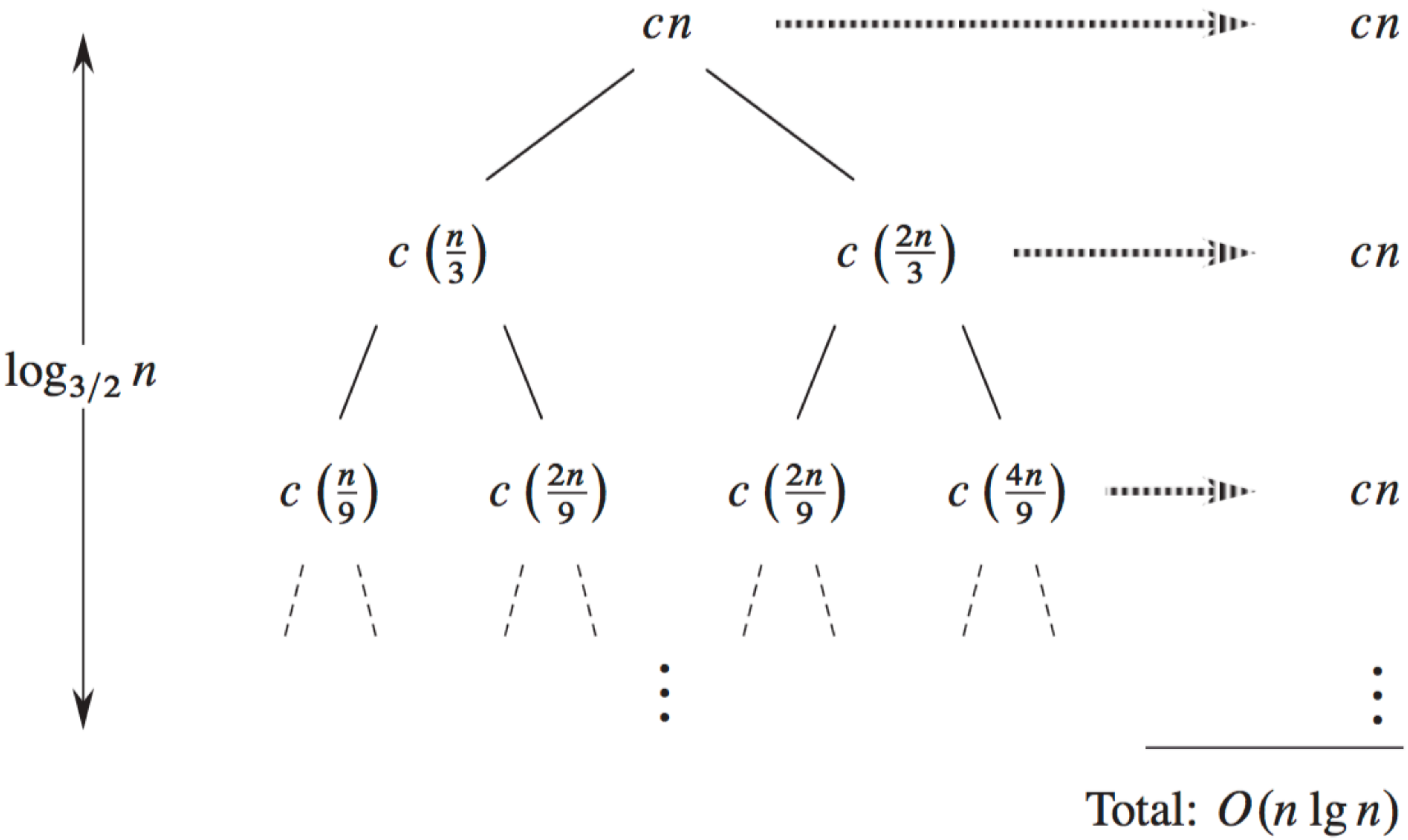
So, $T(n) = cn$ which is $O(n)$

A not so simple example

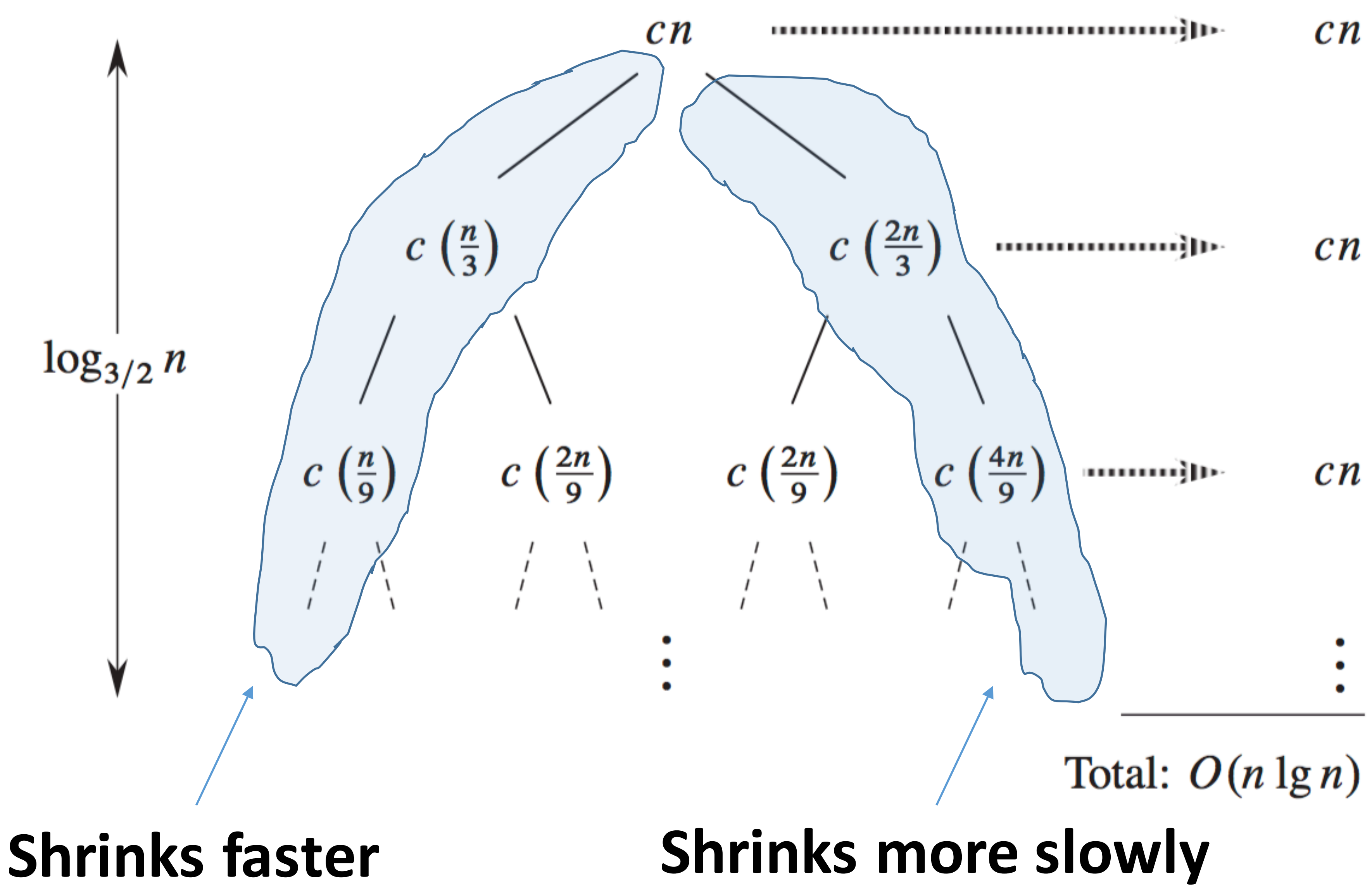
$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ T\left(\frac{n}{3}\right) + T\left(\frac{2\,n}{3}\right) + \Theta(n) & if\ n > 1 \end{cases}$$

- Find a good asymptotic upper bound (using big-O) for this recurrence.

A not so simple example



A not so simple example



A not so simple example

- In this example, some branches approach $n = 1$ faster, and some approach it later.
- However, we can still say that the number of levels is $O(\log n)$, because there are **at most $\boldsymbol{log_{3/2}\, n}$** levels and we know that $\log_{3/2} n = \Theta(\log n)$ – this can be proved using a property of logs
- So, again, $T(n) = O(n \log n)$.

Recursion tree method

- Recursion trees can easily get complicated for example try drawing the tree for $T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n^2)$

- Fortunately, we can use another method for recurrences like this.

Master method

MASTER THEOREM

The solution to **almost all** recurrences of the form

$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1, b > 1$

Master method

MASTER THEOREM

The solution to **almost all** recurrences of the form

$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1, b > 1$

Master method has **3 cases** based on what a, b, and $f(n)$ are.

Master method

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b > 1$$

Some examples of this form:

- $T(n) = 3T(n/4) + \Theta(n^2)$
- $T(n) = T(n/2) + 10n - 1$
- $T(n) = 2T(n/2) + 3$

Master method

$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1, b > 1$

Case 1: If $f(n) = O(n^{\log_\mathrm{b} a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

This means that if $f(n)$ is **polynomially smaller** than $n^{\log_b a}$, then $n^{\log_b a}$ determines the solution

Polynomial difference

All 5 functions on the left are smaller than n^3

$$\left.\begin{array}{c} n \\ n^2 \\ n^{2.5}\log n \\ \dfrac{n^3}{\log n} \\ \dfrac{n^3}{\log^5 n} \end{array}\right\} < n^3$$

Polynomial difference

All 5 functions on the left are smaller than n^3

Question: But can you guess which ones are polynomially smaller?

$$\left.\begin{array}{c} n \\ n^2 \\ n^{2.5}\log n \\ \dfrac{n^3}{\log n} \\ \dfrac{n^3}{\log^5 n} \end{array}\right\} < n^3$$

Polynomial difference

All 5 functions on the left are smaller than n^3

Question: But can you guess which ones are polynomially smaller?
Answer: Only $n, n^2, n^{2.5}\log n$

$$\left.\begin{array}{c} n \\ n^2 \\ n^{2.5}\log n \\ \dfrac{n^3}{\log n} \\ \dfrac{n^3}{\log^5 n} \end{array}\right\} < n^3$$

Polynomial difference

So, again some logarithmic or polylogarithmic function is consider much smaller than any polynomial.

Ex: $\log^6 n = o(n^{0.001})$ if n is large enough

notation	name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Polynomial difference

So, again some logarithmic or polylogarithmic function is consider much smaller than any polynomial.

The reason for requiring a significant polynomial difference is that the proof of master theorem won't work otherwise.

notation	name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Master method

Case 1: If $f(n) = O(n^{\log_{\mathrm{b}} a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(\sqrt{n})$$

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$$

Master method

Case 1: If $f(n) = O(n^{\log_{\mathrm{b}} a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

$T(n) = 4T\left(\frac{n}{2}\right) + n \qquad \rightarrow T(n) = \Theta(n^2)$

$T(n) = 2T\left(\frac{n}{2}\right) + \sqrt{n} \qquad \rightarrow T(n) = \Theta(n)$

$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$ → Master theorem does not apply

Master method

Case 2: If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta(n^{\log_b a} \log n)$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

Master method

Case 2: If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta(n^{\log_b a} \log n)$

$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \quad \rightarrow T(n) = \Theta(n^2 \log n)$

$T(n) = 2T\left(\frac{n}{2}\right) + n \quad \rightarrow T(n) = \Theta(n \log n)$

$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \quad \rightarrow T(n) = \log n$

(because we have $n^{\log_2 1} = 1 = \Theta(1)$)

Master method

Case 3: If $f(n) = \Omega\left(n^{\log_b a+\epsilon}\right)$, for some constant $\epsilon > 0$, then $T(n) = \Theta(f(n))$

$$T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n^2)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n\sqrt{n}\log n$$

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$$

Master method

Case 3: If $f(n) = \Omega\left(n^{\log_b a+\epsilon}\right)$, for some constant $\epsilon > 0$, then $T(n) = \Theta(f(n))$

$T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n^2)$ $\quad\rightarrow T(n) = \Theta(n^2)$

$T(n) = 2T\left(\frac{n}{2}\right) + n\sqrt{n}\log n$ $\quad\rightarrow T(n) = \Theta(n\sqrt{n}\log n)$

$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2\log n)$ Does not apply, but …

Master method

MASTER THEOREM

If $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1, b > 1$:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$, for some constant $\epsilon > 0$, and also $cf(n) \geq af\left(\frac{n}{b}\right)$ for some constant $c < 1$, then $T(n) = \Theta(f(n))$

In simple terms, assuming that $g(n) = n^{\log_b a}$, either f and g should be **asympt. the same**, or their gap should be **polynomial**

Master method

Master Theorem

If $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1, b > 1$:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$, for some constant $\epsilon > 0$, and also $\underbrace{cf(n) \geq af\left(\frac{n}{b}\right)}_{\text{Regularity Condition}}$ for some constant $c < 1$, then $T(n) = \Theta(f(n))$

Don't worry too much about the regularity condition. It's satisfied in most cases we deal with.

Generalization of case 2

Case 2: If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$, where $k \geq 0$

$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$ - The but example from before

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta\left(\frac{n^2}{\log n}\right)$$

Generalization of case 2

Case 2: If $f(n) = \Theta\left(n^{\log_{\mathrm{b}} a} \log^k n\right)$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$, where $k \geq 0$

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 \log n) \qquad \rightarrow T(n) = \Theta(n^2 \log^2 n)$$

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta\left(\frac{n^2}{\log n}\right) \qquad \rightarrow k = -1$$

so we can't use master theorem for this