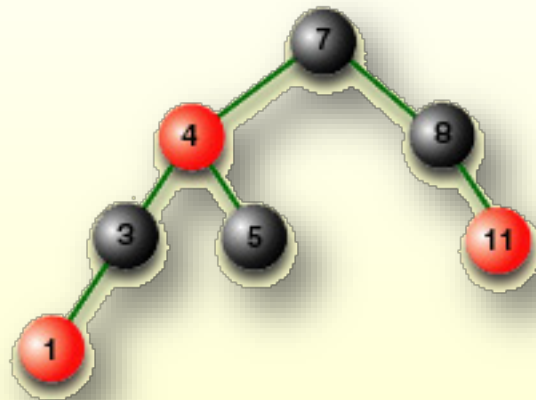


CS161: Design and Analysis of Algorithms



Lecture 3 Leonidas Guibas

Outline

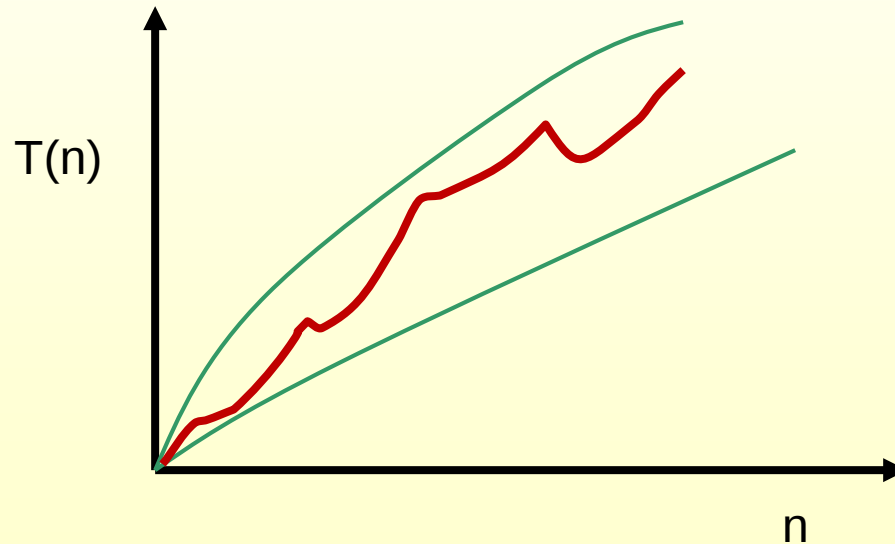
- ◆ Review of last lecture (asymptotic notations, recurrence relations)
- ◆ Key Topic Today: **Solving Recurrences**
 - ◆ using recursion trees (or iteration)
 - ◆ the master method
 - ◆ the substitution method

Slides modified from

- <http://www.cs.virginia.edu/~luebke/cs332/>

Asymptotic Bounds on Algorithm Performance

- Worst-case and average-case are difficult to analyze precisely -- the details can be **very complicated**



It may be easier to talk about upper and lower bounds on the function $T(n)$.

Review: Asymptotic Notations

- ◆ O : Big-Oh
- ◆ Ω : Big-Omega
- ◆ Θ : Theta
- ◆ o : Small-oh
- ◆ ω : Small-omega

Big O

- Informally, $O(g(n))$ is the set of all functions with a smaller or same order of growth as $g(n)$, within a constant multiple

Intuitively, O is like $\leq$
an upper bound notation

- If we say $f(n)$ is in $O(g(n))$, this means that $g(n)$ is an **asymptotic upper bound** on $f(n)$
 - Formally. $\exists C (>0) \ \& \ n_0, f(n) \leq Cg(n)$ for $\forall n \geq n_0$

$g(n)$ should be a “simple” function

Big Ω

- ◆ Informally, $\Omega(g(n))$ is the set of all functions with a larger or same order of growth as $g(n)$, within a constant multiple
- ◆ $f(n) \in \Omega(g(n))$ means $g(n)$ is an **asymptotic lower bound** of $f(n)$

Intuitively, Ω is like $f(n) \geq g(n)$

a lower bound notation

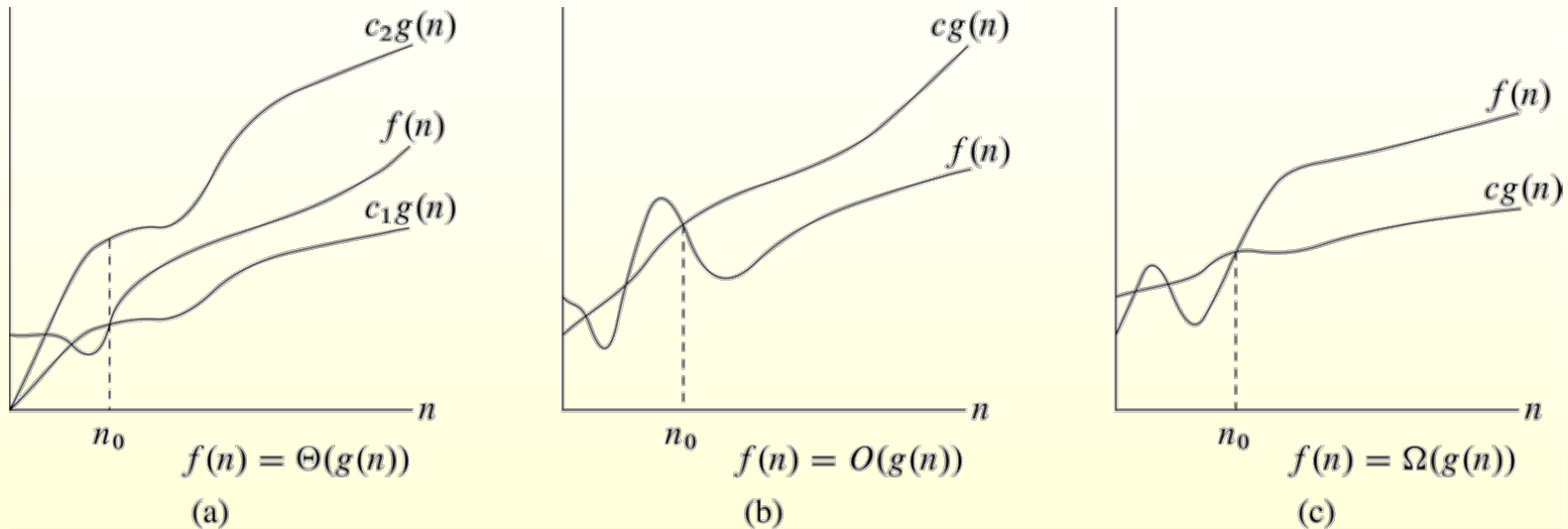
Theta (Θ): $\Theta = O$ and Ω

- ✦ Informally, $\Theta(g(n))$ is the set of all functions with the same order of growth as $g(n)$, within a constant multiple

Θ is like =

- ✦ $f(n) \in \Theta(g(n))$ means $g(n)$ is an **asymptotically tight bound** on $f(n)$
 - ✦ Intuitively, it is like $f(n) = g(n)$

O , Ω , and Θ



The definitions imply a constant n_0 *beyond which* they are satisfied. We do not care about small values of n – we care about **asymptotic behavior**.

Algorithm Efficiency via Recurrences

$$T(n) = T(n-1) + 1$$

$$T(n) = T(n-1) + n$$

$$T(n) = T(n/2) + 1$$

$$T(n) = 2T(n/2) + 1$$

Challenge: how to solve the recurrence to get a tight bound, e.g. $T(n) = \Theta(n^2)$ or $T(n) = \Theta(n \lg n)$, or at least an upper bound such as $T(n) = O(n^2)$?

Solving Recurrences

- ◆ The running time of many algorithms can be expressed in one of the following two recursive forms

$$T(n) = aT(n - b) + f(n)$$

or

$$T(n) = aT(n / b) + f(n)$$

Both can be hard to solve. We focus on relatively easy ones, of the kind you will encounter frequently in many real algorithms (and exams...)

Solving Recurrences

1. Recursion tree / iteration method
2. Master method
3. Substitution method

The Recursion Tree Method



Review: Back to MergeSort

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

Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.

Recurrence for MergeSort

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

- We saw that the cost of the Merge step is $\Theta(n)$.
- We'll 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).

Recursion Tree

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

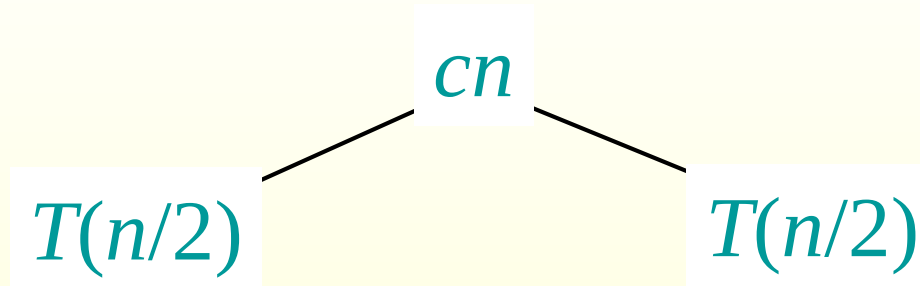
Recursion Tree

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

$$T(n)$$

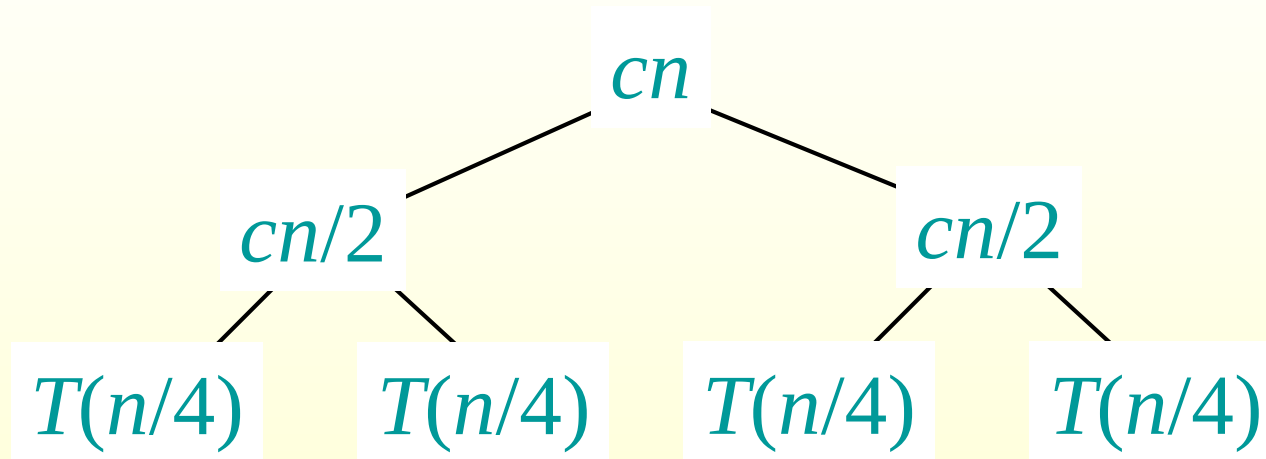
Recursion Tree

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



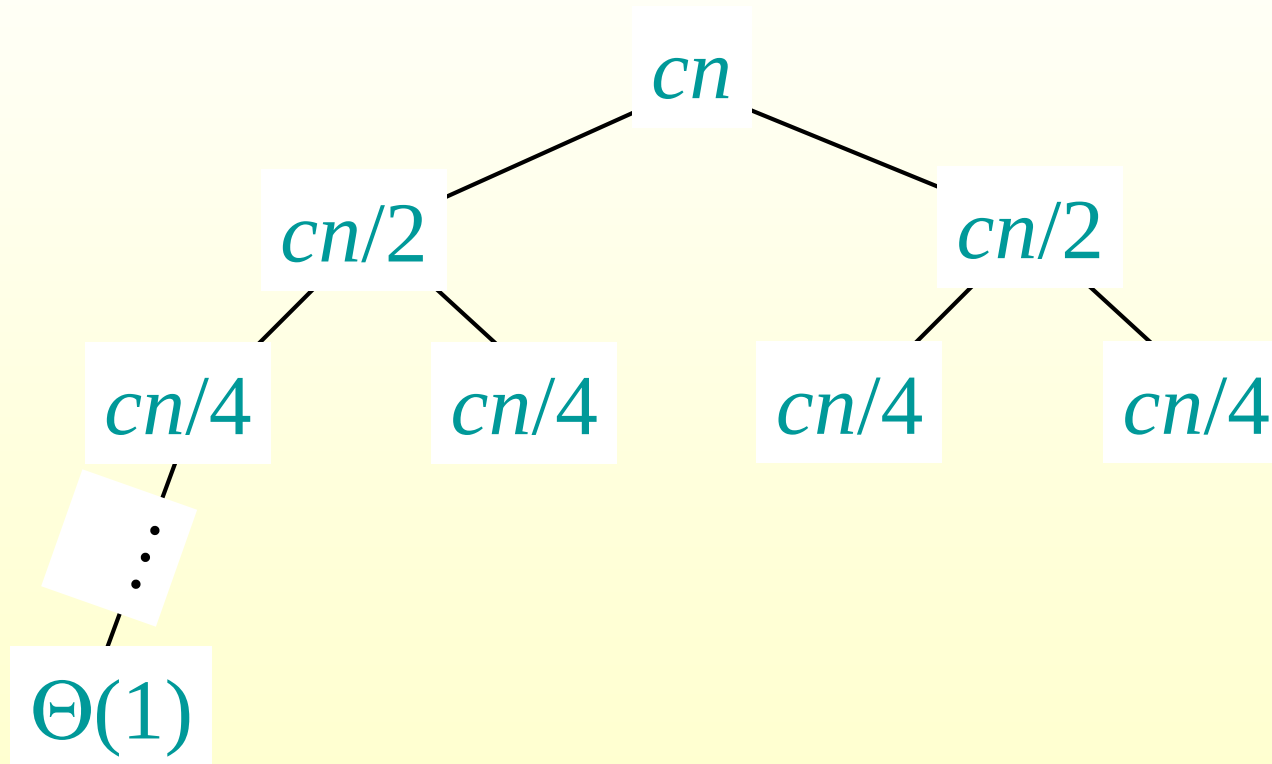
Recursion Tree

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



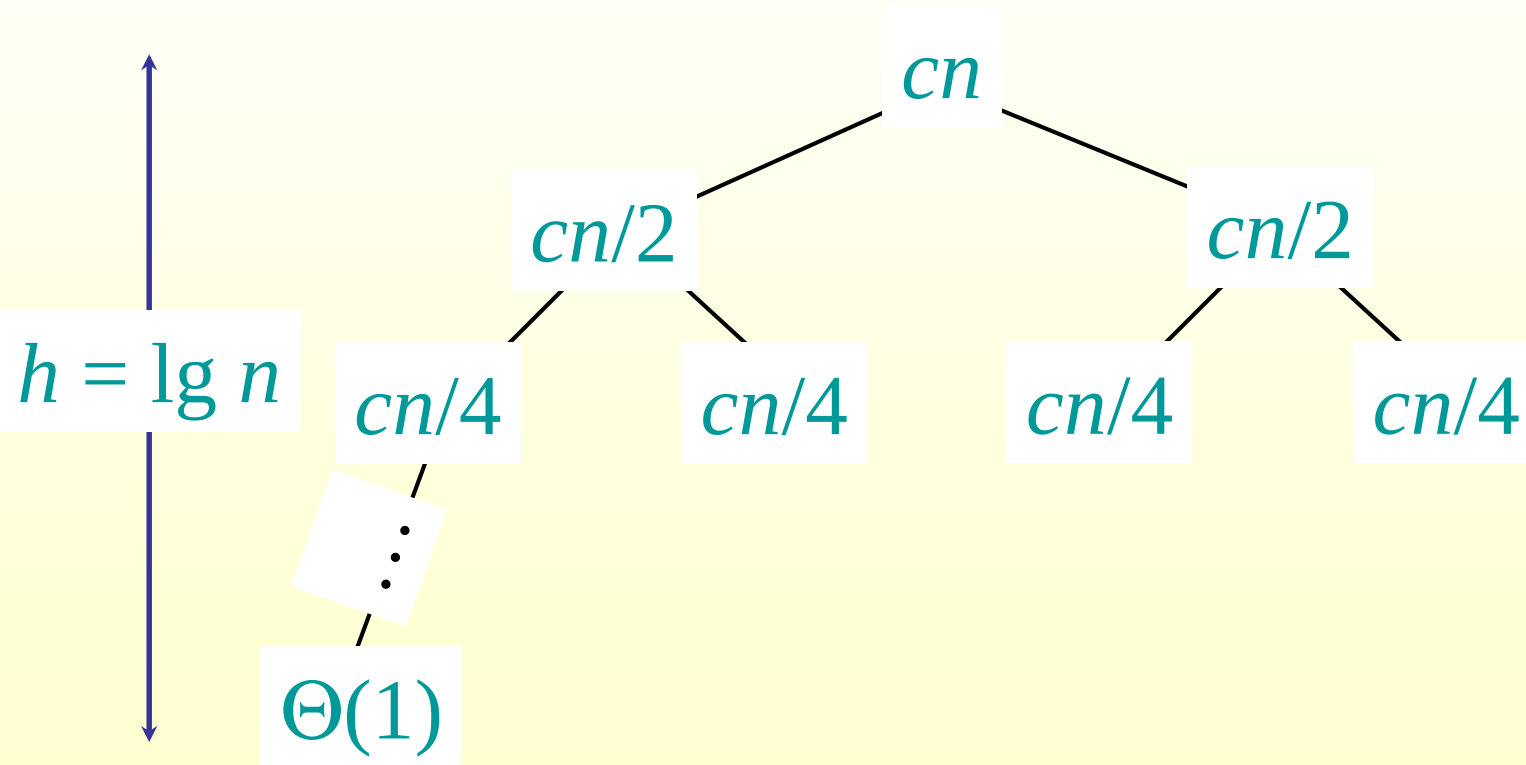
Recursion Tree

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



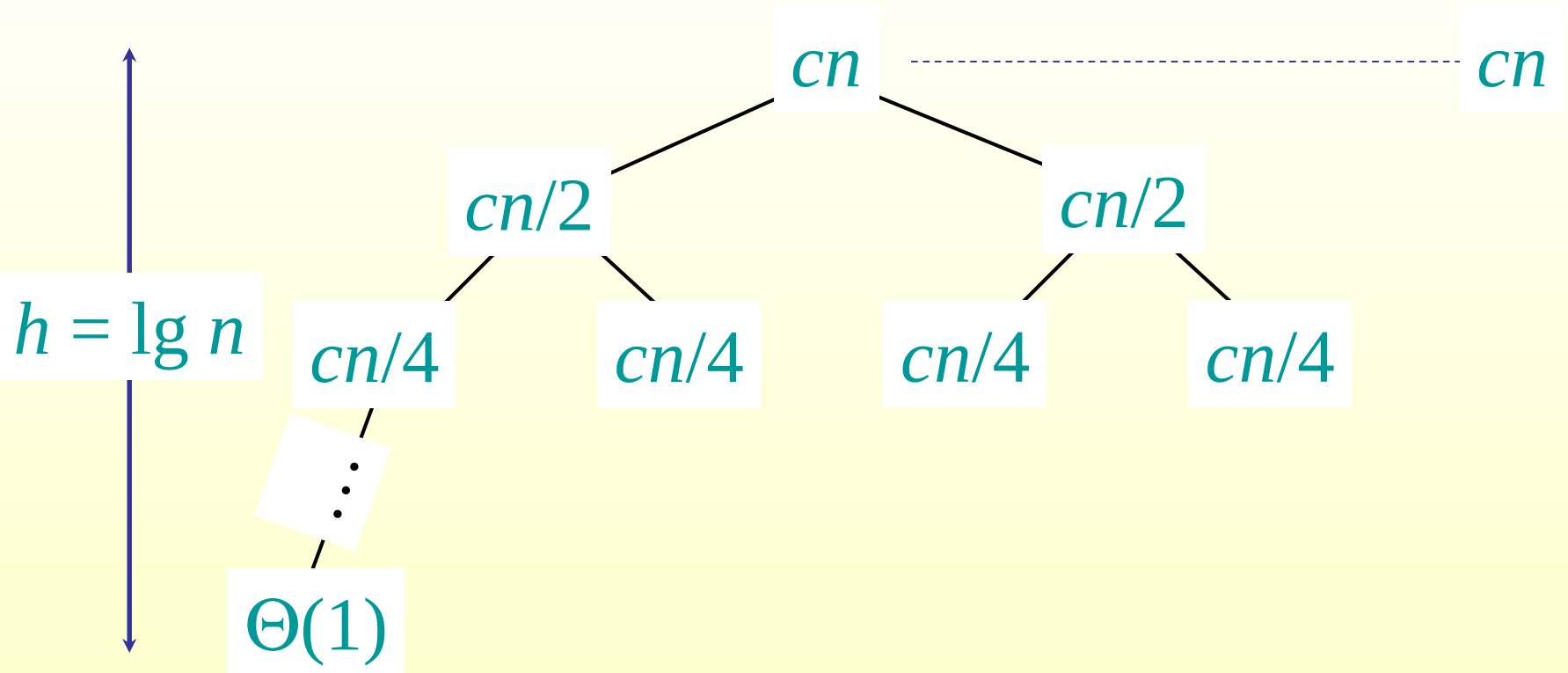
Recursion Tree

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



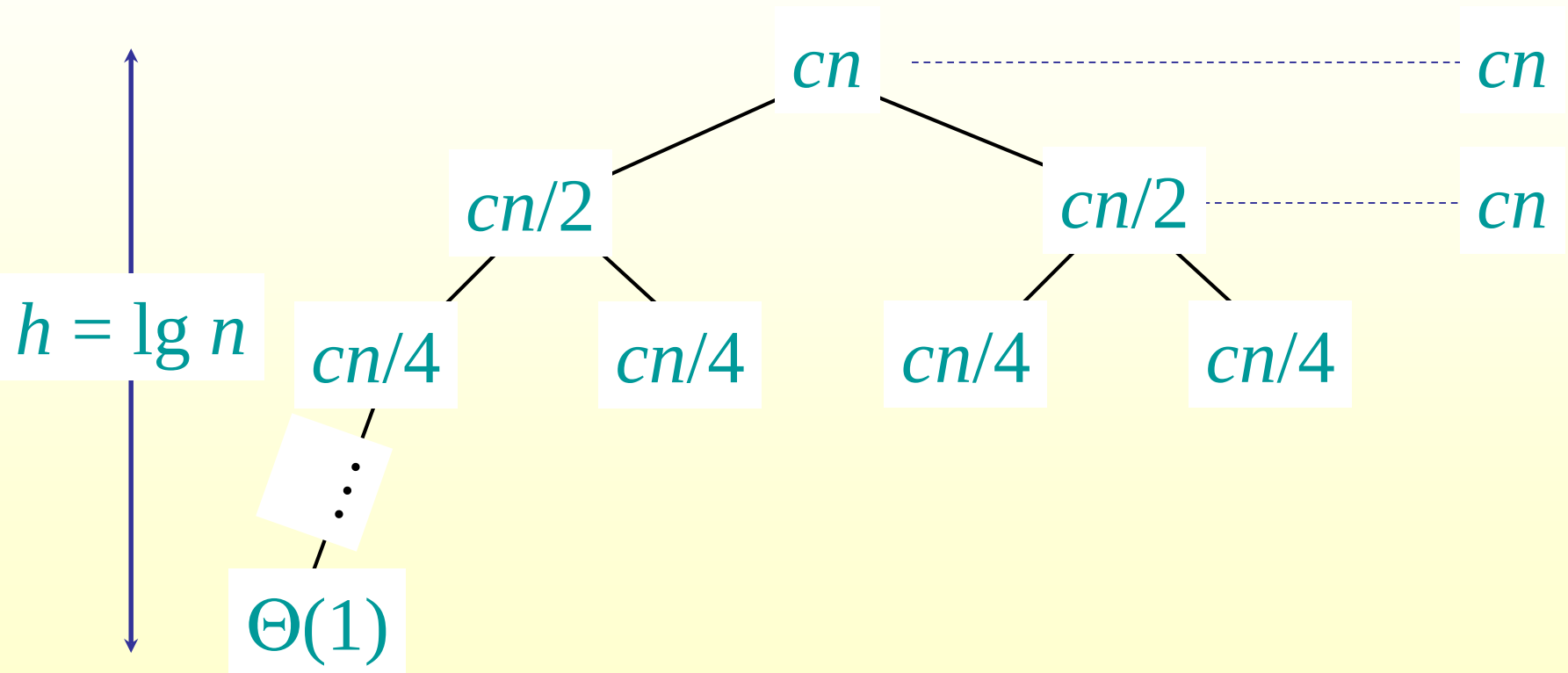
Recursion Tree

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



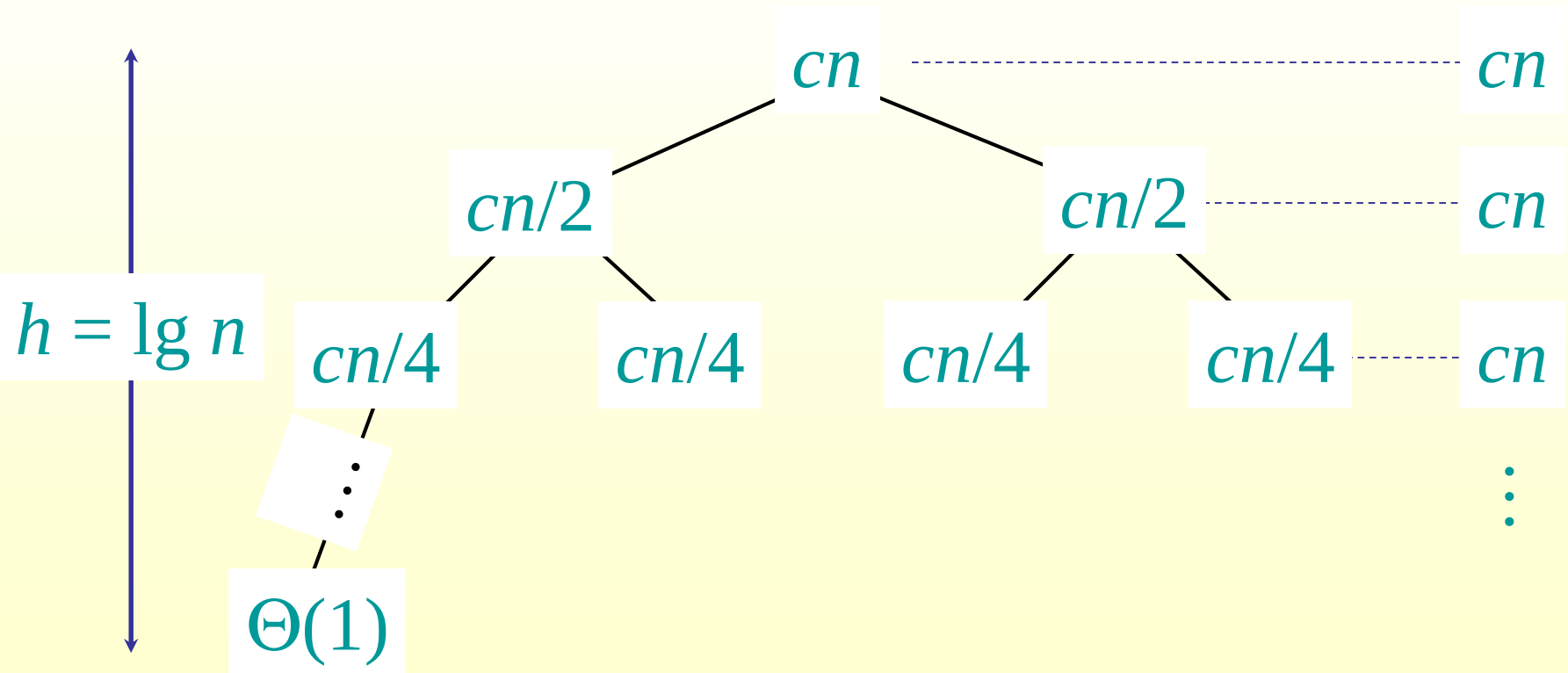
Recursion Tree

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



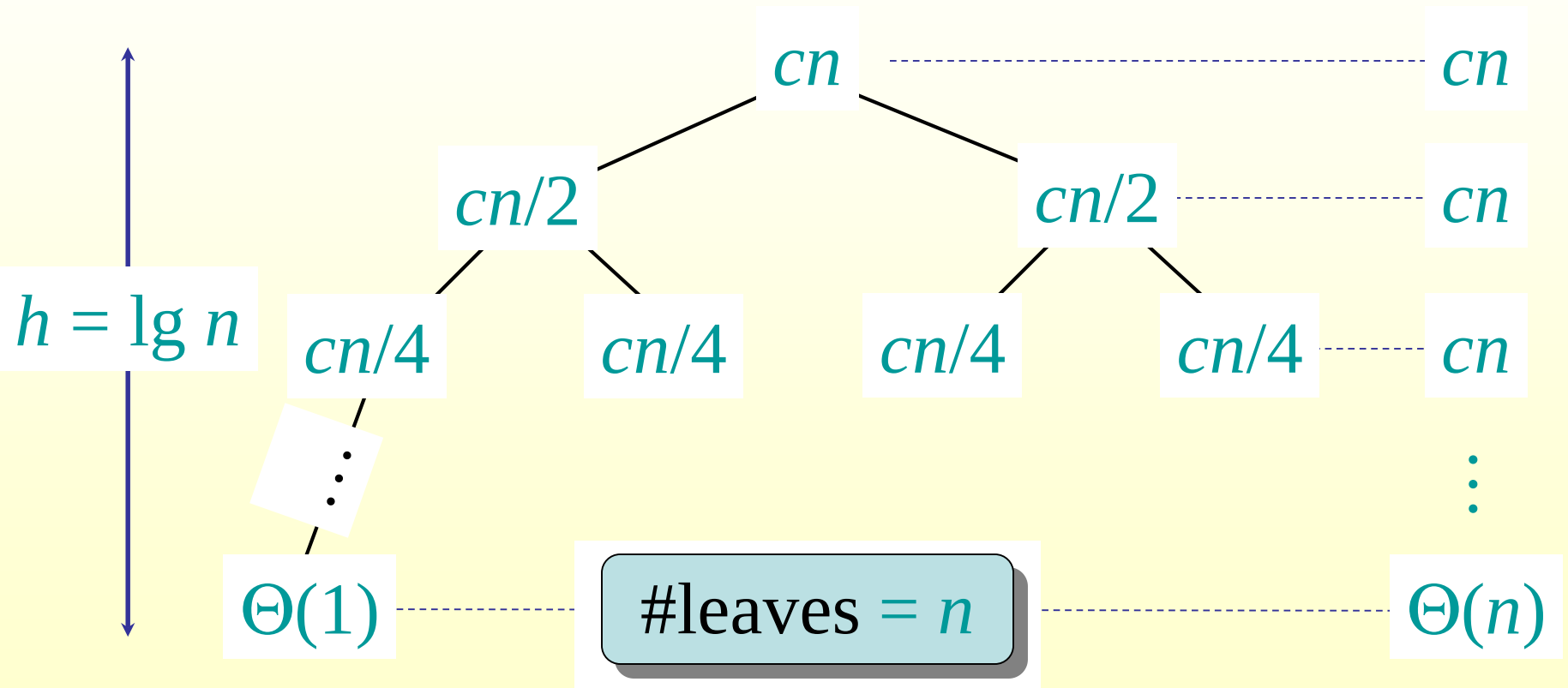
Recursion Tree

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



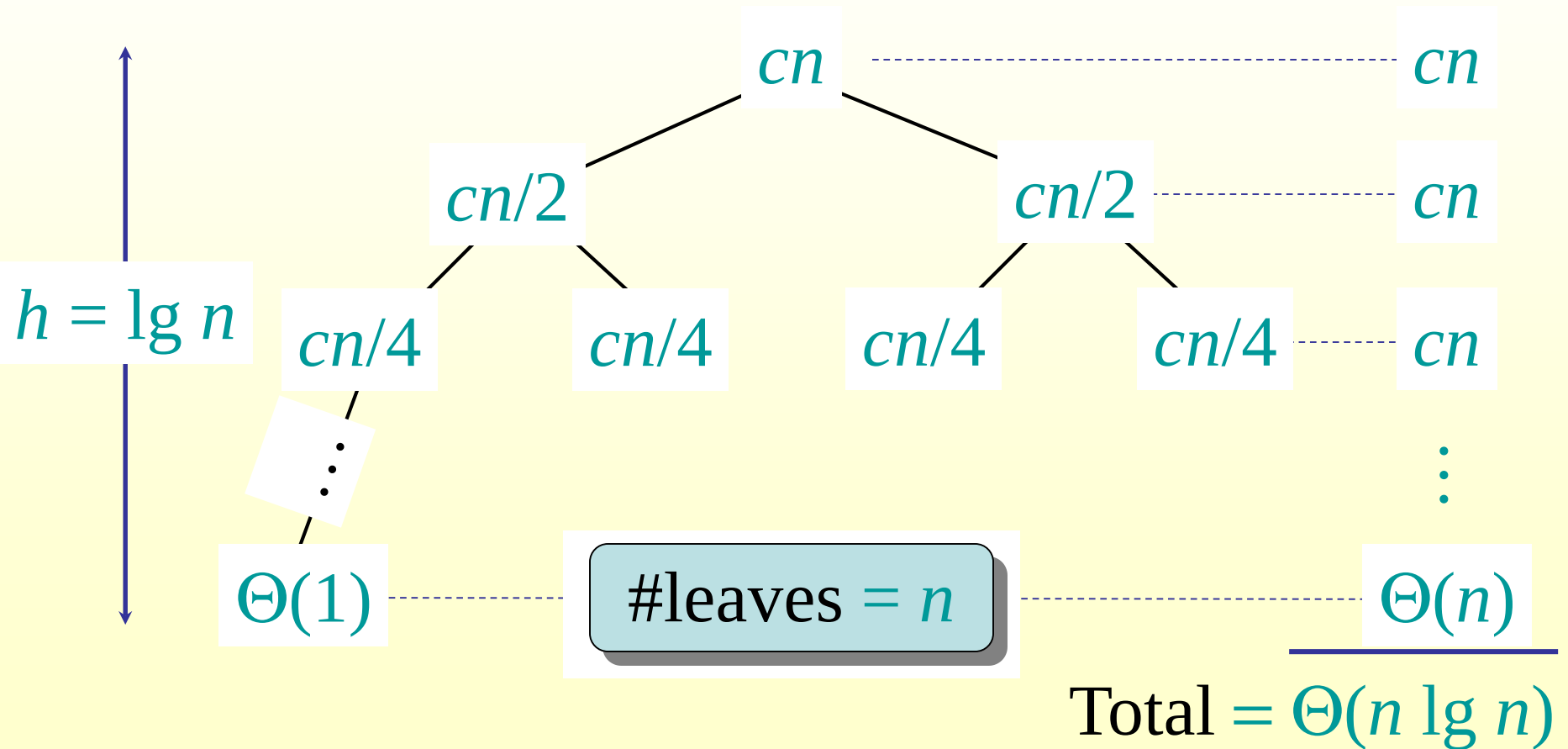
Recursion Tree

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.



Another Example

- How many multiplications do we need to compute 3^{16} ?

$$3^{16} = 3 \times 3 \times 3 \dots \times 3$$

Answer: 15

$$3^{16} = 3^8 \times 3^8$$

$$3^8 = 3^4 \times 3^4$$

$$3^4 = 3^2 \times 3^2$$

$$3^2 = 3 \times 3$$

Answer: 4

Pseudocode for Recursion

```
int pow (b, n)    // compute  $b^n$ 
    m = n >> 1;   // divide by 2
    p = pow (b, m);
    p = p * p;
    if (n % 2)
        return p * b;
    else
        return p;
```

Pseudocode Variations

```
int pow (b, n)
    m = n >> 1;
    p = pow (b, m);
    p = p * p;
    if (n % 2)
        return p * b;
    else
        return p;
```

```
int pow (b, n)
    m = n >> 1;
    p = pow(b,m) * pow(b,m);
    if (n % 2)
        return p * b;
    else
        return p;
```

Recurrence for Computing Power

int pow (b, n) Alg1

 m = n >> 1;

 p = pow (b, m);

 p = p * p;

 if (n % 2)

 return p * b;

 else

 return p;

$T(n) = T(n/2) + \Theta(1)$

int pow (b, n) Alg2

 m = n >> 1;

 p = pow(b, m) * pow(b, m);

 if (n % 2)

 return p * b;

 else

 return p;

$T(n) = 2T(n/2) + \Theta(1)$

Which algorithm is more efficient asymptotically?

Time Complexity for Alg1

Solve $T(n) = T(n/2) + 1$

$$\star T(n) = T(n/2) + 1$$

$$= T(n/4) + 1 + 1$$

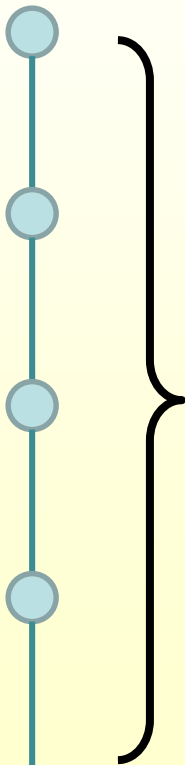
$$= T(n/8) + 1 + 1 + 1$$

$$= T(1) + 1 + 1 + \dots + 1$$

$$\underbrace{\hspace{10em}}_{\log(n)}$$

$$= \Theta(\log(n))$$

$\log(n)$



Iteration method

Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.

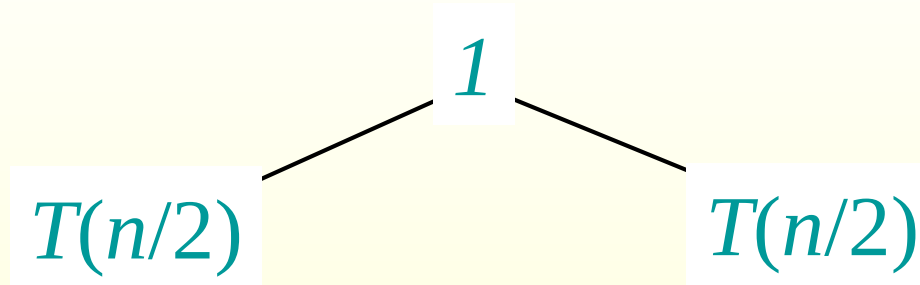
Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.

$T(n)$

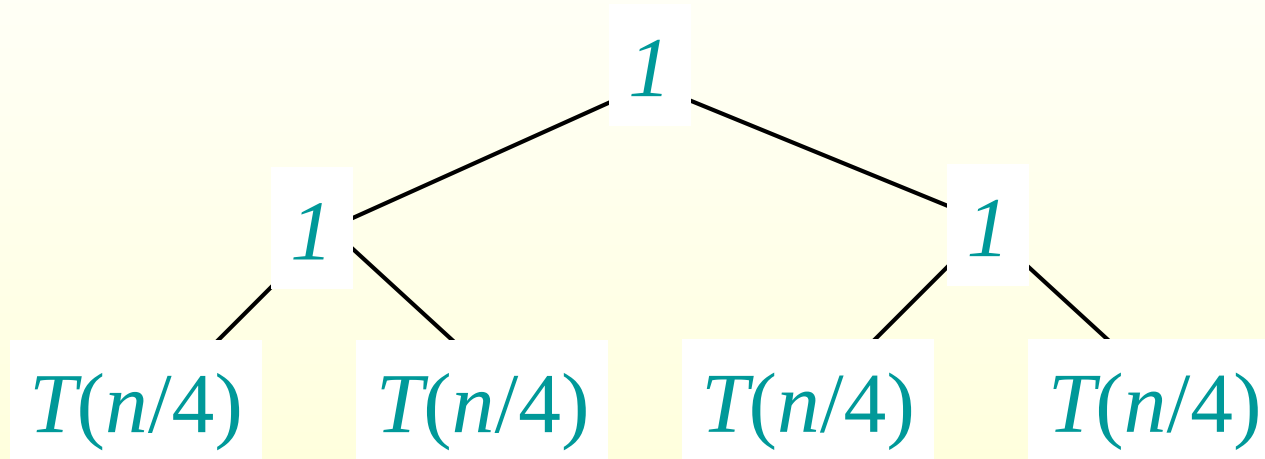
Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



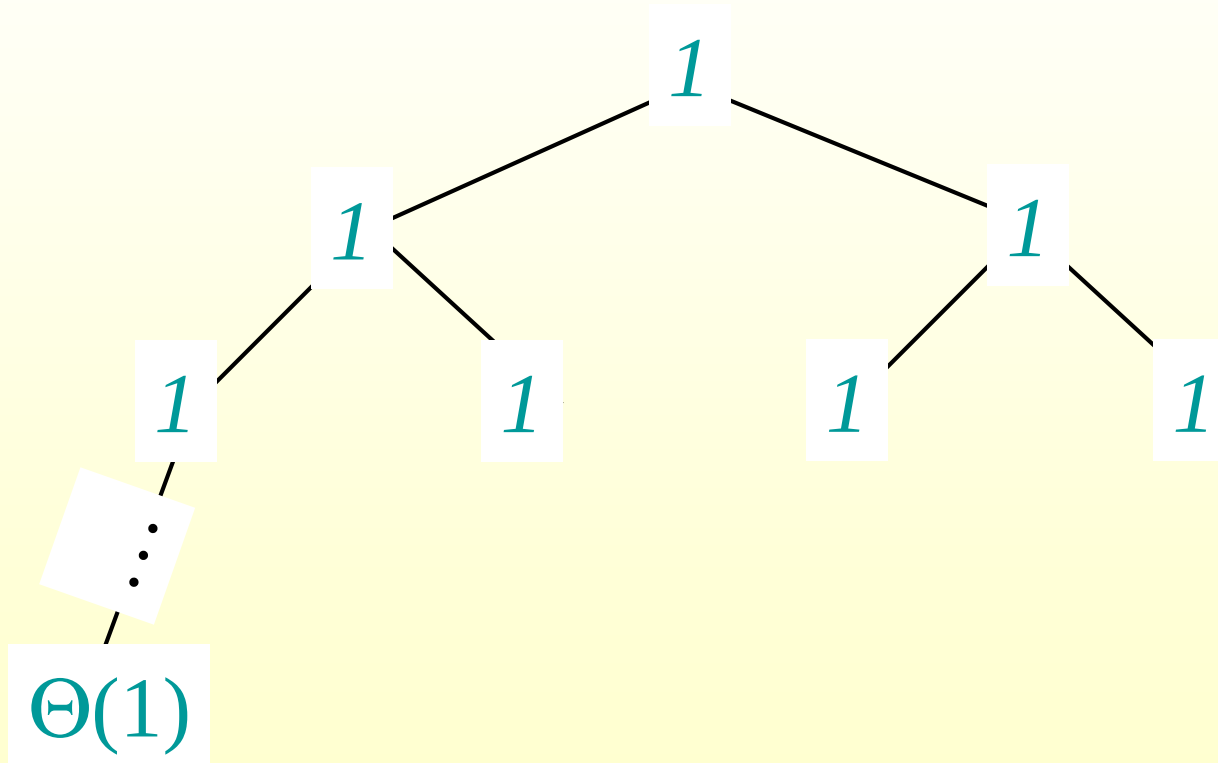
Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



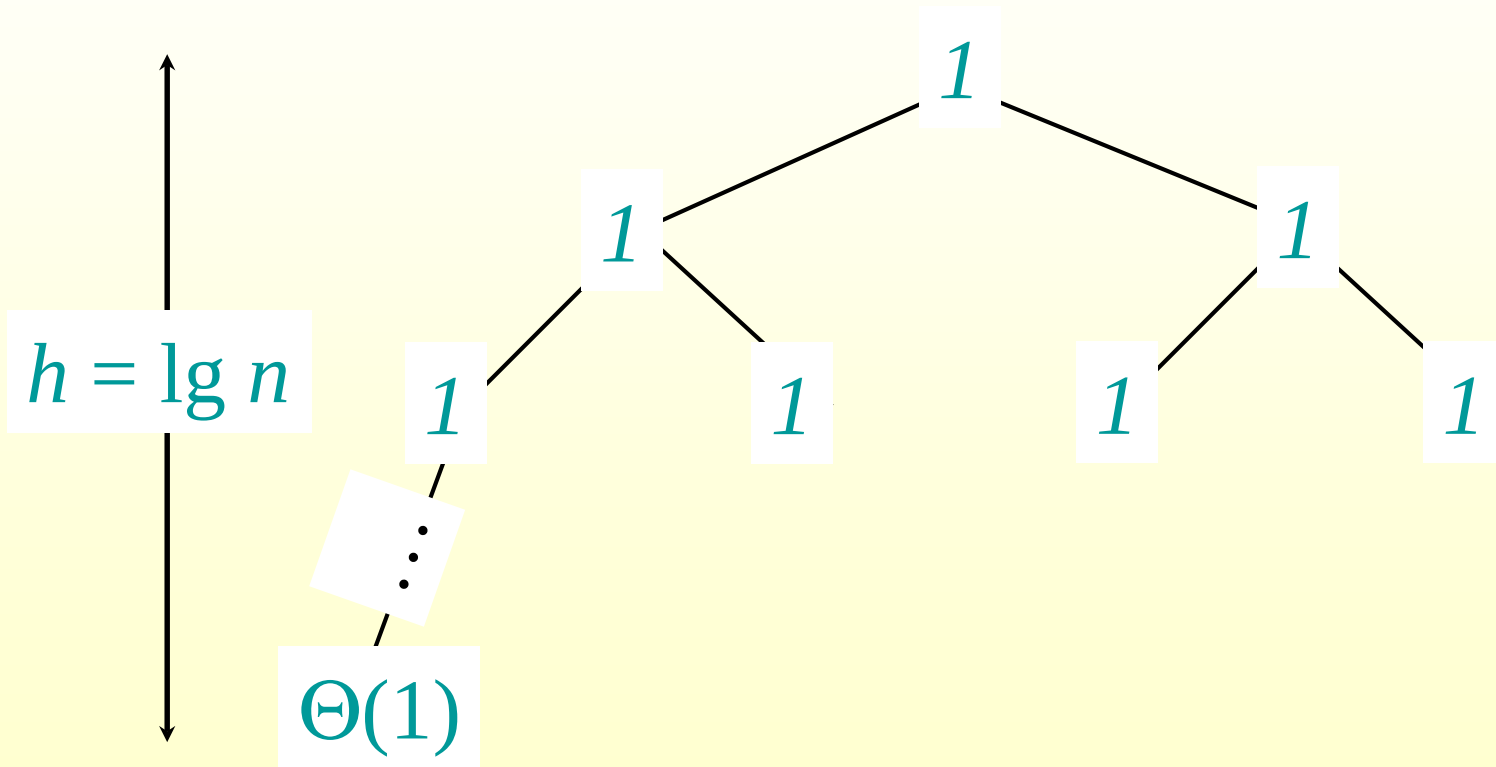
Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



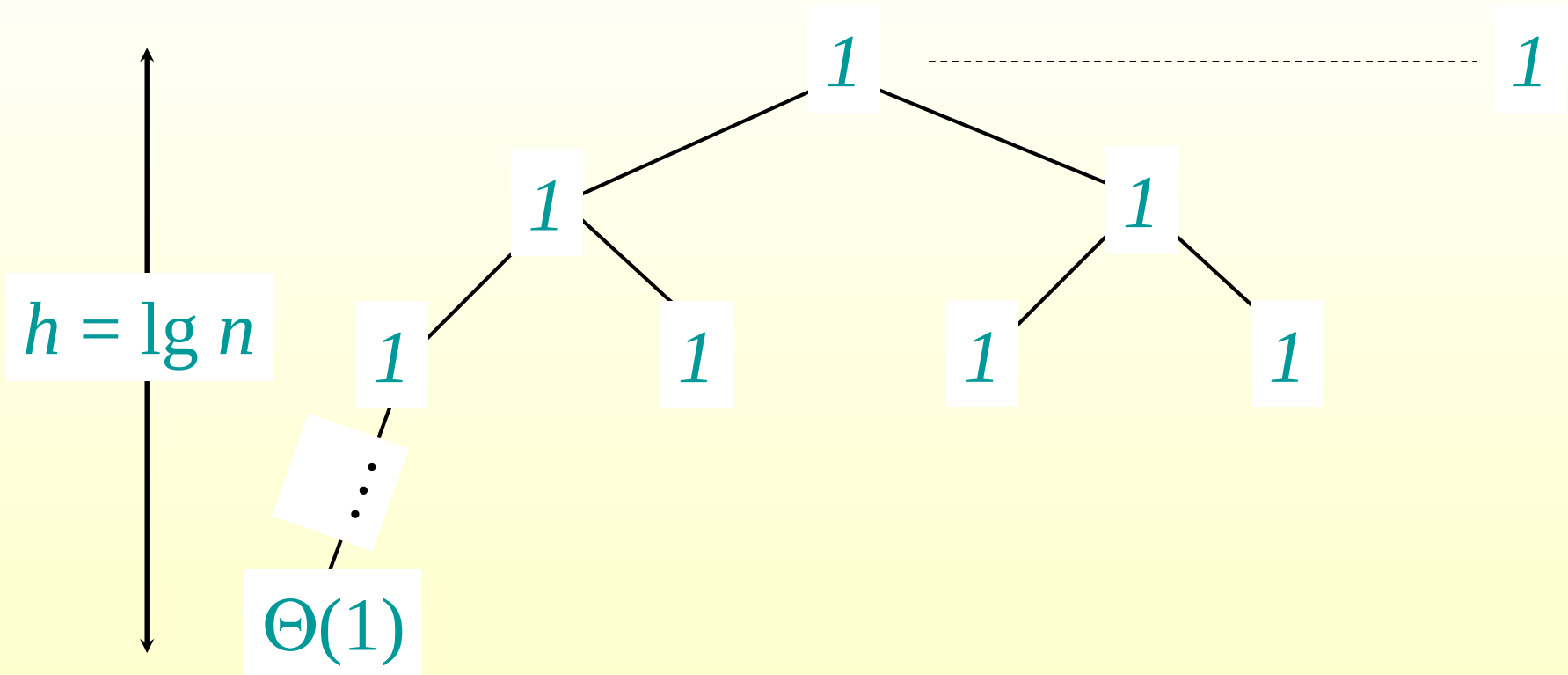
Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



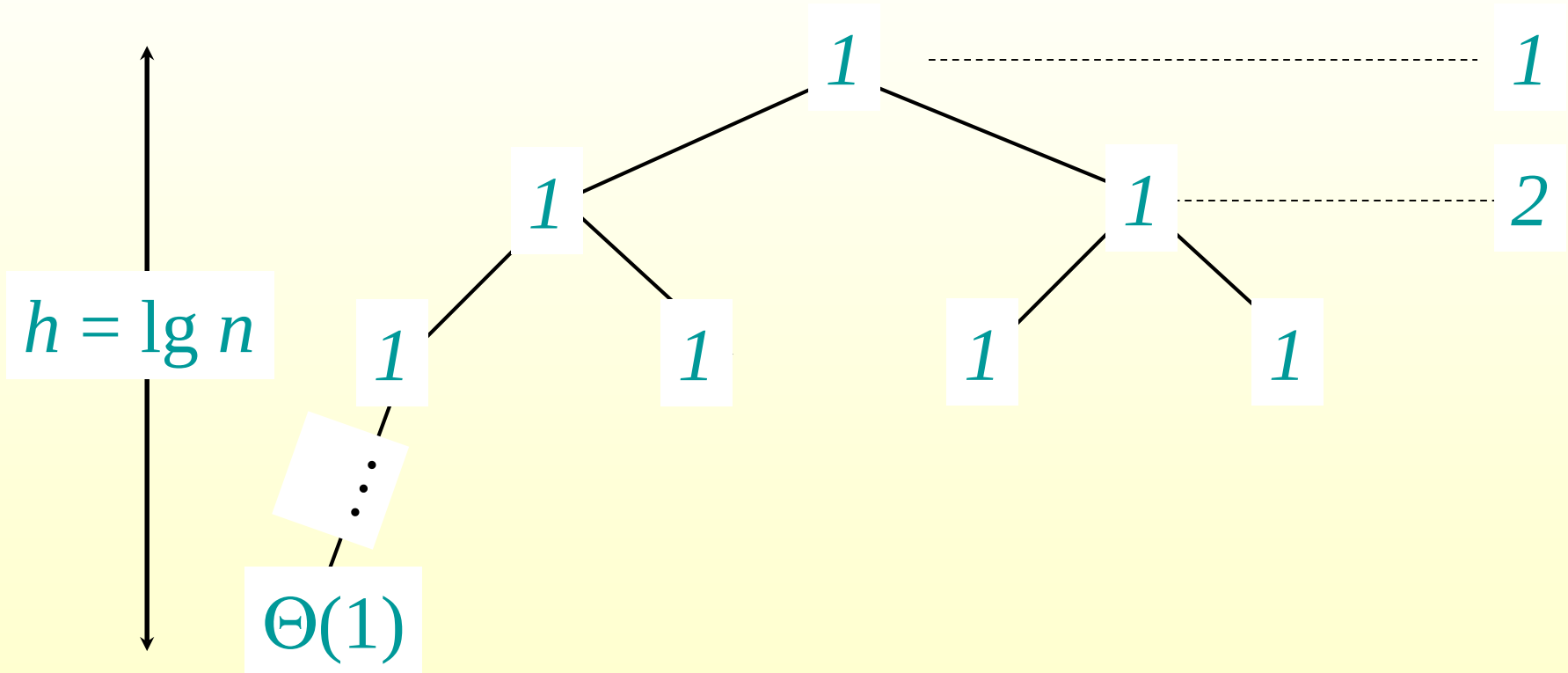
Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



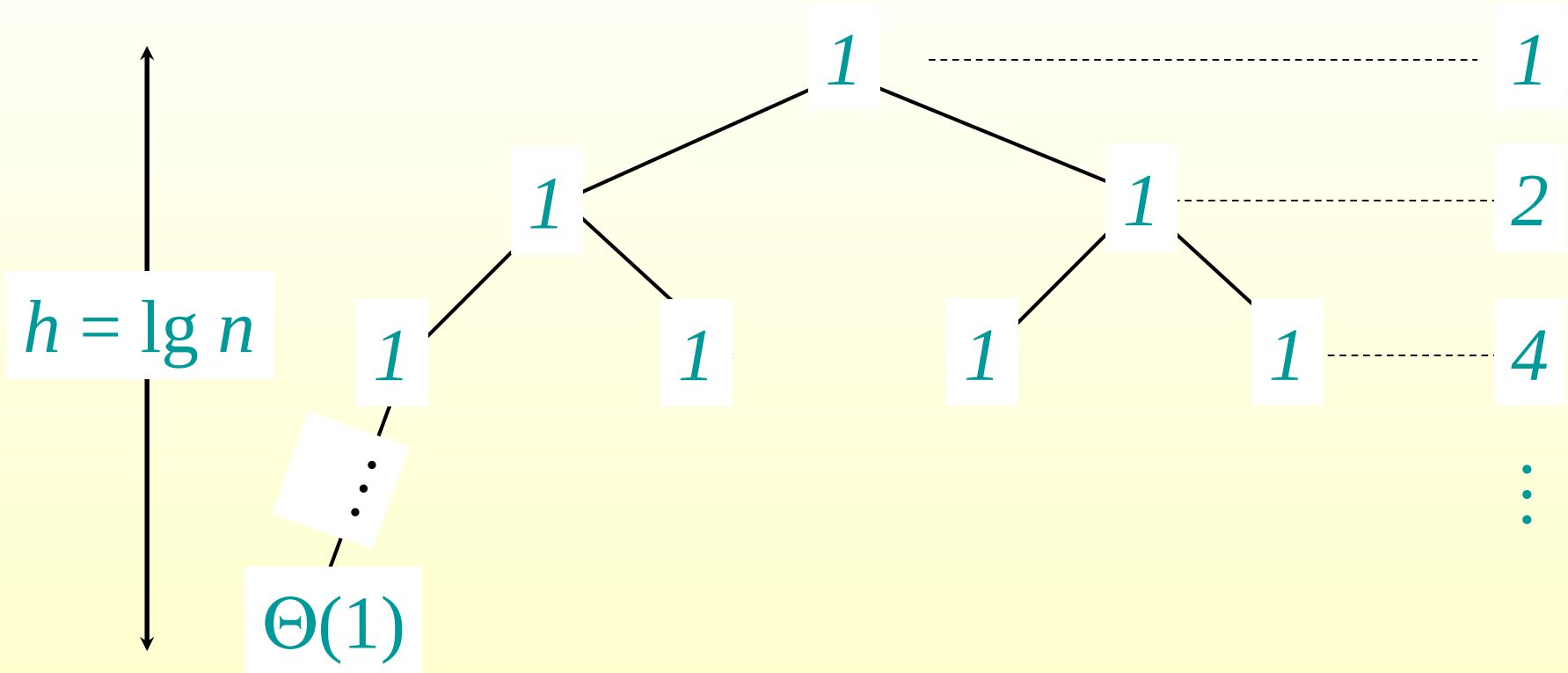
Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



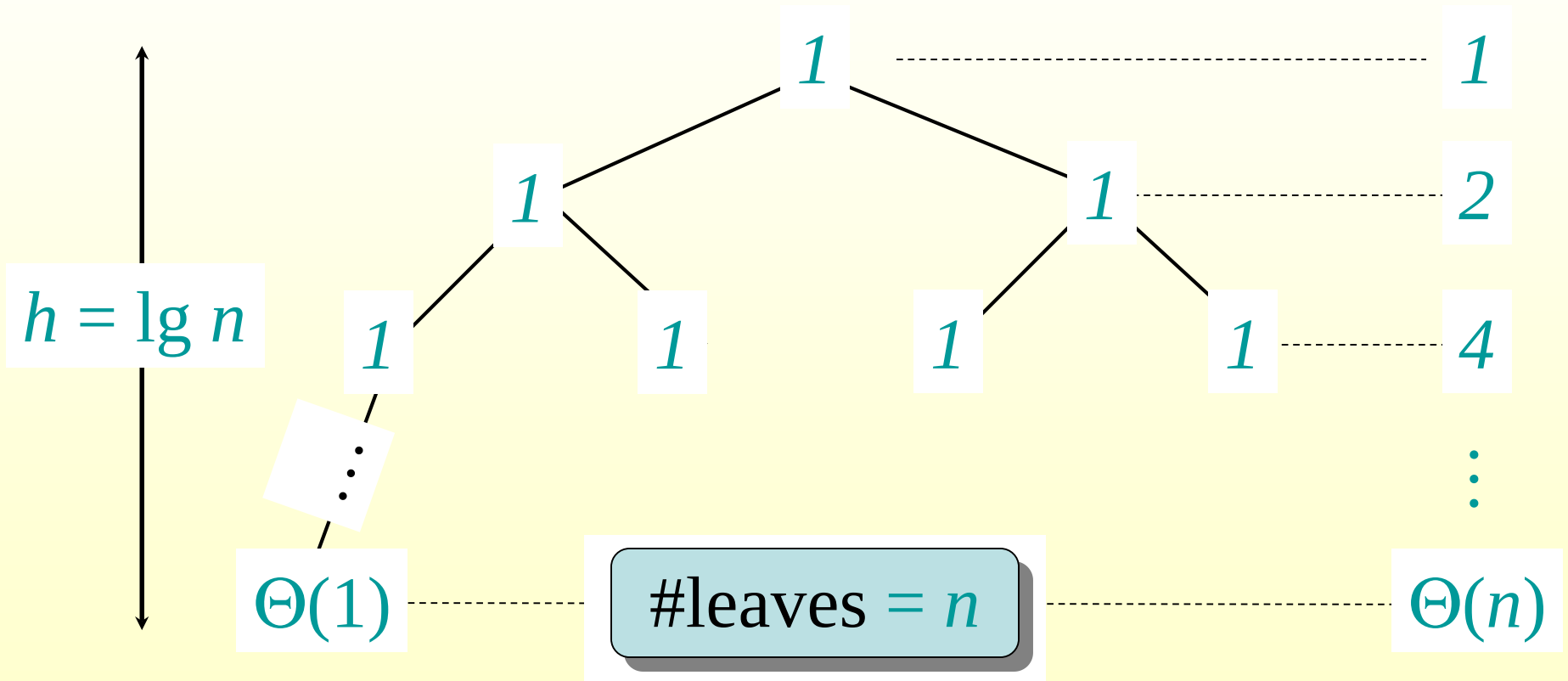
Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



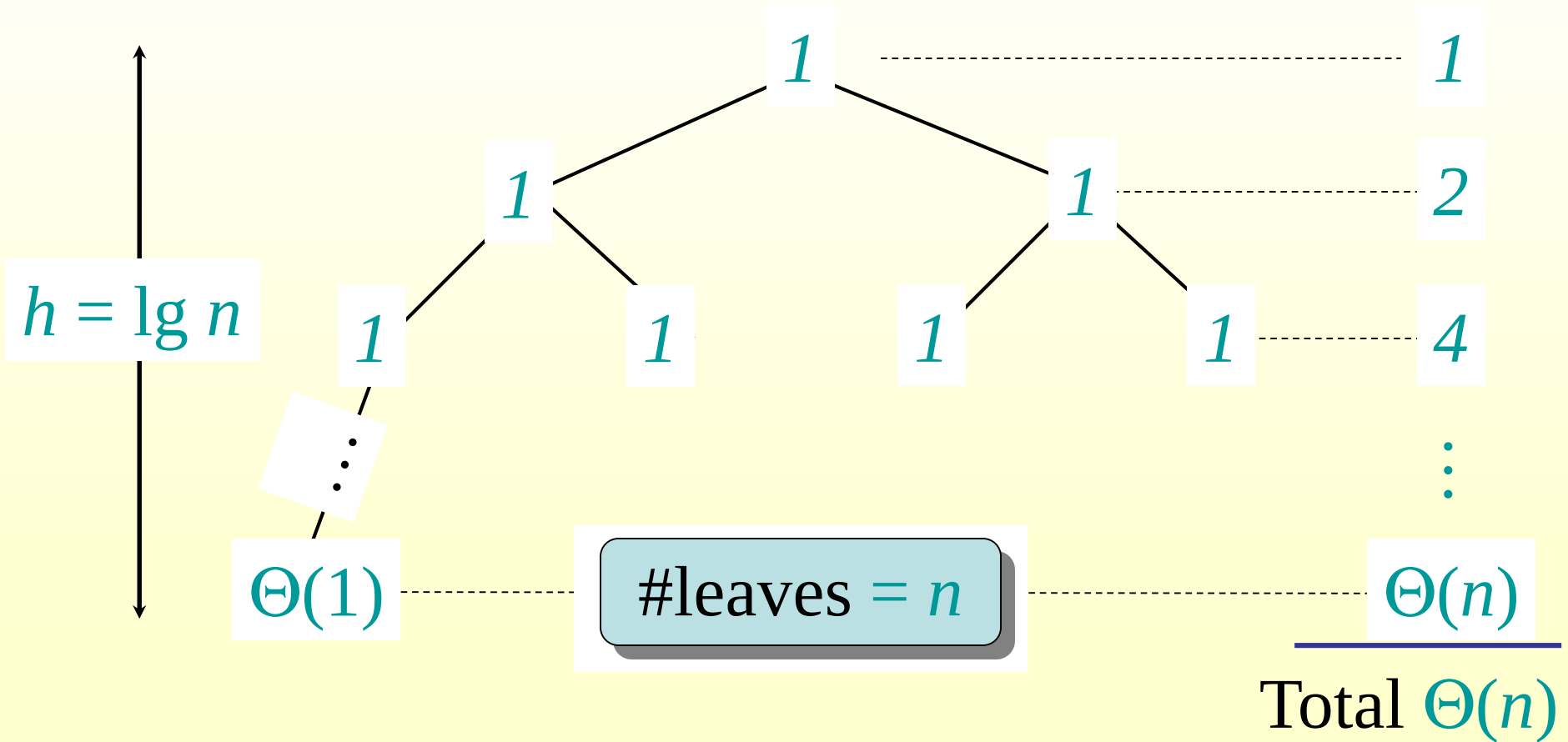
Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



Time Complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



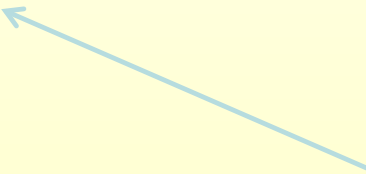
$$1 + 2 + 4 + \dots + 2^k = 2^{k+1} - 1$$

More Iteration Method Examples

$$\begin{aligned} \spadesuit T(n) &= T(n-1) + 1 \\ &= T(n-2) + 1 + 1 \\ &= T(n-3) + 1 + 1 + 1 \\ &= T(1) + \underbrace{1 + 1 + \dots + 1}_{n-1} \\ &= \Theta(n) \end{aligned}$$

More Iteration Method Examples

$$\begin{aligned} \spadesuit T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &= T(1) + 2 + 3 + \dots + n \\ &= \Theta(n^2) \end{aligned}$$



Saw the same sum in InsertionSort

3-Way-MergeSort

```
3-way-merge-sort (A[1..n])
```

```
  If ( $n \leq 1$ ) return;
```

```
  3-way-merge-sort(A[1.. $n/3$ ]);
```

```
  3-way-merge-sort(A[ $n/3+1$ .. $2n/3$ ]);
```

```
  3-way-merge-sort(A[ $2n/3+1$ ..  $n$ ]);
```

```
  Merge A[1.. $n/3$ ] and A[ $n/3+1$ .. $2n/3$ ];
```

```
  Merge A[1.. $2n/3$ ] and A[ $2n/3+1$ .. $n$ ];
```

- Is this algorithm correct?
- What's the recurrence function for the running time?
- What does the recurrence function solve to?

Unbalanced-MergeSort

```
ub-merge-sort (A[1..n])
```

```
  if ( $n \leq 1$ ) return;
```

```
  ub-merge-sort(A[1.. $n/3$ ]);
```

```
  ub-merge-sort(A[ $n/3+1$ ..  $n$ ]);
```

```
  Merge A[1..  $n/3$ ] and A[ $n/3+1$ .. $n$ ].
```

- Is this algorithm correct?
- What's the recurrence function for the running time?
- What does the recurrence function solve to?

More Recursion Tree Examples

◆ $T(n) = 3T(n/3) + n$ [3-Way MergeSort]

◆ $T(n) = T(n/3) + T(2n/3) + n$ [ub-MergeSort]

◆ $T(n) = 3T(n/4) + n$

◆ $T(n) = 3T(n/4) + n^2$

The Master Method



The Master Method

The master method applies to recurrences of the form

$$T(n) = a T(n/b) + f(n) ,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

1. **Divide** the problem into a subproblems, **each** of size n/b
 2. **Conquer** the subproblems by solving them recursively.
 3. **Combine** subproblem solutions
- Divide + combine takes $f(n)$ time.

Master Theorem

$$T(n) = a T(n/b) + f(n)$$

Key: compare $f(n)$ with $n^{\log_b a}$

CASE 1: $f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$

.

CASE 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $\underline{a f(n/b) \leq c f(n)}$

Regularity Condition

$\Rightarrow T(n) = \Theta(f(n))$.

$$n^{\log_b a} = a^{\log_b n}$$

Case 1

$f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.

Alternatively: $n^{\log_b a} / f(n) = \Omega(n^\epsilon)$

Intuition: $f(n)$ grows **polynomially** slower than $n^{\log_b a}$

Or: $n^{\log_b a}$ dominates $f(n)$ by an n^ϵ factor for some $\epsilon > 0$

Solution: $T(n) = \Theta(n^{\log_b a})$

$$T(n) = 4T(n/2) + n$$

$$b = 2, a = 4, f(n) = n$$

$$\log_2 4 = 2$$

$$f(n) = n = O(n^{2-\epsilon}), \text{ or}$$

$$n^2 / n = n^1 = \Omega(n^\epsilon), \text{ for } \epsilon = 1$$

$$\therefore T(n) = \Theta(n^2)$$

$$T(n) = 2T(n/2) + n/\log n$$

$$b = 2, a = 2, f(n) = n / \log n$$

$$\log_2 2 = 1$$

$$f(n) = n/\log n \notin O(n^{1-\epsilon}), \text{ or}$$

$$n^1 / f(n) = \log n \notin \Omega(n^\epsilon), \text{ for any } \epsilon > 0$$

$\therefore$ CASE 1 does not apply

Case 2

$$f(n) = \Theta(n^{\log_b a}).$$

Intuition: $f(n)$ and $n^{\log_b a}$ have the same asymptotic order.

Solution: $T(n) = \Theta(n^{\log_b a} \log n)$

e.g. $T(n) = T(n/2) + 1$

$$\log_b a = 0$$

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

$$\log_b a = 1$$

$$T(n) = 4T(n/2) + n^2$$

$$\log_b a = 2$$

$$T(n) = 8T(n/2) + n^3$$

$$\log_b a = 3$$

Case 3

$f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.

Alternatively: $f(n) / n^{\log_b a} = \Omega(n^\varepsilon)$

Intuition: $f(n)$ grows **polynomially** faster than $n^{\log_b a}$

Or: $f(n)$ dominates $n^{\log_b a}$ by an n^ε factor for some $\varepsilon > 0$

Solution: $T(n) = \Theta(f(n))$

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

$$b = 2, a = 1, f(n) = n$$

$$n^{\log_2 1} = n^0 = 1$$

$$f(n) = n = \Omega(n^{0+\varepsilon}), \text{ or}$$

$$n / 1 = n = \Omega(n^\varepsilon)$$

$$\therefore T(n) = \Theta(n)$$

$$T(n) = T(n/2) + \log n$$

$$b = 2, a = 1, f(n) = \log n$$

$$n^{\log_2 1} = n^0 = 1$$

$$f(n) = \log n \notin \Omega(n^{0+\varepsilon}), \text{ or}$$

$$f(n) / n^{\log_2 1} = \log n \notin \Omega(n^\varepsilon)$$

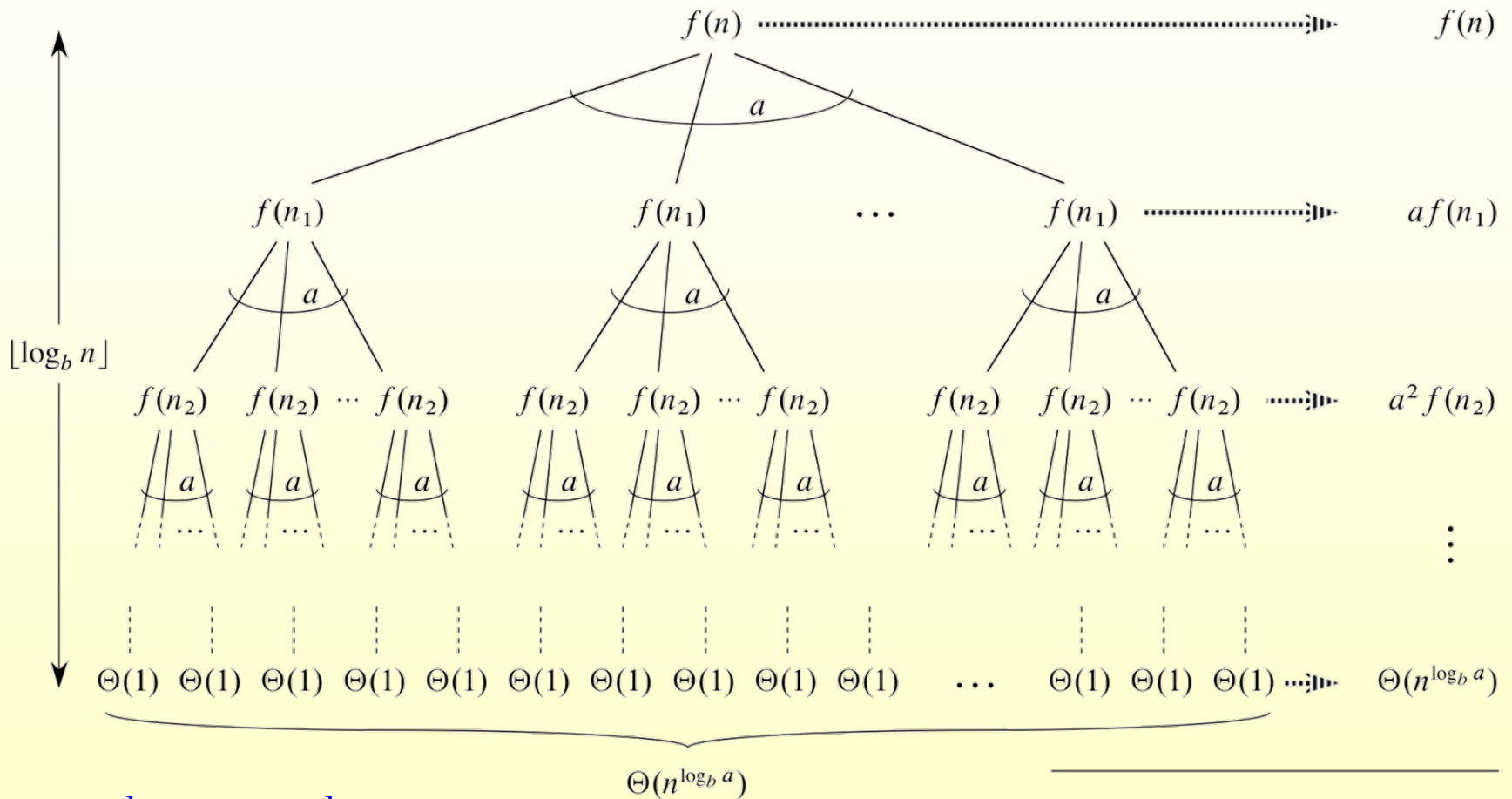
$\therefore$ CASE 3 does not apply

Regularity Condition

- ◆ $a f(n/b) \leq c f(n)$ for some $c < 1$ and all sufficiently large n
- ◆ This is needed for the master method to be mathematically correct.
 - ◆ to deal with some non-converging functions such as sine or cosine functions
- ◆ For most $f(n)$ you'll see (e.g., polynomial, logarithm, exponential), you can safely ignore this condition, because it is implied by the first condition $f(n) = \Omega(n^{\log_b a + \epsilon})$

Proof by Picture

$$n_i = n / b^i$$



$$n^{\log_b a} = a^{\log_b n}$$

$$\text{Total: } \Theta(n^{\log_b a}) + \sum_{j=0}^{\lceil \log_b n \rceil - 1} a^j f(n_j) \quad 54$$

Examples

$$T(n) = 4T(n/2) + n$$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n.$$

CASE 1: $f(n) = O(n^{2-\epsilon})$ for $\epsilon = 1$.

$$\therefore T(n) = \Theta(n^2).$$

$$T(n) = 4T(n/2) + n^2$$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2.$$

CASE 2: $f(n) = \Theta(n^2)$.

$$\therefore T(n) = \Theta(n^2 \log n).$$

Examples

$$T(n) = 4T(n/2) + n^3$$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^3.$$

CASE 3: $f(n) = \Omega(n^{2+\epsilon})$ for $\epsilon = 1$
and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.
 $\therefore T(n) = \Theta(n^3)$.

$$T(n) = 4T(n/2) + n^2/\log n$$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2/\log n.$$

Master method does not apply. In particular, for every constant $\epsilon > 0$, we have $n^\epsilon = \omega(\log n)$.

Examples

$$T(n) = 4T(n/2) + n^{2.5}$$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^{2.5}.$$

CASE 3: $f(n) = \Omega(n^{2+\epsilon})$ for $\epsilon = 0.5$
and $4(n/2)^{2.5} \leq cn^{2.5}$ (reg. cond.) for $c = 0.75$.
 $\therefore T(n) = \Theta(n^{2.5})$.

$$T(n) = 4T(n/2) + n^2 \log n$$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2 \log n.$$

Master method does not apply. In particular, for every constant $\epsilon > 0$, we have $n^\epsilon = \omega(\log n)$.

How do I know which case to use? Do I need to try all three cases one by one?

◆ Compare $f(n)$ with $n^{\log_b a}$

check if $n^{\log_b a} / f(n) \in \Omega(n^\epsilon)$

◆ $f(n) \in \begin{cases} o(n^{\log_b a}) & \text{Possible CASE 1} \\ \Theta(n^{\log_b a}) & \text{CASE 2} \\ \omega(n^{\log_b a}) & \text{Possible CASE 3} \end{cases}$

check if $f(n) / n^{\log_b a} \in \Omega(n^\epsilon)$

Examples

- a. $T(n) = 4T(n/2) + n;$ $\log_b a = 2. n = o(n^2) \Rightarrow$ Check case 1
- b. $T(n) = 9T(n/3) + n^2;$ $\log_b a = 2. n^2 = \Theta(n^2) \Rightarrow$ Check case 2
- c. $T(n) = 6T(n/4) + n;$ $\log_b a = 1.3. n = o(n^{1.3}) \Rightarrow$ Check case 1
- d. $T(n) = 2T(n/4) + n;$ $\log_b a = 0.5. n = \omega(n^{0.5}) \Rightarrow$ Check case 3
- e. $T(n) = T(n/2) + n \log n;$ $\log_b a = 0. n \log n = \omega(n^0) \Rightarrow$ Check case 3
- f. $T(n) = 4T(n/4) + n \log n.$ $\log_b a = 1. n \log n = \omega(n) \Rightarrow$ Check case 3

Some Tricks

- ◆ Changing variables
- ◆ Obtaining upper and lower bounds
 - ◆ Make a guess based on the bounds
 - ◆ Prove using the substitution method

Changing Variables

$$T(n) = 2T(n-1) + 1$$

◆ Let $n = \lg m$, i.e., $m = 2^n$

$$\Rightarrow T(\lg m) = 2 T(\lg (m/2)) + 1$$

◆ Let $S(m) = T(\lg m) = T(n)$

$$\Rightarrow S(m) = 2S(m/2) + 1$$

$$\Rightarrow S(m) = \Theta(m)$$

$$\Rightarrow T(n) = S(m) = \Theta(m) = \Theta(2^n)$$

Changing Variables

$$T(n) = T(\sqrt{n}) + 1$$

◆ Let $n = 2^m$

$$\Rightarrow \text{sqrt}(n) = 2^{m/2}$$

◆ We then have $T(2^m) = T(2^{m/2}) + 1$

◆ Let $T(n) = T(2^m) = S(m)$

$$\Rightarrow S(m) = S(m/2) + 1$$

$$\Rightarrow S(m) = \Theta(\log m) = \Theta(\log \log n)$$

$$\Rightarrow T(n) = \Theta(\log \log n)$$

Changing Variables

- ◆ $T(n) = 2T(n-2) + 1$

- ◆ Let $n = \lg m$, i.e., $m = 2^n$

$$\Rightarrow T(\lg m) = 2 T(\lg m/4) + 1$$

- ◆ Let $S(m) = T(\lg m) = T(n)$

$$\Rightarrow S(m) = 2S(m/4) + 1$$

$$\Rightarrow S(m) = m^{1/2}$$

$$\Rightarrow T(n) = S(m) = (2^n)^{1/2} = (\text{sqrt}(2))^n \approx 1.4^n$$

Obtaining Bounds

Solve the Fibonacci variant:

$$T(n) = T(n-1) + T(n-2) + 1$$

$$\spadesuit T(n) \geq 2T(n-2) + 1 \quad [1]$$

$$\spadesuit T(n) \leq 2T(n-1) + 1 \quad [2]$$

$\spadesuit$ Solving [1], we obtain $T(n) \geq 1.4^n$

$\spadesuit$ Solving [2], we obtain $T(n) \leq 2^n$

$\spadesuit$ Actually, $T(n) \approx 1.62^n$

Obtaining Bounds

- ◆ $T(n) = T(n/2) + \log n$
- ◆ $T(n) \in \Omega(\log n)$
- ◆ $T(n) \in O(T(n/2) + n^\varepsilon)$
- ◆ Solving $T(n) = T(n/2) + n^\varepsilon$,
we obtain $T(n) = O(n^\varepsilon)$, for any $\varepsilon > 0$
- ◆ So: $T(n) \in O(n^\varepsilon)$ for any $\varepsilon > 0$
 - ◆ $T(n)$ is unlikely polynomial
 - ◆ Actually, $T(n) = \Theta(\log^2 n)$ by extended case 2

Extended Case 2

CASE 2: $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$.

Extended CASE 2: ($k \geq 0$)

$f(n) = \Theta(n^{\log_b a} \log^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Solving Recurrences

1. Recursion tree / iteration method
 - Good for guessing an answer
 - Need to **verify** guess
2. Master method
 - Easy to learn, useful in **limited cases** only
 - Some tricks may help in other cases
3. Substitution method
 - Generic method, rigid, may be hard

The Substitution Method



Substitutions



<i>For</i>	<i>Use</i>
Buttermilk - 1 cup	1 TB lemon juice + enough milk to = 1 cup
Whole Milk - 1 cup	½ c. evaporated milk + ½ c. water
Unsweetened Chocolate - 1 oz	1 TB fat + 3 TB cocoa
Honey - 1 cup	¼ c. liquid + 1 ¼ c. sugar
Shortening (for baking) - 1 cup	1 1/8 c. butter or margarine less ½ tsp of salt in recipe
Corn Syrup - 1 cup	1 c. sugar + ¼ c. of liquid
Cornstarch - 1 ½ tsp	1 TB flour
1 whole egg	2 egg yolks + 1 TB water
Peppermint extract - 1 TB	¼ c. fresh mint, chopped
Cream ½ & ½ - 1 cup	3 TB oil + milk to = 1 cup
Cream, heavy for baking & cooking - 1 cup	¾ c. milk + ½ c. butter or margarine
Marshmallow Creme - 1 cup (jar = 2 1/8 cups)	16 lg (160 sm) marshmallows + 2 TB corn syrup (melted in double broiler)
Catsup	1 c. tomato sauce, ½ c. sugar, 2 TB vinegar

Substitution Method

The most general method to solve a recurrence (prove O and Ω separately):

- 1. *Guess*** the form of the solution
(e.g. by recursion tree / iteration method)
- 2. *Verify*** by induction (inductive step).
- 3. *Solve*** for O/Ω -constants n_0 and c (base cases of induction)

Substitution Method

By log we
mean lg

- ◆ Recurrence: $T(n) = 2T(n/2) + n$.
- ◆ Guess: $T(n) = O(n \log n)$. (e.g., by recursion tree method)
- ◆ To prove, have to show $T(n) \leq c n \log n$ for some $c > 0$ and for all $n > n_0$
- ◆ Proof by induction: assume it is true for $T(n/2)$, prove that it is also true for $T(n)$. This means:

- ◆ Given: $T(n) = 2T(n/2) + n$
- ◆ Need to Prove: $T(n) \leq c n \log (n)$
- ◆ Assuming: $T(n/2) \leq cn/2 \log (n/2)$

Proof

- ◆ Given: $T(n) = 2T(n/2) + n$
- ◆ Need to Prove: $T(n) \leq c n \log (n)$
- ◆ Assuming: $T(n/2) \leq cn/2 \log (n/2)$

- ◆ *Proof:*

Substituting $T(n/2) \leq cn/2 \log (n/2)$ into the recurrence, we get

$$\begin{aligned} T(n) &= 2 T(n/2) + n \\ &\leq cn \log (n/2) + n \\ &\leq c n \log n - c n + n \\ &\leq c n \log n - (c - 1) n \\ &\leq c n \log n \text{ for all } n > 0 \text{ (if } c \geq 1\text{).} \end{aligned}$$

Therefore, by definition, $T(n) = O(n \log n)$.

Substitution method – Example 2

- ◆ Recurrence: $T(n) = 2T(n/2) + n$.
- ◆ Guess: $T(n) = \Omega(n \log n)$.
- ◆ To prove, have to show $T(n) \geq c n \log n$ for some $c > 0$ and for all $n > n_0$
- ◆ Proof by induction: assume it is true for $T(n/2)$, prove that it is also true for $T(n)$. This means:

- ◆ Given: $T(n) = 2T(n/2) + n$
- ◆ Need to Prove: $T(n) \geq c n \log (n)$
- ◆ Assuming: $T(n/2) \geq cn/2 \log (n/2)$

Proof

- ◆ Given: $T(n) = 2T(n/2) + n$
- ◆ Need to Prove: $T(n) \geq c n \log (n)$
- ◆ Assuming: $T(n/2) \geq cn/2 \log (n/2)$

- ◆ *Proof:*

Substituting $T(n/2) \geq cn/2 \log (n/2)$ into the recurrence, we get

$$\begin{aligned} T(n) &= 2 T(n/2) + n \\ &\geq cn \log (n/2) + n \\ &\geq c n \log n - c n + n \\ &\geq c n \log n + (1 - c) n \\ &\geq c n \log n \text{ for all } n > 0 \text{ (if } c \leq 1). \end{aligned}$$

Therefore, by definition, $T(n) = \Omega(n \log n)$.

More Substitution Examples [1]

- ◆ Prove that $T(n) = 3T(n/3) + n = O(n \log n)$
- ◆ Need to show that $T(n) \leq c n \log n$ for some c , and sufficiently large n
- ◆ Assume above is true for $T(n/3)$, i.e.
$$T(n/3) \leq cn/3 \log (n/3)$$

3-way Merge Sort

$$\begin{aligned}
T(n) &= 3 T(n/3) + n \\
&\leq 3 cn/3 \log (n/3) + n \\
&\leq cn \log n - cn \log 3 + n \\
&\leq cn \log n - (cn \log 3 - n) \\
&\leq cn \log n \text{ (if } cn \log 3 - n \geq 0)
\end{aligned}$$

$$\begin{aligned}
&cn \log 3 - n \geq 0 \\
\Rightarrow &c \log 3 - 1 \geq 0 \text{ (for } n > 0) \\
\Rightarrow &c \geq 1/\log 3 \\
\Rightarrow &c \geq \log_3 2
\end{aligned}$$

Therefore, $T(n) = 3 T(n/3) + n \leq cn \log n$ for $c = \log_3 2$ and $n > 0$. By definition, $T(n) = O(n \log n)$.

More Substitution Examples [2]

- ◆ Prove that $T(n) = T(n/3) + T(2n/3) + n = O(n \log n)$
- ◆ Need to show that $T(n) \leq c n \log n$ for some c , and sufficiently large n
- ◆ Assume above is true for $T(n/3)$ and $T(2n/3)$, i.e.

$$T(n/3) \leq cn/3 \log (n/3)$$

$$T(2n/3) \leq 2cn/3 \log (2n/3)$$

Unbalanced Merge Sort

$$\begin{aligned}
T(n) &= T(n/3) + T(2n/3) + n \\
&\leq cn/3 \log(n/3) + 2cn/3 \log(2n/3) + n \\
&\leq cn \log n + n - cn (\log 3 - 2/3) \\
&\leq cn \log n + n(1 - c \log 3 + 2c/3) \\
&\leq cn \log n, \text{ for all } n > 0 \text{ (if } 1 - c \log 3 + 2c/3 \leq 0)
\end{aligned}$$

$$\begin{aligned}
&c \log 3 - 2c/3 \geq 1 \\
\Rightarrow c &\geq 1 / (\log 3 - 2/3) > 0
\end{aligned}$$

Therefore, $T(n) = T(n/3) + T(2n/3) + n \leq cn \log n$ for $c = 1 / (\log 3 - 2/3)$ and $n > 0$. By definition, $T(n) = O(n \log n)$.

More Substitution Examples [3]

- ◆ Prove that $T(n) = 3T(n/4) + n^2 = O(n^2)$
- ◆ Need to show that $T(n) \leq c n^2$ for some c , and sufficiently large n
- ◆ Assume above is true for $T(n/4)$, i.e.
$$T(n/4) \leq c(n/4)^2 = cn^2/16$$

$$\begin{aligned}
T(n) &= 3T(n/4) + n^2 \\
&\leq 3c n^2 / 16 + n^2 \\
&\leq (3c/16 + 1) n^2 \\
&\stackrel{?}{\leq} cn^2
\end{aligned}$$

$3c/16 + 1 \leq c$ implies that $c \geq 16/13$

Therefore, $T(n) = 3(n/4) + n^2 \leq cn^2$ for $c = 16/13$ and all n . By definition, $T(n) = O(n^2)$.

Avoiding Pitfalls

- ◆ Guess $T(n) = 2T(n/2) + n = O(n)$ [really $O(n \log n)$]
- ◆ Need to prove that $T(n) \leq c n$
- ◆ Assume $T(n/2) \leq cn/2$
- ◆ $T(n) \leq 2 * cn/2 + n = cn + n = O(n)$
- ◆ What's wrong?
- ◆ Need to prove $T(n) \leq cn$, not $T(n) \leq cn + n = (c+1)n$

Subtleties

- ◆ Prove that $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 = O(n)$
- ◆ Need to prove that $T(n) \leq cn$
- ◆ Assume above is true for $T(\lfloor n/2 \rfloor)$ & $T(\lceil n/2 \rceil)$

$$\begin{aligned} T(n) &\leq c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 \\ &\leq cn + 1 \end{aligned}$$

Is it a correct proof?

No! have to prove $T(n) \leq cn$

However we can prove $T(n) = O(n - 1)$

Making a Good Guess

$$T(n) = 2T(n/2 + 17) + n$$

When n approaches infinity, $n/2 + 17$ are not too different from $n/2$

Therefore can guess $T(n) = \Theta(n \log n)$

Prove Ω :

Assume $T(n/2 + 17) \geq c (n/2 + 17) \log (n/2 + 17)$

Then we have

$$\begin{aligned} T(n) &= n + 2T(n/2 + 17) \\ &\geq n + 2c (n/2 + 17) \log (n/2 + 17) \\ &\geq n + c n \log (n/2 + 17) + 34 c \log (n/2 + 17) \\ &\geq c n \log (n/2 + 17) + 34 c \log (n/2 + 17) \end{aligned}$$

....

Maybe can guess $T(n) = \Theta((n-17) \log (n-17))$ (trying to get rid of the +17).

Details skipped.

Summary: Solving Recurrences

1. Recursion tree / iteration method
 - Good for guessing an answer
2. Master method
 - Easy to learn, useful in limited cases only
 - Some tricks may help in other cases
3. Substitution method
 - Generic method, rigid, may be hard

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

subproblems at level i of recursion tree
 3^i

Size of subproblems

$$\text{total work} = \sum_{i=0}^{\log_2 n} 3^i \frac{n}{2^i} = n \sum_{i=0}^{\log_2 n} \left(\frac{3}{2}\right)^i$$

$$= n O\left(\left(\frac{3}{2}\right)^{\log_2 n}\right) = \frac{n}{n} O(3^{\log_2 n})$$

$$= O(n^{\log_2 3})$$

$$a^{\log_b n} = n^{\log_b a}$$