Asymptotic Notation

Time Complexity and Space Complexity

- Generally, there is always more than one way to solve a problem in computer science with different algorithms.
- Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal.
- The method must be:
  - Independent of the machine and its configuration, on which the algorithm is running on.
  - Shows a direct correlation with the number of inputs.
  - Can distinguish two algorithms clearly without ambiguity.
- There are two such methods used, **time complexity** and **space complexity**

Time Complexity

- **Time Complexity:** The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- In order to calculate time complexity on an algorithm, it is assumed that a **constant time c** is taken to execute one operation, and then the total operations for an input length on **N** are calculated

Why Time complexity is IMP??

- For example, if we have 4 billion elements to search for, then, in its worst case, linear search will take 4 billion operations to complete its task.
- Binary search will complete this task in just 32 operations. That's a big difference.
- Now let's assume that if one operation takes 1 ms for completion, then binary search will take only 32 ms whereas linear search will take 4 billion ms (that is approx. 46 days). That's a significant difference.

https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/

What is meant by the Time Complexity of an Algorithm?

- *Instead of measuring actual time required in executing each statement in the code, **Time Complexity considers how many times each statement executes.***
- **Example 1:** Consider the below simple code to print Hello World

```
#include <iostream>
using namespace std;
 int main()
{
   cout << "Hello World";
   return 0;
}
```

- **Output**Hello World

- **Time Complexity:** In the above code “Hello World” is printed **only once** on the screen.
  So, the time complexity is **constant: O(1)** i.e. every time a constant amount of time is required to execute code, no matter which operating system or which machine configurations you are using.

*

```
#include <iostream>
using namespace std;
int main()
{
    int i, n = 8;
   for (i = 1; i <= n; i++) {
      cout << "Hello World !!!\n";
   }
   return 0;
}
```

- **Time Complexity:** In the above code “Hello World !!!” is printed only **n times** on the screen, as the value of n can change.
  So, the time complexity is **linear: O(n)**

```
#include <iostream>
using namespace std;
 int main()
{
     int i, n = 8;
    for (i = 1; i <= n; i=i*2) {
       cout << "Hello World !!!\n";
    }
    return 0;
}
```

- In the above code “Hello World !!!” is printed only **4 times** on the screen
- **Time Complexity:** $O(\log_2(n))$

- Binary search is an example with complexity O(log n).
- Binary search is a divide-and-conquer algorithm, and we will need (at most) 4 comparisons to find the record we are searching for in **16 item dataset**.
- Assume we had instead a dataset with **32 elements**.
- we will now need 5 comparisons to find what we are searching for.
- As a result, the complexity of the algorithm can be described as a logarithmic order.

- In those cases, the number of times you can divide a **data input** (e.g. list, array, etc…) of length **n** in half before you get down to single-element arrays is $\log_2 n$.
- and in computer science, exponential growth usually happens as a consequence of discrete processes like the divide-and-conquer

Space Complexity

- **Space Complexity:** The **space complexity** of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input.
- Space complexity is a parallel concept to time complexity.
- If we need to create an array of size n, this will require O(n) space.
- If we create a two-dimensional array of size n*n, this will require O(n^2) space

Asymptotic Complexity

- Running time of an algorithm as a function of input size n **for large n**.
- Expressed using only the **highest-order term** in the expression for the exact running time.
  - Instead of exact running time, say $\Theta(n^2)$.
- Describes behavior of function in the limit.
- Written using ***Asymptotic Notation***.

Asymptotic Notation

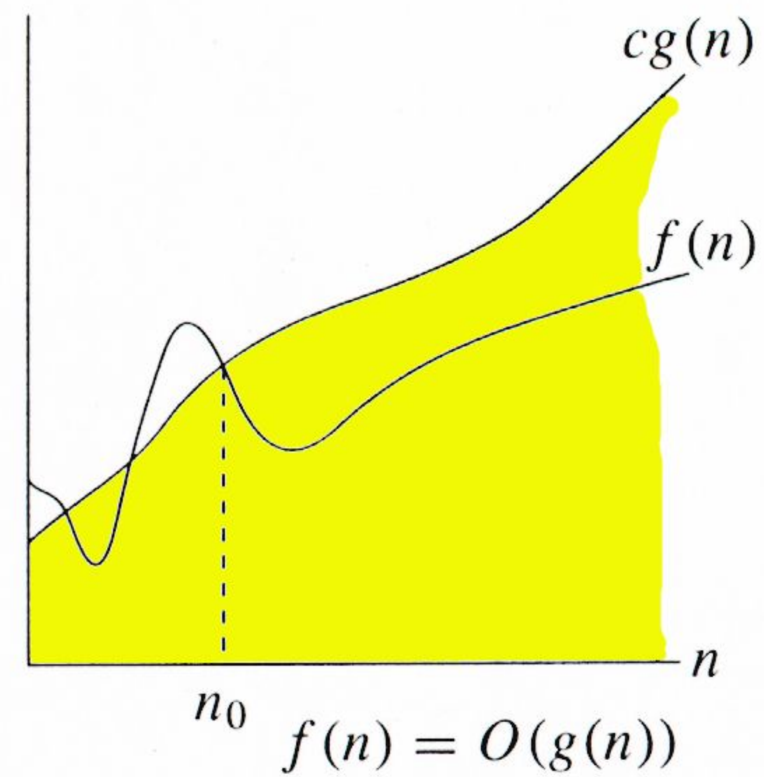
- Θ**,** ***O*****,** Ω**,** ***o*****,** ω
- Defined for functions over the natural numbers.
  - **Ex:** $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.
- Define a ***set*** of functions; in practice used to compare two function sizes.
- The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

O-notation

O-notation is an upper-bound notation.

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n) :$
$\exists$ **positive constants c and n_0,**
such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq cg(n)$ $\}$



Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

$g(n)$ is an *asymptotic upper bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$.

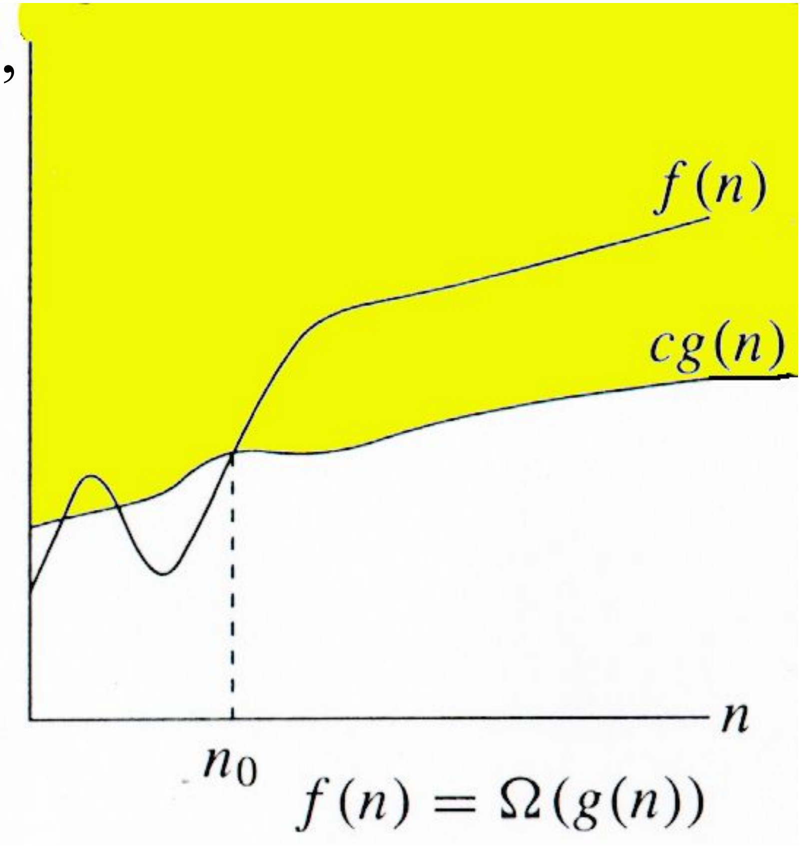
$\Theta(g(n)) \subset O(g(n))$.

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n) :$
$\exists$ positive constants c and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.



$g(n)$ is an *asymptotic lower bound* for $f(n)$.

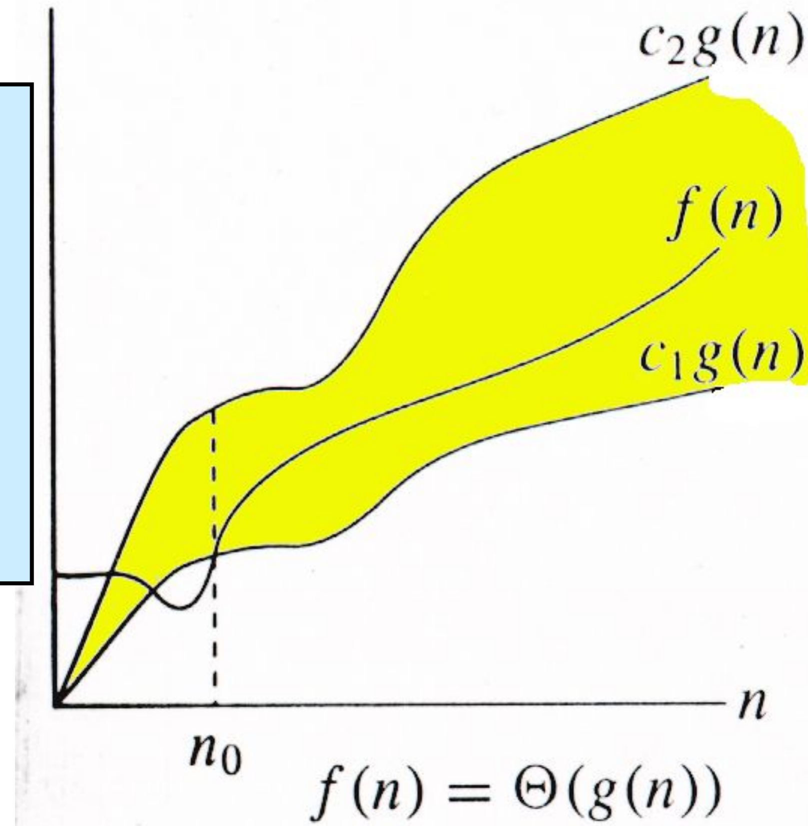
$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$.

$\Theta(g(n)) \subset \Omega(g(n))$.

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n) :$
$\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$



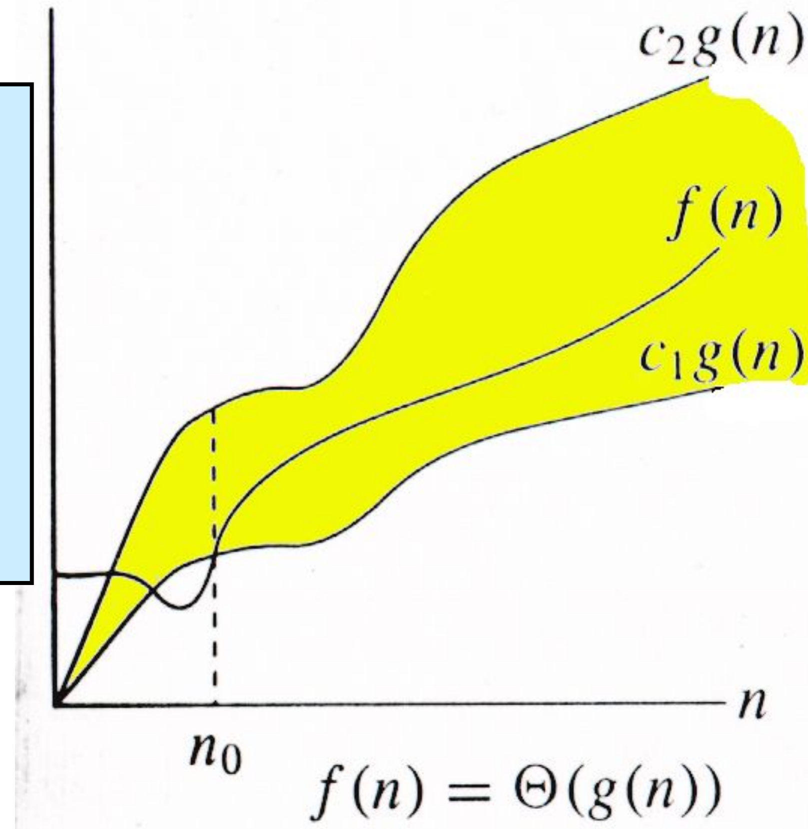
Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

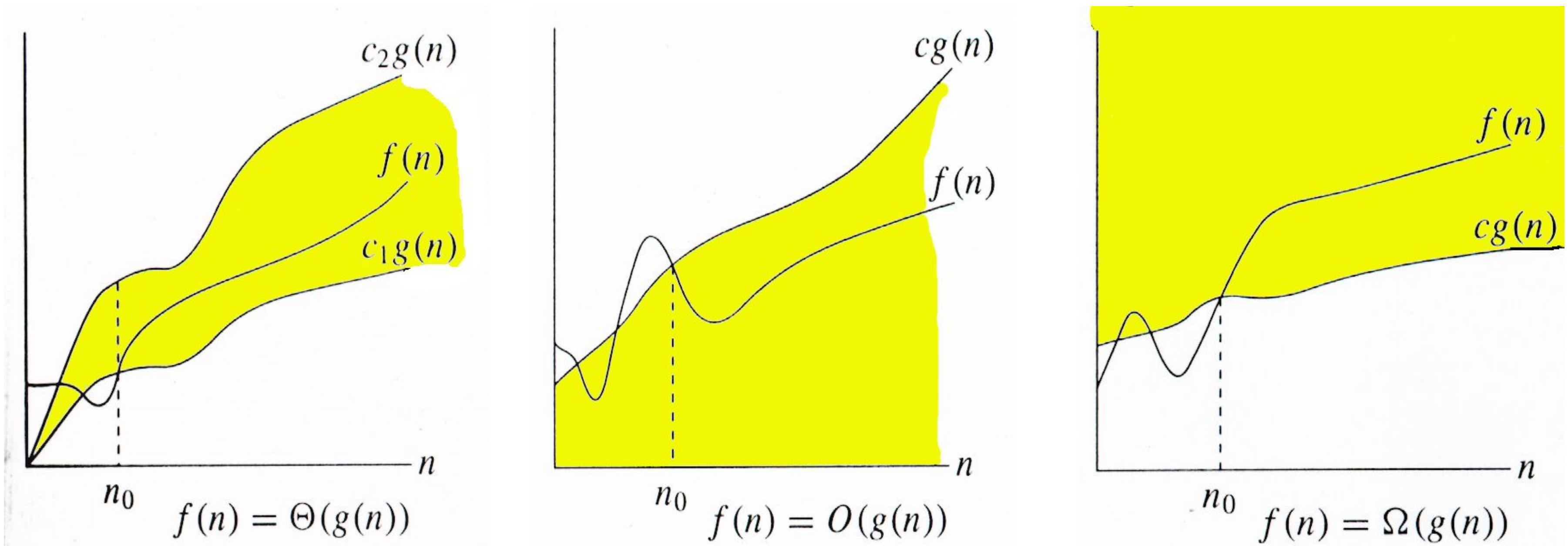
$\Theta(g(n)) = \{f(n) :$
$\exists$ positive constants c_1, c_2, and n_0, such that $\forall\, n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$

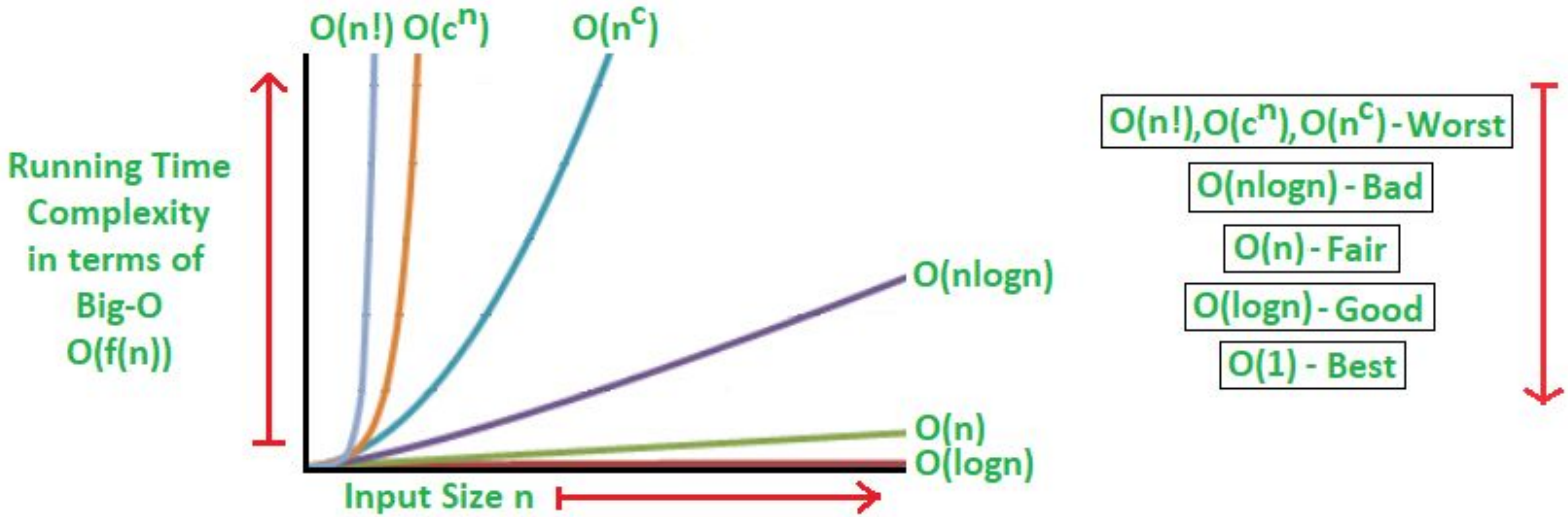


Technically, $f(n) \in \Theta(g(n))$.
Older usage, $f(n) = \Theta(g(n))$.
I'll accept either…

$f(n)$ and $g(n)$ are nonnegative, for large n.

Relations Between Θ, *O*, Ω




O(n!) O(c^n)
O(n^c)
Running Time Complexity in terms of Big-O O(f(n))
O(nlogn)
O(n)
O(logn)
Input Size n
O(n!),O(c^n),O(n^c) - Worst
O(nlogn) - Bad
O(n) - Fair
O(logn) - Good
O(1) - Best

*

Number of elements	Simple search	Binary search
The run time in Big O notation	O(n)	O(log n)
10	10 ms	3 ms
100	100 ms	7 ms
10.000	10 sec	14 ms
1000.000.000	11 days	32 ms

*

Relations Between Θ, Ω, O

Theorem : For any two functions $g(n)$ and $f(n)$,

$\boldsymbol{f(n) = \Theta(g(n))}$ iff

$\boldsymbol{f(n) = O(g(n))}$ **and** $\boldsymbol{f(n) = \Omega(g(n))}$.

- I.e., $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- In practice, asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

o-notation and ω-notation

- O-notation and Ω-notation are like ≤ and ≥. o-notation and ω-notation are like < and >.
- $o(g(n)) = \{ f(n)$: for any constant $c > 0$, there is a constant $n0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n0 \}$
- Example: $2n2 = o(n3)$

- $\omega(g(n)) = \{ f(n)$: for any constant $c > 0$, there is a constant $n0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n0 \}$
- EXAMPLE: $n = \omega(\lg n)$

Running Times

- "Running time is $O(f(n))$" $\Rightarrow$ Worst case is $O(f(n))$
- $O(f(n))$ bound on the worst-case running time $\Rightarrow$ $O(f(n))$ bound on the running time of every input.
- $\Theta(f(n))$ bound on the worst-case running time $\not\Rightarrow$ $\Theta(f(n))$ bound on the running time of every input.
- "Running time is $\Omega(f(n))$" $\Rightarrow$ Best case is $\Omega(f(n))$
- Can still say "Worst-case running time is $\Omega(f(n))$"
  - Means worst-case running time is given by some unspecified function $g(n) \in \Omega(f(n))$.

Example

- ***Insertion sort*** takes $\Theta(n^2)$ in the worst case, so sorting (as a *problem*) is $O(n^2)$. **Why?**

- Any sort algorithm must look at each item, so sorting is $\Omega(n)$.

- In fact, using (e.g.) merge sort, sorting is $\Theta(n \lg n)$ in the worst case.
  - Later, we will prove that we cannot hope that any comparison sort to do better in the worst case.

Asymptotic Notation in Equations

- Can use asymptotic notation in equations to replace expressions containing lower-order terms.
- For example,

  $4n^3 + 3n^2 + 2n + 1 = 4n^3 + 3n^2 + \Theta(n)$

  $= 4n^3 + \Theta(n^2) = \Theta(n^3)$. **How to interpret?**
- In equations, $\Theta(f(n))$ always stands for an ***anonymous function*** $g(n) \in \Theta(f(n))$
  - In the example above, $\Theta(n^2)$ stands for $3n^2 + 2n + 1$.

Properties

- **Transitivity**

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$
$$f(n) = o\,(g(n)) \;\&\; g(n) = o\,(h(n)) \Rightarrow f(n) = o\,(h(n))$$
$$f(n) = \omega(g(n)) \;\&\; g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

- **Reflexivity**

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

Properties

- **Symmetry**

  $f(n) = \Theta(g(n))$ *iff* $g(n) = \Theta(f(n))$

- **Complementarity**

  $f(n) = O(g(n))$ *iff* $g(n) = \Omega(f(n))$

  $f(n) = o(g(n))$ *iff* $g(n) = \omega((f(n))$

- https://www.geeksforgeeks.org/examples-of-big-o-analysis/
- https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

*

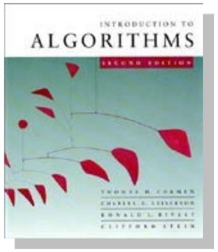
Recurrence

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- For example, the worst-case running time T (n) of the MERGE-SORT procedure by the recurrence
- $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,; \\ 2T(n/2) + \Theta(n) & \text{if } n>1; \end{cases}$

whose solution we claimed to be $T(n) = \Theta(n \lg n)$

- three methods for solving recurrences

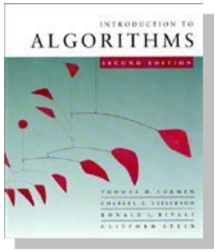
1. Substitution Method
2. Recurrence Tree
3. Master Theorem



Substitution method

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.



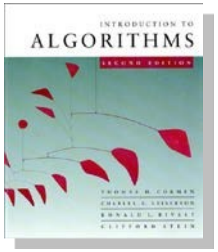
Substitution method

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

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

- [Assume that $T(1) = \Theta(1)$.]
- Guess $O(n^3)$. (Prove O and Ω separately.)
- Assume that $T(k) \le ck^3$ for $k < n$.
- Prove $T(n) \le cn^3$ by induction.



Example of substitution

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

$$\leq 4c(n/2)^3 + n$$

$$= (c/2)n^3 + n$$

$$= cn^3 - ((c/2)n^3 - n) \leftarrow$$ *desired – residual*

$$\leq cn^3$$ $\leftarrow$ *desired*

whenever $(c/2)n^3 - n \geq 0$, for example, if $c \geq 2$ and $n \geq 1$.

residual

Example 2

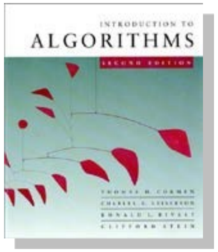
- $T(n) = 2T(n/2) + n$

  $<= 2cn/2Log(n/2) + n$

  $= cnLogn - cnLog2 + n$

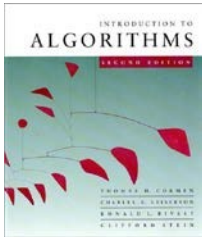
  $= cnLogn - cn + n$

  $<= cnLogn$



Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- The recursion-tree method promotes intuition, however.
- The recursion tree method is good for generating guesses for the substitution method.



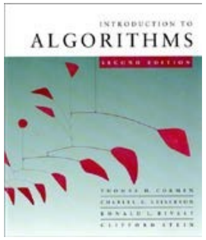
Recurrence for merge sort

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

- We shall 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

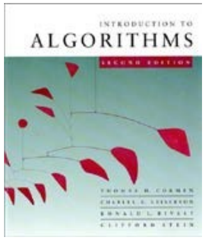
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

Recursion tree

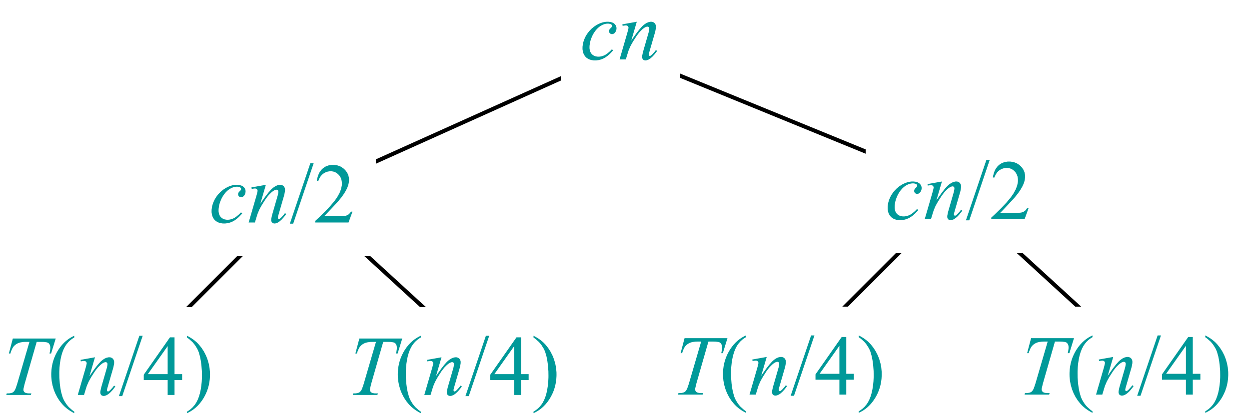
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

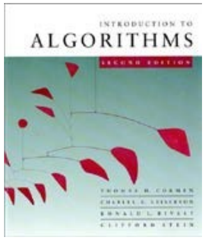
$$
\begin{array}{ccc}
 & cn & \\
\swarrow & & \searrow \\
T(n/2) & & T(n/2)
\end{array}
$$



Recursion tree

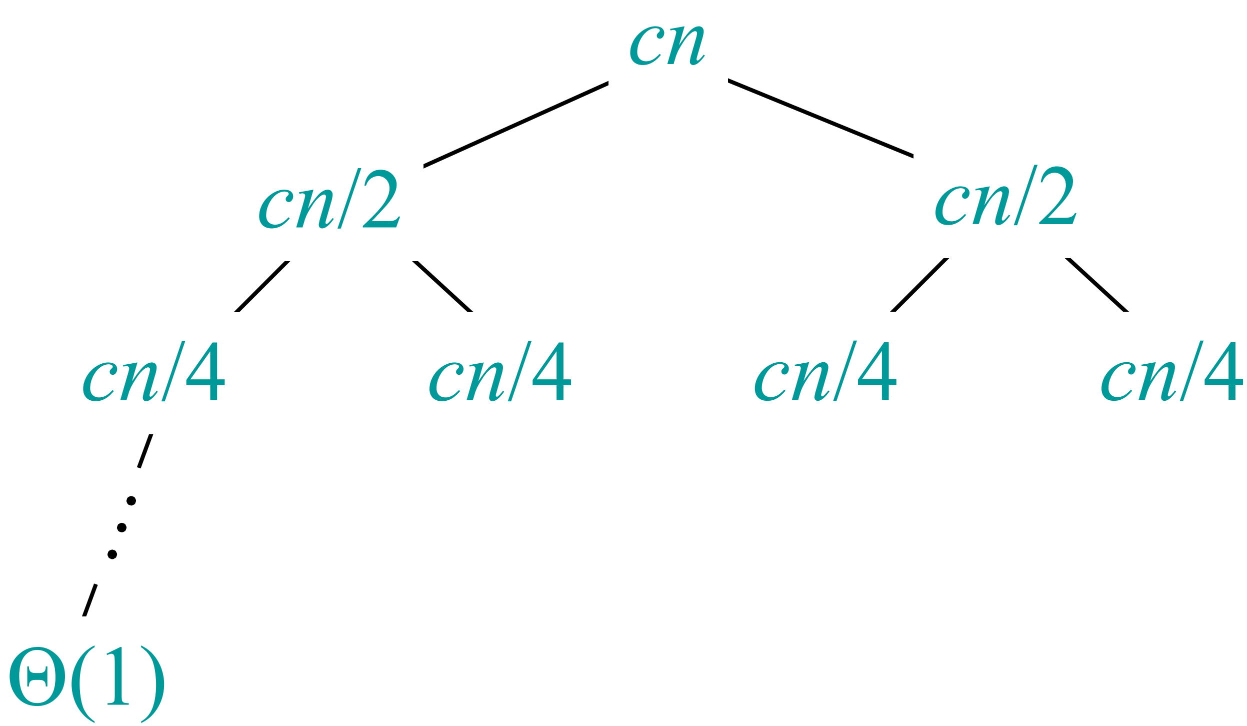
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

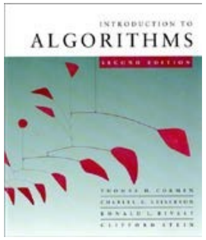




Recursion tree

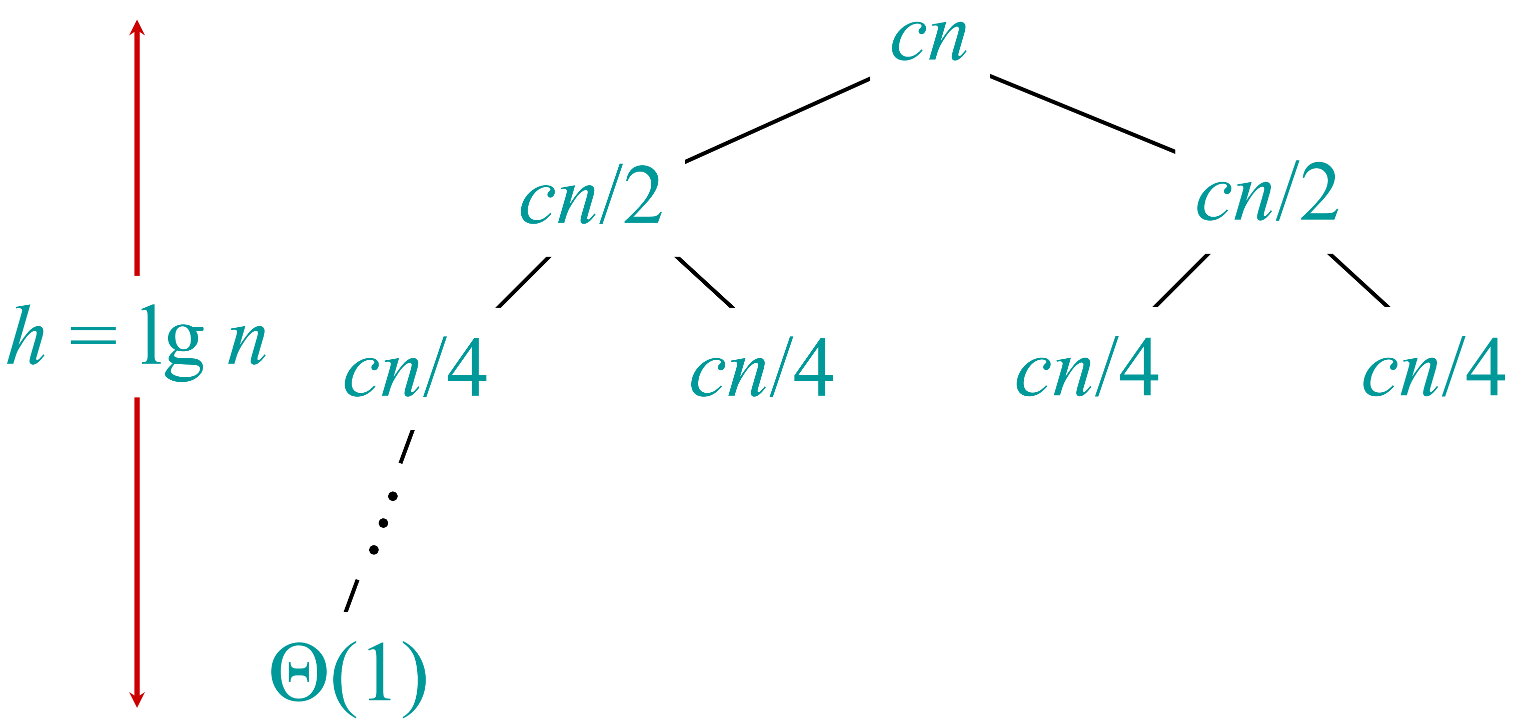
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

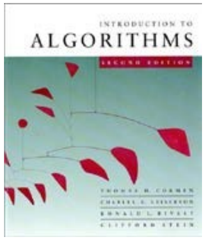




Recursion tree

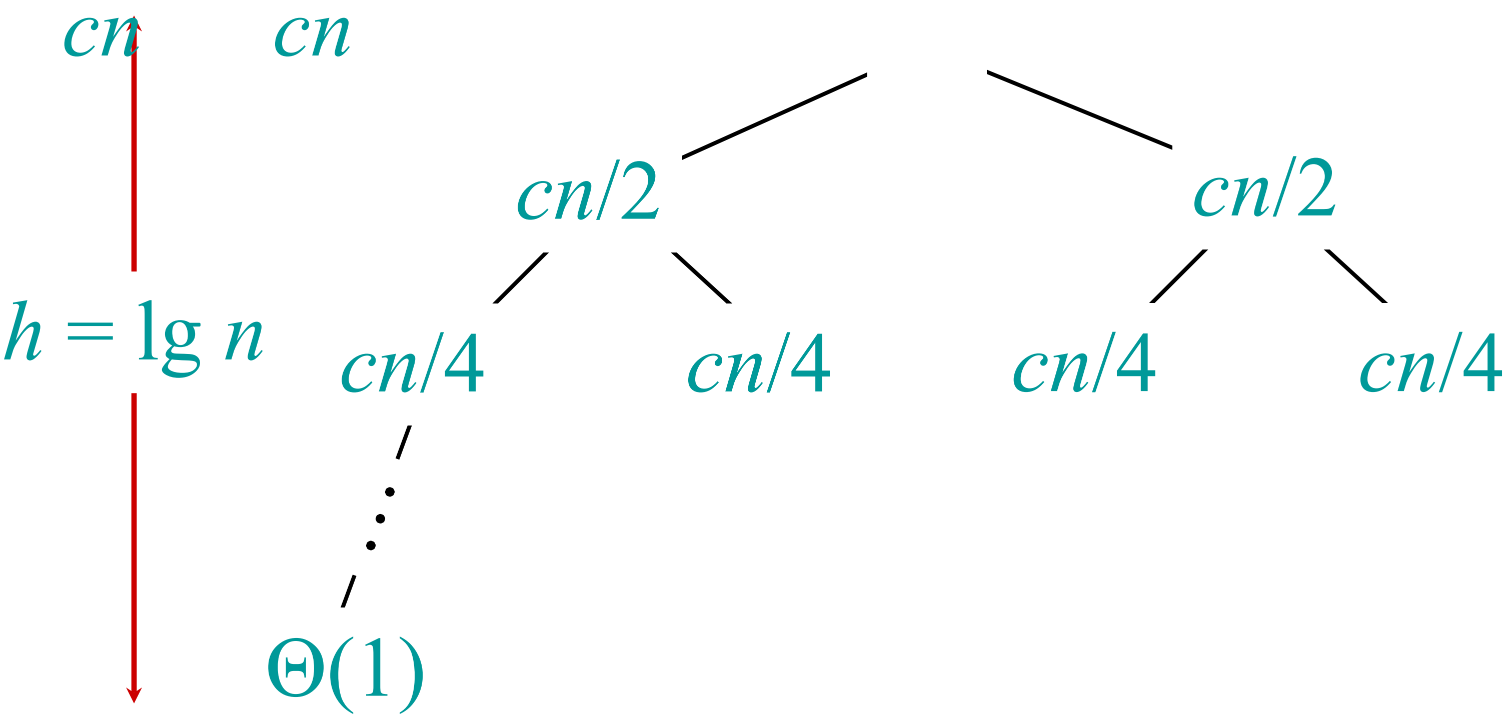
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

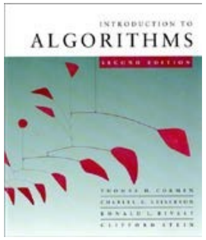




Recursion tree

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 $\quad\quad cn$

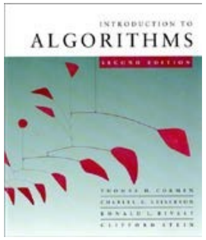
$cn/2 \quad\quad \cancel{cn/2} \quad cn$

$h = \lg n$

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

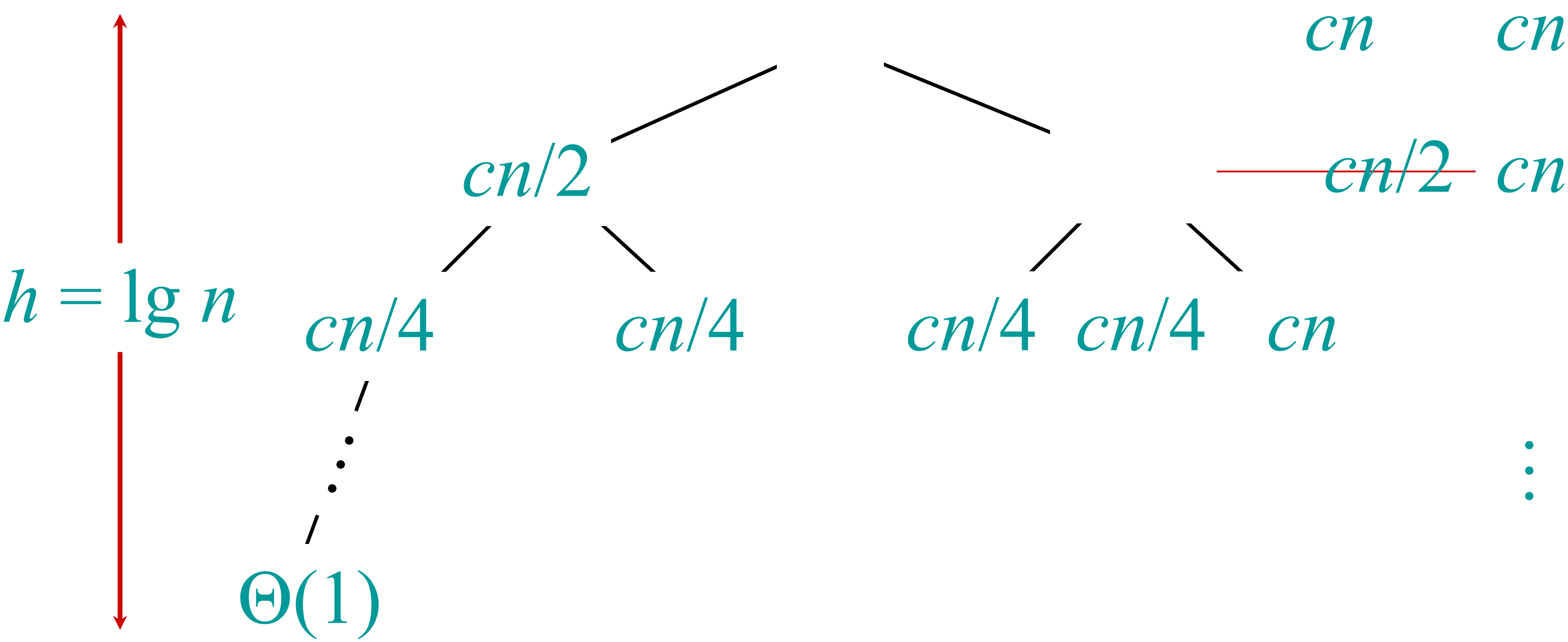
$\vdots$

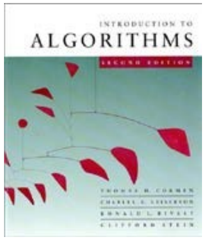
$\Theta(1)$



Recursion tree

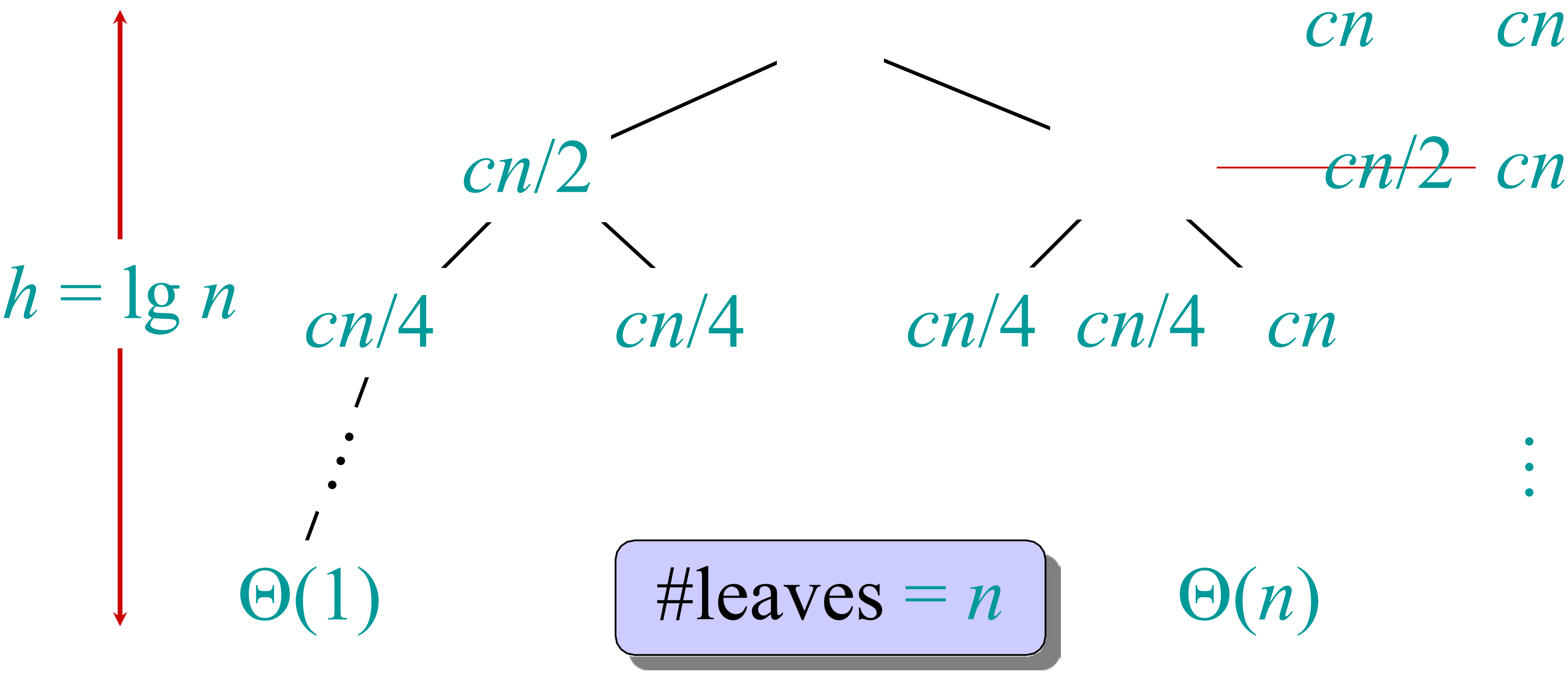
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

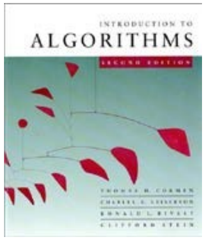




Recursion tree

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

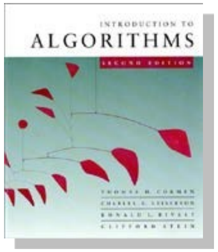
$cn/2$ $\quad$ $\cancel{cn/2}$ $\quad$ cn

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

$\vdots$

$\Theta(1)$ $\quad$ $\boxed{\#\text{leaves} = n}$ $\quad$ $\Theta(n)$

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



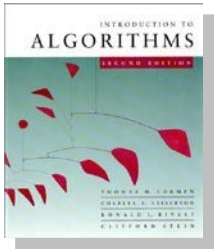
Example 2 of recursion tree

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

Example of recursion tree

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

$T(n)$

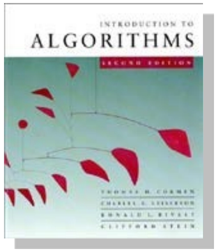


Example of recursion tree

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

$$n^2$$

$$T(n/4) \qquad\qquad T(n/2)$$



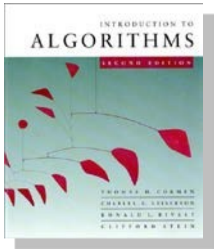
Example of recursion tree

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

$$n^2$$

$$(n/4)^2 \qquad\qquad (n/2)^2$$

$$T(n/16) \quad T(n/8) \qquad\qquad T(n/8) \quad T(n/4)$$



Example of recursion tree

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

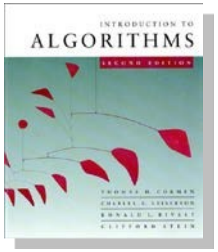
$$n^2$$

$$(n/4)^2 \qquad (n/2)^2$$

$$(n/16)^2 \qquad (n/8)^2 \qquad (n/8)^2 \qquad (n/4)^2$$

$$\vdots$$

$$\Theta(1)$$



Example of recursion tree

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

n^2 $n2$

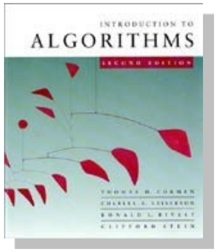
$(n/4)^2$ $(n/2)^2$

$(n/16)^2$ $(n/8)^2$ $(n/4)^2$

$(n/8)^2$

$\vdots$

$\Theta(1)$



Example of recursion tree

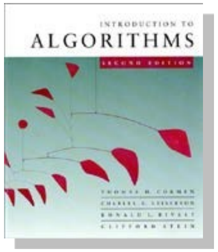
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

n^2 $n2$

$(n/4)^2$ $(n/2)^2$ $\frac{5}{16}n2$

$(n/16)^2$ $(n/8)^2$ $(n/8)^2$ $(n/4)^2$

$\Theta(1)$



Example of recursion tree

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

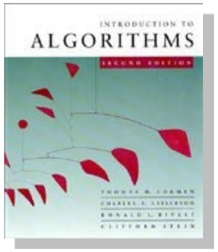
n^2 $\quad$ n^2

$(n/4)^2$ $\quad$ $(n/2)^2$ $\quad$ $\frac{5}{16}n^2$

$(n/16)^2$ $\quad$ $(n/8)^2$ $\quad$ $(n/8)^2$ $\quad$ $(n/4)^2$ $\quad$ $\frac{25}{256}n^2$

$\vdots$

$\Theta(1)$



Example of recursion tree

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

n^2 $\quad n^2$

$(n/4)^2 \qquad (n/2)^2 \qquad \frac{5}{16}n^2$

$(n/16)^2 \quad (n/8)^2 \quad (n/8)^2 \quad (n/4)^2 \qquad \frac{25}{256}n^2$

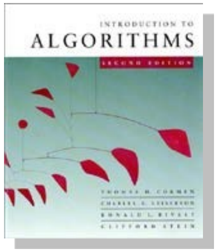
$\vdots \qquad \vdots$

$\Theta(1)$

$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

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

geometric series

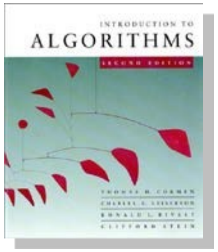


The master method

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.



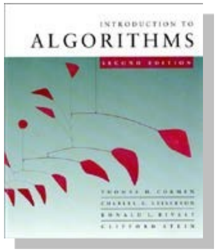
Examples

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

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

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

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



Examples

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

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

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

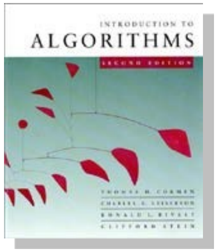
$\therefore T(n) = \Theta(n^2)$.

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

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

CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

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



Examples

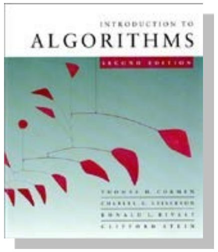
Ex. $T(n) = 4T(n/2) + n^3$

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

Case 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore T(n) = \Theta(n^3)$.



Examples

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

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

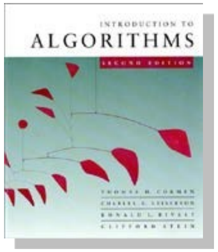
CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(n/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore T(n) = \Theta(n^3)$.

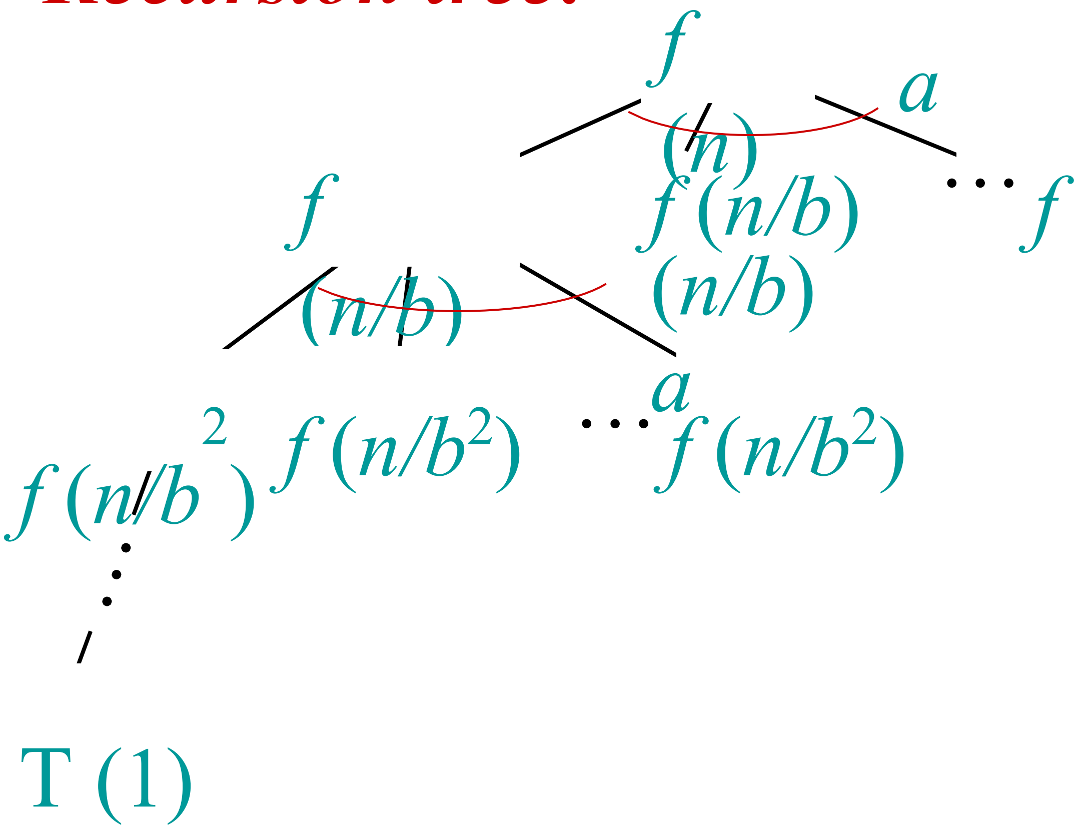
Ex. $T(n) = 4T(n/2) + n^2/\lg n$

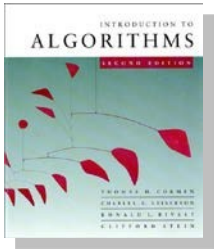
$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2/\lg n.$ Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\lg n)$.



Idea of master theorem

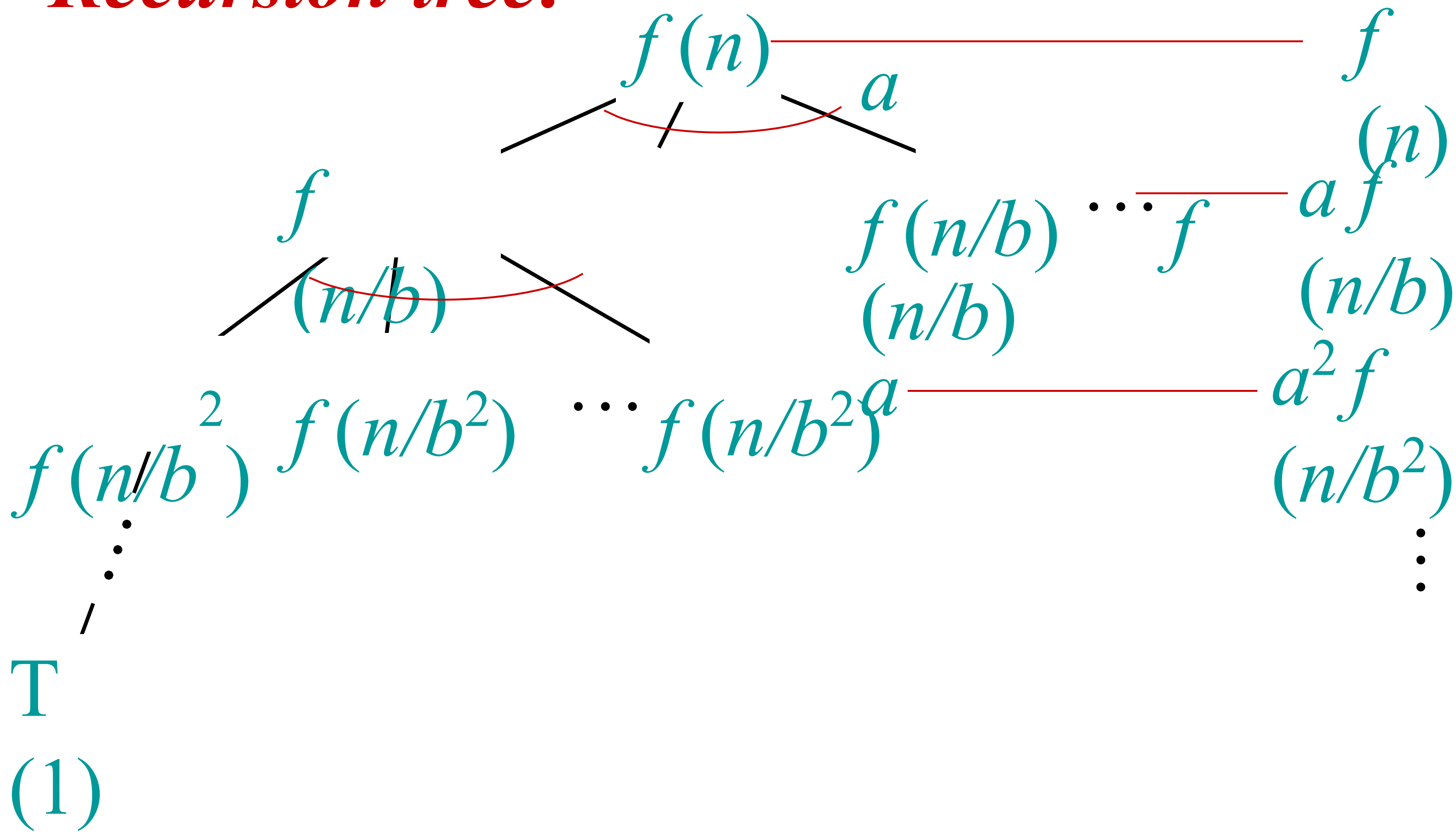
Recursion tree:

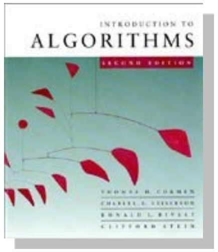




Idea of master theorem

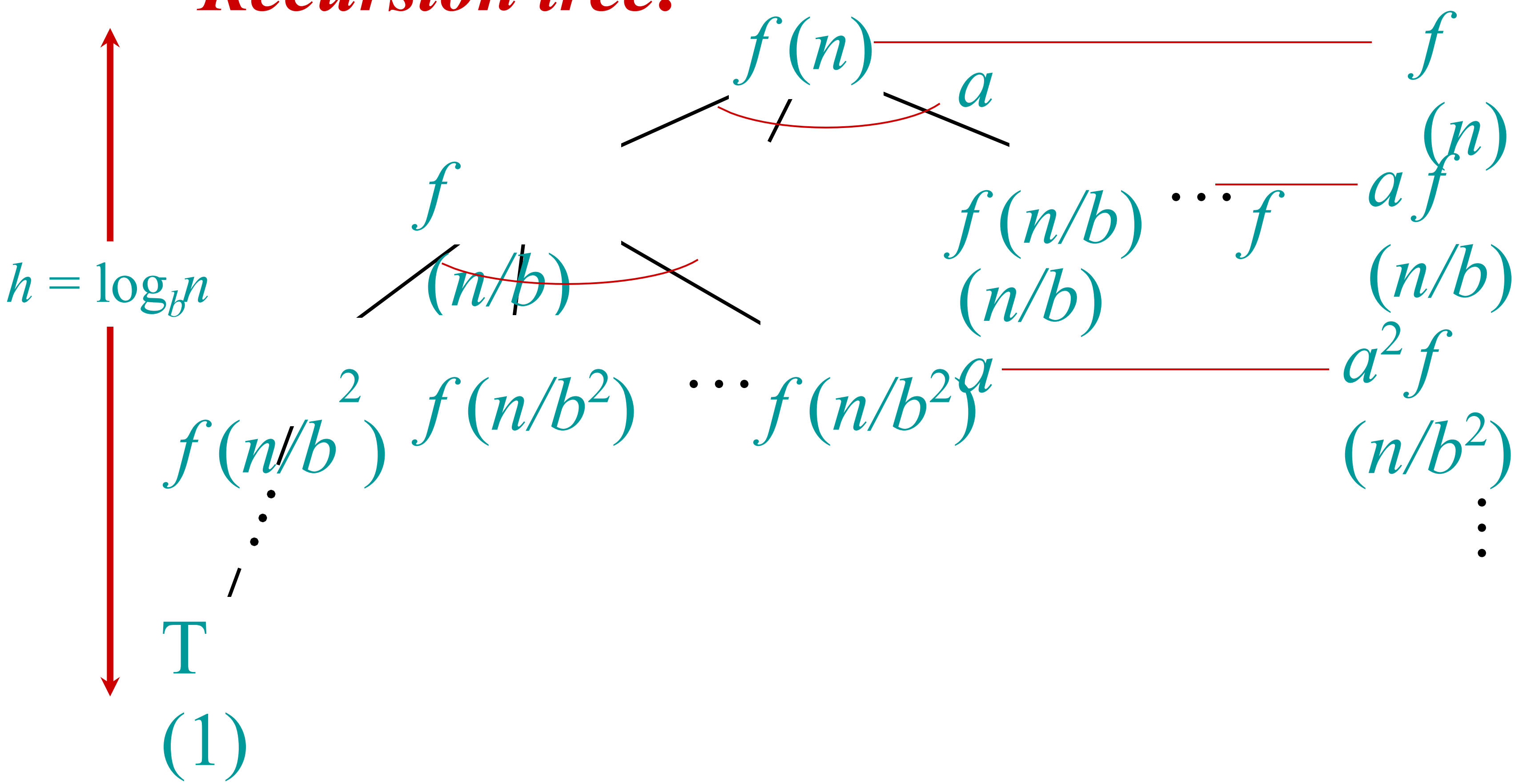
Recursion tree:

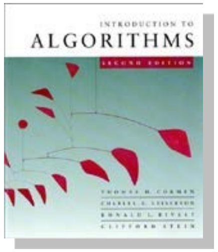




Idea of master theorem

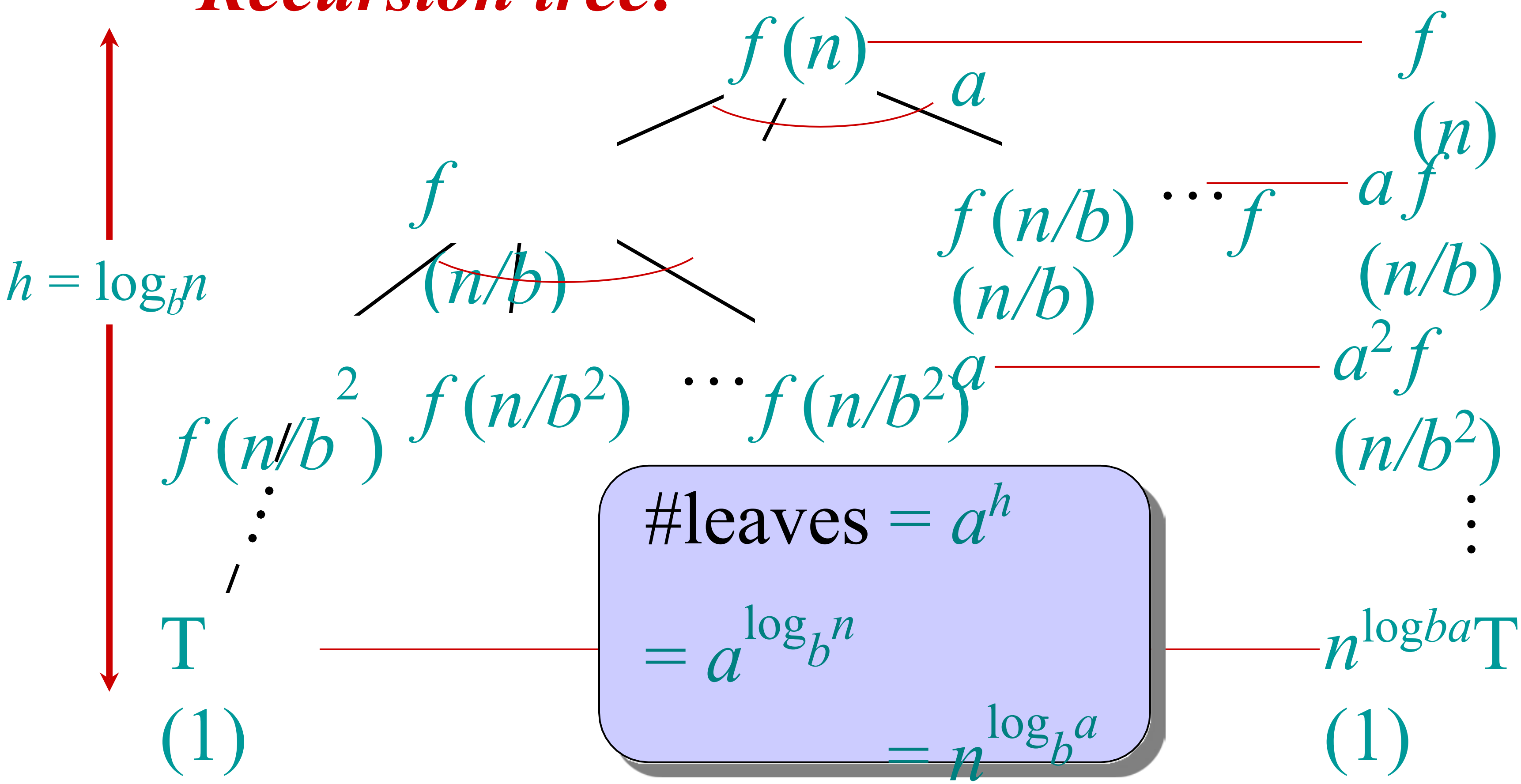
Recursion tree:

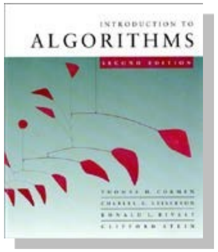




Idea of master theorem

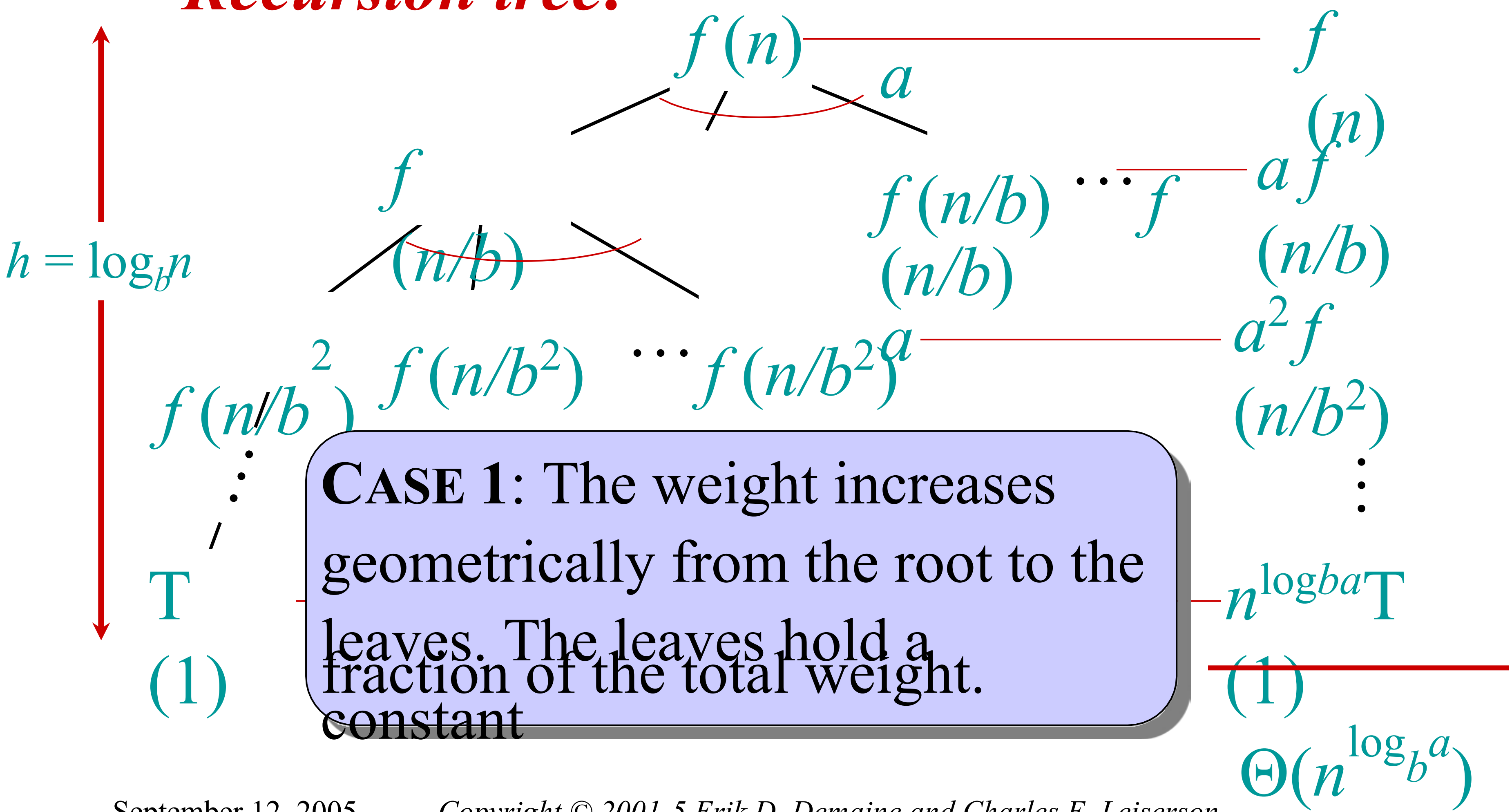
Recursion tree:



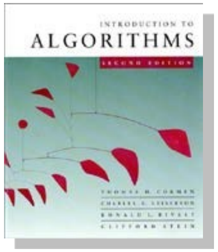


Idea of master theorem

Recursion tree:

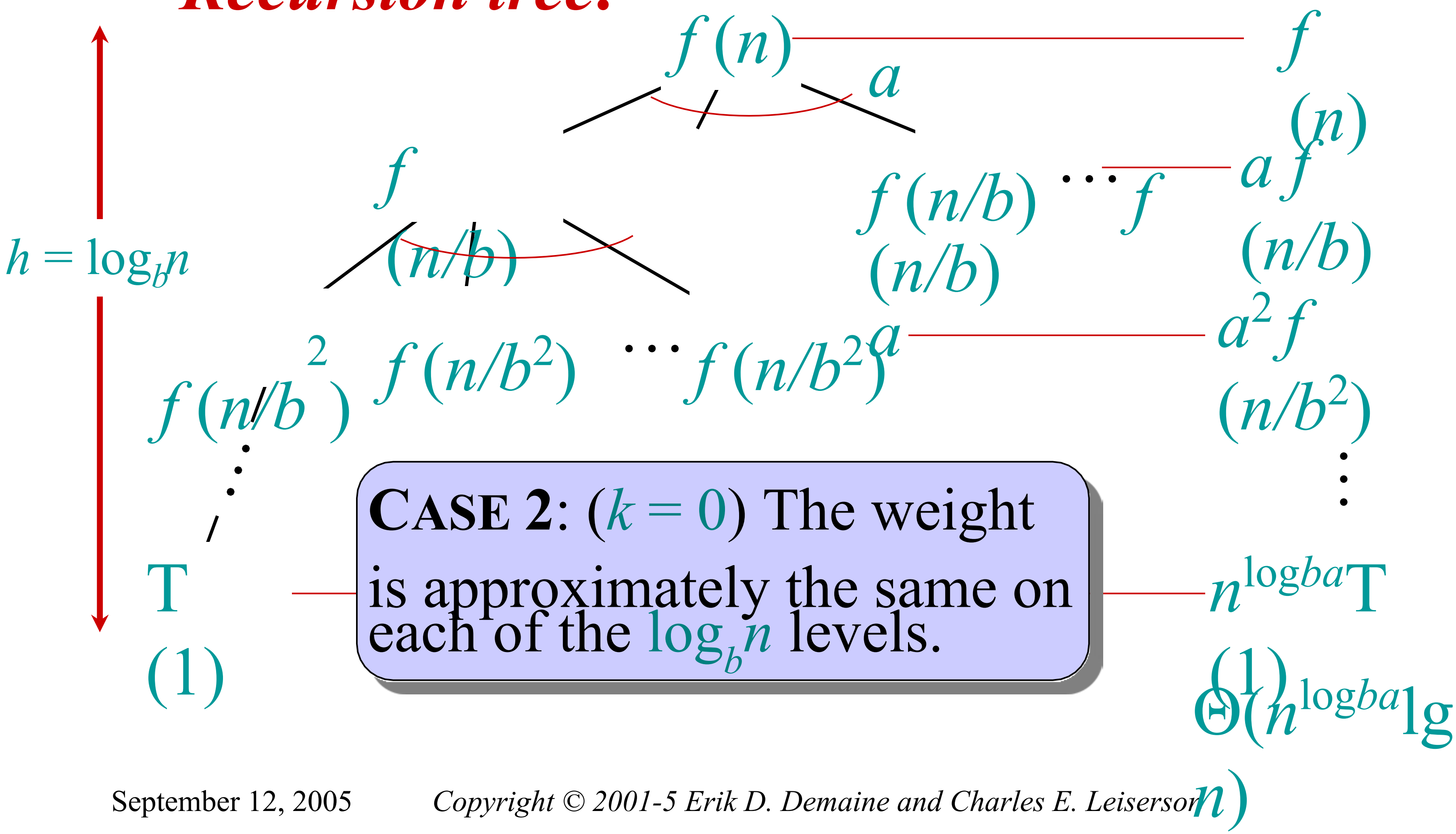


CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

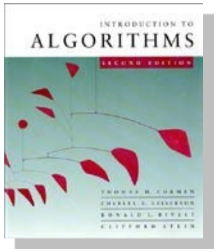


Idea of master theorem

Recursion tree:



CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels.



Idea of master theorem

Recursion tree:

$f(n)$ — a — $f(n)$

$f(n/b)$ $f(n/b)$ $\cdots$ $f(n/b)$ — $a\,f(n/b)$

$h = \log_b n$

$f(n/b^2)$ $f(n/b^2)$ $\cdots$ $f(n/b^2)$ — $a^2 f(n/b^2)$

$\vdots$

$T(1)$ — $n^{\log_b a} T(1)$

$\Theta(f(n))$

CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

Master Method cont..

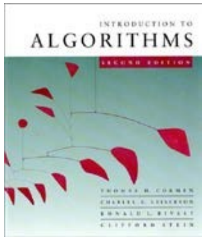
- The master method works only for the following type of recurrences or for recurrences that can be transformed into the following type.

- $T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

- If $f(n) = O(n^c)$ where $c < Log_b a$ then $T(n) = \Theta\ (n^{Logba})$
- If $f(n) = \Theta\ (n^c)$ where $c = Log_b a$ then $T(n) = \Theta\ (n^c Log\ n)$
- If $f(n) = \Omega\ (n^c)$ where $c > Log_b a$ then $T(n) = \Theta\ (f(n))$

- The master method is mainly derived from the recurrence tree method.
- If we draw the recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that
  - the work done at the root is $f(n)$,
  - work done at all leaves is ?(n^c) where c is $Log_b a$.
  - the height of the recurrence tree is $Log_b n$

- In the recurrence tree method, we calculate the total work done.
- If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1).
- If work done at leaves and root is asymptotically the same, then our result becomes height multiplied by work done at any level (Case 2).
- If work done at the root is asymptotically more, then our result becomes work done at the root (Case 3).



Conclusions

- $\Theta(n \lg n)$ 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.
- Go test it out for yourself!