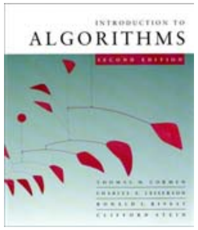
CS 5633 -- Spring 2005



Recurrences and Divide & Conquer

Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk

Merge sort

```
MERGE-SORT $A[1 \,..\, n]$
    1. If $n = 1$, done.
    2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$
       and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$ .
    3. "Merge" the 2 sorted lists.
```

Key subroutine: MERGE

Merging two sorted arrays

20 12

13 11

7 9

2 1

Merging two sorted arrays

20 12

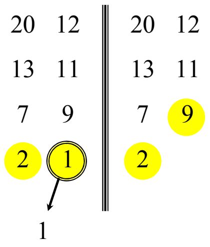
13 11

7 9

2 1

1

Merging two sorted arrays



20 12 20 12

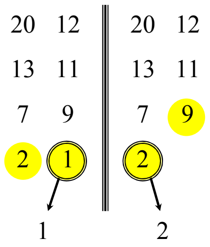
13 11 13 11

7 9 7 9

2 1 2

1

Merging two sorted arrays



20 12 20 12

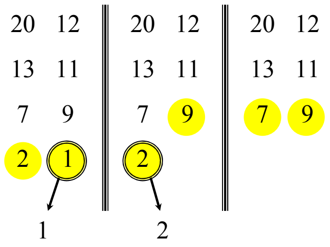
13 11 13 11

7 9 7 9

2 1 2

1 2

Merging two sorted arrays



20 12 20 12 20 12

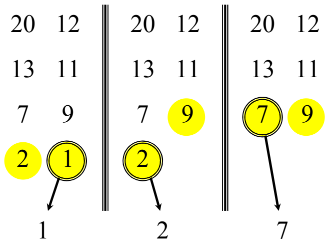
13 11 13 11 13 11

7 9 7 9 7 9

2 1 2

1 2

Merging two sorted arrays



20 12 20 12 20 12

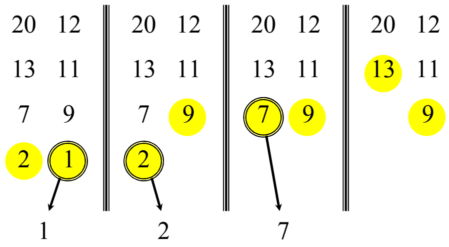
13 11 13 11 13 11

7 9 7 9 7 9

2 1 2

1 2 7

Merging two sorted arrays



20 12 20 12 20 12 20 12

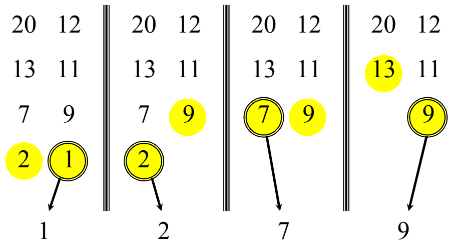
13 11 13 11 13 11 13 11

7 9 7 9 7 9 9

2 1 2

1 2 7

Merging two sorted arrays



20 12 20 12 20 12 20 12

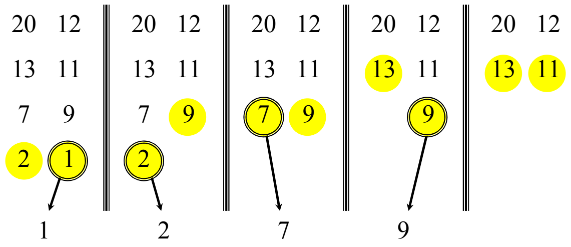
13 11 13 11 13 11 13 11

7 9 7 9 7 9 9

2 1 2

1 2 7 9

Merging two sorted arrays



20 12 20 12 20 12 20 12 20 12

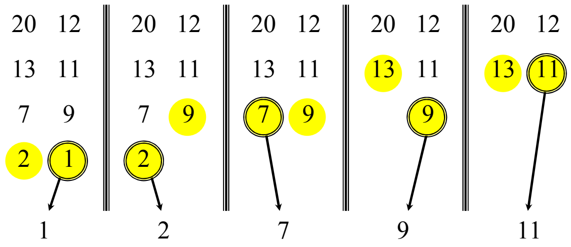
13 11 13 11 13 11 13 11 13 11

7 9 7 9 7 9 9

2 1 2

1 2 7 9

Merging two sorted arrays



20 12 20 12 20 12 20 12 20 12

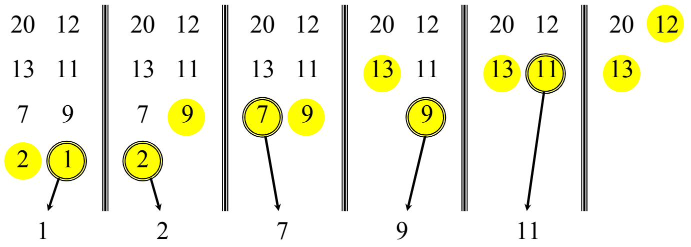
13 11 13 11 13 11 13 11 13 11

7 9 7 9 7 9 9

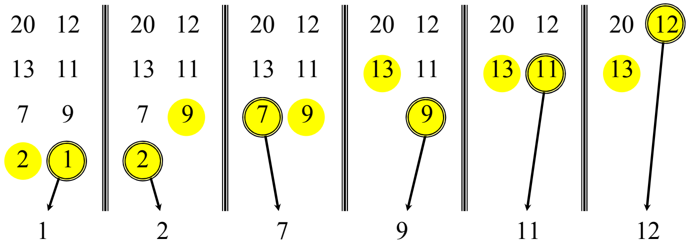
2 1 2

1 2 7 9 11

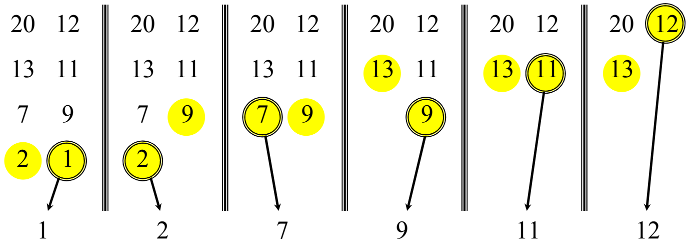
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



Time $dn = \Theta(n)$ to merge a total of n elements (linear time).

Analyzing merge sort

$T(n)$	MERGE-SORT $A[1 \,.\,.\, n]$
d_0	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
dn	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 merge sort

$$T(n) = \begin{cases} d_0 \text{ if } n = 1; \\ 2T(n/2) + dn \text{ if } n > 1. \end{cases}$$

- Later we shall often 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.

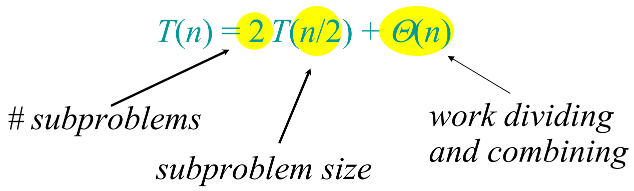
- But what does $T(n)$ solve to? I.e., is it $O(n)$ or $O(n^2)$ or $O(n^3)$ or …?

The divide-and-conquer design paradigm

1. ***Divide*** the problem (instance) into subproblems.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.

Example: merge sort

1. ***Divide:*** Trivial.
2. ***Conquer:*** Recursively sort 2 subarrays.
3. ***Combine:*** Linear-time merge.



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

# subproblems

subproblem size

work dividing and combining

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

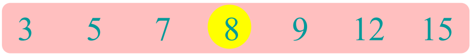
3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

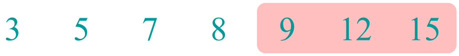

3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

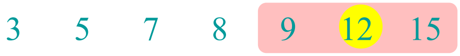

3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

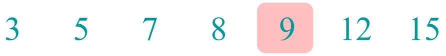

3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9


3 5 7 8 9 12 15

Binary search

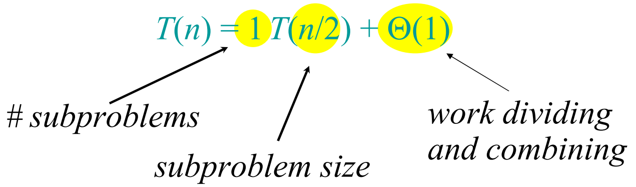
Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Recurrence for binary search



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

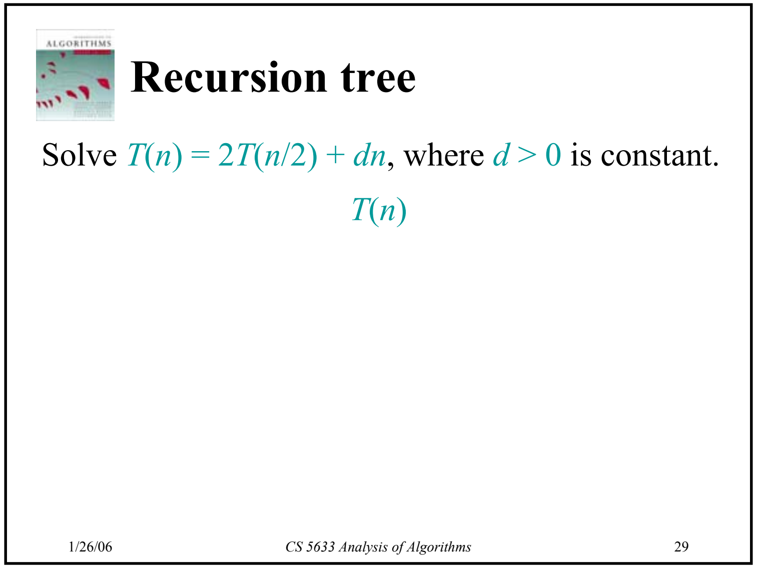
Recurrence for merge sort

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

- How do we solve $T(n)$? I.e., how do we find out if it is $O(\mathrm{n})$ or $O(\mathrm{n}^2)$ or …?

Recursion tree

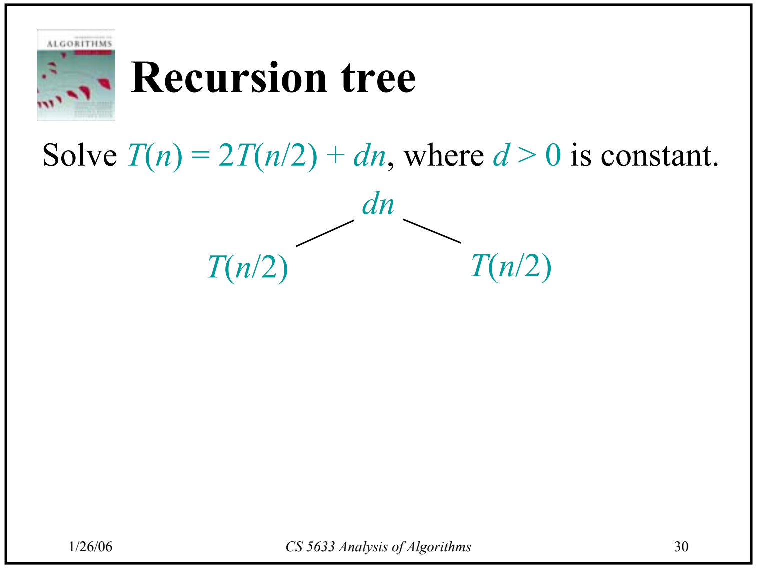
Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.



Recursion tree

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

$T(n)$

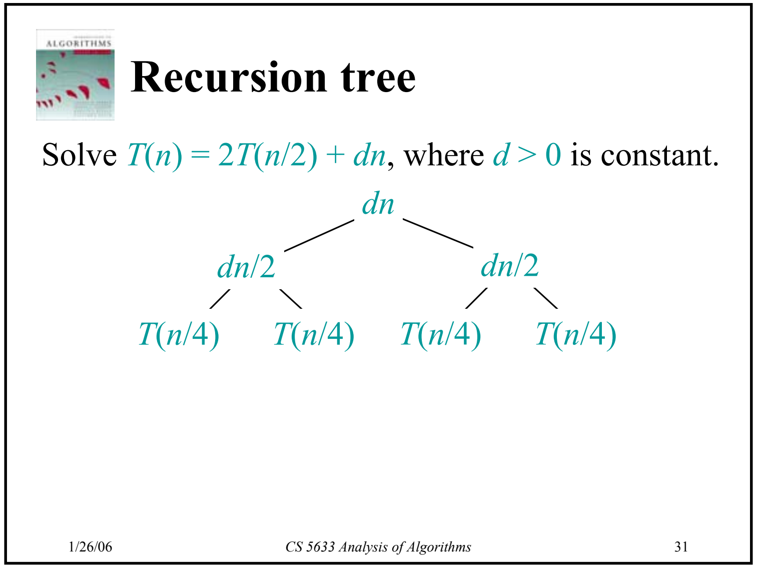


Recursion tree

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

dn

$T(n/2)$ $T(n/2)$



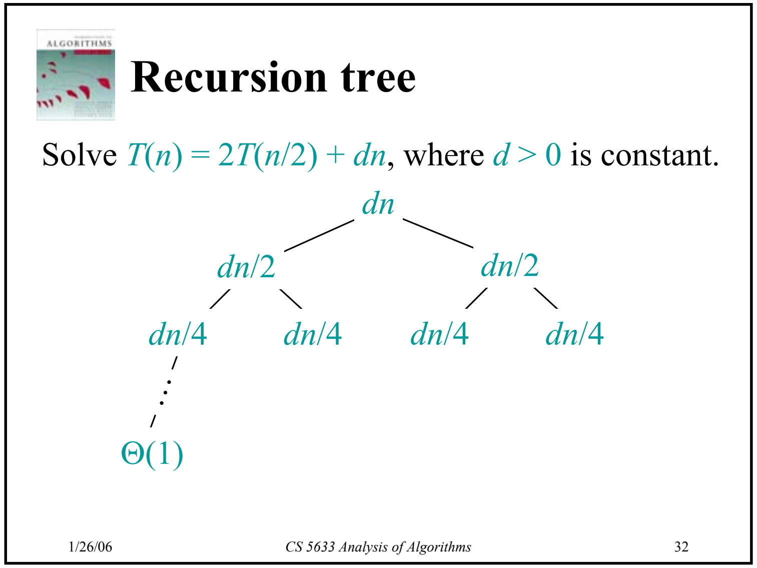
Recursion tree

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

dn

$dn/2$ $dn/2$

$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$



Recursion tree

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

dn

$dn/2$ $dn/2$

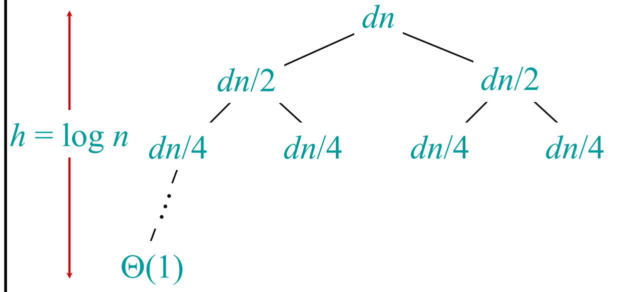
$dn/4$ $dn/4$ $dn/4$ $dn/4$

$\vdots$

$\Theta(1)$

Recursion tree

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



dn

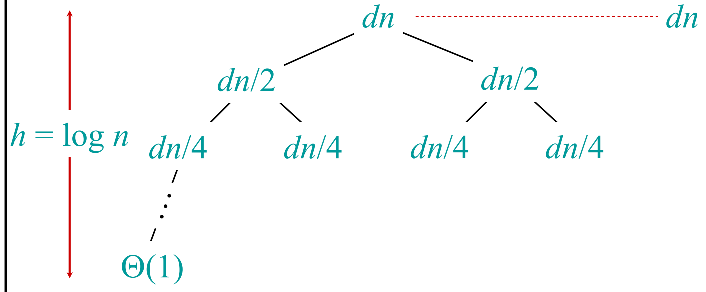
$dn/2$ $dn/2$

$h = \log n$ $dn/4$ $dn/4$ $dn/4$ $dn/4$

$\Theta(1)$

Recursion tree

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



dn dn

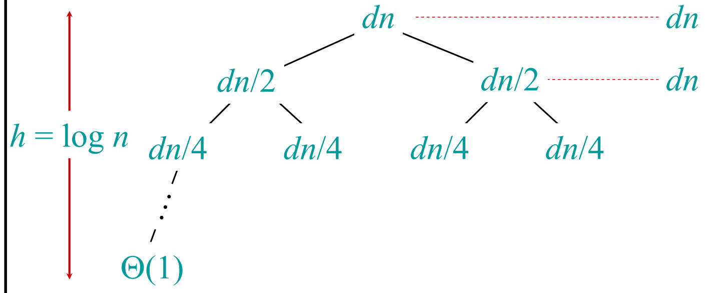
$dn/2$ $dn/2$

$h = \log n$ $dn/4$ $dn/4$ $dn/4$ $dn/4$

$\Theta(1)$

Recursion tree

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



dn dn

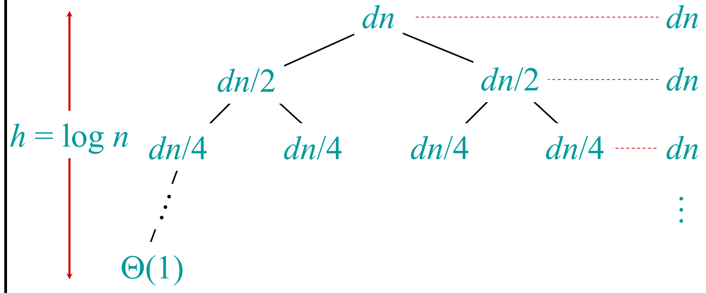
$dn/2$ $dn/2$ dn

$h = \log n$ $dn/4$ $dn/4$ $dn/4$ $dn/4$

$\Theta(1)$

Recursion tree

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



dn dn

$dn/2$ $dn/2$ dn

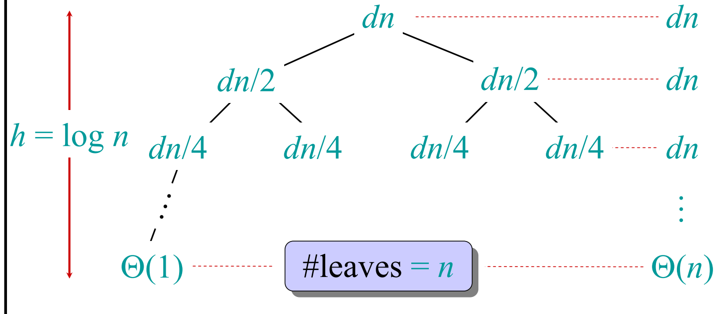
$h = \log n$ $dn/4$ $dn/4$ $dn/4$ $dn/4$ dn

$\vdots$

$\Theta(1)$

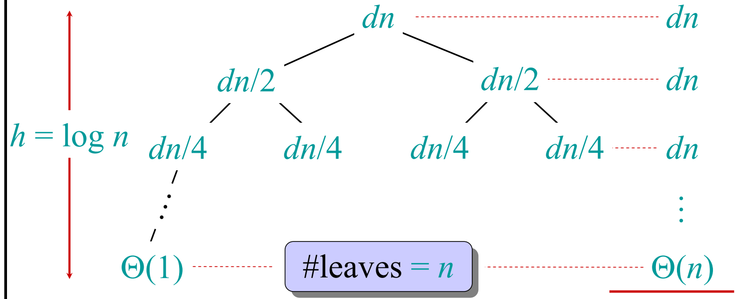
Recursion tree

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



Recursion tree

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



Total $\Theta(n \log n)$

Conclusions

- Merge sort runs in $\Theta(n \log n)$ time.
- $\Theta(n \log n)$ 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. (Why not earlier?)

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 (…).
- It is good for generating **guesses** of what the runtime could be.

But: Need to **verify** that the guess is right.
→ Induction (substitution method)

Substitution method

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

1. ***Guess*** the form of the solution:
   (e.g. using recursion trees, or expansion)
2. ***Verify*** by induction (inductive step).
3. ***Solve*** for O-constants n_0 and c (base case of induction)

The divide-and-conquer design paradigm

1. ***Divide*** the problem (instance) into subproblems.

   a subproblems, **each** of size n/b
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.

   Runtime is $f(n)$

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.

Three common cases

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

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.
2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Three common cases (cont.)

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

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the ***regularity condition*** that $a f(n/b) \le c f(n)$ for some constant $c < 1$.

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

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 $4(cn/2)^3 \le cn^3$ (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 $n^{\varepsilon} = \omega(\lg n)$.

Master theorem (summary)

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

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

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$
$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a f(n/b) \le c f(n)$
$\Rightarrow T(n) = \Theta(f(n))$.

Merge sort: $a = 2,\ b = 2 \ \Rightarrow\ n^{\log_b a} = n$
$\Rightarrow$ CASE 2 $(k = 0)$ $\Rightarrow$ $T(n) = \Theta(n \lg n)$.