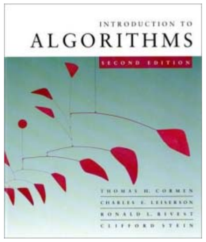
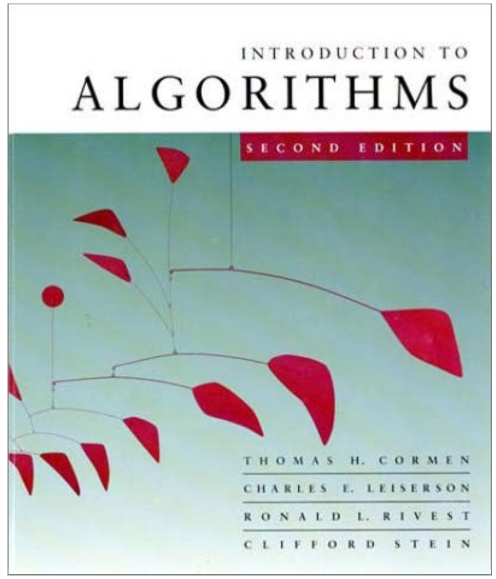


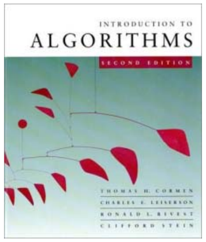
CS 5633 – Fall 2012



Divide-and-Conquer

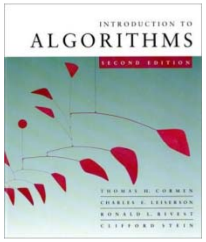
Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk



The divide-and-conquer design paradigm

1. ***Divide*** the problem (instance) into subproblems of sizes that are fractions of the original problem size.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.



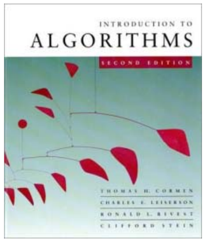
Binary search

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



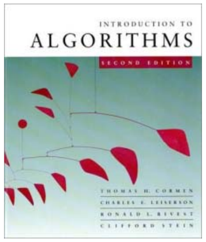
Binary search

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



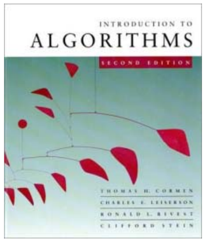
Binary search

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



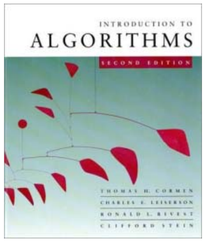
Binary search

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



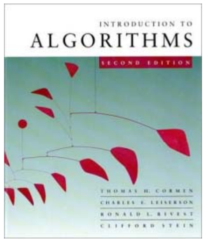
Binary search

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



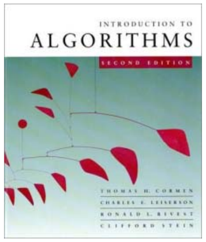
Binary search

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

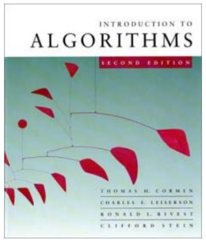


Merge sort

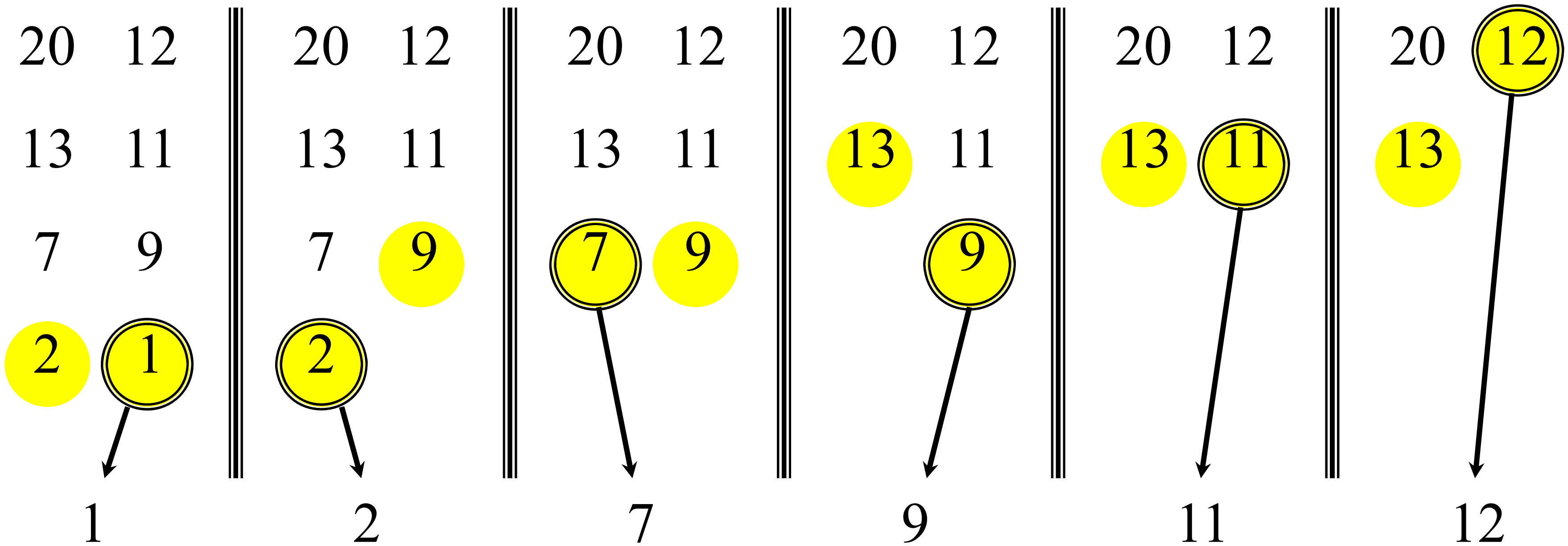
1. ***Divide:*** Trivial.
2. ***Conquer:*** Recursively sort 2 subarrays of size $n/2$
3. ***Combine:*** Linear-time key subroutine **MERGE**

MERGE-SORT ($A[1 \mathinner{.\,.} n]$)

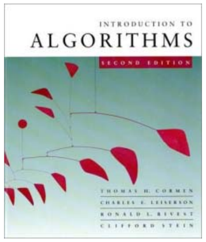
1. If $n = 1$, done.
2. **MERGE-SORT** ($A[\,1 \mathinner{.\,.} \lceil n/2 \rceil\,]$)
3. **MERGE-SORT** ($A[\,\lceil n/2 \rceil + 1 \mathinner{.\,.} n\,]$)
4. "***Merge***" the 2 sorted lists.



Merging two sorted arrays



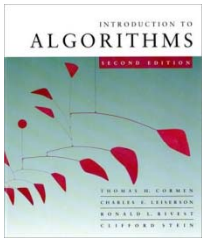
Time $dn \in \Theta(n)$ to merge a total of n elements (linear time).



Analyzing merge sort

Cost	
$T(n)$	**MERGE-SORT** $(A[1 \,..\, n])$
d_0	1. If $n = 1$, done.
$T(n/2)$	2. **MERGE-SORT** $(A[\,1 \,..\, \lceil n/2 \rceil\,])$
$T(n/2)$	3. **MERGE-SORT** $(A[\,\lceil n/2 \rceil + 1 \,..\, n\,])$
dn	4. "***Merge***" the 2 sorted lists.

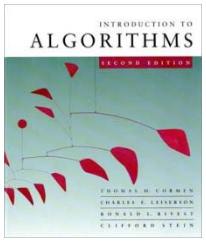
Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.



Recurrence for merge sort

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

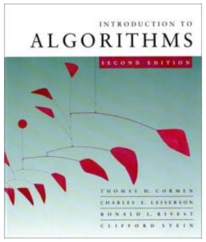
- But what does $T(n)$ solve to? I.e., is it $O(n)$ or $O(n^2)$ or $O(n^3)$ or …?



Recursion tree

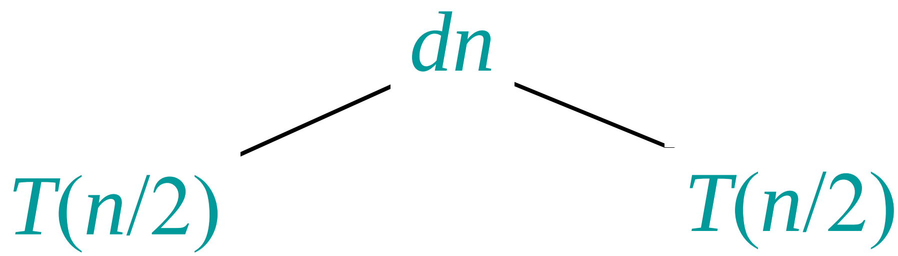
Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

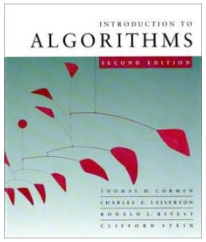
$$T(n)$$



Recursion tree

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





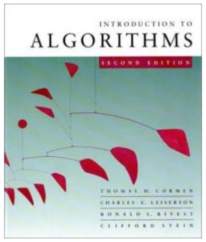
Recursion tree

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

dn

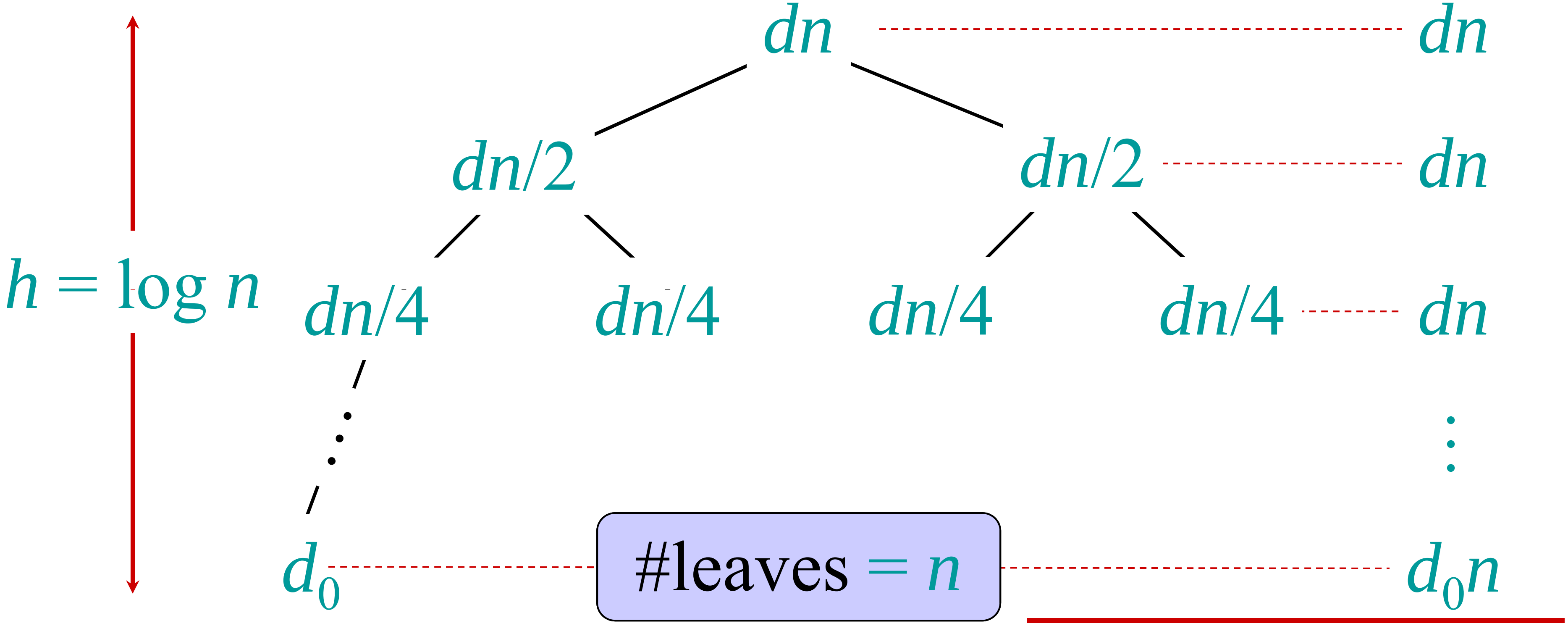
$dn/2$ $dn/2$

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

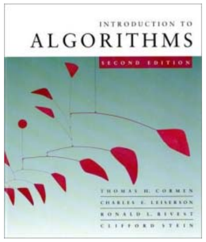


Recursion tree

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

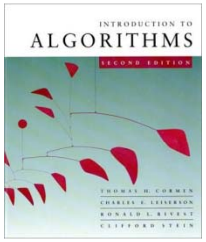


Total $dn \log n + d_0 n$



Mergesort Conclusions

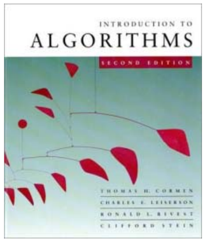
- Merge sort runs in $\Theta(n \log n)$ time.
- $\Theta(n \log n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so. (Why not earlier?)



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.

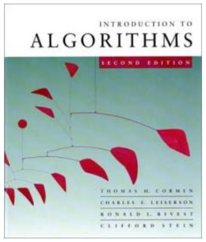
But: Need to **verify** that the guess is correct.
→ Induction (substitution method)



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)



Powering a number

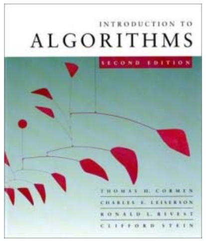
Problem: Compute a^n, where $n \in \mathbf{N}$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm: **(recursive squaring)**

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \;\Rightarrow\; T(n) = \Theta(\log n) \,.$$



Matrix multiplication

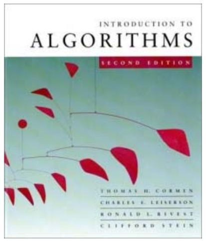
Input: $A = [a_{ij}],\ B = [b_{ij}].$

Output: $C = [c_{ij}] = A \cdot B.$

$$\left.\begin{array}{l} \\ \\ \end{array}\right\} i, j = 1, 2, \ldots, n.$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

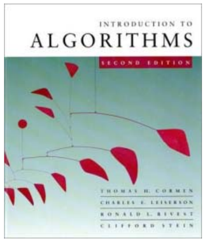
$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$



Standard algorithm

for $i \leftarrow 1$ **to** n
  do for $j \leftarrow 1$ **to** n
    do $c_{ij} \leftarrow 0$
      for $k \leftarrow 1$ **to** n
        do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$

Running time $= \Theta(n^3)$



Divide-and-conquer algorithm

IDEA:

$n \times n$ matrix $= 2 \times 2$ matrix of $(n/2) \times (n/2)$ submatrices:

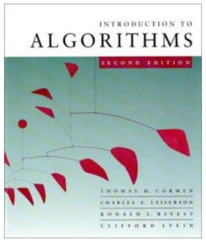
$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

$$\left.\begin{aligned} r &= a \cdot e + b \cdot g \\ s &= a \cdot f + b \cdot h \\ t &= c \cdot e + d \cdot g \\ u &= c \cdot f + d \cdot h \end{aligned}\right\}$$

8 recursive mults of $(n/2) \times (n/2)$ submatrices

4 adds of $(n/2) \times (n/2)$ submatrices



Analysis of D&C algorithm

$$T(n) = 8\,T(n/2) + \Theta(n^2)$$

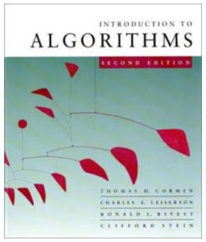
 # submatrices → 8

submatrix size → $T(n/2)$

work adding submatrices → $\Theta(n^2)$

Solves to $T(n) = \Theta(n^3) = \Theta(n^{\log 8})$

No better than the ordinary matrix multiplication algorithm.



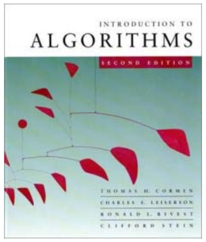
Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

7 mults, 18 adds/subs.
Note: No reliance on commutativity of mult!

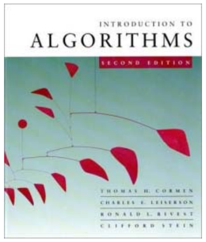


Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

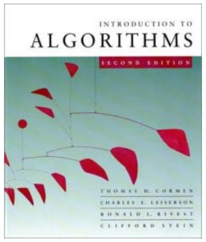
$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
&= (a + d)(e + h) \\
&\quad + d(g - e) - (a + b)h \\
&\quad + (b - d)(g + h) \\
&= ae + ah + de + dh \\
&\quad + dg - de - ah - bh \\
&\quad + bg + bh - dg - dh \\
&= ae + bg
\end{aligned}
$$



Strassen's algorithm

1. ***Divide:*** Partition A and B into $(n/2)\times(n/2)$ submatrices. Form P-terms to be multiplied using $+$ and $-$.
2. ***Conquer:*** Perform 7 multiplications of $(n/2)\times(n/2)$ submatrices recursively.
3. ***Combine:*** Form C using $+$ and $-$ on $(n/2)\times(n/2)$ submatrices.

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$



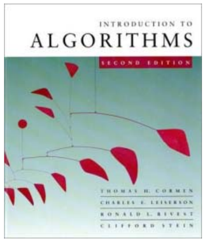
Analysis of Strassen

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

Solves to $T(n) = \Theta(n^{\log 7})$

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 30$ or so.

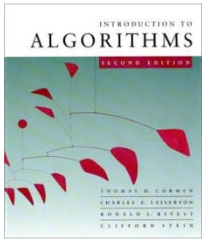
Best to date (of theoretical interest only): $\Theta(n^{2.376\cdots})$.



The divide-and-conquer design paradigm

1. ***Divide*** the problem (instance) into subproblems of sizes that are fractions of the original problem size.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.

$\Rightarrow$ Runtime recurrences

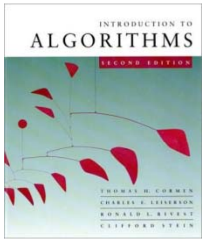


The master method

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.



Example: merge sort

1. ***Divide:*** Trivial.
2. ***Conquer:*** Recursively sort $a=2$ subarrays of size $n/2=n/b$
3. ***Combine:*** Linear-time merge, runtime $f(n) \in O(n)$

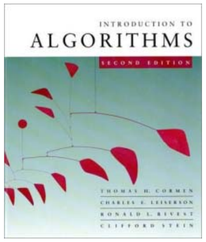
$$T(n) = 2\,T(n/2) + O(n)$$

 # subproblems → 2 and a

subproblem size → $n/2$ and n/b

work dividing and combining → $O(n)$ and $f(n)$

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



Master Theorem

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

CASE 1:

$$f(n) = O(n^{\log_b a - \varepsilon}) \quad \Rightarrow T(n) = \Theta(n^{\log_b a})$$

CASE 2:

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

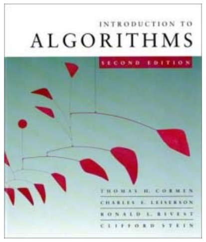
CASE 3:

$f(n) = \Omega(n^{\log_b a + \varepsilon})$

and $a\,f(n/b) \le c\,f(n)$

for some constant $c < 1$

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



How to apply the theorem

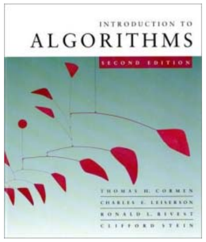
Compare $f(n)$ with $n^{\log_b a}$:

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

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



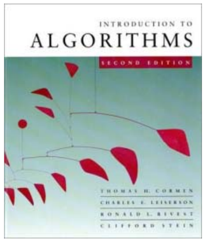
How to apply the theorem

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



Example: merge sort

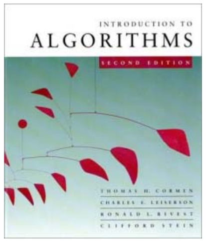
1. ***Divide:*** Trivial.
2. ***Conquer:*** Recursively sort 2 subarrays.
3. ***Combine:*** Linear-time merge.

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

 # subproblems (2), *subproblem size* ($n/2$), *work dividing and combining* ($O(n)$)

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n \Rightarrow \text{CASE 2 } (k = 0)$$

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



Example: binary search

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

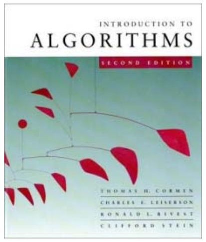
 # subproblems

subproblem size

work dividing and combining

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow \text{CASE } 2\ (k = 0)$$

$$\Rightarrow T(n) = \Theta(\lg n)\,.$$



Matrix multiplication: Divide-and-conquer algorithm

IDEA:

$n\times n$ matrix = 2×2 matrix of $(n/2)\times(n/2)$ submatrices:

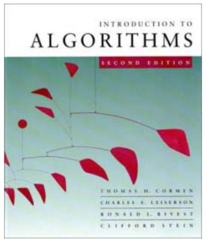
$$\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]$$

$$C \quad = \quad A \quad \cdot \quad B$$

$$\left.\begin{aligned} r &= a\cdot e + b\cdot g \\ s &= a\cdot f + b\cdot h \\ t &= c\cdot e + d\cdot g \\ u &= c\cdot f + d\cdot h \end{aligned}\right\}$$

8 recursive mults of $(n/2)\times(n/2)$ submatrices

4 adds of $(n/2)\times(n/2)$ submatrices



Matrix multiplication: Analysis of D&C algorithm

$$T(n) = 8\,T(n/2) + \Theta(n^2)$$

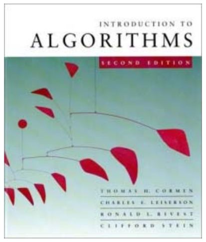
\# submatrices

submatrix size

work adding submatrices

$$n^{\log_b a} = n^{\log_2 8} = n^3 \;\Rightarrow\; \text{CASE 1} \;\Rightarrow\; T(n) = \Theta(n^3)$$

No better than the ordinary matrix multiplication algorithm.

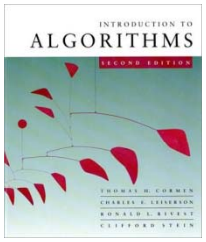


Strassen's algorithm

1. ***Divide:*** Partition A and B into $(n/2)\times(n/2)$ submatrices. Form P-terms to be multiplied using $+$ and $-$.
2. ***Conquer:*** Perform 7 multiplications of $(n/2)\times(n/2)$ submatrices recursively.
3. ***Combine:*** Form C using $+$ and $-$ on $(n/2)\times(n/2)$ submatrices.

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \;\Rightarrow\; \text{CASE 1} \;\Rightarrow\; T(n) = \Theta(n^{\log 7})$$



Master theorem: Examples

Ex. $T(n) = 4T(n/2) + \mathrm{sqrt}(n)$

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

CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1.5$.

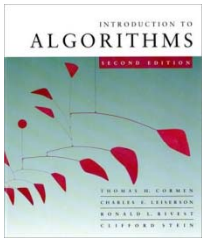
$\therefore T(n) = \Theta(n^2)$.

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

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



Master theorem: 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 $\varepsilon = 1$

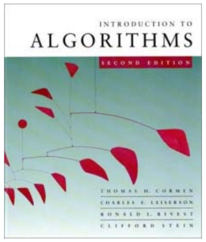
and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.

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

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

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

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $\log n \in o(n^\varepsilon)$.



Conclusion

- Divide and conquer is just one of several powerful techniques for algorithm design.
- Divide-and-conquer algorithms can be analyzed using recurrences and the master method .
- Can lead to more efficient algorithms