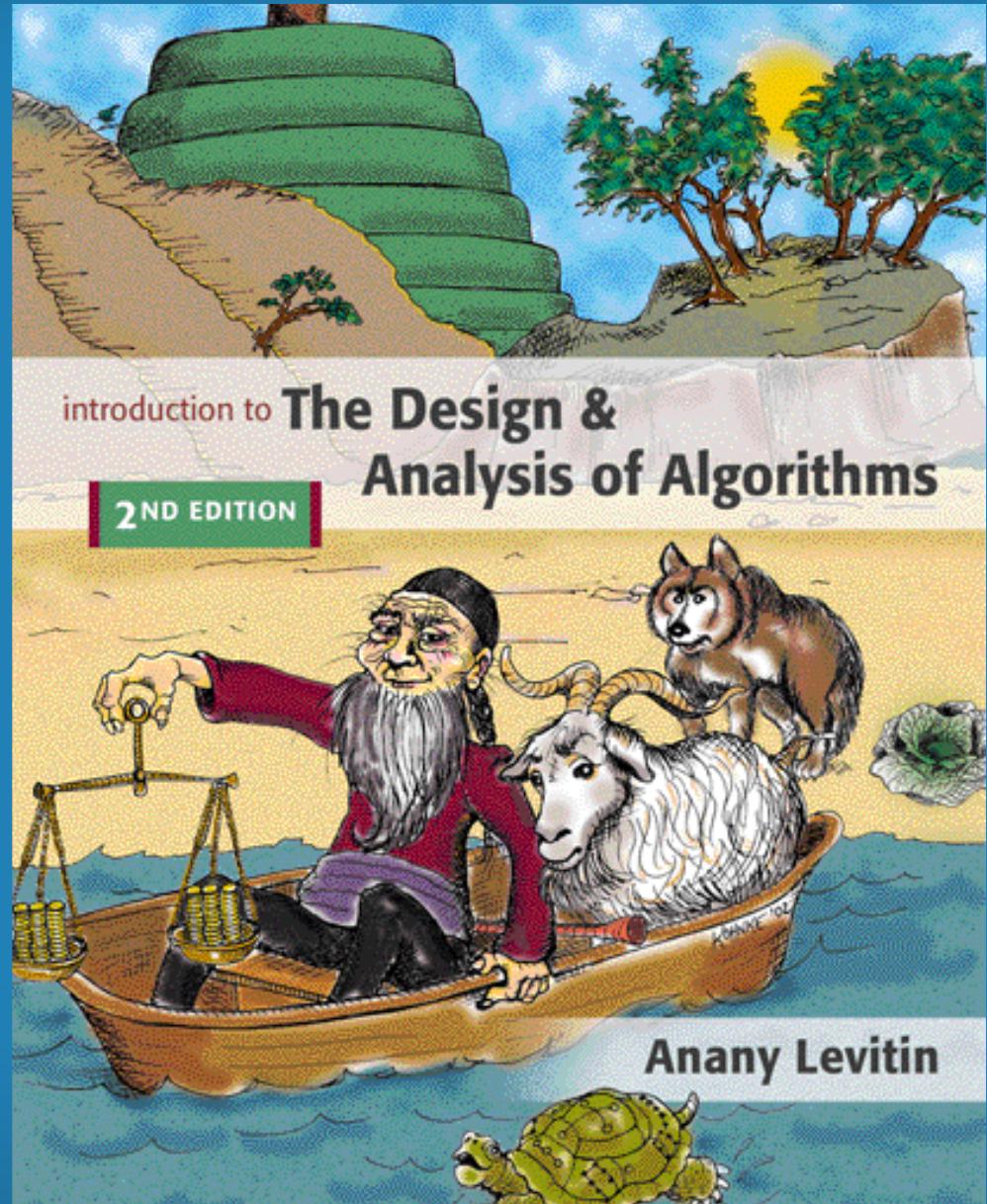


Chapter 4

Divide-and-Conquer



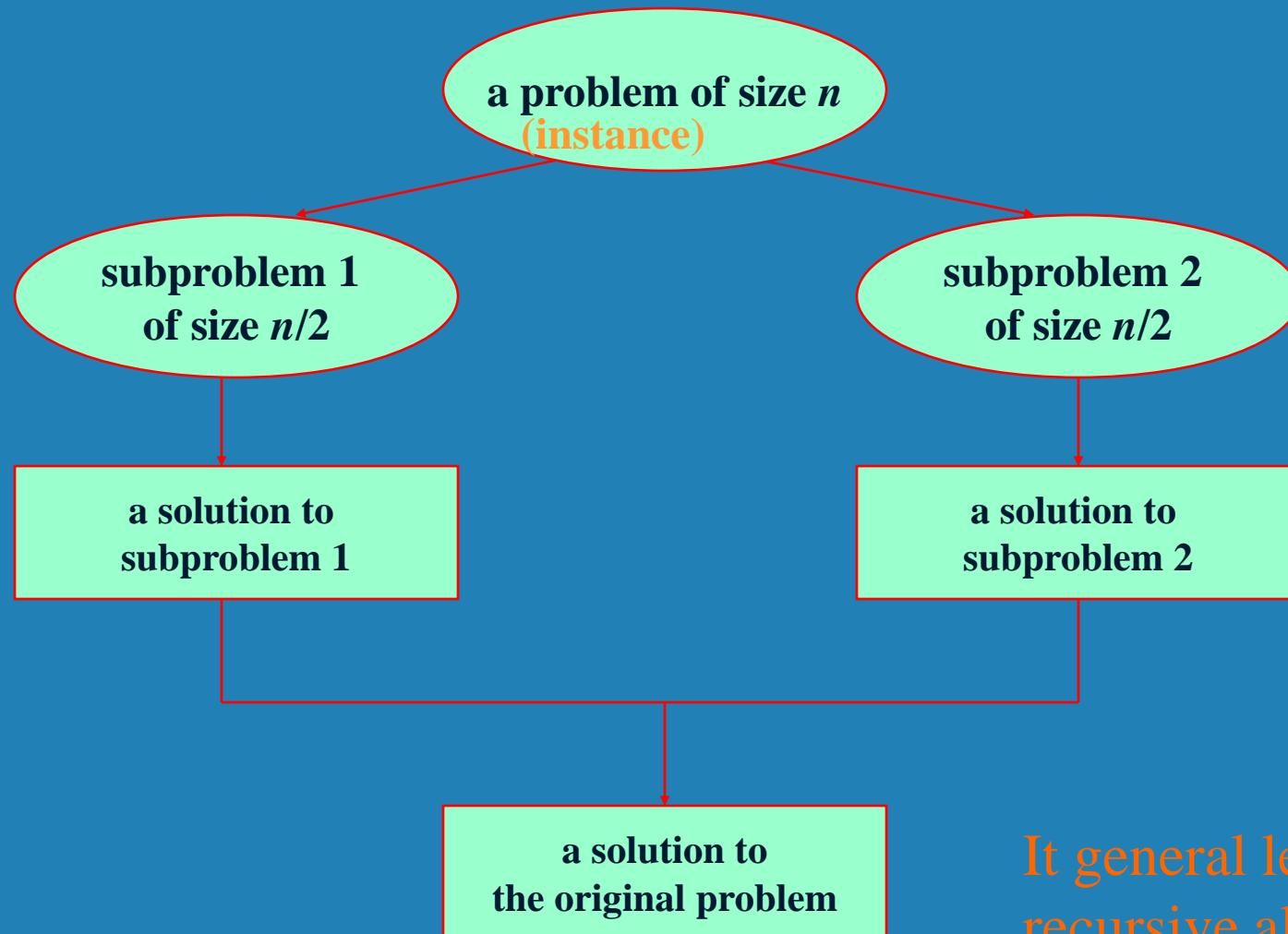
Divide-and-Conquer



The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)



It generally leads to a recursive algorithm!

Divide-and-Conquer Examples



- **Sorting: mergesort and quicksort**
- **Binary tree traversals**
- **Binary search (?)**
- **Multiplication of large integers**
- **Matrix multiplication: Strassen's algorithm**
- **Closest-pair and convex-hull algorithms**



General Divide-and-Conquer Recurrence

$$T(n) = aT(n/b) + f(n) \quad \text{where } f(n) \in \Theta(n^d), \quad d \geq 0$$



Master Theorem:

- If $a < b^d$, $T(n) \in \Theta(n^d)$
- If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
- If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ .

| | |
|---|----------------------|
| Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ | $\Theta(n^2)$ |
| $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ | $\Theta(n^2 \log n)$ |
| $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ | $\Theta(n^3)$ |

Mergesort



- Split array A[0..n-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
 - Repeat the following until no elements remain in one of the arrays:
 - compare the first elements in the remaining unprocessed portions of the arrays
 - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
 - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort

ALGORITHM *Mergesort($A[0..n - 1]$)*

//Sorts array $A[0..n - 1]$ by recursive mergesort
//Input: An array $A[0..n - 1]$ of orderable elements
//Output: Array $A[0..n - 1]$ sorted in nondecreasing order

if $n > 1$

copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor ..n - 1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort($B[0..\lfloor n/2 \rfloor - 1]$)
Mergesort($C[0..\lceil n/2 \rceil - 1]$)
Merge(B, C, A)

Pseudocode of Merge

ALGORITHM *Merge($B[0..p - 1]$, $C[0..q - 1]$, $A[0..p + q - 1]$)*

//Merges two sorted arrays into one sorted array

//Input: Arrays $B[0..p - 1]$ and $C[0..q - 1]$ both sorted

//Output: Sorted array $A[0..p + q - 1]$ of the elements of B and C

$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$

while $i < p$ **and** $j < q$ **do**

if $B[i] \leq C[j]$

$A[k] \leftarrow B[i]$; $i \leftarrow i + 1$

else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$

$k \leftarrow k + 1$

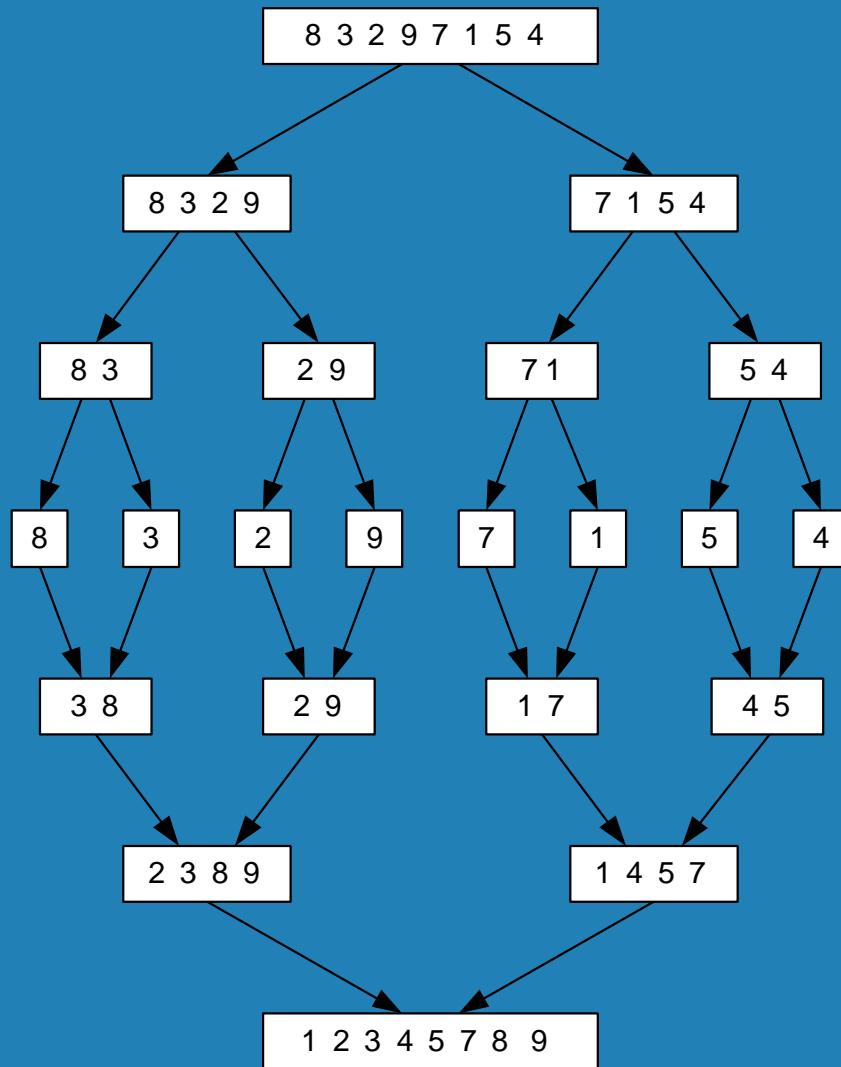
if $i = p$

 copy $C[j..q - 1]$ to $A[k..p + q - 1]$

else copy $B[i..p - 1]$ to $A[k..p + q - 1]$

Time complexity: $\Theta(p+q) = \Theta(n)$ comparisons

Mergesort Example



The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

Analysis of Mergesort

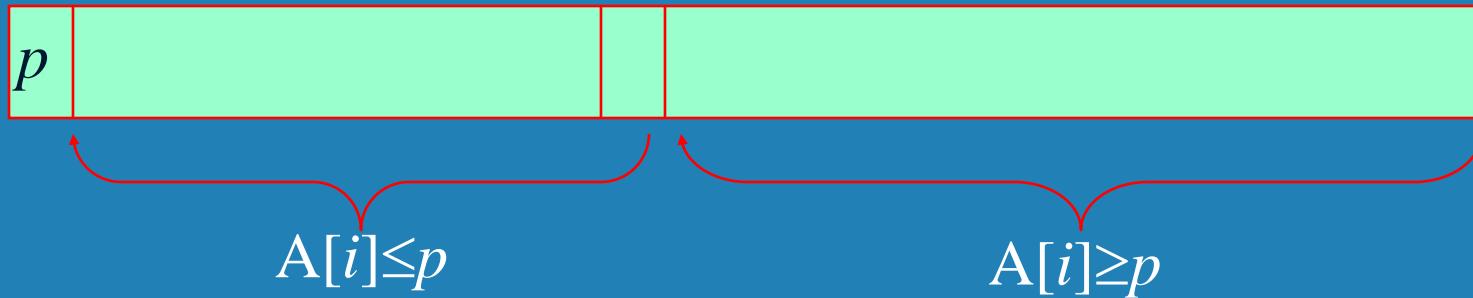


- All cases have same efficiency: $\Theta(n \log n)$
$$T(n) = 2T(n/2) + \Theta(n), T(1) = 0$$
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:
$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$
- Space requirement: $\Theta(n)$ (not in-place)
- Can be implemented without recursion (bottom-up)

Quicksort



- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first s positions are smaller than or equal to the pivot and all the elements in the remaining $n-s$ positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., \leq) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Partitioning Algorithm



Algorithm *Partition*($A[l..r]$)

```
//Partitions a subarray by using its first element as a pivot
//Input: A subarray  $A[l..r]$  of  $A[0..n - 1]$ , defined by its left and right
//       indices  $l$  and  $r$  ( $l < r$ )
//Output: A partition of  $A[l..r]$ , with the split position returned as
//       this function's value

 $p \leftarrow A[l]$ 
 $i \leftarrow l; j \leftarrow r + 1$ 
repeat
    repeat  $i \leftarrow i + 1$  until  $A[i] \geq p$            or  $i > r$ 
    repeat  $j \leftarrow j - 1$  until  $A[j] < p$            or  $j = l$ 
    swap( $A[i], A[j]$ )
until  $i \geq j$ 
swap( $A[i], A[j]$ ) //undo last swap when  $i \geq j$ 
swap( $A[l], A[j]$ )
return  $j$ 
```

Time complexity: $\Theta(r-l)$ comparisons

Quicksort Example



5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

1 2 3 4 5 7 8 9

1 2 3 5 4 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Analysis of Quicksort



- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$ $T(n) = T(n-1) + \Theta(n)$
- Average case: random arrays — $\Theta(n \log n)$

- Improvements:
 - better pivot selection: median of three partitioning
 - switch to insertion sort on small subfiles
 - elimination of recursionThese combine to 20-25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Binary Search



Very efficient algorithm for searching in sorted array:

K

vs

$A[0] \dots A[m] \dots A[n-1]$

If $K = A[m]$, stop (successful search); otherwise, continue searching by the same method in $A[0..m-1]$ if $K < A[m]$ and in $A[m+1..n-1]$ if $K > A[m]$

$l \leftarrow 0; r \leftarrow n-1$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l+r)/2 \rfloor$

if $K = A[m]$ **return** m

else if $K < A[m]$ $r \leftarrow m-1$

else $l \leftarrow m+1$

return -1

Analysis of Binary Search



- Time efficiency
 - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
solution: $C_w(n) = \lceil \log_2(n+1) \rceil$
- This is VERY fast: e.g., $C_w(10^6) = 20$
- Optimal for searching a sorted array
- Limitations: must be a sorted array (not linked list)
- Bad (degenerate) example of divide-and-conquer because only one of the sub-instances is solved
- Has a continuous counterpart called *bisection method* for solving equations in one unknown $f(x) = 0$ (see Sec. 12.4)

Binary Tree Algorithms



Binary tree is a divide-and-conquer ready structure!

Ex. 1: Classic traversals (preorder, inorder, postorder)

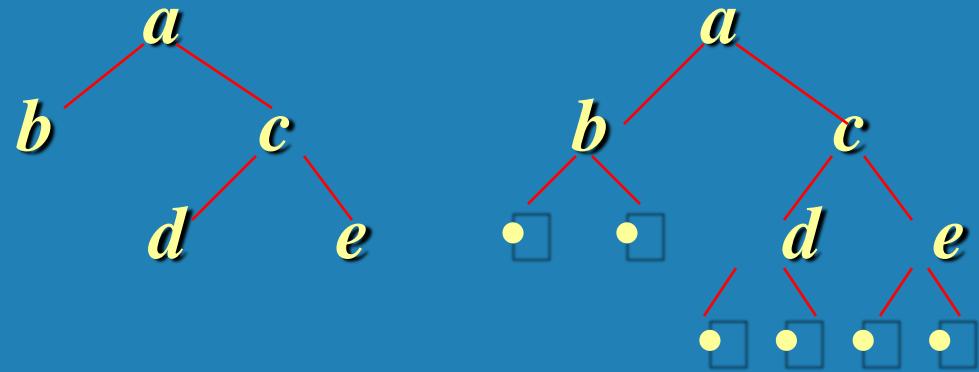
Algorithm *Inorder*(T)

if $T \neq \emptyset$

Inorder(T_{left})

 print(root of T)

Inorder(T_{right})



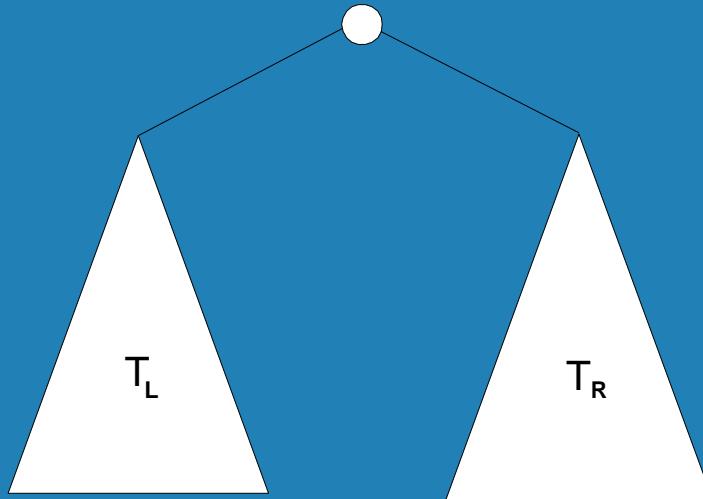
Efficiency: $\Theta(n)$. Why?

Each node is visited/printed once.

Binary Tree Algorithms (cont.)



Ex. 2: Computing the height of a binary tree



$$h(T) = \max\{h(T_L), h(T_R)\} + 1 \text{ if } T \neq \emptyset \text{ and } h(\emptyset) = -1$$

Efficiency: $\Theta(n)$. Why?

Multiplication of Large Integers



Consider the problem of multiplying two (large) n -digit integers represented by arrays of their digits such as:

$$A = 12345678901357986429 \quad B = 87654321284820912836$$

The grade-school algorithm:

$$\begin{array}{r} a_1 \ a_2 \dots \ a_n \\ b_1 \ b_2 \dots \ b_n \\ \hline (d_{10}) \ d_{11} \ d_{12} \dots \ d_{1n} \\ (d_{20}) \ d_{21} \ d_{22} \dots \ d_{2n} \\ \dots \ \dots \ \dots \ \dots \ \dots \ \dots \\ (d_{n0}) \ d_{n1} \ d_{n2} \dots \ d_{nn} \end{array}$$

Efficiency: $\Theta(n^2)$ single-digit multiplications

First Divide-and-Conquer Algorithm

A small example: $A * B$ where $A = 2135$ and $B = 4014$

$$A = (21 \cdot 10^2 + 35), \quad B = (40 \cdot 10^2 + 14)$$

$$\text{So, } A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$$

$$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$$

In general, if $A = A_1 A_2$ and $B = B_1 B_2$ (where A and B are n -digit, A_1, A_2, B_1, B_2 are $n/2$ -digit numbers),

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$$

Recurrence for the number of one-digit multiplications $M(n)$:

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution: $M(n) = n^2$

Second Divide-and-Conquer Algorithm



$$\mathbf{A} * \mathbf{B} = \mathbf{A}_1 * \mathbf{B}_1 \cdot 10^n + (\mathbf{A}_1 * \mathbf{B}_2 + \mathbf{A}_2 * \mathbf{B}_1) \cdot 10^{n/2} + \mathbf{A}_2 * \mathbf{B}_2$$

The idea is to decrease the number of multiplications from 4 to 3:

$$(\mathbf{A}_1 + \mathbf{A}_2) * (\mathbf{B}_1 + \mathbf{B}_2) = \mathbf{A}_1 * \mathbf{B}_1 + (\mathbf{A}_1 * \mathbf{B}_2 + \mathbf{A}_2 * \mathbf{B}_1) + \mathbf{A}_2 * \mathbf{B}_2,$$

I.e., $(\mathbf{A}_1 * \mathbf{B}_2 + \mathbf{A}_2 * \mathbf{B}_1) = (\mathbf{A}_1 + \mathbf{A}_2) * (\mathbf{B}_1 + \mathbf{B}_2) - \mathbf{A}_1 * \mathbf{B}_1 - \mathbf{A}_2 * \mathbf{B}_2$, which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications $M(n)$:

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

What if we count both multiplications and additions?

Example of Large-Integer Multiplication



2135 * 4014

$$= (21*10^2 + 35) * (40*10^2 + 14)$$

$$= (21*40)*10^4 + c1*10^2 + 35*14$$

where $c1 = (21+35)*(40+14) - 21*40 - 35*14$, and

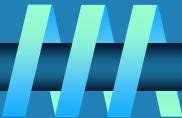
$$21*40 = (2*10 + 1) * (4*10 + 0)$$

$$= (2*4)*10^2 + c2*10 + 1*0$$

where $c2 = (2+1)*(4+0) - 2*4 - 1*0$, etc.

This process requires 9 digit multiplications as opposed to 16.

Conventional Matrix Multiplication



□ Brute-force algorithm

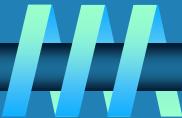
$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$
$$= \begin{pmatrix} a_{00} * b_{00} + a_{01} * b_{10} & a_{00} * b_{01} + a_{01} * b_{11} \\ a_{10} * b_{00} + a_{11} * b_{10} & a_{10} * b_{01} + a_{11} * b_{11} \end{pmatrix}$$

8 multiplications

Efficiency class in general: $\Theta(n^3)$

4 additions

Strassen's Matrix Multiplication



- Strassen's algorithm for two 2×2 matrices (1969):

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$
$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

- $m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
- $m_2 = (a_{10} + a_{11}) * b_{00}$
- $m_3 = a_{00} * (b_{01} - b_{11})$
- $m_4 = a_{11} * (b_{10} - b_{00})$
- $m_5 = (a_{00} + a_{01}) * b_{11}$
- $m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
- $m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

7 multiplications

18 additions

Strassen's Matrix Multiplication



Strassen observed [1969] that the product of two matrices can be computed in general as follows:

$$\begin{array}{c} \left(\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array} \right) = \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array} \right) * \left(\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array} \right) \\ \\ = \left(\begin{array}{cc} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{array} \right) \end{array}$$

Formulas for Strassen's Algorithm



$$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$$

$$M_2 = (A_{10} + A_{11}) * B_{00}$$

$$M_3 = A_{00} * (B_{01} - B_{11})$$

$$M_4 = A_{11} * (B_{10} - B_{00})$$

$$M_5 = (A_{00} + A_{01}) * B_{11}$$

$$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$$

$$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$$



Analysis of Strassen's Algorithm



If n is not a power of 2, matrices can be padded with zeros.

What if we count both
multiplications and additions?

Number of multiplications:

$$M(n) = 7M(n/2), \quad M(1) = 1$$

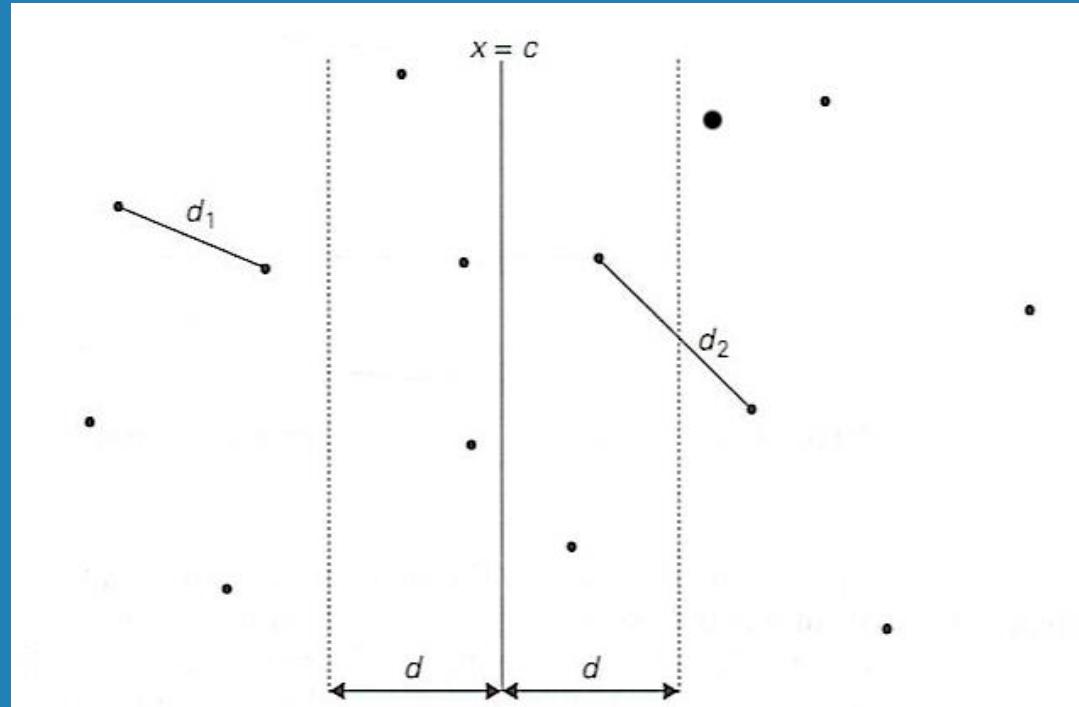
Solution: $M(n) = 7^{\log 2^n} = n^{\log 2^7} \approx n^{2.807}$ vs. n^3 of brute-force alg.

Algorithms with better asymptotic efficiency are known but they are even more complex and not used in practice.

Closest-Pair Problem by Divide-and-Conquer

Step 0 Sort the points by x (list one) and then by y (list two).

Step 1 Divide the points given into two subsets S_1 and S_2 by a vertical line $x = c$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.



Closest Pair by Divide-and-Conquer (cont.)



Step 2 Find recursively the closest pairs for the left and right subsets.

Step 3 Set $d = \min\{d_1, d_2\}$

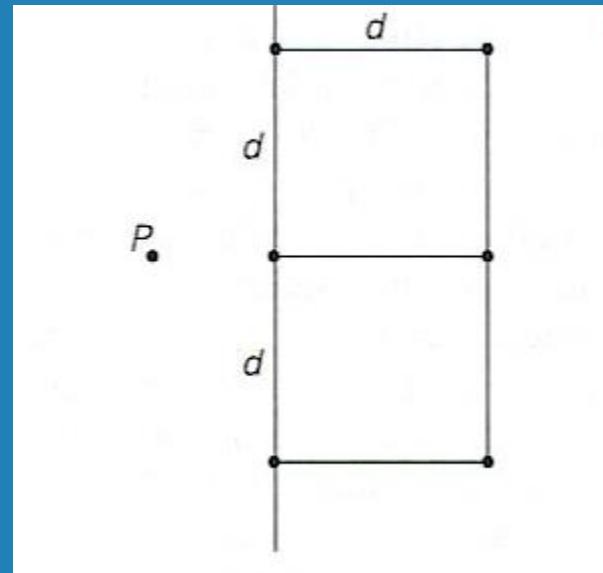
We can limit our attention to the points in the symmetric vertical strip of width $2d$ as possible closest pair. Let C_1 and C_2 be the subsets of points in the left subset S_1 and of the right subset S_2 , respectively, that lie in this vertical strip. The points in C_1 and C_2 are stored in increasing order of their y coordinates, taken from the second list.

Step 4 For every point $P(x,y)$ in C_1 , we inspect points in C_2 that may be closer to P than d . There can be no more than 6 such points (because $d \leq d_2$)!

Closest Pair by Divide-and-Conquer: Worst Case



The worst case scenario is depicted below:



Efficiency of the Closest-Pair Algorithm



Running time of the algorithm (without sorting) is:

$$T(n) = 2T(n/2) + M(n), \text{ where } M(n) \in \Theta(n)$$

By the Master Theorem (with $a = 2, b = 2, d = 1$)

$$T(n) \in \Theta(n \log n)$$

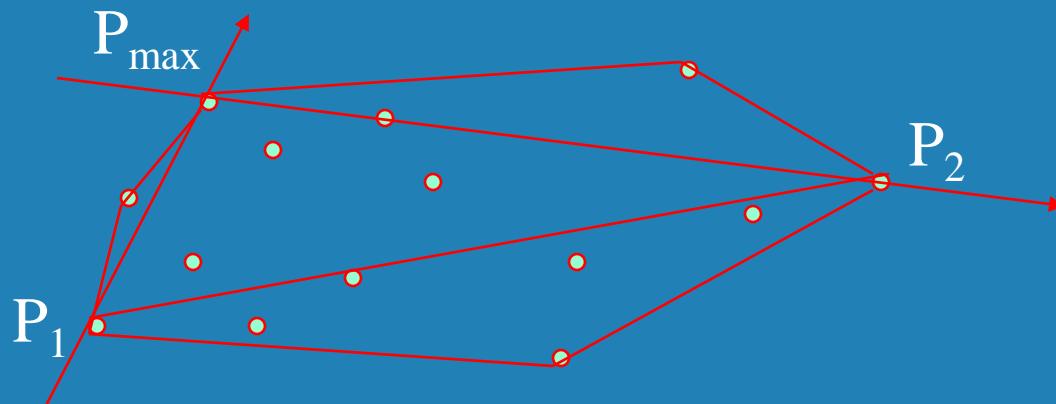
So the total time is $\Theta(n \log n)$.

Quickhull Algorithm



Convex hull: smallest convex set that includes given points. An $O(n^3)$ bruteforce time is given in Levitin, Ch 3.

- Assume points are sorted by x -coordinate values
- Identify *extreme points* P_1 and P_2 (leftmost and rightmost)
- Compute *upper hull* recursively:
 - find point P_{\max} that is farthest away from line P_1P_2
 - compute the upper hull of the points to the left of line P_1P_{\max}
 - compute the upper hull of the points to the left of line $P_{\max}P_2$
- Compute *lower hull* in a similar manner



Efficiency of Quickhull Algorithm



- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency: $T(n) = T(x) + T(y) + T(z) + T(v) + O(n)$,
where $x + y + z + v \leq n$.
 - worst case: $\Theta(n^2)$ $T(n) = T(n-1) + O(n)$
 - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)
- If points are not initially sorted by x -coordinate value, this can be accomplished in $O(n \log n)$ time
- Several $O(n \log n)$ algorithms for convex hull are known