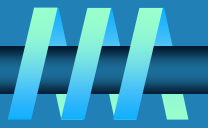
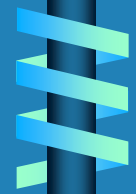
# Divide-and-Conquer

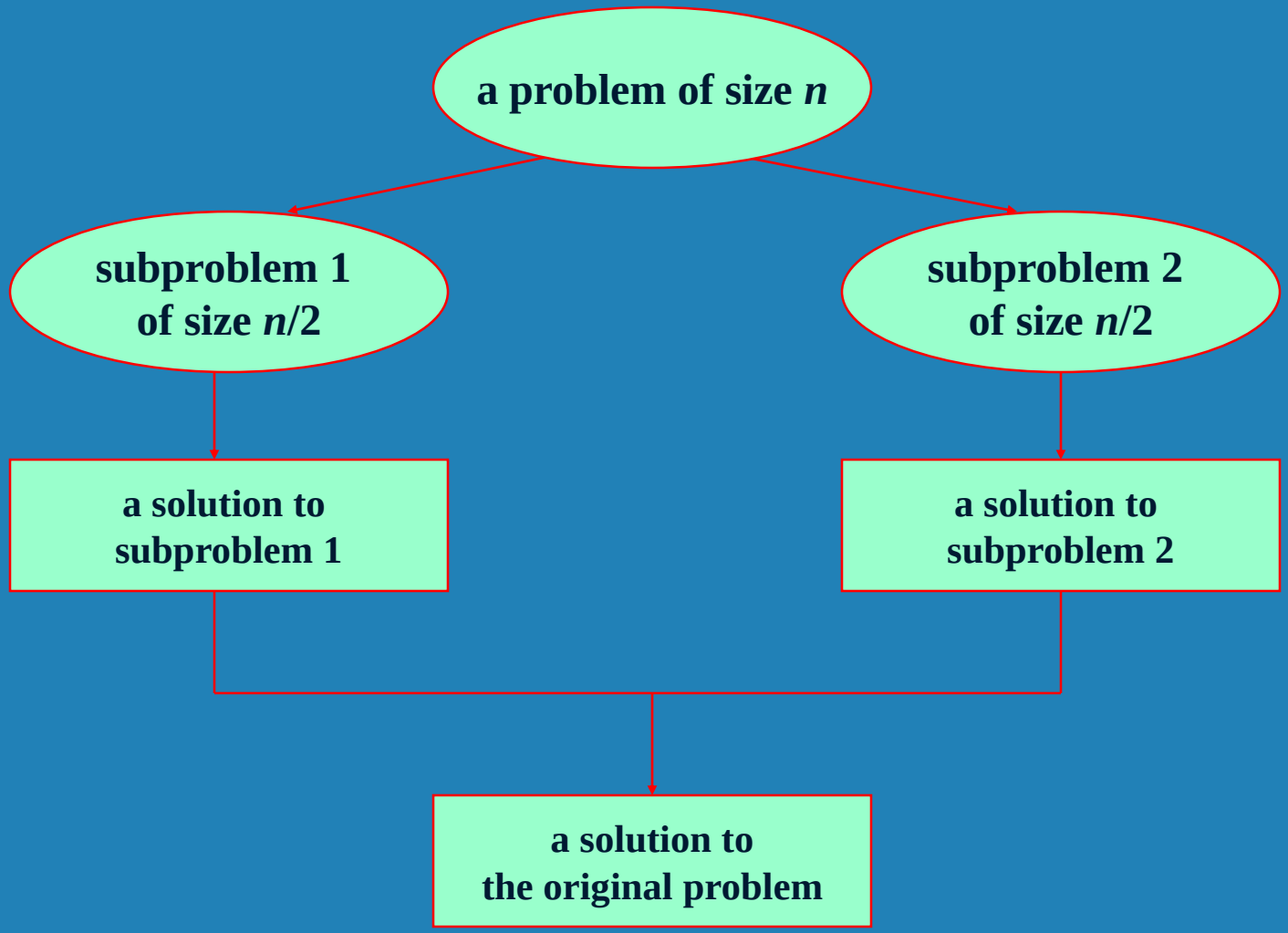


**The most-well known algorithm design strategy:**

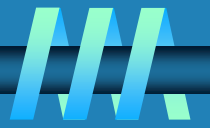
1. **Divide instance of problem into two or more smaller instances**
2. **Solve smaller instances recursively**
3. **Obtain solution to original (larger) instance by combining these solutions**



# Divide-and-Conquer Technique (cont.)



# Divide-and-Conquer Examples



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

---

- Binary search: decrease-by-half (or degenerate divide&conq.)

# General Divide-and-Conquer Recurrence

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

**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:** $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

# Mergesort

- Split array A[0..*n*-1] in two 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*$(A[0..n-1])$

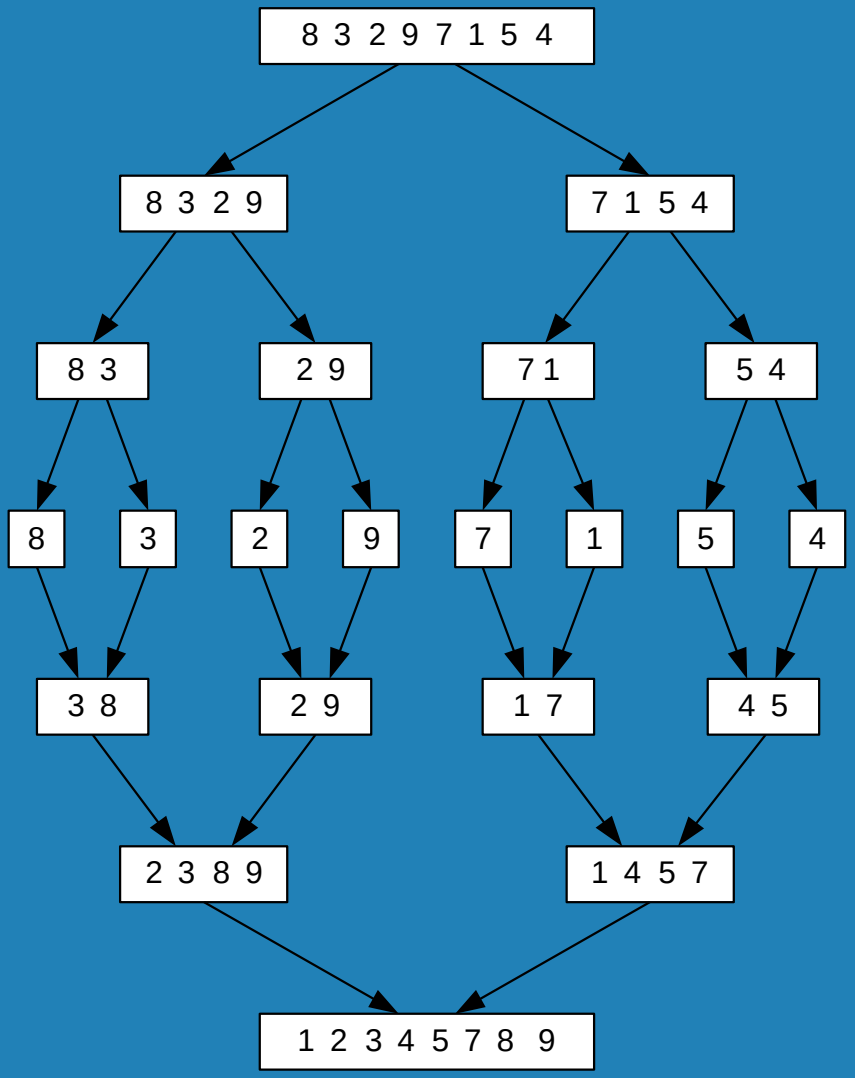
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
**if** $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    *Mergesort*$(B[0..\lfloor n/2 \rfloor - 1])$
    *Mergesort*$(C[0..\lceil n/2 \rceil - 1])$
    *Merge*$(B, C, A)$

# Pseudocode of Merge

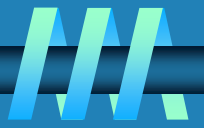
**ALGORITHM** *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//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 $A[k..p+q-1]$

# Mergesort Example



# Analysis of Mergesort



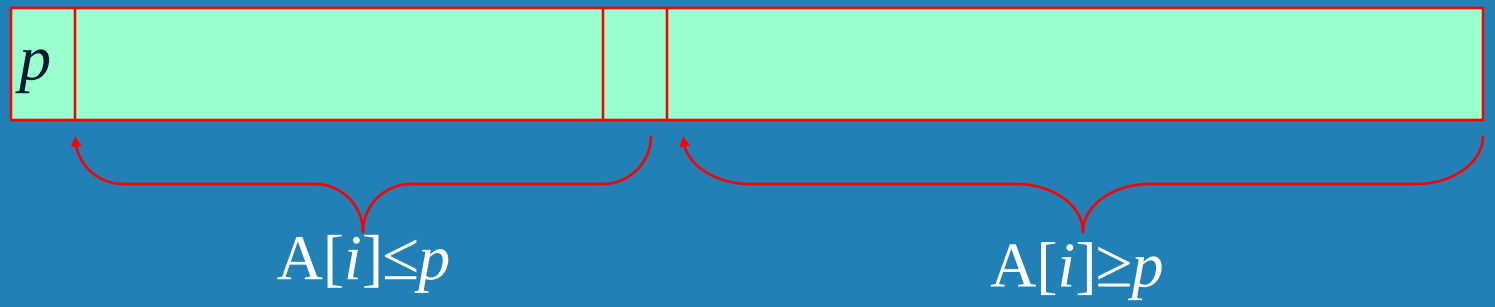
- All cases have same efficiency: $\Theta(n \log n)$
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

- Space requirement: $\Theta(n)$ (<u>not</u> 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)

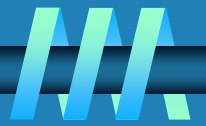



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

# Hoare's Partitioning Algorithm

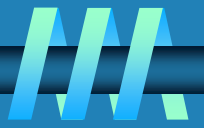
**Algorithm** *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
//  indices $l$ and $r$ $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
//  this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
**repeat**
  **repeat** $i \leftarrow i+1$ **until** $A[i] \geq p$
  **repeat** $j \leftarrow j-1$ **until** $A[j] \cdot p$
  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** $j$

# Quicksort Example



5 3 1 9 8 2 4 7

# Analysis of Quicksort



- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- 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 recursion

  These combine to 20-25% improvement

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

# Binary Tree Algorithms

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

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

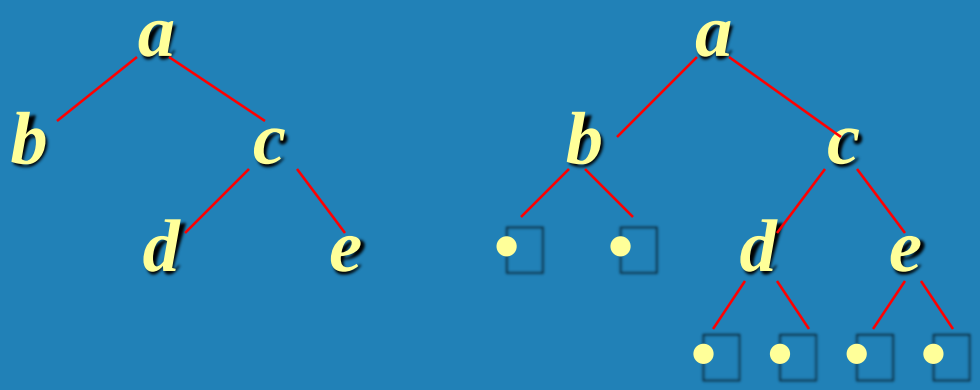
**Algorithm *Inorder*(*T*)**

**if** $T \neq \varnothing$

  *Inorder*($T_{left}$)

  **print(root of *T*)**

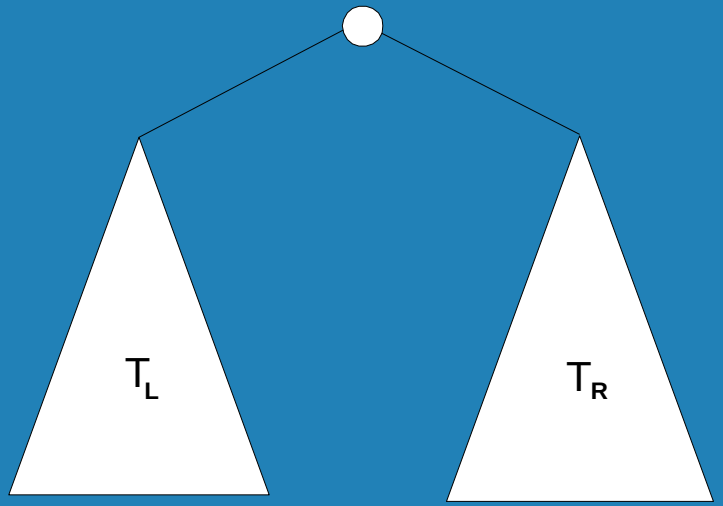
  *Inorder*($T_{right}$)




**Efficiency: Θ(*n*)**

# Binary Tree Algorithms (cont.)

**Ex. 2: Computing the height of a binary tree**




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

**Efficiency: $\Theta(n)$**

# Multiplication of Large Integers

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

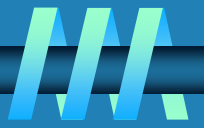
**A = 12345678901357986429 B = 87654321284820912836**

**The grade-school algorithm:**

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

**Efficiency: $n^2$ one-digit multiplications**

# First Divide-and-Conquer Algorithm



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

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

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

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

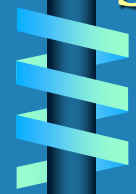
**In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are $n$-digit, $A_1$, $A_2$, $B_1$, $B_2$ are $n/2$-digit numbers),**

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

**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

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

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

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$$

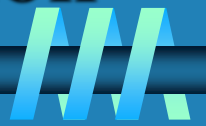
I.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$ which requires only 3 multiplications at the expense of (4-1) extra add/sub.

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

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

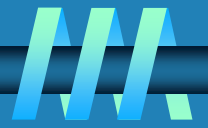
Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

# Example of Large-Integer Multiplication



**2135 * 4014**

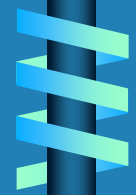
# Strassen's Matrix Multiplication



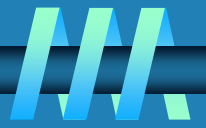
**Strassen observed [1969] that the product of two matrices can be computed as follows:**

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$



# Formulas for Strassen's Algorithm



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

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

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

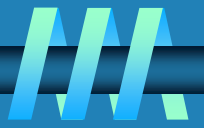
$$M_4 = A_{11} * (B_{10} - B_{00})$$

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

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

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

# Analysis of Strassen's Algorithm



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

**Number of multiplications:**

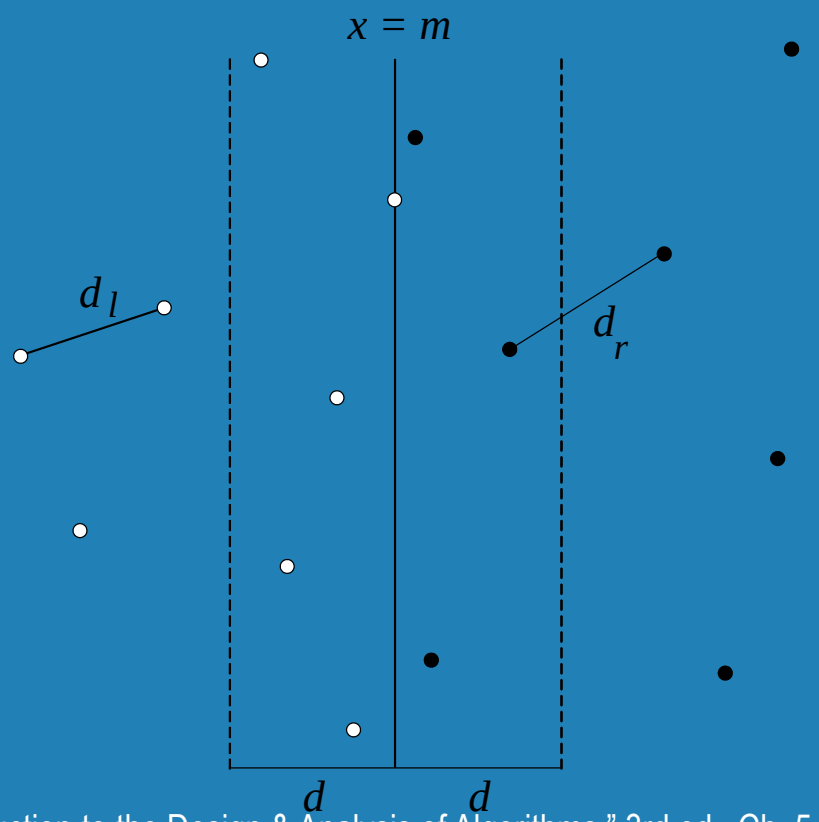
$$M(n) = 7M(n/2), \quad M(1) = 1$$

**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.**

# Closest-Pair Problem by Divide-and-Conquer

**Step 1** Divide the points given into two subsets $P_l$ and $P_r$ by a vertical line $x = m$ 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.)

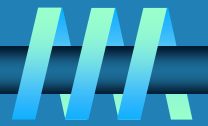
**Step 2** Find recursively the closest pairs for the left and right subsets.

**Step 3** Set $d = \min\{d_l, d_r\}$

We can limit our attention to the points in the symmetric vertical strip $S$ of width $2d$ as possible closest pair. (The points are stored and processed in increasing order of their $y$ coordinates.)

**Step 4** Scan the points in the vertical strip $S$ from the lowest up. For every point $p(x,y)$ in the strip, inspect points in in the strip that may be closer to $p$ than $d$. There can be no more than 5 such points following $p$ on the strip list!

# Efficiency of the Closest-Pair Algorithm



**Running time of the algorithm is described by**

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

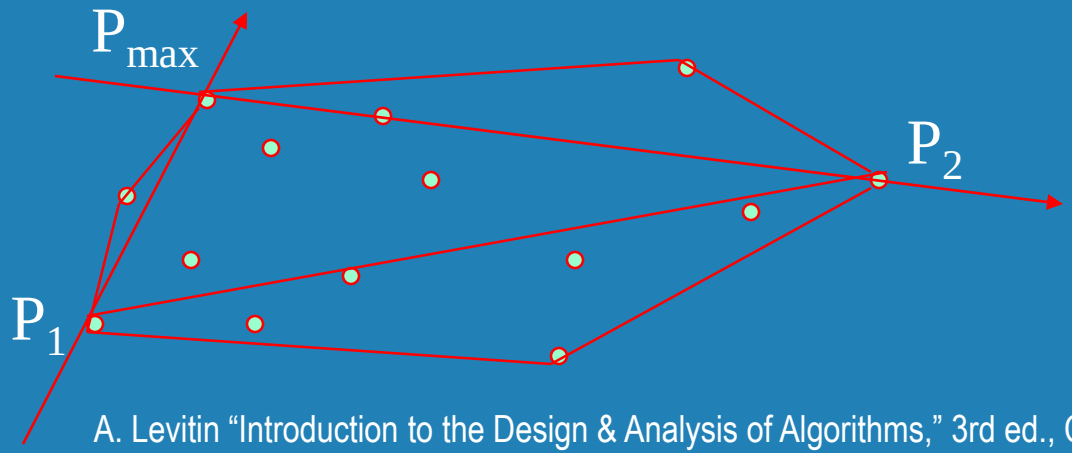
**By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)**

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

# Quickhull Algorithm

*Convex hull*: smallest convex set that includes given points

- 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:
  - worst case: $\Theta(n^2)$ (as quicksort)
  - 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