Algorithms and Their Applications
- Divide-and-Conquer -

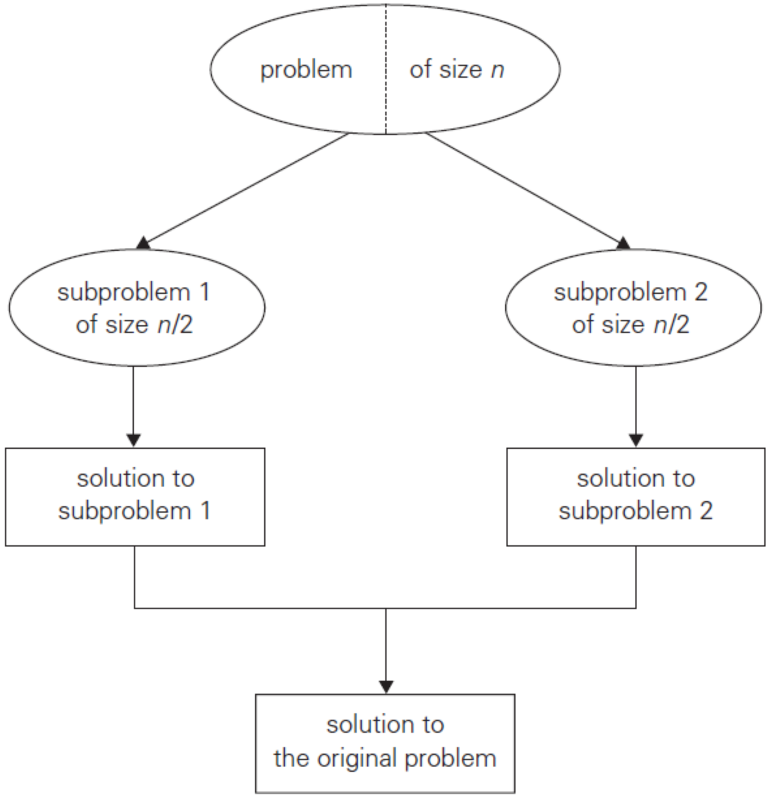
Won-Yong Shin

April 20th, 2020

연세대학교
YONSEI UNIVERSITY

Divide-and-Conquer

- The most-well known algorithm design strategy:
  - 1. Divide an instance of problem into two or more smaller instances
  - 2. Solve smaller instances *recursively*
  - 3. Obtain a solution to the original (larger) instance by combining these solutions


problem
of size *n*
subproblem 1 of size *n*/2
subproblem 2 of size *n*/2
solution to subproblem 1
solution to subproblem 2
solution to the original problem

General Divide-and-Conquer Recurrence (1/2)

- Recurrence for the running time $T(n)$:
  - **$T(n) = aT(n/b) + f(n)$** where $f(n) \in \Theta(n^d)$, $d \geq 0$

- **Master theorem**

$$\text{If } a < b^d, \quad T(n) \in \Theta(n^d)$$

$$\text{If } a = b^d, \quad T(n) \in \Theta(n^d \log n)$$

$$\text{If } a > b^d, \quad T(n) \in \Theta(n^{\log_b a})$$

  - Note: The same results hold with O instead of Θ
  - Brief sketch of the proof:
    - **Step 1**: $n = b^k$, $k = 1, 2, \ldots$
    - **Step 2**: $T(b^k) = aT(b^{k-1}) + f(b^k)$

$$= a[aT(b^{k-2}) + f(b^{k-1})] + f(b^k) = a^2T(b^{k-2}) + af(b^{k-1}) + f(b^k)$$

$$= \cdots$$

$$= a^k[T(1) + \sum_{j=1}^{k} f(b^j)/a^j]$$

    - **Step 3**: $T(n) = n^{\log_b a}[T(1) + \sum_{j=1}^{\log_b n} b^{jd}/a^j] = n^{\log_b a}[T(1) + \sum_{j=1}^{\log_b n} (b^d/a)^j]$

General Divide-and-Conquer Recurrence (2/2)

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

Divide-and-Conquer Examples

- Examples
  - Sorting: mergesort and quicksort
  - The recursion-tree methods for solving recurrences
  - Multiplication of large integers
  - Matrix multiplication: Strassen's algorithm
  - Closest-pair and convex-hull algorithms

Mergesort

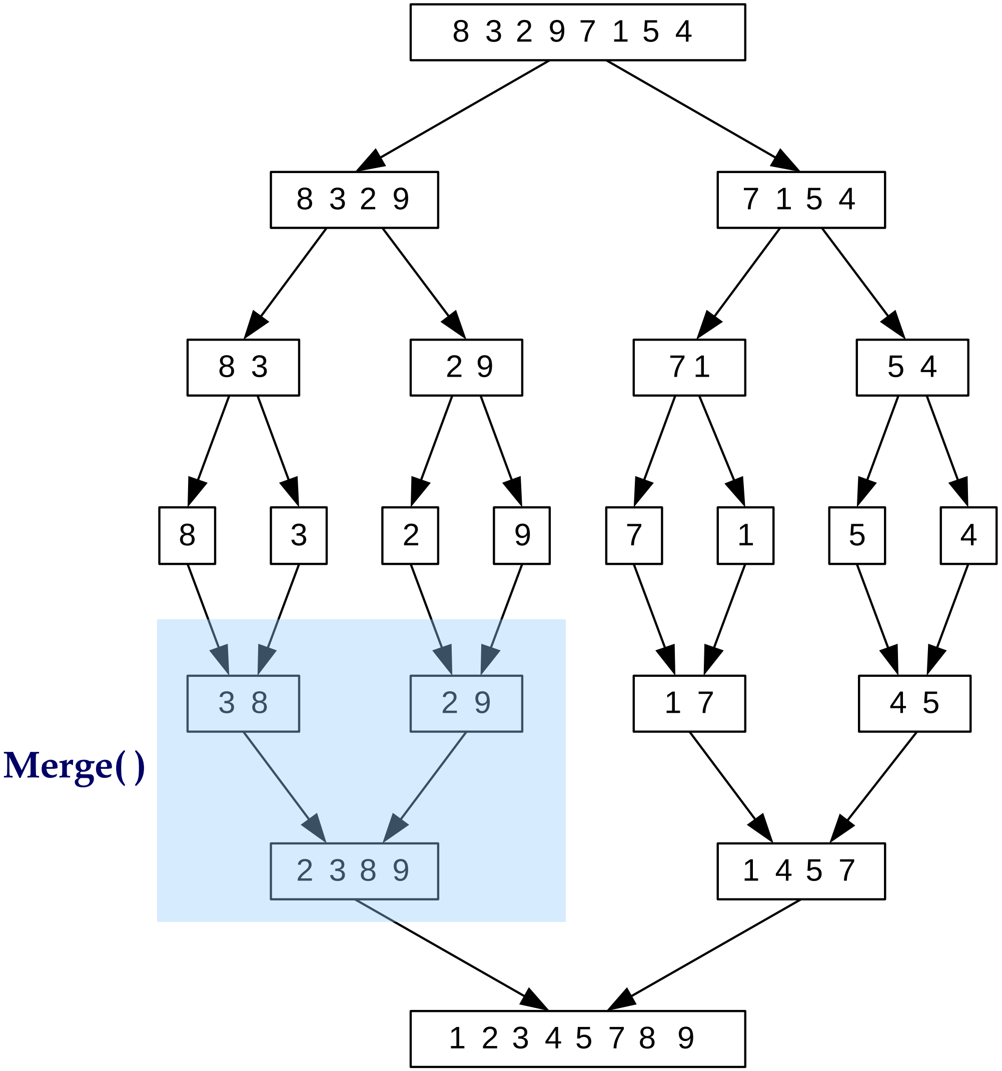
- **Step 1**
  - Split array *A*[0..*n*-1] in <u>two about equal halves</u> and make copies of each half in arrays *B* and *C*

- **Step 2**
  - Sort arrays *B* and *C* *recursively*

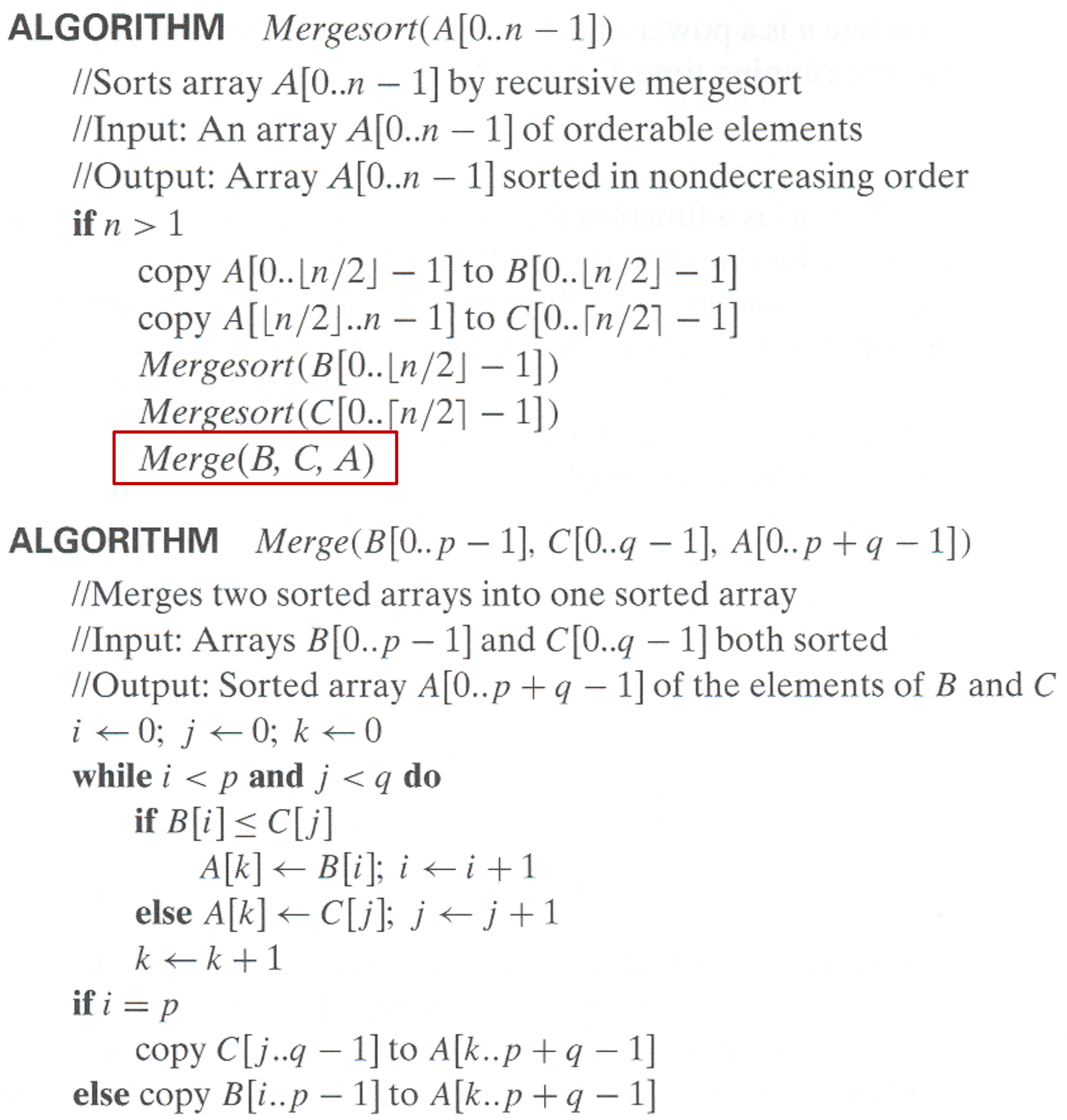
- **Step 3**
  - Merge sorted arrays *B* and *C* into array *A* as follows:
    - Repeat the following until no elements remain in one of the arrays:
      - ✓ Compare the first elements in the remaining unprocessed portions of the arrays
      - ✓ Copy the smaller of the two into *A*, while incrementing the index indicating the unprocessed portion of that array

- **Step 4**
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into *A*

Mergesort Example



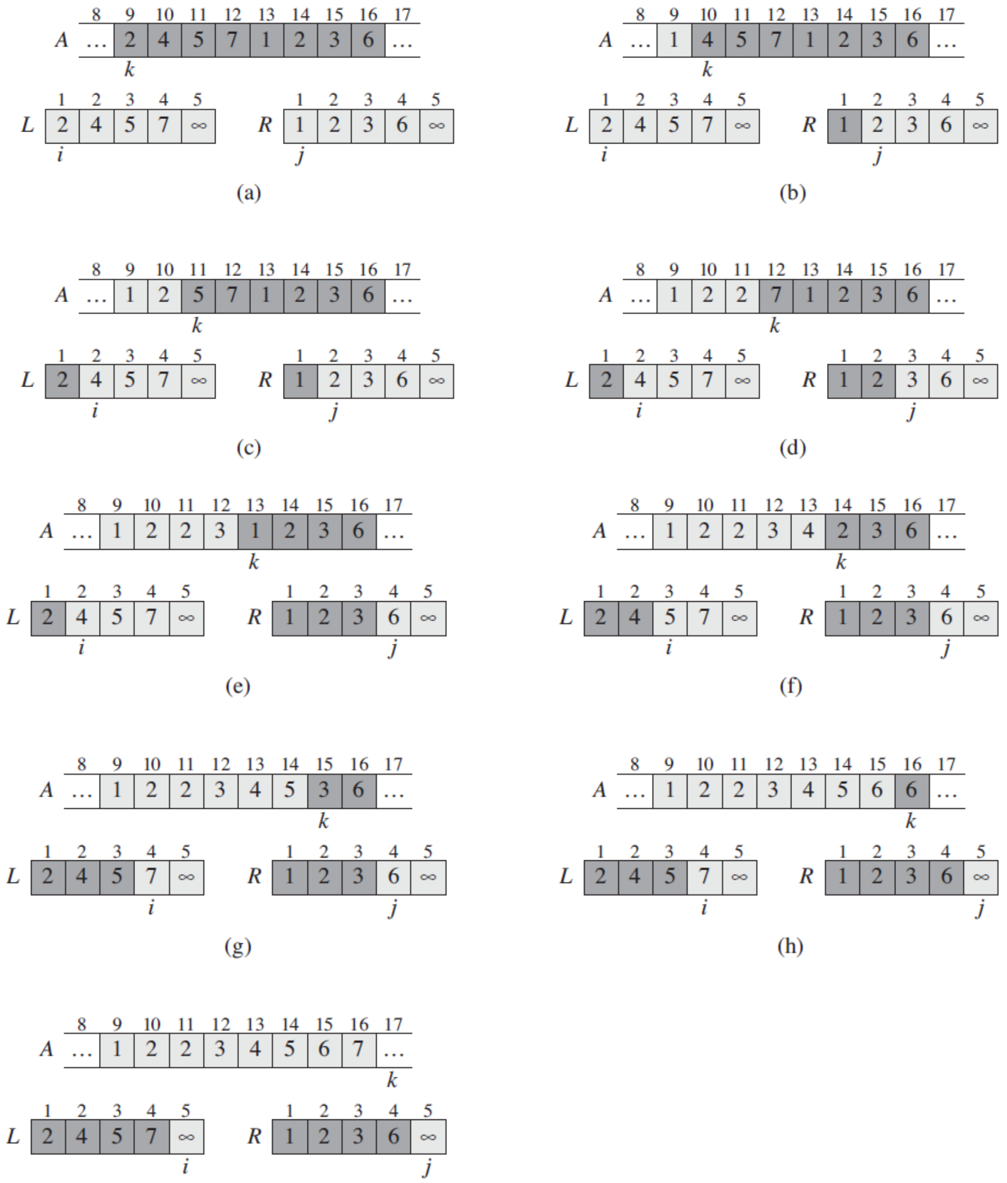
Pseudocode of Mergesort



```
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 $C[0..\lceil n/2 \rceil - 1]$
        $Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
        $Mergesort(C[0..\lceil n/2 \rceil - 1])$
        $\boxed{Merge(B, C, A)}$
```

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

The Operation of **Merge**( )



(a) (b) (c) (d) (e) (f) (g) (h)

Analysis of Mergesort

- “Worst-case” efficiency:
  - Assume that *n* is a power of 2
  - The number of **key comparisons**:
    $C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0$
  - Worst case analysis: $C_{worst}(n) = 2C_{worst}(n/2) + n - 1$
  - **$C(n) = \Theta(n \log n)$**

- Variation of mergesort
  - Can be implemented *without recursion* (bottom-up)
  - Can divide a list to be sorted in *more than two* parts

Quicksort

- **Step 1**
  - Select a *pivot* (partitioning element) – here, the first element

- **Step 2**
  - 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*

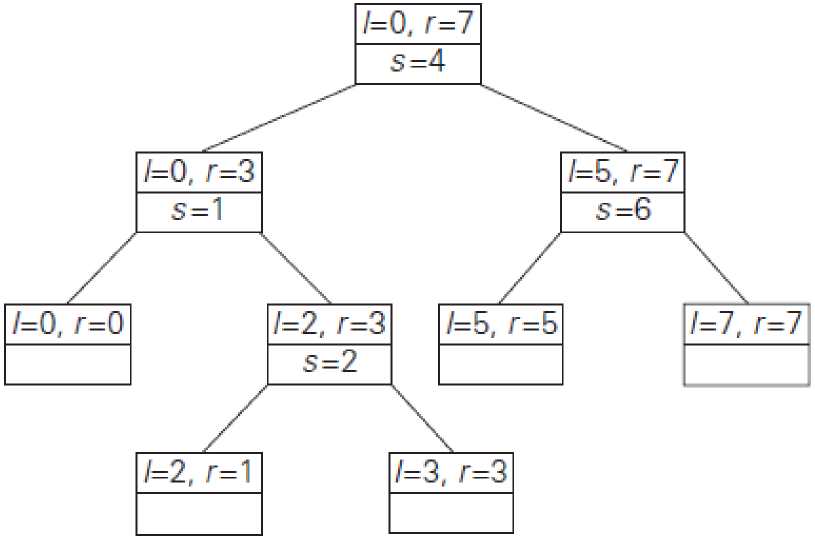
$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

- **Step 3**
  - Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position

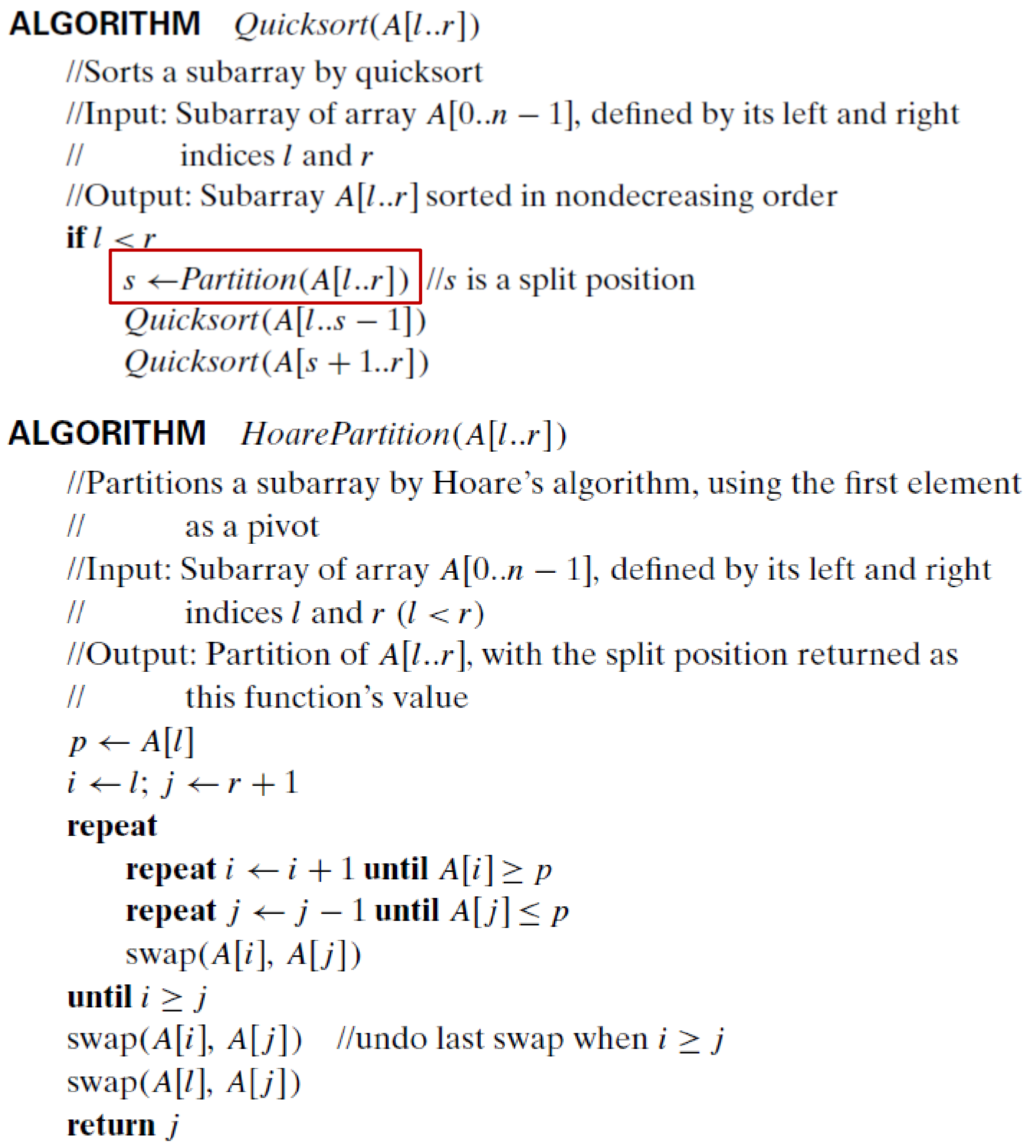
- **Step 4**
  - Sort the two subarrays *recursively*

Quicksort Example

0	1	2	3	4	5	6	7
5	i 3	1	9	8	2	4	j 7
5	3	1	i 9	8	2	j 4	7
5	3	1	i 4	8	2	j 9	7
5	3	1	4	i 8	j 2	9	7
5	3	1	4	i 2	j 8	9	7
5	3	1	4	j 2	i 8	9	7
2	3	1	4	**5**	8	9	7
2	i 3	1	j 4				
2	i 3	j 1	4				
2	i 1	j 3	4				
2	j 1	i 3	4				
1	**2**	3	4				
1							
		3	i j 4				
		j **3**	i 4				
			4				
					8	i 9	j 7
					8	i 7	j 9
					8	j 7	i 9
					7	**8**	9
					7		
							9


l=0, r=7
s=4
l=0, r=3
s=1
l=5, r=7
s=6
l=0, r=0
l=2, r=3
s=2
l=5, r=5
l=7, r=7
l=2, r=1
l=3, r=3

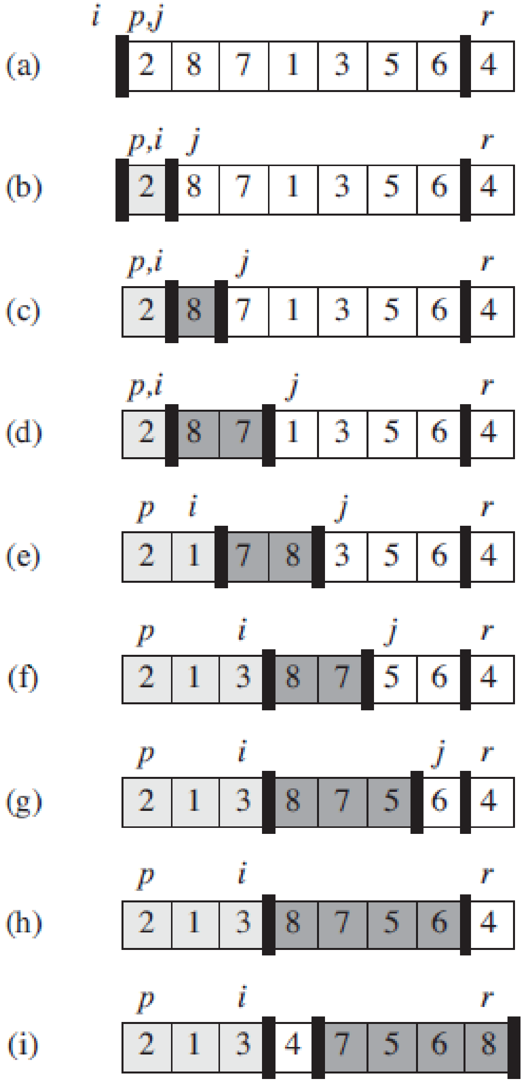
Pseudocode of Quicksort



```
ALGORITHM  $Quicksort(A[l..r])$
    //Sorts a subarray by quicksort
    //Input: Subarray of array $A[0..n-1]$, defined by its left and right
    //         indices $l$ and $r$
    //Output: Subarray $A[l..r]$ sorted in nondecreasing order
    if $l < r$
        $\boxed{s \leftarrow Partition(A[l..r])}$ //$s$ is a split position
        $Quicksort(A[l..s-1])$
        $Quicksort(A[s+1..r])$

ALGORITHM  $HoarePartition(A[l..r])$
    //Partitions a subarray by Hoare's algorithm, using the first element
    //         as a pivot
    //Input: Subarray of array $A[0..n-1]$, defined by its left and right
    //         indices $l$ and $r$ $(l < r)$
    //Output: 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 $A[j] \leq p$
        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 $j$
```

The Operation of **Partition( )**: Alternative Approach



```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Analysis of Quicksort (1/2)

- Best case efficiency:
  - When all the splits are in the *middle*
  - The number of **key comparisons**:

    $C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0$
  - $C(n) = \Theta(n \log n)$

- Worst case efficiency:
  - When all the splits are skewed to the extreme
    - Sorted array!
  - $C(n) = \Theta(n^2)$

Analysis of Quicksort (2/2)

- Average case efficiency:
  - When the partition split happen in each position *s* with the same probability
    - Random arrays
  - The number of **key comparisons**:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)]$$

  - $C(n) = \Theta(n \log n)$

- Improvements:
  - Better pivot selection: ***median-of-three*** partitioning
  - Elimination of *recursion*

→ These combine to 20-25% improvement

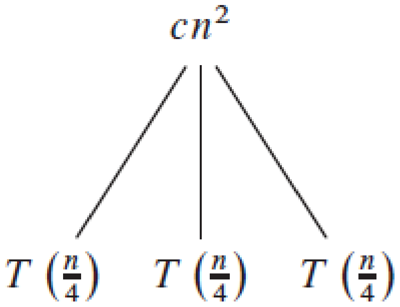
Divide-and-Conquer Examples

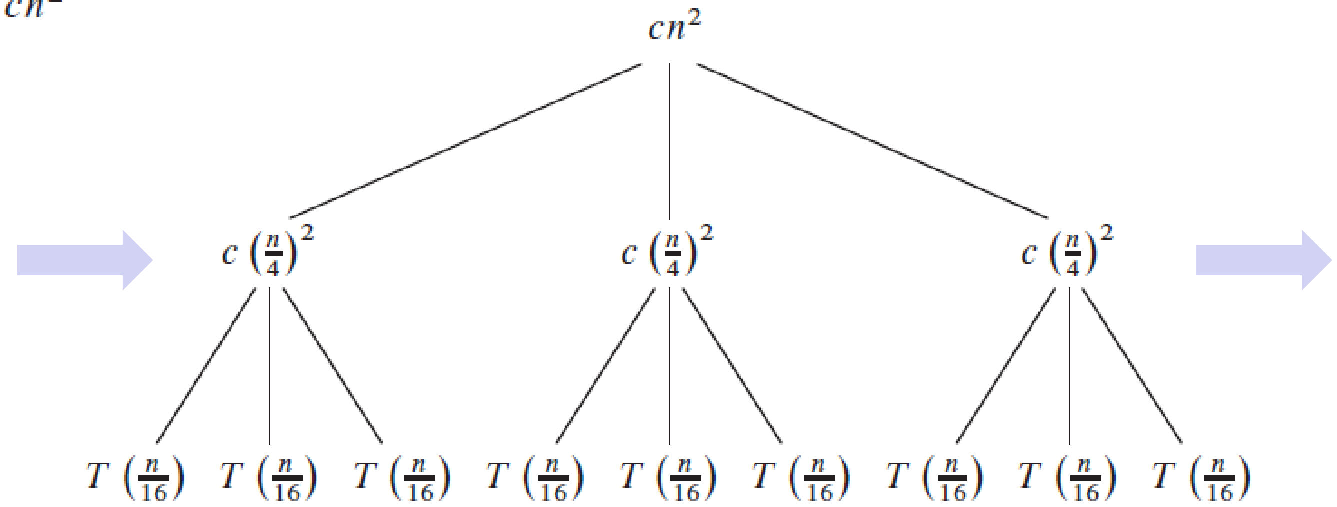
- Examples
  - Sorting: mergesort and quicksort
  - The recursion-tree methods for solving recurrences
  - Multiplication of large integers
  - Matrix multiplication: Strassen’s algorithm
  - Closest-pair algorithm
  - Convex-hull algorithm

The Recursion-Tree Method (1/2)

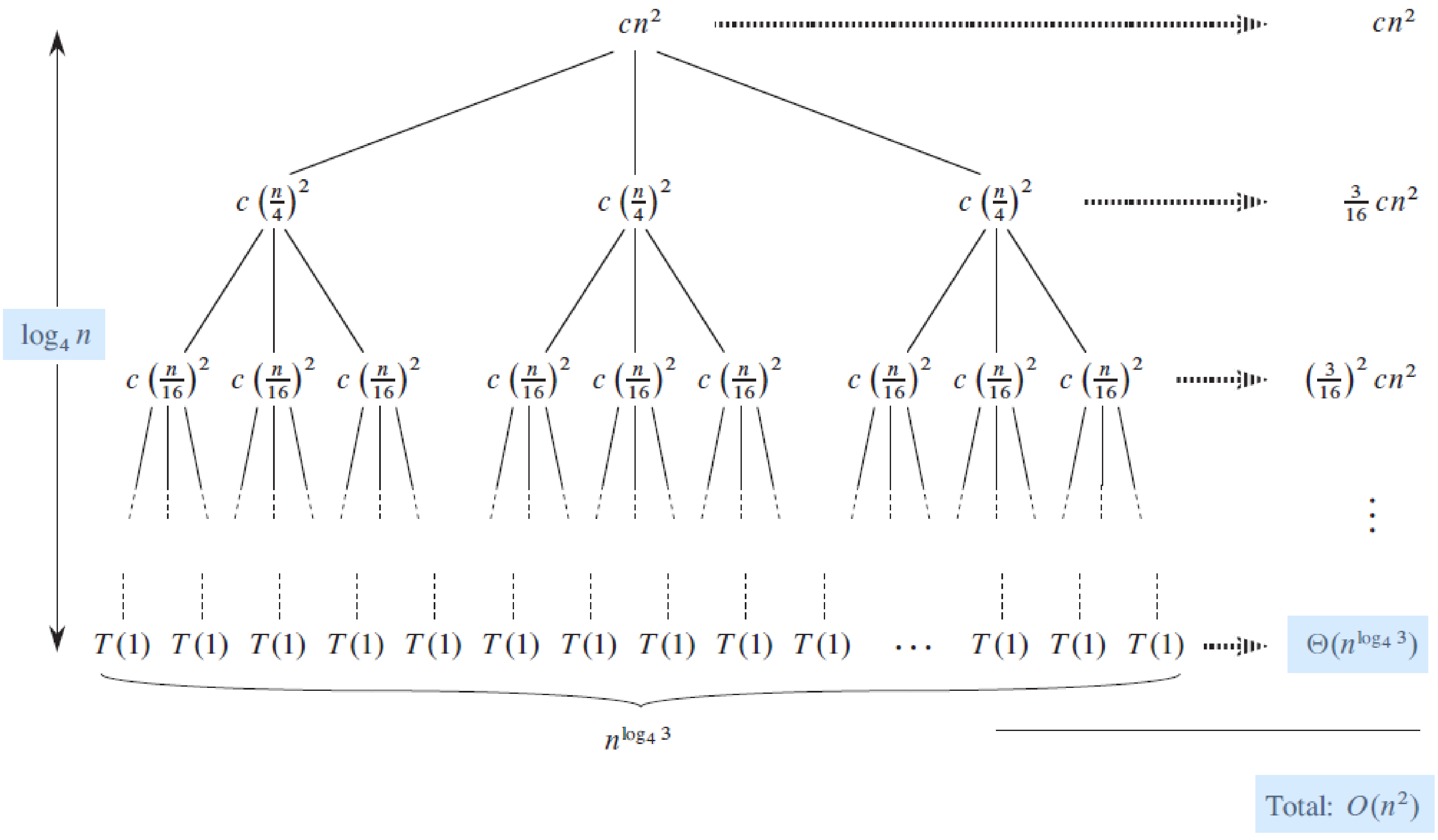
- How a *recursion tree* provides a **good guess**

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


cn^2
$T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$


cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$

The Recursion-Tree Method (2/2)


cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$T(1)$
$\dots$
$n^{\log_4 3}$
$\log_4 n$
cn^2
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\vdots$
$\Theta(n^{\log_4 3})$
Total: $O(n^2)$

The Recursion-Tree Method: Guess Work

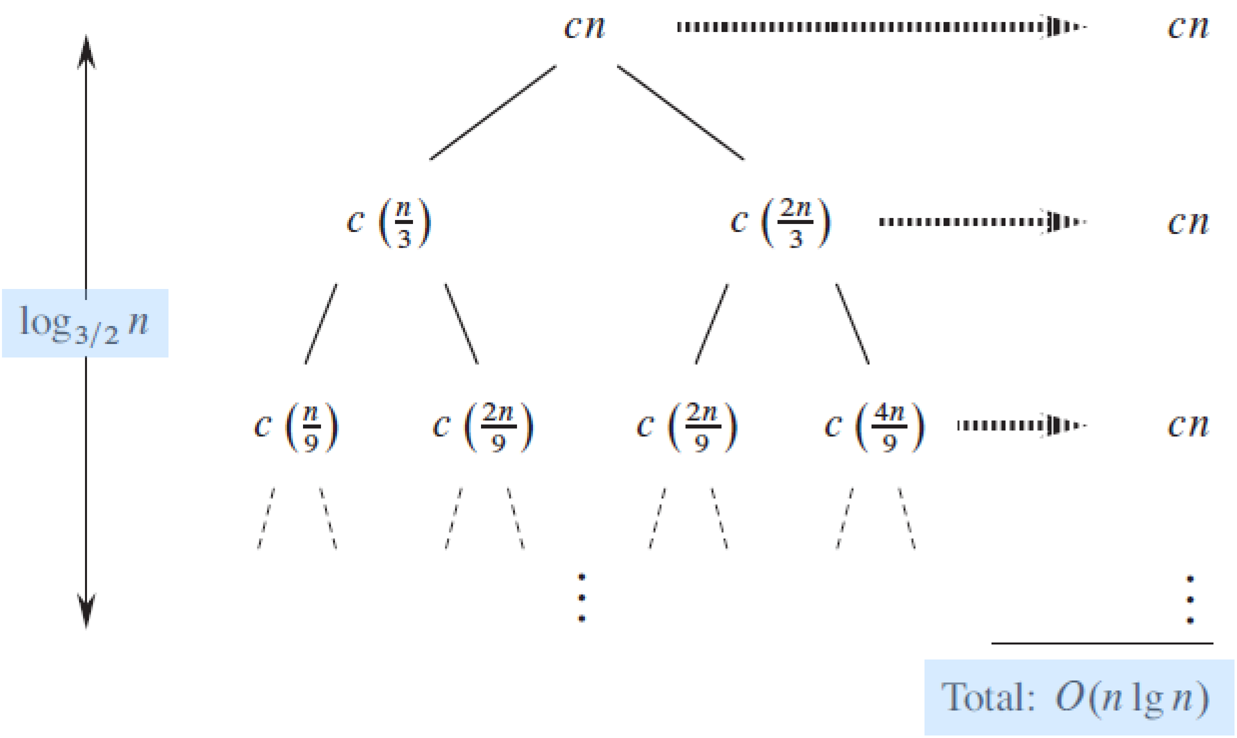
- Guess work

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2) \end{aligned}$$

The Recursion-Tree Method: Another Example

- A recursion tree for the recurrence:

$T(n) = T(n/3) + T(2n/3) + cn$


cn
cn
$c\left(\frac{n}{3}\right)$
$c\left(\frac{2n}{3}\right)$
cn
$\log_{3/2} n$
$c\left(\frac{n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{4n}{9}\right)$
cn
Total: $O(n \lg n)$

Multiplication of Large Integers

- **Algorithm 1**
  - Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

    A = 12345678901357986429 B = 87654321284820912836

    <u>The grade-school algorithm</u>:

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

  - Efficiency: n^2 one-digit multiplications

First Divide-and-Conquer Algorithm

- **Algorithm 2**
  - 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, and B_2 are $n/2$-digit numbers), then

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

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

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

    ⇨ **Solution:** $\boldsymbol{M(n) = n^2}$

Second Divide-and-Conquer Algorithm

- **Algorithm 3**
  - $A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$
  - The idea is to decrease the number of multiplications from 4 to 3:

$$\boxed{A_1 * B_2 + A_2 * B_1 = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2}$$

- which requires only 3 multiplications at the expense of extra additions/subtractions

  - The recurrence for the number of multiplications

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1$$

⇒ $M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2})$

$n = 2^k$ $\quad = \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k$

⇒ $M(n) \approx \boxed{n^{1.585}}$

  - The recurrence for the number of additions

$$A(n) = 3A(n/2) + cn \quad \text{for } n > 1, \quad A(1) = 1$$

⇒ $A(n) \approx \boxed{n^{1.585}}$

Strassen's Matrix Multiplication

- Strassen's formula:

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

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

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

- Comparison
  - **Brute-force algorithm**: 8 multiplications and 4 additions
  - **Strassen's algorithm**: 7 multiplications and 18 additions/subtractions
  - *Asymptotic* superiority as n goes to infinity

Strassen's Algorithm: Analysis

- Generalization
  - Divide *A*, *B*, and *C* into 4 $n/2$ x $n/2$ submatrices:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

- The recurrence for the number of multiplications

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1$$

$$\Rightarrow \quad M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \cdots$$

$$n = 2^k \qquad = 7^iM(2^{k-i}) \cdots = 7^kM(2^{k-k}) = 7^k$$

$$\Rightarrow \quad M(n) \approx \boxed{n^{2.807}}$$

- The recurrence for the number of additions

$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \quad A(1) = 0$$

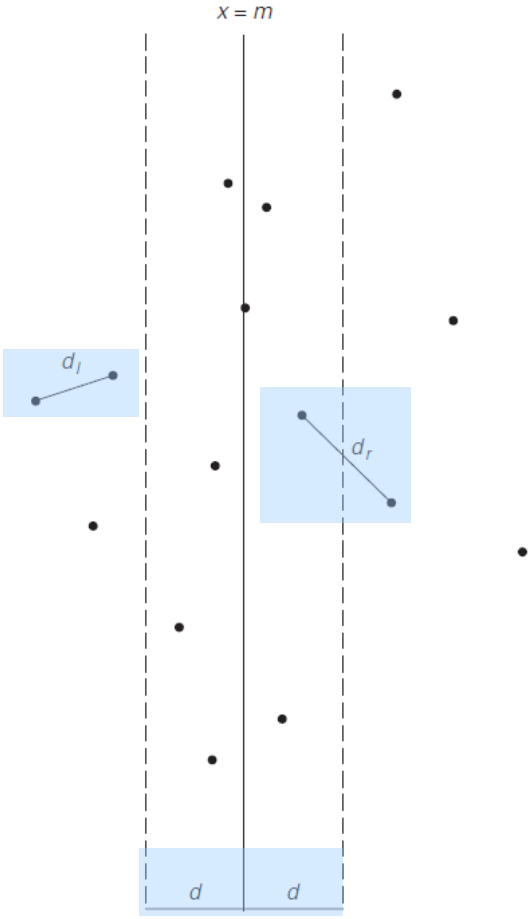
$$\Rightarrow \quad A(n) \approx \boxed{n^{2.807}}$$

Divide-and-Conquer Examples

- Examples
  - Sorting: mergesort and quicksort
  - The recursion-tree methods for solving recurrences
  - Multiplication of large integers
  - Matrix multiplication: Strassen's algorithm
  - Closest-pair algorithm
  - Convex-hull algorithm

Closest-Pair Problem by Divide-and-Conquer (1/2)

- **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



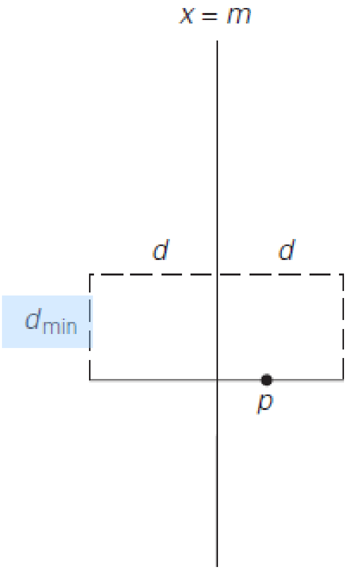
Closest-Pair Problem by Divide-and-Conquer (2/2)

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

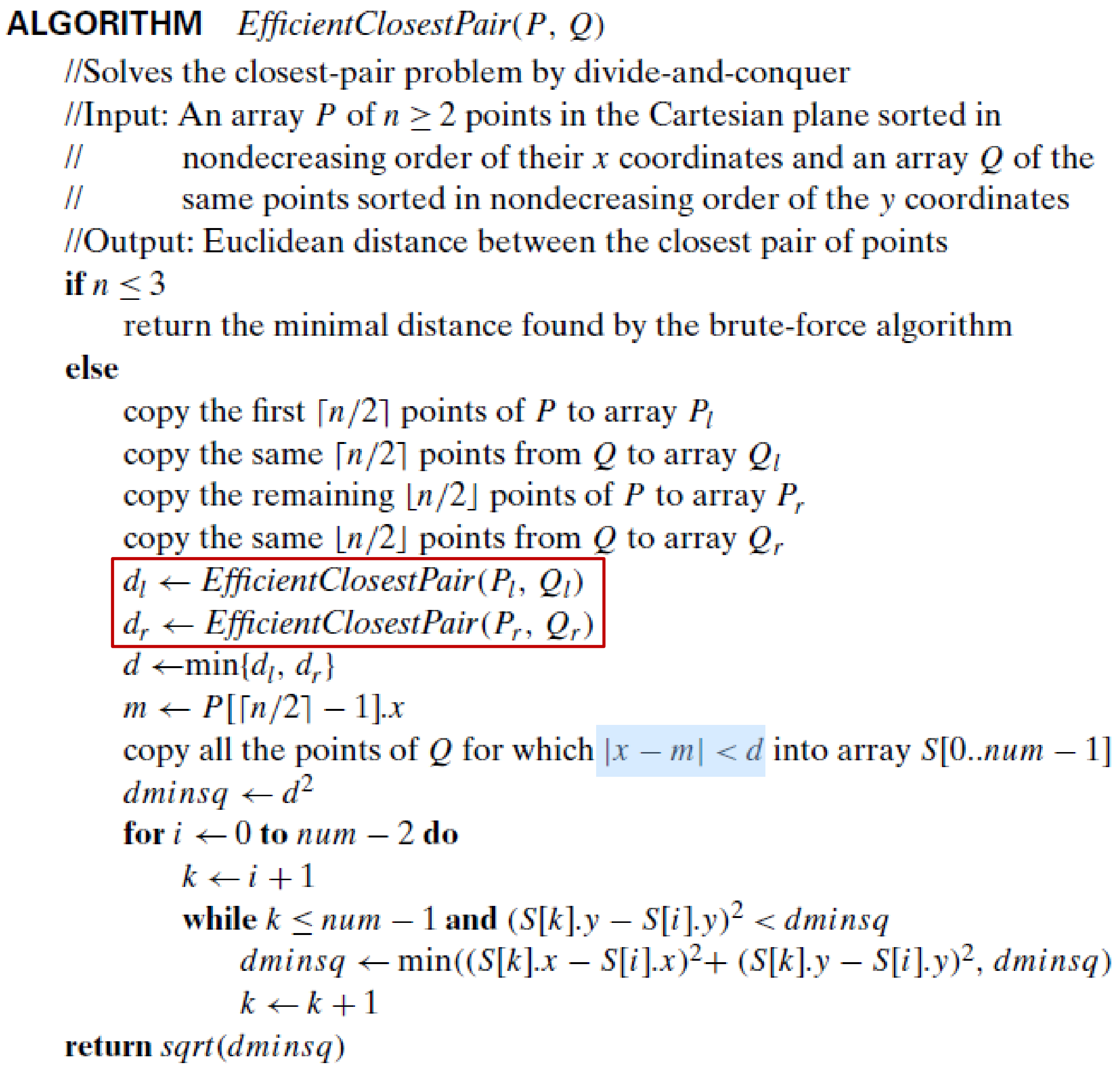
- **Step 3**
  - Set $d = \min\{d_l, d_r\}$
  - We can limit our attention to the points in the symmetric vertical strip S of width $2d$ as possible closest pair

    (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 the strip that may be closer to p than d



Pseudocode of Closest-Pair



```
ALGORITHM  EfficientClosestPair(P, Q)
    //Solves the closest-pair problem by divide-and-conquer
    //Input: An array P of $n \geq 2$ points in the Cartesian plane sorted in
    //       nondecreasing order of their x coordinates and an array Q of the
    //       same points sorted in nondecreasing order of the y coordinates
    //Output: Euclidean distance between the closest pair of points
    if $n \leq 3$
        return the minimal distance found by the brute-force algorithm
    else
        copy the first $\lceil n/2 \rceil$ points of P to array $P_l$
        copy the same $\lceil n/2 \rceil$ points from Q to array $Q_l$
        copy the remaining $\lfloor n/2 \rfloor$ points of P to array $P_r$
        copy the same $\lfloor n/2 \rfloor$ points from Q to array $Q_r$
        $d_l \leftarrow EfficientClosestPair(P_l, Q_l)$
        $d_r \leftarrow EfficientClosestPair(P_r, Q_r)$
        $d \leftarrow \min\{d_l, d_r\}$
        $m \leftarrow P[\lceil n/2 \rceil - 1].x$
        copy all the points of Q for which $|x - m| < d$ into array $S[0..num - 1]$
        $dminsq \leftarrow d^2$
        for $i \leftarrow 0$ to $num - 2$ do
            $k \leftarrow i + 1$
            while $k \leq num - 1$ and $(S[k].y - S[i].y)^2 < dminsq$
                $dminsq \leftarrow \min((S[k].x - S[i].x)^2 + (S[k].y - S[i].y)^2, dminsq)$
                $k \leftarrow k + 1$
    return $sqrt(dminsq)$
```

Linear in *n*

Efficiency of the Closest-Pair Algorithm

- Recurrence:

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

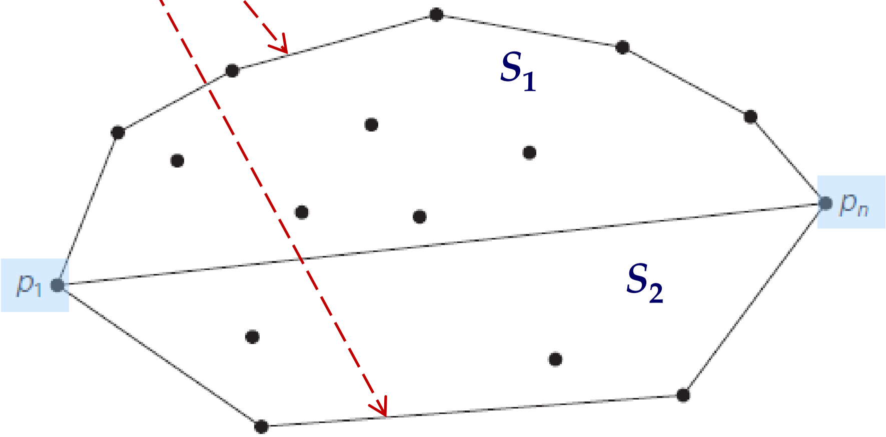
- By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

⇒ **Solution: $T(n) \in \Theta(n \log n)$**

- Comparison with the brute-force algorithm

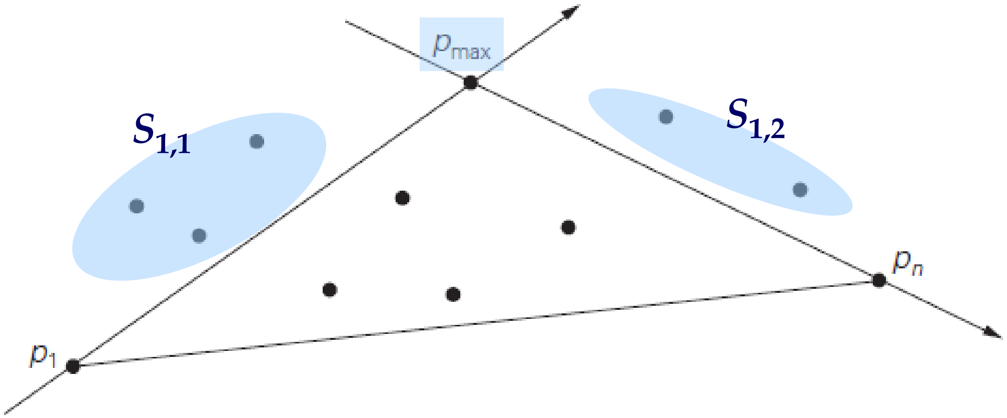
Convex-Hull Problem

- Convex hull
  - Smallest convex set that includes given points
  - Assume that points are sorted by *x*-coordinate values
  - Identify *extreme points* P_1 and P_n (*leftmost* and *rightmost*)
  - Compute *upper hull* recursively
  - Compute *lower hull* in a similar manner



Quickhull

- Compute *upper hull* recursively
  - Find point $\boldsymbol{P}_{\mathbf{max}}$ that is farthest away from line P_1P_n
  - 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 $P_{\max}P_n$
- Compute *lower hull* in a similar manner



Efficiency of the Quickhull Algorithm

- Time efficiency:
  - Finding point farthest away from line P_1P_n can be done in *linear* time
  - **Worst case**: $\Theta(n^2)$ (same as quicksort)
  - **Average case**: $\Theta(n)$ (under the reasonable assumption that distribution of points given is *uniform*)
    - Check with the Master Theorem by finding *a*, *b*, and *d*

- Discussion on efficiencies
  - 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

- Comparison with the brute-force algorithm

Graham’s Scan

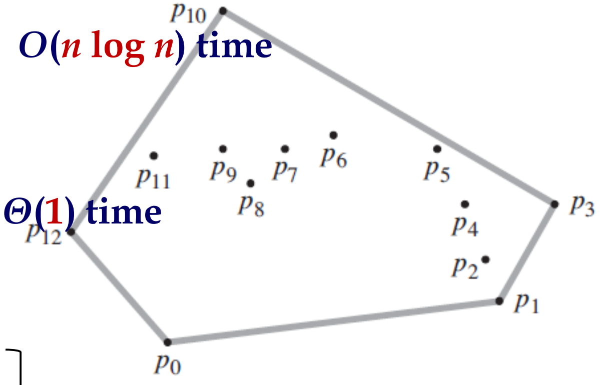
```
GRAHAM-SCAN($Q$)
let $p_0$ be the point in $Q$ with the minimum $y$-coordinate,
        or the leftmost such point in case of a tie
let $\langle p_1, p_2, \ldots, p_m \rangle$ be the remaining points in $Q$,
        sorted by polar angle in counterclockwise order around $p_0$
        (if more than one point has the same angle, remove all but
        the one that is farthest from $p_0$)
let $S$ be an empty stack
PUSH($p_0, S$)
PUSH($p_1, S$)
PUSH($p_2, S$)
for $i = 3$ to $m$
        while the angle formed by points NEXT-TO-TOP($S$), TOP($S$),
                    and $p_i$ makes a nonleft turn
                POP($S$)
        PUSH($p_i, S$)
return $S$
```

$O(n)$ time

$O(n \log n)$ time

$\Theta(1)$ time

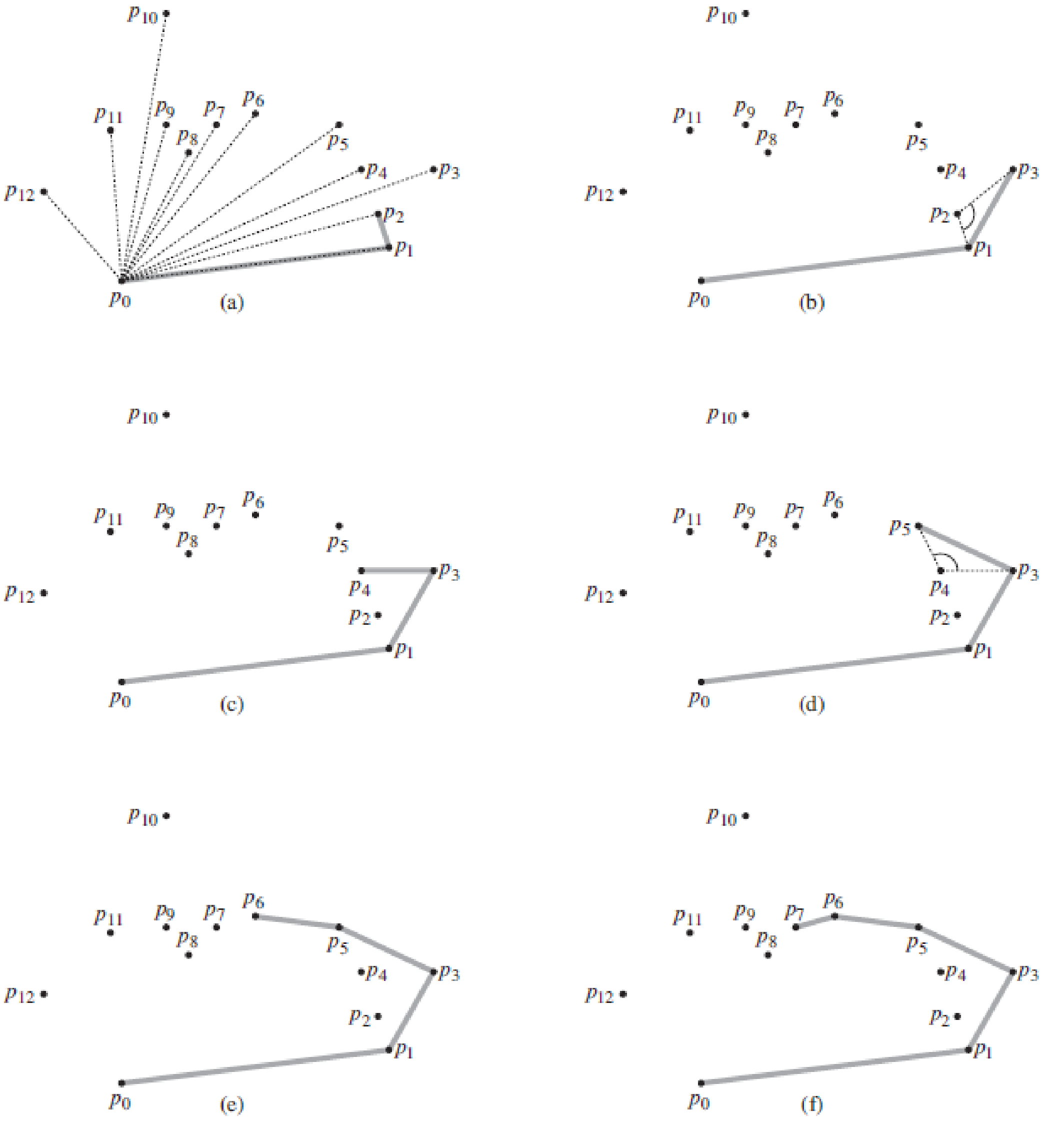
$O(n)$ time



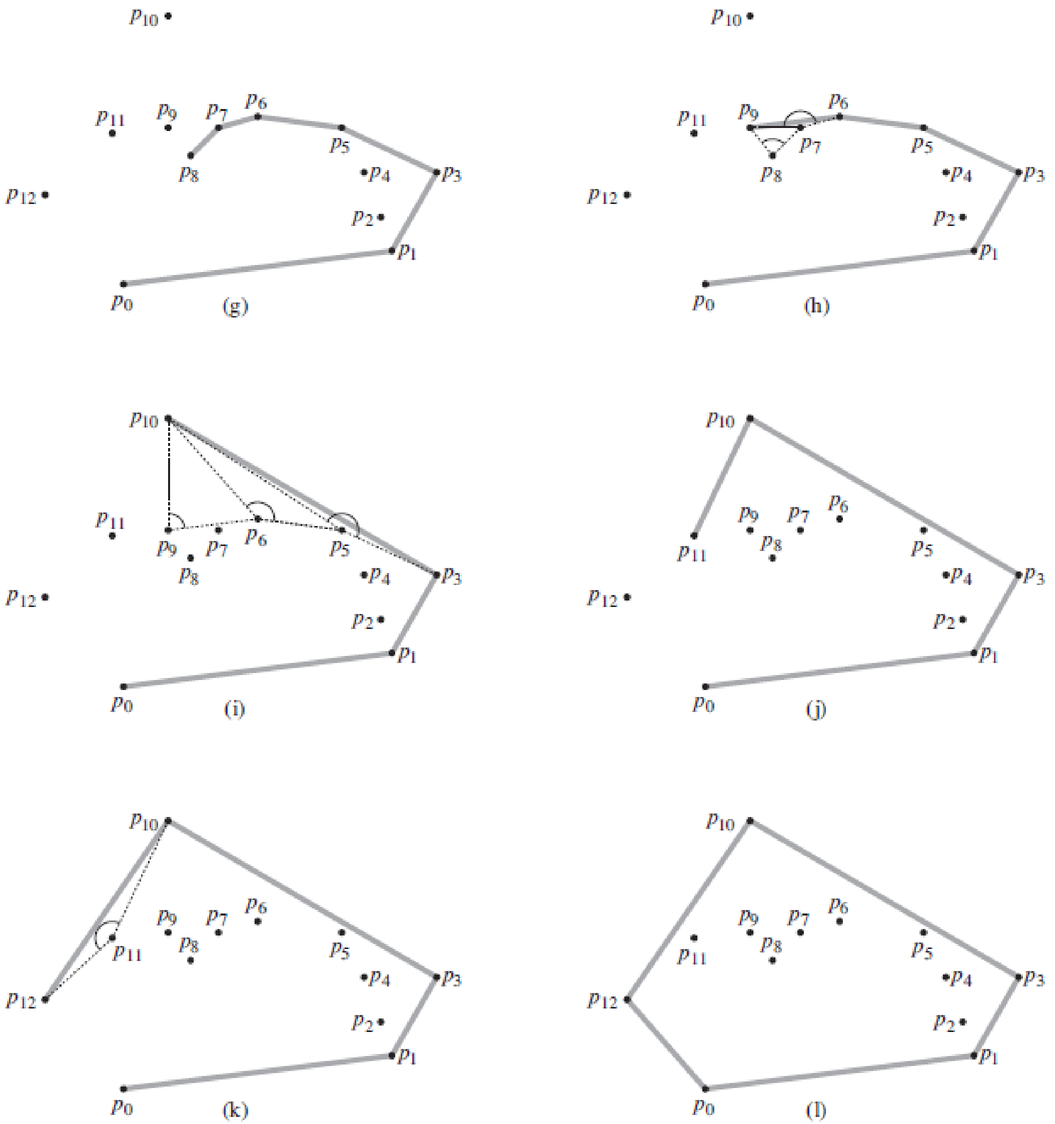
$Q = \{p_0, p_1, \ldots, p_{12}\}$

Total: $O(n \log n)$ time

The Execution of Graham-Scan (1/2)



The Execution of Graham-Scan (2/2)



(g) (h) (i) (j) (k) (l)