بسم الله الرحمن الرحیم

ساختمان‌های داده

جلسه ۲۷

مجتبی خلیلی
دانشکده برق و کامپیوتر
دانشگاه صنعتی اصفهان

مسئله مرتب‌سازی

- مسئله مرتب‌سازی:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

مسئله مرتب‌سازی

- مسئله مرتب‌سازی:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- مرتب‌سازی مقایسه‌ای:

A comparison sort uses only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, it performs one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order.

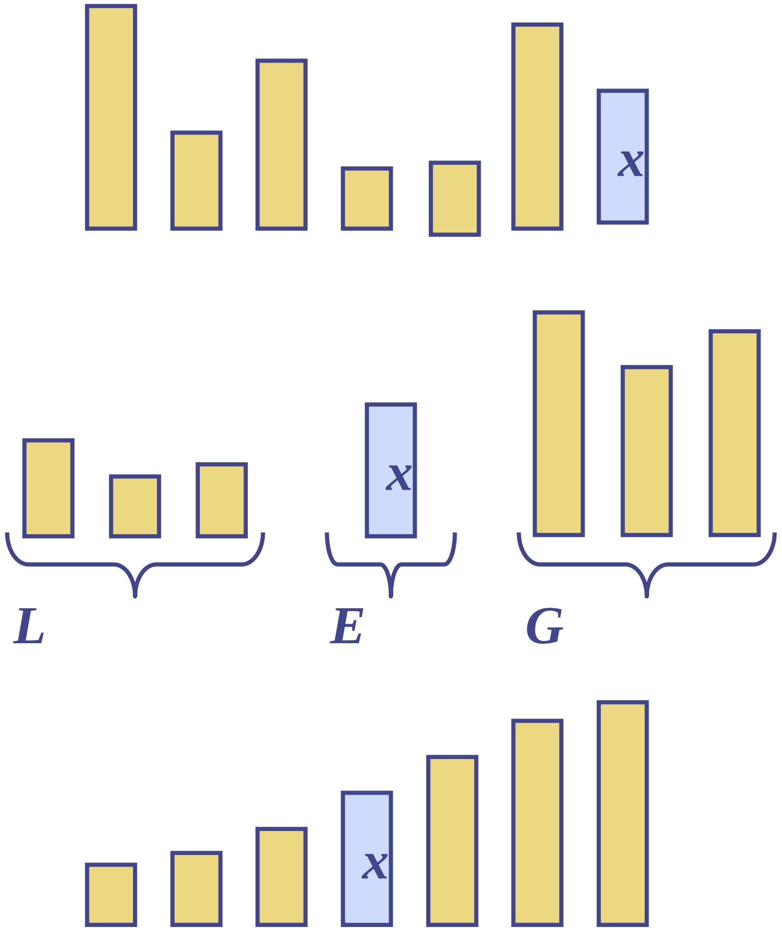
مسئله مرتب‌سازی

- مرتب‌سازی غیرمقایسه‌ای:
  - بدون مقایسه کلیدهای عناصر باهم، مرتب‌سازی میکند.
  - مانند شمارشی (counting)، مبنایی (radix) و سطلی (bucket)
- پیچیدگی این الگوریتمها؟

Quick-Sort

Quick-Sort

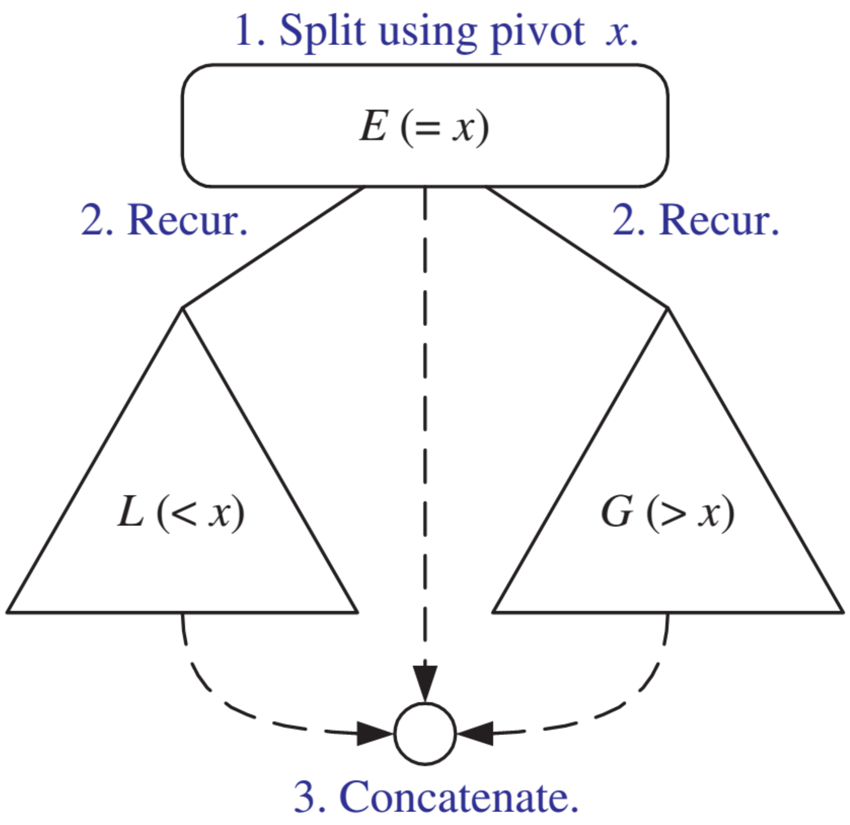
- Quick-sort is a sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: select a specific element x, called pivot, (common selection is the last element) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



Quick-Sort

IUT-ECE

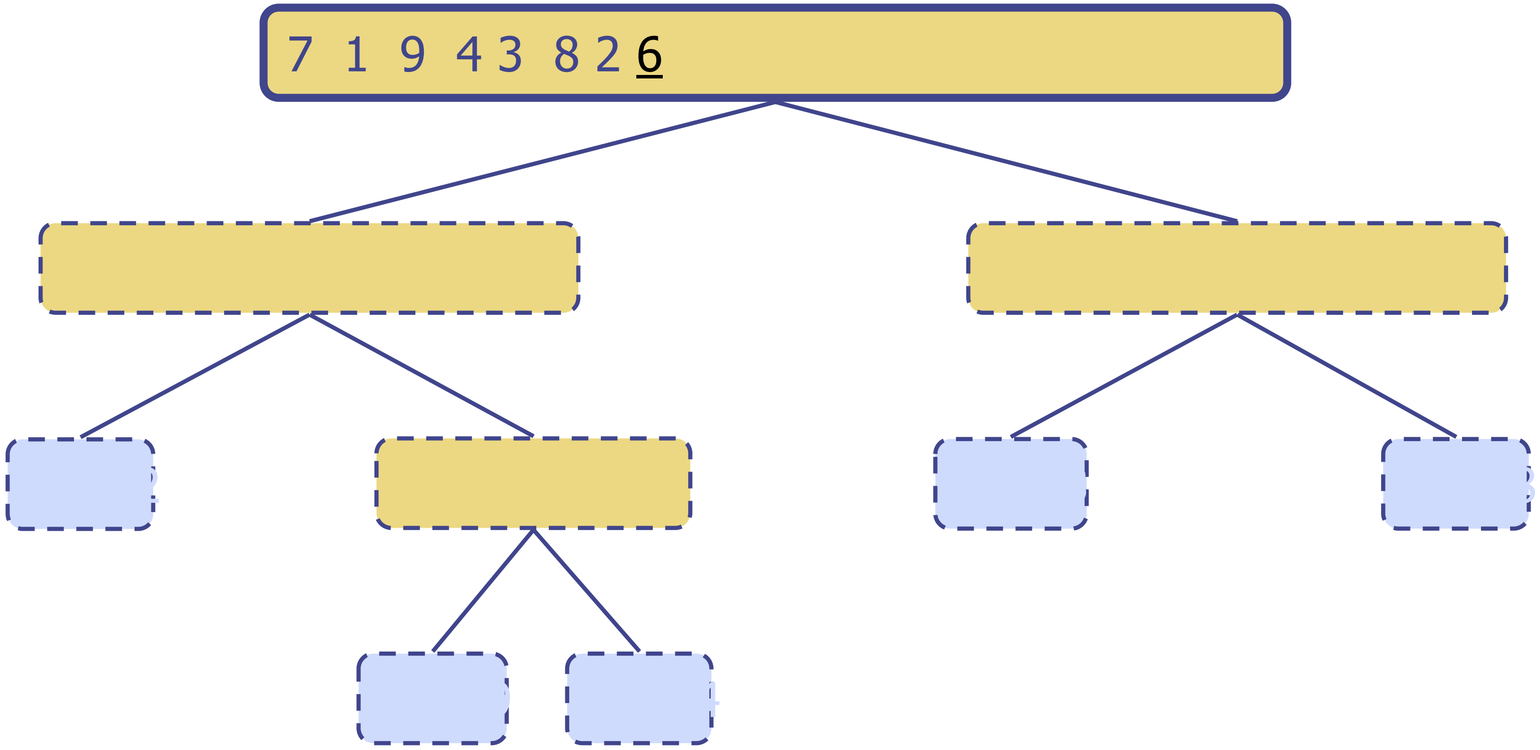
QUICKSORT (A)
  (L, E, G) ← PARTITION (A)
  QUICKSORT (L)
  QUICKSORT (G)



Execution Example

IUT-ECE

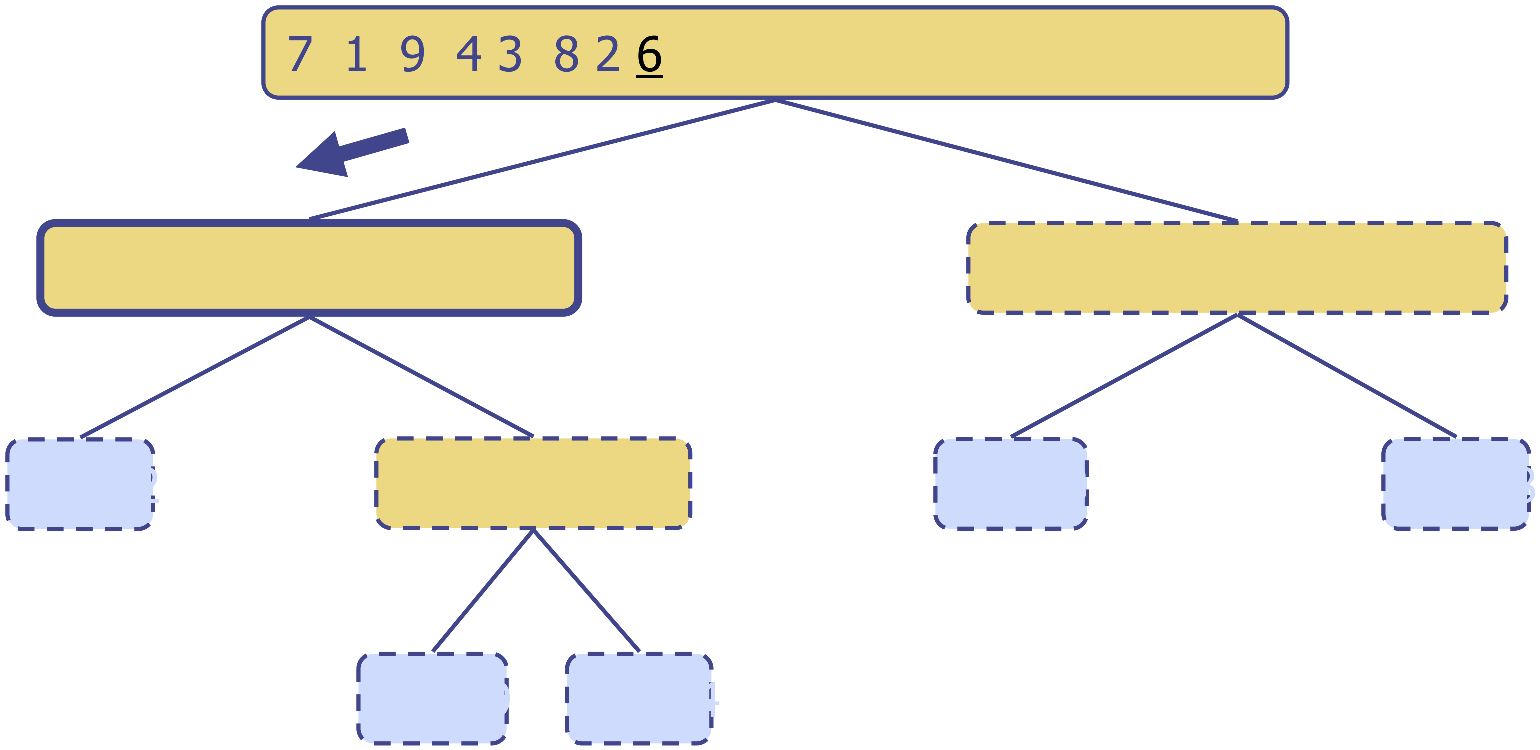
- Pivot selection



Execution Example (cont.)

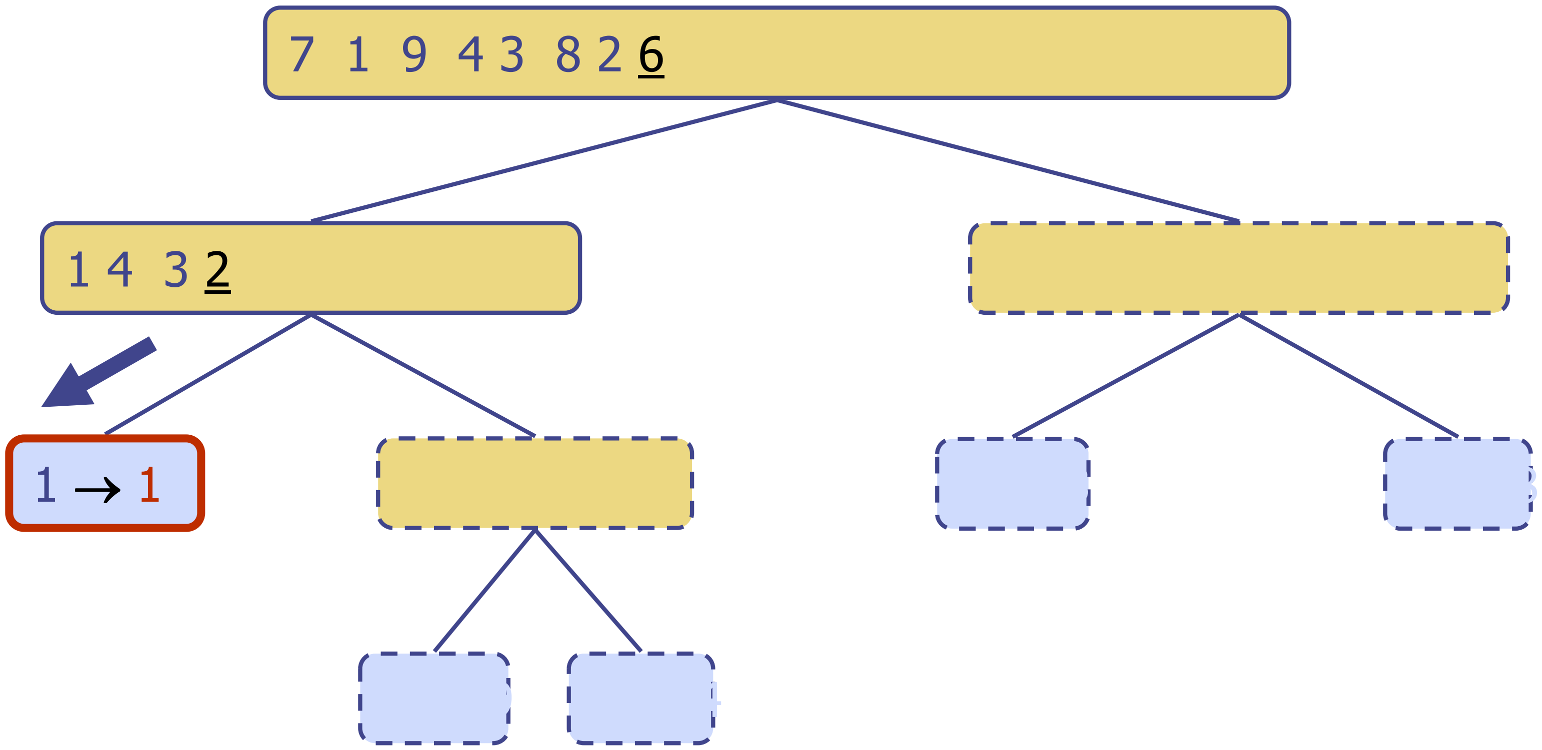
IUT-ECE

- Partition, recursive call, pivot selection



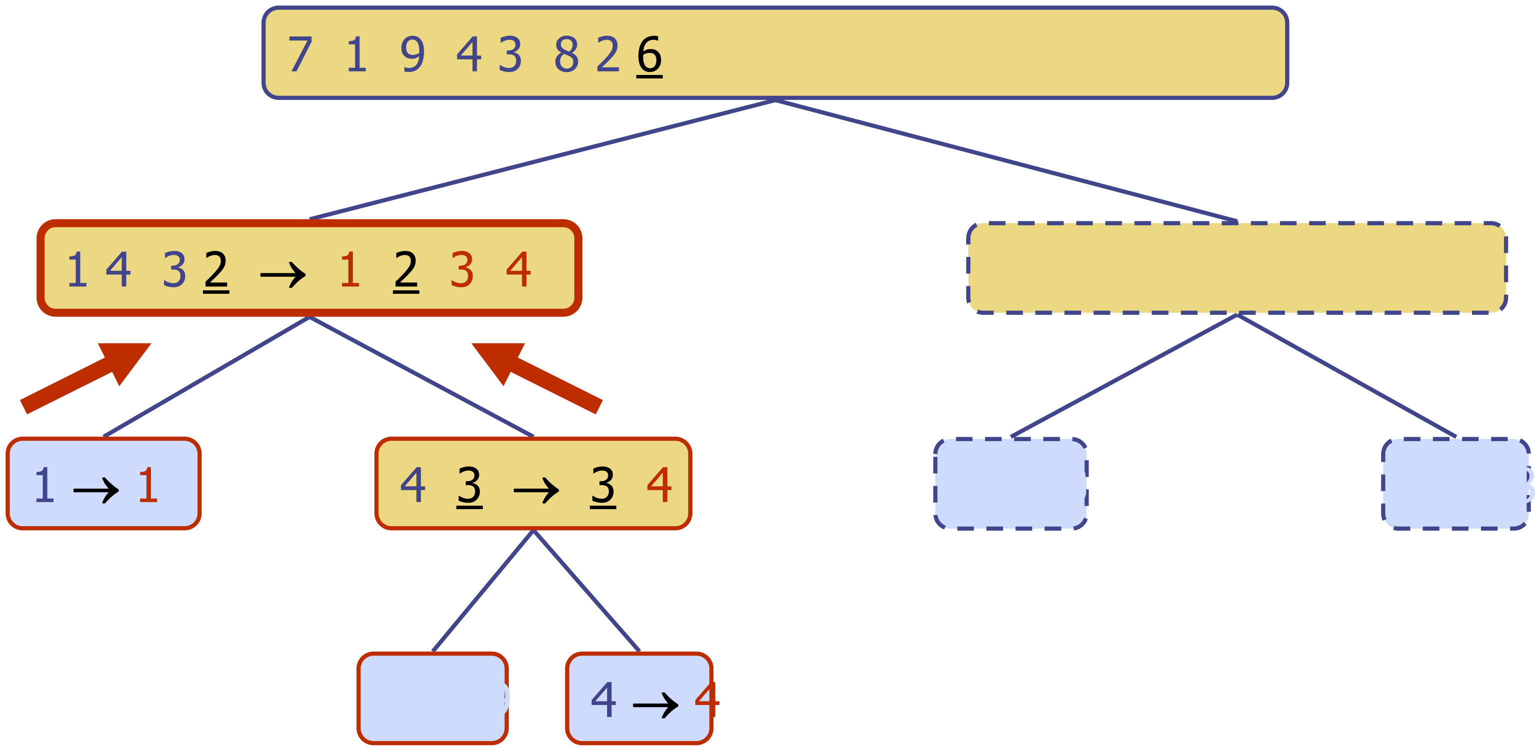
Execution Example (cont.)

- Partition, recursive call, base case



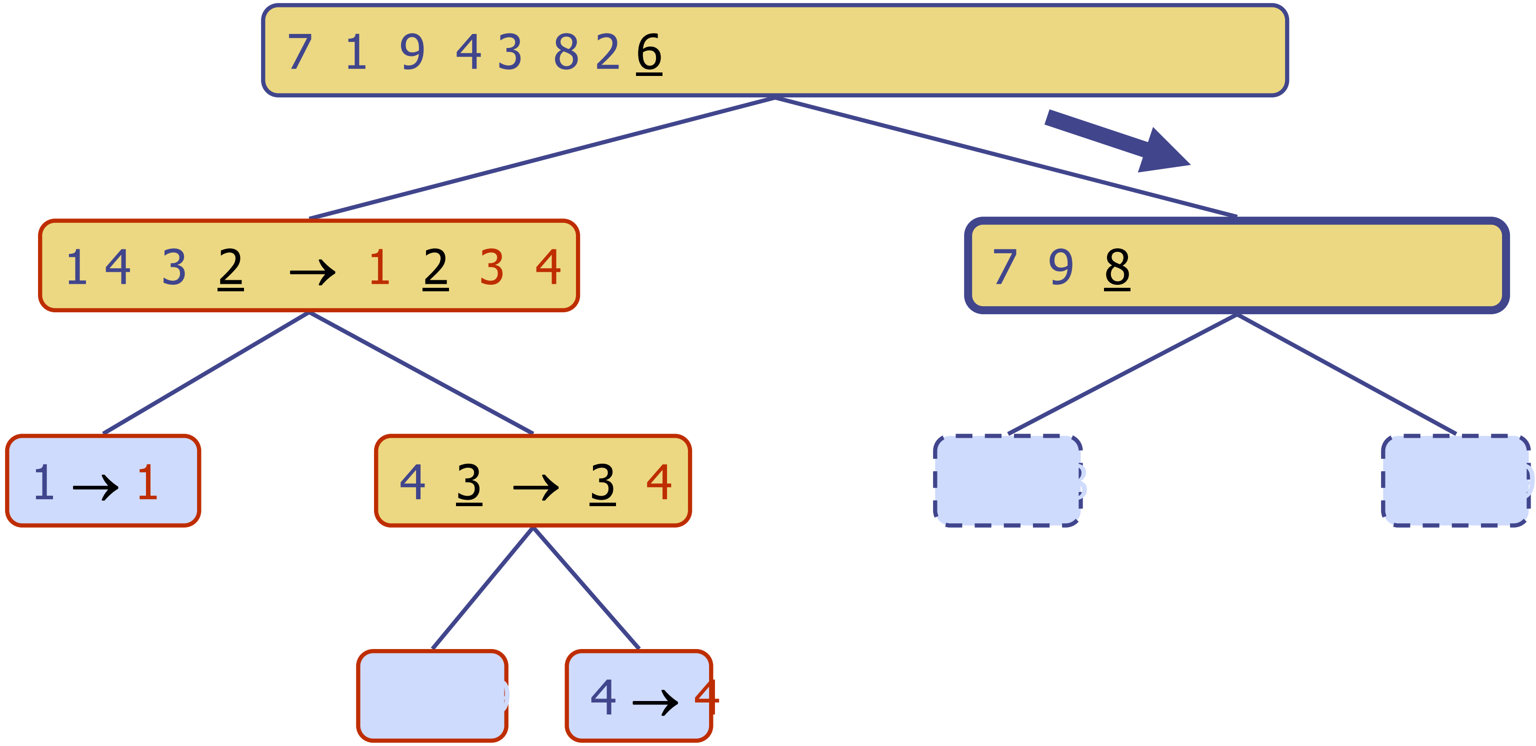
Execution Example (cont.)

- Recursive call, …, base case, join



Execution Example (cont.)

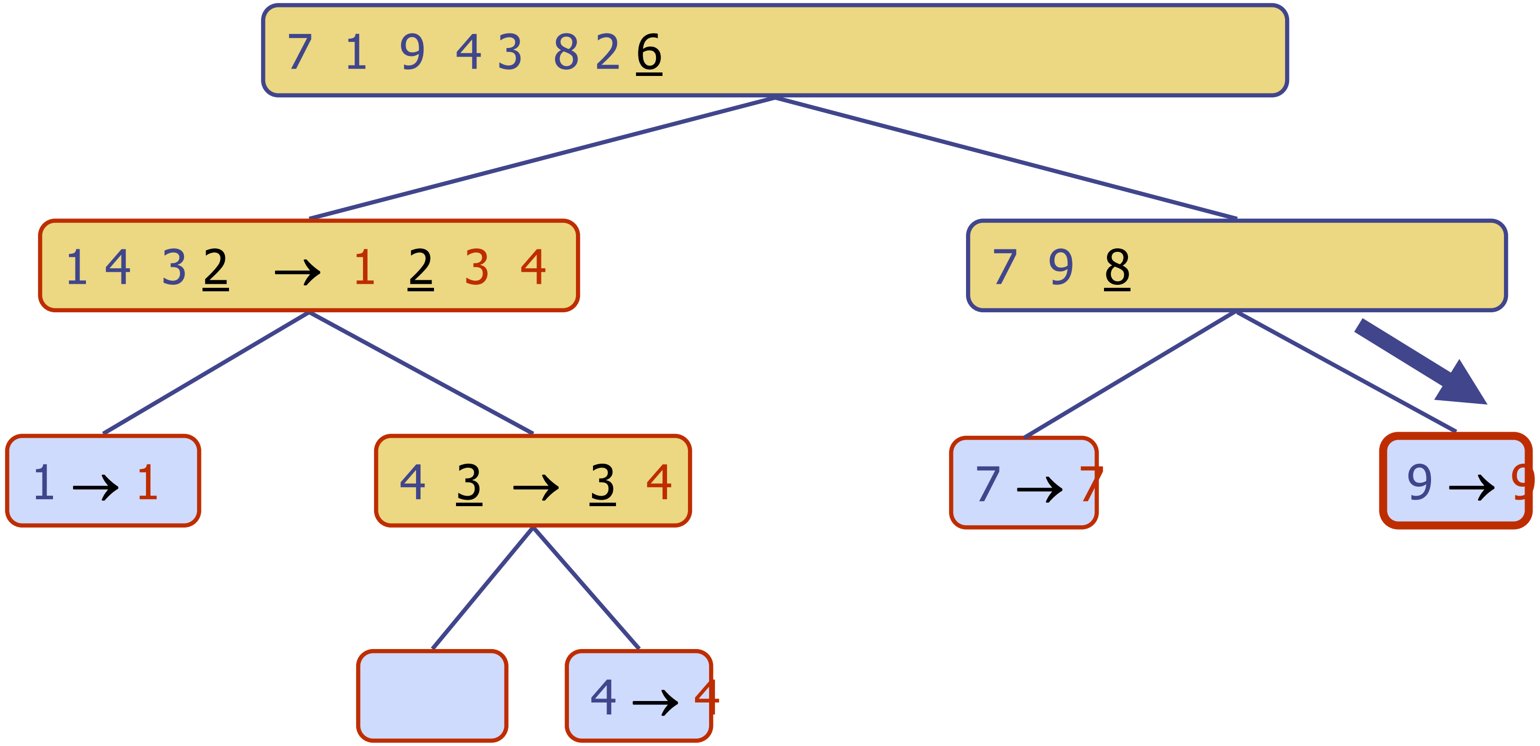
- Recursive call, pivot selection



Execution Example (cont.)

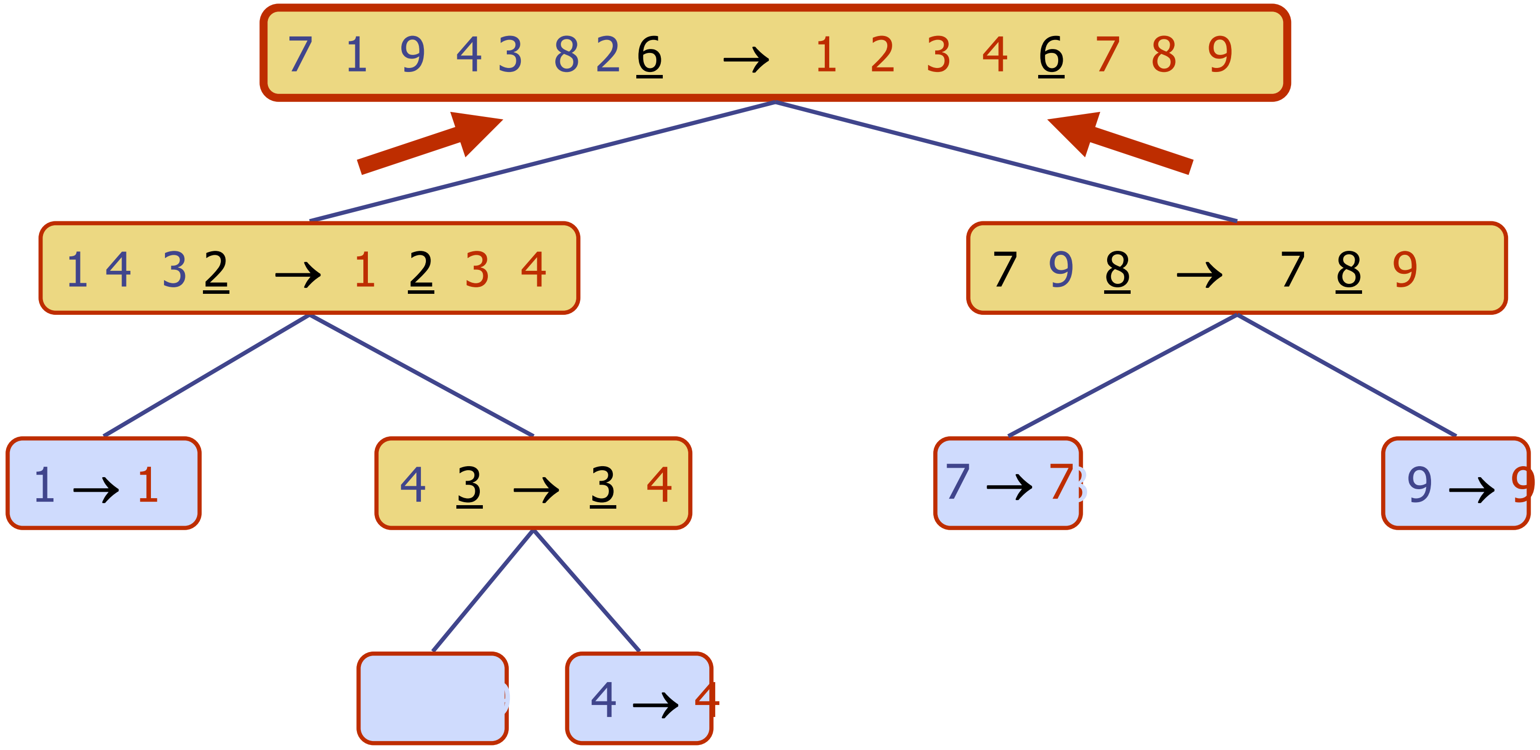
IUT-ECE

- Partition, …, recursive call, base case



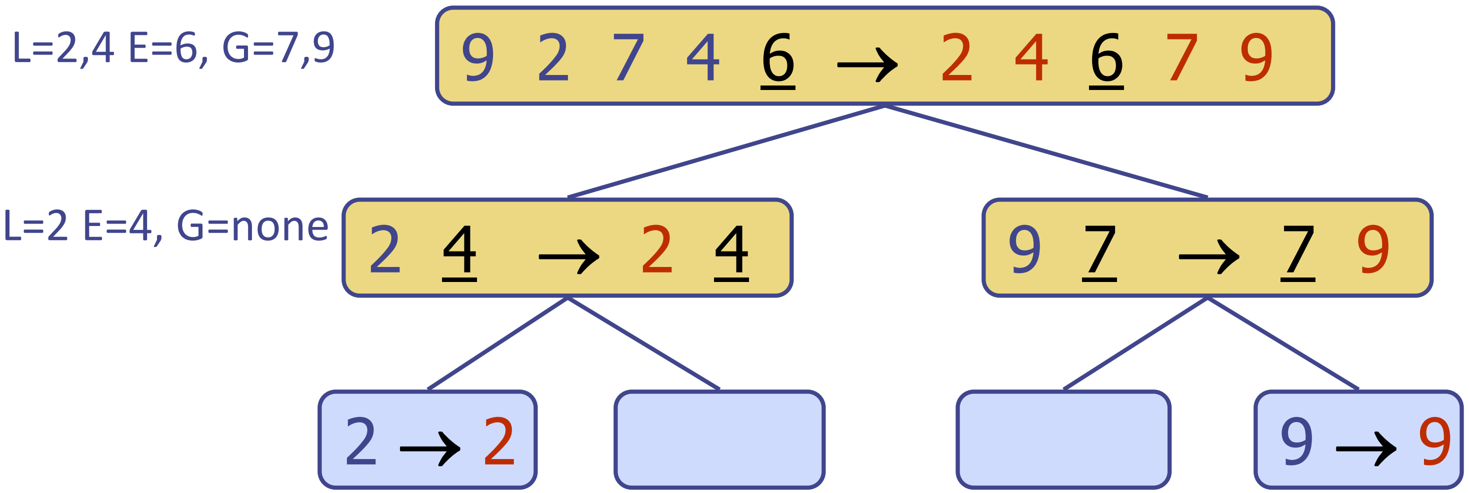
Execution Example (cont.)

- Join, join



Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



Algorithm QuickSort(S):

Input: A sequence S implemented as an array or linked list

Output: The sequence S in sorted order

```
if S.size() ≤ 1 then
   return            {S is already sorted in this case}
p ← S.back().element()          {the pivot}
Let L, E, and G be empty list-based sequences
while !S.empty() do {scan S backwards, dividing it into L, E, and G}
   if S.back().element() < p then
      L.insertBack(S.eraseBack())
   else if S.back().element() = p then
      E.insertBack(S.eraseBack())
   else {the last element in S is greater than p}
      G.insertBack(S.eraseBack())
QuickSort(L)          {Recur on the elements less than p}
QuickSort(G)          {Recur on the elements greater than p}
while !L.empty() do {copy back to S the sorted elements less than p}
   S.insertBack(L.eraseFront())
while !E.empty() do {copy back to S the elements equal to p}
   S.insertBack(E.eraseFront())
while !G.empty() do {copy back to S the sorted elements greater than p}
   S.insertBack(G.eraseFront())
return           {S is now in sorted order}
```

Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

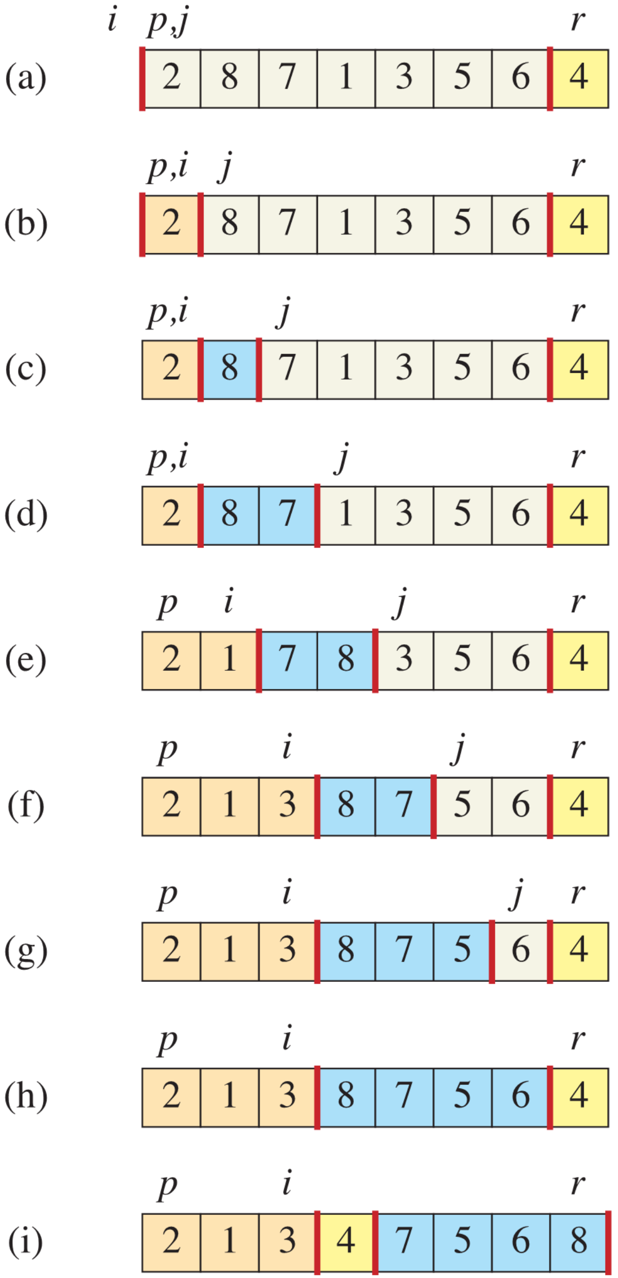
```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.erase(p)
    while ¬S.empty()
        y ← S.eraseFront()
        if y < x
            L.insertBack(y)
        else if y = x
            E.insertBack(y)
        else { y > x }
            G.insertBack(y)
    return L, E, G
```

CLRS version

The QUICKSORT procedure implements quicksort. To sort an entire n-element array $A[1:n]$, the initial call is QUICKSORT$(A, 1, n)$.

```
QUICKSORT(A, p, r)
if p < r
    // Partition the subarray around the pivot, which ends up in A[q].
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)   // recursively sort the low side
    QUICKSORT(A, q + 1, r)   // recursively sort the high side
```

```
PARTITION(A, p, r)
x = A[r]                           // the pivot
i = p - 1                          // highest index into the low side
for j = p to r - 1                 // process each element other than the pivot
    if A[j] ≤ x                    // does this element belong on the low side?
        i = i + 1                      // index of a new slot in the low side
        exchange A[i] with A[j]        // put this element there
exchange A[i + 1] with A[r]        // pivot goes just to the right of the low side
return i + 1                       // new index of the pivot
```




```
PARTITION(A, p, r)
x = A[r]                               // the pivot
i = p − 1                              // highest index into the low side
for j = p to r − 1                     // process each element other than the pivot
    if A[j] ≤ x                        // does this element belong on the low side?
        i = i + 1                      // index of a new slot in the low side
        exchange A[i] with A[j]        // put this element there
exchange A[i + 1] with A[r]            // pivot goes just to the right of the low side
return i + 1                           // new index of the pivot
```

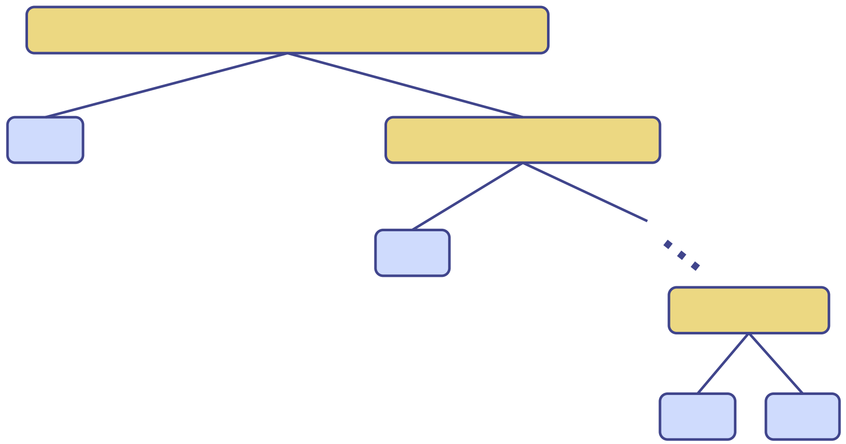
Worst-case (partitioning) Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of $\boldsymbol{L}$ and $\boldsymbol{G}$ has size $\boldsymbol{n} - 1$ and the other has size 0
- The running time is proportional to the sum

$$\boldsymbol{n} + (\boldsymbol{n} - 1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $\boldsymbol{O}(\boldsymbol{n}^2)$

depth	time
0	$\boldsymbol{n}$
1	$\boldsymbol{n} - 1$
…	…
$\boldsymbol{n} - 1$	1



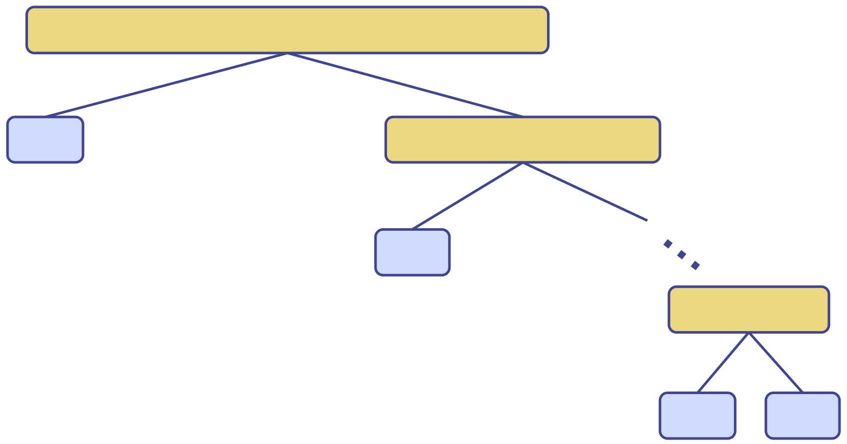
Worst-case (partitioning) Running Time

IUT-ECE

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(n) \\
&= T(n-1) + \Theta(n) .
\end{aligned}
$$

depth	time
0	n
1	$n-1$
…	…
$n-1$	1

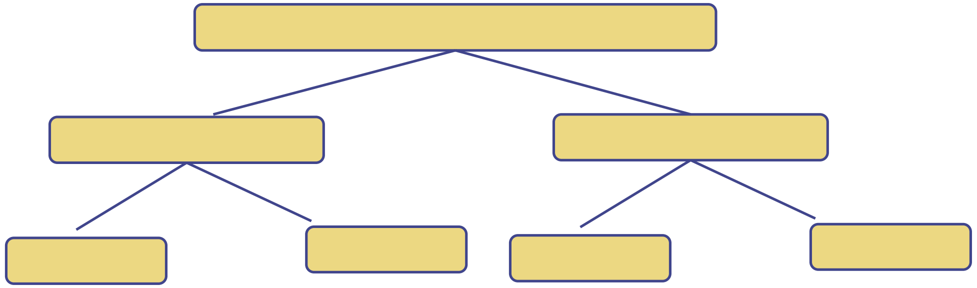


Best-case (partitioning) Running Time

- PARTITION produces two subproblems, each of size no more than n/2.

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

$$T(n) = \Theta(n \lg n)$$



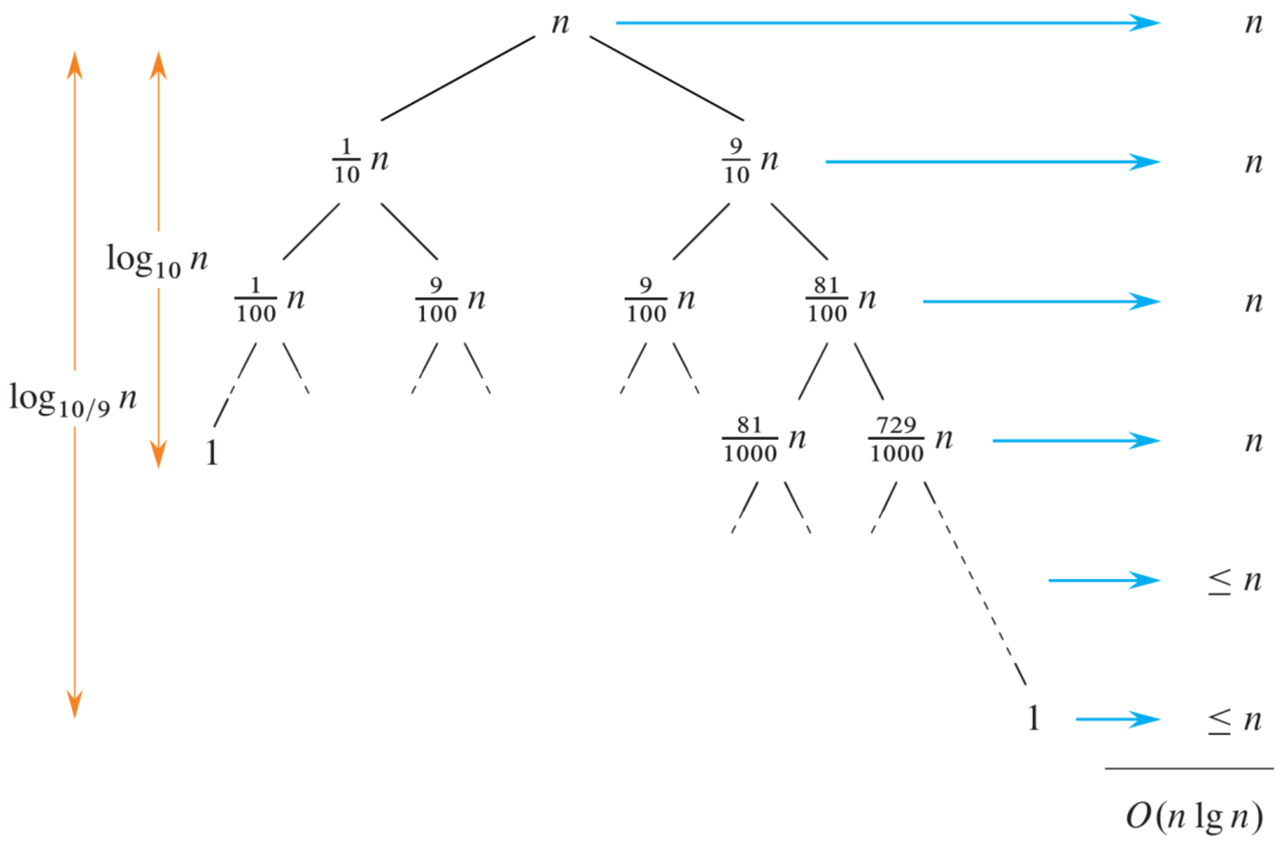
Balanced partitioning Running Time

- Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split,

$$T(n) = T(9n/10) + T(n/10) + \Theta(n) ,$$

Balanced partitioning Running Time

- Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split,

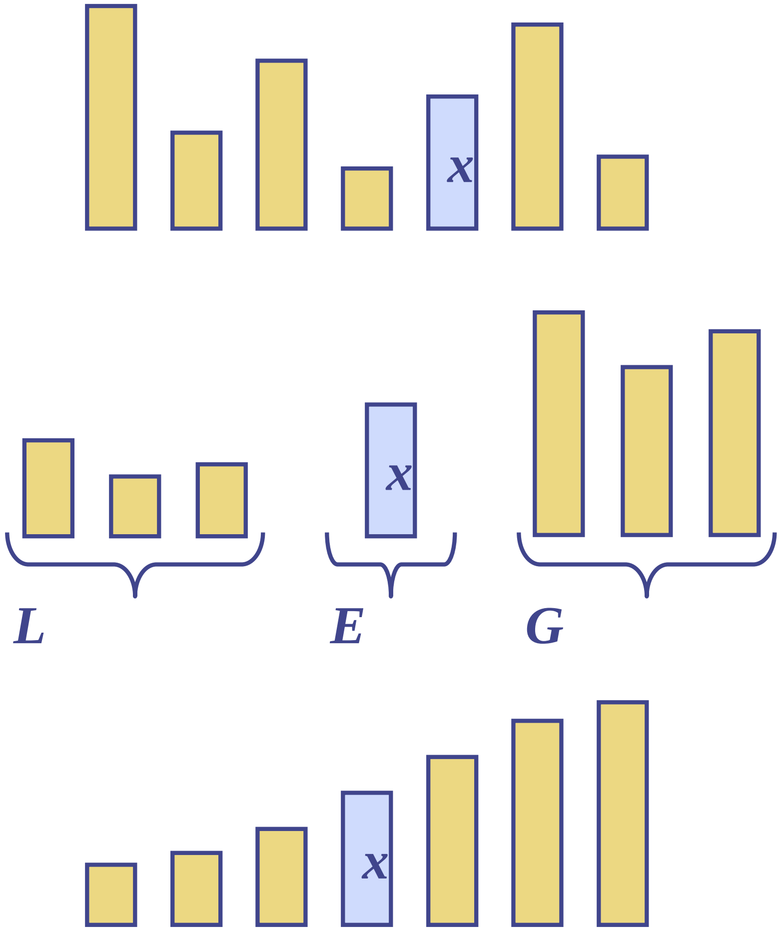


Pivot

- بهترین حالت چه زمانی رخ داد؟
- Pivot چگونه انتخاب شود؟

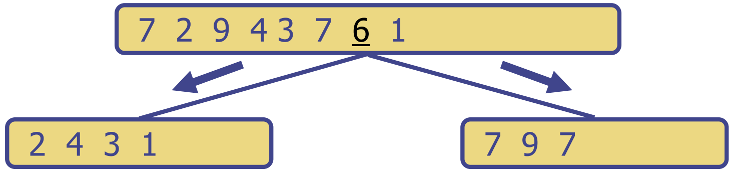
Quick-Sort (randomized version)

- Quick-sort is a (randomized) sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a **random** element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G

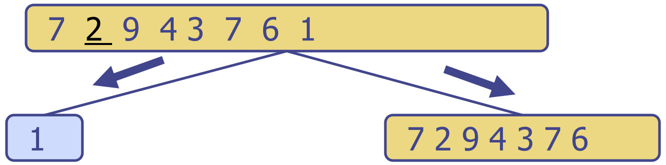


Expected Running Time (1)

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$ (“unbiased to some degree”)
  - **Bad call:** one of L and G has size greater than $3s/4$ (“biased to some degree”)

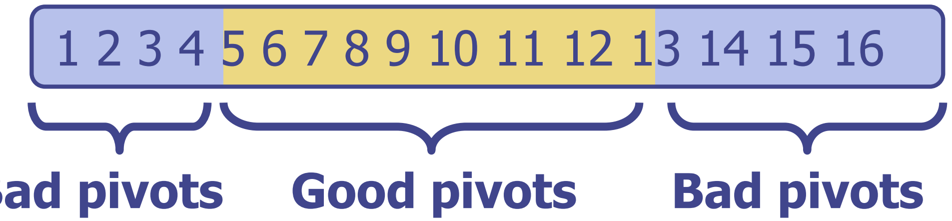


Good call



Bad call

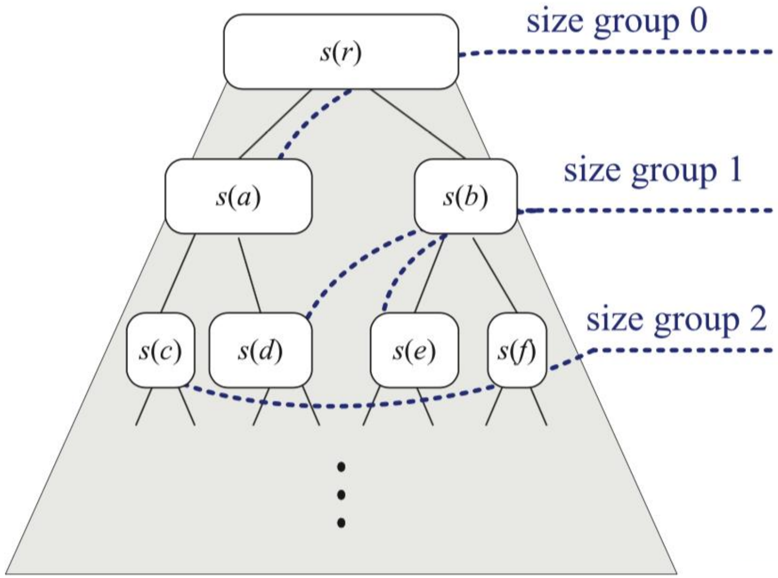
- A call is good with probability $1/2$
  - $1/2$ of the possible pivots cause good calls:



Expected Running Time (2)

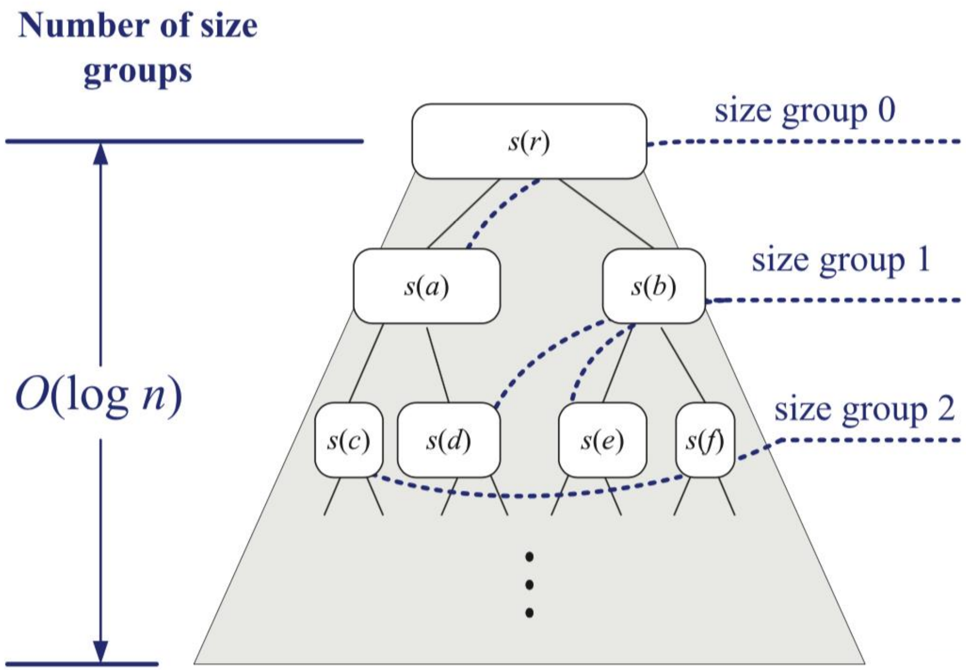
IUT-ECE

- Consider a binary tree T used in the Quick-sort.
- Definition
  - A node v (a collection of elements) in T is said to be in size group i if $\left(\frac{3}{4}\right)^{\{i+1\}} n \leq$ the size of v's subproblem $\leq \left(\frac{3}{4}\right)^{\{i\}} n$
  - Thus, every node is in some size group (e.g., the root node is in size group 0)



Expected Running Time (3)

- Q1. How many size groups?
  - (Ans) i, such that $\left(\frac{3}{4}\right)^{\{i\}} n = 1$, i.e., ***i = 2log$_{4/3}$n***



Expected Running Time (3)

IUT-ECE

- Q1. How many size groups?
  - (Ans) i, such that $\left(\frac{3}{4}\right)^{\{i\}} n$ = 1, i.e., ***i = 2log$_{4/3}$n***

- Q2. What is the expected time spent working on all the subproblems for nodes in size group *i* (which we denote by T)?
  - If the answer is O(n), then we are done, because the number of size groups * expected running time for each size group = n * log n.

  - T = sum of the expected times for each node, say *v*, in size group *i* (linearity of expectation). Thus, our question is "what is the expected time for a node in size group i"?
  - v's subproblem may be either of good call or bad call.
  - (Two facts) Since a probability of good call is ½,
    - (i) The expected number of consecutive calls before a good call is 2 (i.e., constant)
    - (ii) As soon as we have a good call for node v (in size group i), its children will be in size groups higher than i. (because at least ¾ reduction of the original size happens)

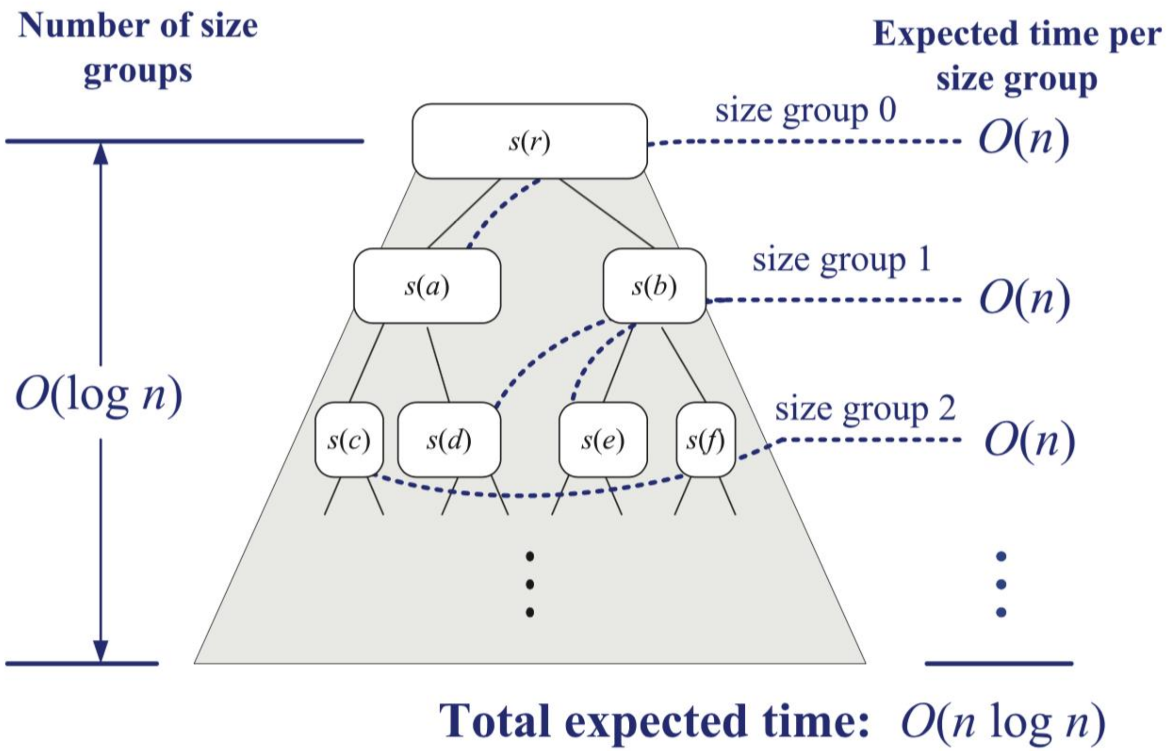
Expected Running Time (4)

- Q1. How many size groups?
  - (Ans) i, such that $\left(\frac{3}{4}\right)^{\{i\}} n$ = 1, i.e., ***i = 2log$_{4/3}$n***
- Q2. What is the expected time spent working on all the subproblems for nodes in size group *i* (which we denote by T)?
  - Thus, for any elements x in the input list, the expected number of nodes in size group i containing x in their subproblems is 2. (on average, constant number times of being at a bad call group and then move to the size group higher than i)
  - → Expected total size of all the subproblems in size group i is 2n
    - → Non-recursive work we perform for any subproblem is proportional to its size
    - → Expected time per each size group is O(n)
- Thus,
  - log n size groups & n computations per each size group
  - → O(n log n)

Expected Running Time (4)

IUT-ECE

- Consider a binary tree T used in the Quick-sort.
- Definition
  - A node v (a collection of elements) in T is said to be in size group i if $\left(\frac{3}{4}\right)^{\{i+1\}} n \leq$ the size of v's subproblem $\leq \left(\frac{3}{4}\right)^{\{i\}} n$
  - Thus, every node is in some size group (e.g., the root node is in size group 0)



Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)