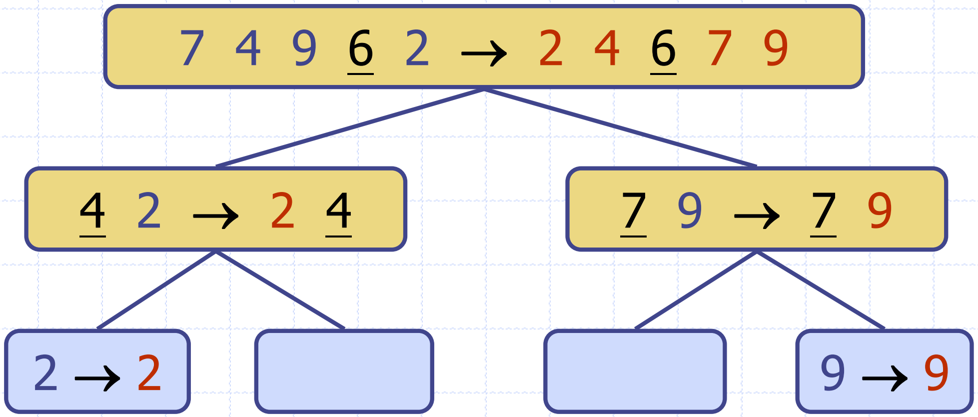
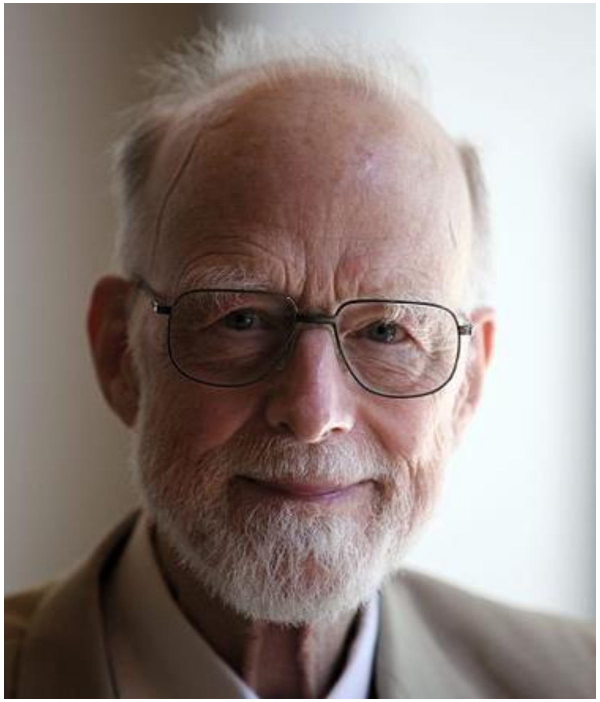
Quick-Sort



Quick-Sort Inventor

- Invented by Hoare in 1962

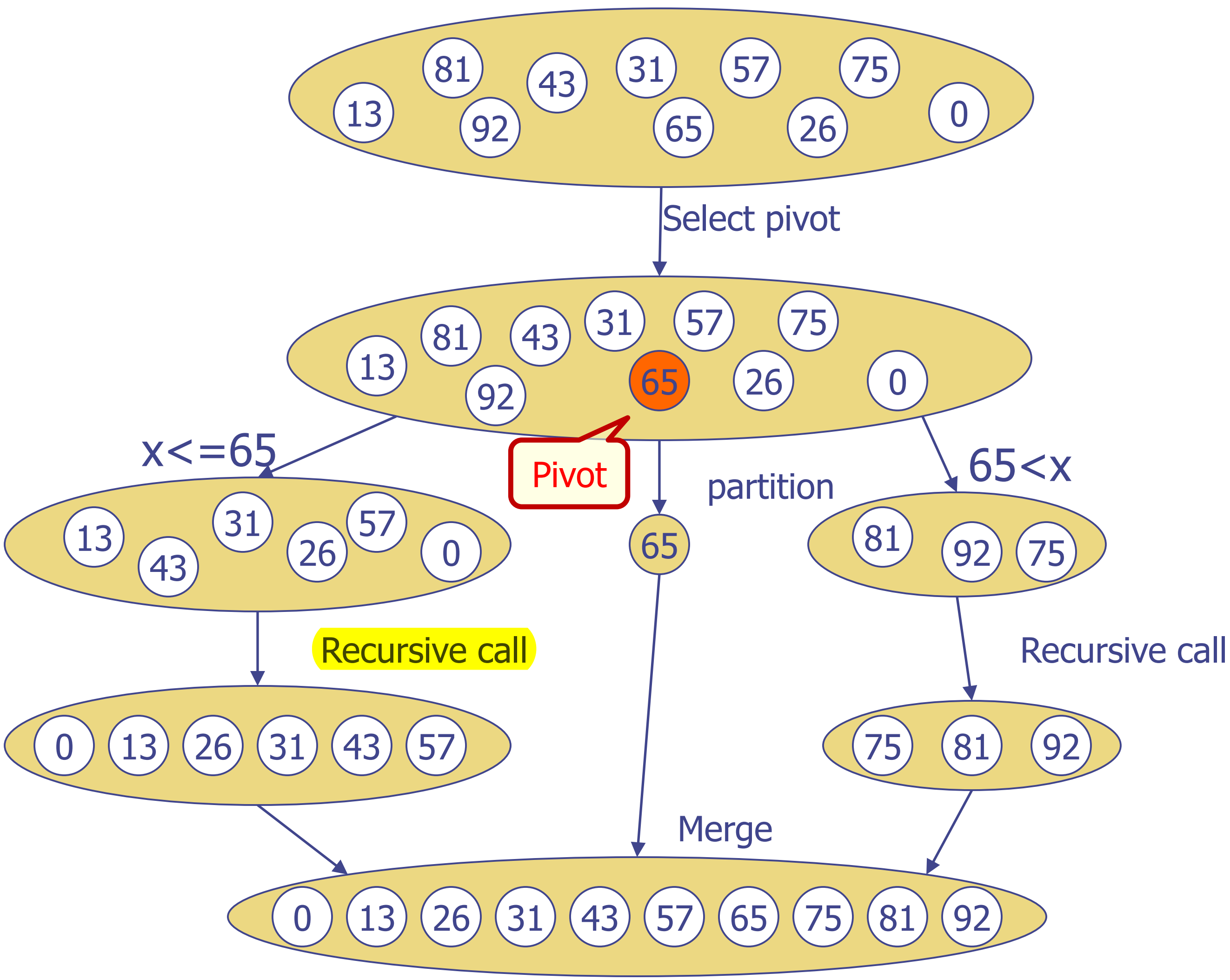


查爾斯·安東尼·理察·霍爾爵士(**Charles Antony Richard Hoare**，縮寫為 **C. A. R. Hoare**，**1934**年**1**月**11**日－ **)**，生於斯里蘭卡可倫坡，英國計算機科學家，圖靈獎得主。他設計了可快速進行排序程序的快速排序(**quick sort)**演算法，提出可驗證程式正確性的霍爾邏輯(**Hoare logic)** 、以及提出可訂定並時程序(**concurrent process)**的交互作用(如哲學家用餐問題(dining philosophers problem) 的 交 談 循 序 程 續 **(CSP, Communicating Sequential Processes)**架構 。**(**圖及說明摘自維基百科**)**

About Quick-Sort

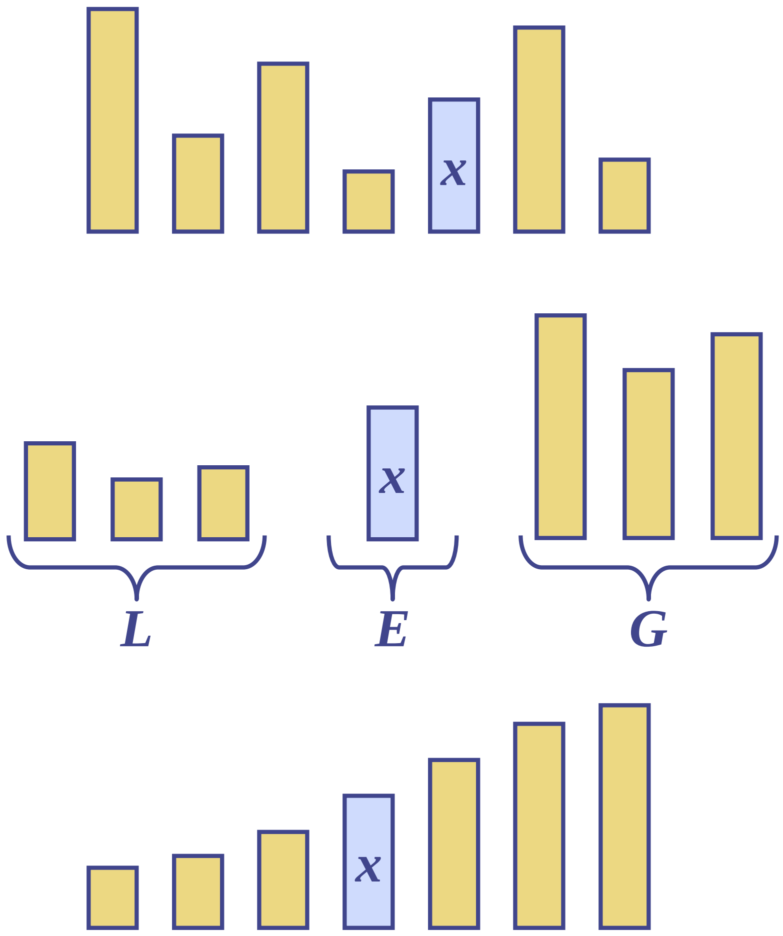
- Fastest known sorting algorithm in practice
  Under some assumptions
  - Caveat: not stable
  - In-place, good for internal sorting
- Complexity
  - Average-case complexity $O(n \log n)$
  - Worse-case complexity $O(n^2)$
    - Rarely happens if remedied correctly

Quick-Sort Example



Quick-Sort

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



Key to the success of quicksort:
- Select a good pivot
- In-place partition

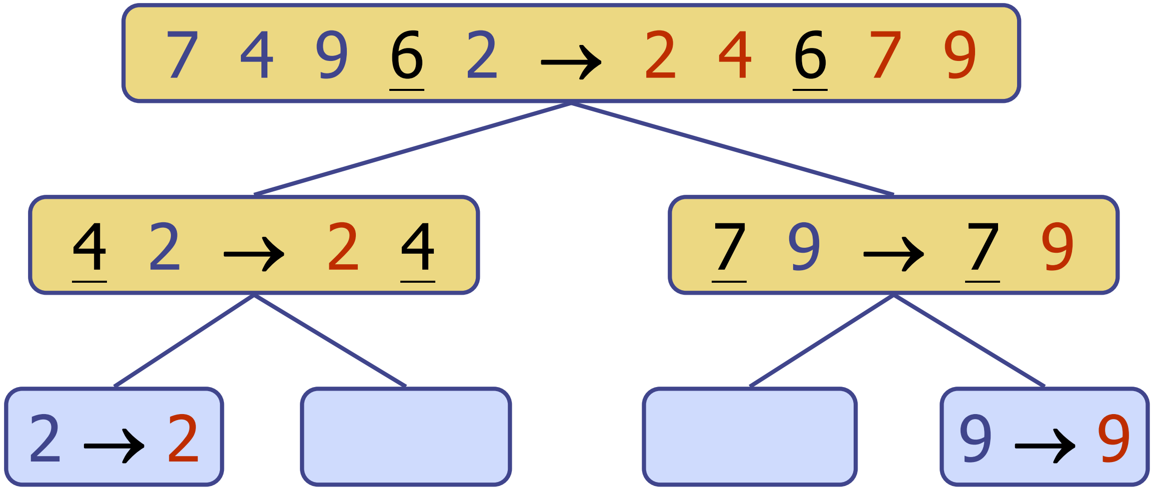
Partition

- 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
```

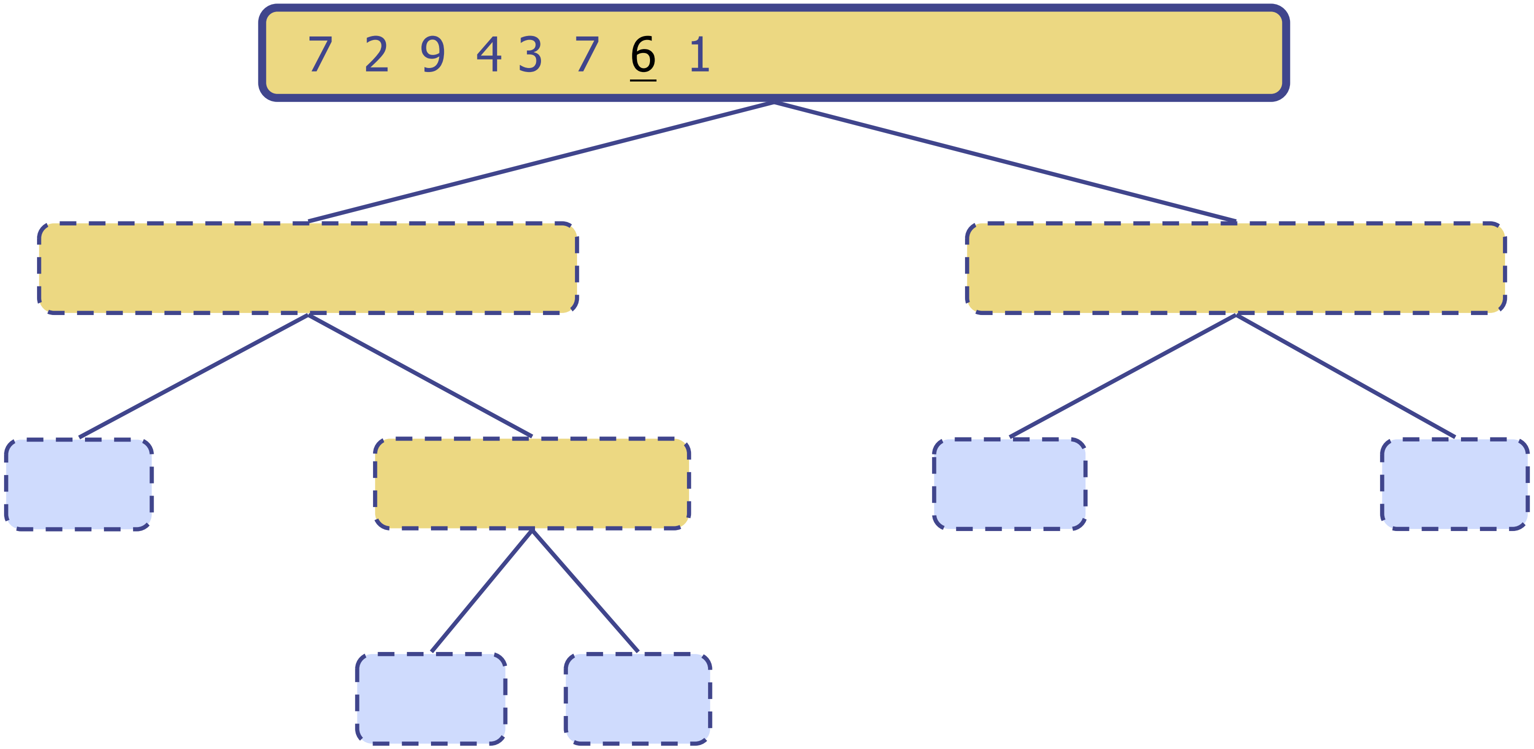
Quick-Sort Tree

- 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



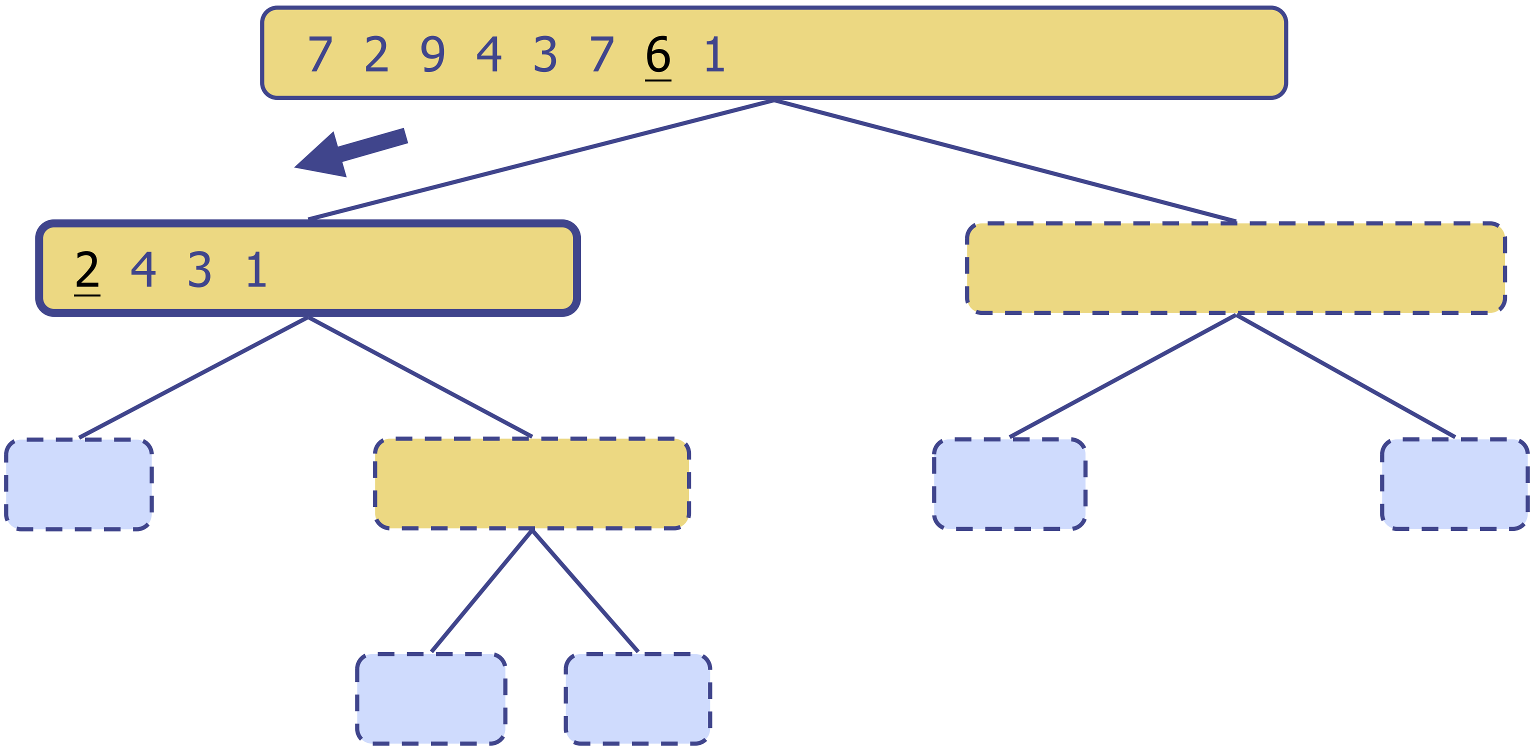
Execution Example

- Pivot selection



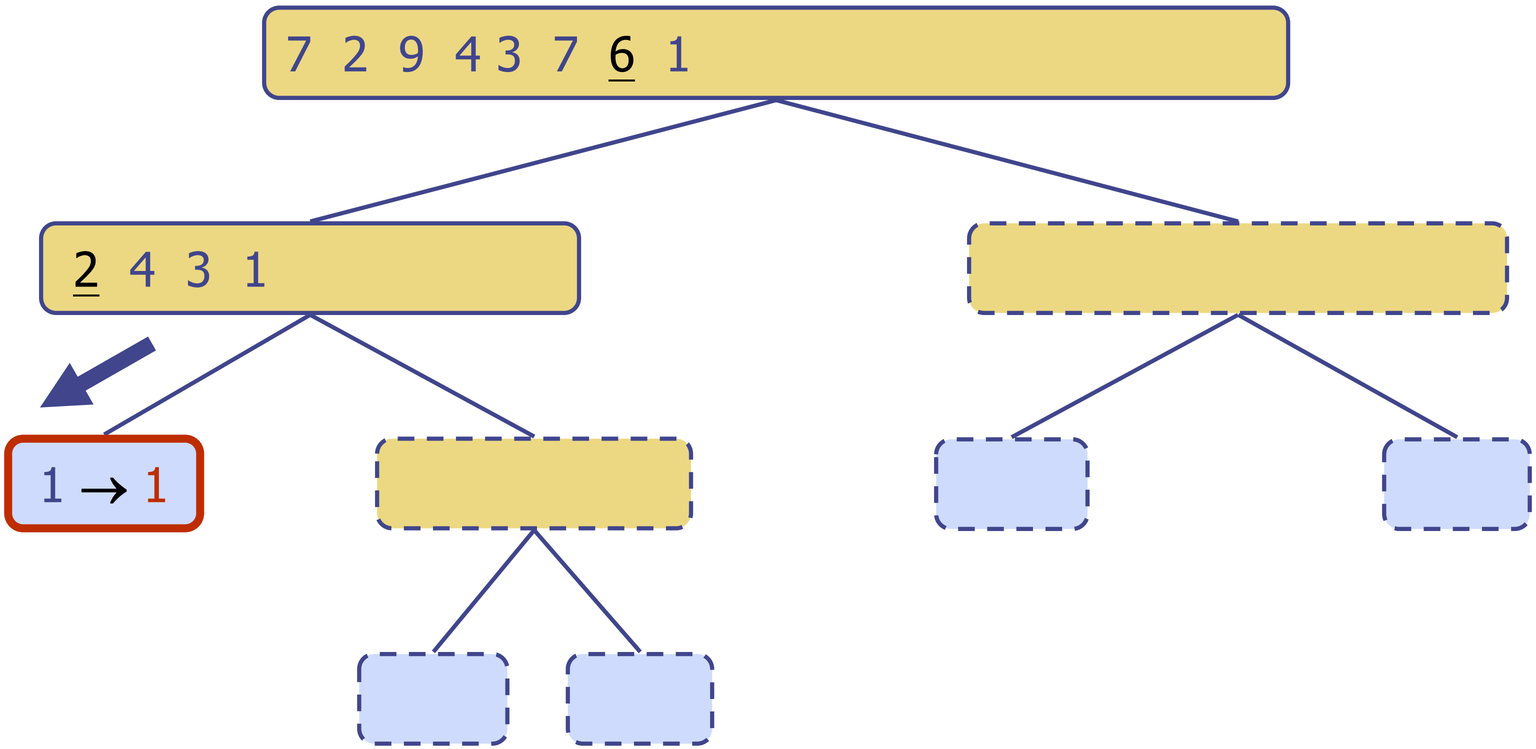
Execution Example (cont.)

- Partition, recursive call, pivot selection



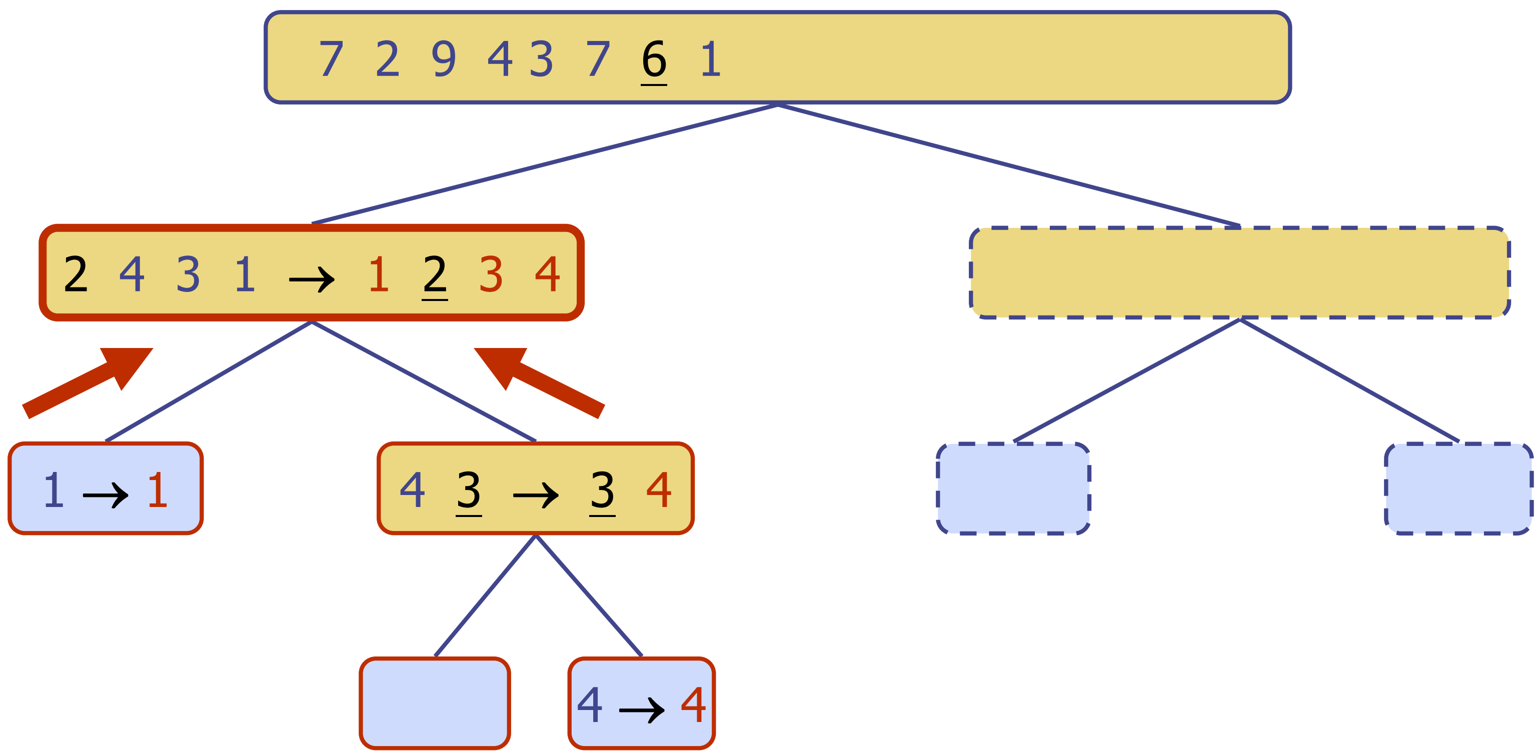
Execution Example (cont.)

- Partition, recursive call, base case



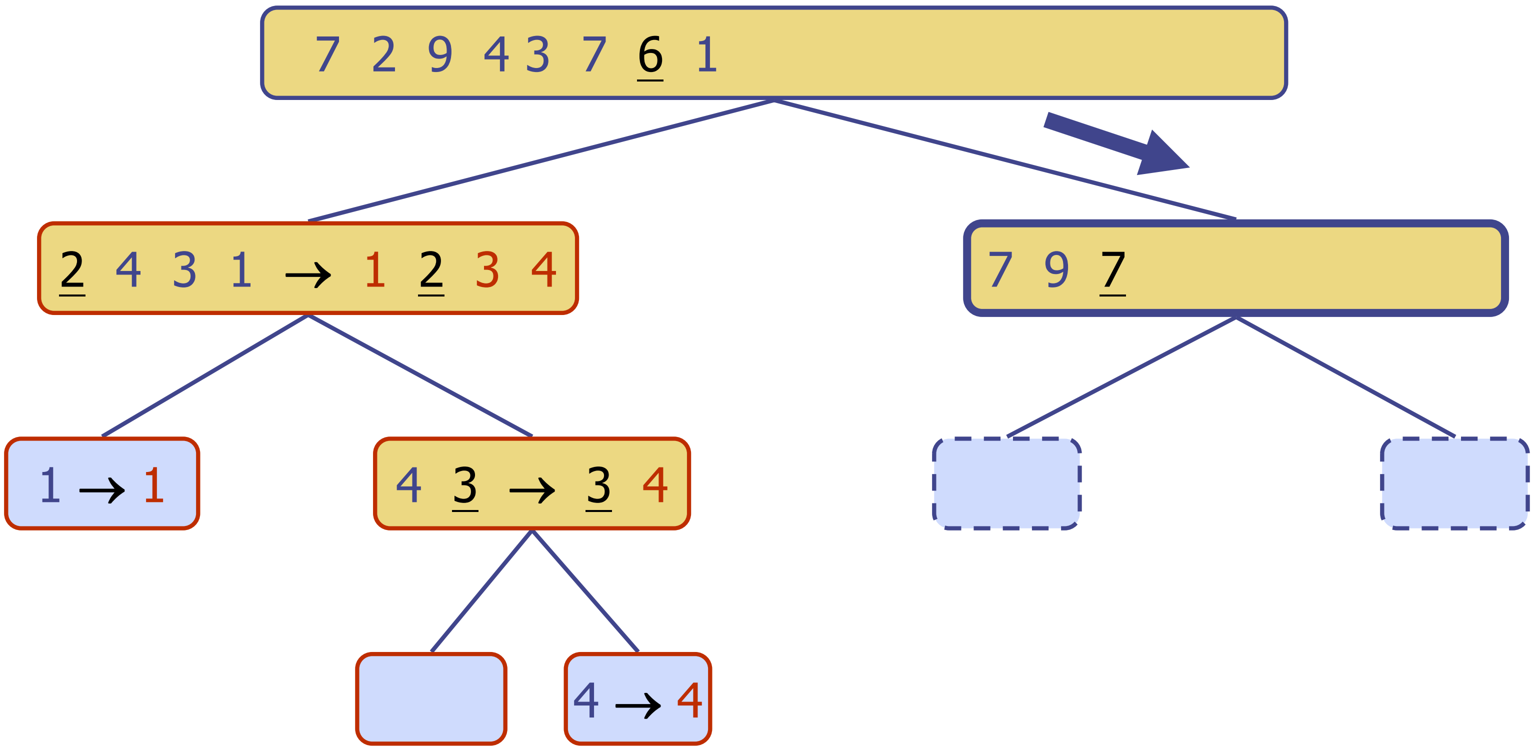
Execution Example (cont.)

- Recursive call, …, base case, join



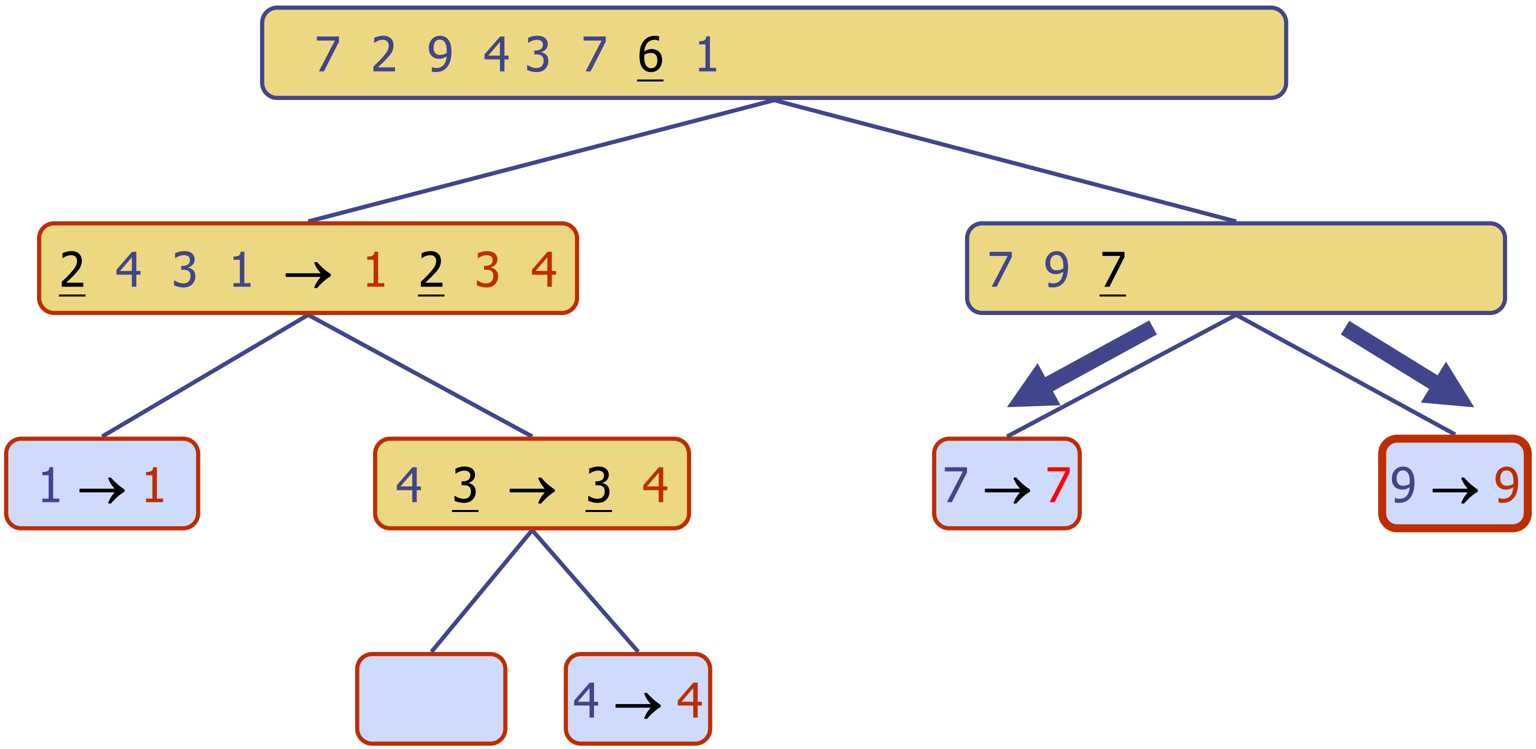
Execution Example (cont.)

- Recursive call, pivot selection



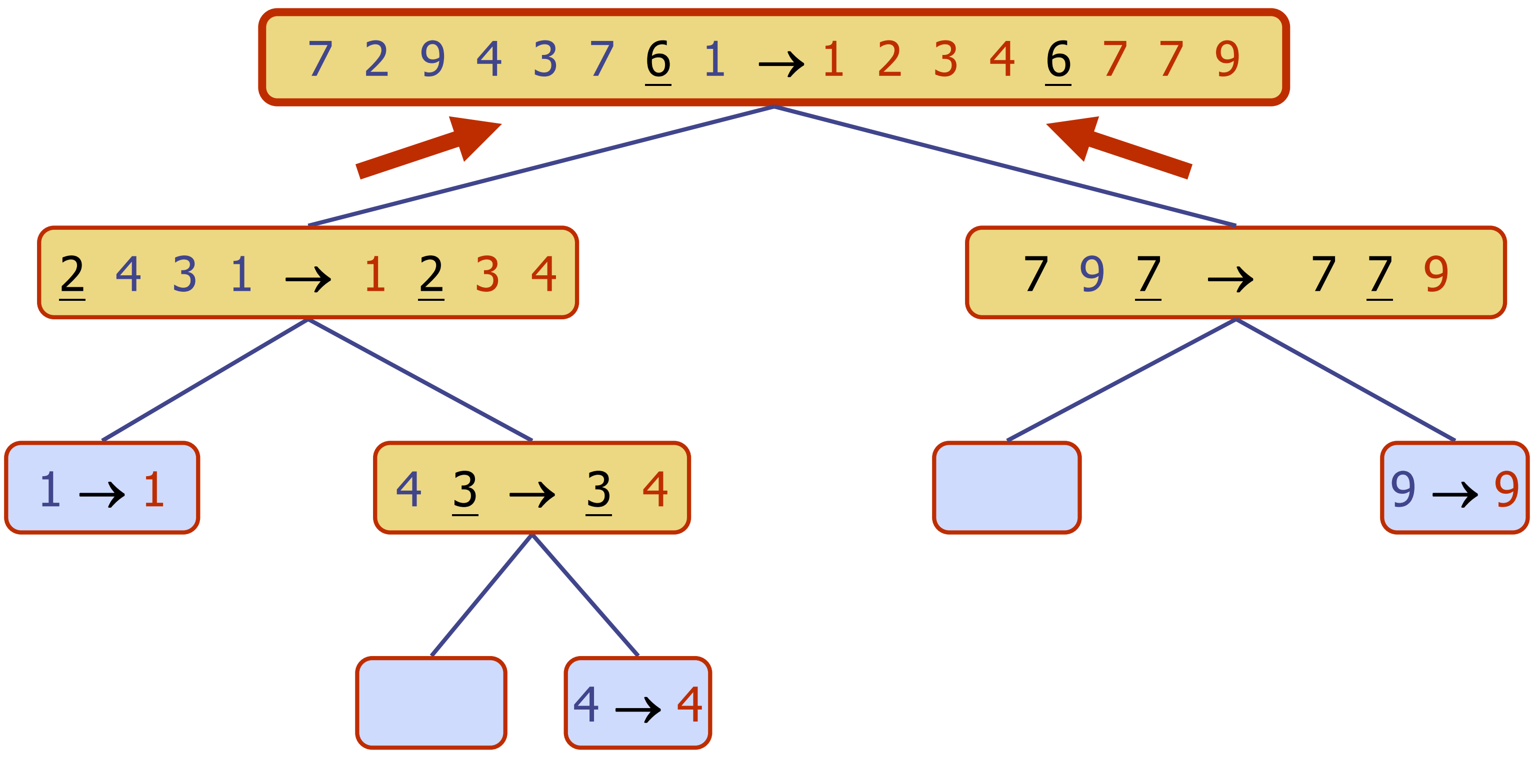
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join



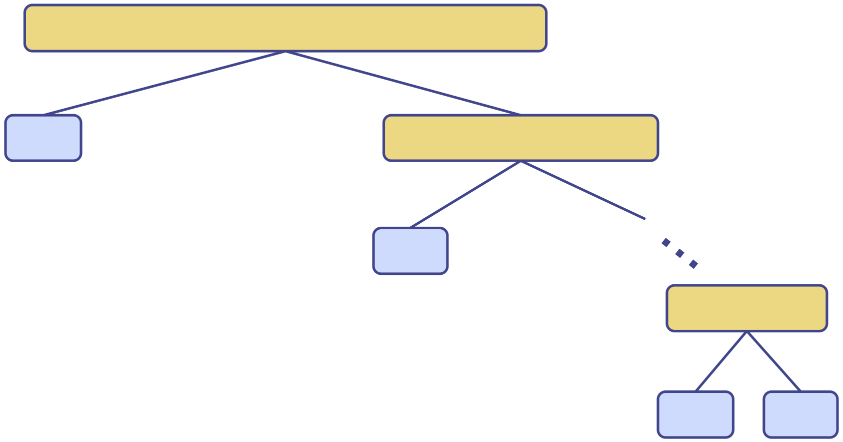
Worst-case Running Time

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

$$n + (n - 1) + \ldots + 2 + 1$$

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

depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



Expected Running Time

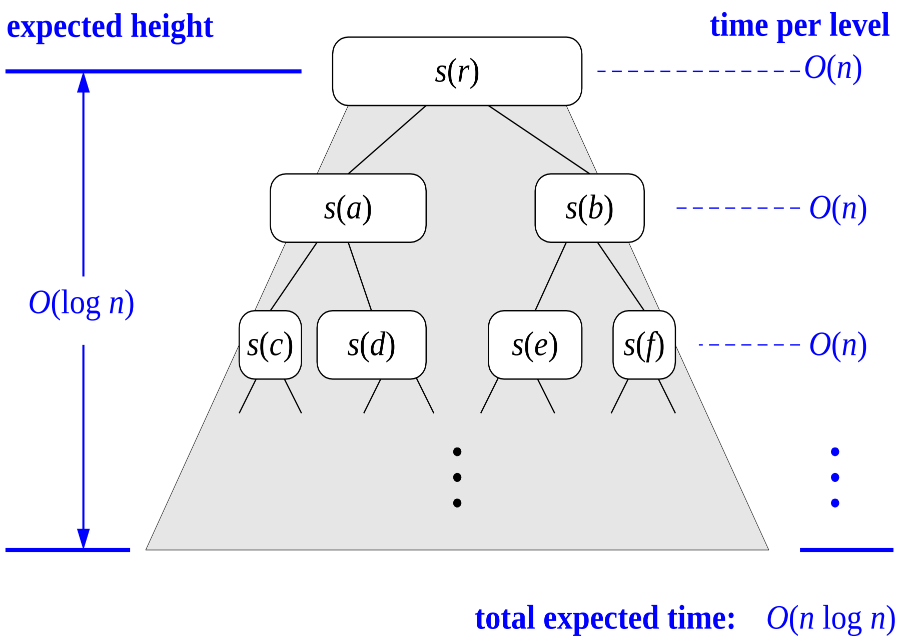
- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$

Good call **Bad call**

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

Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



In-Place Quick-Sort

```
template <typename E, typename C>                      // quick-sort S
void quickSort(std::vector<E>& S, const C& less) {
  if (S.size() <= 1) return;                            // already sorted
  quickSortStep(S, 0, S.size()-1, less);                // call sort utility
}

template <typename E, typename C>
void quickSortStep(std::vector<E>& S, int a, int b, const C& less) {
  if (a >= b) return;                                   // 0 or 1 left? done
  E pivot = S[b];                                       // select last as pivot
  int l = a;                                            // left edge
  int r = b - 1;                                        // right edge
  while (l <= r) {
    while (l <= r && !less(pivot, S[l])) l++;           // scan right till larger
    while (r >= l && !less(S[r], pivot)) r--;           // scan left till smaller
    if (l < r)                                          // both elements found
      std::swap(S[l], S[r]);
  }                                                     // until indices cross
  std::swap(S[l], S[b]);                                // store pivot at l
  quickSortStep(S, a, l-1, less);                       // recur on both sides
  quickSortStep(S, l+1, b, less);
}
```

Code Fragment 11.7: A coding of in-place quick-sort, assuming distinct elements.

Partitioning Algorithm Illustrated

	l								r	pivot
	8	1	4	9	0	3	5	2	7	6
	l							r		pivot
Move	8	1	4	9	0	3	5	2	7	6
	l							r		pivot
swap	2	1	4	9	0	3	5	8	7	6
				l			r			pivot
move	2	1	4	9	0	3	5	8	7	6
				l			r			pivot
swap	2	1	4	5	0	3	9	8	7	6
						r	l			pivot
move	2	1	4	5	0	3	9	8	7	6
Swap S[i] with pivot	2	1	4	5	0	3	6	8	7	9
						r	l			
							pivot			

l and r have crossed!

Animation

How to Pick the Pivot (1/2)

- Strategy 1: Pick the first or the last element
  - Works only if the input S is random
  - $O(n^2)$ if input S is sorted or almost sorted
- Strategy 2: Pick a random element
  - Usually works well
  - Extra computation for random number generation
- Strategy 3: Perform random permutation of input S first
  - Usually works well

random_shuffle() in STL

How to Pick the Pivot (2/2)

- Strategy 4: Median of three

Quiz!

  - Ideally, the pivot should be the median of input S, which divides the input into two sequences of almost the same length
  - However, computing median takes O(n)
  - So we find the approximate median via
    - Pivot = median of the left-most, right-most, and the center element of the array S

Example of Median of Three

- Let input S = {6, 1, 2, 9, 0, 3, 5, 2, 7, 8}
  - left=0 and S[left] = 6
  - right=9 and S[right] = 8
  - center = (left+right)/2 = 4 and S[center] = 0
  - Pivot = median of {6, 8, 0} = 6

Dealing with Small Arrays

- For small arrays (say, $N \le 20$),
  - Insertion sort is faster than quicksort
- Quicksort is recursive
  - So it can spend a lot of time sorting small arrays
- Hybrid algorithm:
  - Switch to using insertion sort when problem size is small (say for $N < 20$)

Quiz!

Summary of Sorting Algorithms

Algorithm	Time	Notes [Comprehensive list!]
selection-sort	$O(n^2)$	▪ in-place, unstable ▪ slow, for small data sets (< 1K)
insertion-sort bubble-sort	$O(n^2)$	▪ in-place, stable ▪ slow, for small data sets (< 1K)
quick-sort	$O(n \log n)$ expected	▪ in-place, unstable ▪ fastest (?), for large data sets (1K ~ 1M)
heap-sort	$O(n \log n)$	▪ in-place, unstable ▪ fast, for large data sets (1K ~ 1M)
merge-sort	$O(n \log n)$	▪ not in-place, stable ▪ fast, for huge data sets (> 1M)

More About Selection Sort

- How come it's unstable?
  - Example: 2_a 2_b 1

Quiz!

- How to make it stable?
  - Quickest fix: Use "insert" instead of "swap"
    - Expensive for arrays
    - Cheap for linked lists

http://www.geeksforgeeks.org/stable-selection-sort

More About Heap Sort

- How come it's unstable?
  - Example: 2_a 2_b 1
- How to make it stable?
  - Please post it to FB

How to Make it Stable?

Quiz!

- How come make a general-purpose unstable sort algorithm stable?
  - Use the original key and a new key of the array's index to perform multiple-key comparison for sorting
  - Example of selection sort
    - [2 2 1]

Make the key unique!

http://www.quora.com/What-should-be-done-to-make-unstable-sorting-algorithms-stable