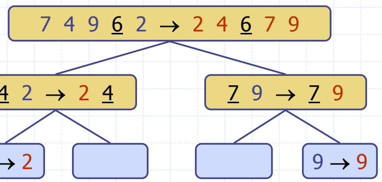
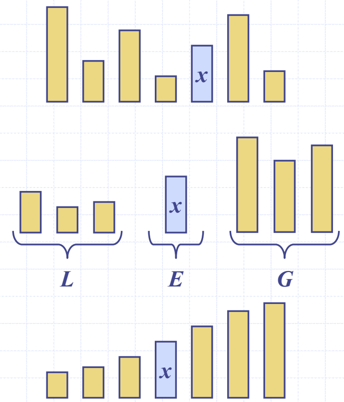
Quick-Sort



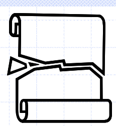
Quick-Sort (§ 10.2)

- 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



Partition

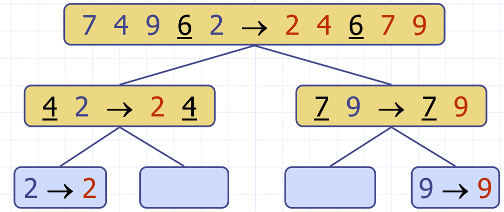
- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the 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.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.insertLast(y)
        else if y = x
            E.insertLast(y)
        else { y > x }
            G.insertLast(y)
    return L, E, G
```

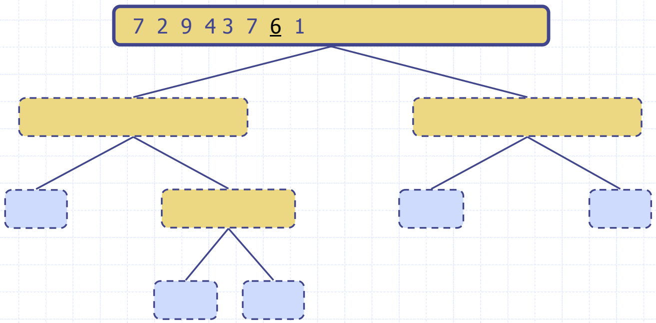

Quick-Sort Tree

- 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



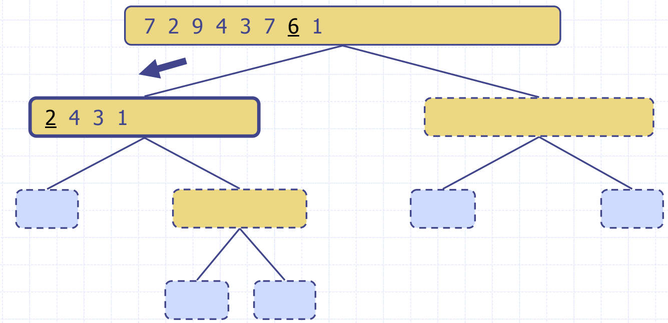
Execution Example

- Pivot selection



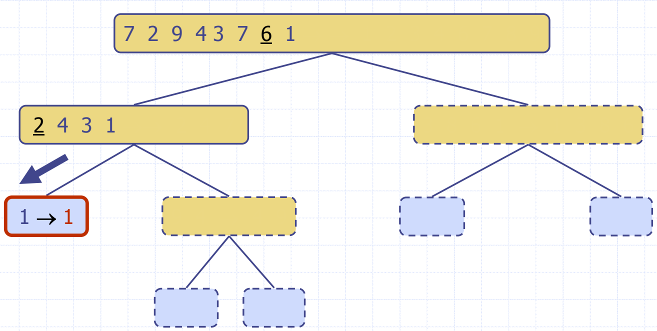
Execution Example (cont.)

- Partition, recursive call, pivot selection



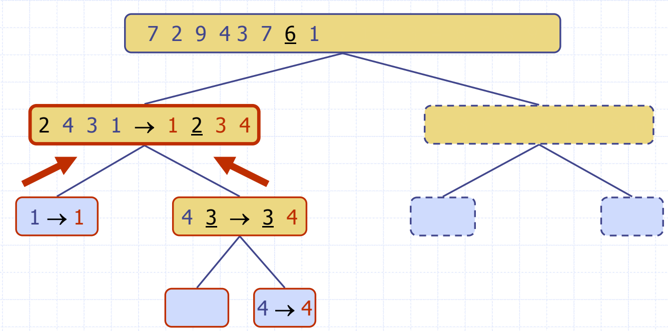
Execution Example (cont.)

- Partition, recursive call, base case



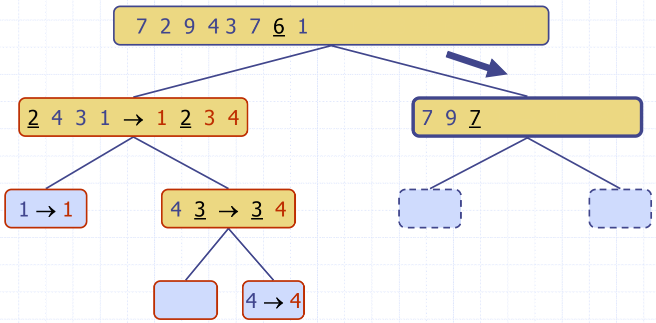
Execution Example (cont.)

- Recursive call, …, base case, join



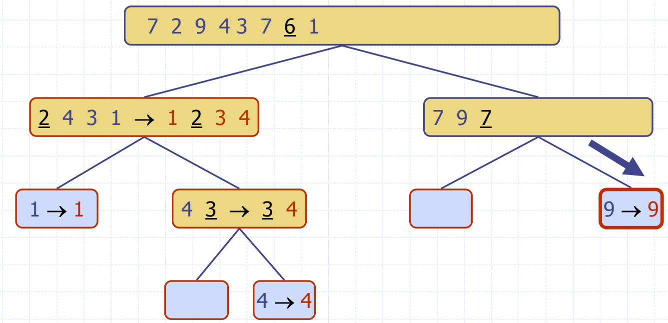
Execution Example (cont.)

- Recursive call, pivot selection



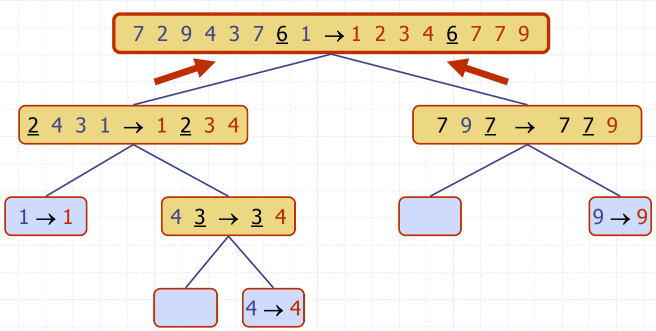
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



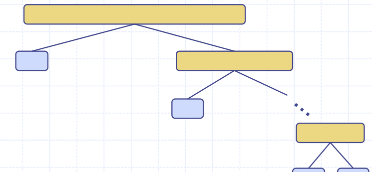
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique 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

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

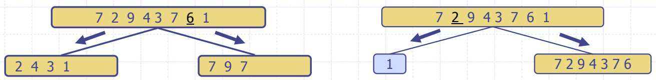
- Thus, the worst-case running time of quick-sort is $O(n^2)$

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



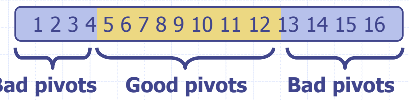
Expected Running Time

- 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 $3s/4$



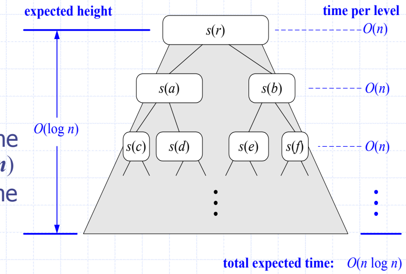
Good call **Bad call**

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

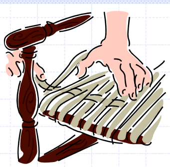


Expected Running Time, Part 2

- 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 $O(n \log n)$



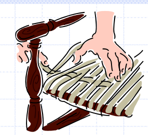
In-Place Quick-Sort



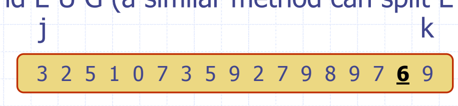
- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than h
  - the elements equal to the pivot have rank between h and k
  - the elements greater than the pivot have rank greater than k
- The recursive calls consider
  - elements with rank less than h
  - elements with rank greater than k

```
Algorithm inPlaceQuickSort(S, l, r)
  Input sequence S, ranks l and r
  Output sequence S with the
     elements of rank between l and r
     rearranged in increasing order
  if l ≥ r
     return
  i ← a random integer between l and r
  x ← S.elemAtRank(i)
  (h, k) ← inPlacePartition(x)
  inPlaceQuickSort(S, l, h − 1)
  inPlaceQuickSort(S, k + 1, r)
```


In-Place Partitioning

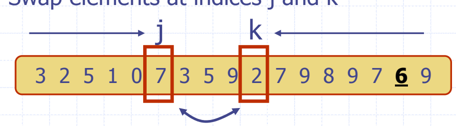


- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).



(pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element ≥ x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k



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)

Java Implementation

```
public static void quickSort (Object[] S, Comparator c) {
   if (S.length < 2) return; // the array is already sorted in this case
   quickSortStep(S, c, 0, S.length   1); // recursive sort method
 }
 private static void quickSortStep (Object[] S, Comparator c,
                                 int leftBound, int rightBound ) {
   if (leftBound >= rightBound) return; // the indices have crossed
   Object temp;  // temp object used for swapping
   Object pivot = S[rightBound];
   int leftIndex = leftBound;     // will scan rightward
   int rightIndex = rightBound   1; // will scan leftward
   while (leftIndex <= rightIndex) { // scan right until larger than the pivot
     while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0) )
       leftIndex++;
     // scan leftward to find an element smaller than the pivot
     while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0))
       rightIndex-  ;
     if (leftIndex < rightIndex) { // both elements were found
       temp = S[rightIndex];
      S[rightIndex] = S[leftIndex]; // swap these elements
      S[leftIndex] = temp;
     }
   } // the loop continues until the indices cross
   temp = S[rightBound]; // swap pivot with the element at leftIndex
   S[rightBound] = S[leftIndex];
   S[leftIndex] = temp; // the pivot is now at leftIndex, so recurse
   quickSortStep(S, c, leftBound, leftIndex   1);
   quickSortStep(S, c, leftIndex+1, rightBound);
 }
```

only works for distinct elements