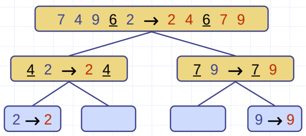
Quick-Sort



9/2/2002 3:15 AM Quick-Sort 1

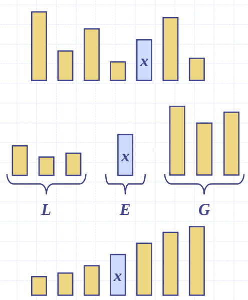
Outline and Reading

- Quick-sort (§4.3)
  - Algorithm
  - Partition step
  - Quick-sort tree
  - Execution example
- Analysis of quick-sort (4.3.1)
- In-place quick-sort (§4.8)
- Summary of sorting algorithms

9/2/2002 3:15 AM Quick-Sort 2

Quick-Sort

- 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



9/2/2002 3:15 AM Quick-Sort 3

Partition

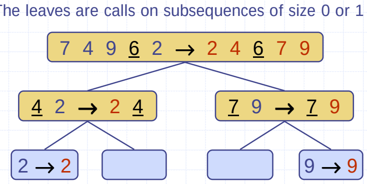
- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the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

9/2/2002 3:15 AM Quick-Sort 4

Quick-Sort Tree

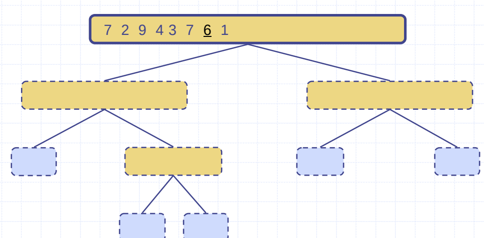
- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences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



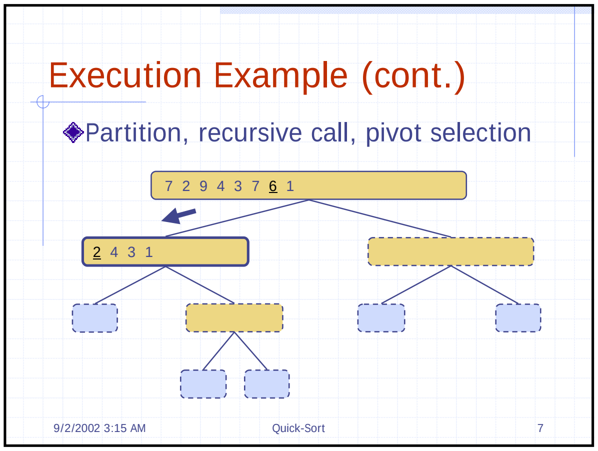
9/2/2002 3:15 AM Quick-Sort 5

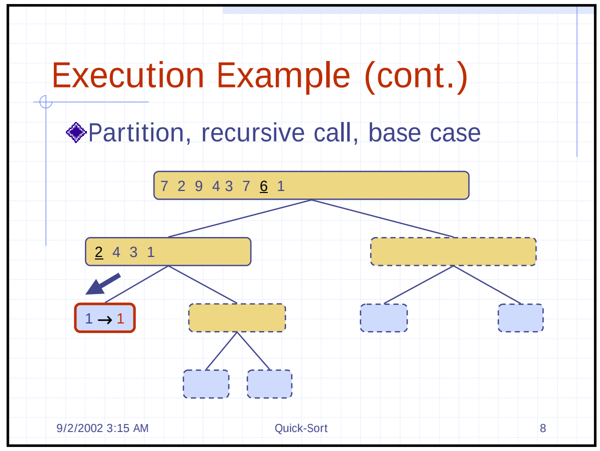
Execution Example

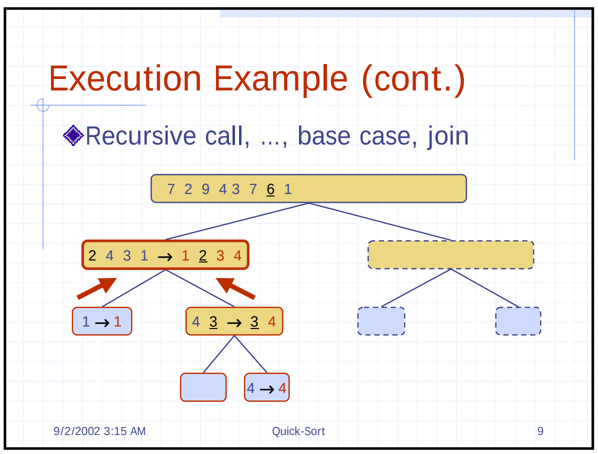
- Pivot selection

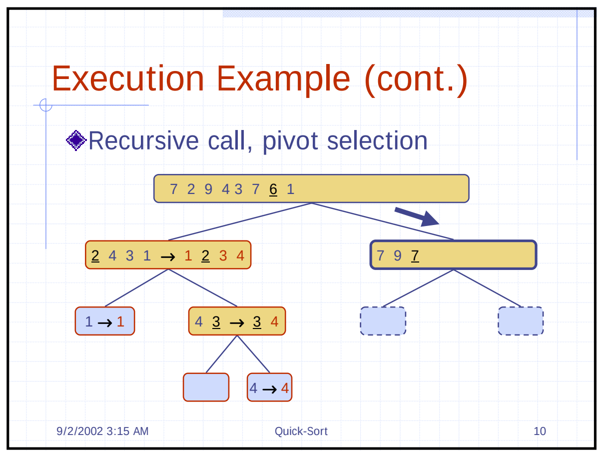


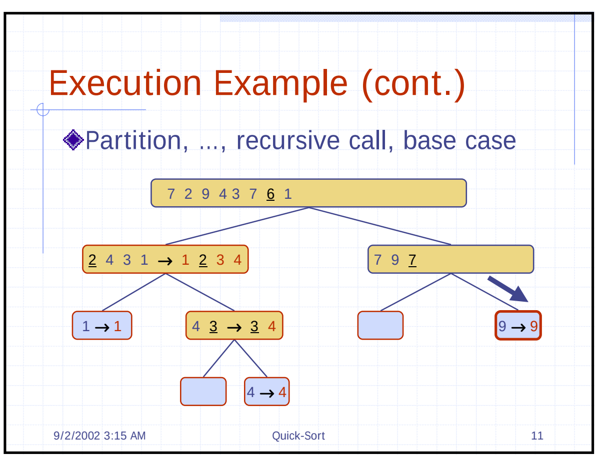
9/2/2002 3:15 AM Quick-Sort 6

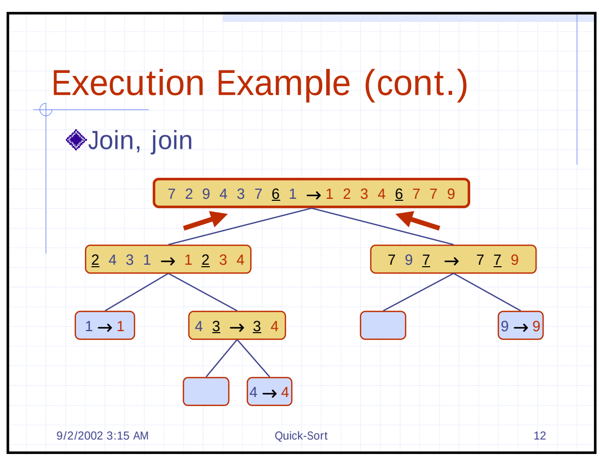

Execution Example (cont.)
Partition, recursive call, pivot selection
7 2 9 4 3 7 6 1
2 4 3 1
9/2/2002 3:15 AM
Quick-Sort
7


Execution Example (cont.)
Partition, recursive call, base case
7 2 9 4 3 7 6 1
2 4 3 1
1 → 1
9/2/2002 3:15 AM
Quick-Sort
8


Execution Example (cont.)
Recursive call, ..., base case, join
7 2 9 4 3 7 6 1
2 4 3 1 → 1 2 3 4
1 → 1
4 3 → 3 4
4 → 4
9/2/2002 3:15 AM
Quick-Sort
9


Execution Example (cont.)
Recursive call, pivot selection
7 2 9 4 3 7 6 1
2 4 3 1 → 1 2 3 4
7 9 7
1 → 1
4 3 → 3 4
4 → 4
9/2/2002 3:15 AM
Quick-Sort
10


Execution Example (cont.)
Partition, ..., recursive call, base case
7 2 9 4 3 7 6 1
2 4 3 1 → 1 2 3 4
7 9 7
1 → 1
4 3 → 3 4
9 → 9
4 → 4
9/2/2002 3:15 AM
Quick-Sort
11

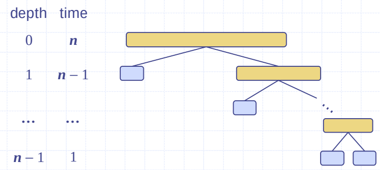

Execution Example (cont.)
Join, join
7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 7 9
2 4 3 1 → 1 2 3 4
7 9 7 → 7 7 9
1 → 1
4 3 → 3 4
9 → 9
4 → 4
9/2/2002 3:15 AM
Quick-Sort
12

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 $O(n^2)$



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 $3s/4$
- A call is good with probability $1/2$
- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- Hence, for a node of depth i, we expect that
  - $i/2$ parent nodes are associated with good calls
  - the size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Thus, we have
  - For a node of depth $2\log_{4/3}n$, the expected size of the input sequence is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The overall amount or work done at the nodes of the same depth of the quick-sort tree 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)
```


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)