Quicksort

Algorithms & Data Structures
ITCS 6114/8114

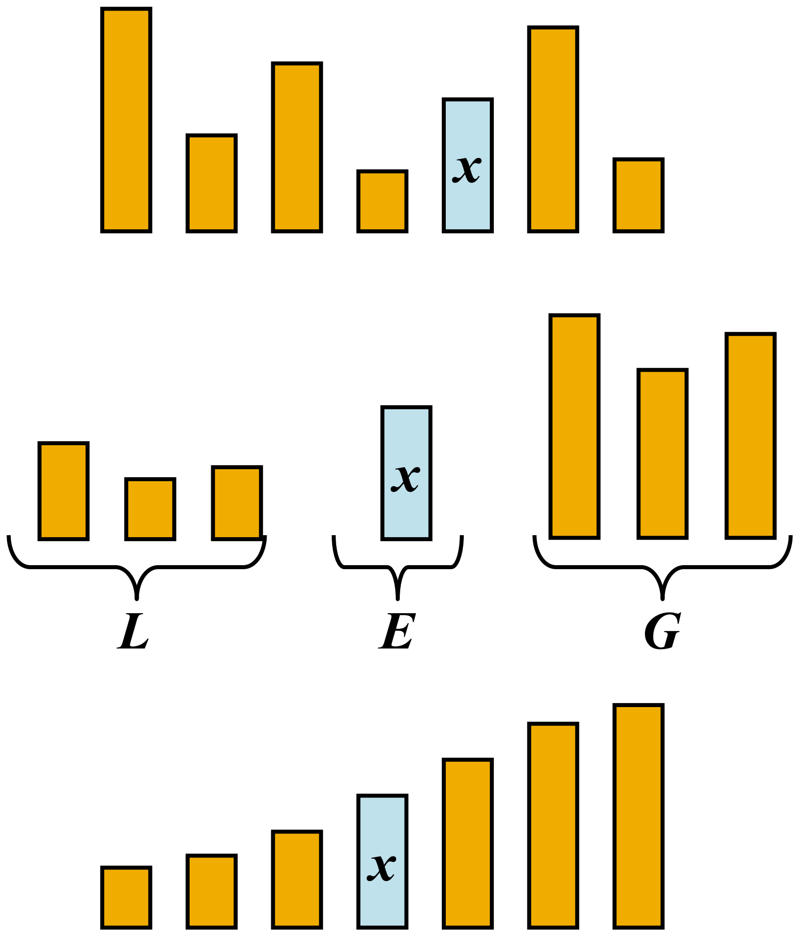
Dr. Dewan Tanvir Ahmed
Department of Computer Science
University of North Carolina at Charlotte

Outline and Reading

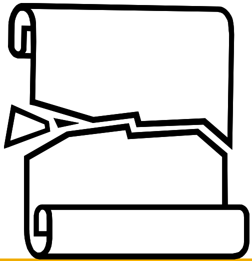
- Quick-sort (§4.3)
  - Algorithm
  - Partition step
  - Quick-sort tree
  - Execution example
- Analysis of quick-sort (4.3.1)
- In-place quick-sort (§4.8)
- Summary of sorting algorithms

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 to x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



Partition

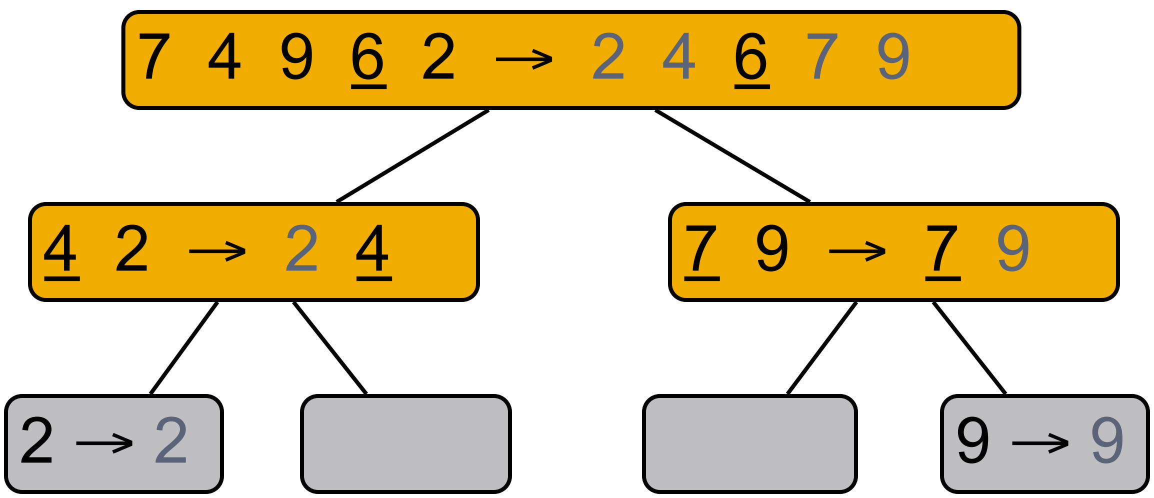


- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the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

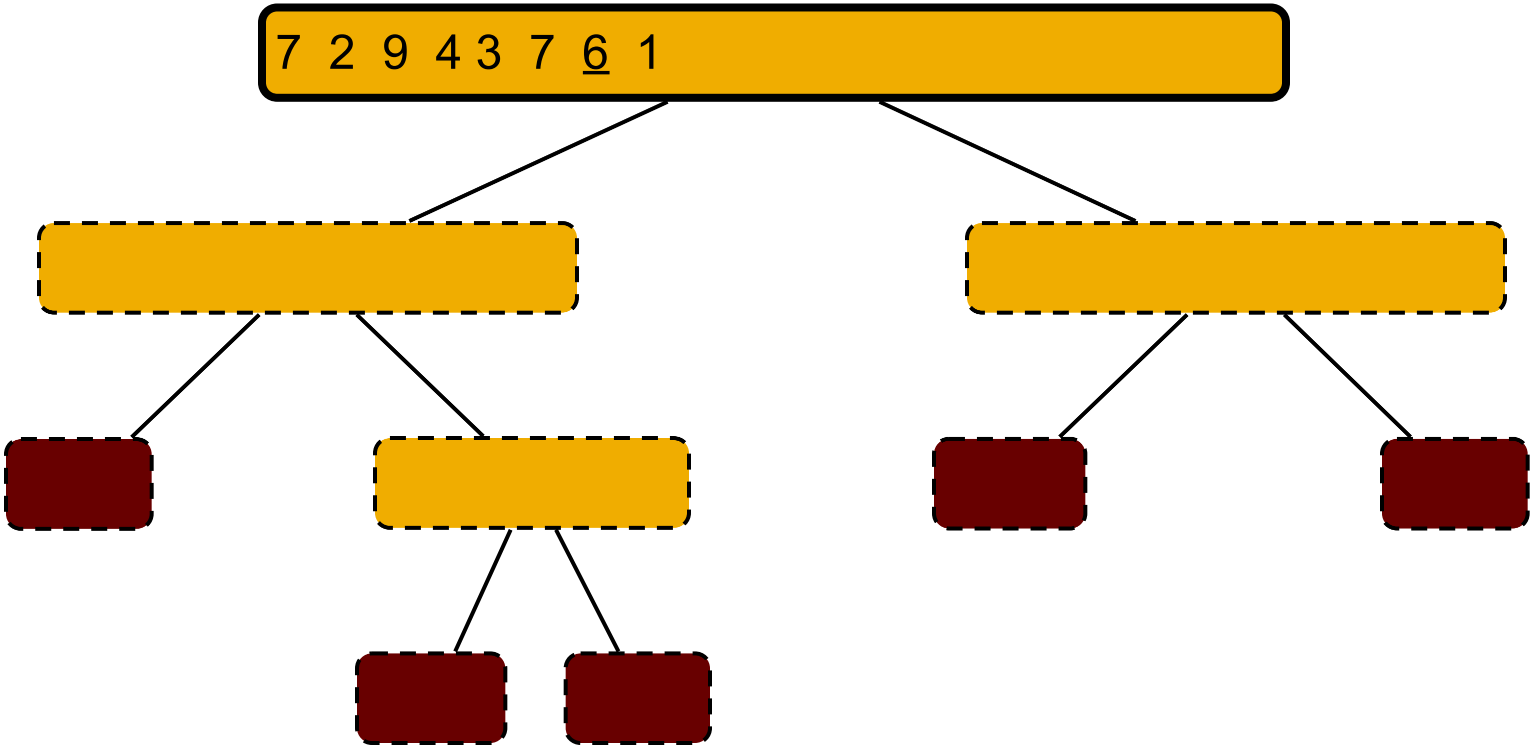
Quick-Sort Tree

- 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



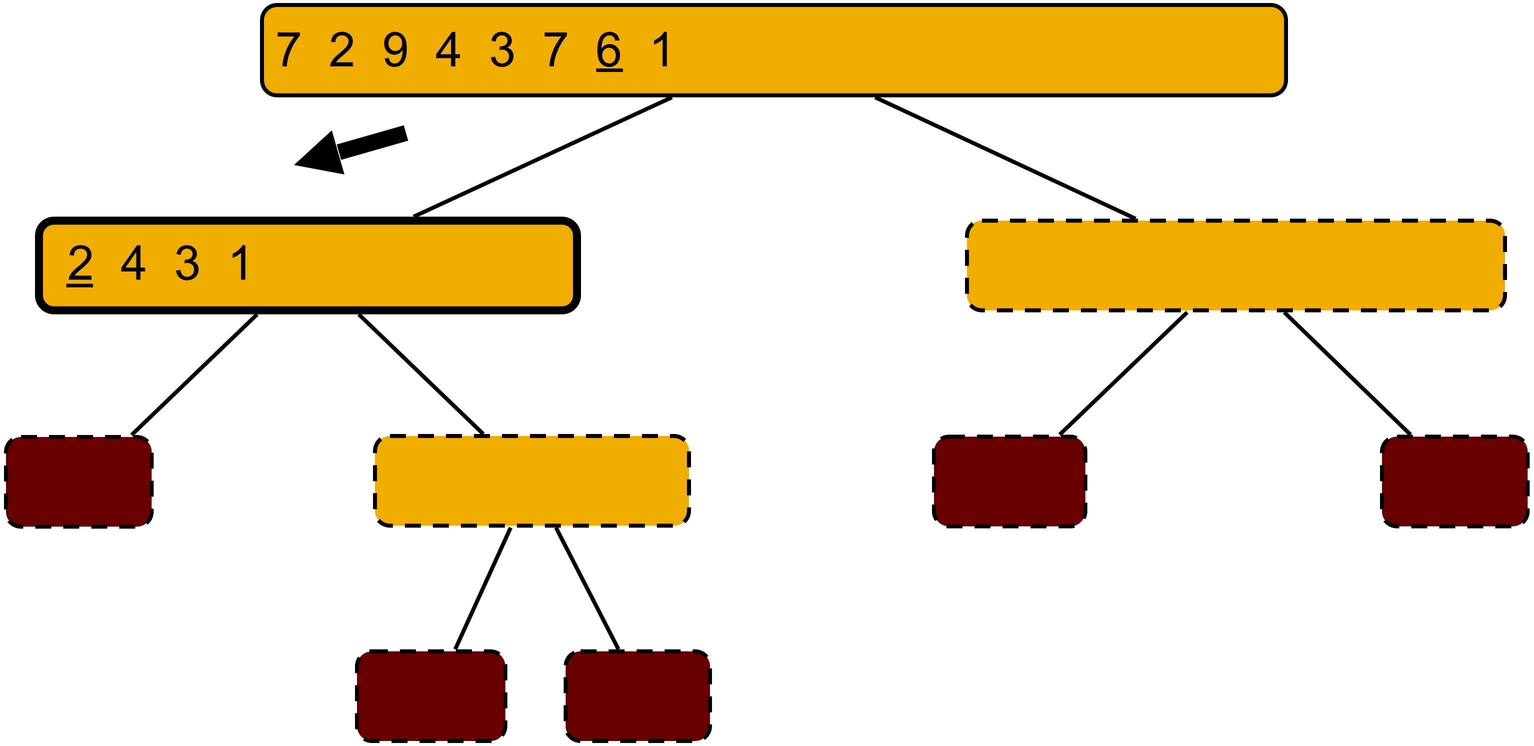
Execution Example

- Pivot selection



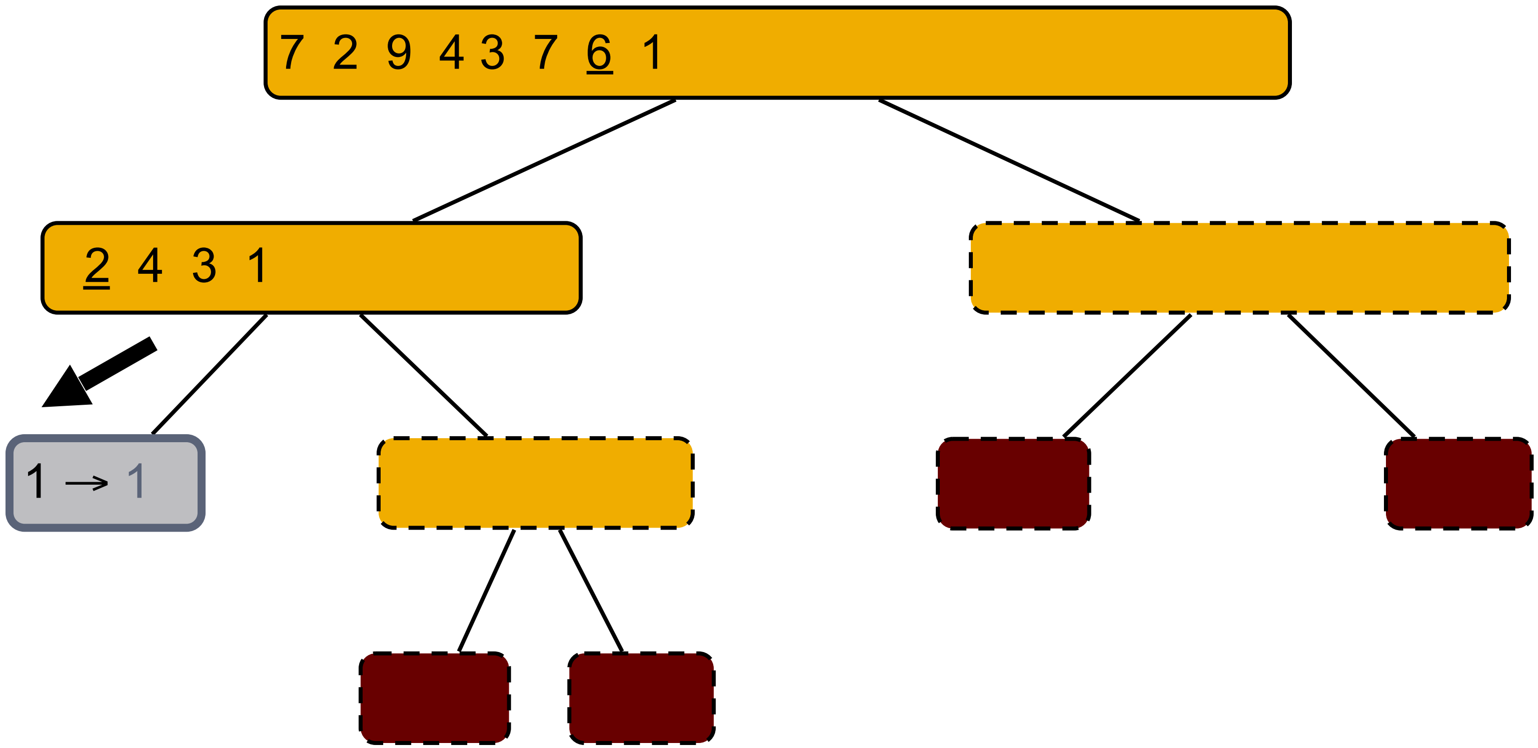
Execution Example (cont.)

- Partition, recursive call, pivot selection



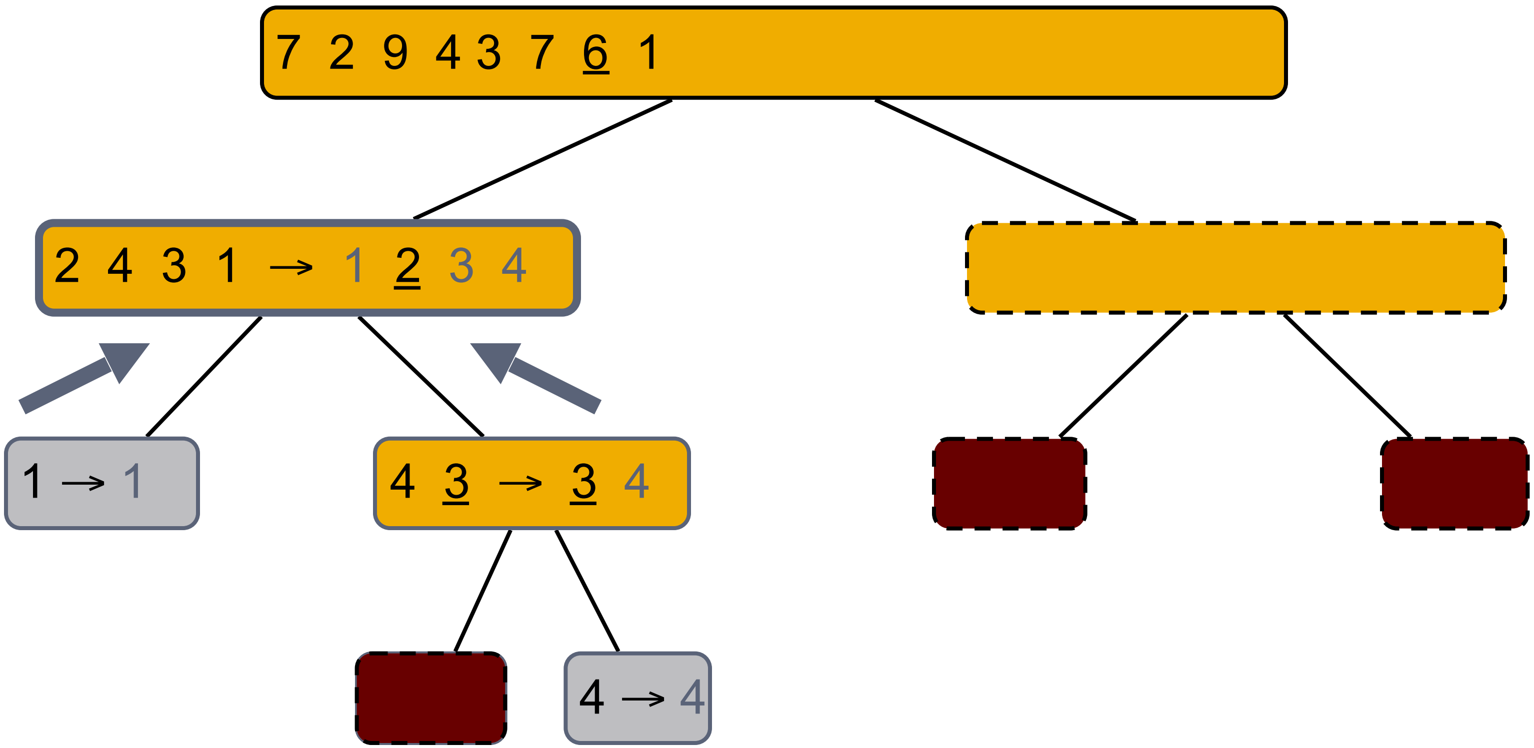
Execution Example (cont.)

- Partition, recursive call, base case



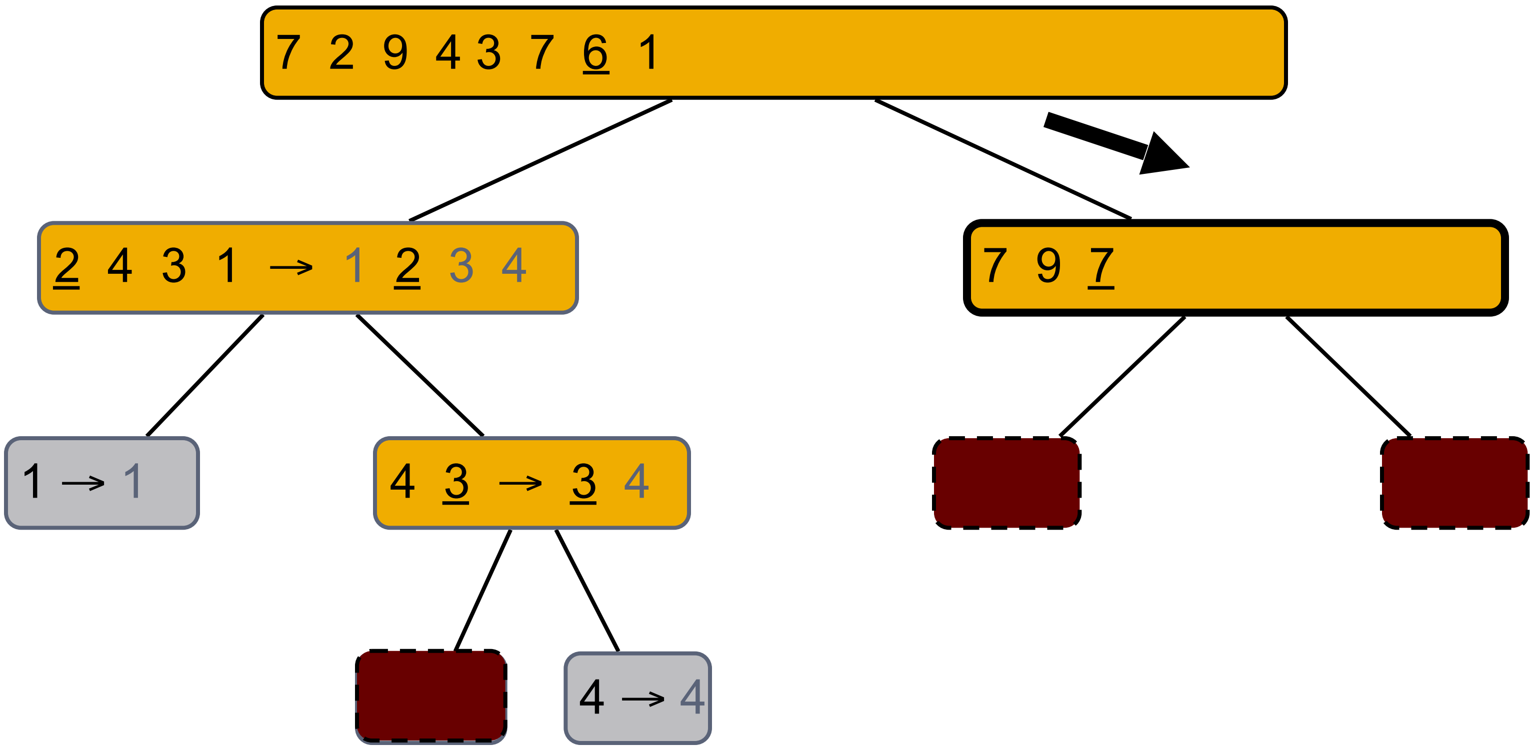
Execution Example (cont.)

- Recursive call, …, base case, join



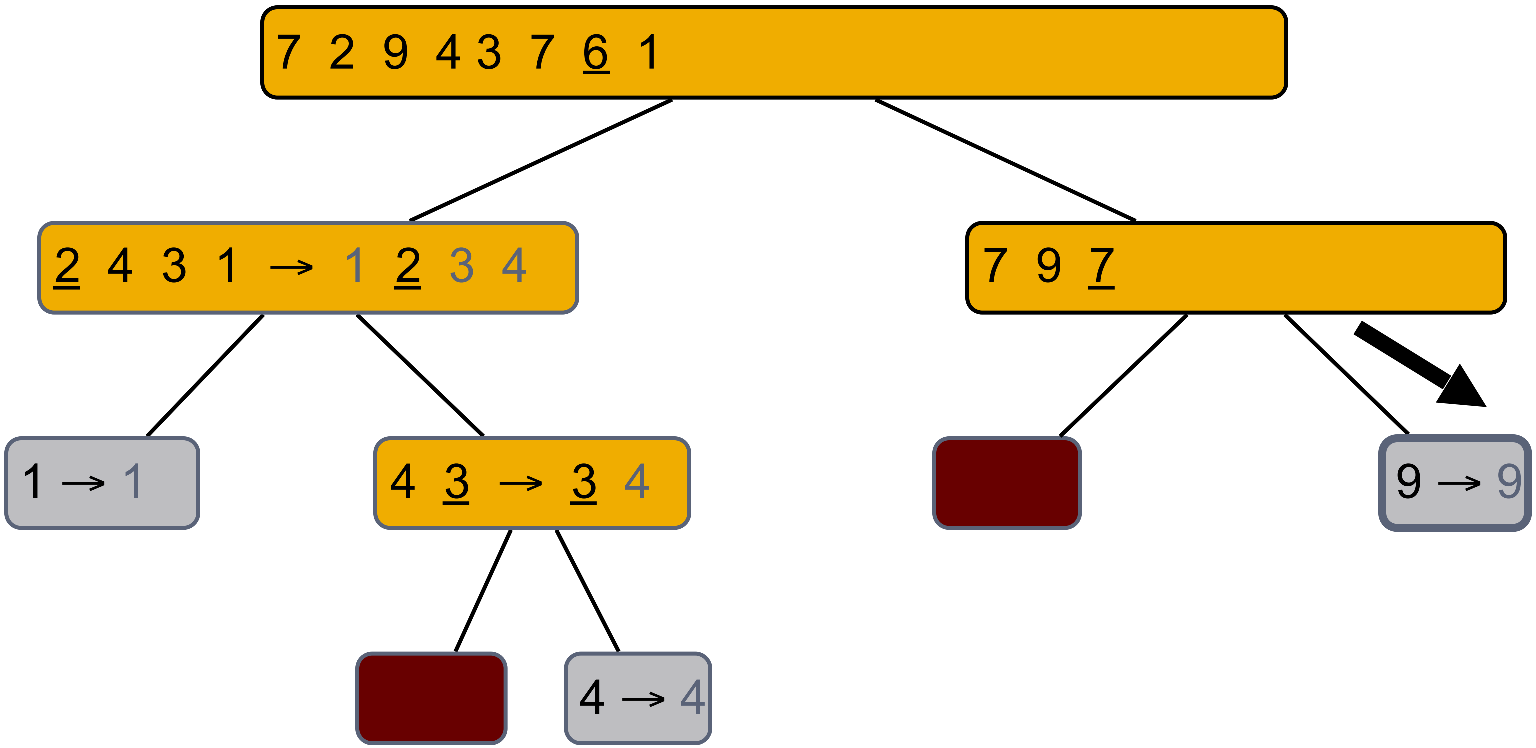
Execution Example (cont.)

- Recursive call, pivot selection



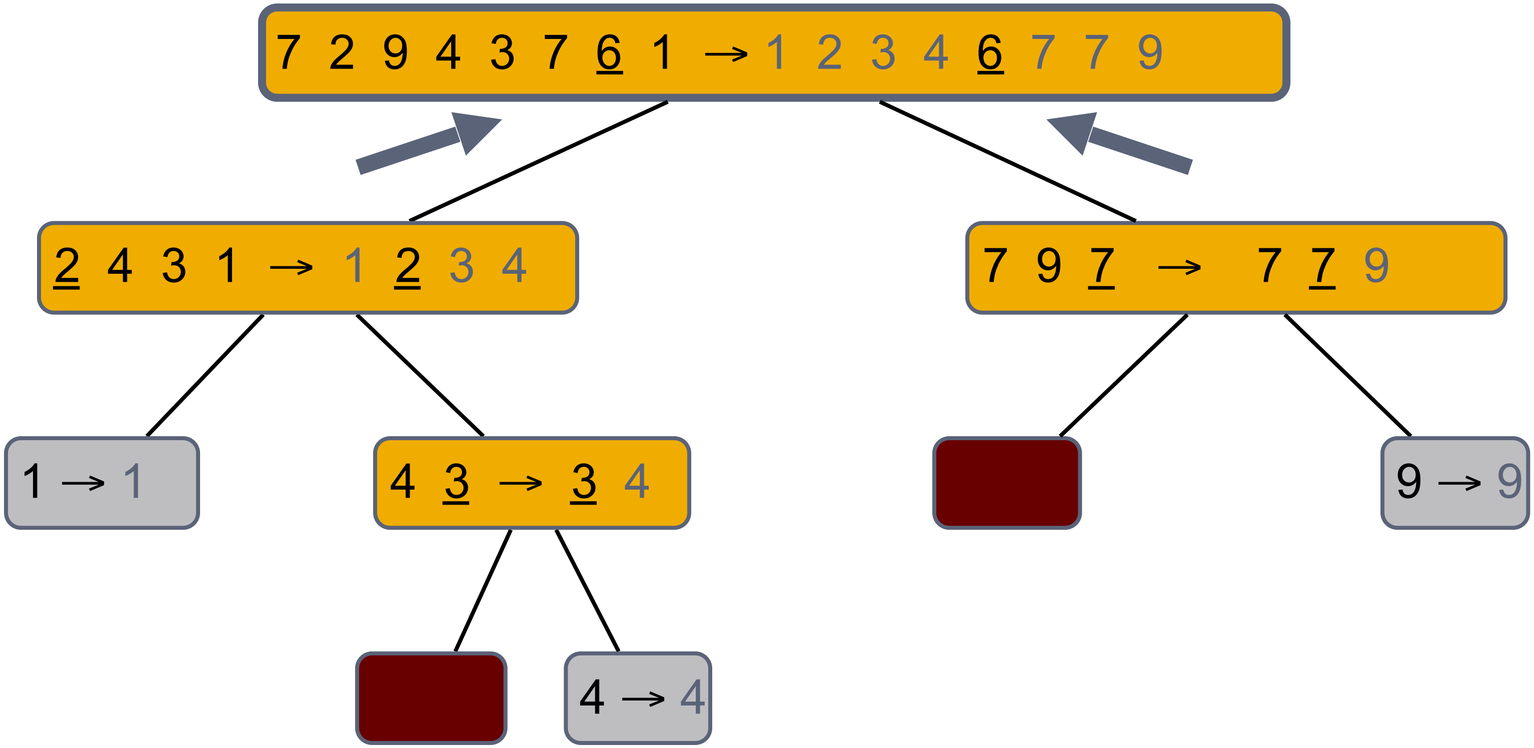
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



Quicksort - Best case

- We cut the array size in half each time
- So the depth of the recursion in $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $O(\log_2 n) * O(n) = O(n\log_2 n)$
- Hence in the best case, quicksort has time complexity $O(n\log_2 n)$

Worst-case Running Time

- In the worst-case, partitioning always divides the size `n` array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length **`n-1`** part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length **`n-1`** part requires (in the worst case) recurring to depth **`n-1`**

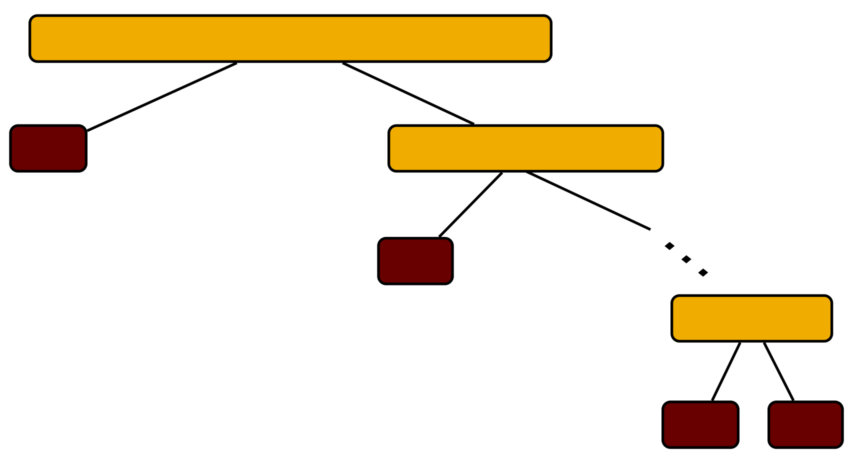
Worst-case Running Time (cont..)

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



Worst-case Running Time

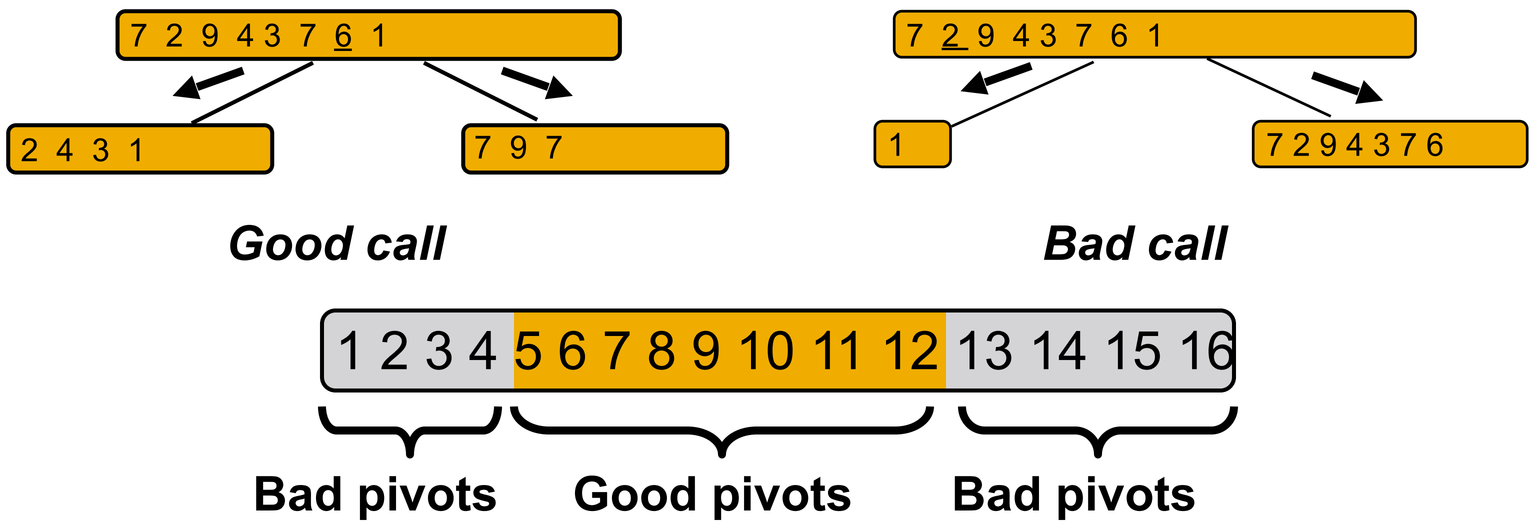
- When does this happen?
  - There are many arrangements that *could* make this happen
  - Here are two common cases:
    - When the array is already sorted
    - When the array is *inversely* sorted (sorted in the opposite order)

Typical case for quicksort

- If the array is sorted to begin with, Quicksort is terrible: **$O(n^2)$**
- However, Quicksort is *usually* $O(n \log_2 n)$
- Quicksort is generally the fastest algorithm known
- Most real-world sorting is done by Quicksort

Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size n
  - **Good call:** the sizes of L and G are at most $3n/4$ and at least n/4
  - **Bad call:** one of L and G has size greater than $3n/4$
- A call is **good** with probability $1/2$
  - 1/2 of the possible pivots cause good calls:

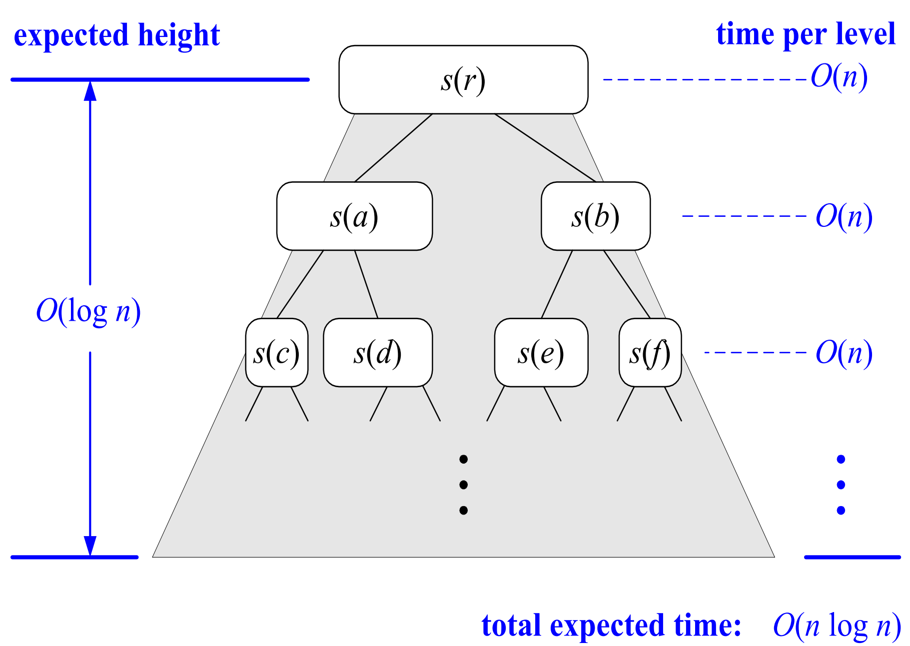


Expected Running Time, Part 2

- 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

If a node v of the quicksort tree T is associated with a "good" recursive call, the input size of the children of v are at most $3s(v)/4$ [i.e. $\frac{s(v)}{4/3}$]

If we take a path in T from the root to an external node, then the length of this path is at most the number of invocations that have to be made until achieving $\log_{4/3} n$ good invocations.

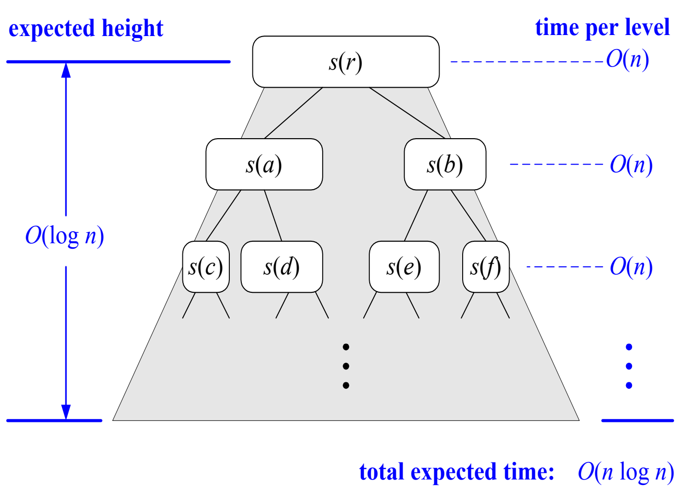


Expected Running Time, Part 2

- The expected number of invocations we must make until this occurs is $2log_{4/3}n$ (if a path terminates before this level, this is better)
- *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).

j *k*

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 6 9 *(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`**

j *k*

3 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 **6** 9

Example of partitioning

choose pivot:	4 3 6 9 2 7 3 1 2 1 8 9 3 5 6
search:	4 3 6 9 2 7 3 1 2 1 8 9 3 5 6
swap:	4 3 3 9 2 7 3 1 2 1 8 9 6 5 6
search:	4 3 3 9 2 7 3 1 2 1 8 9 6 5 6
swap:	4 3 3 1 2 7 3 1 2 9 8 9 6 5 6
search:	4 3 3 1 2 7 3 1 2 9 8 9 6 5 6
swap:	4 3 3 1 2 2 3 1 7 9 8 9 6 5 6
search:	4 3 3 1 2 2 3 1 7 9 8 9 6 5 6 (left > right)
swap with pivot:	1 3 3 1 2 2 3 4 7 9 8 9 6 5 6

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)

Choice Of Pivot

Three ways to choose the pivot:

- Pivot is **rightmost (or leftmost)** element in list that is to be sorted
  - When sorting *A*[1:30], use *A*[30] (or A[1]) as the pivot
- **Randomly** select one of the elements to be sorted as the pivot
  - When sorting *A*[1:30], generate a random number *r* in the range [1, 30]
  - Use *A*[*r*] as the pivot

Small arrays

- Quicksort does not perform well for very small arrays
- How small depends on many factors, such as
  - the time spent making a recursive call, the compiler, etc.
- So, do not use quicksort recursively for small arrays
  - Instead, use a sorting algorithm that is efficient for small arrays, such as **insertion sort**

Pivot: median of three

- Use the median of the array
  - Partitioning always cuts the array into roughly half
  - An optimal quicksort ($O(n \log n)$)
  - However, hard to find the exact median
    - e.g., sort an array to pick the value in the middle

Pivot: median of three

- **Median-of-Three** rule - from the leftmost, middle, and rightmost elements of the list to be sorted, select the one with median key as the pivot
  - When sorting $A[1{:}20]$
    - examine $A[1]$, $A[10]$ $((1+20)/2)$, and $A[20]$
    - Select the element with median (i.e., middle) key
  - If
    - $A[1] = 30$,
    - $A[10] = 3$, and
    - $A[20] = 12$,
    - $A[20]$ becomes the pivot

Pivot: median of three

- We will use median of three
  - Compare just three elements: the leftmost, rightmost and center
  - Swap these elements if necessary so that
    - A[left] = Smallest
    - A[right] = Largest
    - A[center] = Median of three
  - Pick A[center] as the pivot
  - Swap A[center] and A[right – 1] so that pivot is at second last position (why?)

Pivot: median of three

median3

```
center = (left+right) / 2;

 if ( a[center] < a[left] )
     swap( a+left, a+center);
 if ( a[right] < a[left] )
     swap( a+left, a+right);
 if ( a[right] < a[center] )
     swap( a+center, a+right);

 swap(a+center, a+right-1);
```

Pivot: median of three

2	5	6	4	**13**	3	12	19	6

A[left] = 2, A[center] = 13, A[right] = 6

2	5	6	4	6	3	12	19	13

Swap A[center] and A[right]

2	5	6	4	6	3	12	19	13

pivot

Choose A[center] as pivot

2	5	6	4	19	3	12	6	13

pivot

Swap pivot and A[right – 1]

So now only need to partition A[left + 1, …, right – 2]. Why?

Main Quicksort Routine

```
if (left + 10 <= right)
{
    int pivot = medianOfThree_pivot (a, left, right);
    int i = left, j = right - 2;
    for( ; ; ){
        while ( a[++i] < pivot );
        while ( pivot < a[--j] );
        if ( i < j )
            swap(a+i, a+j);
        else break;
    }
    swap (a+i, a+right-1);
    median3QuickSort( a, left, i-1);
    median3QuickSort( a, i+1, right);
}
else
    insertionSort(a, left, right);
```

Choose pivot

Partitioning

Recursion

For small arrays

Quicksort Faster than Mergesort

- Both quicksort and mergesort take $O(n \log n)$ in the average case.
- Why is quicksort faster than mergesort?
  - The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
  - There is no extra juggling as in mergesort.

```
int i = left, j = right - 2;
for( ; ; ){
    while ( a[++i] < pivot );
    while ( pivot < a[--j] );
    if ( i < j )
        swap(a+i, a+j);
    else break;
}
```

inner loop

Analysis

- Assumptions:
  - A random pivot (no median-of-three partitioning)
  - No cutoff for small arrays
- Running time
  - pivot selection: constant time, i.e. O(1)
  - partitioning: linear time, i.e. O(N)
  - running time of the two recursive calls
- $T(N) = T(i) + T(N - i - 1) + cN$ where c is a constant
  - i: number of elements in S_1

Worst-Case Analysis

- What will be the worst case?
  - The pivot is the smallest element, all the time
  - Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

Best-case Analysis

$$T(N) = 2T\left(\frac{N}{2}\right) + cN$$

$$\frac{T(N)}{N} = \frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} + c$$

$$\frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} = \frac{T\left(\frac{N}{4}\right)}{\frac{N}{4}} + c$$

$$\frac{T\left(\frac{N}{4}\right)}{\frac{N}{4}} = \frac{T\left(\frac{N}{8}\right)}{\frac{N}{8}} + c$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

- What will be the best case?
  - Partition is perfectly balanced.
  - Pivot is always in the middle (median of the array)

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c\log N$$

$$T(N) = cN\log N + N = O(N\log N)$$

Average-Case Analysis

- Assume
  - Each of the sizes for S1 is equally likely
- This assumption is valid for our pivoting (median-of-three) strategy
- On average, the running time is O(N log N)

Reference

- Algorithm Design: Foundations, Analysis, and Internet Examples. Michael T. Goodrich and Roberto Tamassia. John Wiley & Sons.
- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.