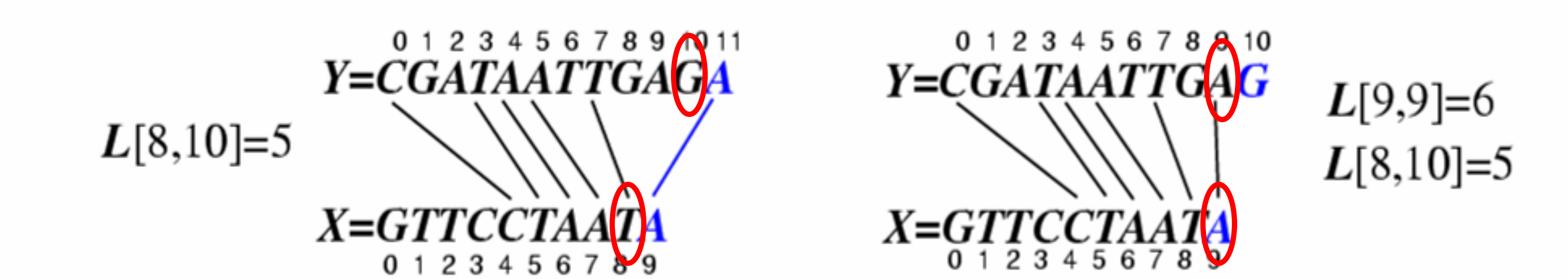
A Poor Approach to the LCS Problem

- A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.
- Analysis:
  - If X is of length n, then it has 2^n subsequences
  - This is an exponential-time algorithm!

A Dynamic-Programming Approach to the LCS Problem

- Define L[i,j] to be the length of the longest common subsequence of X[0..i] and Y[0..j].
- Allow for -1 as an index, so L[-1,k] = 0 and L[k,-1]=0, to indicate that the null part of X or Y has no match with the other.
- Then we can define L[i,j] in the general case as follows:
  1. If $x_i=y_j$, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
  2. If $x_i \neq y_j$, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)



An LCS Algorithm

Algorithm LCS(X,Y):

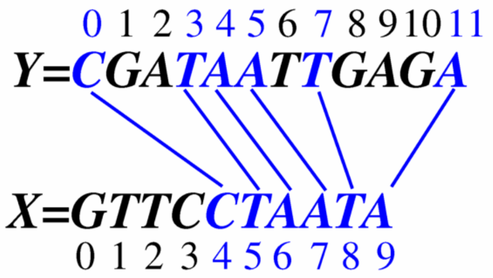
Input: Strings X and Y with n and m elements, respectively

Output: For i = 0,…,n-1, j = 0,...,m-1, the length L[i, j] of a longest string that is a subsequence of both the string X[0..i] = $x_0x_1x_2…x_i$ and the string Y [0.. j] = $y_0y_1y_2…y_j$

for i =1 to n-1 **do**
    L[i,-1] = 0
for j =0 to m-1 **do**
    L[-1,j] = 0
for i =0 to n-1 **do**
    for j =0 to m-1 **do**
        if $x_i = y_j$ **then**
            L[i, j] = L[i-1, j-1] + 1
        else
            L[i, j] = max{L[i-1, j] , L[i, j-1]}
return array L

Visualizing the LCS Algorithm

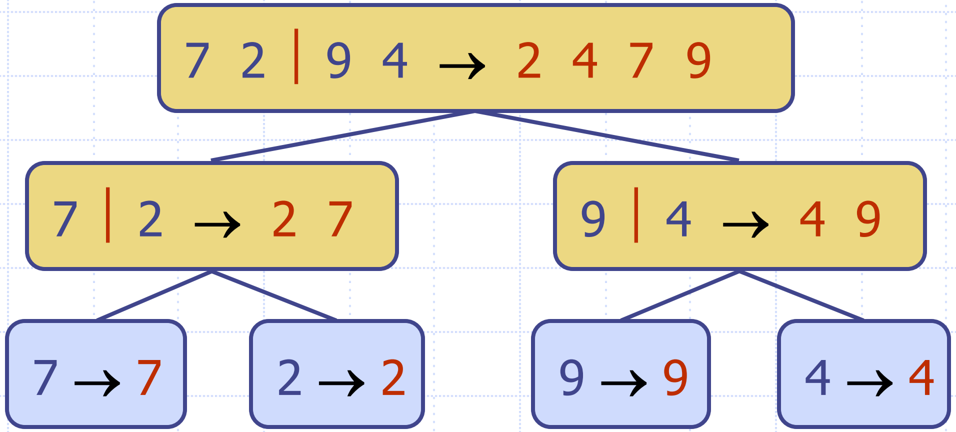
L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**



Analysis of LCS Algorithm

- We have two nested loops
  - The outer one iterates n times
  - The inner one iterates m times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is $O(nm)$
- Answer is contained in L[n,m] (and the subsequence can be recovered from the L table).

Sorting and More Sorting



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

```
Algorithm mergeSort(S)
    Input sequence S with n
        elements
    Output sequence S sorted
        according to C
    if S.size() > 1
        (S1, S2) ← partition(S, n/2)
        mergeSort(S1)
        mergeSort(S2)
        S ← merge(S1, S2)
```

Merging Two Sorted Sequences

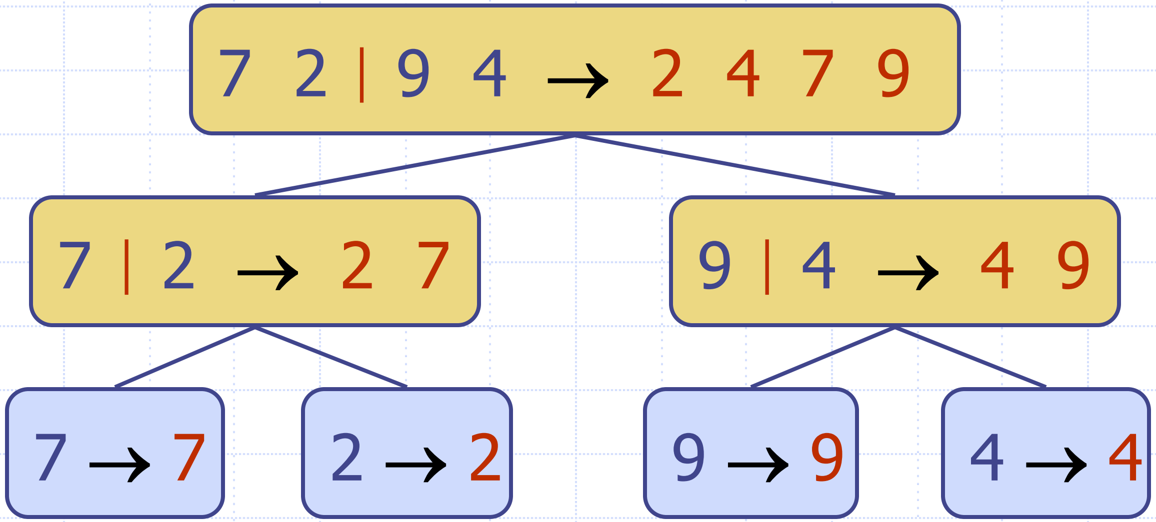
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Algorithm *merge*(A, B)
    Input sequences A and B with
        $n/2$ elements each
    Output sorted sequence of $A \cup B$

    $S \leftarrow$ empty sequence
    while $\neg A.isEmpty() \wedge \neg B.isEmpty()$
        if $A.first().element() <$ $B.first().element()$
            $S.addLast(A.remove(A.first()))$
        else
            $S.addLast(B.remove(B.first()))$
    while $\neg A.isEmpty()$
        $S.addLast(A.remove(A.first()))$
    while $\neg B.isEmpty()$
        $S.addLast(B.remove(B.first()))$
    return S

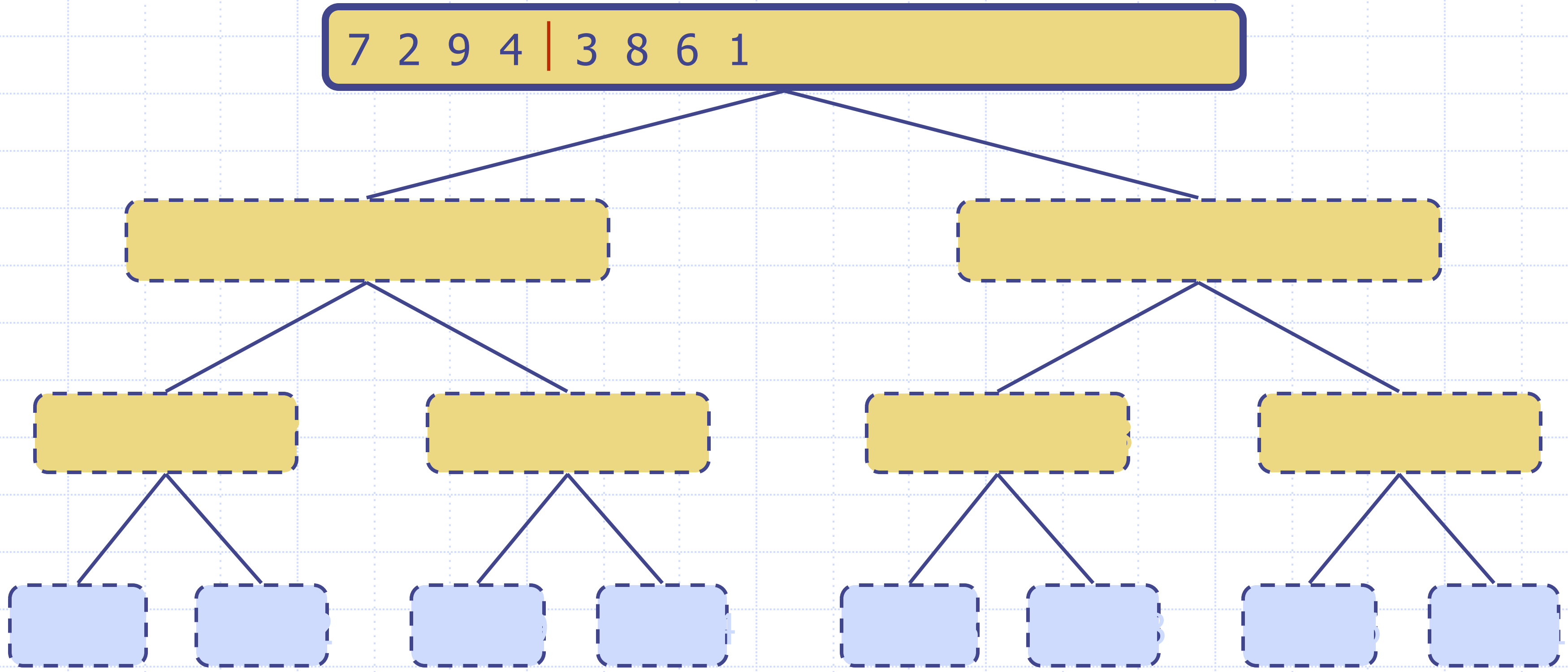
Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - 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



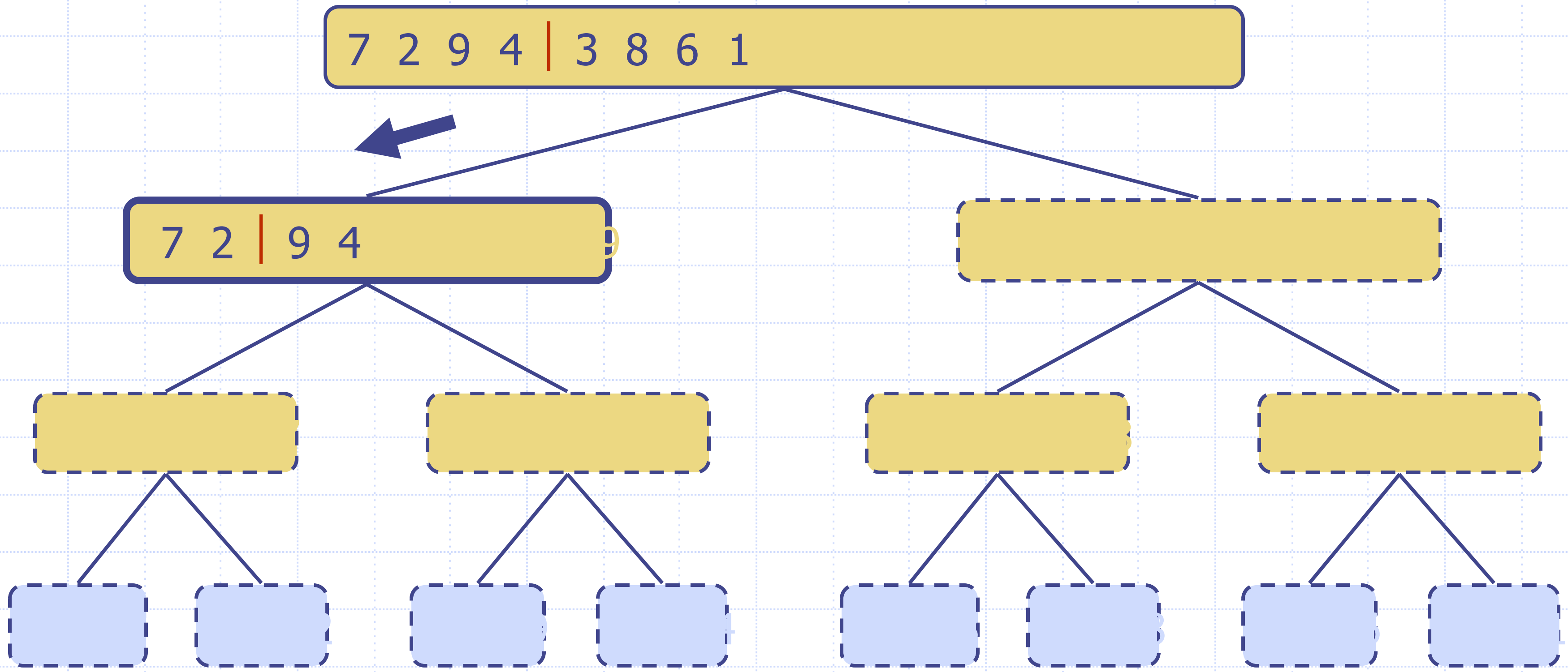
Execution Example

- Partition



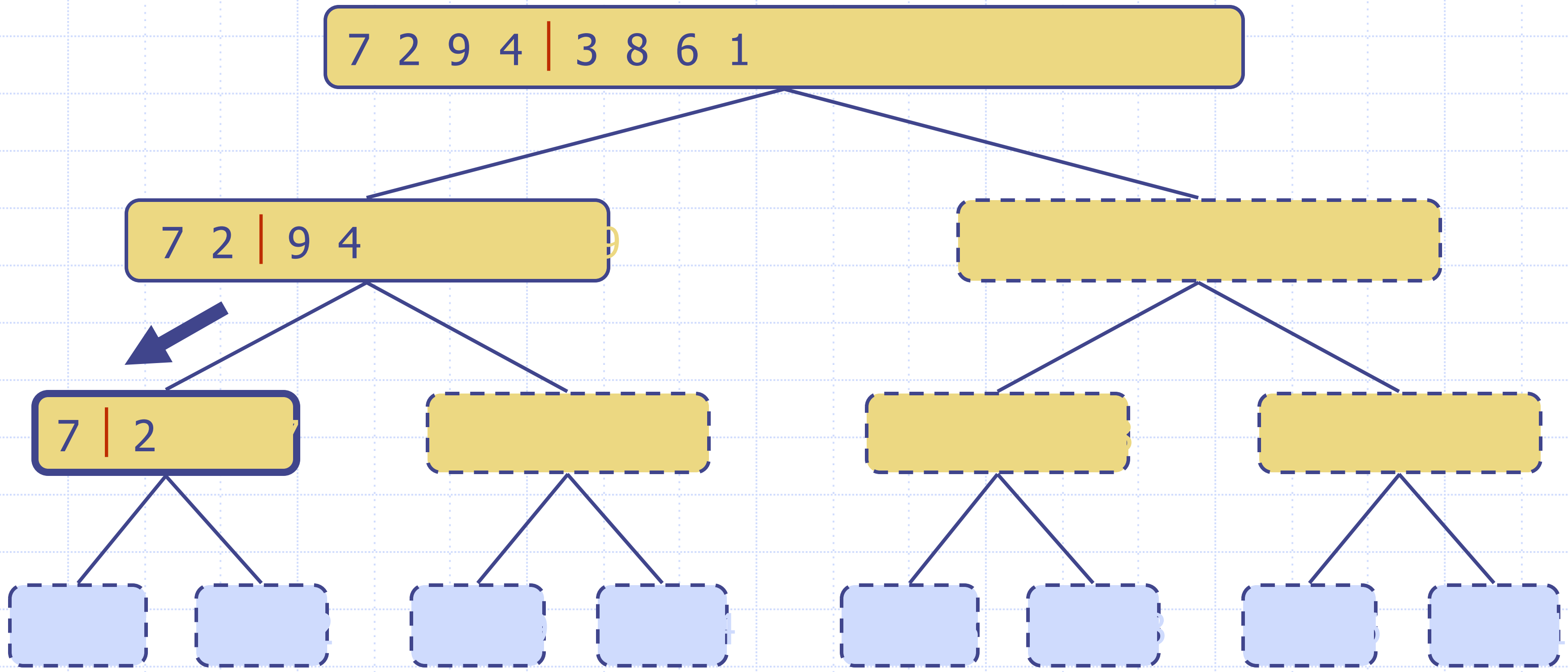
Execution Example (cont.)

- Recursive call, partition



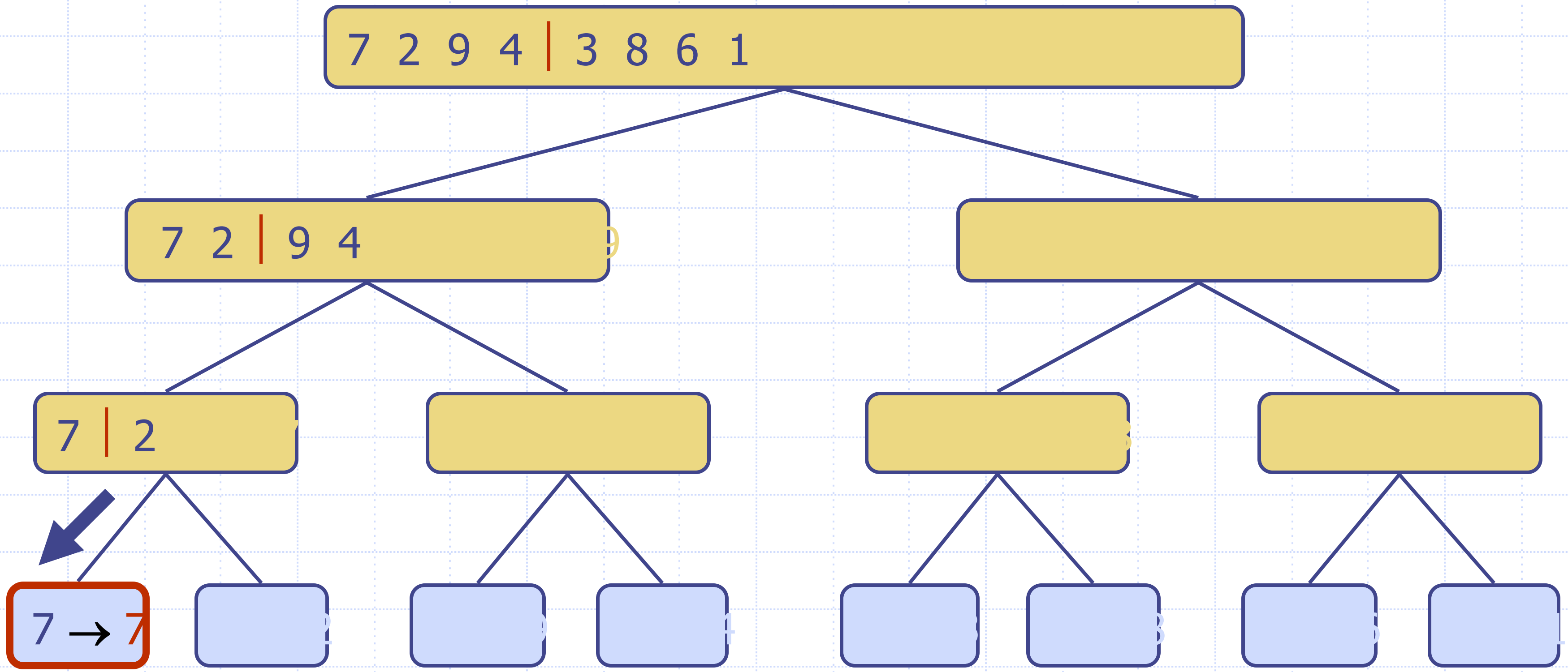
Execution Example (cont.)

- Recursive call, partition



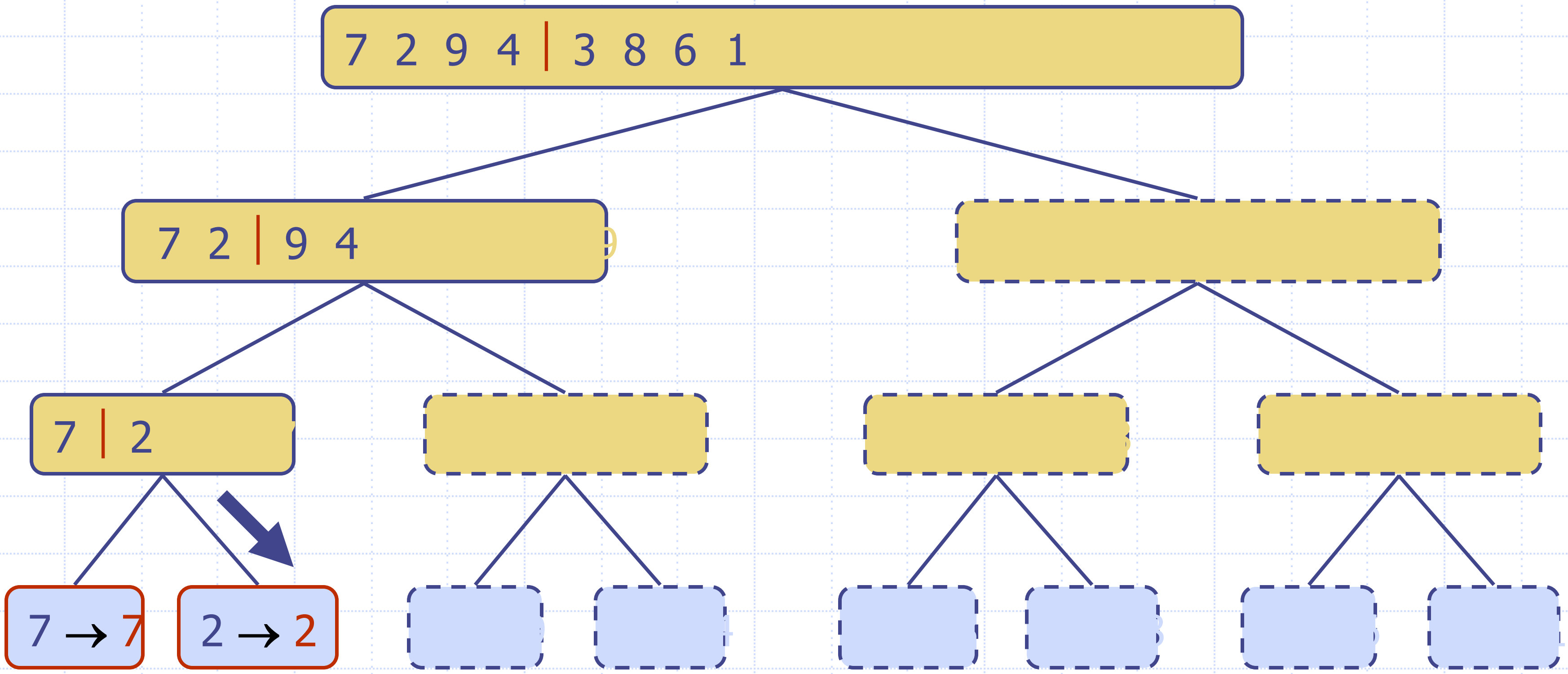
Execution Example (cont.)

- Recursive call, base case



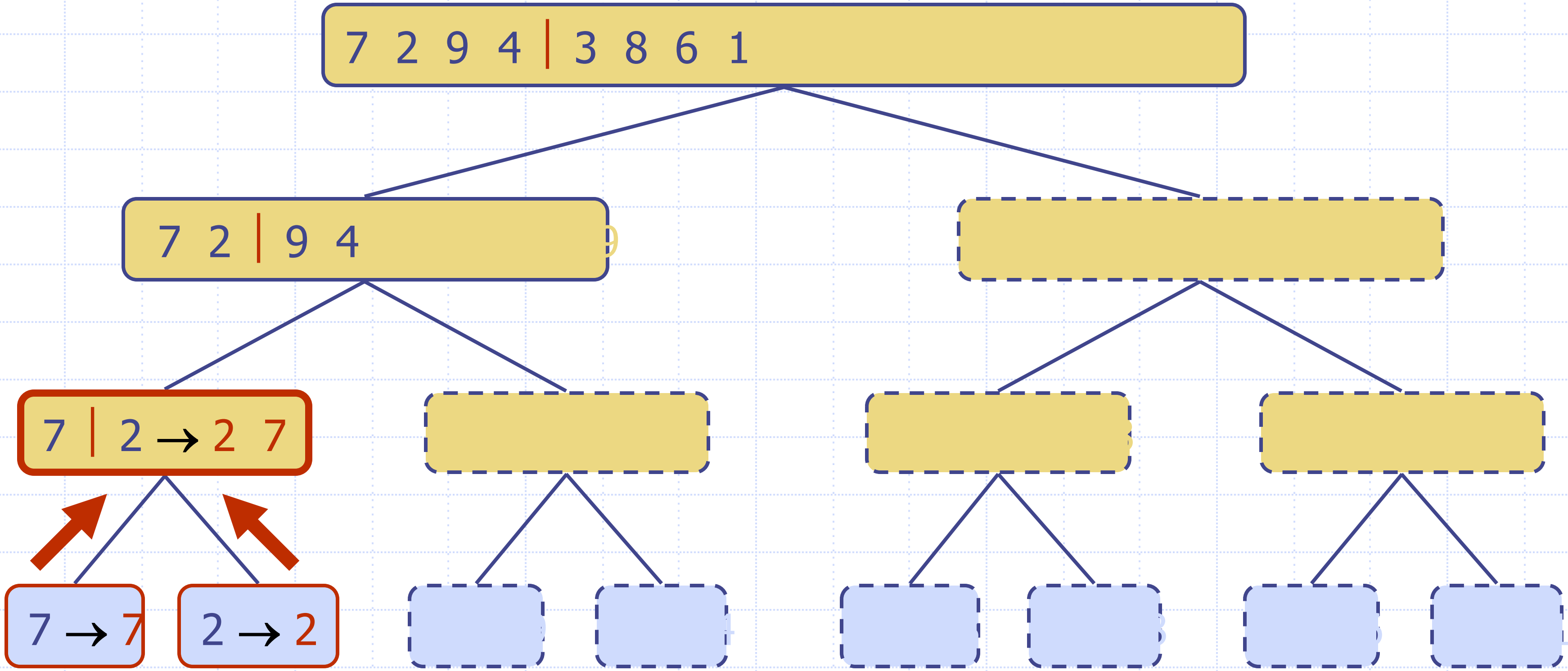
Execution Example (cont.)

- Recursive call, base case



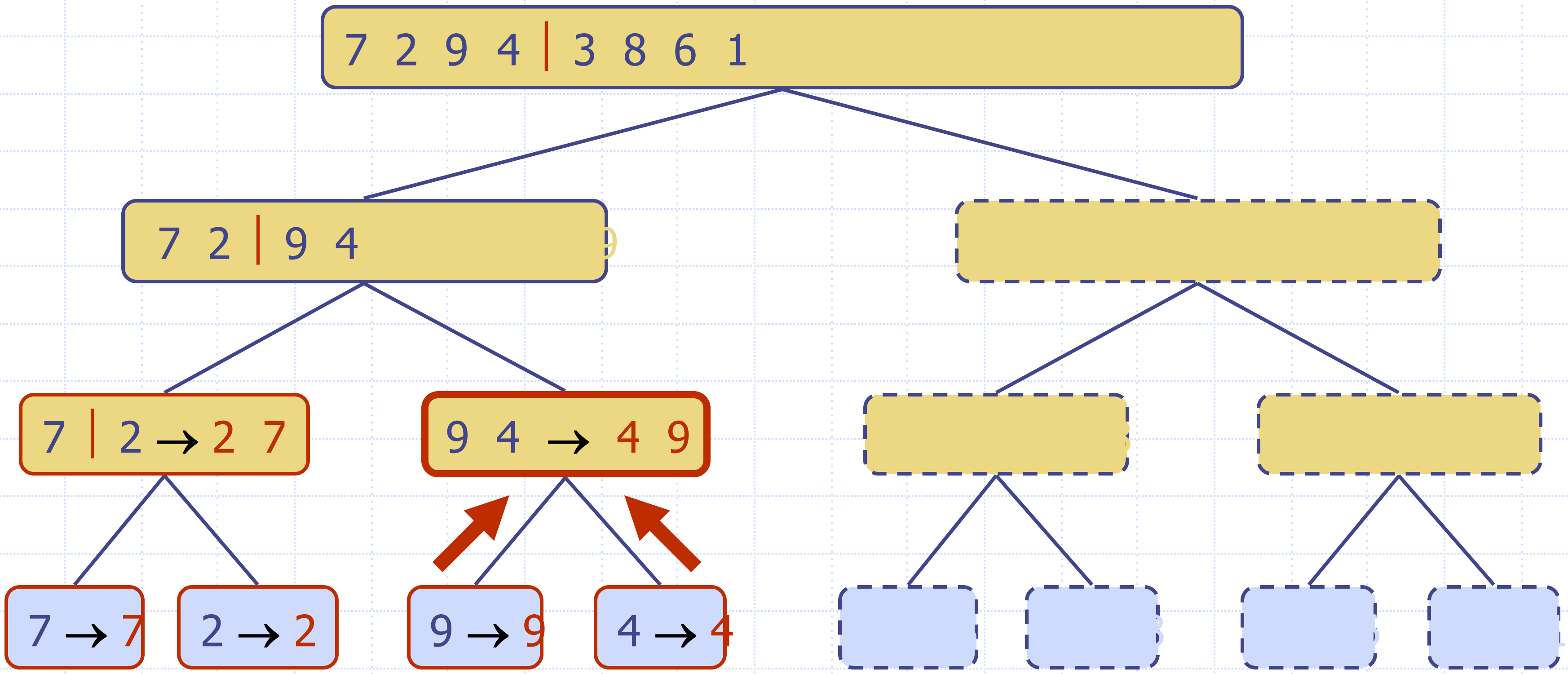
Execution Example (cont.)

- Merge



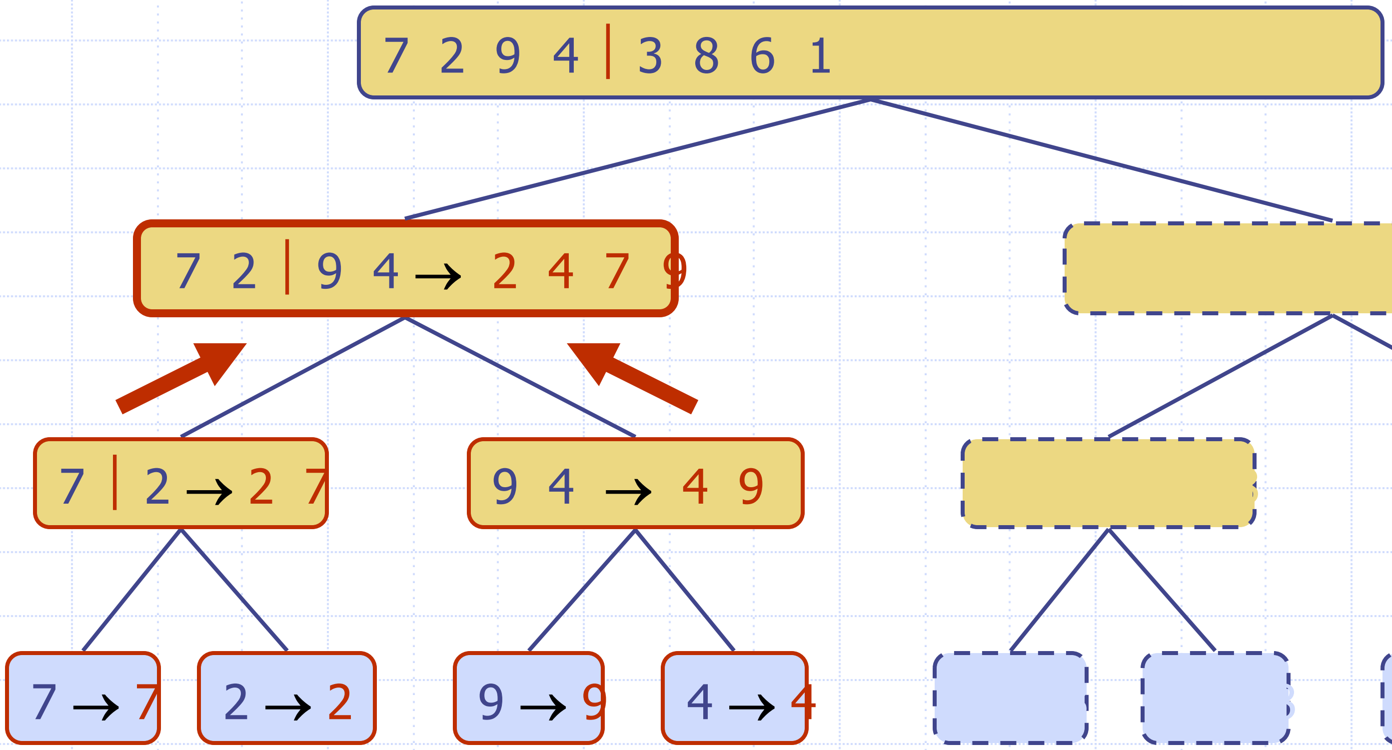
Execution Example (cont.)

- Recursive call, …, base case, merge



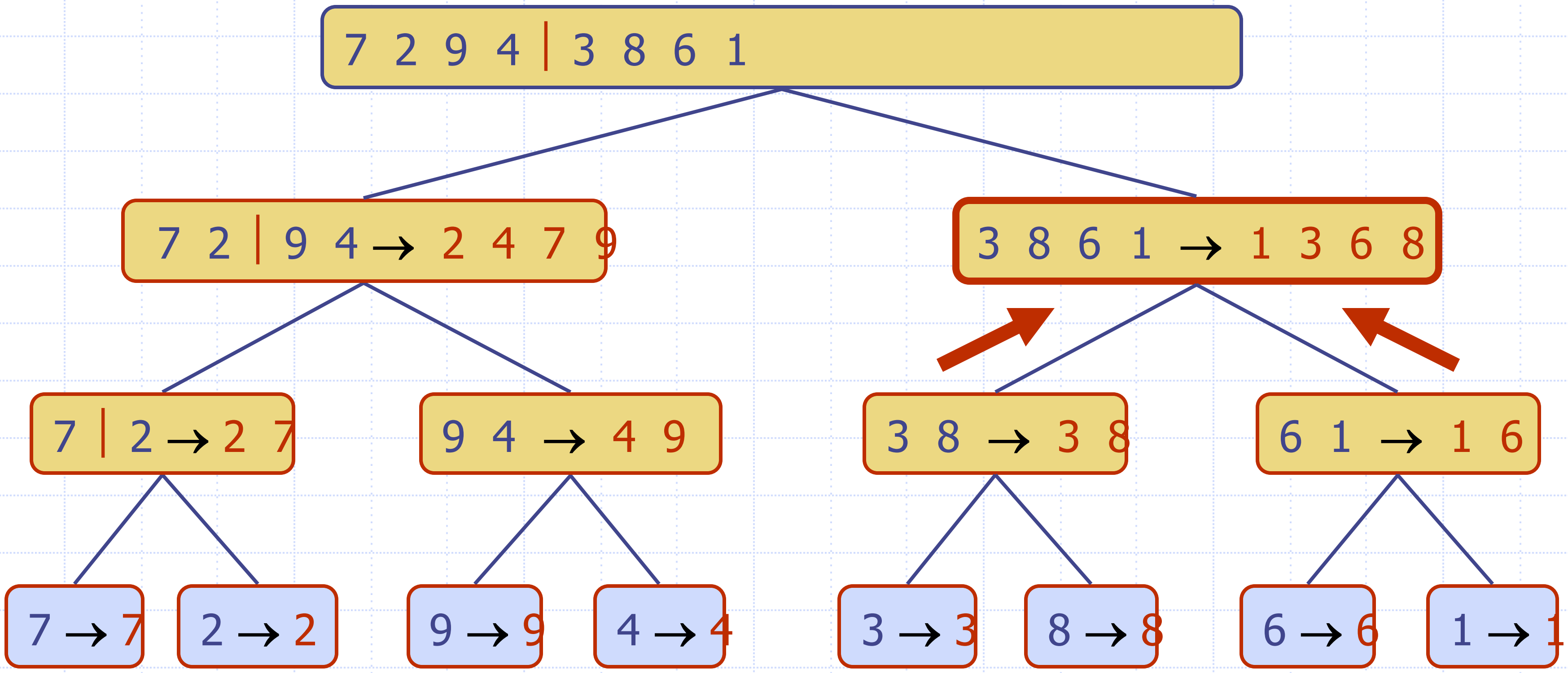
Execution Example (cont.)

- Merge



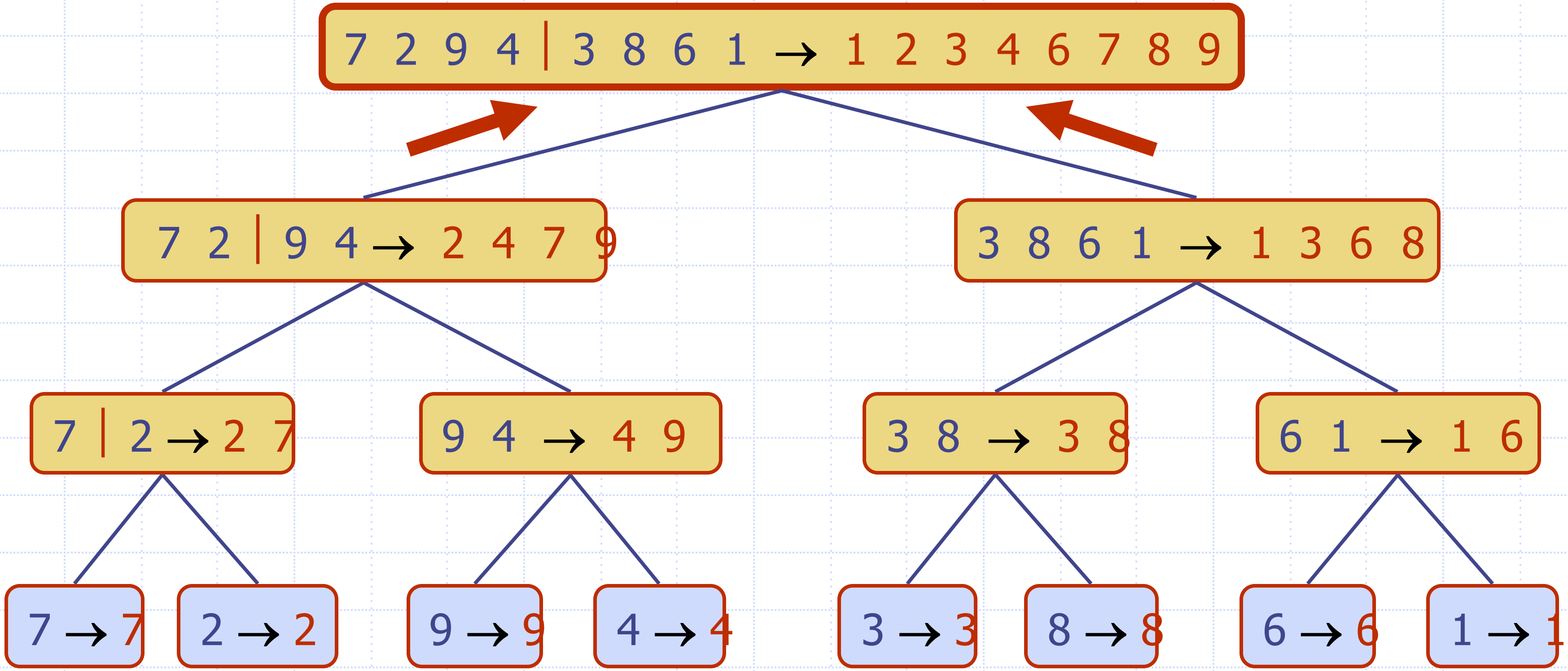
Execution Example (cont.)

- Recursive call, …, merge, merge

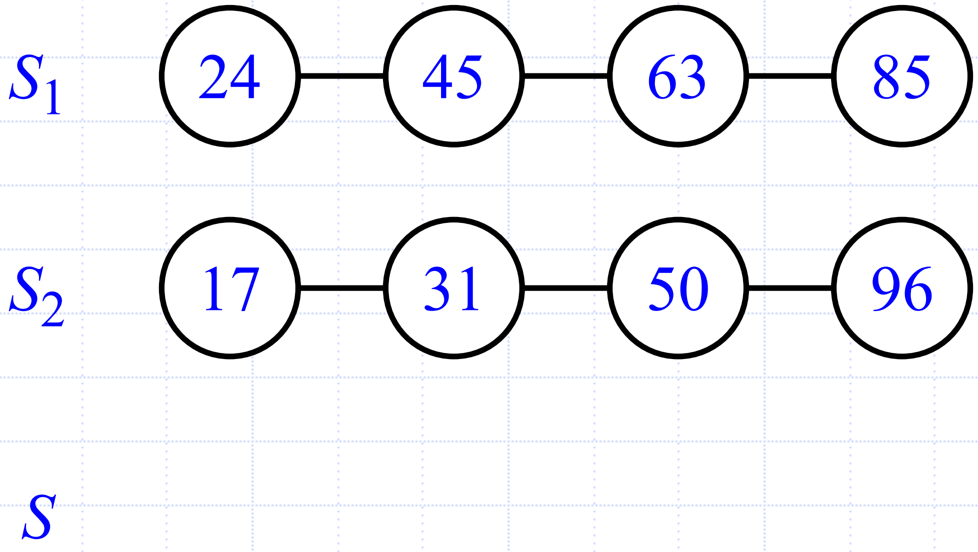


Execution Example (cont.)

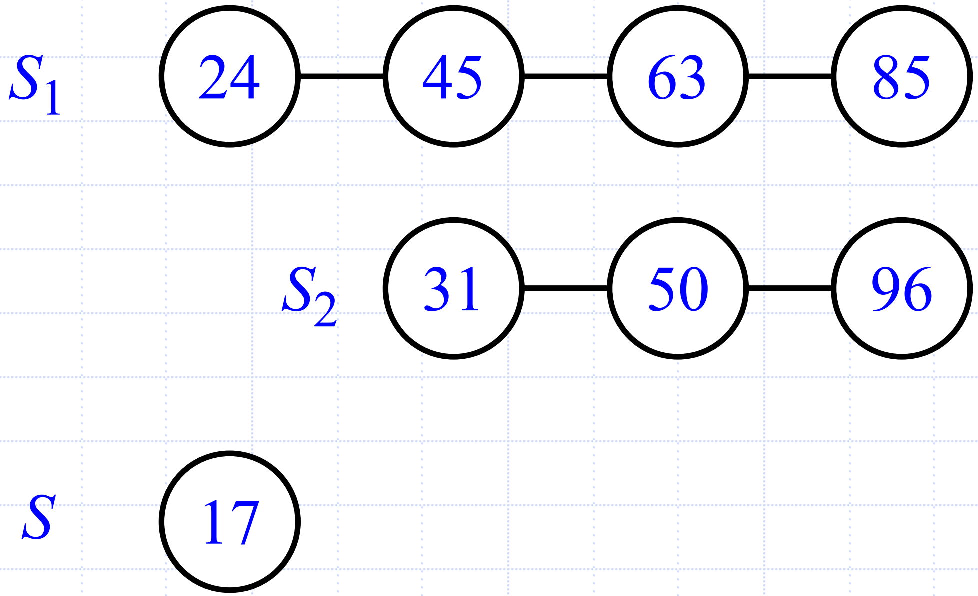
- Merge



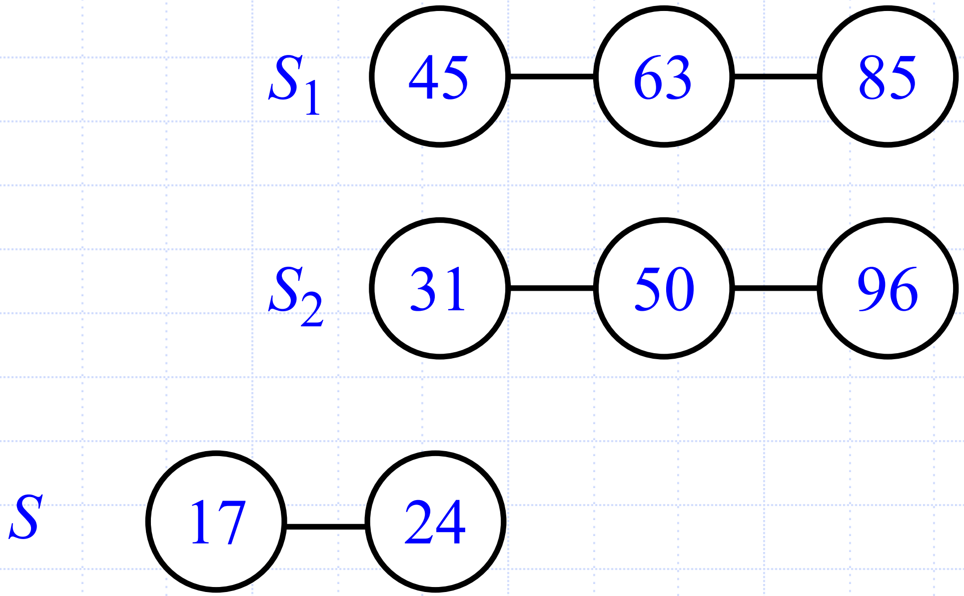
Merge - Example



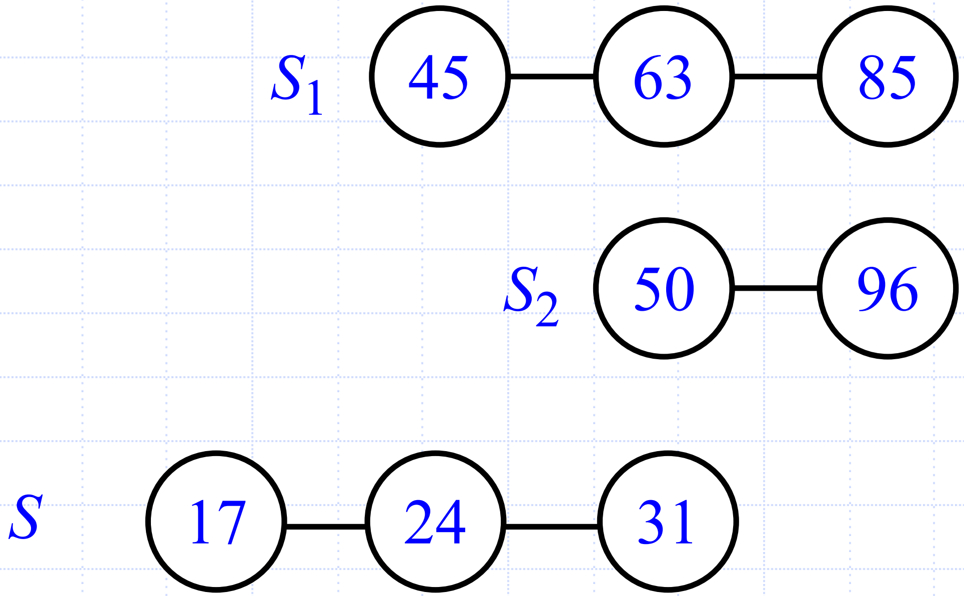
Merge - Example



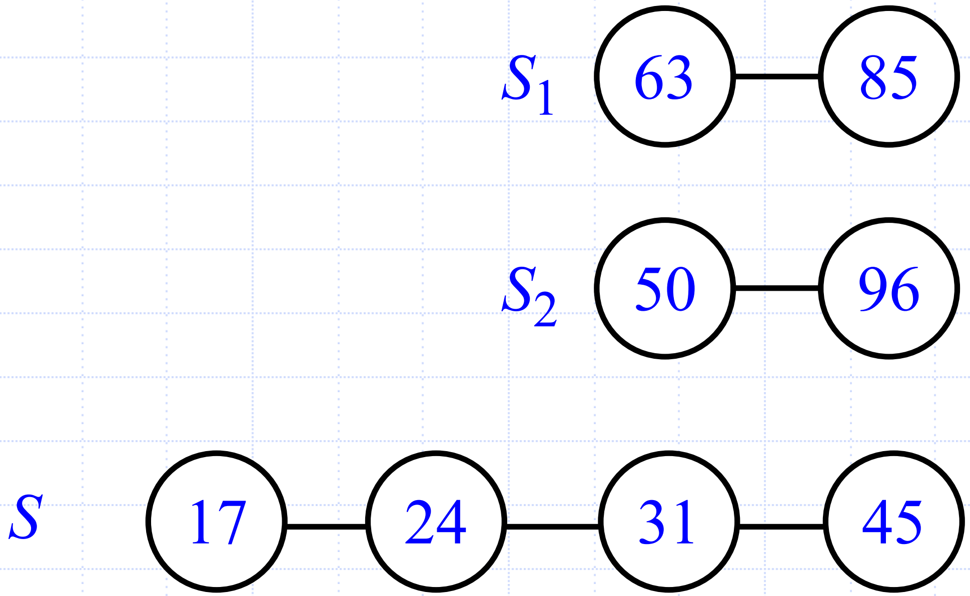
Merge - Example



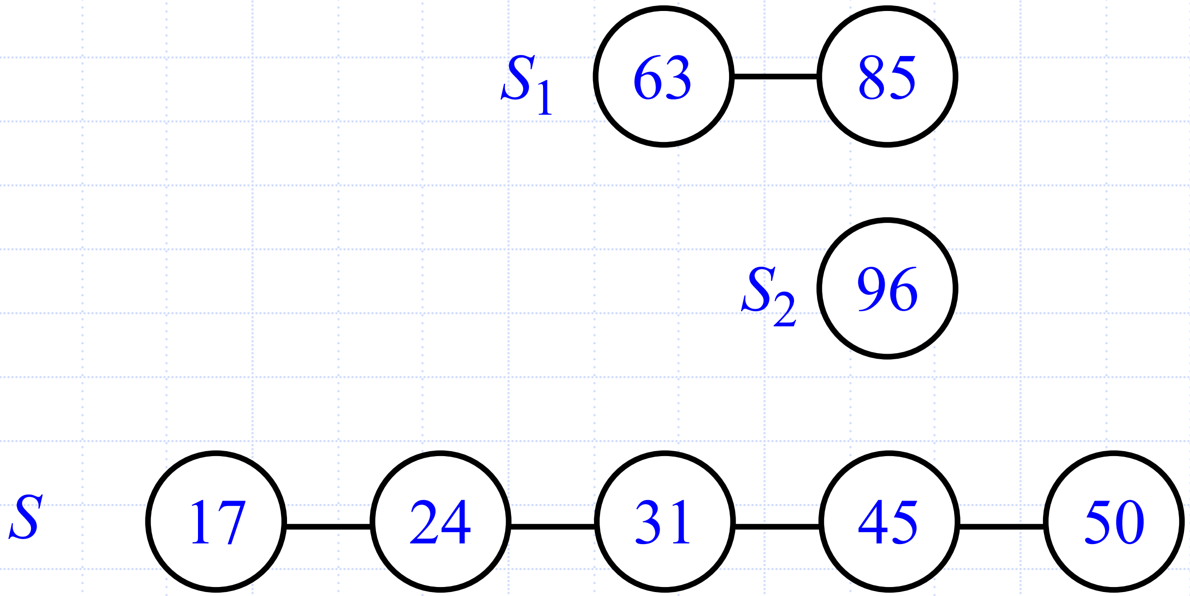
Merge - Example



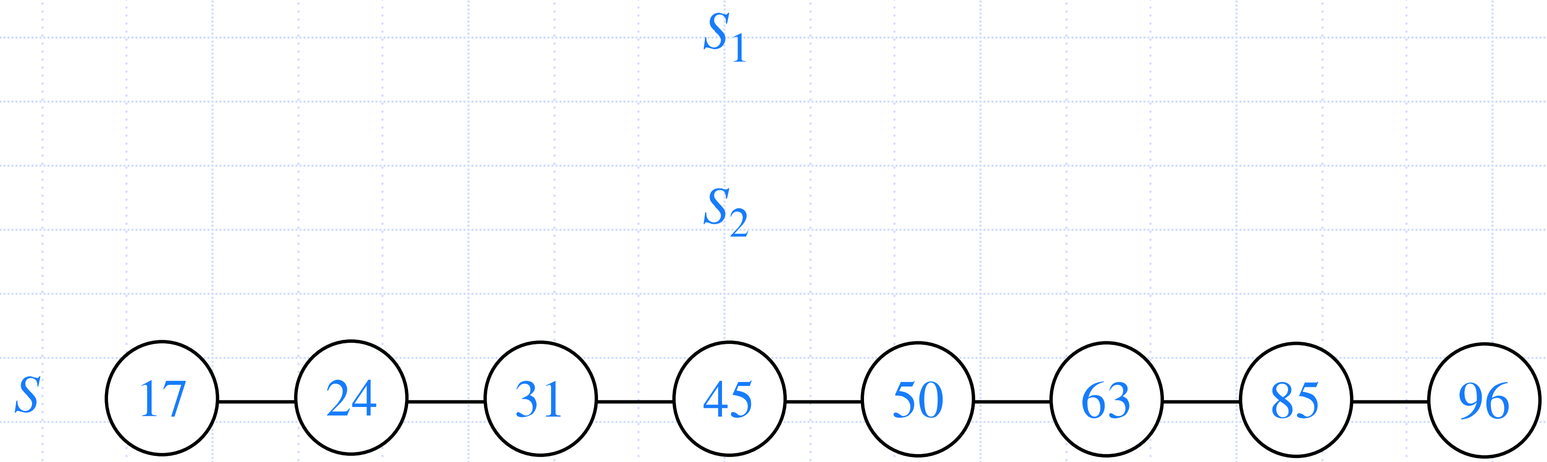
Merge - Example



Merge - Example



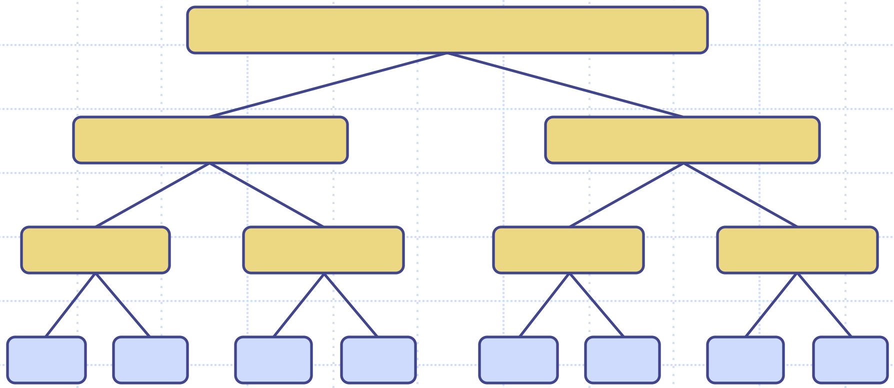
Merge - Example



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount of work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…

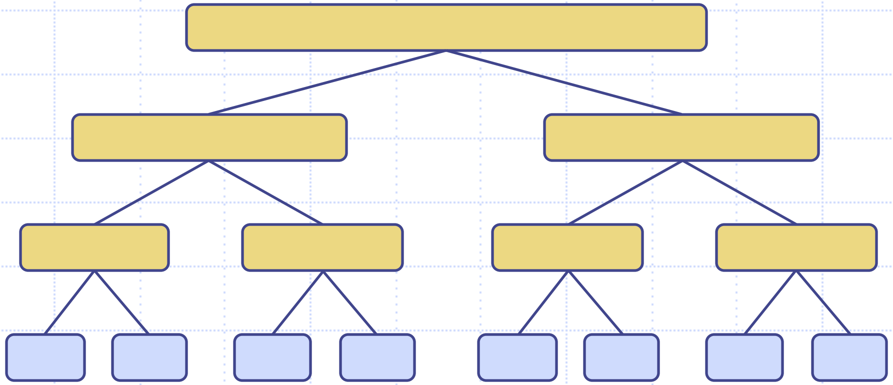


The Recursion Tree

- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



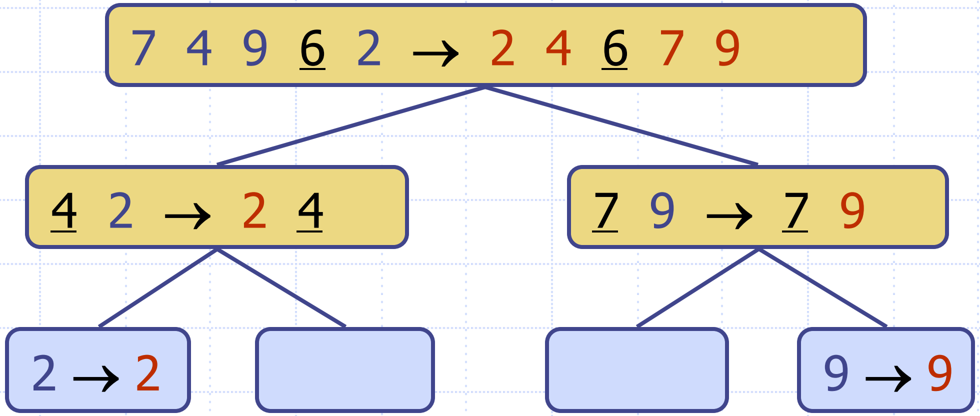
Total time = $bn + bn \log n$

(last level plus all previous levels)

Summary of Sorting Algorithms

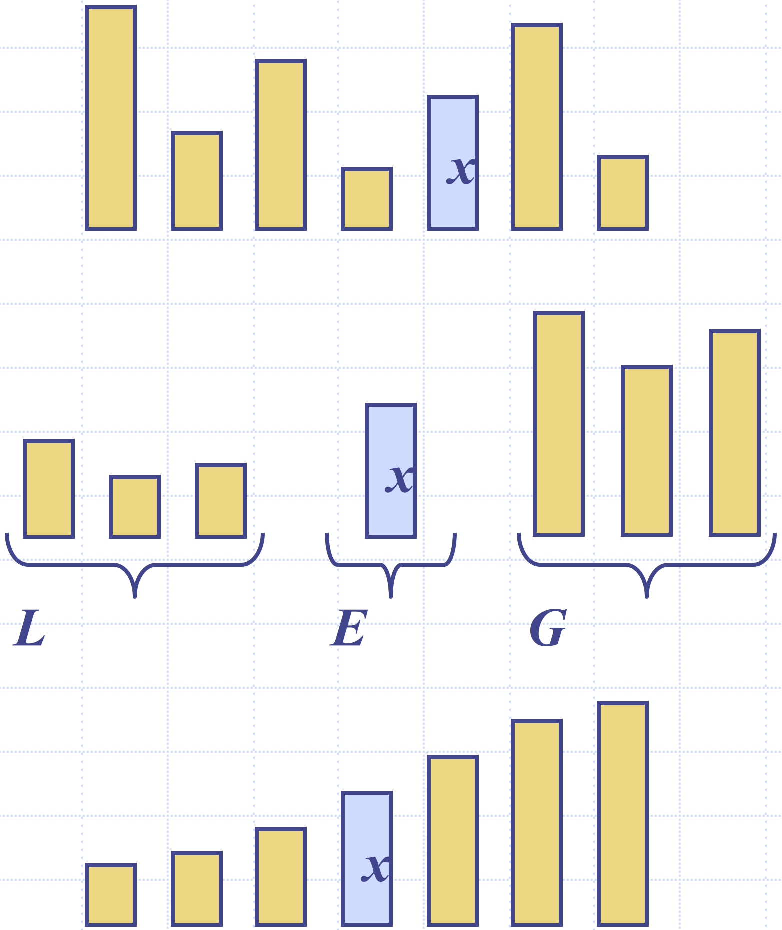
Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	▪ fast ▪ in-place ▪ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	▪ fast ▪ sequential data access ▪ for huge data sets (> 1M)

Quick-Sort



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



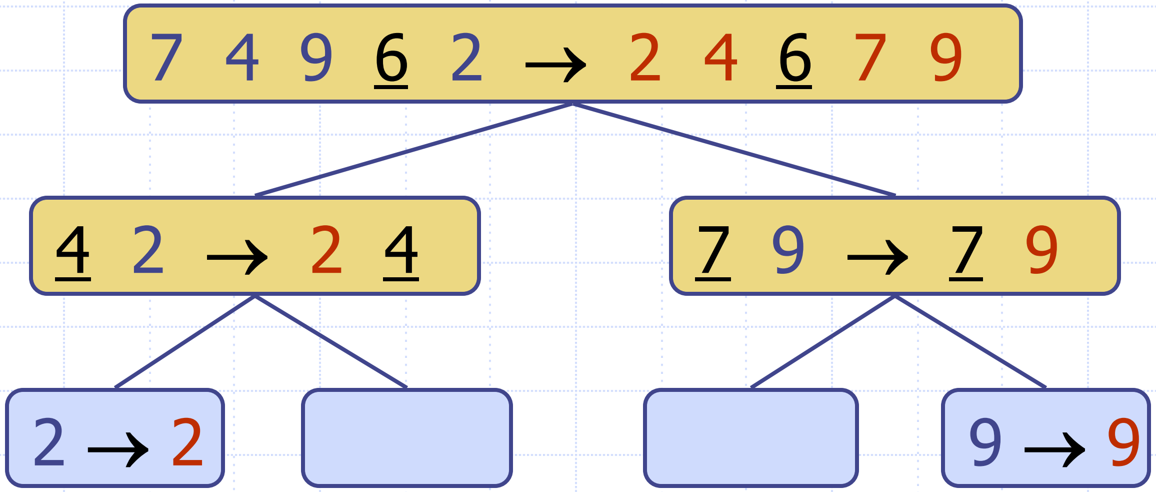
Partition

- 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.
    $L, E, G \leftarrow$ empty sequences
    $x \leftarrow S.remove(p)$
    while $\neg S.isEmpty()$
        $y \leftarrow S.remove(S.first())$
        if $y < x$
            $L.addLast(y)$
        else if $y = x$
            $E.addLast(y)$
        else $\{ y > x \}$
            $G.addLast(y)$
    return L, E, G

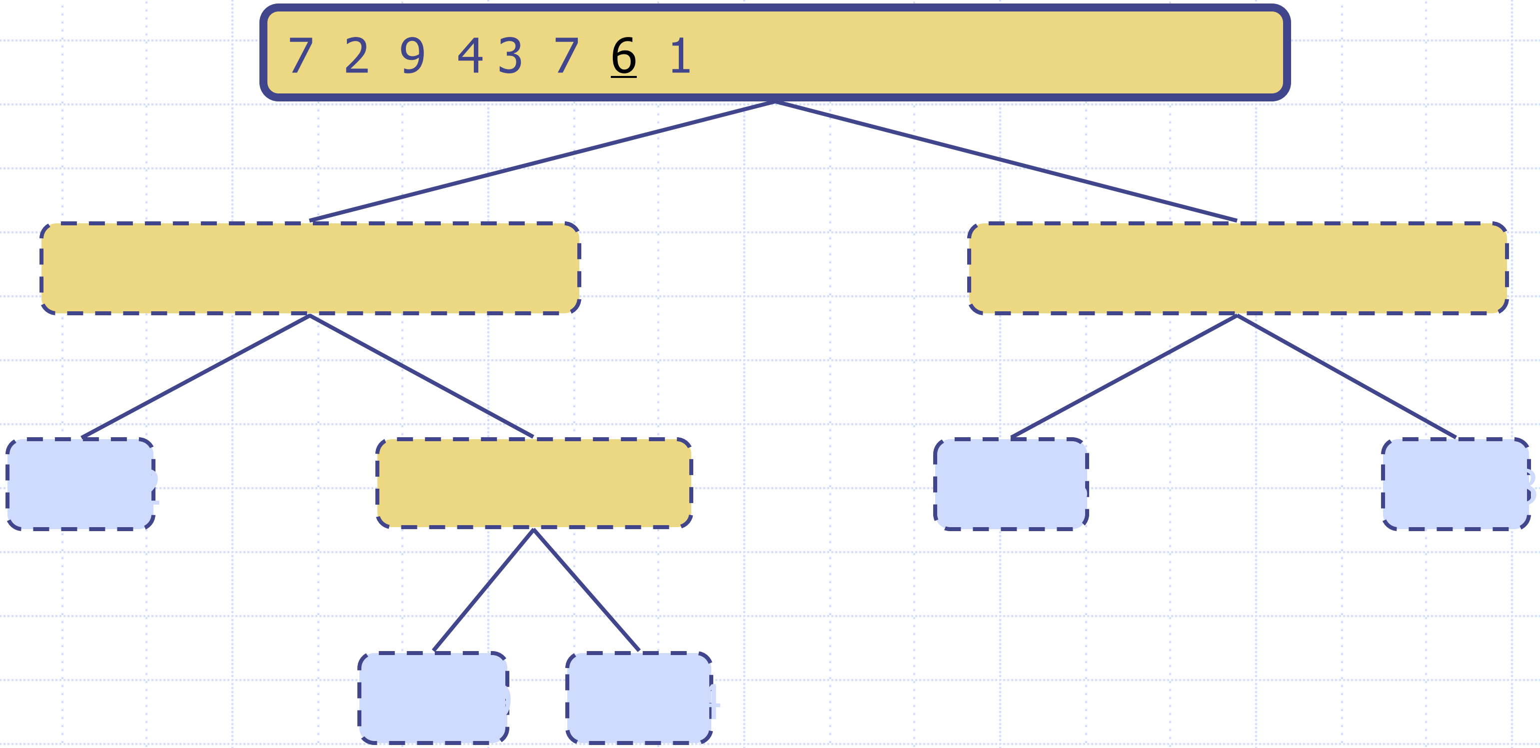
Quick-Sort Tree

- 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



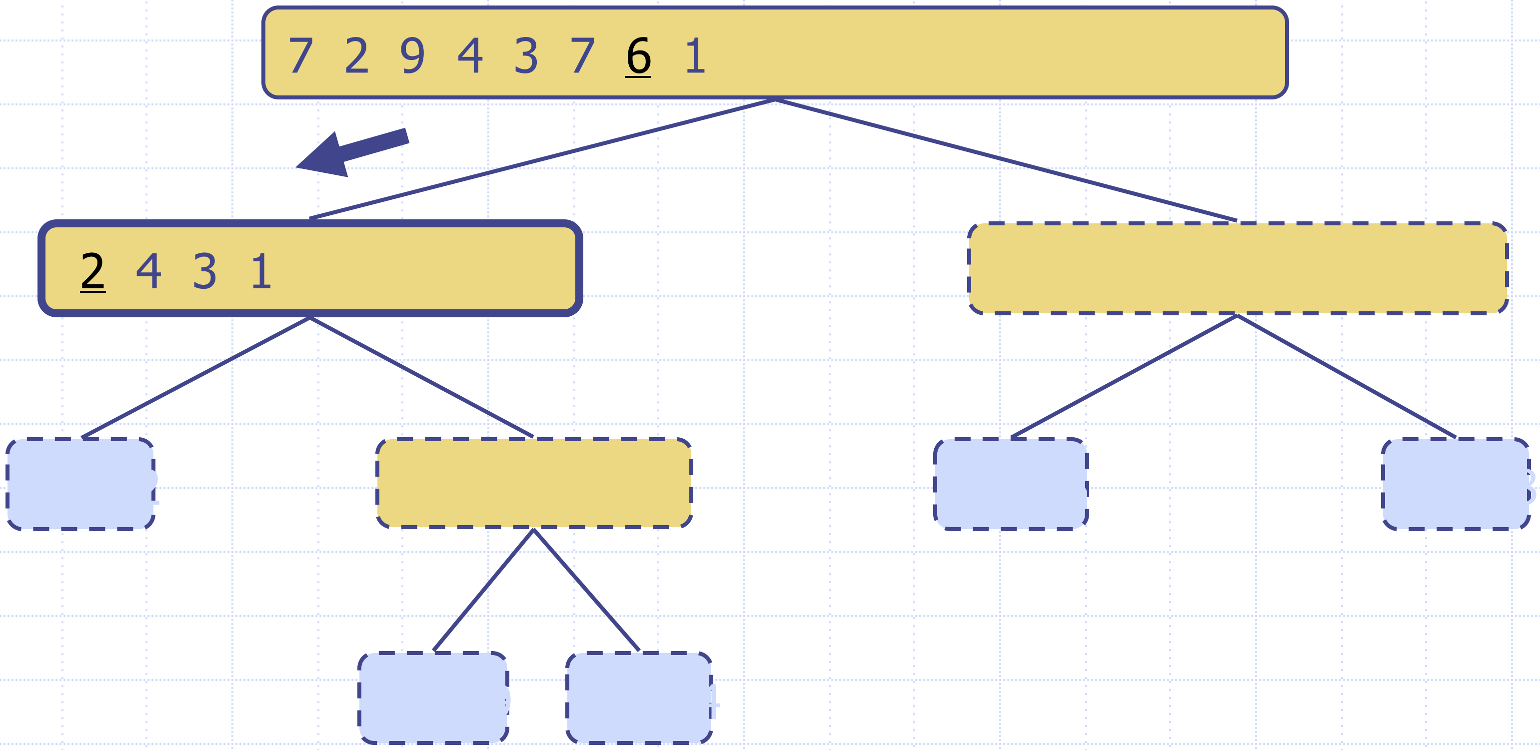
Execution Example

- Pivot selection



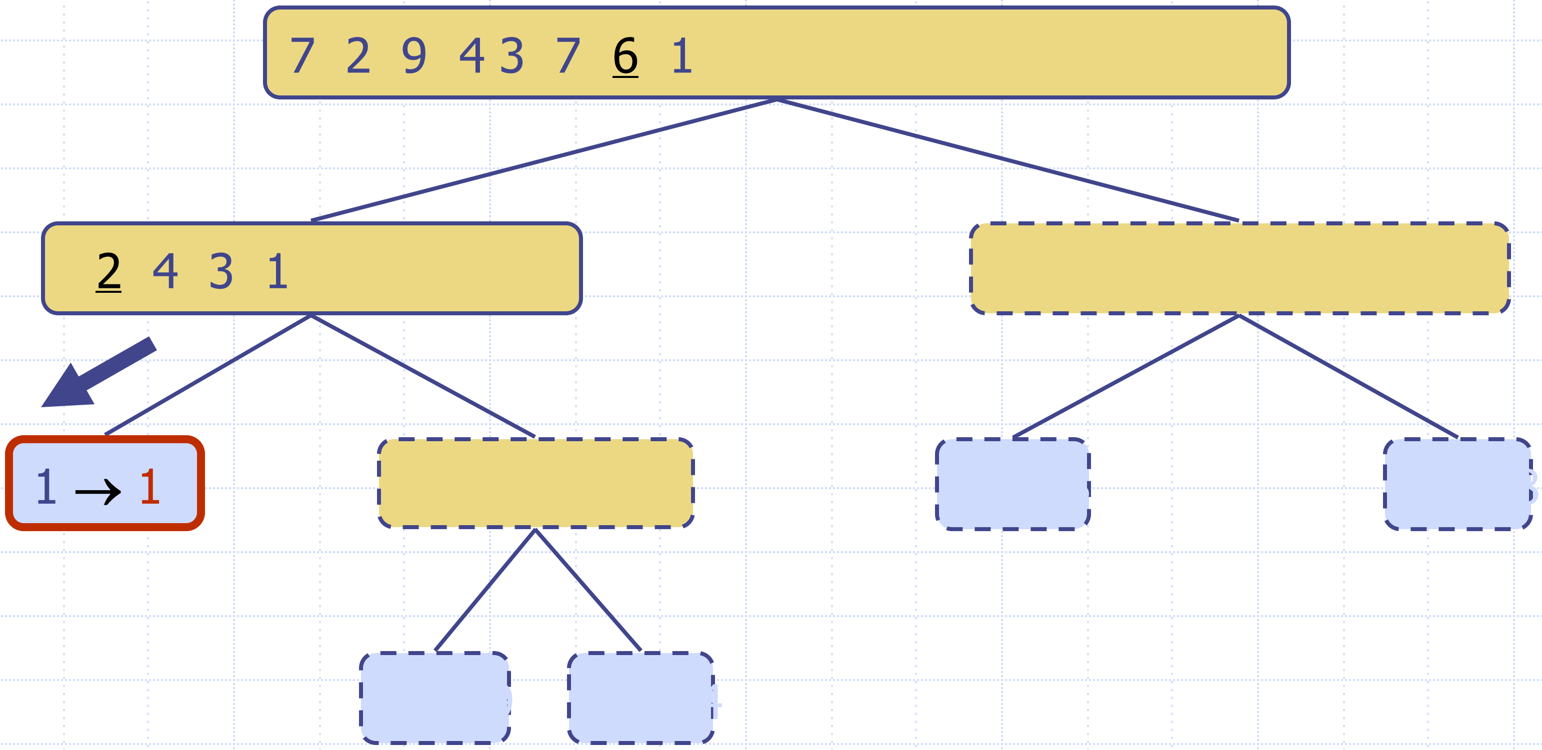
Execution Example (cont.)

- Partition, recursive call, pivot selection



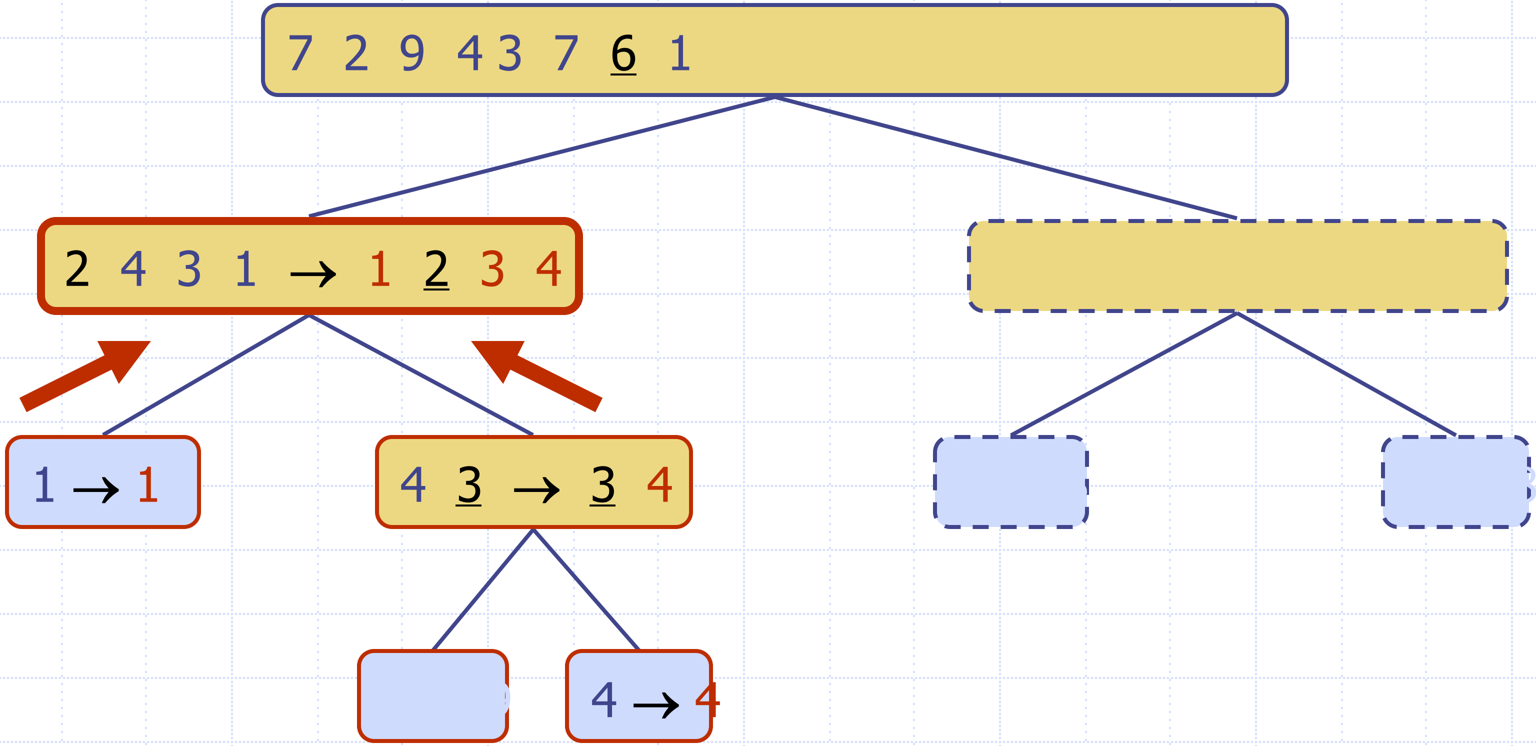
Execution Example (cont.)

- Partition, recursive call, base case



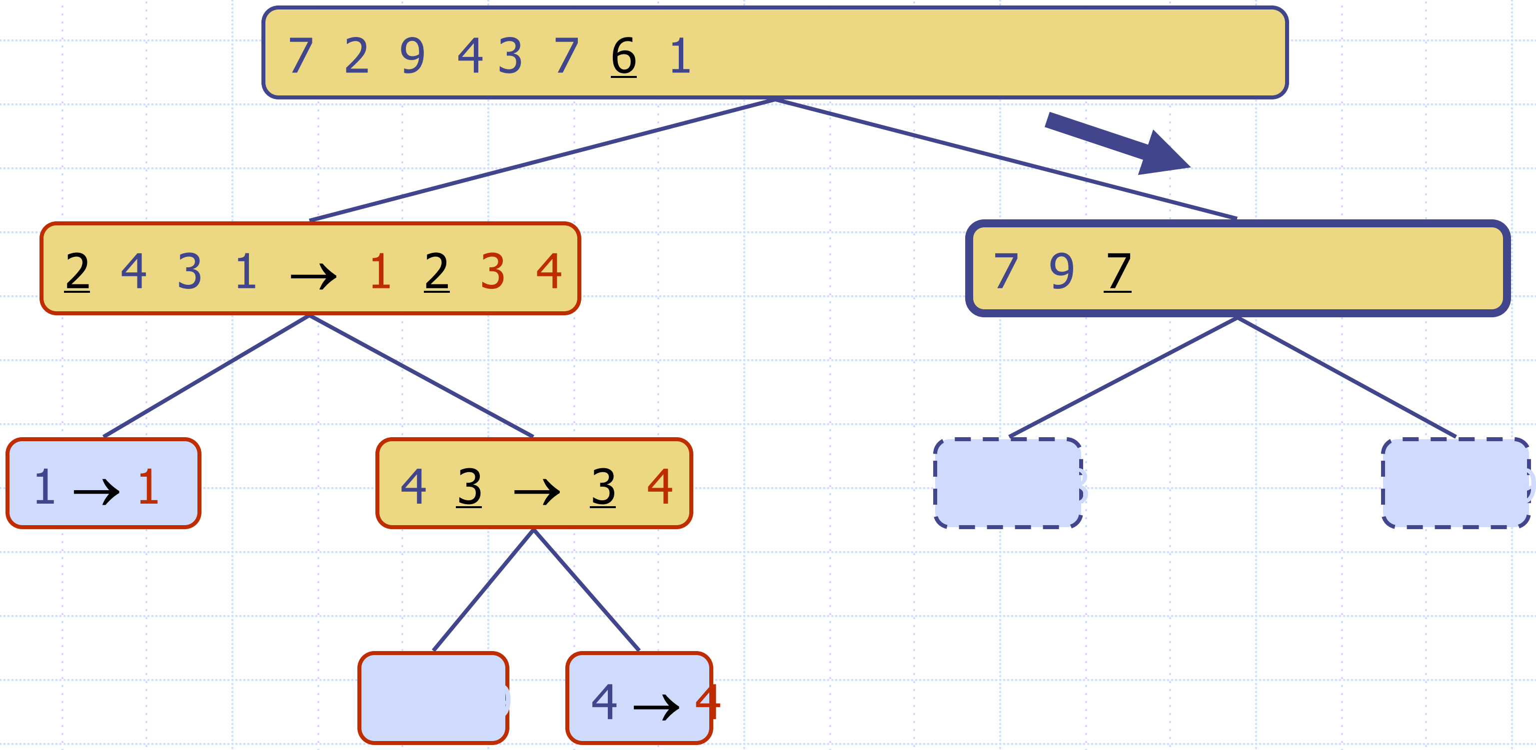
Execution Example (cont.)

- Recursive call, …, base case, join



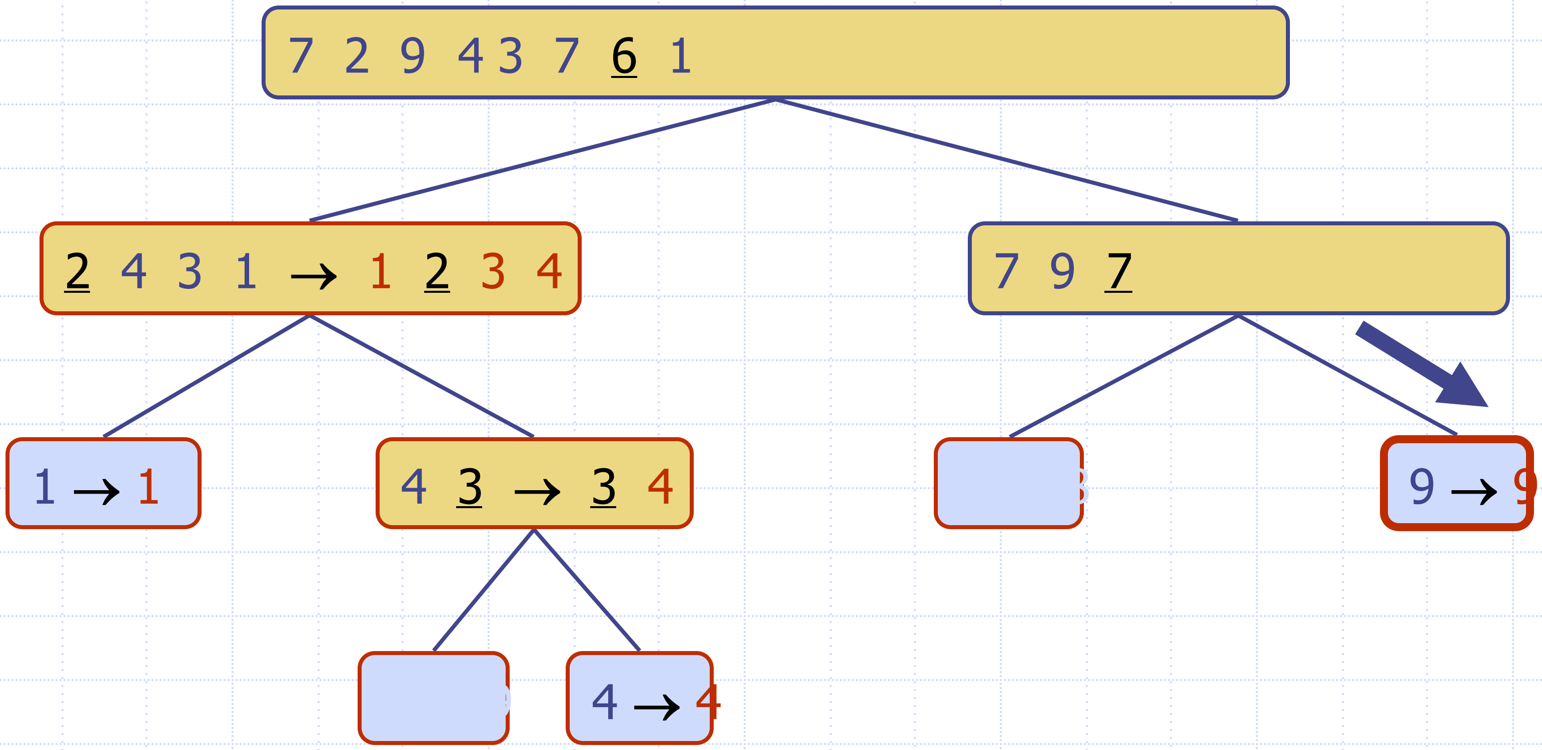
Execution Example (cont.)

- Recursive call, pivot selection



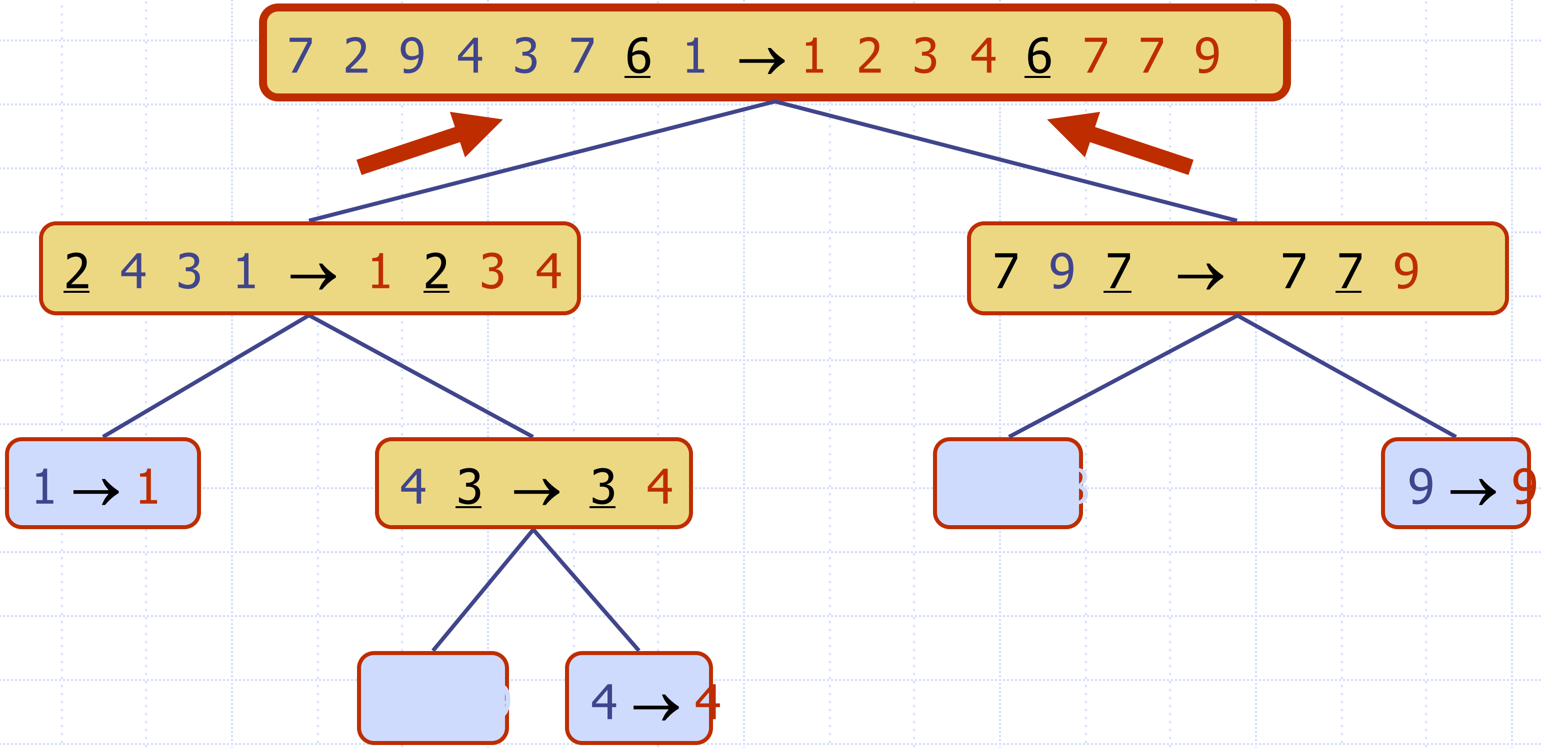
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join

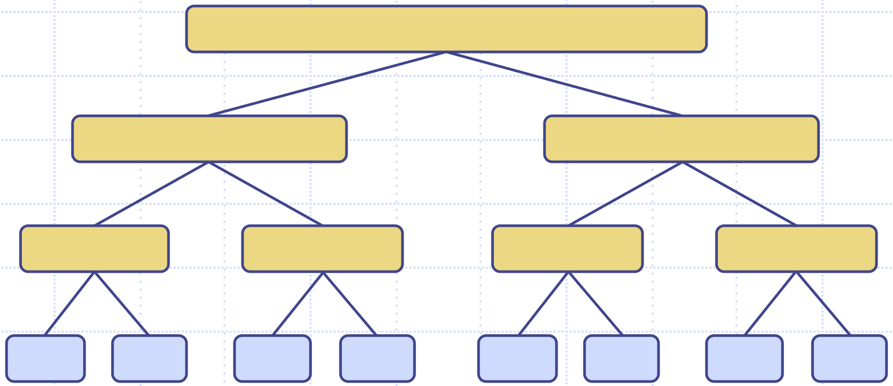


Best-case Running Time

- If we are lucky, Partition splits the array evenly

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T’s	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



Total time = $bn + bn \log n$

(last level plus all previous levels)

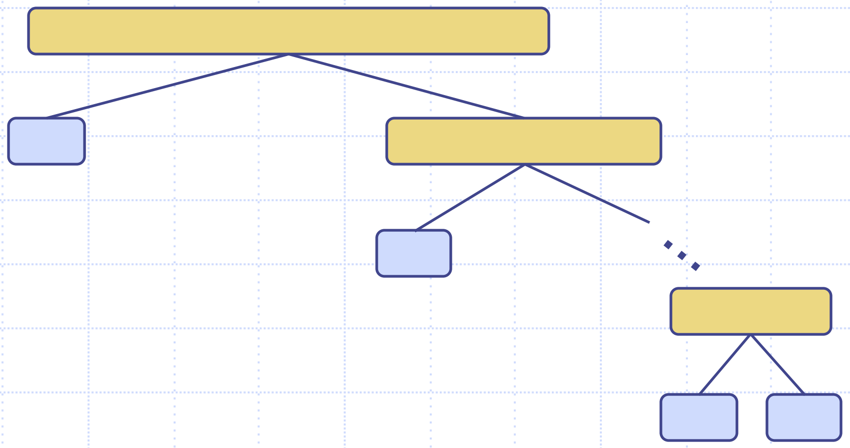
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element and the input sequence is in ascending or descending order
- One of L and G has size $n - 1$ and the other has size 1
- 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 Time Analysis

- Fix the input
  - expectation is over different randomly selected pivots
- Let T(n) be the expected number of comparisons needed to quicksort n numbers
- Probability of each split – 1/n
  - T(n) = T(j-1)+T(n-j)+n-1 with probability 1/n

$$T(n) = \frac{1}{n}\sum_{j=1}^{n}(T(j-1) + T(n-j) + n - 1)$$

$$= \frac{2}{n}\sum_{j=0}^{n-1} T(j) + n - 1$$

Expected Time Analysis (2)

- Since

$$T(n-1) = \frac{2}{n-1}\sum_{j=0}^{n-2} T(j) + n - 2$$

- we have

$$\frac{2}{n}\sum_{j=0}^{n-2} T(j) = \frac{n-1}{n}(T(n-1) - n + 2)$$

- substituting in the expression for T(n)

$$\begin{aligned} T(n) &= \frac{n-1}{n}(T(n-1) - n + 2) + \frac{2}{n}T(n-1) + n - 1 \\ &= \frac{n+1}{n}T(n-1) + \frac{2(n-1)}{n} \end{aligned}$$

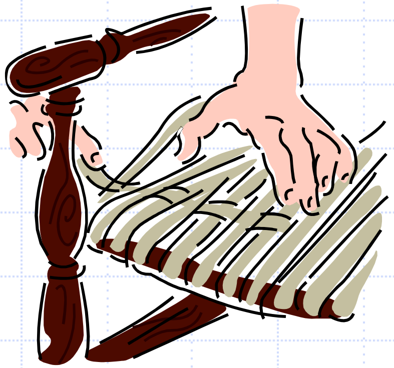
Expected Time Analysis (3)

$$
\begin{aligned}
T(n) &= \frac{n+1}{n}T(n-1) + \frac{2(n-1)}{n} \\
&< \frac{n+1}{n}T(n-1) + 2 \\
&< \frac{n+1}{n}\left(\frac{n}{n-1}T(n-2) + 2\right) + 2 \\
&= \frac{n+1}{n-1}T(n-2) + \frac{2(n+1)}{n} + 2 \\
&< \frac{n+1}{n-2}T(n-3) + 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1}\right) + 2 \\
&< \frac{n+1}{n-3}T(n-4) + 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2}\right) + 2 \\
&< (n+1)T(0) + 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right) + 2 \\
&= 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right) + 2
\end{aligned}
$$

Expected Time Analysis

$$
\begin{aligned}
T(n) &< 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right) + 2 \\
&= 2(n+1)\int_1^n \frac{dx}{x} + 2 \\
&= 2(n+1)\log n + 2 \\
&= O(n \log n)
\end{aligned}
$$

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 $l \geq r$
        return
    $i \leftarrow$ a random integer between l and r
    $x \leftarrow S.elemAtRank(i)$
    $(h, k) \leftarrow inPlacePartition(x)$
    $inPlaceQuickSort(S, l, h - 1)$
    $inPlaceQuickSort(S, k + 1, r)$

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 $\geq$ 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

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)

Radix Sort

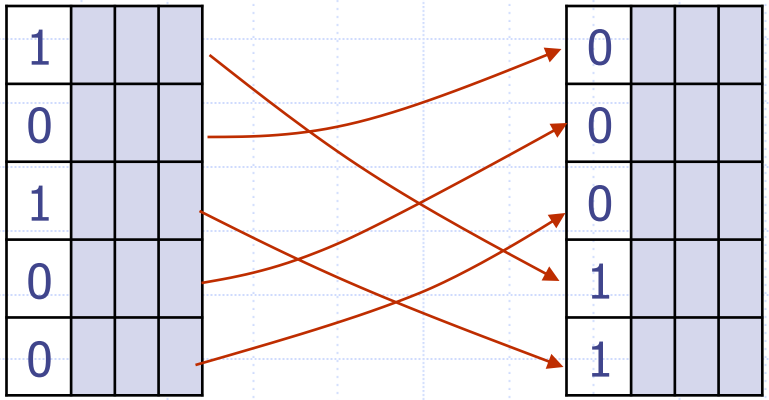
- Considers structure of the keys
- Assume keys are represented in base M number system (M=radix) i.e., if M=1, the keys are represented in binary format.

		8	4	2	1	weight
8	=	1	0	0	0	(b=4)
		3	2	1	0	bit #

- Sorting is performed by comparing bits in the same position
- Extension to keys that are alphanumeric strings

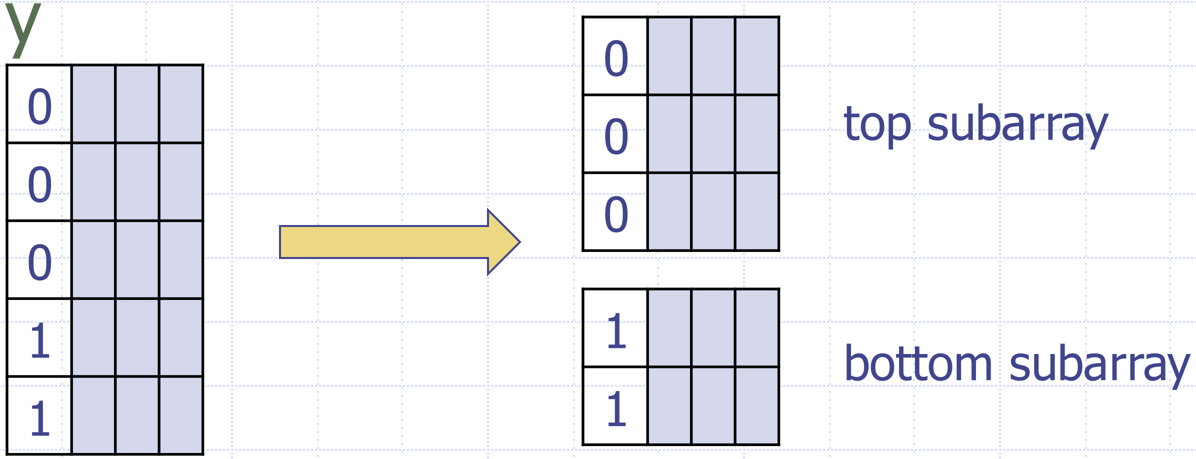
Radix Exchange Sort

- All the keys are represented with a fixed number of bits
- Examine bits from left to right
  - sort array with respect to leftmost bit



Radix Exchange Sort

- All the keys are represented with a fixed number of bits
- Examine bits from left to right
  - sort array with respect to leftmost bit
  - partition array

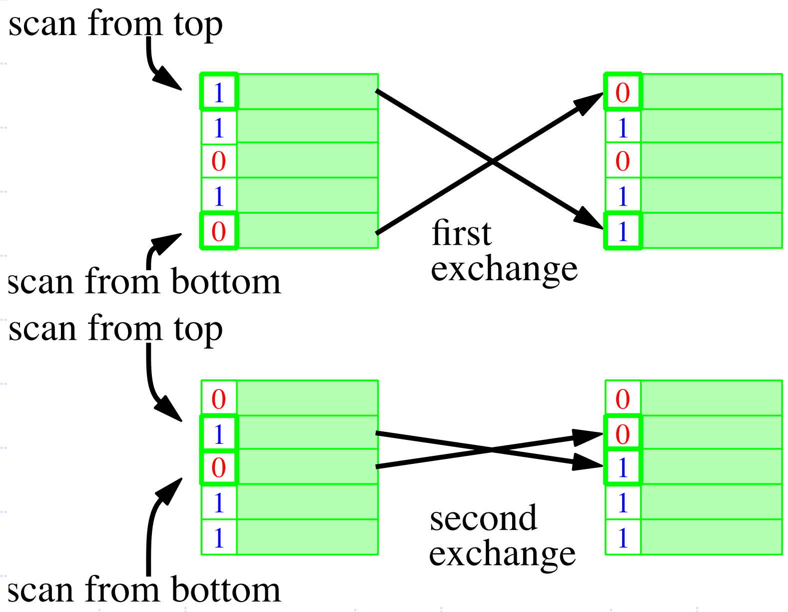


Radix Exchange Sort

- All the keys are represented with a fixed number of bits
- Examine bits from left to right
  - sort array with respect to leftmost bit
  - partition array
  - recursion
    - recursively sort the top subarray, ignoring the leftmost bit
    - recursively sort the bottom subarray, ignoring the leftmost bit
  - Complexity – n numbers of b bits – O(b n)

Radix Exchange Sort

- Partition
  - repeat
    - scan top-down to find key starting with 1
    - scan bottom-up to find key starting with 0
    - swap the keys
  - scan till indices cross.
- Complexity is O(n)



Straight Radix Sort

- Examines bit from right to left
  - for k:=0 to b-1
    - sort the array in a stable way
    - looking only at bit k

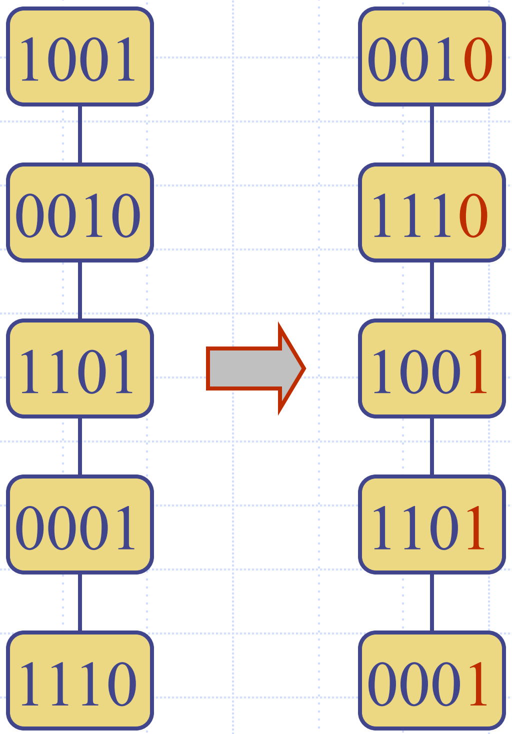
Example

- Sorting a sequence of 4-bit integers

1001	0010	1001	1001	0001
0010	1110	1101	0001	0010
1101	1001	0001	0010	1001
0001	1101	0010	1101	1101
1110	0001	1110	1110	1110

Sort in a Stable Way

- In a stable sort, the initial relative order of equal keys is unchanged
- For example, observe the first step of the sort.
- Note that the relative order of those keys ending with 0 is unchanged, and the same is true for elements ending in 1.



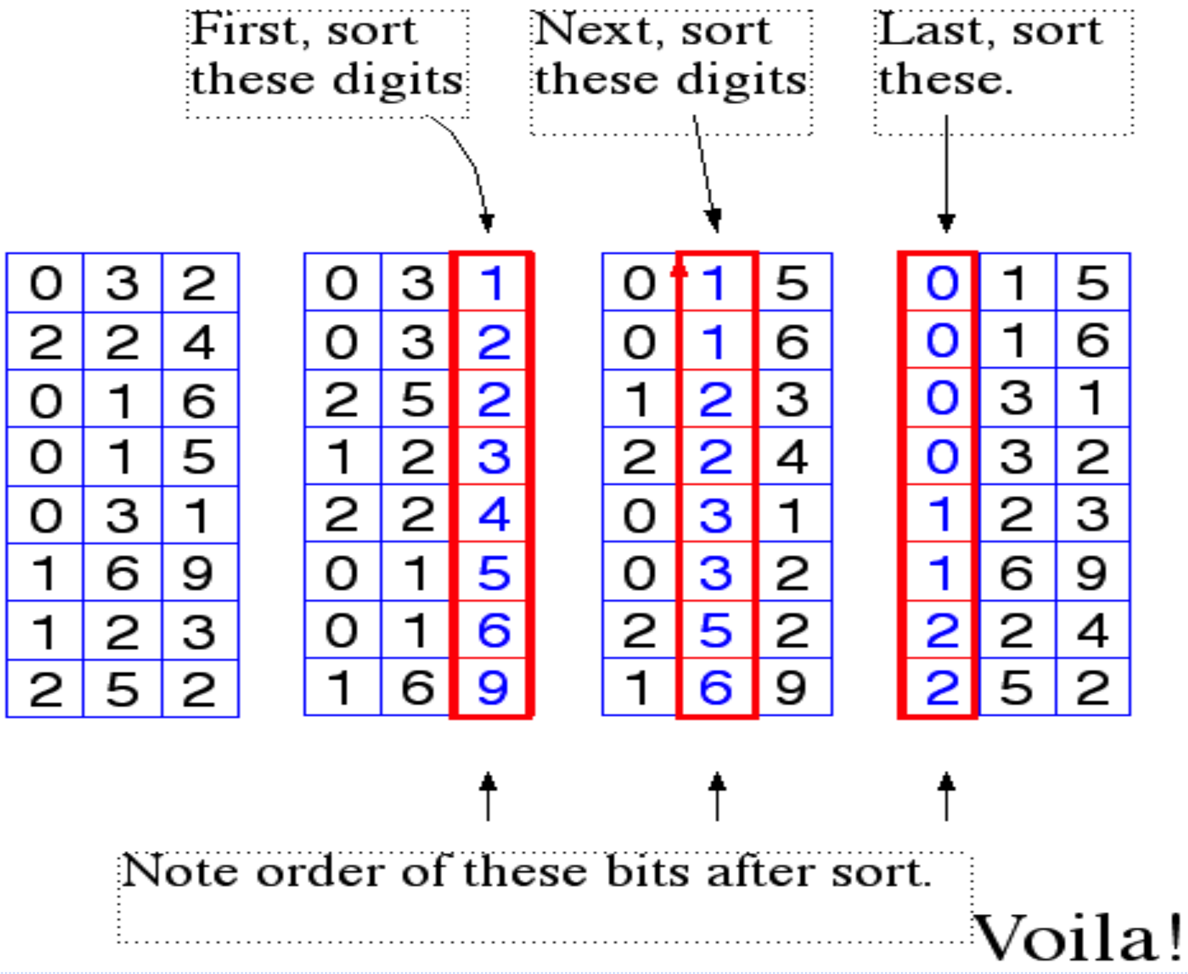
Correctness

- We show that any two keys are in the correct relative order at the end of the algorithm
- Given two keys, let *k* be the leftmost bit-position where they differ

0	1	0	1	1
0	1	1	0	1
		k		

- At step *k* the two keys are put in the correct relative order
- Because of *stability*, the successive steps do not change the relative order of the two keys

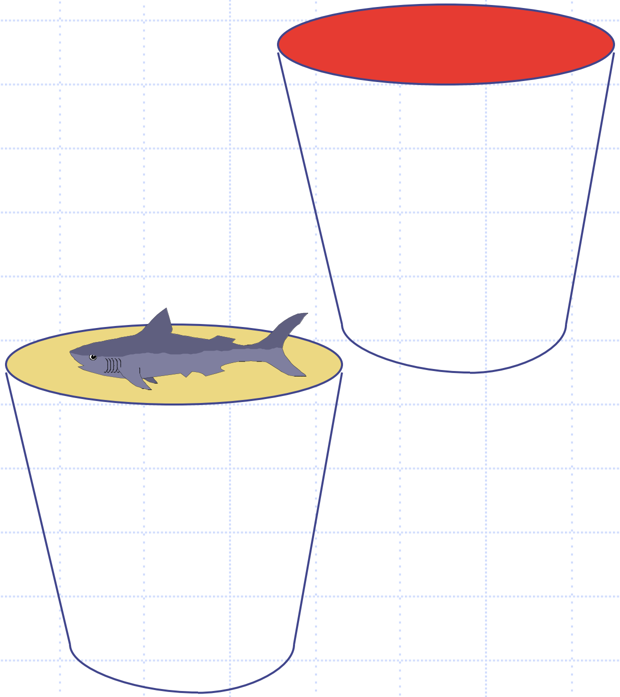
Example – Decimal Numbers



Straight Radix Sort Time Complexity

- for k = 0 to b - 1
  - sort the array in a stable way, looking only at bit k
- Suppose we can perform the stable sort above in O(n) time. The total time complexity would be O(bn)
- We can perform a stable sort based on the keys' k^{th} digit in O(n) time.
  - how?

Bucket Sort



BASICS:

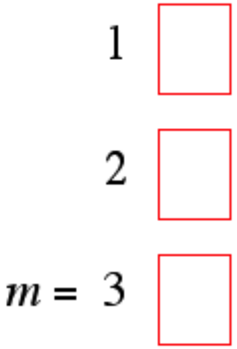
- n numbers
- Each number $\in \{1, 2, 3, \ldots m\}$
- Stable
- Time: $O(n + m)$

For example, $m = 3$ and our array is:

2	1	3	1	2

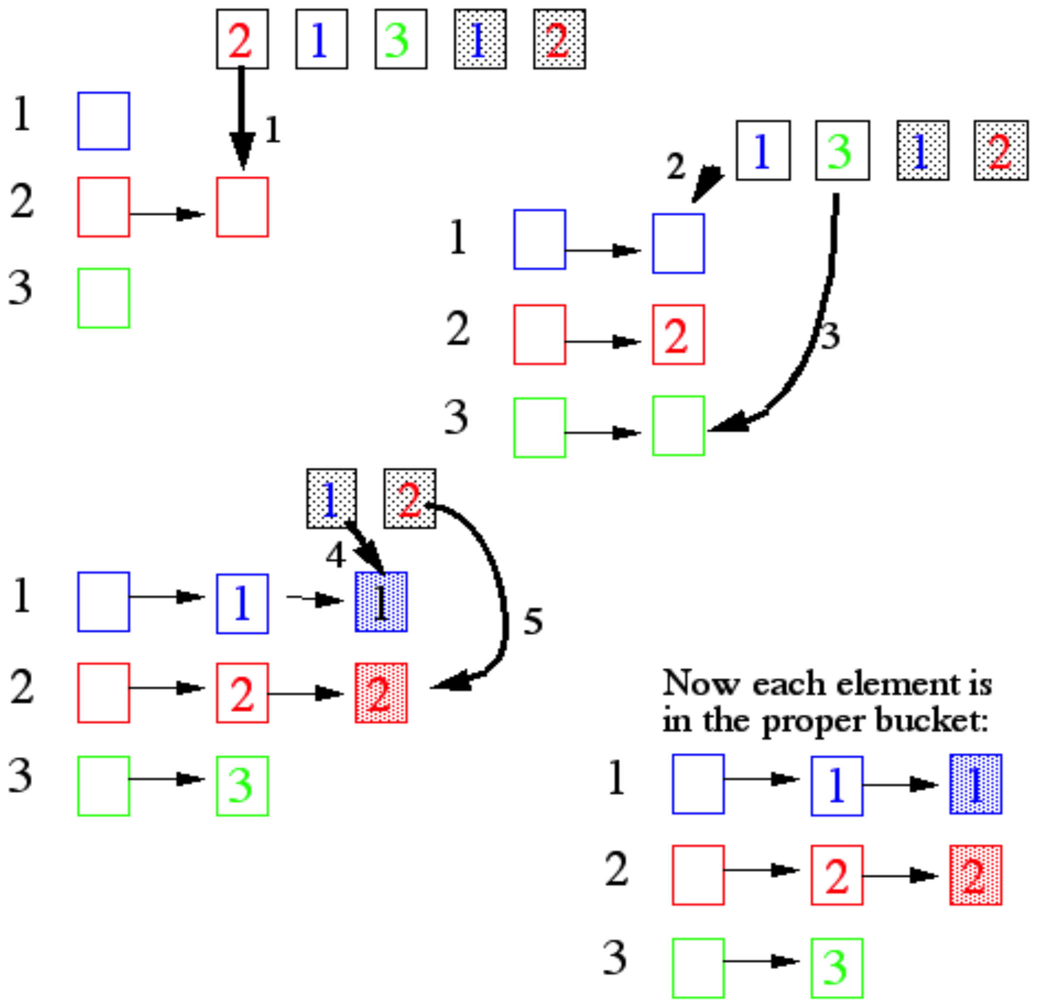
(note that there are two “2”s and two “1”s)

First, we create M “buckets”



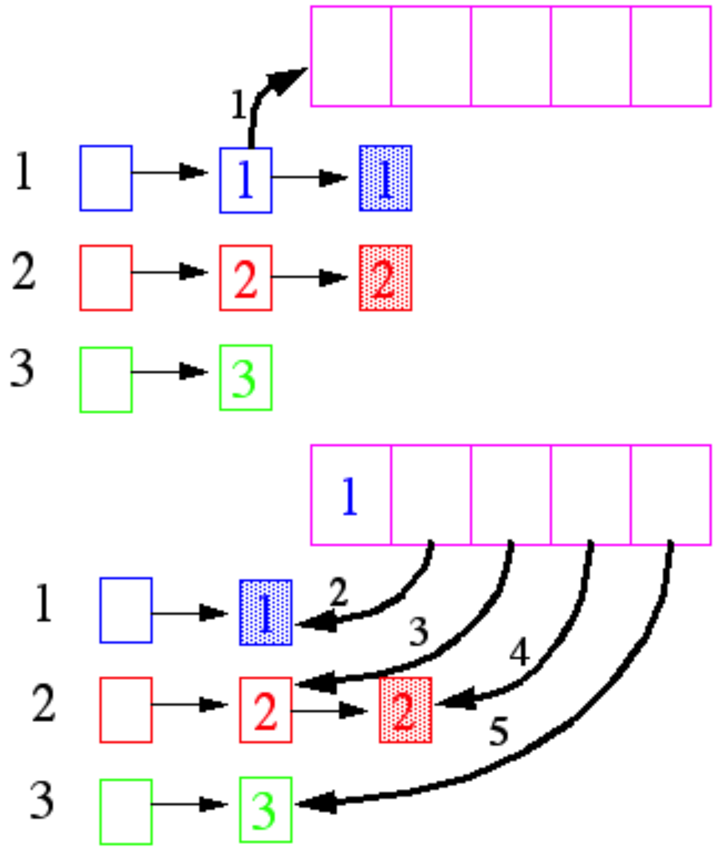
Bucket Sort

Each element of the array is put in one of the m "buckets"

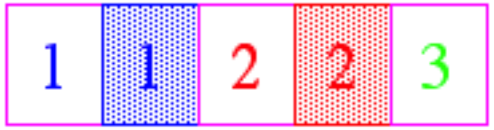


Bucket Sort

Now, pull the elements from the buckets into the array



At last, the sorted array (sorted in a stable way):



In-Place Sorting

- A sorting algorithm is said to be ***in-place*** if
  - it uses no auxiliary data structures (however, O(1) auxiliary variables are allowed)
  - it updates the input sequence only by means of operations replaceElement and swapElements
- Which sorting algorithms seen so far can be made to work in place?

selection-sort	
insertion-sort	
heap-sort	
merge-sort	
quick-sort	
radix-sort	
bucket-sort	