Lecture 10: Sorting Algorithms

(Chapter 11, Sections 11.1, 11.2, 11.3 from the book)

Agenda

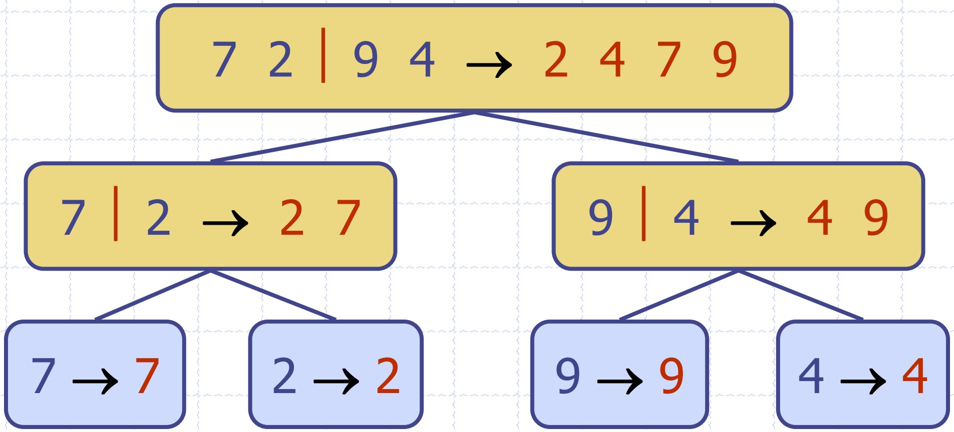
- Sorting
  - Sorting Problem
  - Merge Sort
  - Quick Sort
  - Bucket Sort
- Sorting Lower Bound

Sorting Problem

- Sort a given sequence of data items according to their keys.
  - Examples

  1) Input: (15, …), (26, …), (11, …), (23, …), (7, …), (31, …), (30, …)
     Output: (7, …), (11, …), (15, …), (23, …), (26, …), (30, …), (31, …)
  2) Input: (“go”, …), (“did”, …), (“me”, …), (“bet”, …), (“kit”, …)
     Output: (“bet”, …), (“did”, …), (“go”, …), (“kit”, …), (“me”, …)
- Sorting is a fundamental application for computers.
- Sorting is perhaps the most intensively studied and important operation in computer science.
- An initial sort of the data can significantly enhance the performance of an algorithm.

Merge Sort



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

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: If S has zero or one element, return S immediately. Otherwise, remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S; S_1 contains the first $\lceil n/2 \rceil$ elements of S and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a sorted sequence S
- The base case for the recursion are subproblems of size 0 or 1

$\lceil n/2 \rceil$ – ceiling of *x* – the smallest integer *m*, such that *x*<=*m*, e.g: $\lceil 5/2 \rceil = 3$

$\lfloor n/2 \rfloor$ – floor of *x* – the largest integer *k*, such that *k*<=*x*, e.g: $\lfloor 5/2 \rfloor = 2$

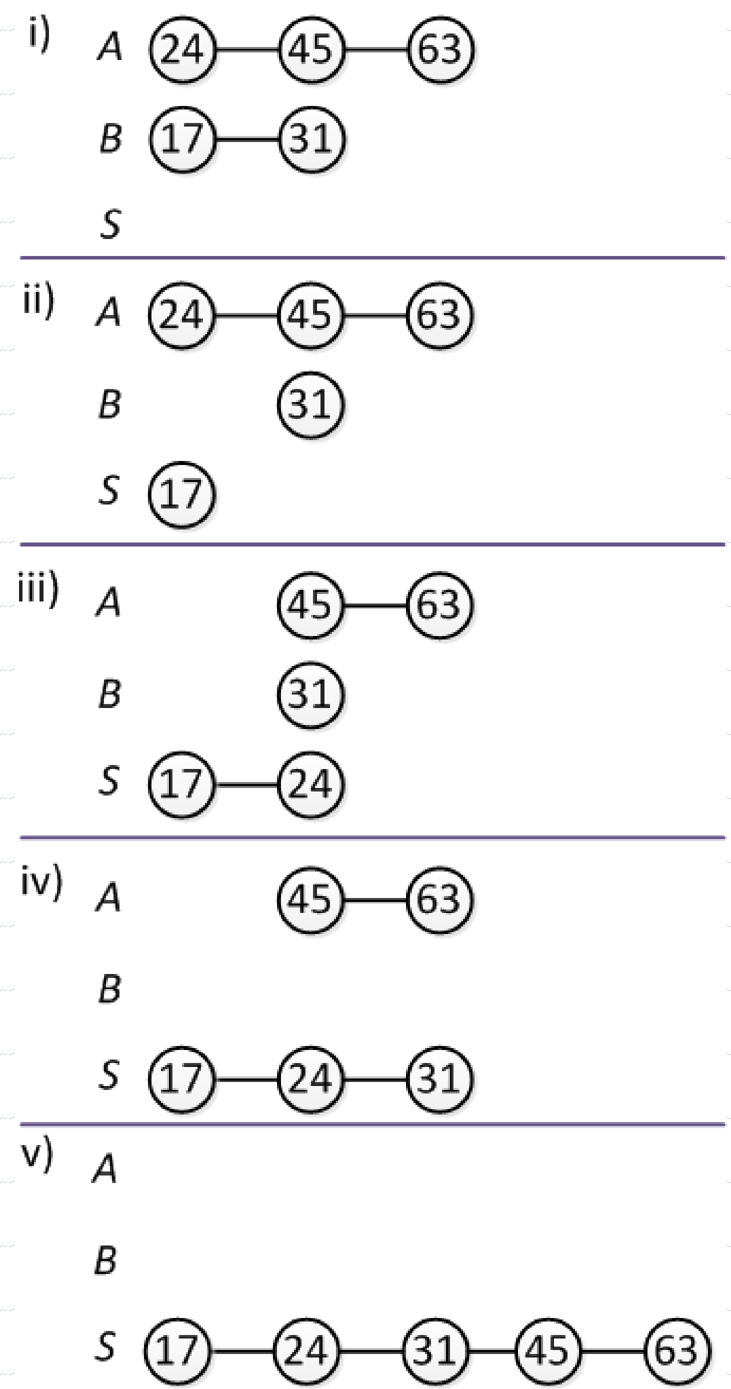
Merge-sort

Algorithm ***mergeSort*(*S, C*)**
Input sequence S with n elements, comparator C
Output sequence S sorted
according to C
if $S.size() > 1$
$(S_1, S_2) \leftarrow partition(S, n/2)$
$mergeSort(S_1, C)$
$mergeSort(S_2, C)$
$S \leftarrow merge(S_1, S_2)$

Merging Two Sorted **List**-based Sequences

```
Algorithm merge(A, B, S)
    Input sorted sequences A and B and an empty
sequence S implemented as linked list
    Output sorted sequence S = A ∪ B

    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        if A.first().element() <= B.first().element()
            {move the first element of A at the end of S}
            S.addLast(A.remove(A.first()))
        else
            {move the first element of B at the end of S}
            S.addLast(B.remove(B.first()))
    {move the remaining elements of A to S}
    while ¬A.isEmpty()
        S.addLast(A.remove(A.first()))
    {move the remaining elements of B to S}
    while ¬B.isEmpty()
        S.addLast(B.remove(B.first()))
    return S
```



Merging Two Sorted **Array**-based Sequences

Algorithm *merge*(*A*, *B*, *S*)

Input sorted sequences ***A*** and ***B*** and an empty sequence ***S***, all of which are implemented as arrays

Output sorted sequence $S = A \cup B$

$i \leftarrow j \leftarrow 0$

while i<*A.size*() $\land$ j<*B.size*() **do**

  if *A.get*(i) <= *B.get*(j)

    {copy ith element of ***A*** to the end of ***S***}

    S.addLast(*A.get*(i))

    $i \leftarrow i+1$

  else

    S.addLast(*B.get*(j))

    {copy jth element of ***B*** to the end of ***S***}

    $j \leftarrow j+1$

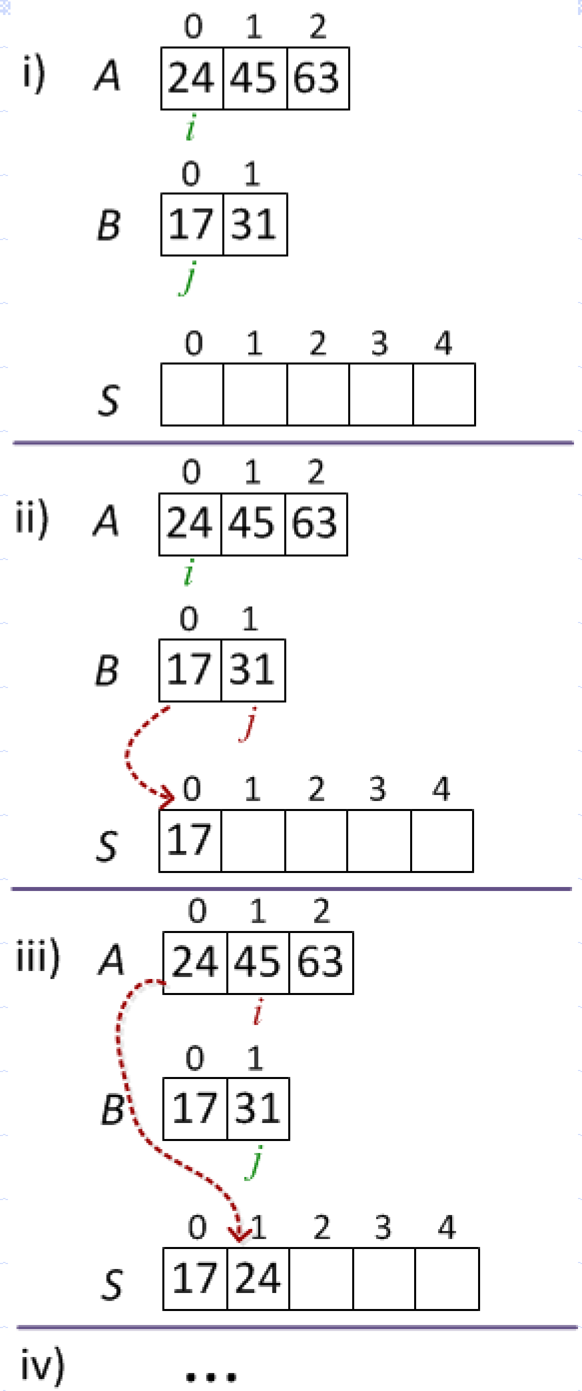
{move the remaining elements of ***A*** to ***S***}

while i<*A.size*() **do** *S.addLast*(*A.get*(i)); $i \leftarrow i+1$

{move the remaining elements of ***B*** to ***S***}

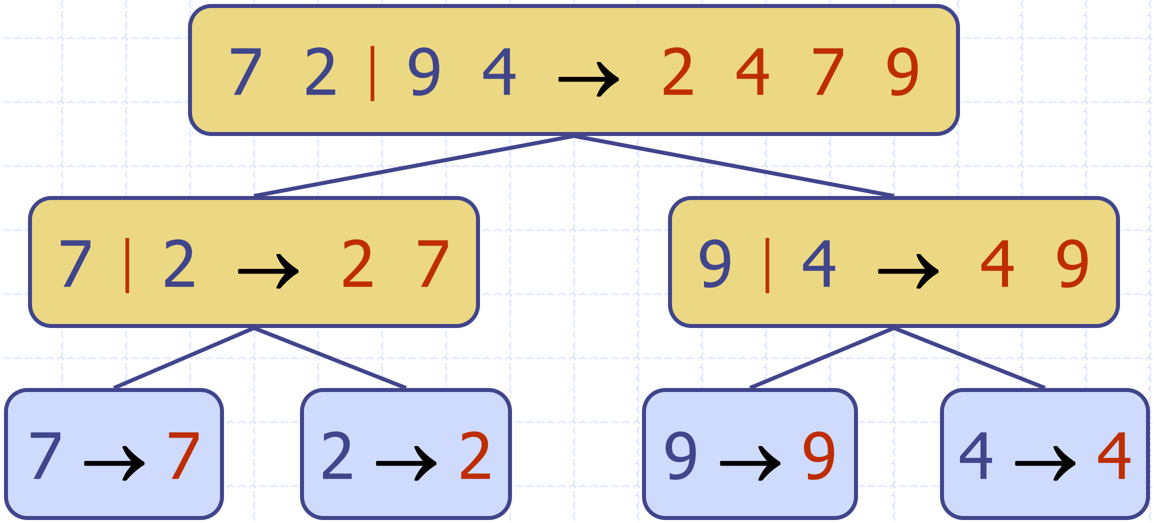
while j<*B.size*() **do** *S.addLast*(*B.get*(j)); $j \leftarrow j+1$

return *S*



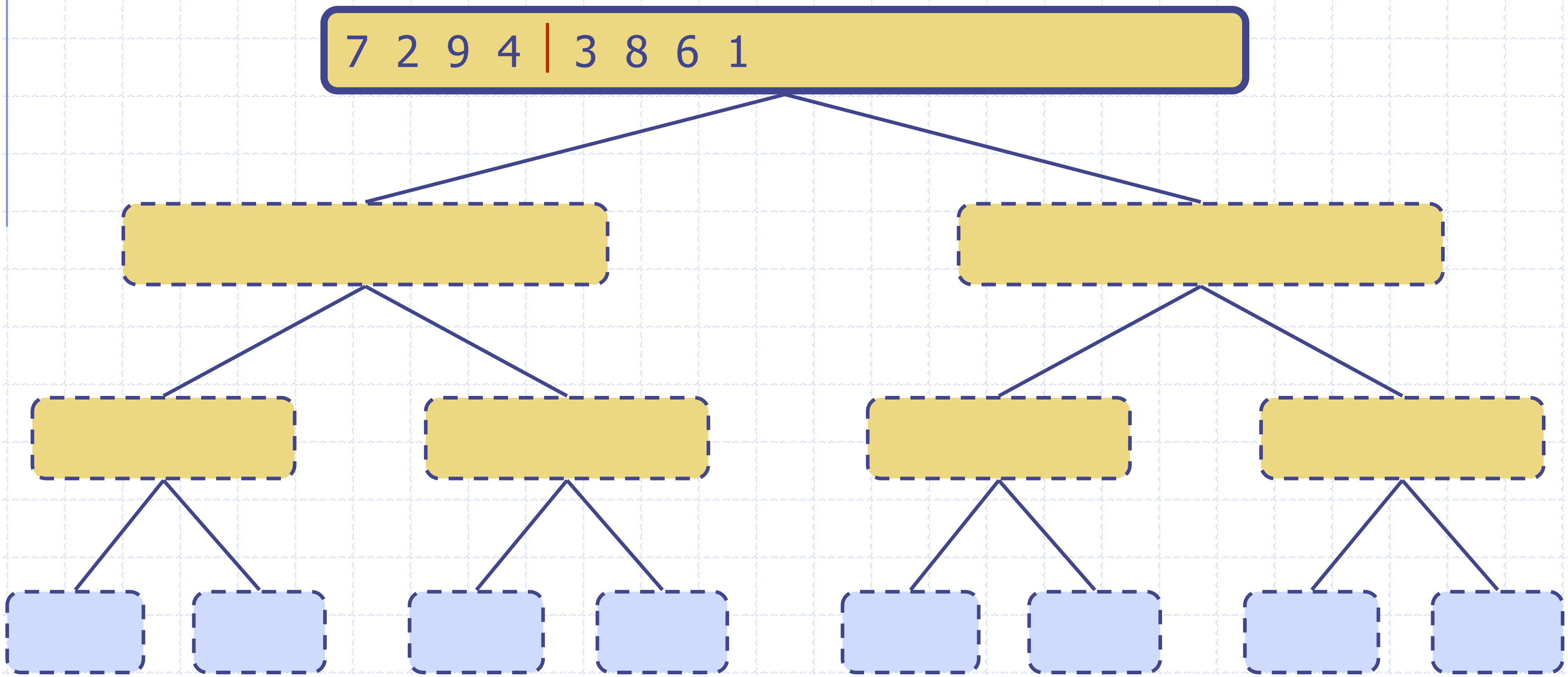
Merge-Sort Tree

- 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
- Sequence: (7,2,9,4)



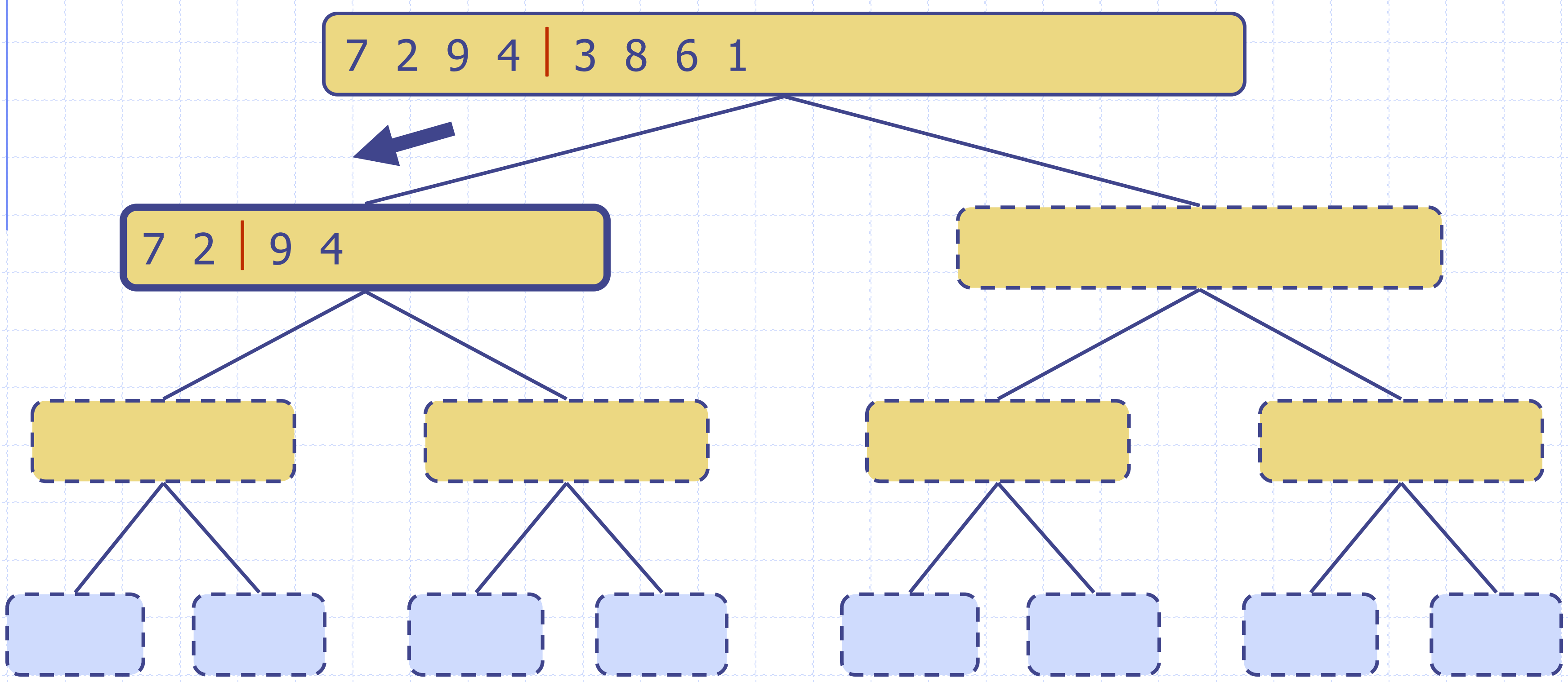
Execution Example (7,2,9,4,3,8,6,1)

- Partition



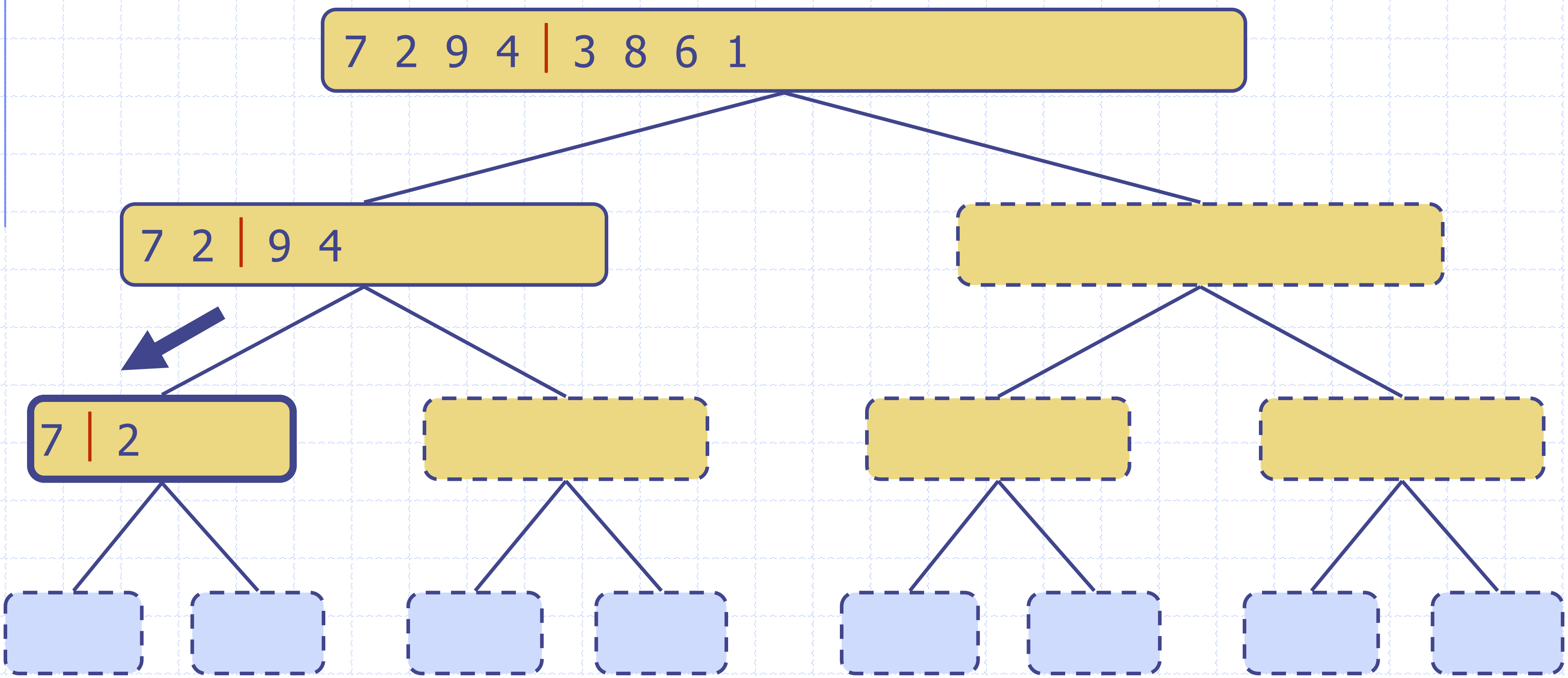
Execution Example (cont.)

- Recursive call, partition



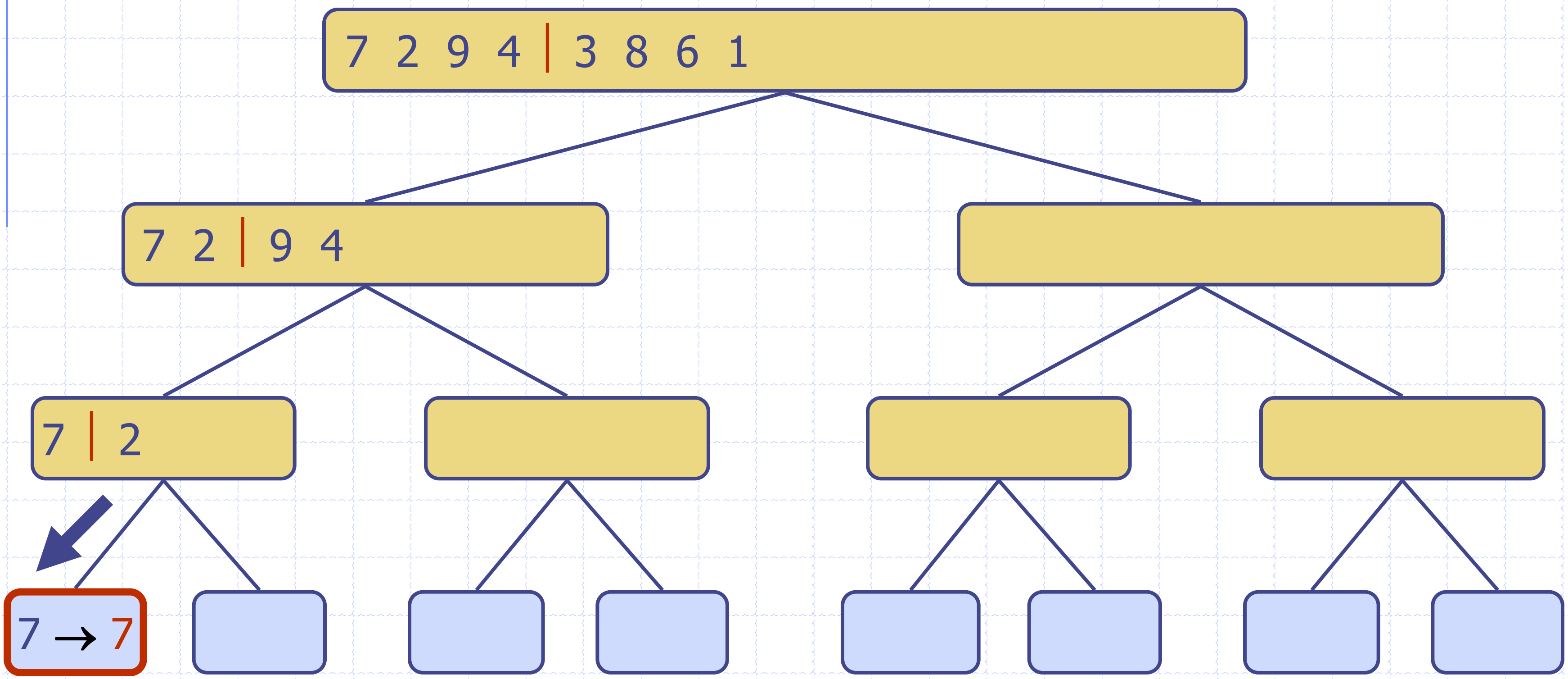
Execution Example (cont.)

- Recursive call, partition



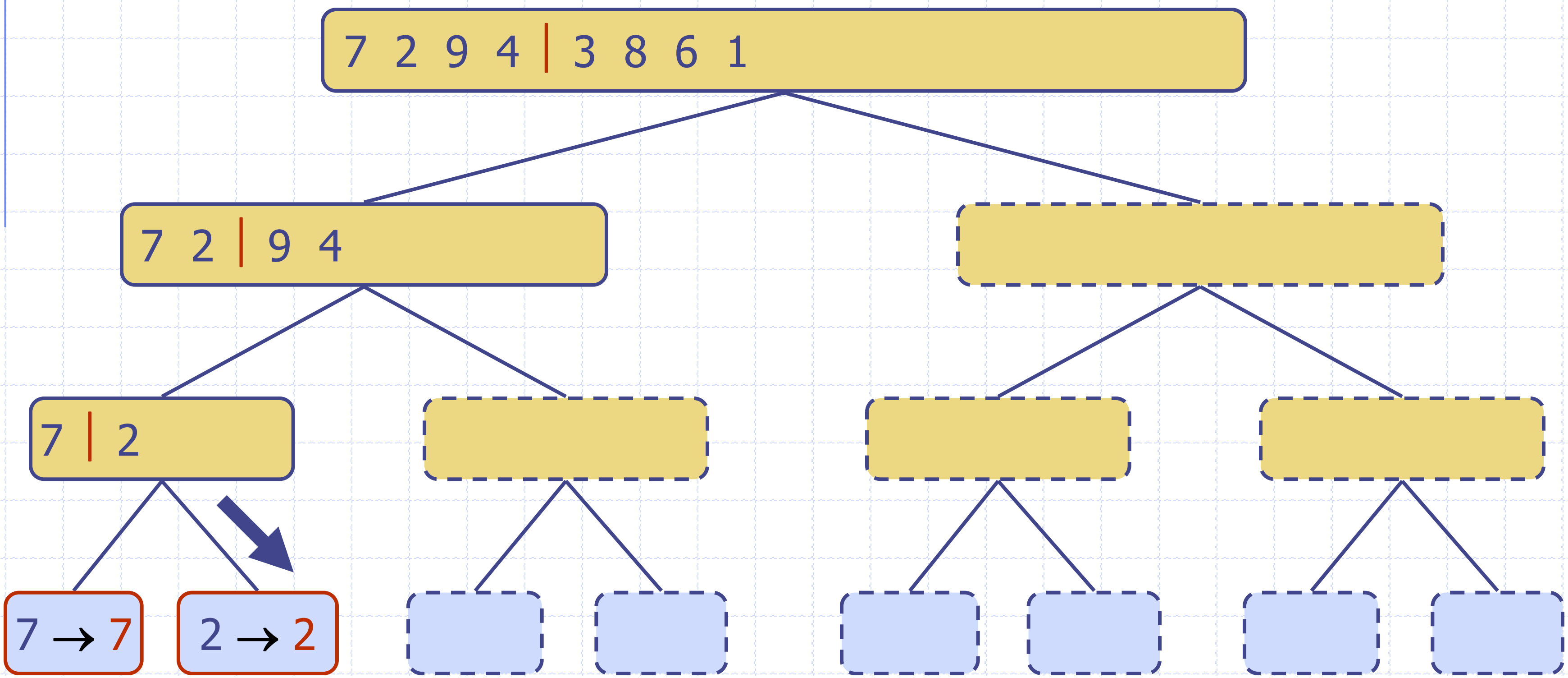
Execution Example (cont.)

- Recursive call, base case



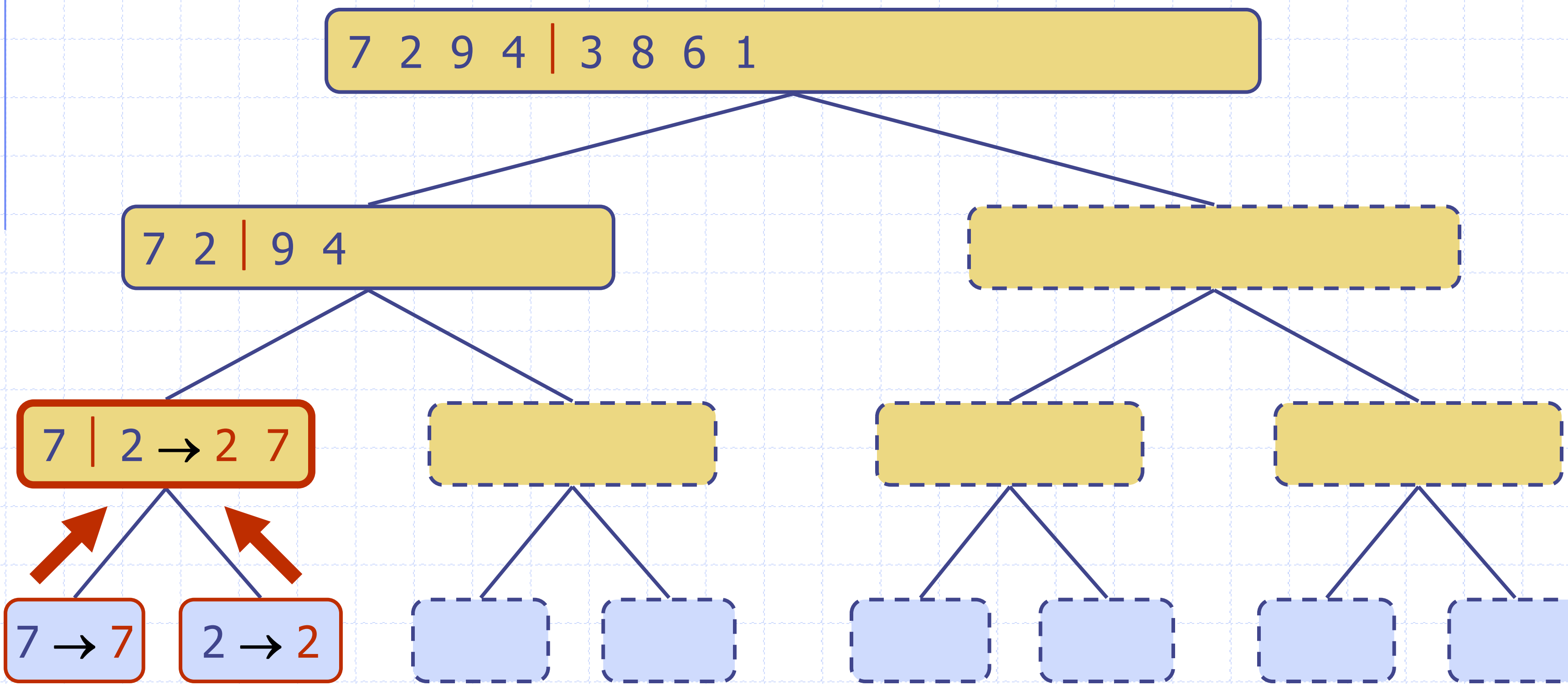
Execution Example (cont.)

- Recursive call, base case



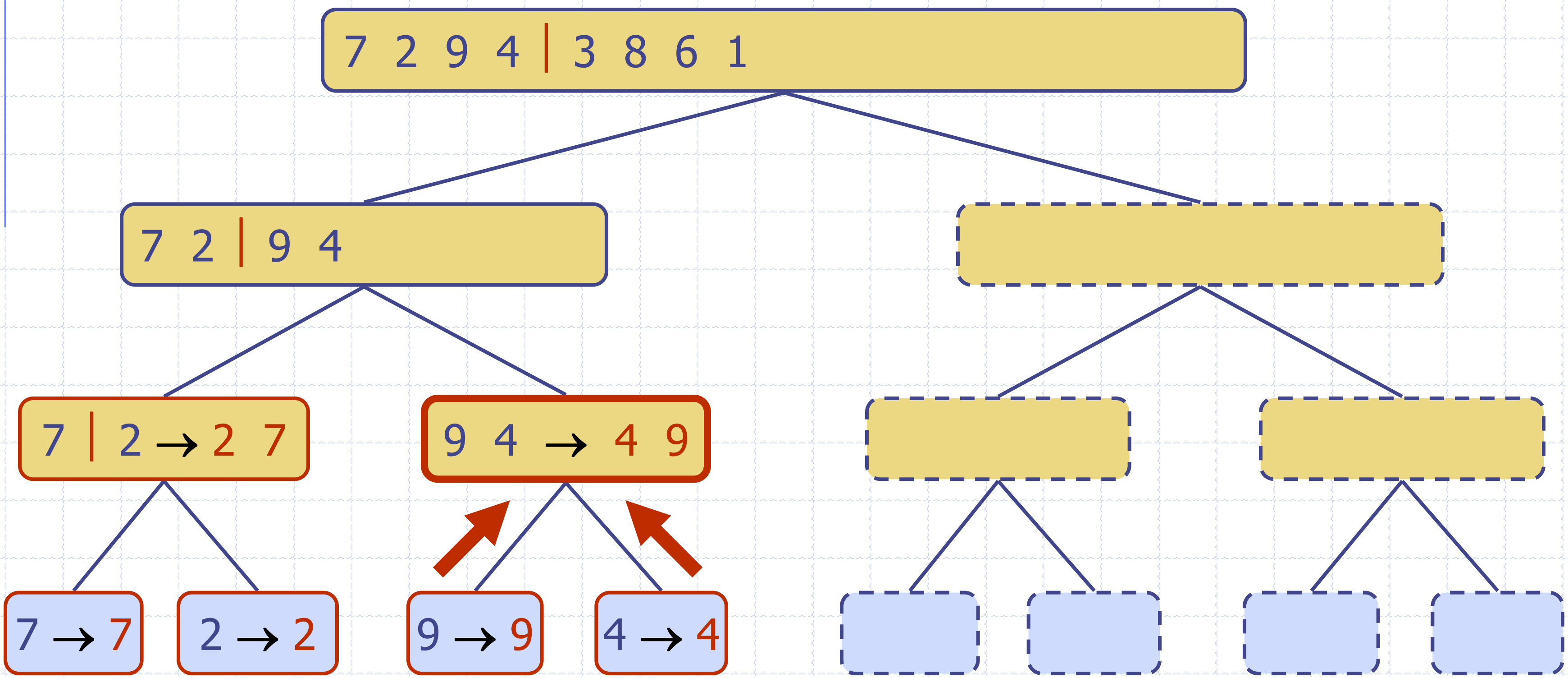
Execution Example (cont.)

- Merge



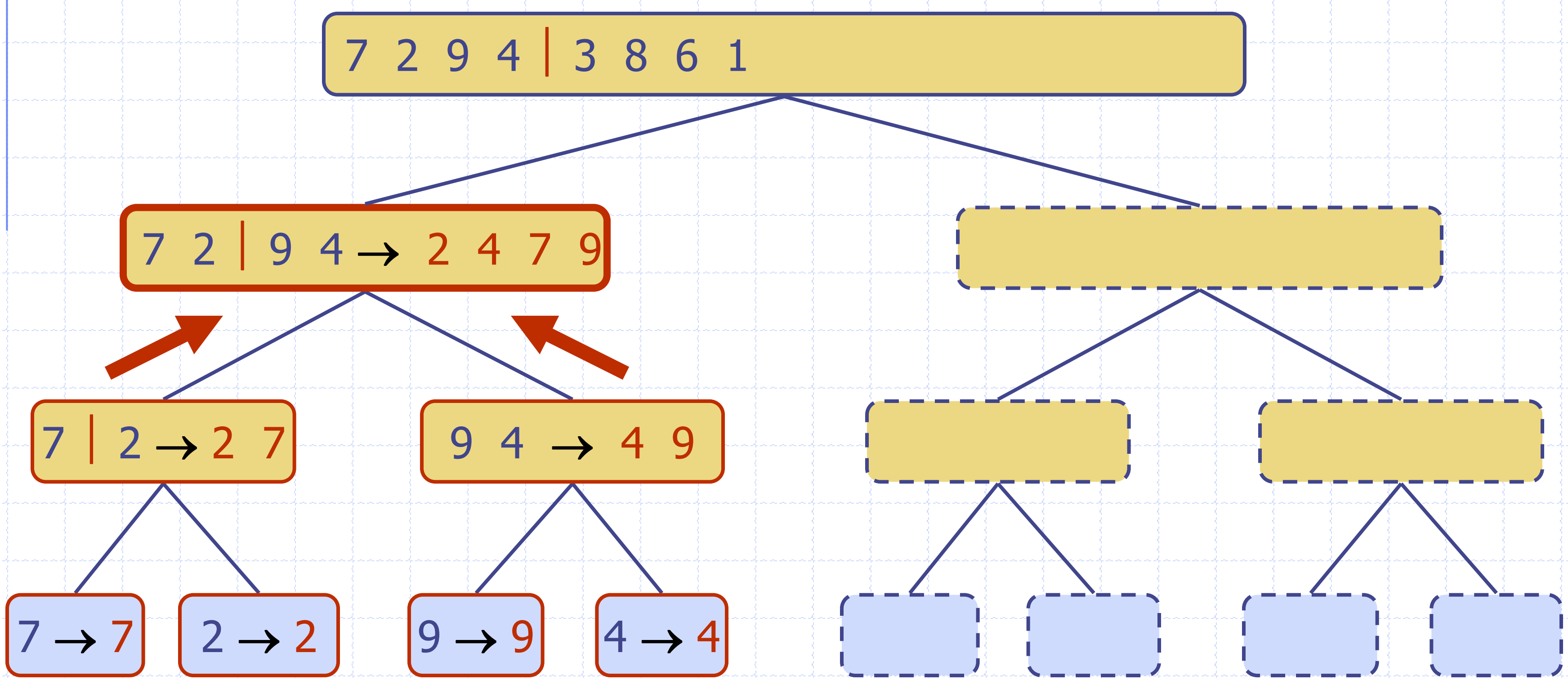
Execution Example (cont.)

- Recursive call, …, base case, merge



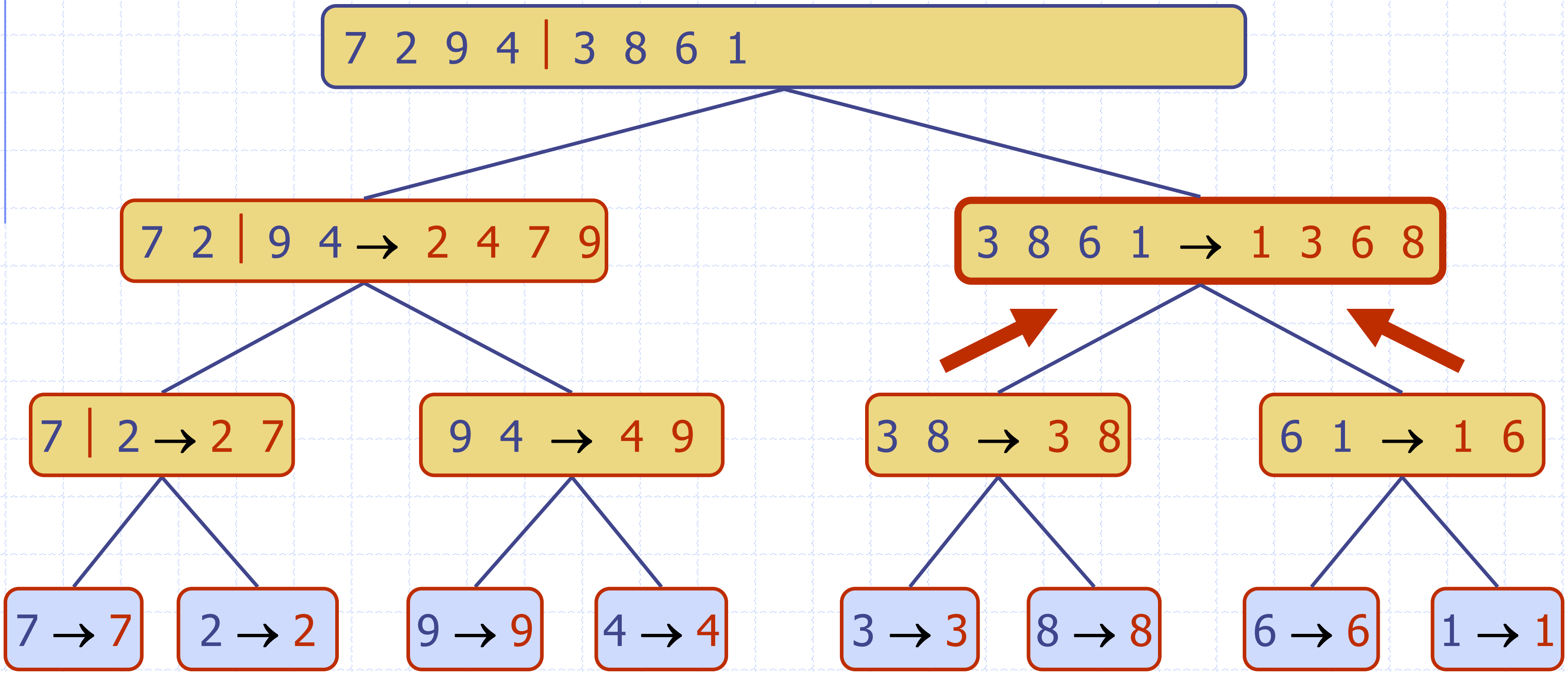
Execution Example (cont.)

- Merge



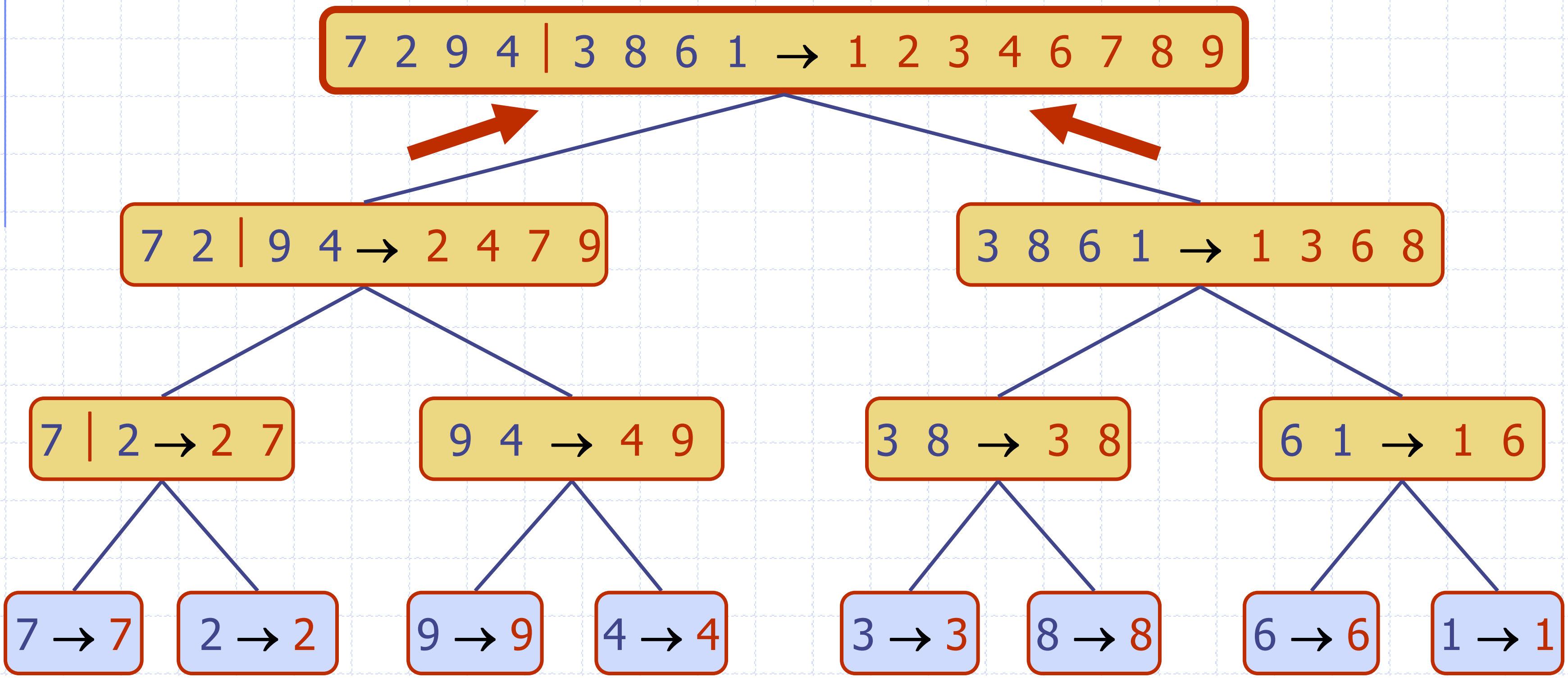
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

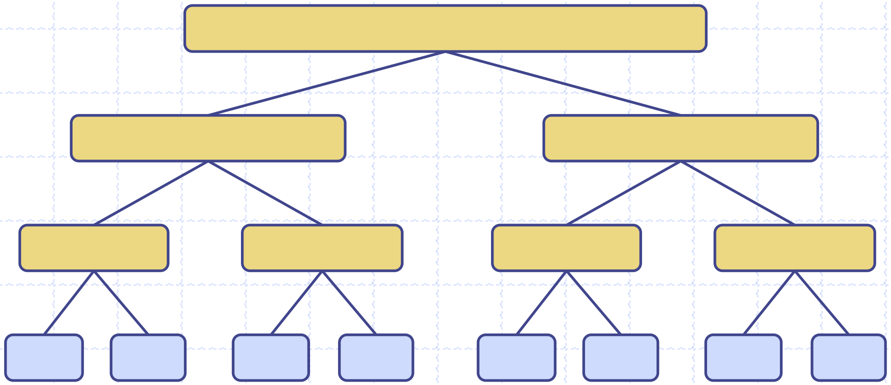
- Merge



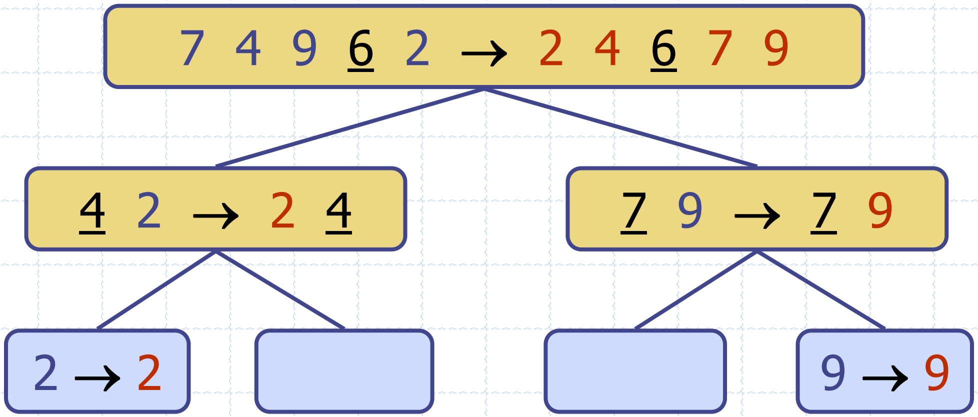
Analysis of Merge-Sort

- 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 or work done at the nodes of depth i is $O(n)$
  - tree has 2^i nodes at depth i
  - we partition and merge 2^i sequences of size $n/2^i$
  - Overall time spent at all the nodes at depth i is $O(2^i \cdot n/ 2^i)$ which is $O(n)$
- 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	$n/2^i$
…	…	…

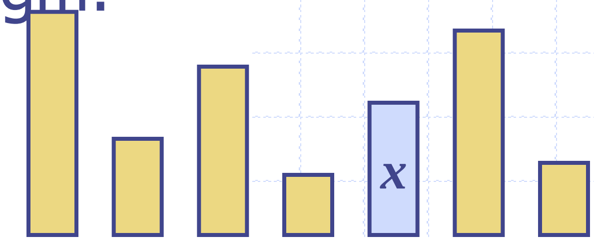


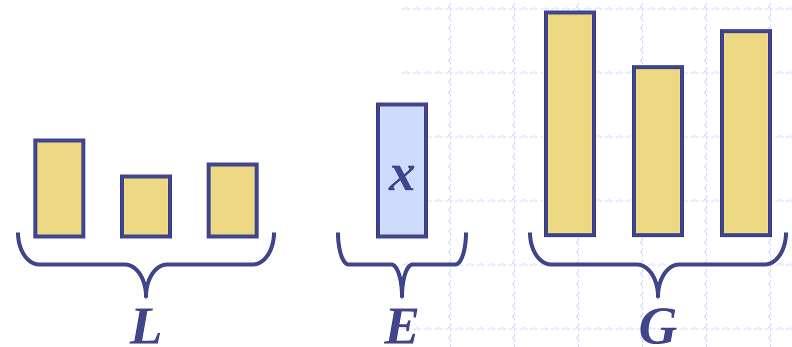
Quick-Sort



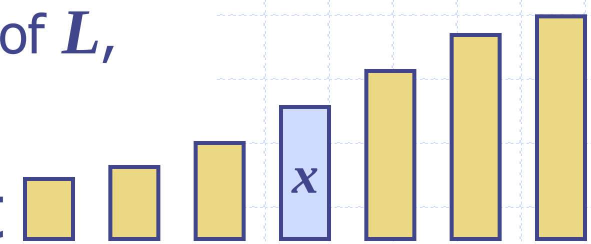
Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: If S has at least two elements (otherwise nothing needs to be done) pick a random element x (called pivot). Remove all the elements from S and put into three sequences:
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: Put back the elements into S in order by first inserting the elements of L, then those of E and finally those of G.





- Common Practice: choose the pivot to be the last element in S



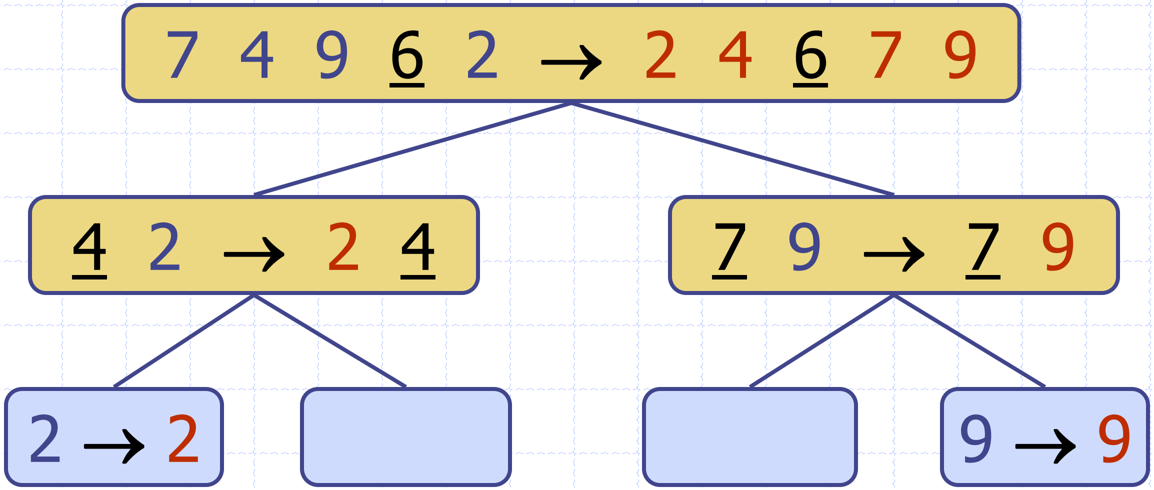
Partition

- 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)
    E.addLast(x)
    while ¬S.isEmpty()
        y ← 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
```

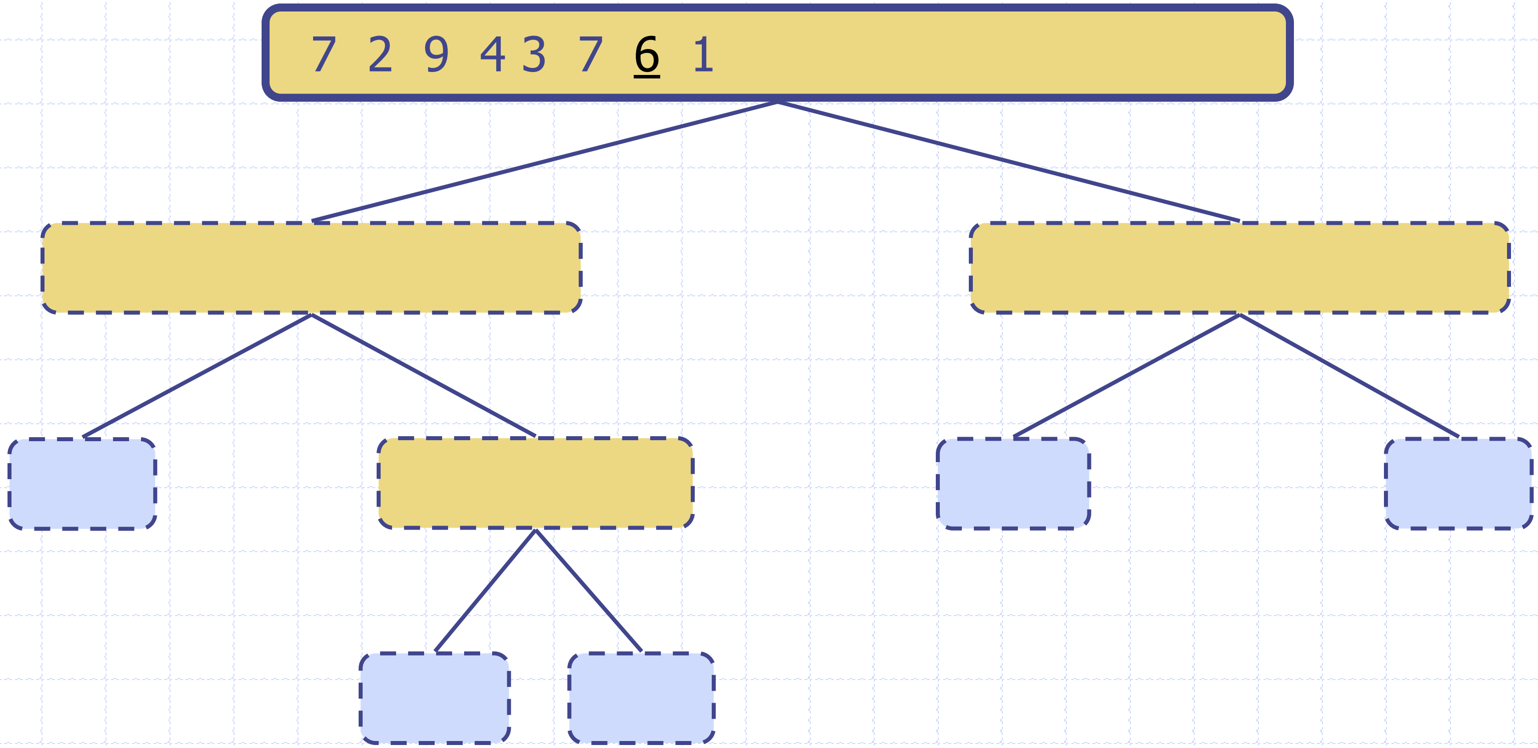
Quick-Sort Tree

- 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



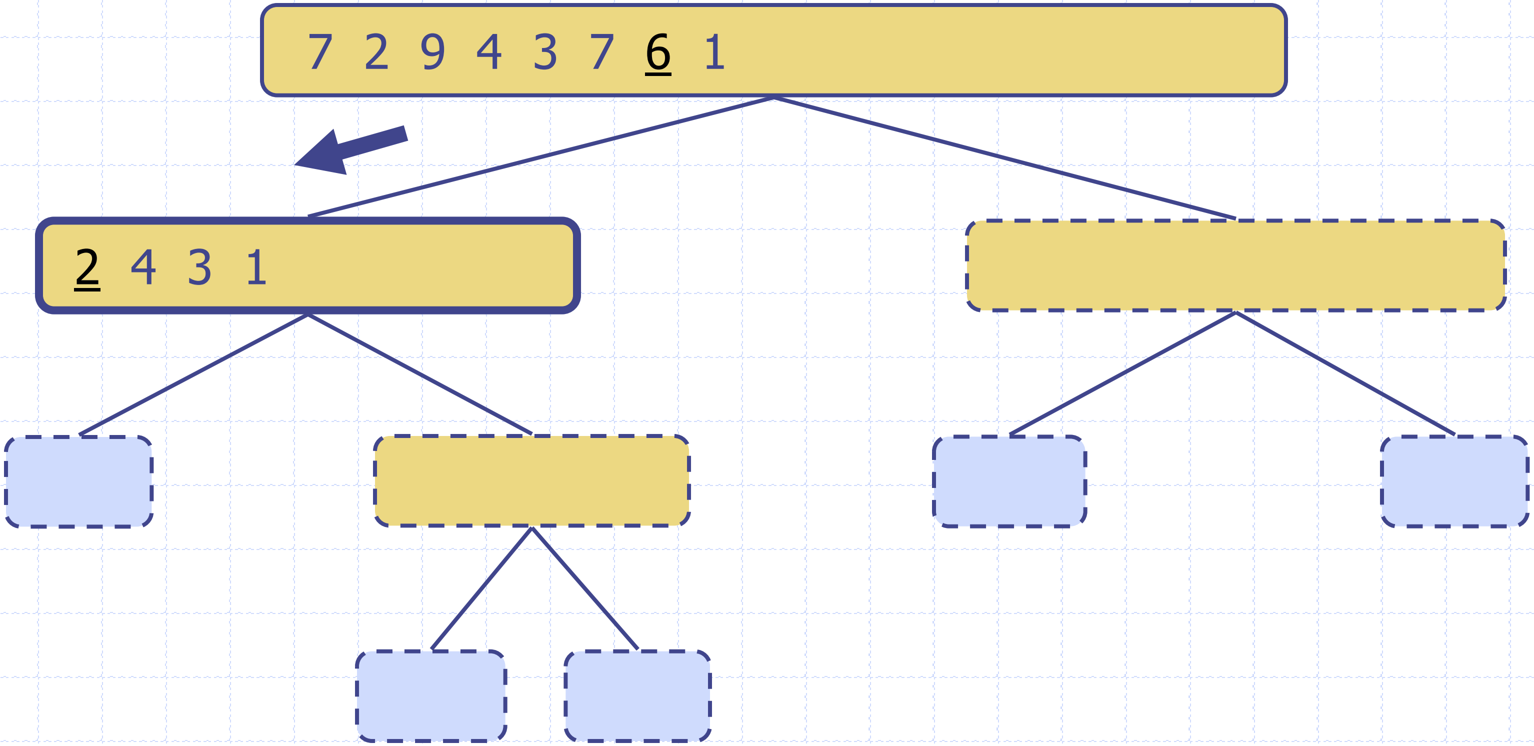
Execution Example (7,2,9,4,3,7,6,1)

- Pivot selection



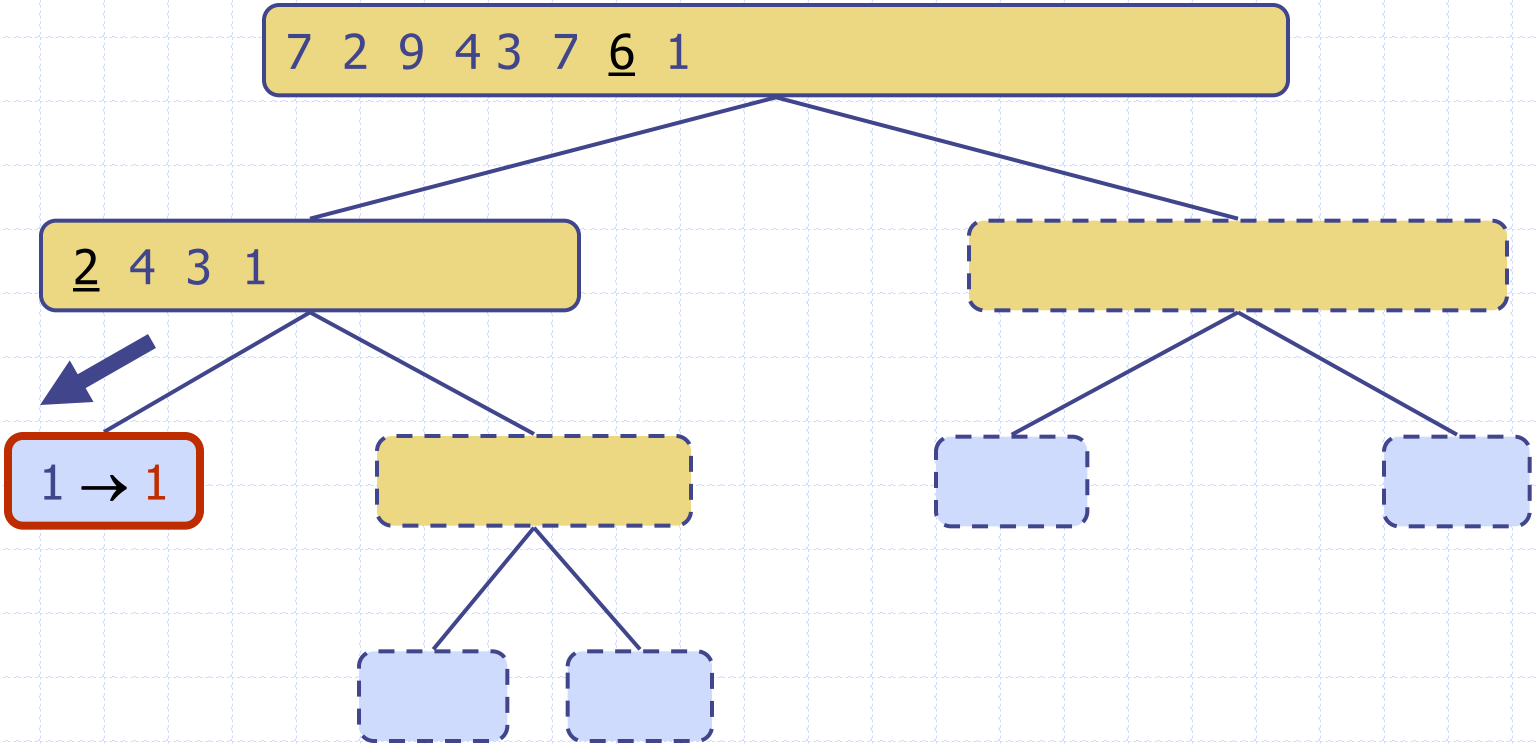
Execution Example (cont.)

- Partition, recursive call, pivot selection



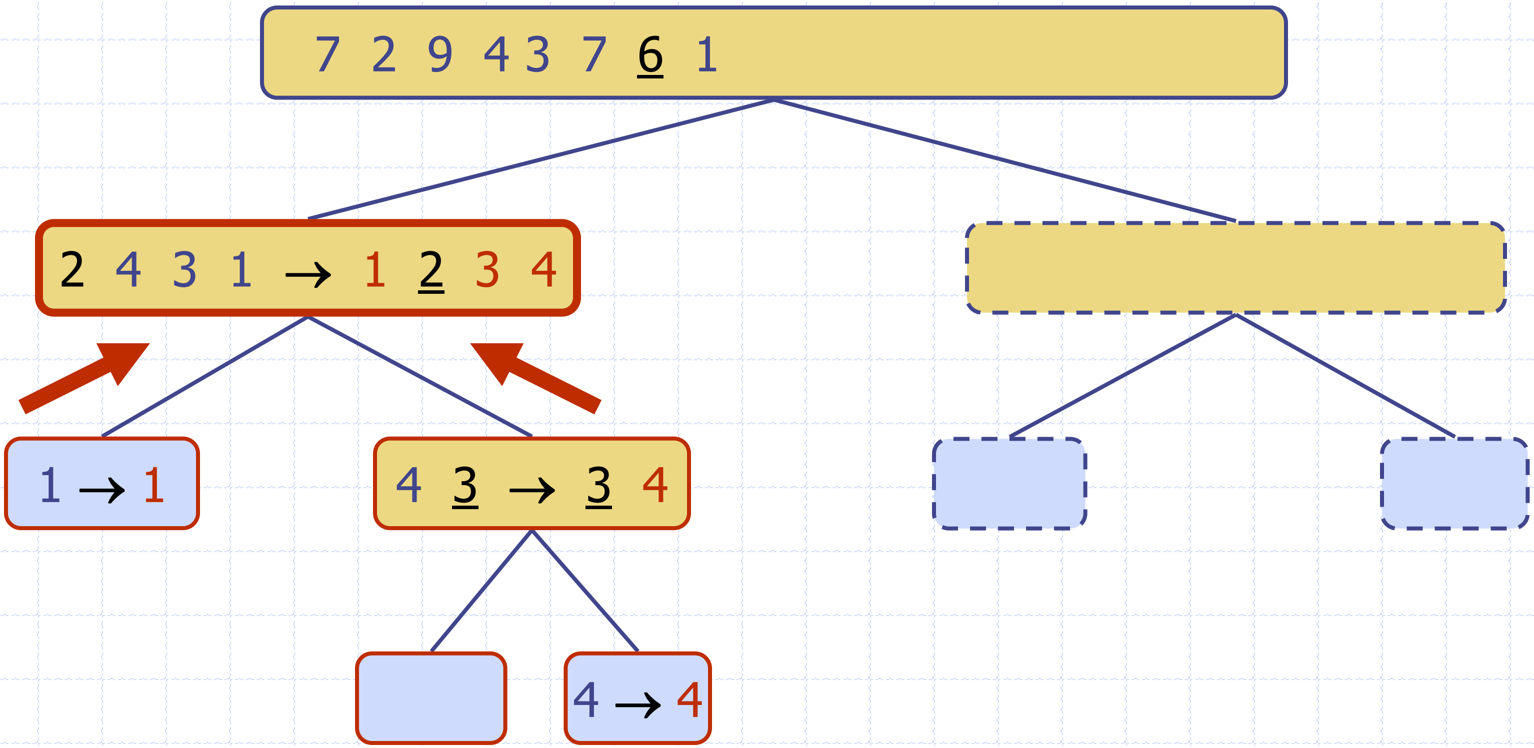
Execution Example (cont.)

- Partition, recursive call, base case



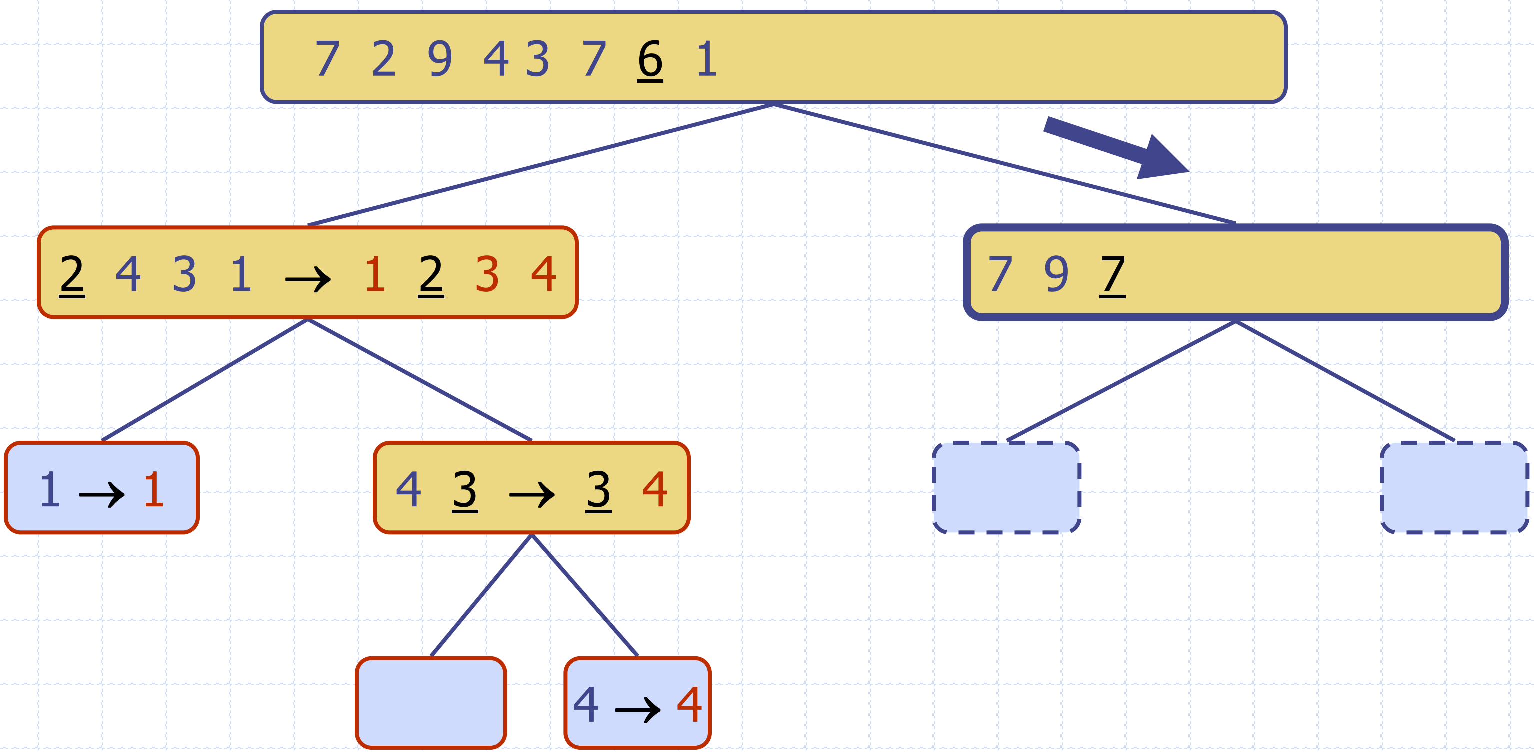
Execution Example (cont.)

- Recursive call, …, base case, join



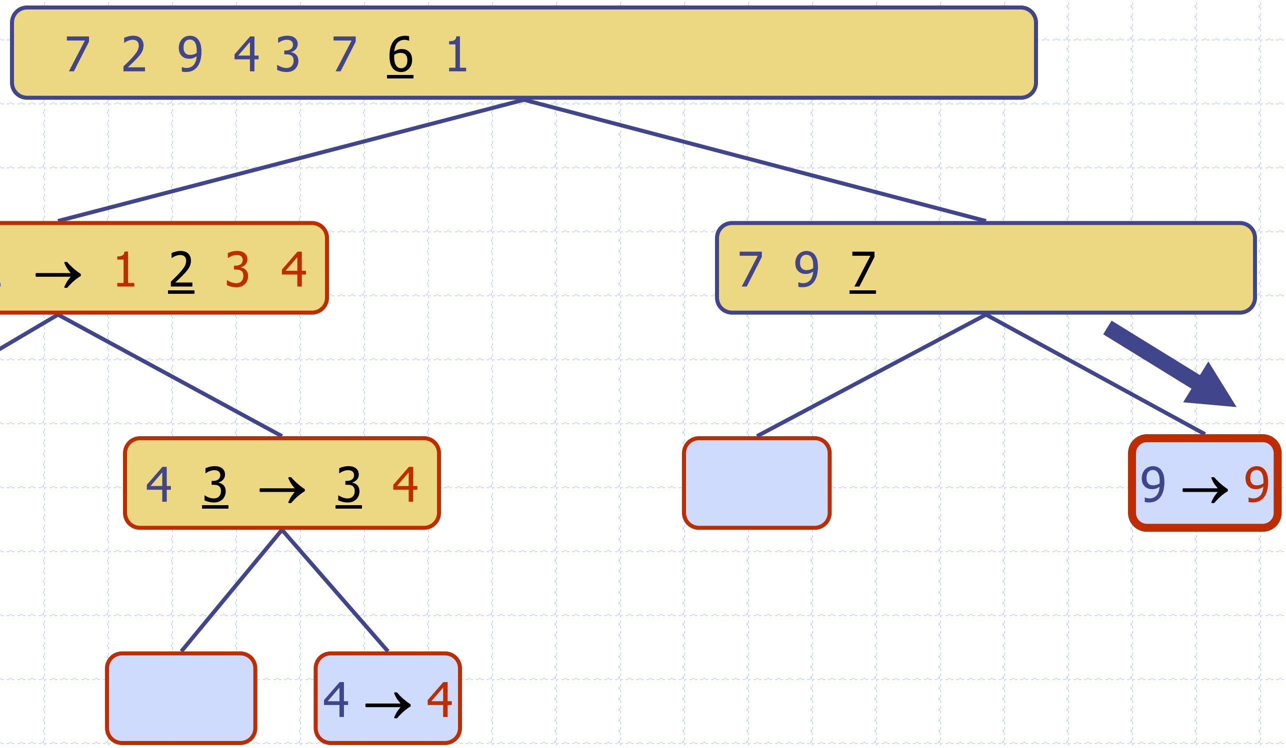
Execution Example (cont.)

- Recursive call, pivot selection



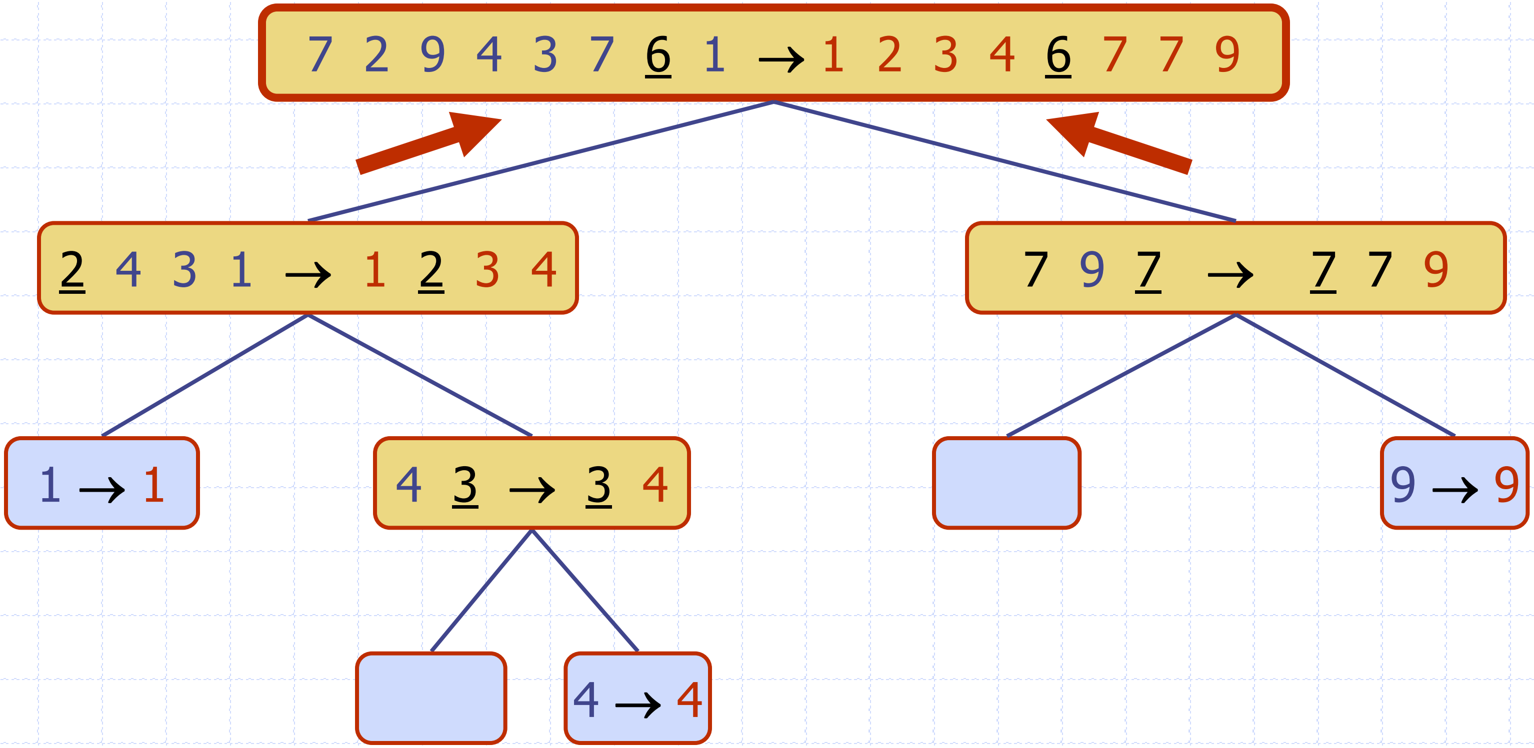
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



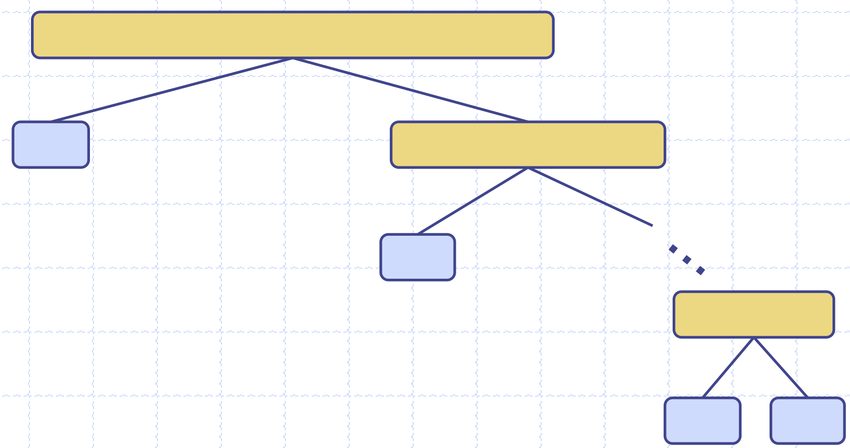
Worst-case Running Time

- 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

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

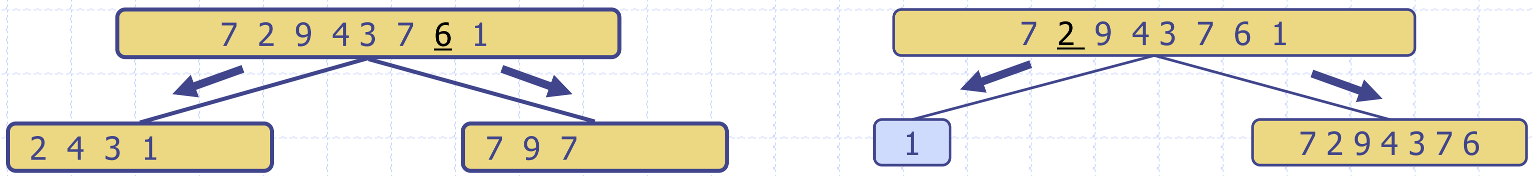
- Thus, the worst-case running time of quick-sort is $O(n^2)$

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



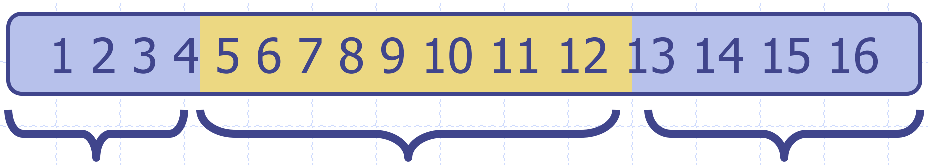
Expected Running Time

- 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**

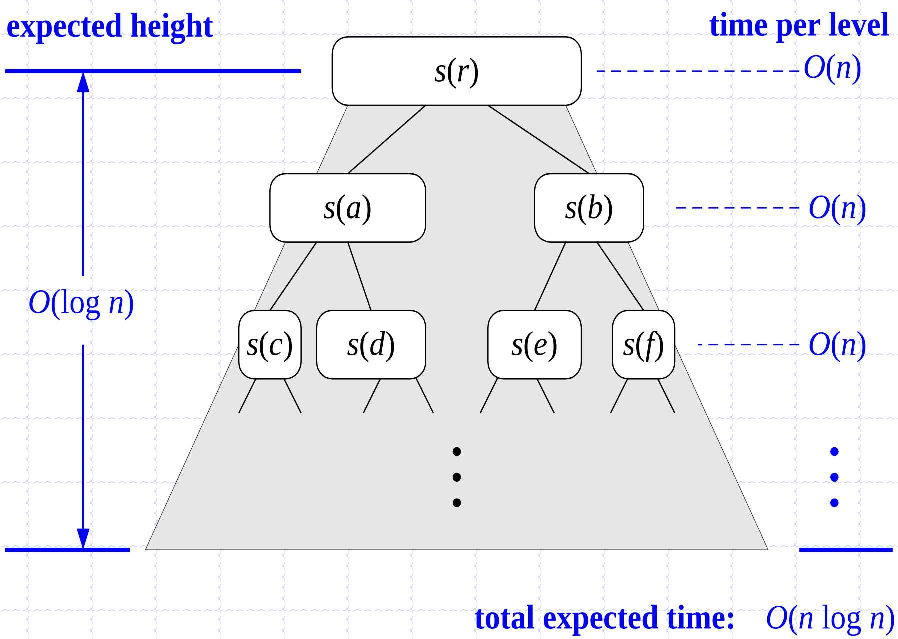
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Bad pivots **Good pivots** **Bad pivots**

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 $O(n \log n)$

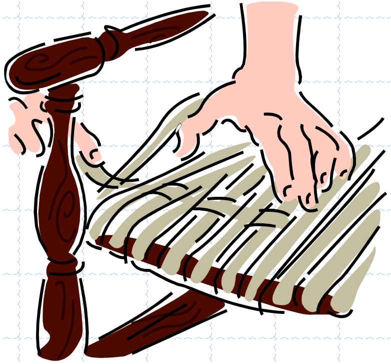


Exercise – Merge-sort and Quick-sort

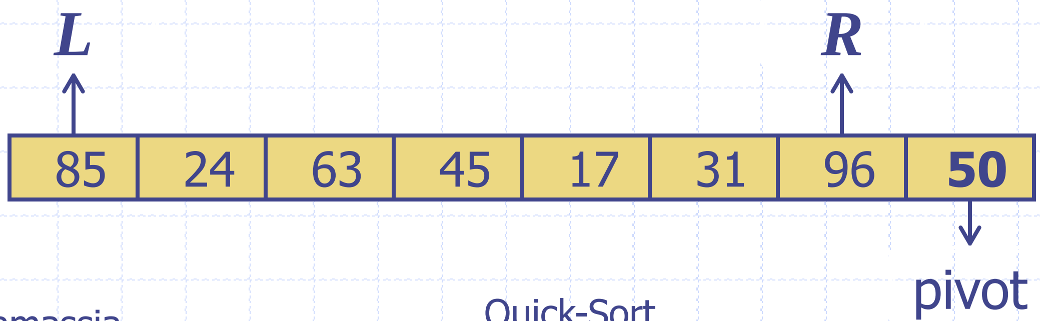
- Perform merge-sort and quick-sort on the following sequence of numbers:

(3, 5, 1, 9, 3)

In-Place Quick-Sort – optimisation of "divide" step



- Quick-sort can be implemented to run in-place
- "Divide" step can be done "in place", that is, without using any additional array, in the following way.
- Assume we want to divide A[leftEnd…rightEnd] with respect to pivot A[rightEnd].
- Maintain two indices, L (left cursor) and R (Right cursor), with initial values leftEnd and rightEnd–1, respectively. The elements which have not been considered yet are in $A[(L+1)..(R-1)]$.



In-Place Quick-Sort – optimisation of "divide" step

L						R	pivot
85	24	63	45	17	31	96	**50**

- Iterate until L<=R:
  - Keep increasing L by 1 until an element $A[L]$ is smaller than the pivot and L<=R.
  - Keep decreasing R by 1 until an element $A[R]$ is larger than the pivot and L<=R.
  - If L<R swap $A[L]$ and $A[R]$, and proceed to the next iteration.
- Swap $A[L]$ and pivot

Iteration 1:

$L < R$ – swap $A[L]$ and $A[R]$

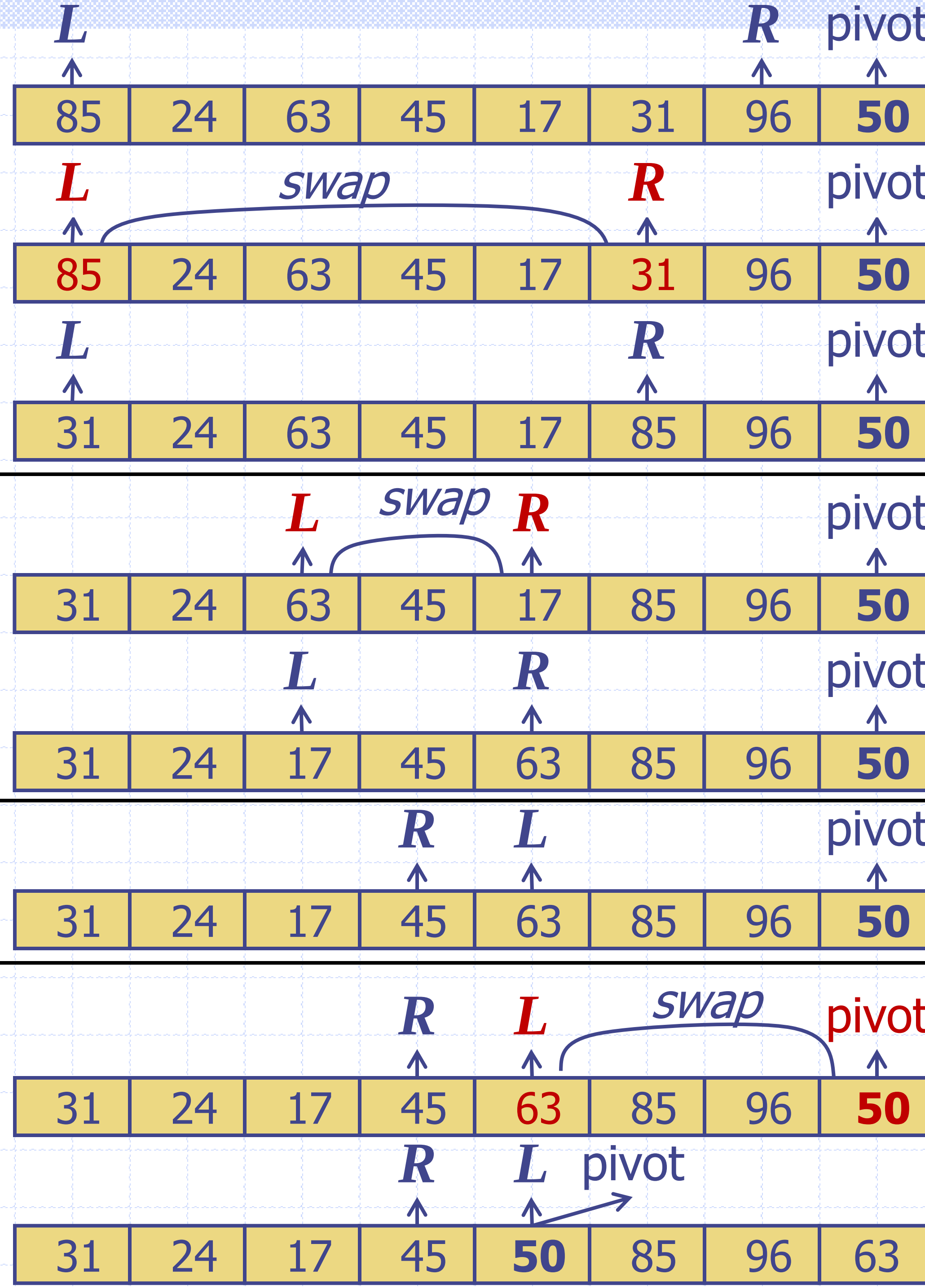
Iteration 2:

$L < R$ – swap $A[L]$ and $A[R]$

Iteration 3:

$L < R$ – swap $A[L]$ and $A[R]$

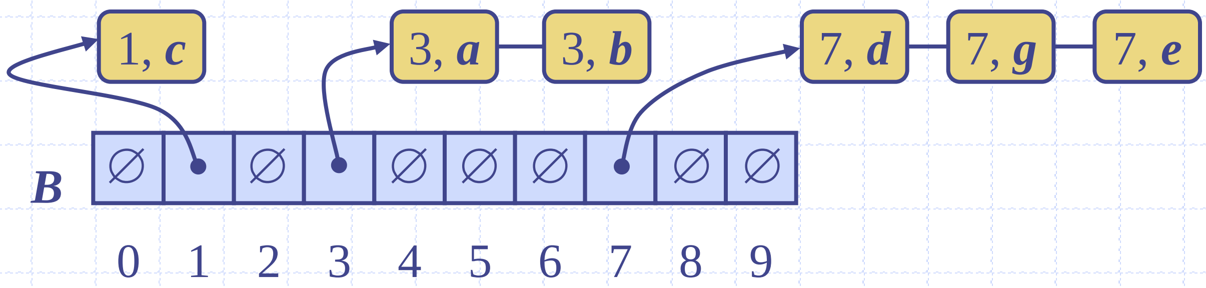
$L > R$ – Swap $A[L]$ and pivot



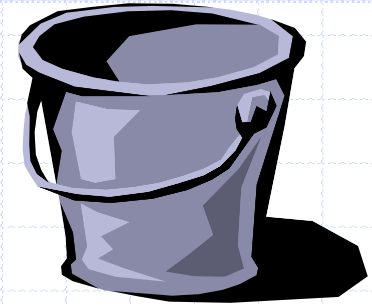
Summary of Sorting Algorithms

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 ▪ 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)

Bucket-Sort



Bucket-Sort



- Bucket-sort does not use comparison
- Let S be a sequence of n (key, element) entries with keys in the range $[0, N-1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
  - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$, move the entries of bucket $B[i]$ to the end of sequence S
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n+N)$ time

Bucket-sort takes $O(n+N)$ time

Algorithm *bucketSort*(S, N)

    Input sequence S of (key, element) items with keys in the range $[0, N-1]$

    Output sequence S sorted by increasing keys

    $B \leftarrow$ array of N empty sequences

    while $\neg S.isEmpty()$

        $f \leftarrow S.first()$

        $(k, o) \leftarrow S.remove(f)$

        $B[k].addLast((k, o))$

    for $i \leftarrow 0$ **to** $N-1$

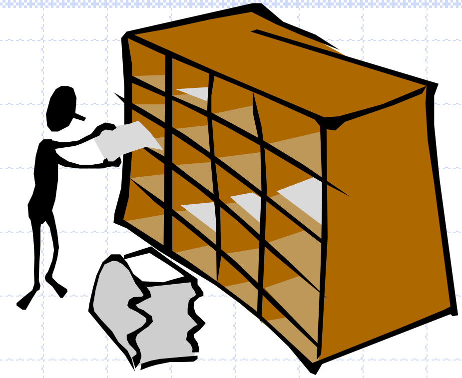
        while $\neg B[i].isEmpty()$

            $f \leftarrow B[i].first()$

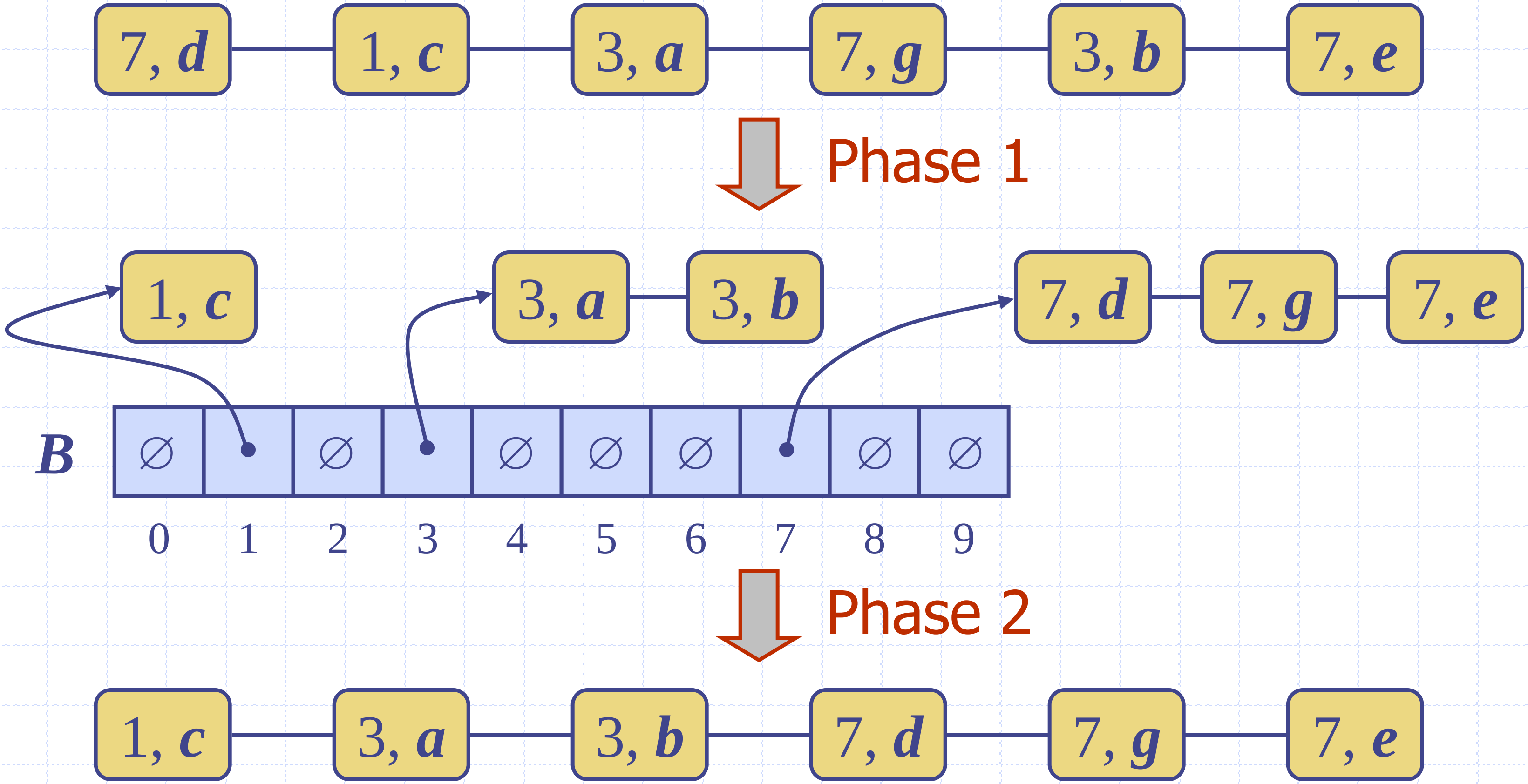
            $(k, o) \leftarrow B[i].remove(f)$

            $S.addLast((k, o))$

Example



- Key range [0, 9]



Exercise – Bucket-sort

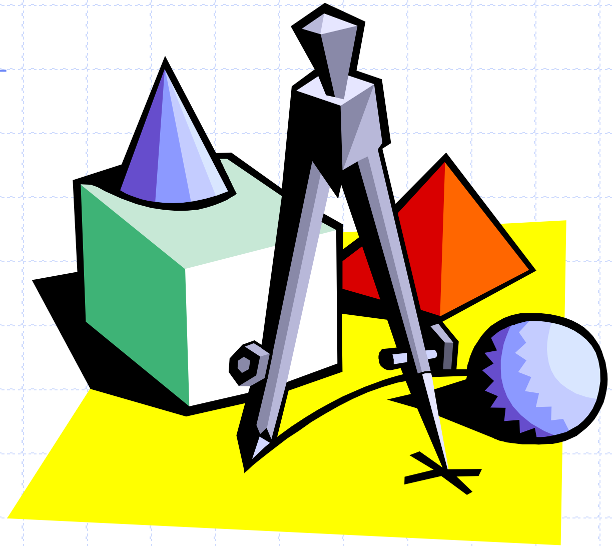
- Perform bucket-sort on the following sequence of numbers:

  (3, 5, 1, 9, 3, 7, 8, 8)

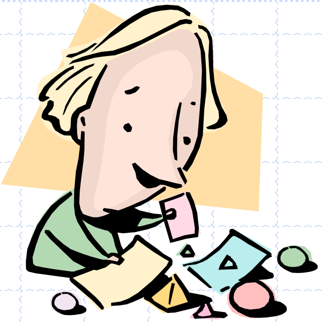
Stability of sorting

- Sorting algorithm is stable if the relative order of any two items with the same key in an input sequence is preserved after the execution of the algorithm.
- Stable sorting algorithms:
  - Merge-sort
  - Bucket-sort
- Unstable sorting algorithms:
  - Quick-sort

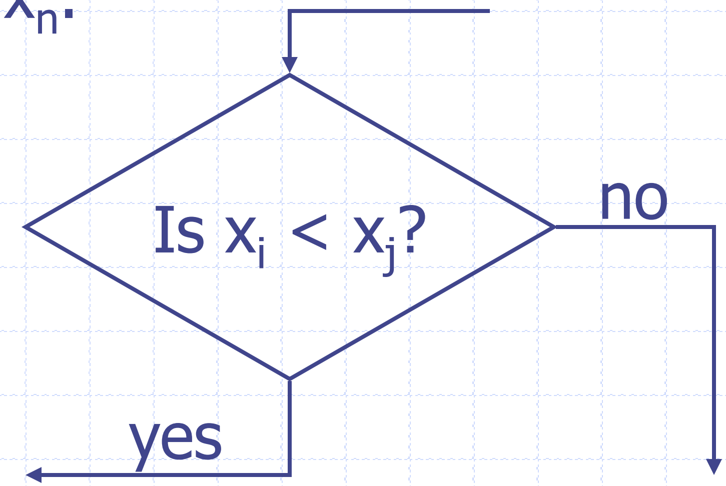
Sorting Lower Bound



Comparison-Based Sorting

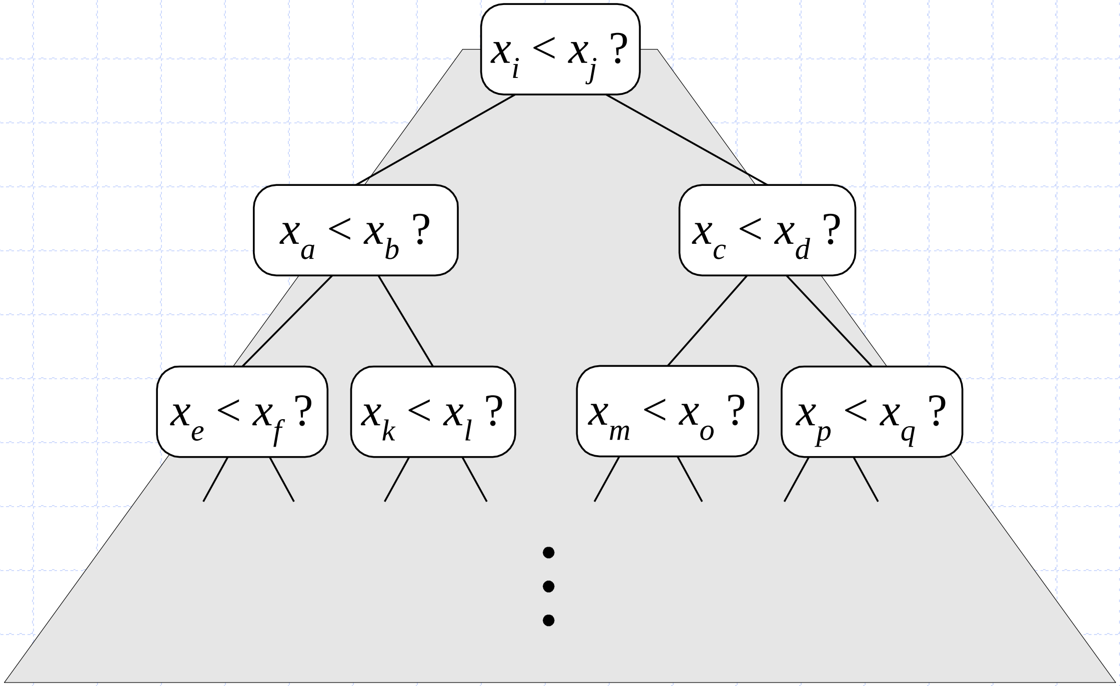


- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.



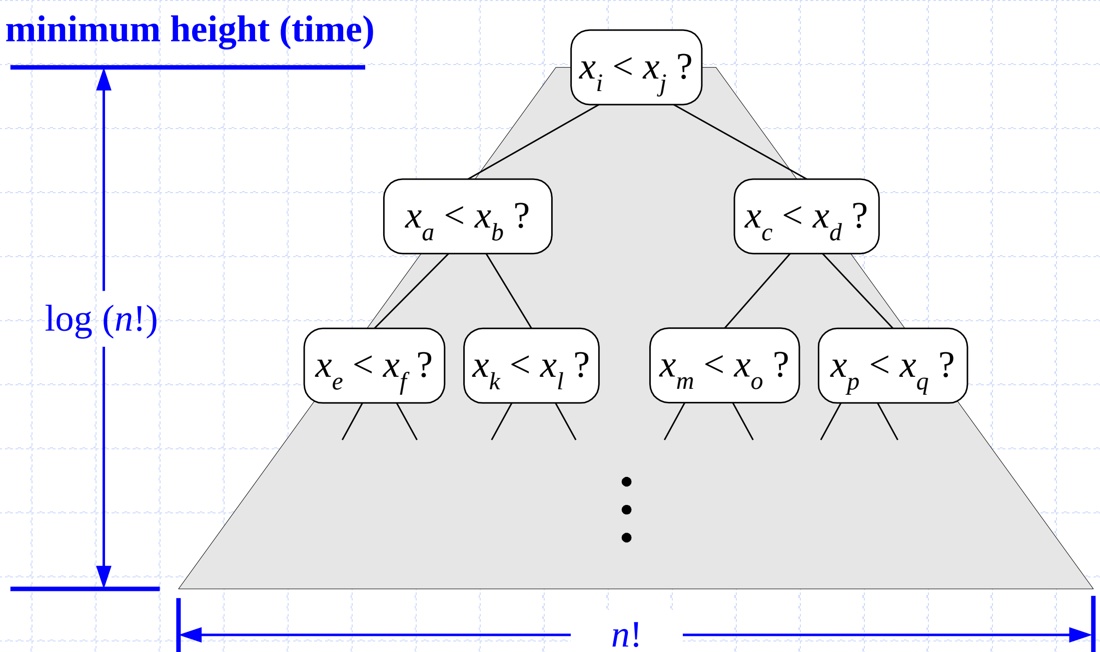
Counting Comparisons

- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree

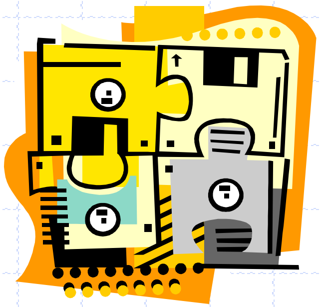


Decision Tree Height

- The height of the decision tree is a lower bound on the running time
- Every input permutation must lead to a separate leaf output
- If not, some input …4…5… would have same output ordering as …5…4…, which would be wrong
- Since there are n!=1·2 · … ·n leaves, the height is at least log (n!)



The Lower Bound



- Any comparison-based sorting algorithms takes at least log (n!) time
- Therefore, any such algorithm takes time at least

$$\log\,(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log\,(n/2).$$

- That is, any comparison-based sorting algorithm must run in Ω(n log n) time.