Lecture 5:
Merge-Sort and Quicksort

Integration of Diversity and Unity

Wholeness Statement

Merge-sort and Quicksort, in effect, organize data into a binary tree, starting from the root (silence) and proceeding to the leaves (dynamism). The root of life, the pure consciousness (silence) experienced during our meditation, sequentially expresses itself as manifest creation (dynamism).

Outline and Reading

- Recursive Programming
- Divide-and-conquer paradigm (§4.1.1)
- Merge-sort (§4.1.1)
  - Execution example
  - Algorithm
  - Merging two sorted sequences
  - Merge-sort tree
  - Analysis
- Generic merging and set operations (§4.2.1)
- Summary of sorting algorithms (§4.2.1)

Recursive Programming

Basic Concepts

- Recognizing Recursion
  - When smaller or simpler instances form sub-constituents of the overall solution
    - E.g., when a function calls itself on smaller subproblem instances to solve the larger, global problem
- Theoretically, any problem that can be solved using iteration (while and for loops) can be solved using recursion (functional style is supported by functional languages)

Types of Recursion

- Linear recursion
- Tail recursion
- Multiple recursion
- Mutual recursion
- Nested recursion

Types of Recursion

- Linear recursion
  - When a method calls itself only once in the body of the function

```
Algorithm sumFirst(n)
    if n < 0 then Throw InvalidInputException
    if n = 0 then
        return 0
    else
        return n + sumFirst(n-1)
```

Types of Recursion

- Tail recursion
  - A special case of linear recursion in which a method calls itself only once but the call occurs as the last operation executed in the body of the method
  - Functional languages optimize tail recursive functions since there is no need to create a new stack frame (activation record)

```
Algorithm sumFirst(n)
    if n < 0 then Throw InvalidInputException
    return sumFirstHelper(n, 0)

Algorithm sumFirstHelper(n, s)
    if n = 0 then
            return s
    else
            return sumFirstHelper(n-1, n+s)
```

Types of Recursion

- Multiple recursion
  - When a function calls itself two or more times
- Example is MergeSort and QuickSort (later)
- Functions that traverse a binary tree (previously)

- Must be careful because multiple recursion algorithms can quickly explode to $O(2^n)$

```
Algorithm Fib(n)
    if n = 0 then
        return 0
    else if n = 1 then
        return 1
    else
        return Fib(n-2) + Fib(n-1)
```

Types of Recursion

- Mutual recursion
  - When a group of methods repeatedly call each other until a base case is reached

```
Algorithm isEven(n)
    if n = 0 then
            return true
    else
            return isOdd(n-1)

Algorithm isOdd(n)
   if n = 0 then
            return false
    else
            return isEven(n-1)
```

Types of Recursion

- Nested recursion
  - When the argument to a recursive call is calculated via another recursive call
  - Sometimes called Double Recursion

```
Algorithm A(n, s) {Ackerman function}
    if n = 0 then
            return s + 1
    else if s = 0 then
            return A(n-1, 1)
    else {n > 0 and s > 0}
            return A(n-1, A(n, s-1))
```

Recursive Thinking

Think declaratively

1. Define the base cases
   - Instance(s) that can be calculated without using recursive calls
2. Decompose the problem into simpler or smaller instances of the original problem
   - A smaller/simpler instance must be moving toward one of the base cases (so the function terminates)
3. Create an induction diagram to determine what to do in addition to the recursive calls

Recursive Thinking (AKA Subgoal Induction)

Input Parameters → Final Solution

↓ ↑

Smaller Input Parms. → Smaller Solutions

Exercises

1. Write a pseudo code function, *isEven*(n) to <u>recursively</u> determine whether a natural number, n, is an even number.
2. Write a pseudo code function, *sum*(n), to <u>recursively</u> calculate the sum of the first n natural numbers.
3. Write a pseudo code function, *sum2*(n), to recursively sum the first n natural numbers but divide the problem in half and make two recursive calls.
4. Write a pseudo code function, power(x, k), that computes x^k. Can you do this in log k time?

Exercise on Binary Trees

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
  - positionIterator positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)
- Additional BinaryTree methods:
  - position leftChild(p)
  - position rightChild(p)
  - position sibling(p)

Exercise:

- Write a recursive method to find the smallest of the integers in a binary tree of integers. Assume the external nodes to not contain integers.

Algorithm findSmallest(T)

Main Point

1. Any iterative algorithm can be computed using recursion, i.e., a function calling itself. In fact, the meaning of while- and for-loops are defined using recursive functions in programming language semantics (Denotational Semantics). Recursive algorithms keep reducing the size of the inputs instances until a base case is reached, then the solution is computed from the base case up to the solution for the whole problem.

   Science of Consciousness: Maharishi describes the process of creation as a self-referral process that unfolds sequentially. The dynamism of the unified field seems chaotic when studied at the macroscopic level, yet it is a field of perfect order, responsible for the order and balance in creation.

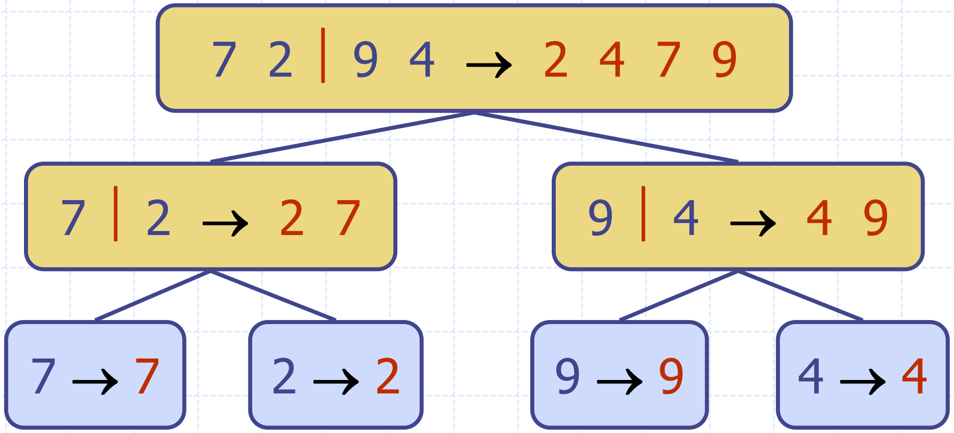
Divide-and-Conquer

- Divide-and conquer is a general algorithm design strategy:
  - 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

Main Idea

- The divide-and-conquer design paradigm has four aspects:
  - handle the base case,
  - partition into sub-cases,
  - process the sub-cases, and
  - combine the sub-case solutions

Merge Sort



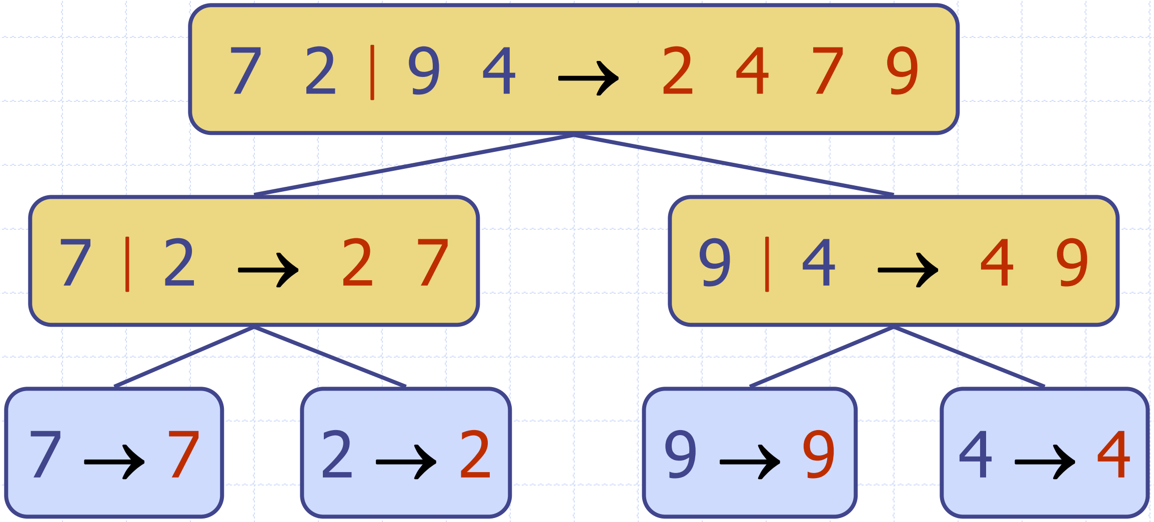
Merge-Sort

- 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 to form the sorted output sequence S

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

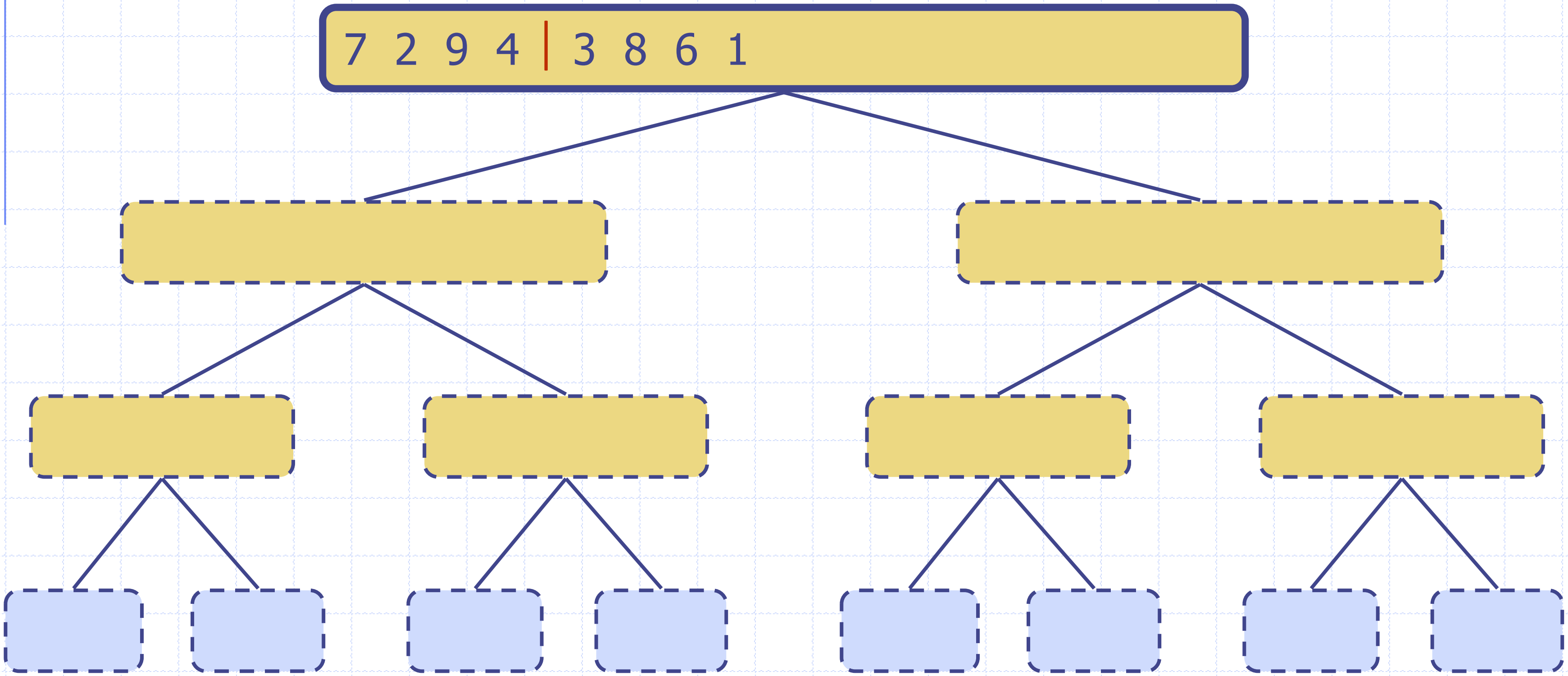
Merge-Sort Tree

- 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



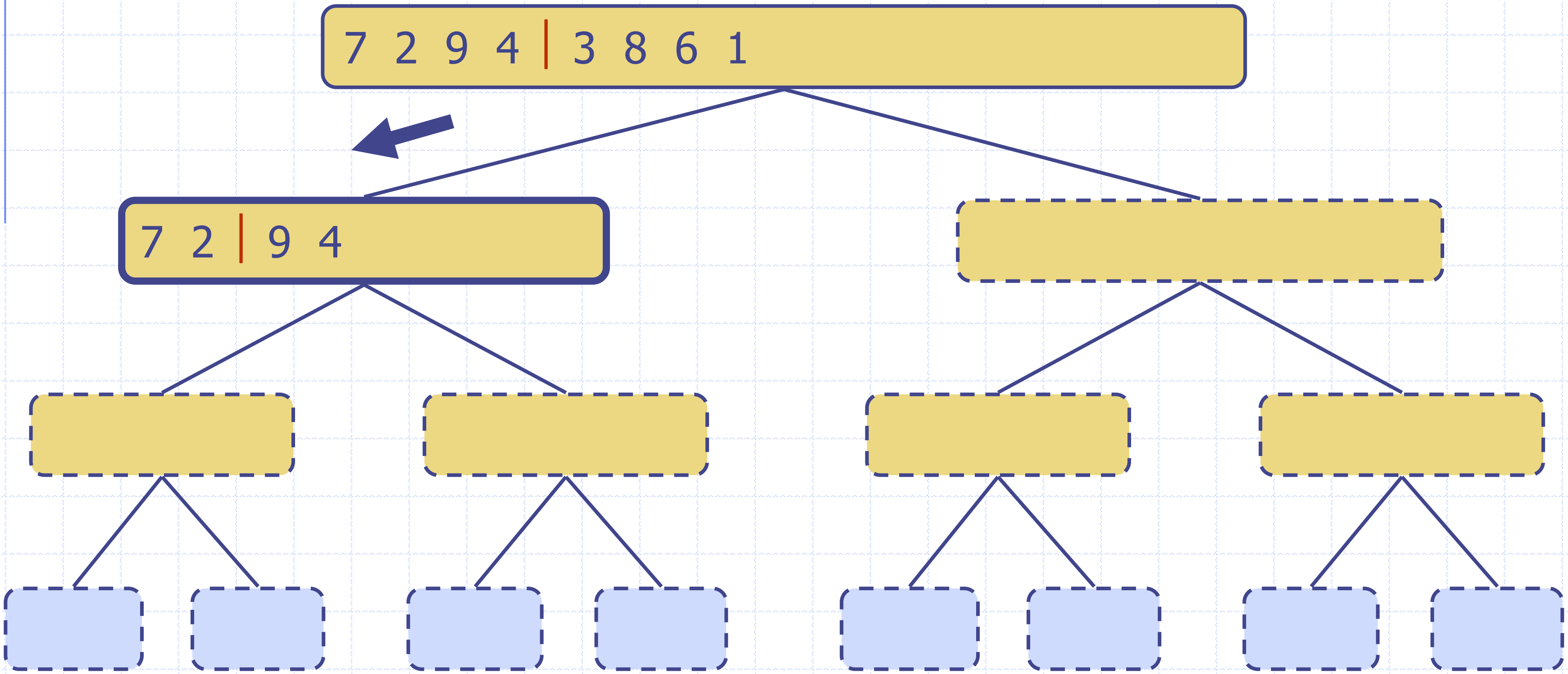
Execution Example

- Partition



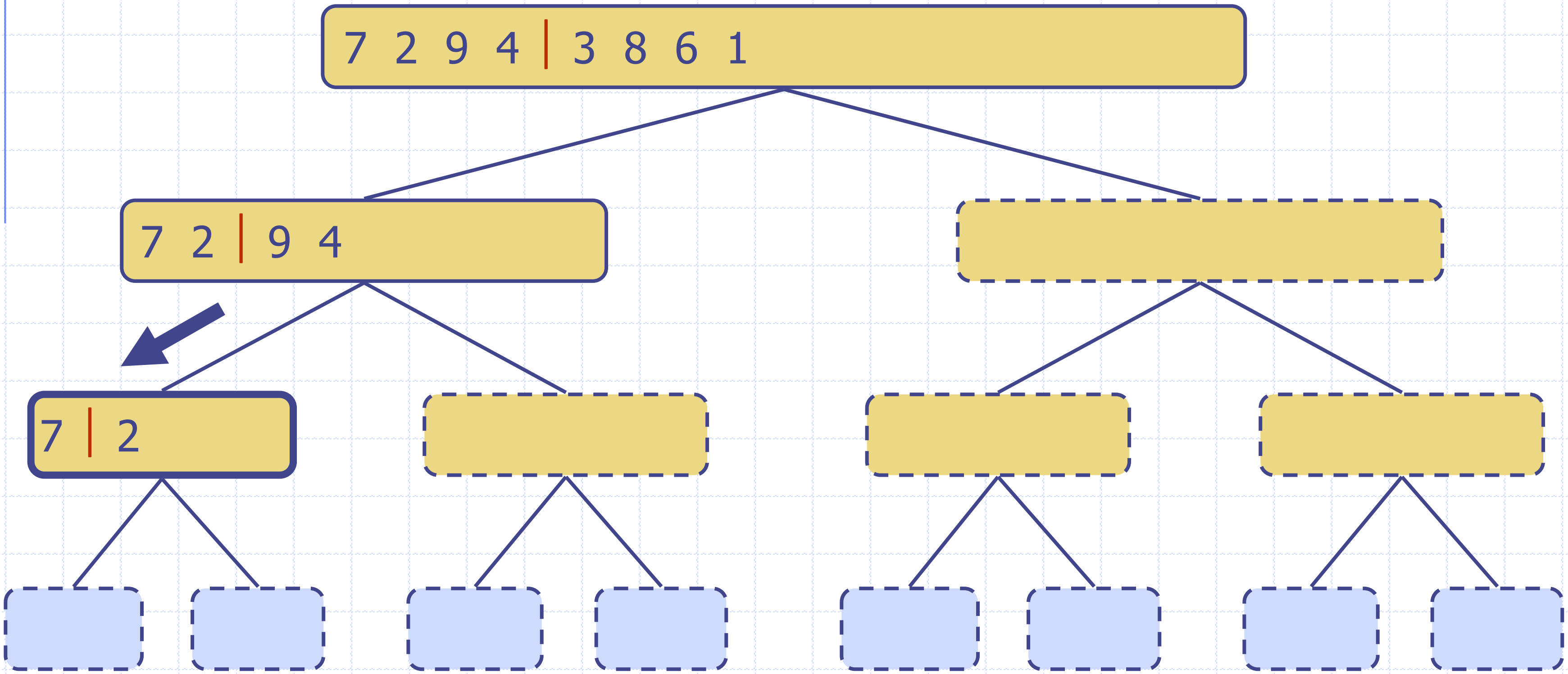
Execution Example (cont.)

- Recursive call, partition



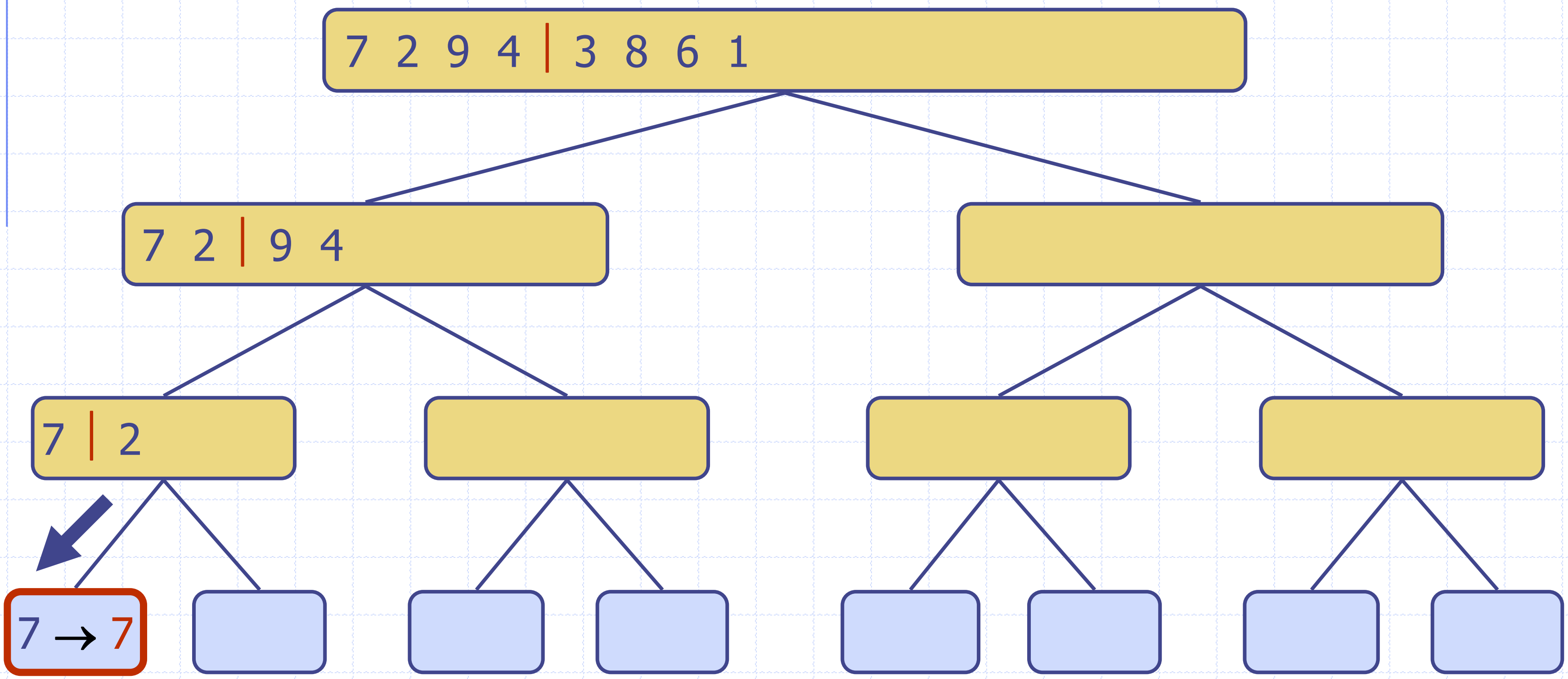
Execution Example (cont.)

- Recursive call, partition



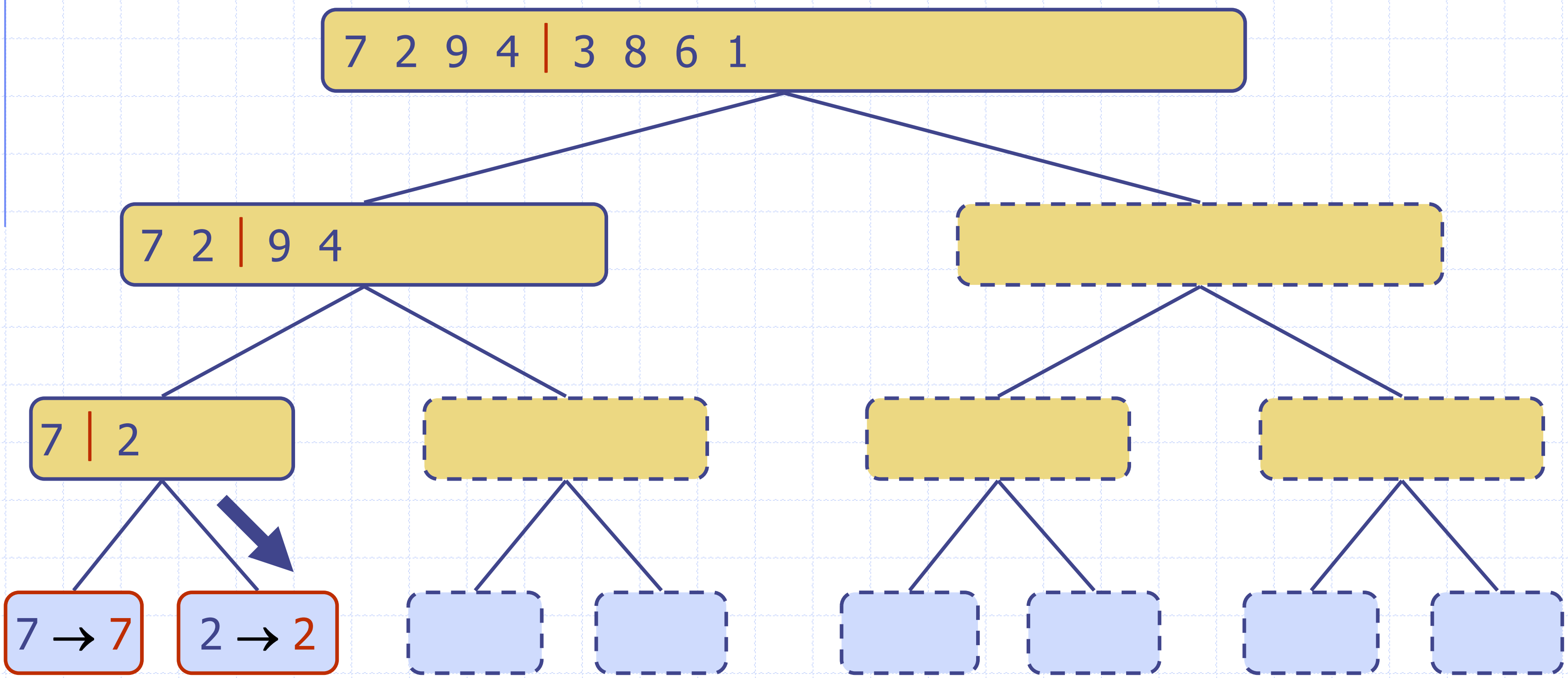
Execution Example (cont.)

- Recursive call, base case



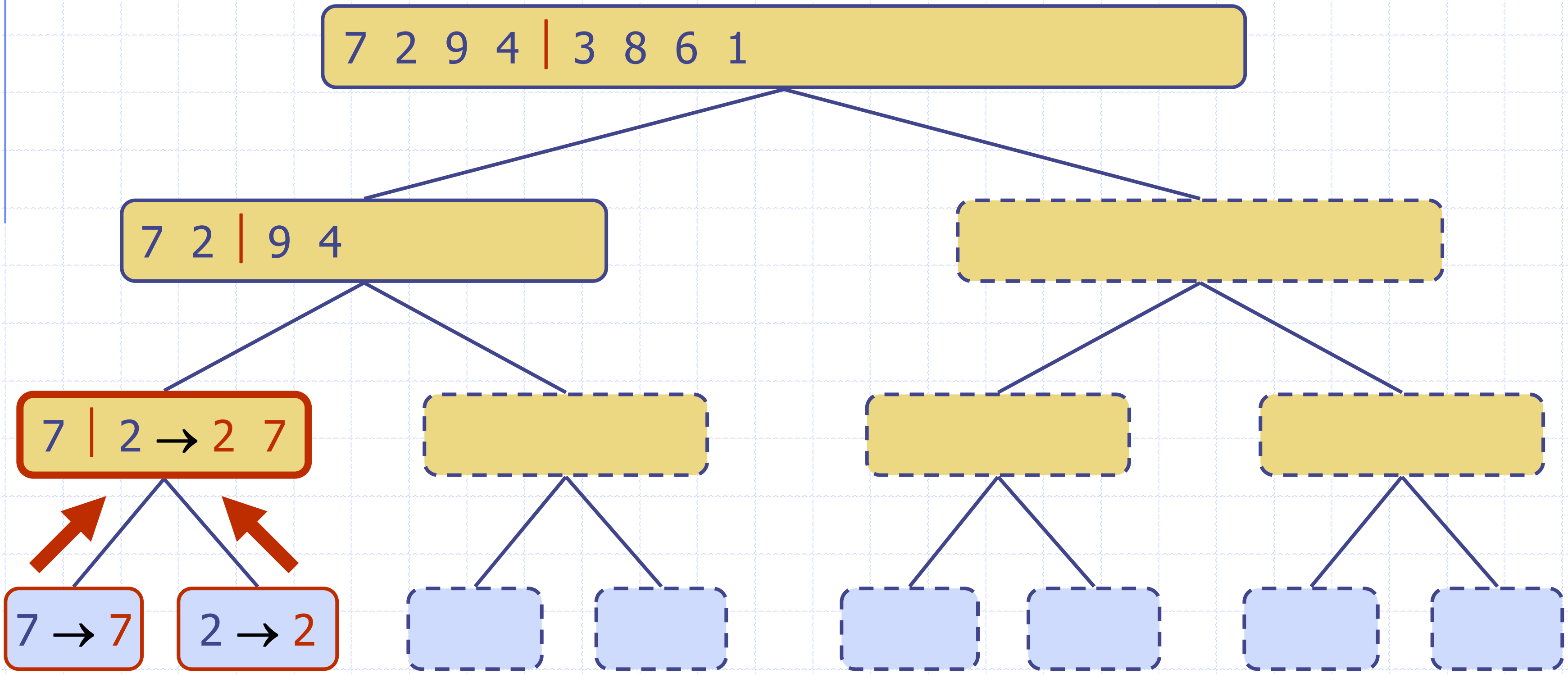
Execution Example (cont.)

- Recursive call, base case



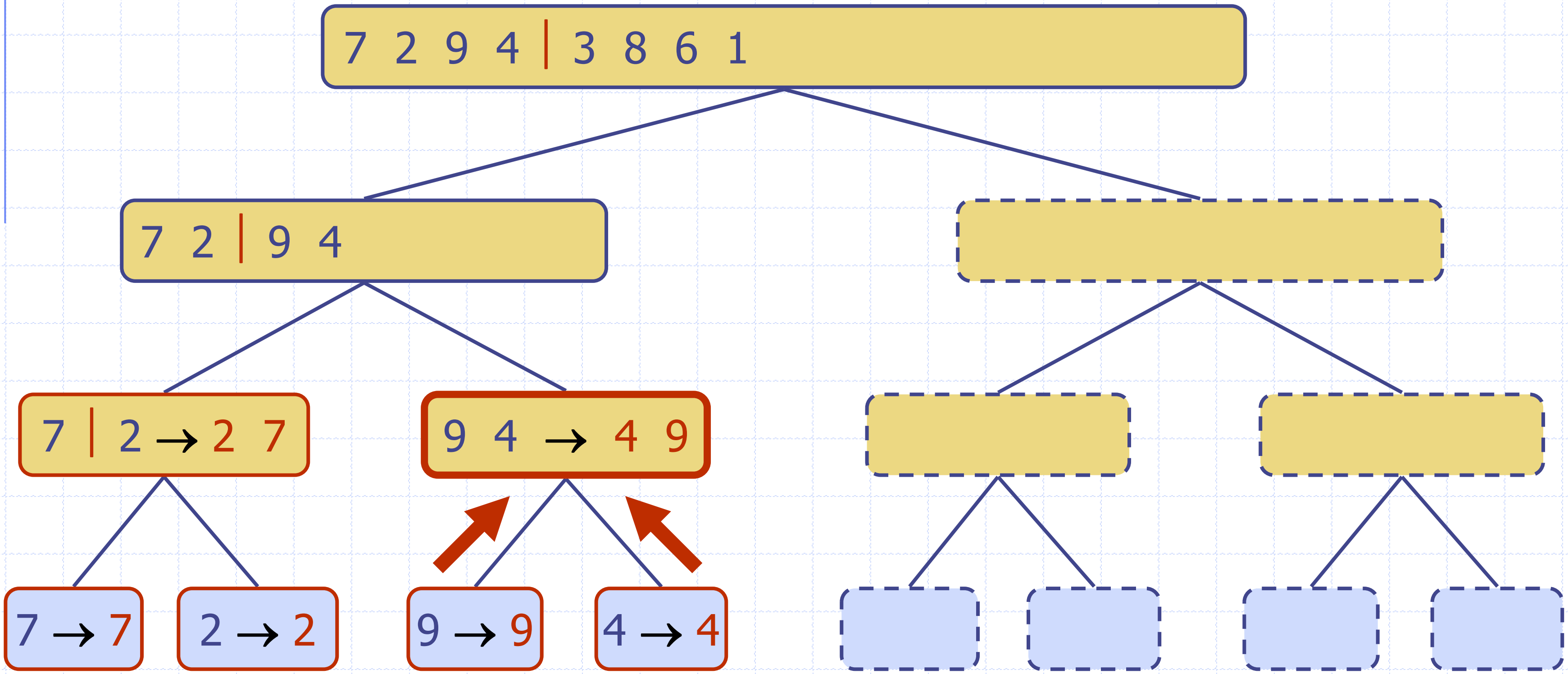
Execution Example (cont.)

- Merge



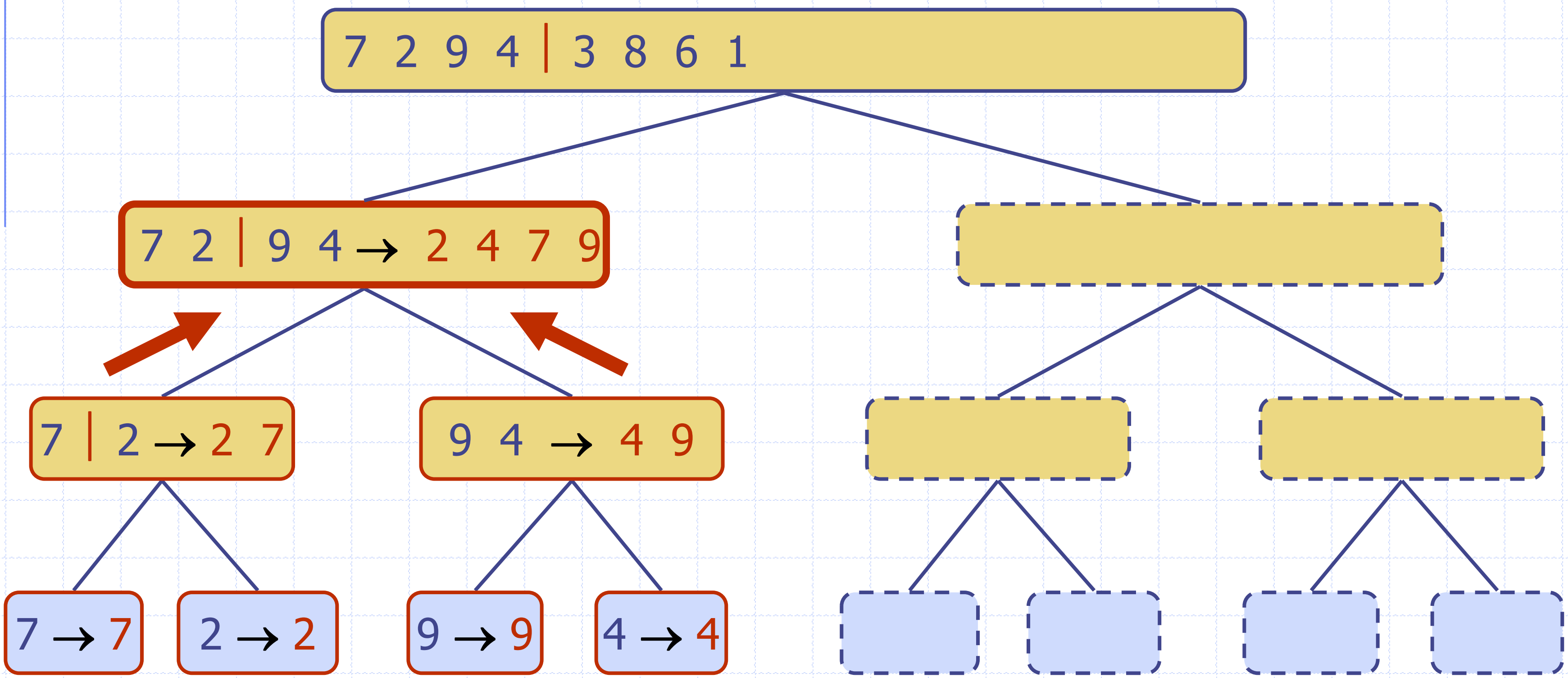
Execution Example (cont.)

- Recursive call, …, base case, merge



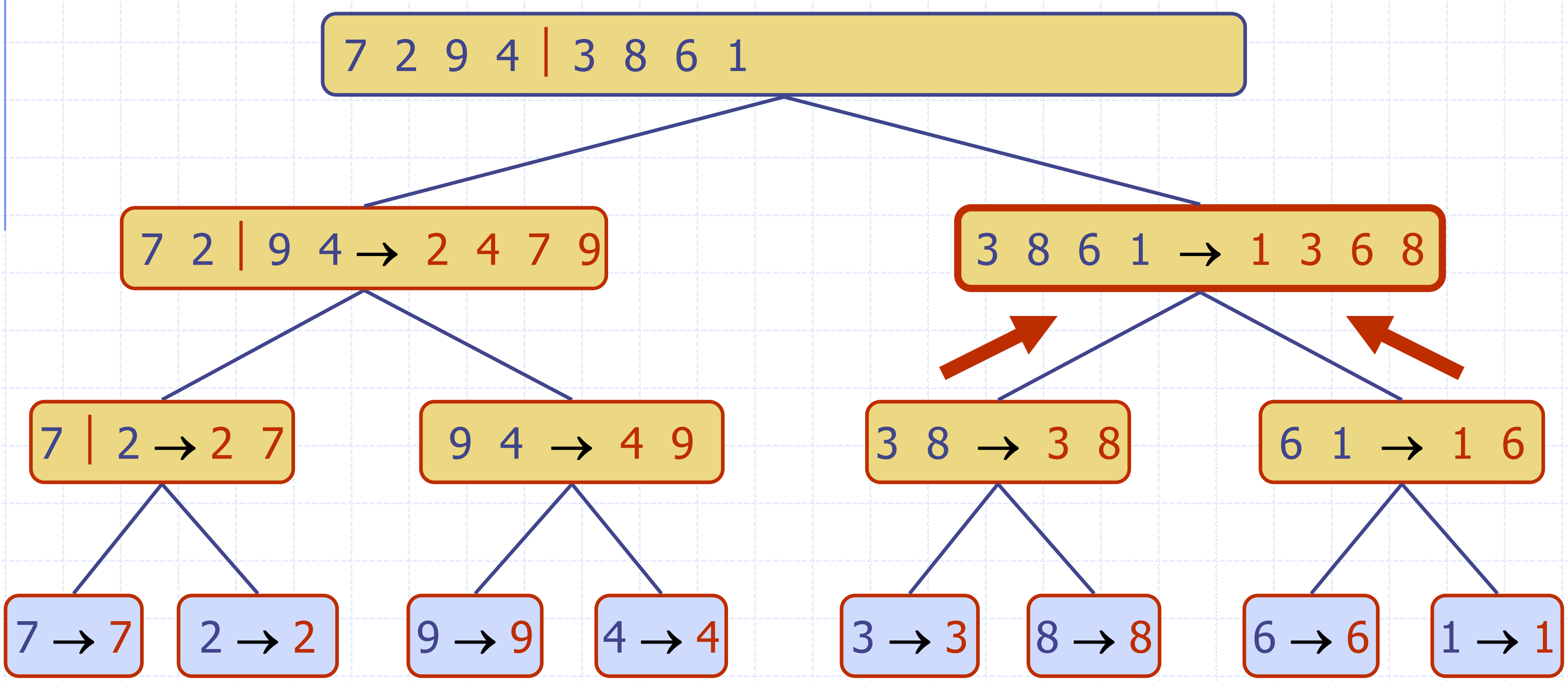
Execution Example (cont.)

- Merge



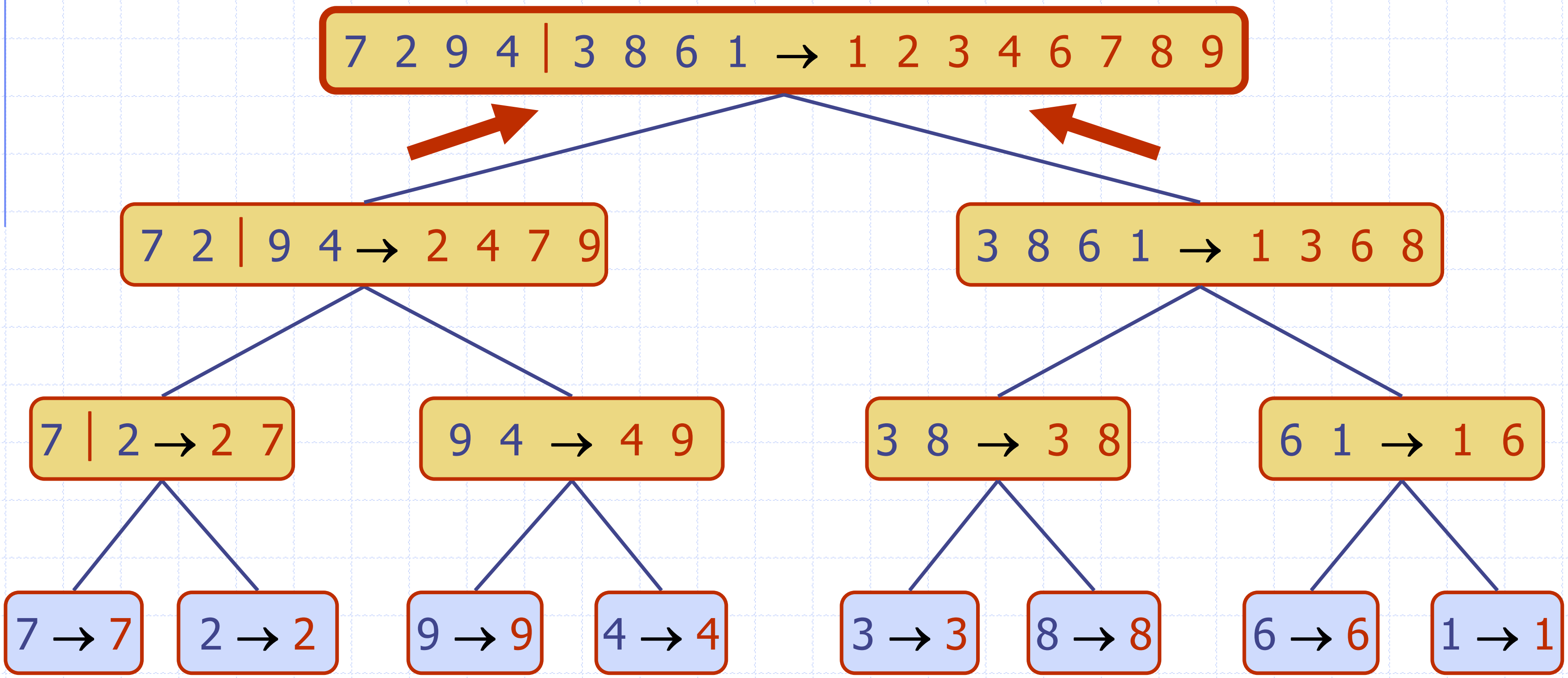
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



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, C, S*)
  Input Sorted sequences ***A*** and ***B*** with ***n***/2 elements each, S is empty, comparator C
  Output S contains sorted sequence of ***A*** ∪ ***B***

```
while !A.isEmpty() ∧ !B.isEmpty() do
   if C.isLessThan( B.first().element(),
                    A.first().element() ) then
         S.insertLast(B.remove(B.first()))
   else
         S.insertLast(A.remove(A.first()))
while !A.isEmpty() do
   S.insertLast(A.remove(A.first()))

while !B.isEmpty() do
   S.insertLast(B.remove(B.first()))
```

Merge-Sort

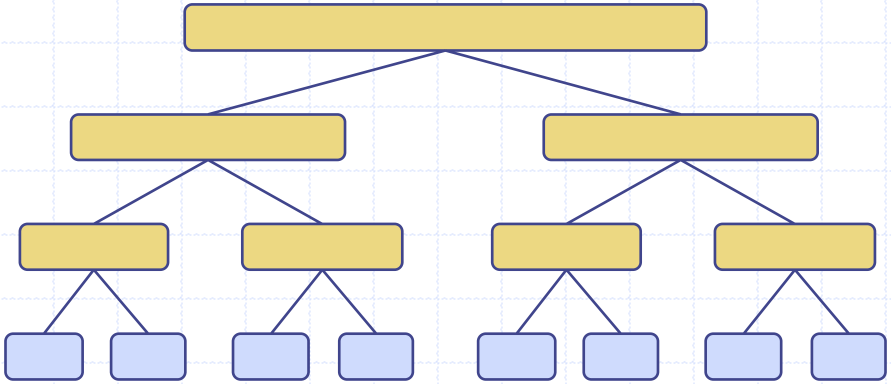
- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 to form the sorted output sequence S

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

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 $O(n)$
  - 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	$n/2^i$
…	…	…



Merge-Sort

- A sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - uses a comparator
  - has $O(n \log n)$ running time
- Unlike heap-sort
  - does not use an auxiliary priority queue
    - Can be done without a priority queue
  - accesses data in a sequential manner
    - (suitable for sorting data on a disk or any data accessed sequentially such as a linked list)

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, not in-place ◆ sequential data access ◆ for huge data sets (> 1M)

Merge-Sort of an Array

- Merge-sort of an array by partitioning into segments of the input array
- Merge-sort on an input sequence S with n integers consists of three steps:
  - Divide: partition S into two segments of about $n/2$ elements each (lo..mid) and (mid+1..hi)
  - Conquer: recursively sort the two segments
  - Combine: merges the two segments back into S in the merge step

Algorithm *mergeSort*(*S*)
    Temp ← *new Sequence of size n*
    mergeSort(*S, 0, S.size()-1, Temp*)

Algorithm *mergeSort*(*S, lo, hi, Temp*)
    Input arrays *S* and *Temp* (work area), and *indices lo, hi*
    Output array *S* with elements between *lo* and *hi* in sorted order
    if *hi* - *lo* + 1 > 1 **then**
        mid ← floor((lo + hi)/2)
        mergeSort(*S, lo, mid, Temp*)
        mergeSort(*S, mid+1, hi, Temp*)
        merge(*S, lo, mid, hi, Temp*)
    return

Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted segments of A back into A in sorted order
- Merging two sorted array segments, each with $n/2$ elements (where n=hi-lo+1) takes $O(n)$ time

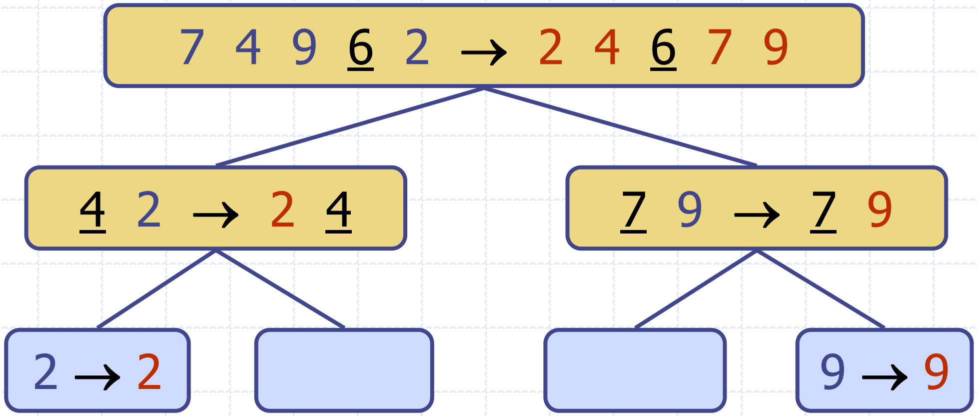
```
Algorithm merge(A, lo, mid, hi, Temp)
    Input Sorted segments of array A between lo..mid and
        mid+1..hi and Temp array is working storage
    Output A contains elements sorted between lo..hi
    size ← hi - lo + 1
    t ← 0
    j ← lo
    k ← mid + 1
    while j ≤ mid  ∧  k ≤ hi  do
        if  A[j] > A[k]  then
                Temp[t] ← A[k]
                k ← k + 1
        else
                Temp[t] ← A[j]
                j ← j + 1
        t ← t + 1
    while  j ≤ mid  do // copy the rest of segment lo .. mid
        Temp[t] ← A[j];
         t ← t+1; j ← j+1;

    while  k ≤ hi  do // copy the rest of segment mid+1 .. hi
        Temp[t] ← A[k];
        t ← t+1; k ← k+1;

    for i ← 0 to size - 1 do // copy sorted part back to A
        A[lo+i] ← Temp[i]
```

Main Point

2. In merge-sort, the input is divided into two equal-sized subsequences, each of which is sorted separately. Then these sorted subsequences are merged together to form the sorted output.

   Science of Consciousness: Through the process of knowing itself, consciousness divides itself into knower and known, yet this 3-in-1 structure is unified at the level of pure consciousness that we experience every day in our meditation.

Quick-Sort



Outline and Reading

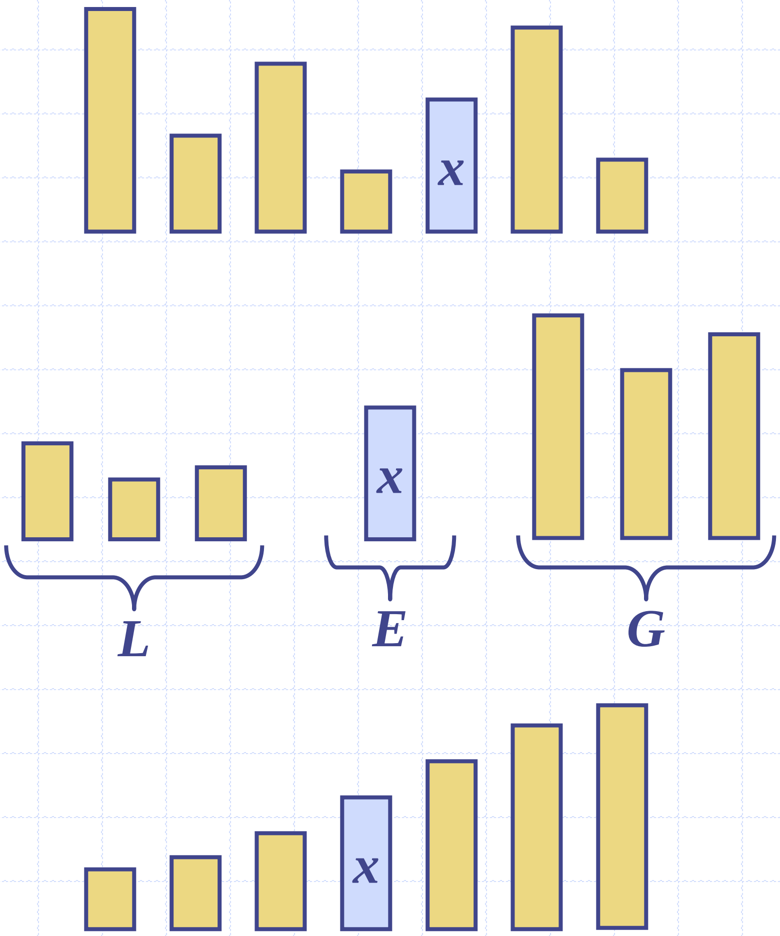
- Quick-sort (§4.3)
  - Algorithm
  - Partition step
  - Quick-sort tree
  - Execution example
- Analysis of quick-sort (4.3.1)
- In-place quick-sort (§4.8)
- Summary of sorting algorithms

Quicksort

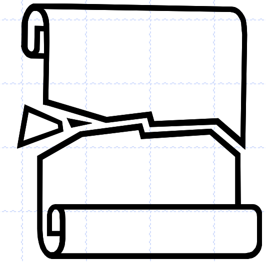
- Divide and Conquer Algorithm
  - The main idea is the moving of a single key (the pivot) to its ultimate location after each partitioning
  - That location is found by
    - moving the smaller values to the left of the pivot and
    - moving the larger values to the right of the pivot
    - the elements are not placed in sorted order in these two partitions
- If sorted in place, no need for a combine step
- Earns its name based on its average behavior

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

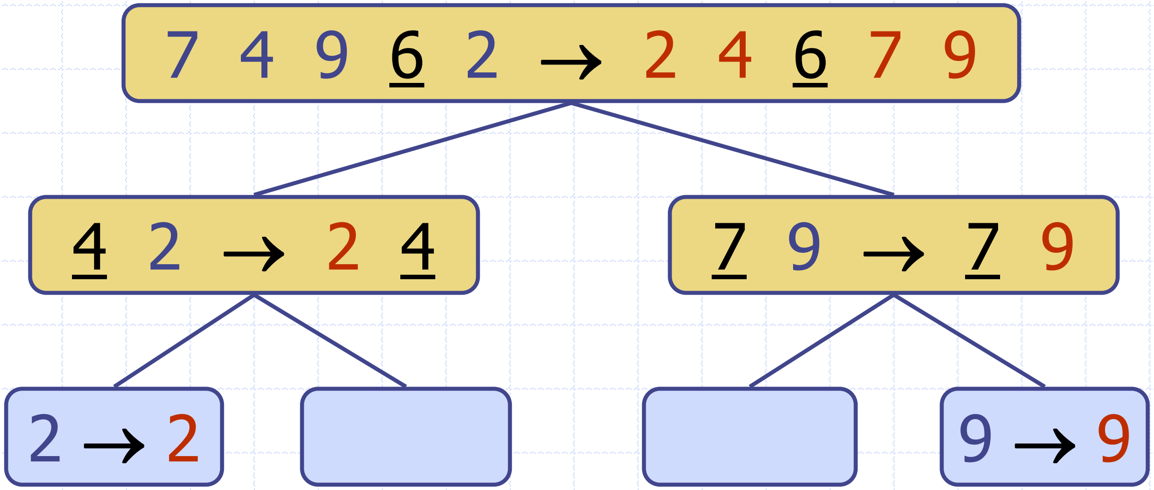


- We partition as follows:
  - Remove each element y from S and
  - insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes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

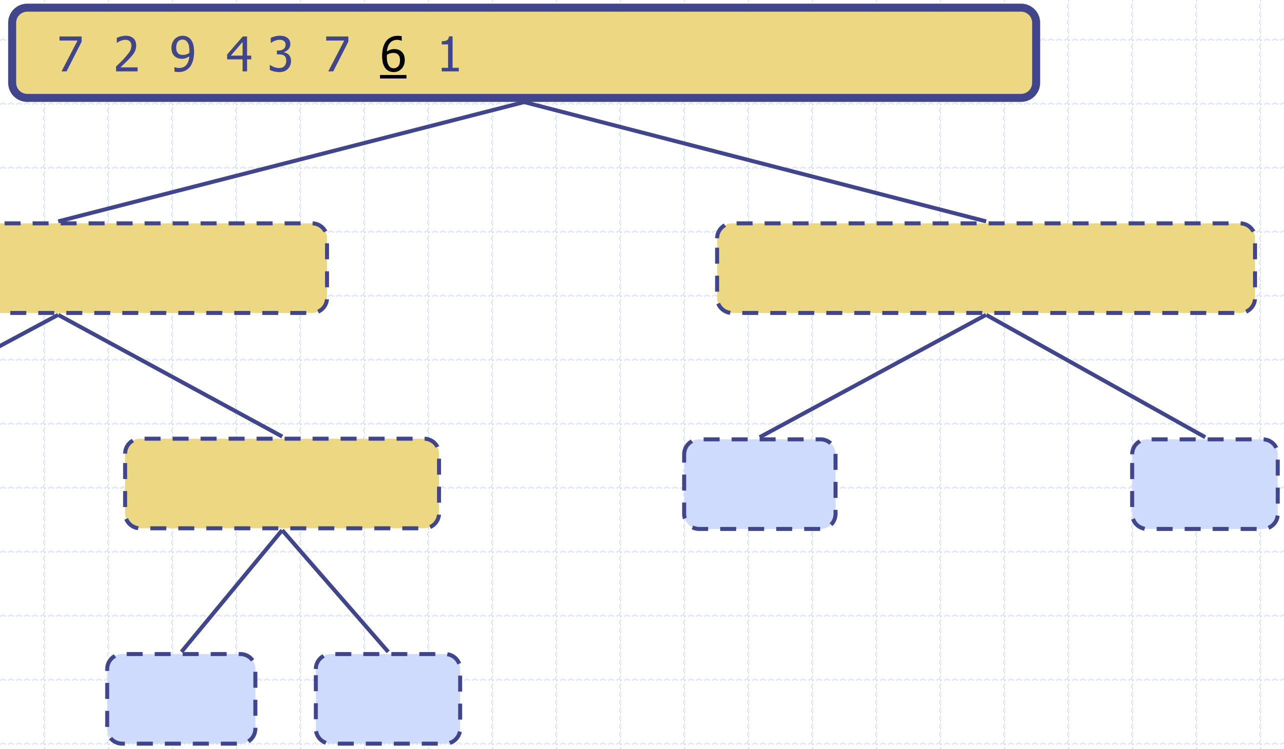
Quick-Sort Tree

- 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



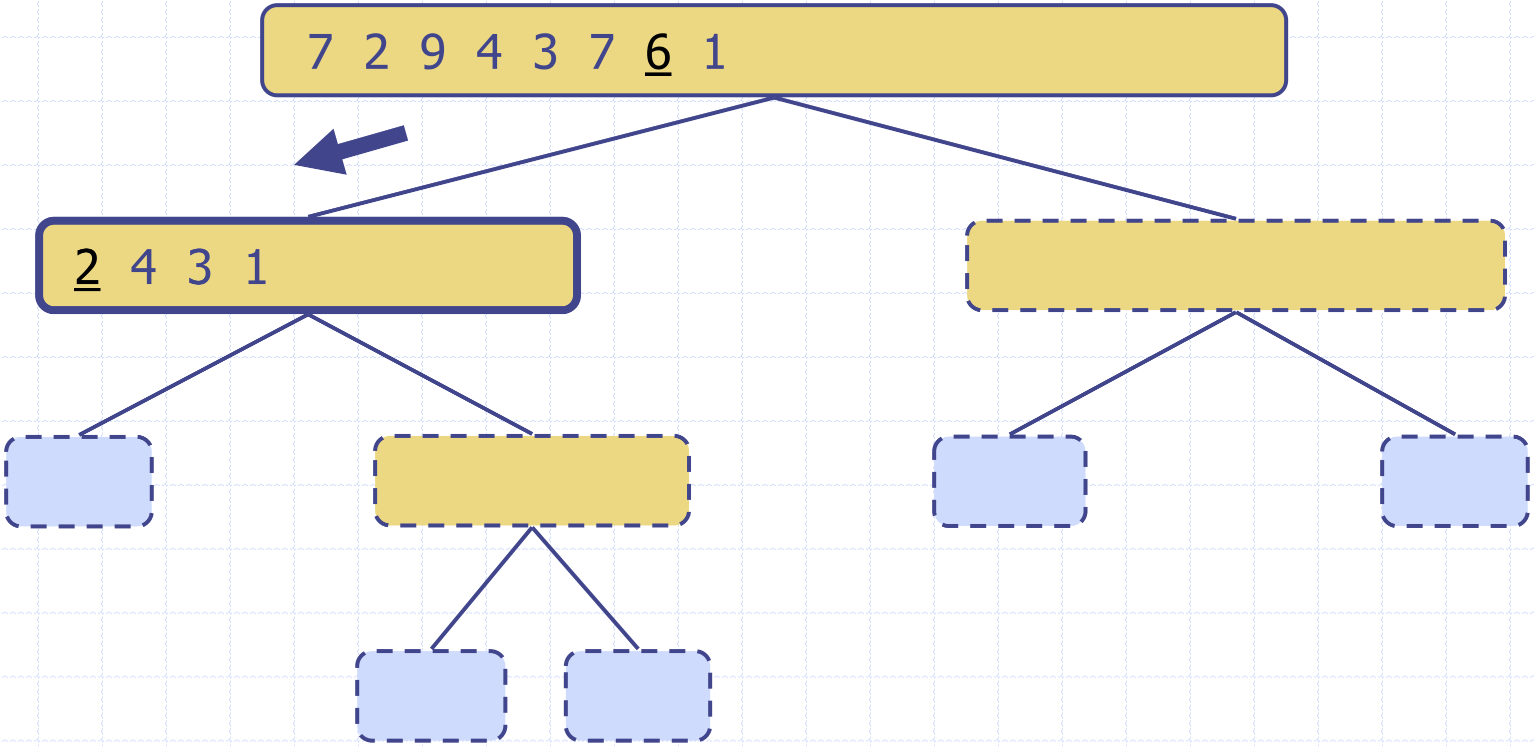
Execution Example

- Pivot selection



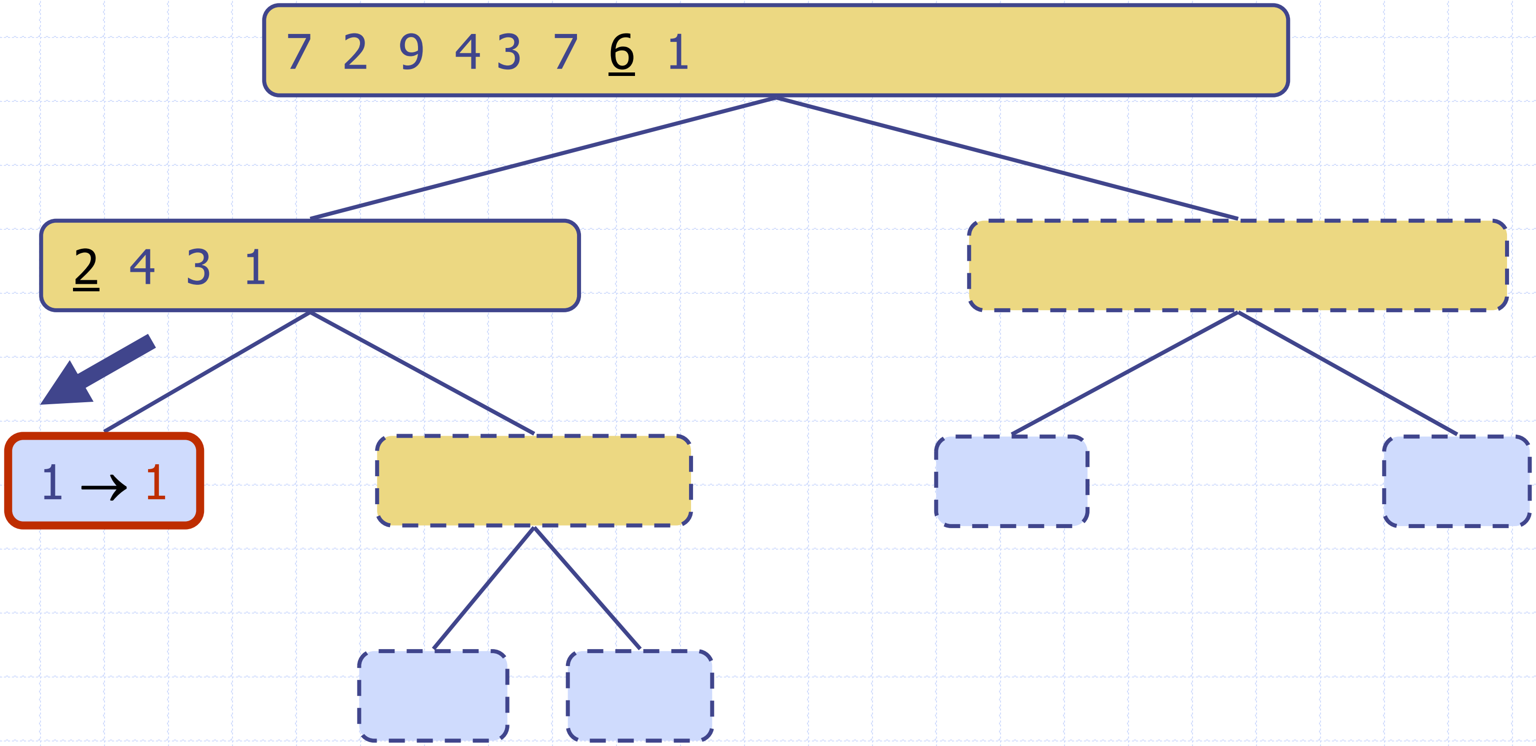
Execution Example (cont.)

- Partition, recursive call, pivot selection



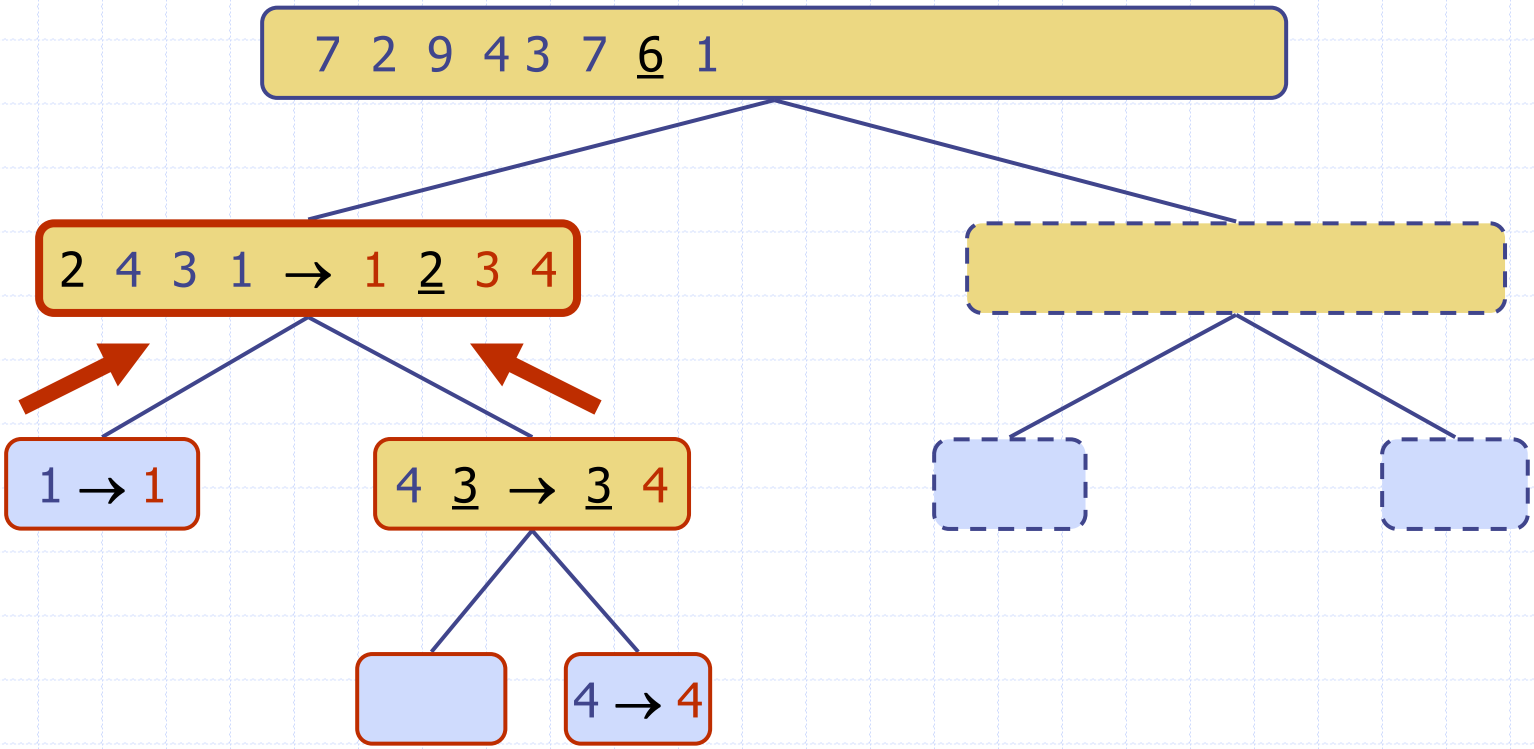
Execution Example (cont.)

- Partition, recursive call, base case



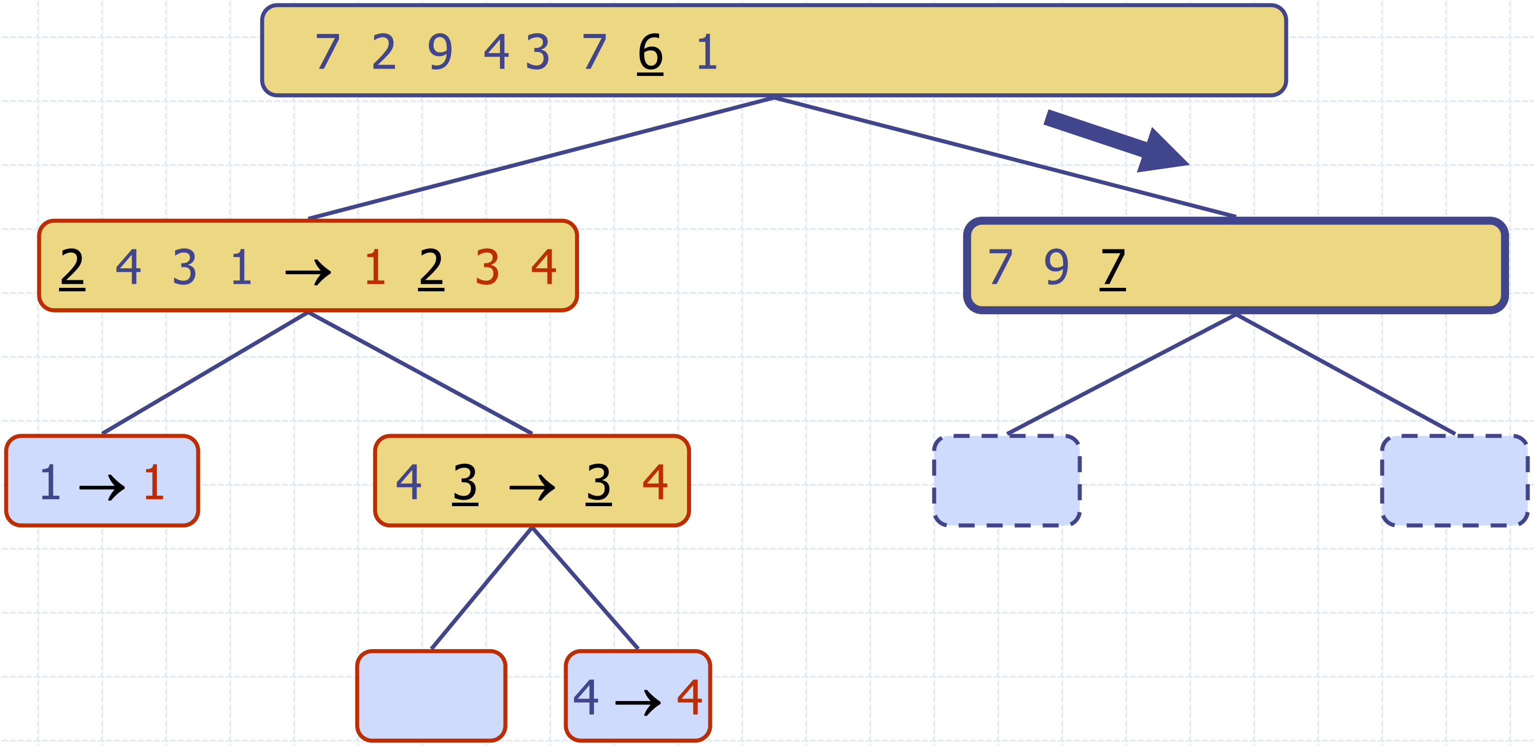
Execution Example (cont.)

- Recursive call, …, base case, join



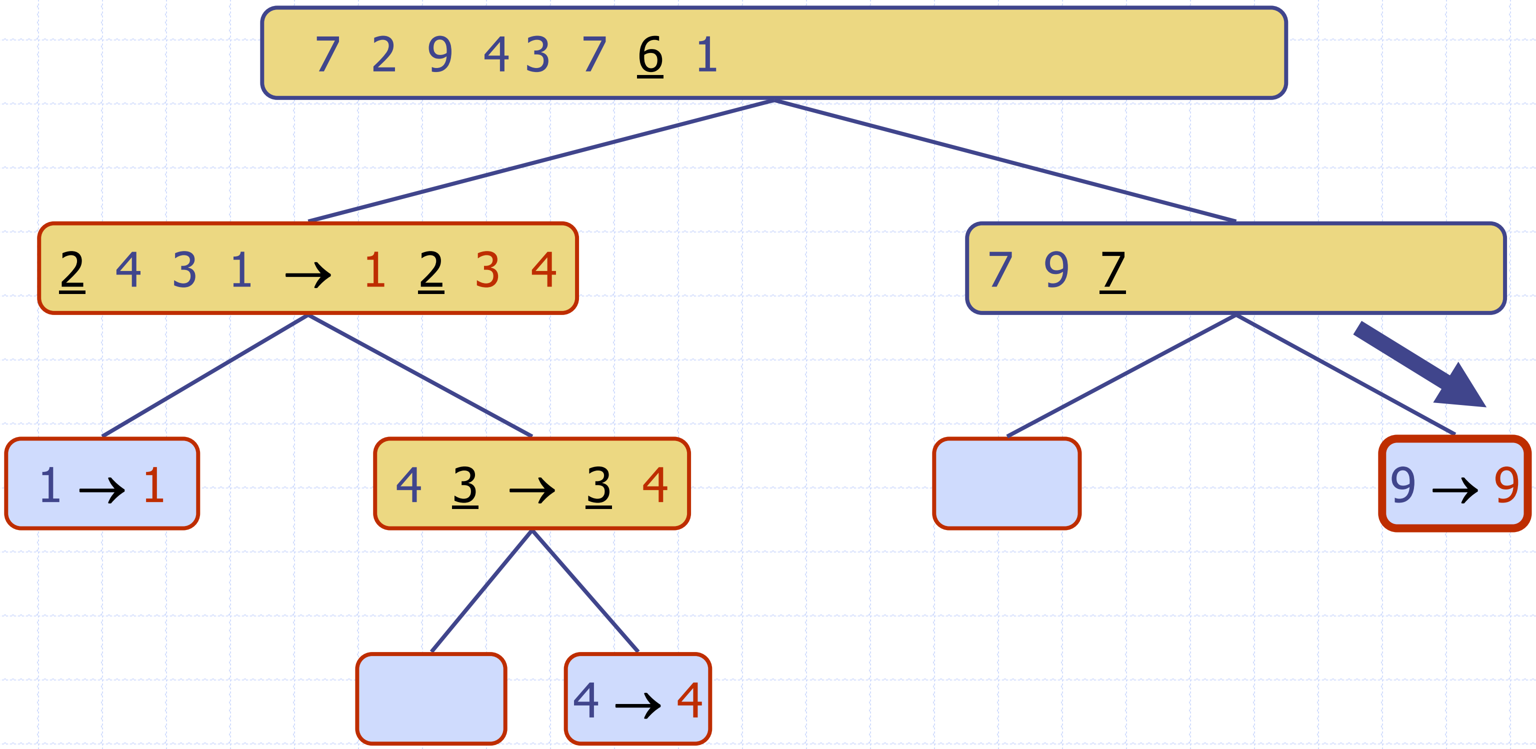
Execution Example (cont.)

- Recursive call, pivot selection



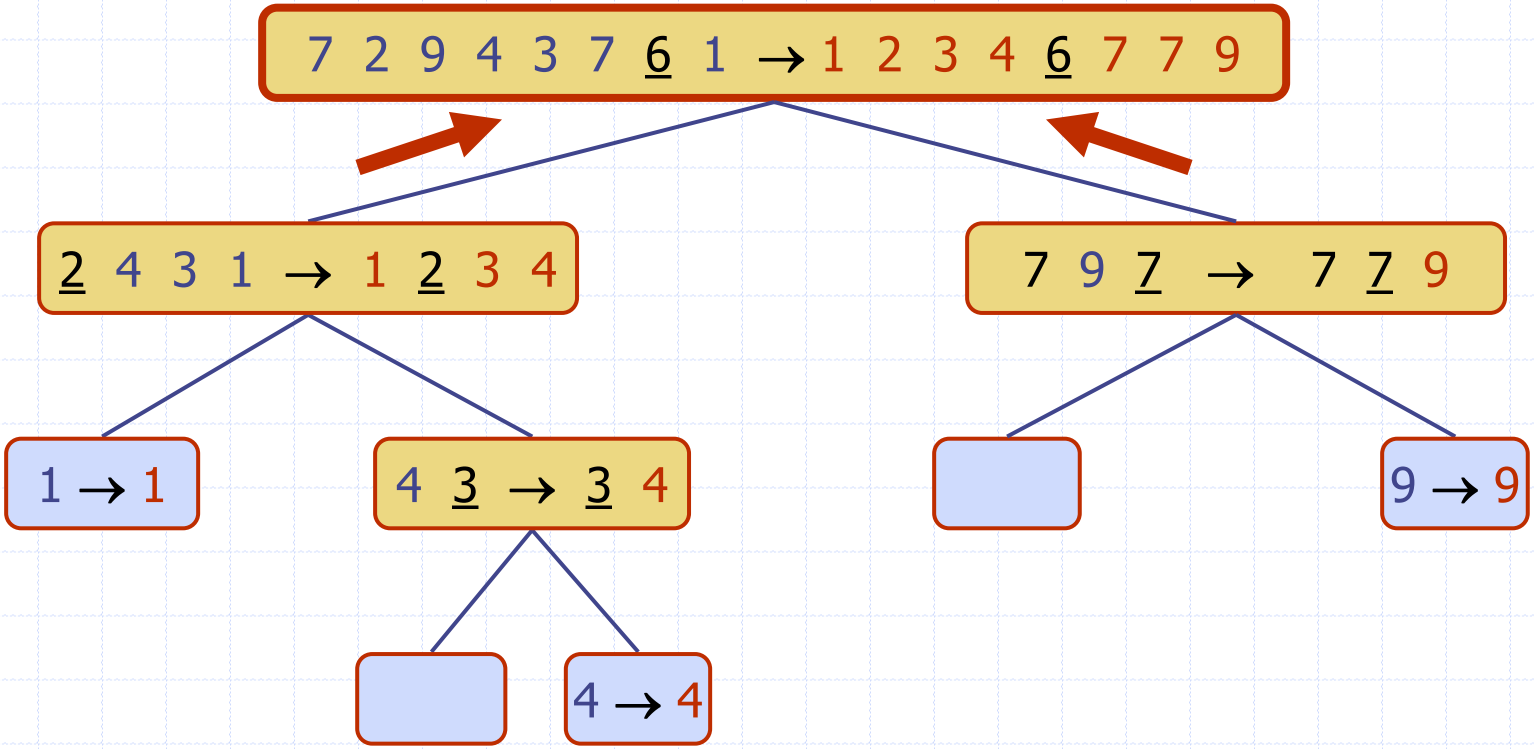
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



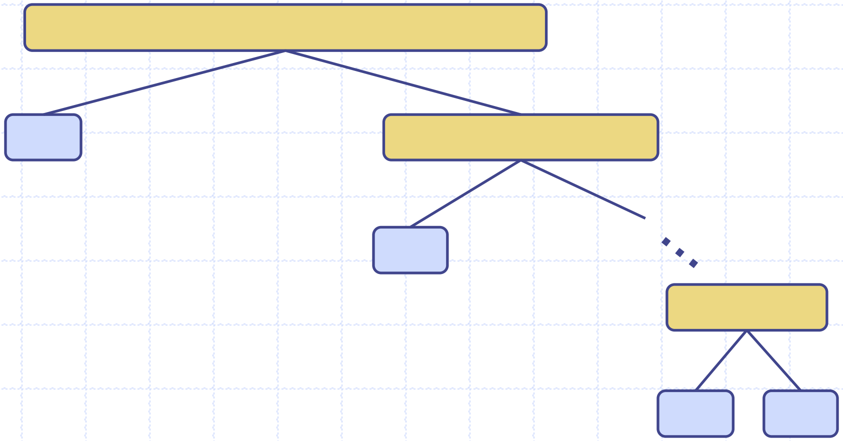
Worst-case Running Time

- 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

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

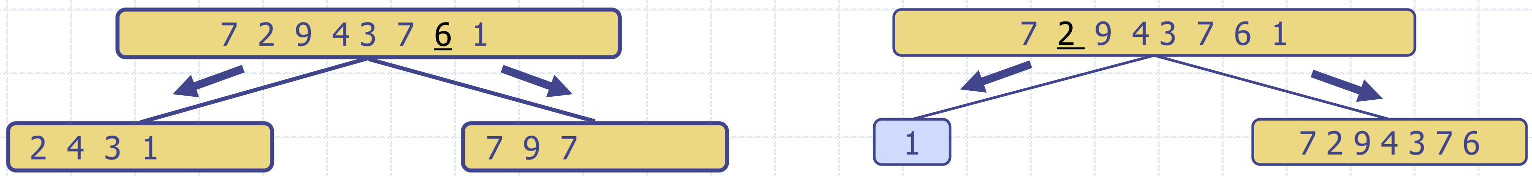
- Thus, the worst-case running time of quick-sort is $O(n^2)$

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



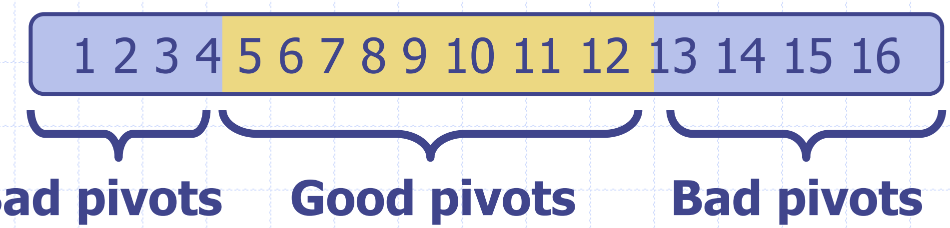
Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are both at least $s/4$
  - **Bad call:** one of L and G has size less than $s/4$



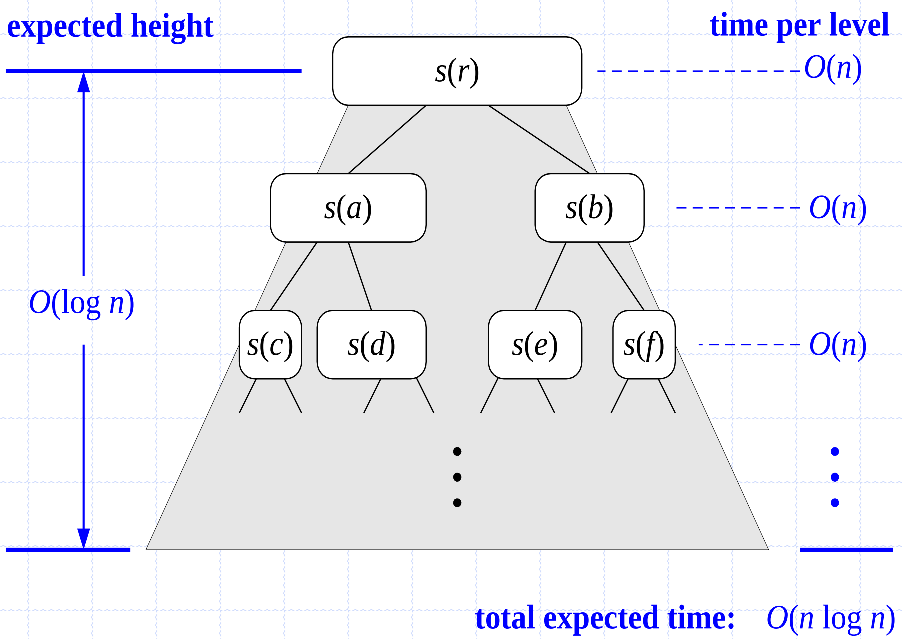
Good call **Bad call**

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

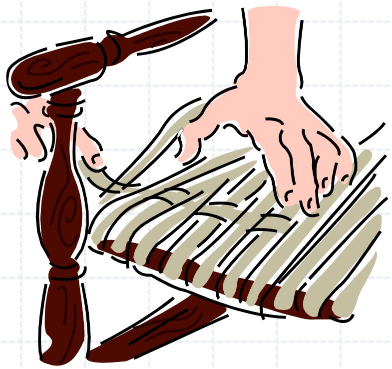


Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get $\boldsymbol{k}$ heads is $2\boldsymbol{k}$
- For a node of depth $\boldsymbol{i}$, we expect
  - $\boldsymbol{i}/2$ ancestors (half) are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{\boldsymbol{i}/2}\boldsymbol{n}$
- Therefore, we have
  - For a node of depth $2\log_{4/3}\boldsymbol{n}$, the expected input size is one
  - The expected height of the quick-sort tree is $\boldsymbol{O}(\log \boldsymbol{n})$
- The amount or work done at the nodes of the same depth is $\boldsymbol{O}(\boldsymbol{n})$
- Thus, the expected running time of quick-sort is $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$



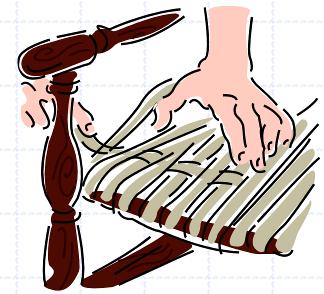
In-Place Quick-Sort



- 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 $l < r$ **then**
        $p \leftarrow$ *inPlacePartition*(S, l, r)
        inPlaceQuickSort(S, l, $p - 1$)
        inPlaceQuickSort(S, $p + 1$, 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

<u>**6**</u> 2 5 1 0 7 3 5 9 2 7 9 8 9 7 8 9 (pivot = 6)

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

j k

<u>**6**</u> 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 8 9

In Place Version of Partition

```
Algorithm inPlacePartition(S, lo, hi)
    Input Sequence S and ranks lo and hi, 0 ≤ lo ≤ hi < S.size()
    Output the pivot is now stored at its sorted rank

    p ← a random integer between lo and hi
    S.swapElements(S.atRank( lo ), S.atRank( p ))
    pivot ← S.elemAtRank(lo)
    j ← lo + 1
    k ← hi
    while  j ≤ k  do
        while  k ≥ j   ∧  S.elemAtRank( k ) ≥  pivot   do
            k ←  k – 1
        while  j ≤  k   ∧  S.elemAtRank( j ) ≤  pivot   do
            j ←  j + 1

        if  j < k  then
            S.swapElements(S.atRank( j ), S.atRank( k ))

    S.swapElements(S.atRank( lo ), S.atRank( k )) {move pivot to sorted rank}
    return k
```

Main Point

3. In Quicksort, the pivot key is the focal point and controls the whole of the sorting process; after being used to partition the input into two smaller subsequences, the pivot is placed in its sorted location and these two subsequences are recursively sorted.
   Science of Consciousness: The ability to maintain broad awareness and sharp focus is cultured through regular practice of the TM technique.

Summary of Sorting Algorithms

Algorithm	Time	Notes
insertion-sort	$O(n^2)$	◆ in-place ◆ slow (good for small inputs)
PQ-sort	$O(n \log n)$	◆ NOT in-place ◆ fast (good for large inputs)
quick-sort	$O(n \log n)$ expected	◆ in-place, randomized ◆ fastest (locality of reference, good for large inputs)
heap-sort	$O(n \log n)$	◆ in-place ◆ fast (fewest key compares)
merge-sort	$O(n \log n)$	◆ sequential data access ◆ fast (good for huge inputs)

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Divide-and-conquer sorting algorithms split the input into subsequences that have to be sorted separately; then the sorted subsequences are recombined until the original input has been sorted.

2. The power of divide-and-conquer sorting algorithms derives from the fact that the input is split in an orderly way into smaller problems so the recombining can be done efficiently and effectively.

3. **<u>Transcendental Consciousness</u>** is the unbounded, silent field of unity, the basis of diversity.
4. **<u>Impulses within Transcendental Consciousness</u>**: The dynamism within this field create and maintain the order in creation with unbounded efficiency.
5. **<u>Wholeness moving within itself</u>:** In Unity Consciousness, the diversity of creation is experienced as waves of intelligence, perfectly orderly fluctuations of one's own self-referral consciousness.