RECALL DIFFERENT SORTING TECHNIQUES SO FAR

1. Heap sort : $O(n \log n)$
2. Insertion sort
3. selection sort

} $O(n^2)$ used Priority Queue

4. In-order traversal on binary search tree / AVL tree

Binary search tree	AVL tree
Construction = $O(n^2)$	Construction = $O(n \log n)$
Traversal = $O(n)$	Traversal = $O(n)$

④ also gives you first or last k sorted elements

Priority Queue Sorting (§ 7.1.4)

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(*S*, *C*)
Input sequence *S*, comparator *C* for the elements of *S*
Output sequence *S* sorted in increasing order according to *C*
P ← priority queue with comparator *C*
while ¬*S.isEmpty* ()
e ← *S.removeFirst* ()
P.insert (*e*, *0*)
while ¬*P.isEmpty*()
e ← *P.removeMin*().*key*()
S.insertLast(*e*)

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

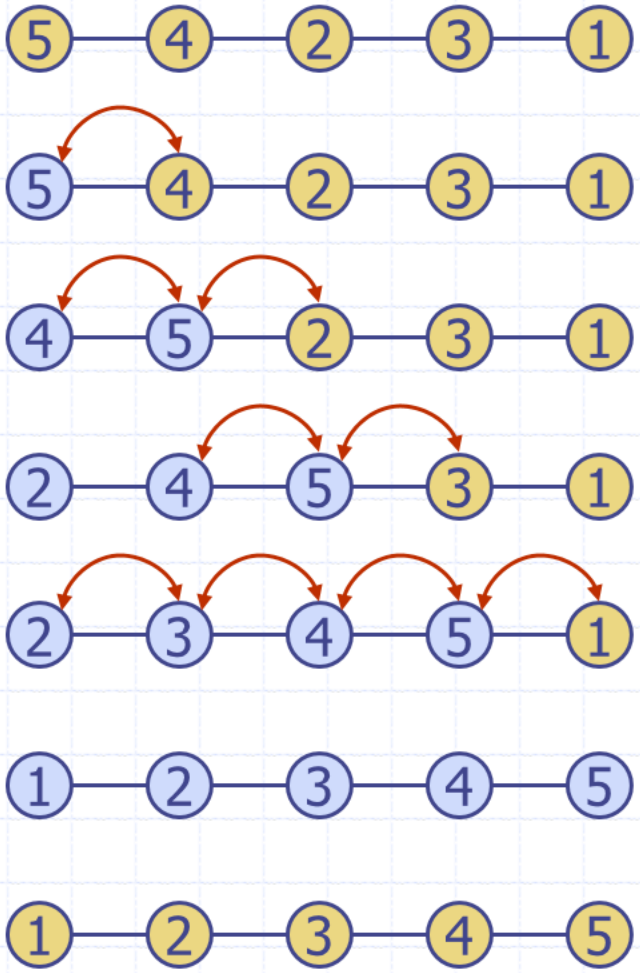
- Selection-sort runs in $O(n^2)$ time

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to
     $$1 + 2 + \ldots + n$$
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

In-place Insertion-sort (w/o PQ)

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



Heap-Sort (§2.4.4)

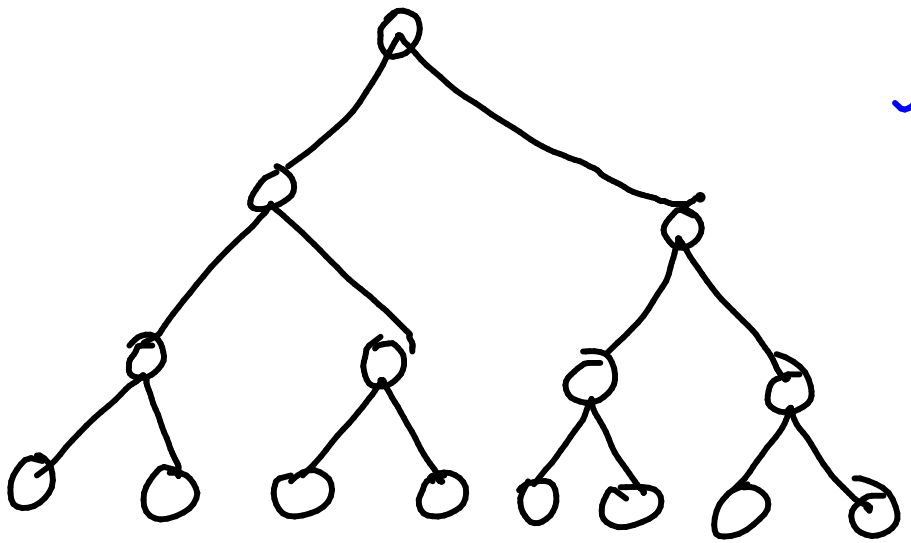
- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, isEmpty, and min take time $O(1)$ time

invoke removeMin n times

- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

H/W question: Understand why heap construction can be done in O(n) time

Bottom-up heap construction



height : # trees

} 3 : 2^0

} 2 : 2^1

} 1 : 2^2

} 0 : 2^3

2^i nodes at level i

$n = 2^{h+1} - 1$

$$\sum_{i=0}^{h} i \times 2^{h-i}$$

$$= 2^h \left[\sum_{i=0}^{h} \frac{i}{2^i} \right]$$

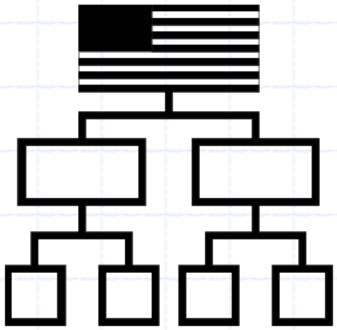
$$= O(2^h) = O(n)$$

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = 2 \quad \text{for } x = 2$$

$$\sum_{i=0}^{\infty} \frac{1}{x^i} = \frac{x}{x-1}$$

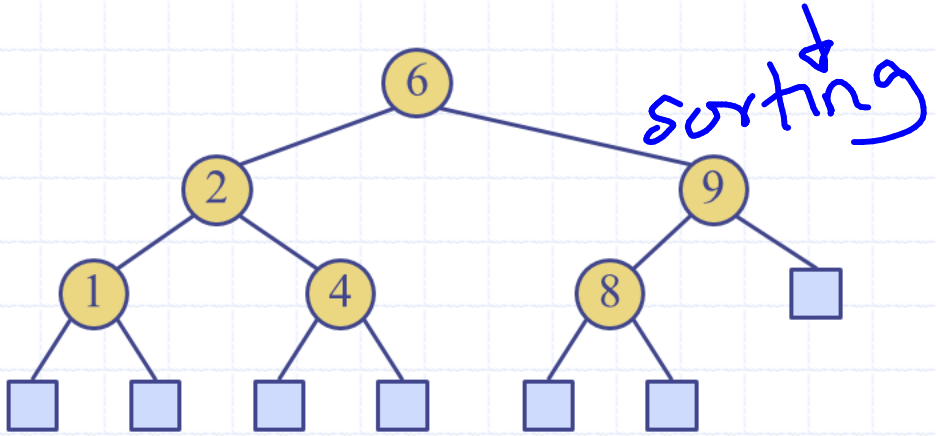
→ take derv.

Binary Search Trees (§ 9.1)

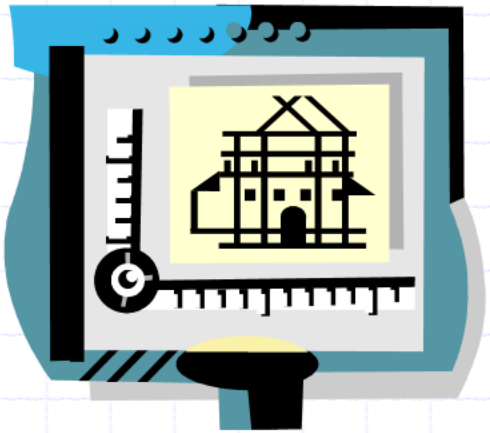


- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items

- An inorder traversal of a binary search trees visits the keys in increasing order



Running Times for AVL Trees

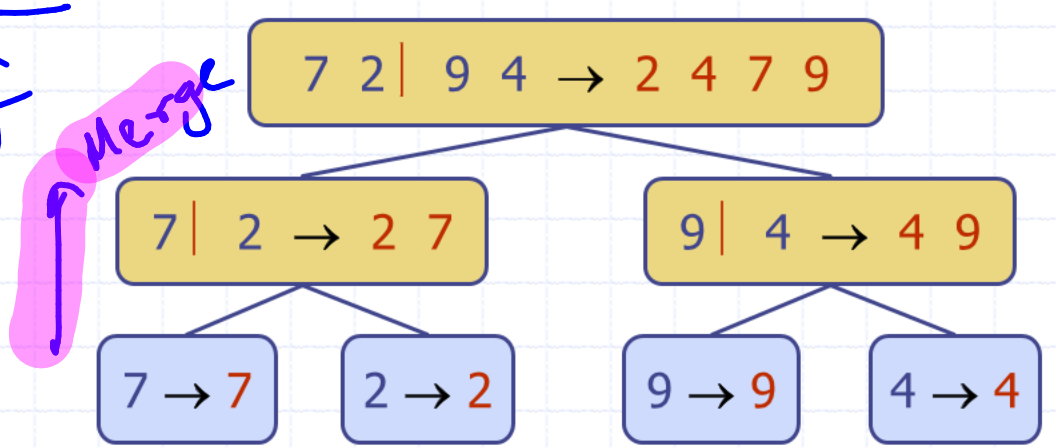


- a single restructure is O(1)
  - using a linked-structure binary tree
- find is O(log n)
  - height of tree is O(log n), no restructures needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- remove is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

∴ Constructing an AVL tree of n elements
$= n \times \log n$
$= O(n \log n)$

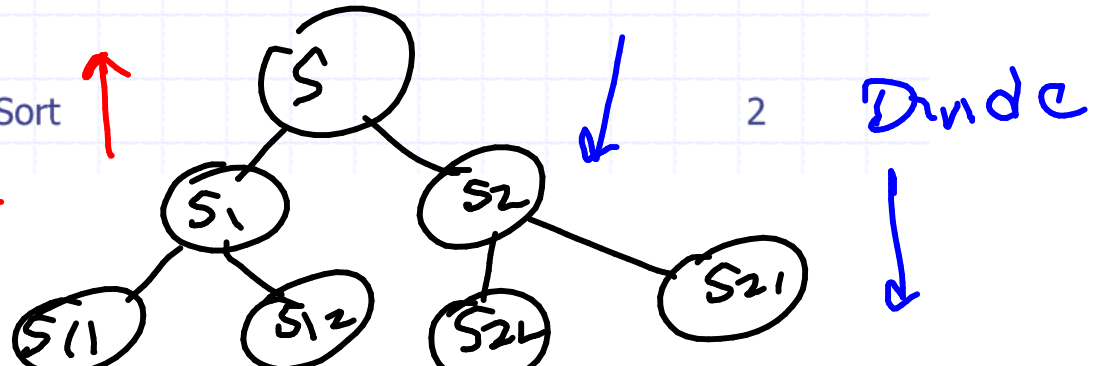
Merge Sort

Generalise idea of sorting using search binary trees



Divide-and-Conquer (§ 10.1.1)

- 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 uses a comparator
  - 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 (§ 10.1)

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

ensures load balancing

S_1 & S_2 already sorted

if $e_{13} < e_{2k}$
$\Rightarrow e_{11}$ & $e_{12} < e_{2k}$
if $e_{1j} < e_{2k}$ $\Rightarrow$ $\forall i<j$, $e_{1i} < e_{2k}$

S_1: $e_{11}\ e_{12}\ \ldots\ e_{1k}$ → ascending

S_2: $e_{21}\ e_{22}\ \ldots\ e_{2k}$ → ascending

Merging Two Sorted Sequences

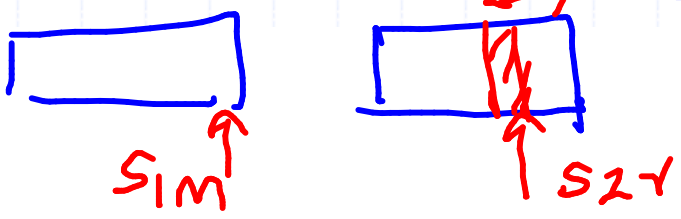
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

```
Algorithm merge(A, B)
    Input sequences A and B with
        n/2 elements each
    Output sorted sequence of A ∪ B

    S ← empty sequence
    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        if A.first().element() < B.first().element()
            S.insertLast(A.remove(A.first()))
        else
            S.insertLast(B.remove(B.first()))
    while ¬A.isEmpty()
        S.insertLast(A.remove(A.first()))
    while ¬B.isEmpty()
        S.insertLast(B.remove(B.first()))
    return S
```

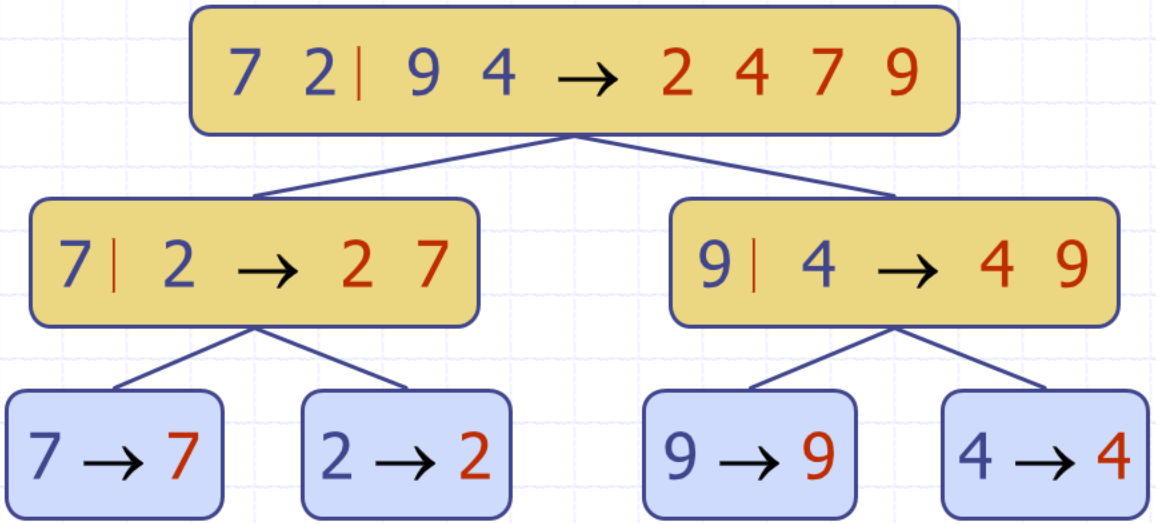
Not necessary that A & B have same length

Handling the excess element after all comparisons over



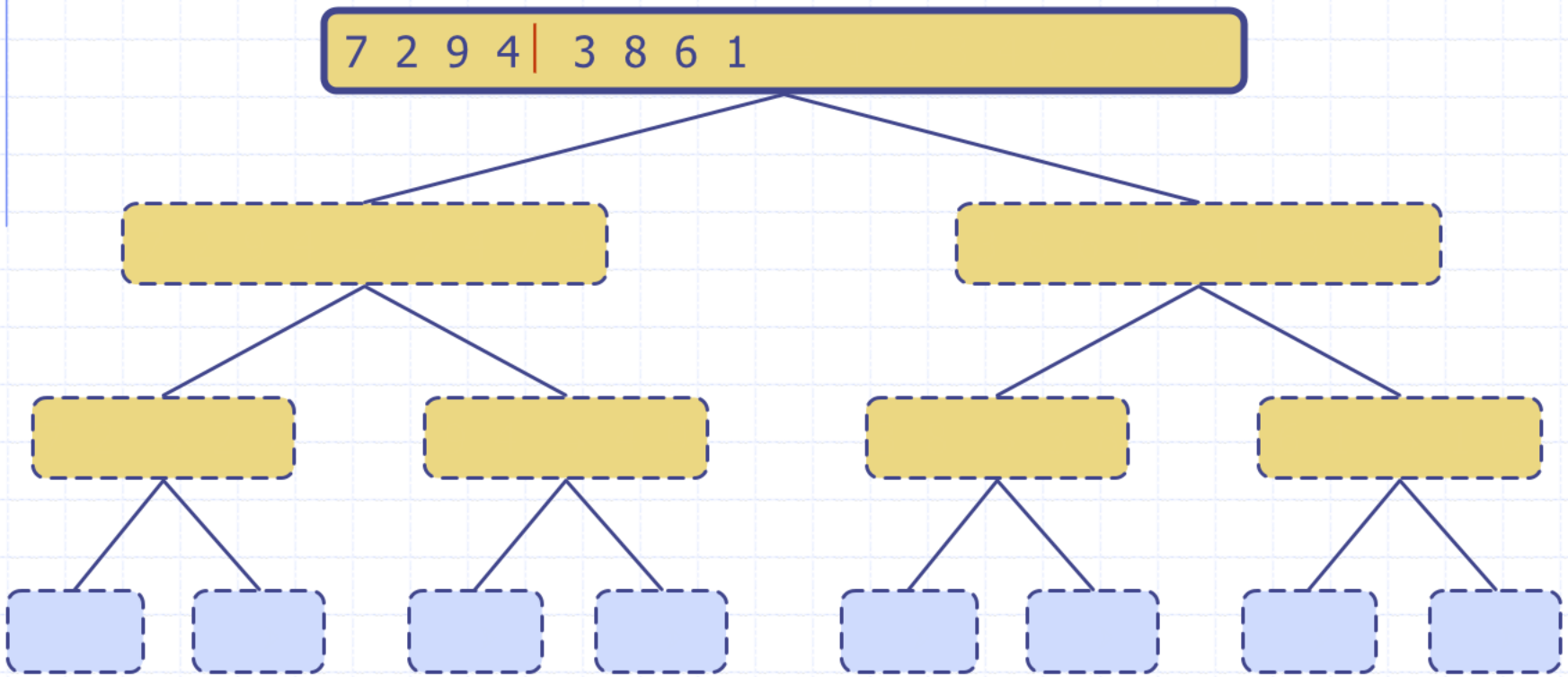
Merge-Sort Tree

- 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



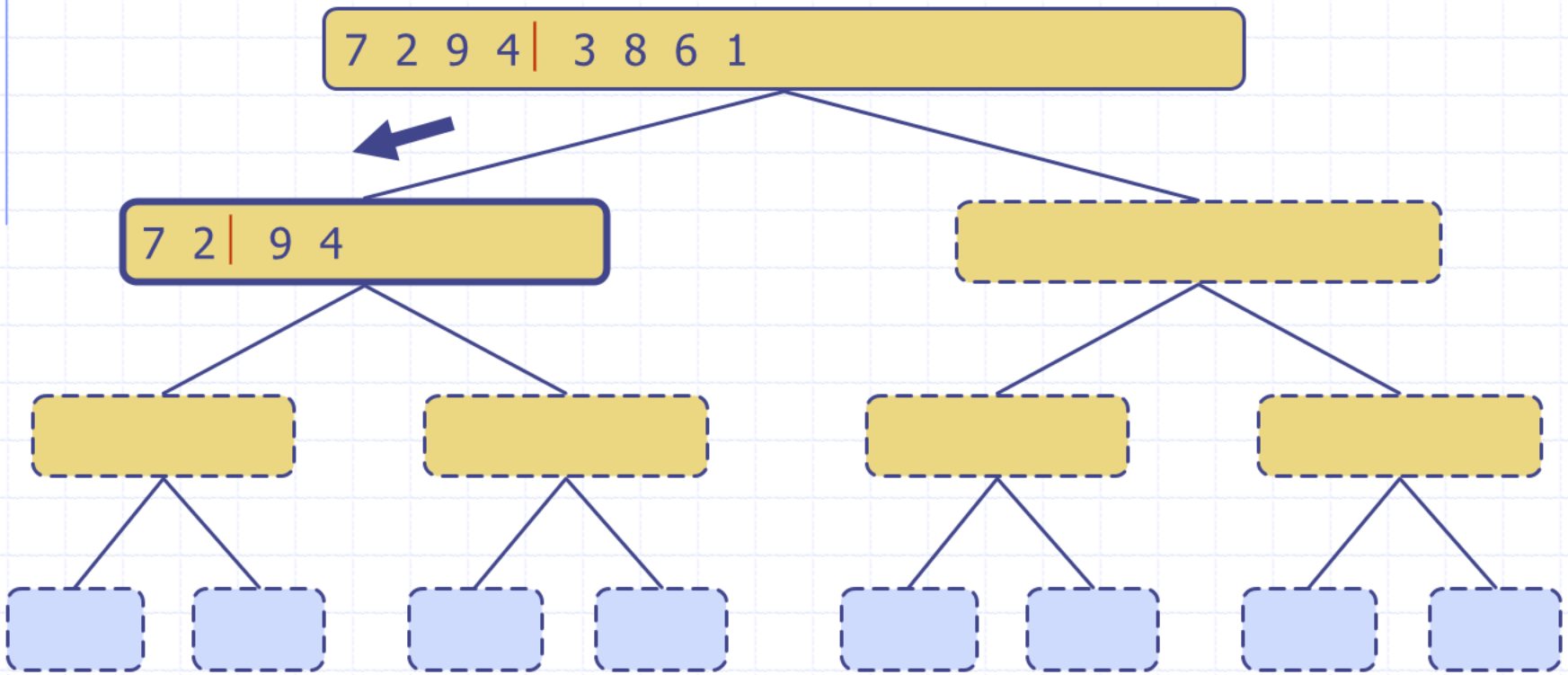
Execution Example

- Partition



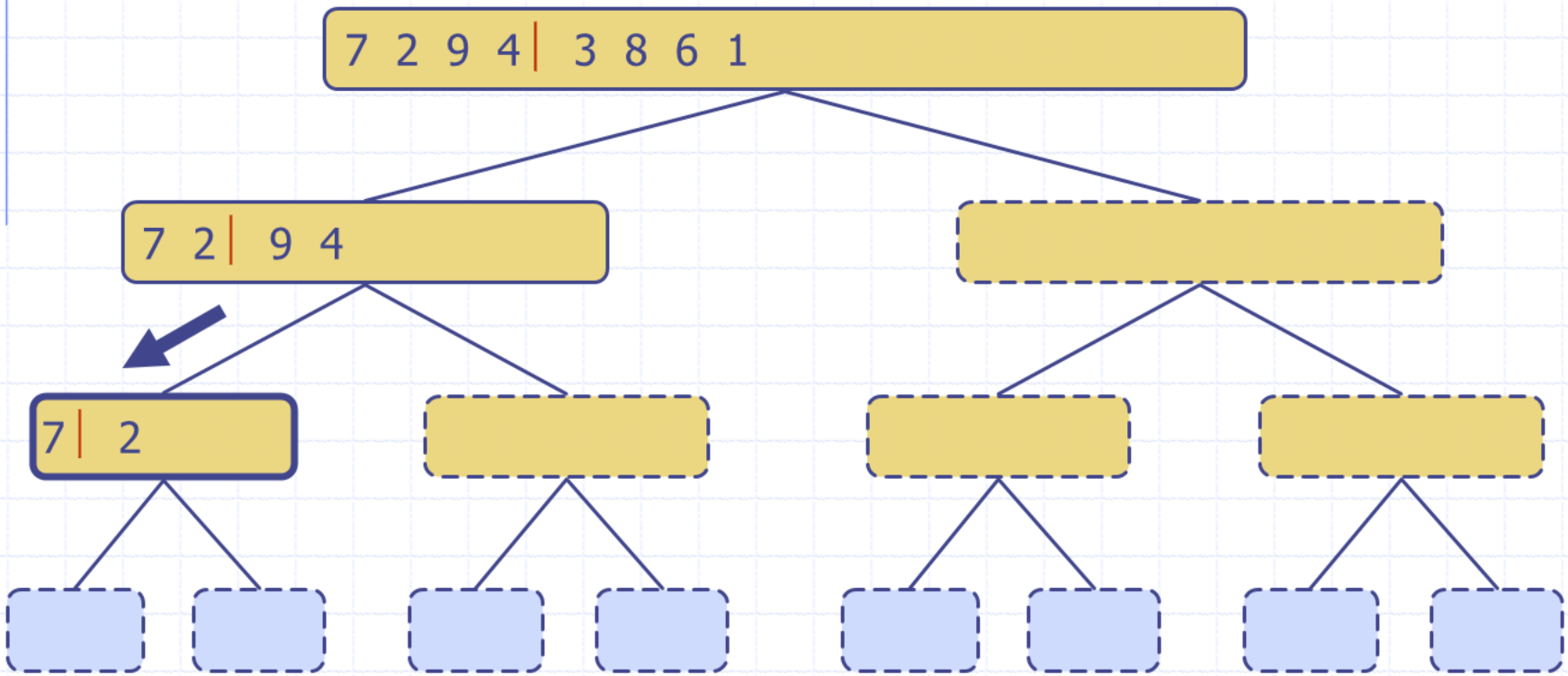
Execution Example (cont.)

- Recursive call, partition



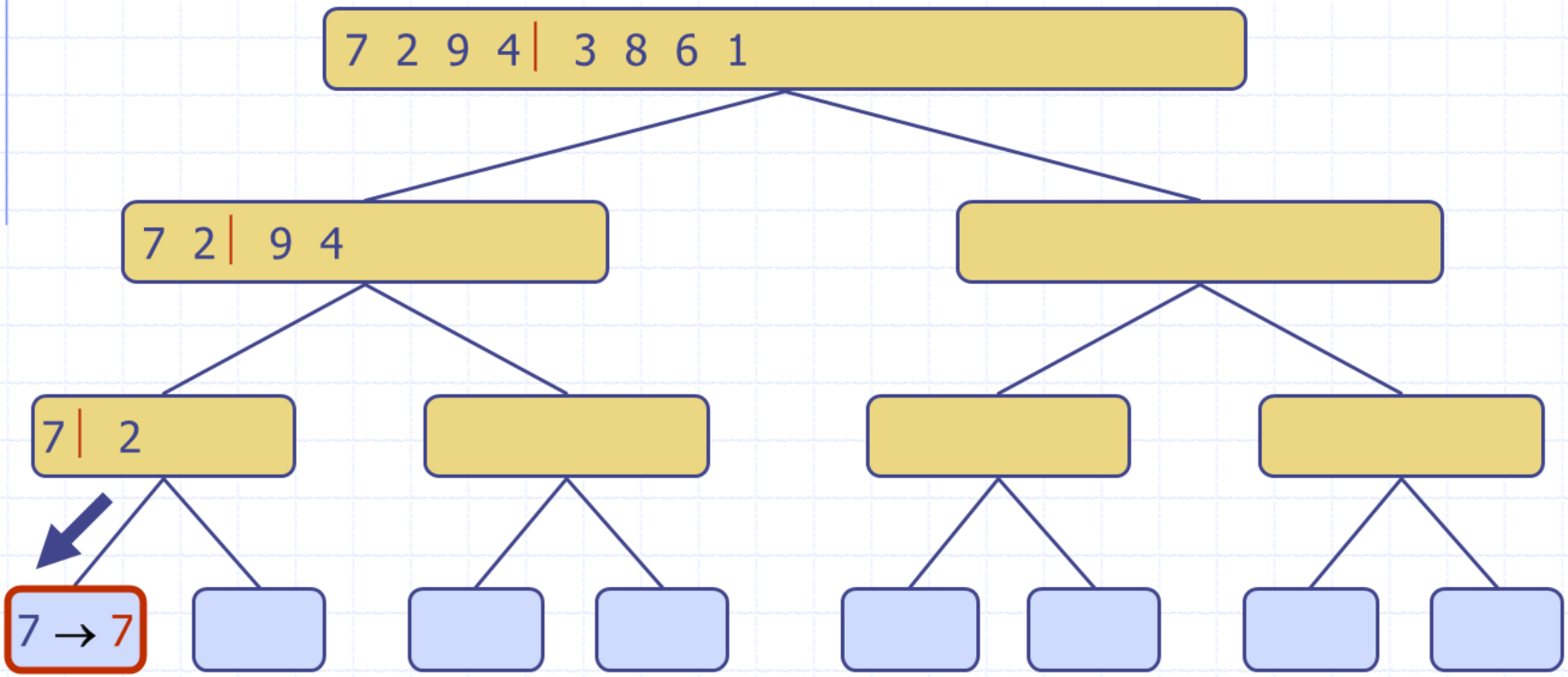
Execution Example (cont.)

- Recursive call, partition



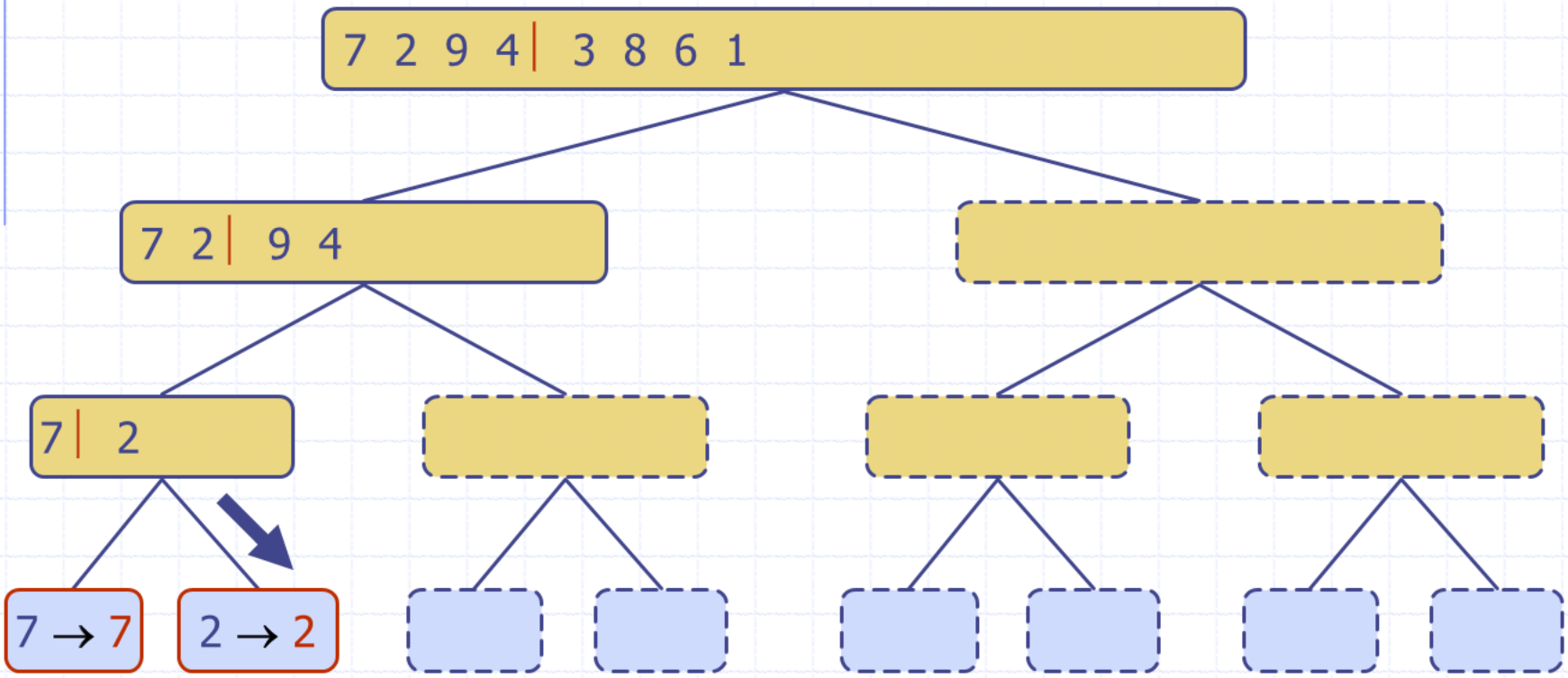
Execution Example (cont.)

- Recursive call, base case



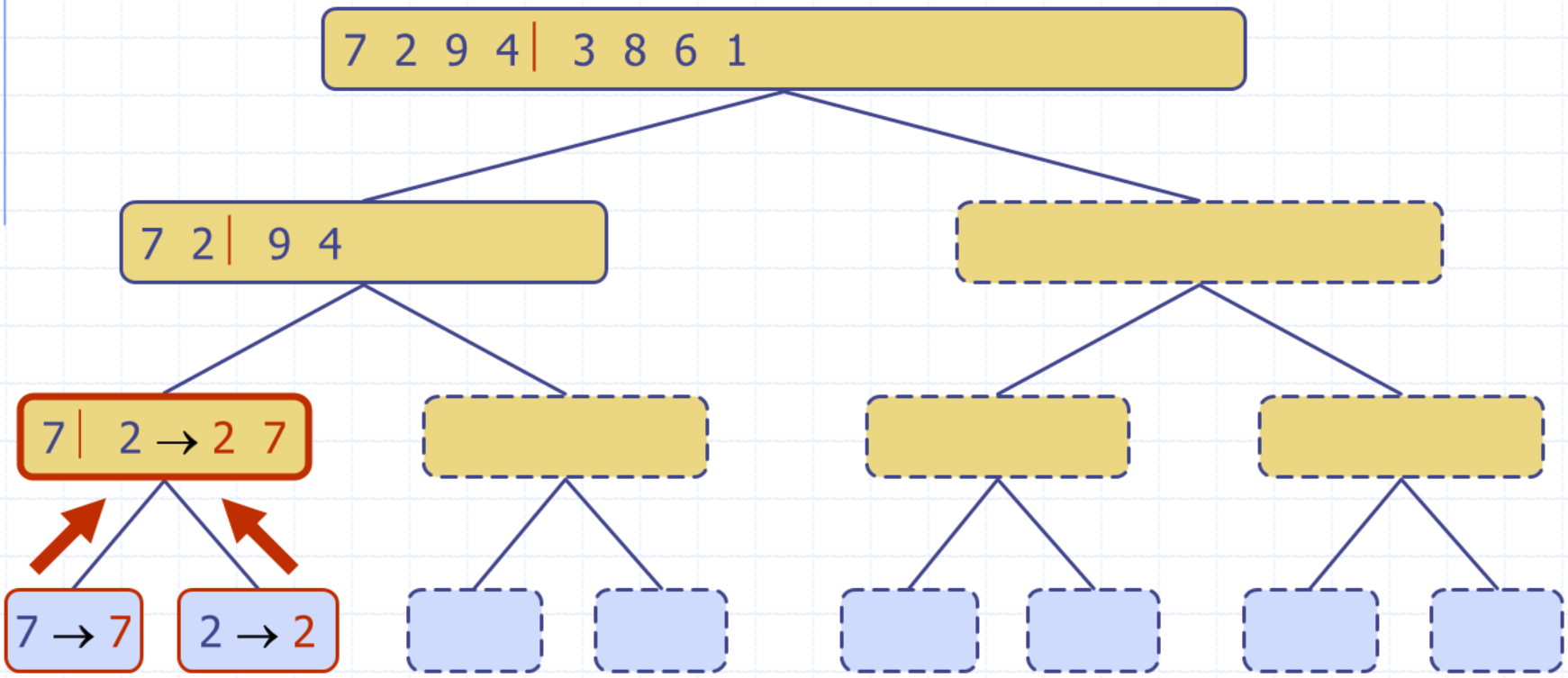
Execution Example (cont.)

- Recursive call, base case



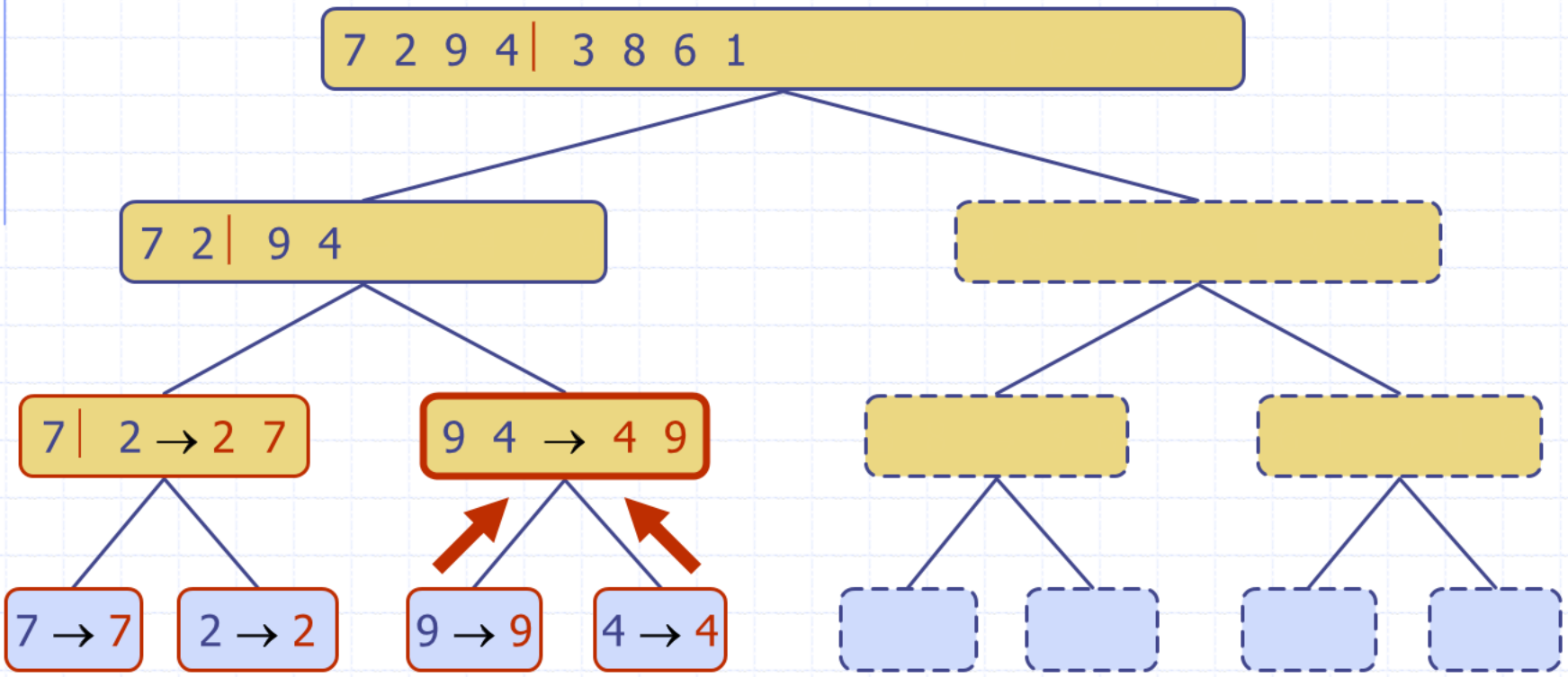
Execution Example (cont.)

- Merge



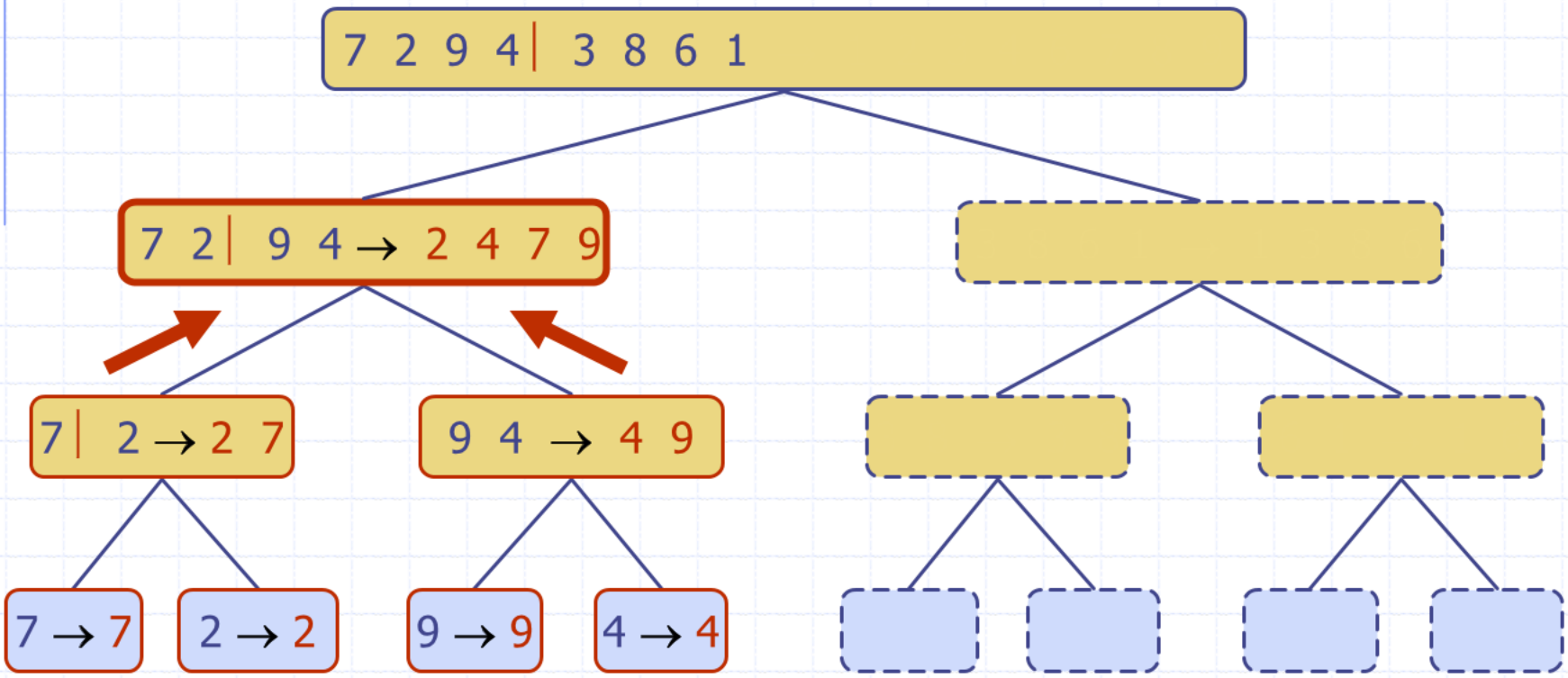
Execution Example (cont.)

- Recursive call, …, base case, merge



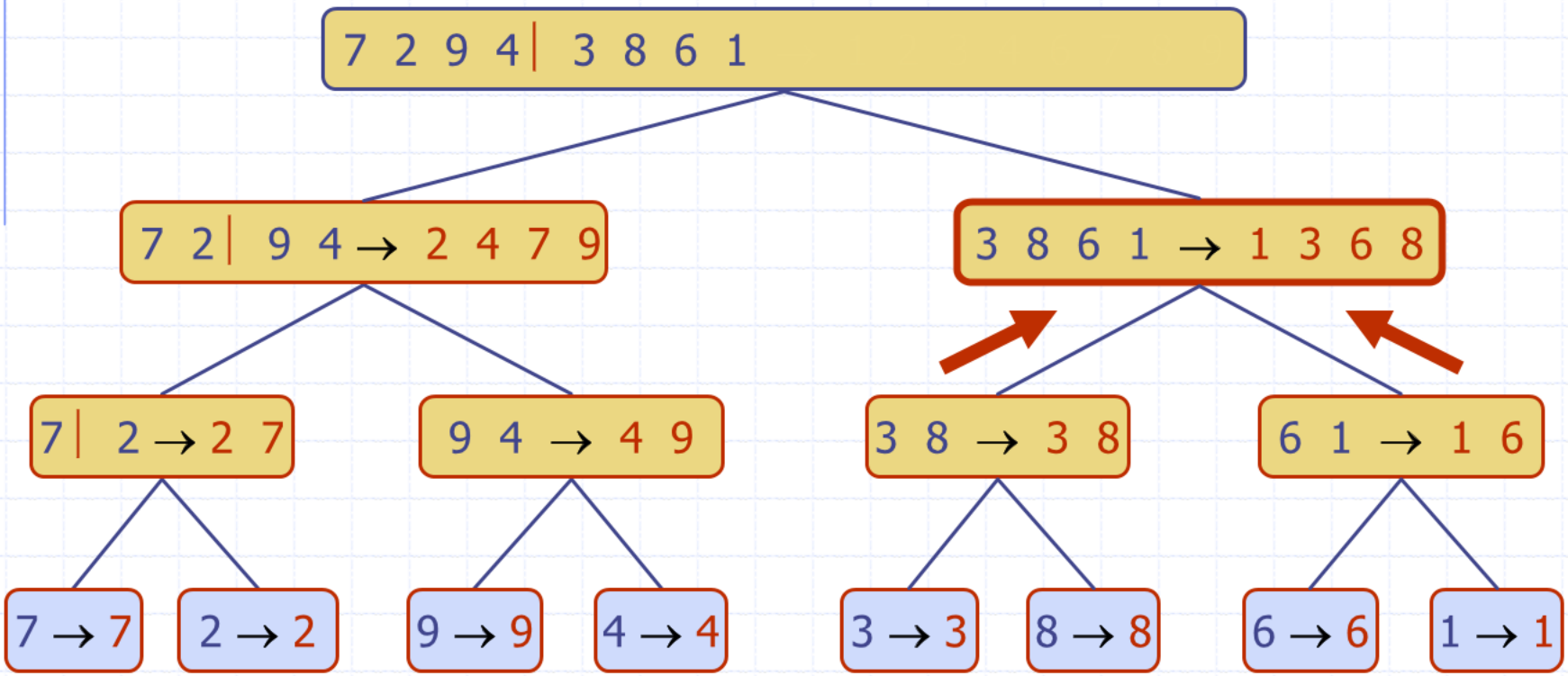
Execution Example (cont.)

- Merge



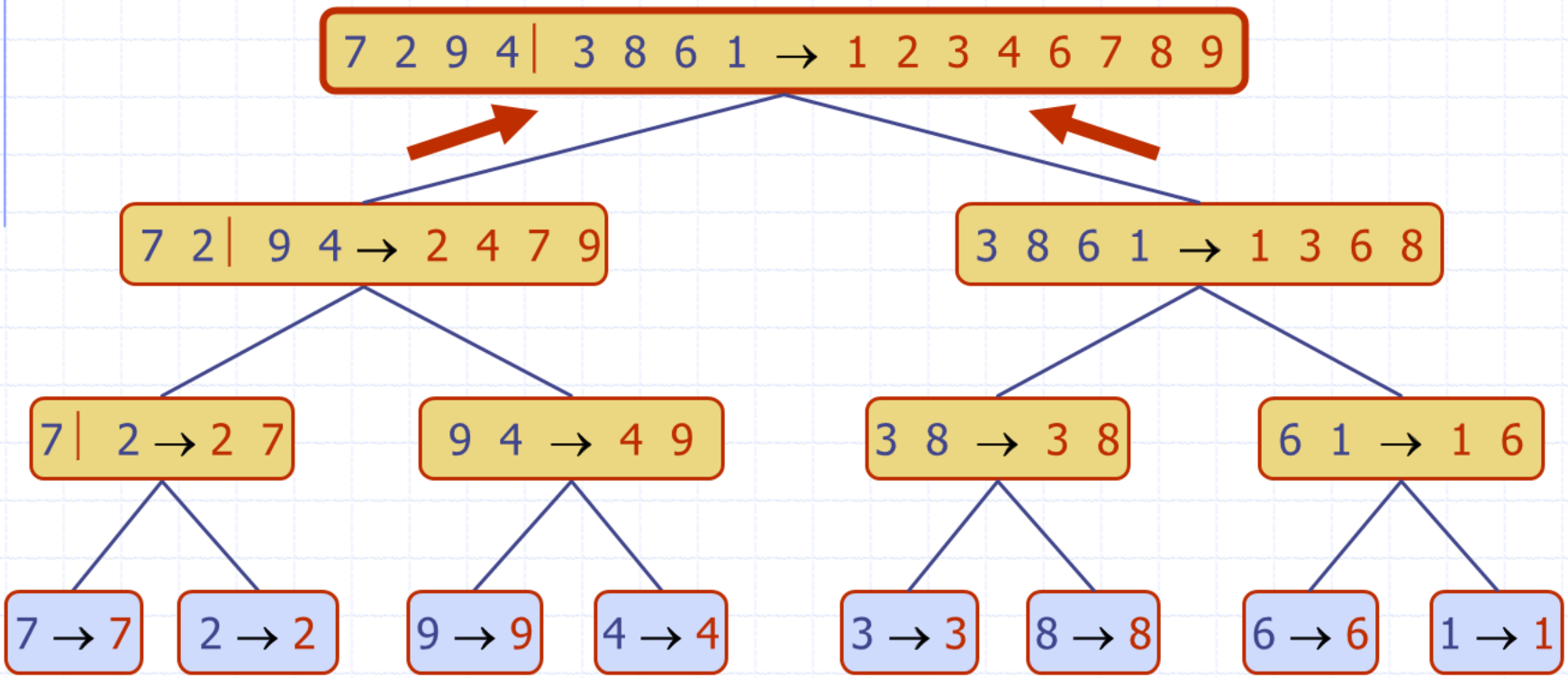
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

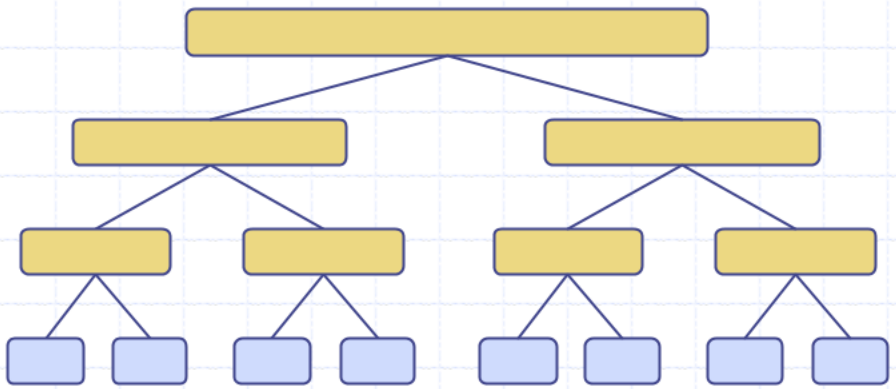
- Merge



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	$n/2^i$
…	…	…



$$\text{Time} = 1 \times n + 2 \times \frac{n}{2} + \cdots + 2^i \, n/2^i = \sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i \frac{n}{2^i} = n \log_2 n$$

Another way of solving:

$$T(n) = 2 * T(n/2) + n = 4 * T(n/4) + 2n$$

$$= 8\,T(n/8) + 3n \ldots = 2^n T(1) + h * n$$

Soln as we saw is $T(n) = n \log n + n$

(Verify)

$$= O(n \log n)$$

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)

} By design you need additional array space

Nonrecursive Merge-Sort

merge runs of length 2, then 4, then 8, and so on

merge two runs in the in array to the out array

```
public static void mergeSort(Object[] orig, Comparator c) { // nonrecursive
  Object[] in = new Object[orig.length]; // make a new temporary array
  System.arraycopy(orig,0,in,0,in.length); // copy the input
  Object[] out = new Object[in.length]; // output array
  Object[] temp; // temp array reference used for swapping
  int n = in.length;
  for (int i=1; i < n; i*=2) { // each iteration sorts all length-2*i runs
    for (int j=0; j < n; j+=2*i)  // each iteration merges two length-i pairs
      merge(in,out,c,j,i); // merge from in to out two length-i runs at j
    temp = in; in = out; out = temp; // swap arrays for next iteration
  }
  // the "in" array contains the sorted array, so re-copy it
  System.arraycopy(in,0,orig,0,in.length);
}
protected static void merge(Object[] in, Object[] out, Comparator c, int start,
    int inc) { // merge in[start..start+inc-1] and in[start+inc..start+2*inc-1]
  int x = start; // index into run #1
  int end1 = Math.min(start+inc, in.length); // boundary for run #1
  int end2 = Math.min(start+2*inc, in.length); // boundary for run #2
  int y = start+inc; // index into run #2 (could be beyond array boundary)
  int z = start; // index into the out array
  while ((x < end1) && (y < end2))
    if (c.compare(in[x],in[y]) <= 0) out[z++] = in[x++];
    else out[z++] = in[y++];
  if (x < end1) // first run didn't finish
    System.arraycopy(in, x, out, z, end1 - x);
  else if (y < end2) // second run didn't finish
    System.arraycopy(in, y, out, z, end2 - y);
}
```

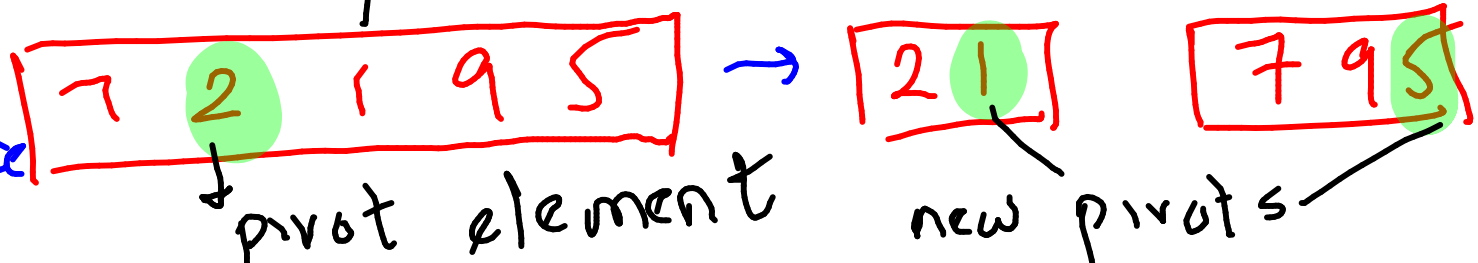

H/w Reading

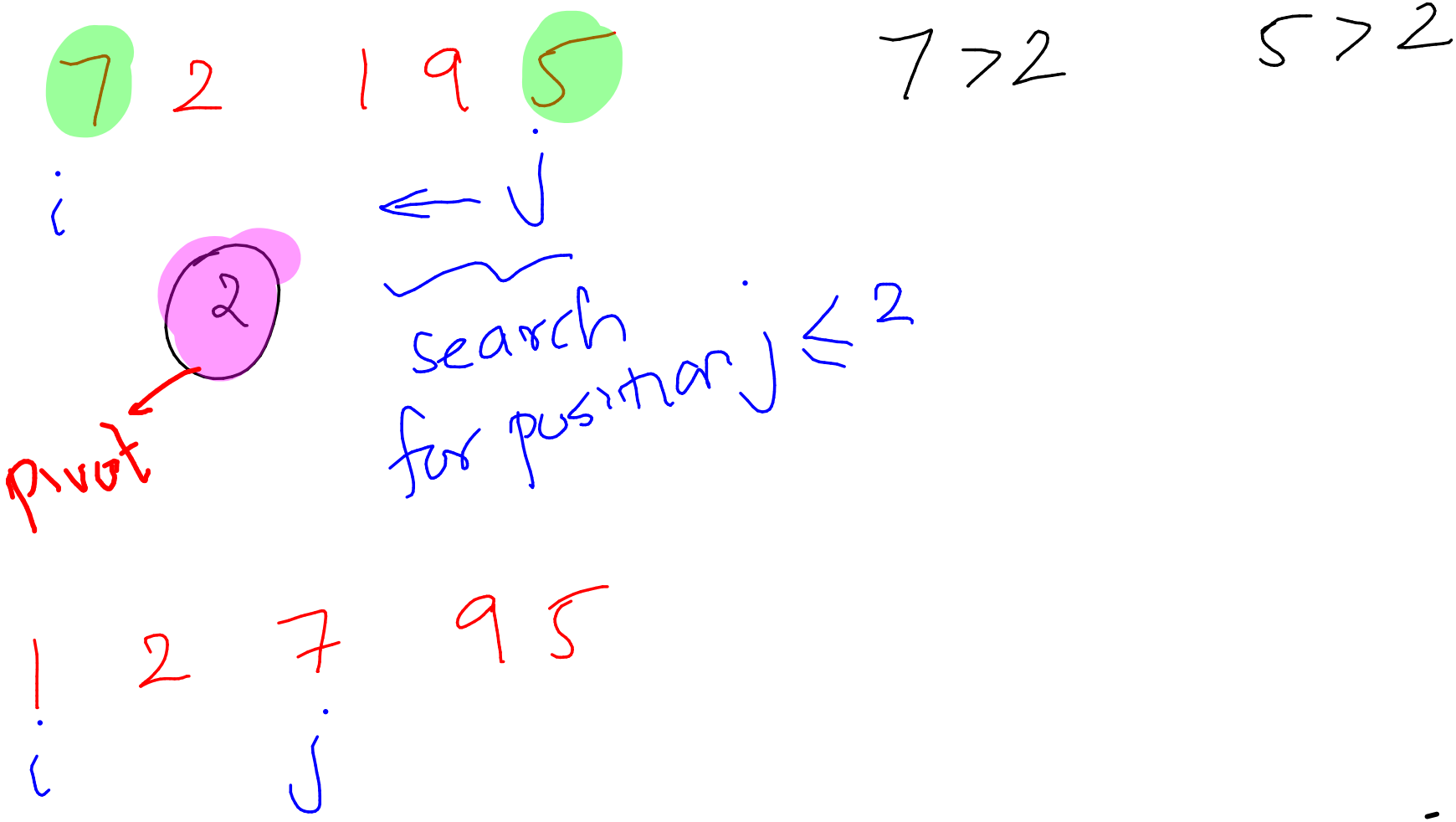
Some next questions:

① Does division have to be "balanced" at every step? ie $n \to n/2 \quad n/2$ is necessary?

② Should the division be based on length only? Can it be based on "pivot"

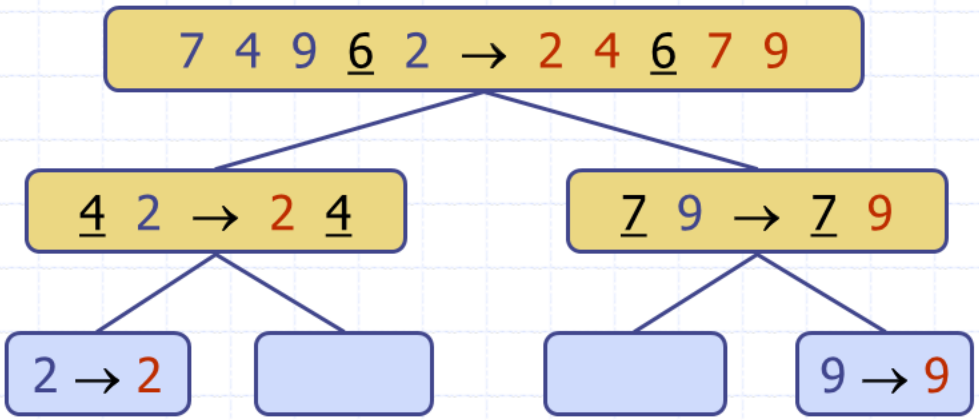
Q: Can I use pivot to do in place divide?





Idea: Move i right if element at j is < pivot & at i is < pivot until element at i becomes > pivot.
Swap elements at i & j
Similarly: Move j left if element at i is > pivot & at j is > pivot until . . .

Quick-Sort



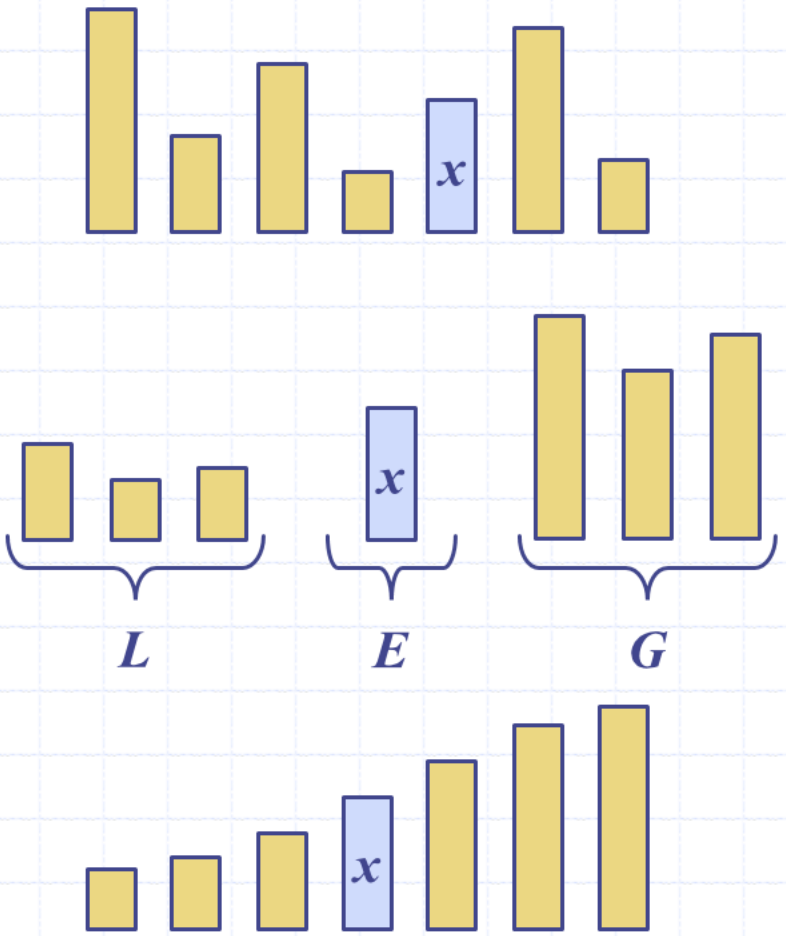
Divide based on pivot

Merge is simply concatenating

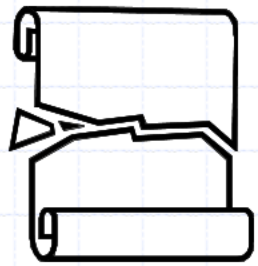
You will not even need a merge step when you divide in-place

Quick-Sort (§ 10.2)

- 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

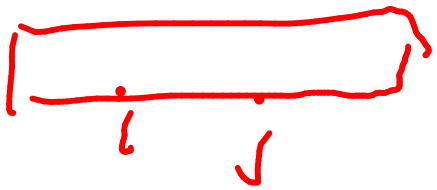
Not an in place version

```
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
```

This is not inplace. Better to use the inplace version we discussed

Array: $A[1..n]$

e = pivot element



$i = 1, j = n$

if $(A[i] > e)$ && $(A[j] > e)$ $j--$

if $(A[i] \leq e)$ && $(A[j] < e)$ $i++$

if $(A[i] > e)$ && $(A[j] \leq e)$ { swap $A[i]$ & $A[j]$

$i++$

$j--$

}

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

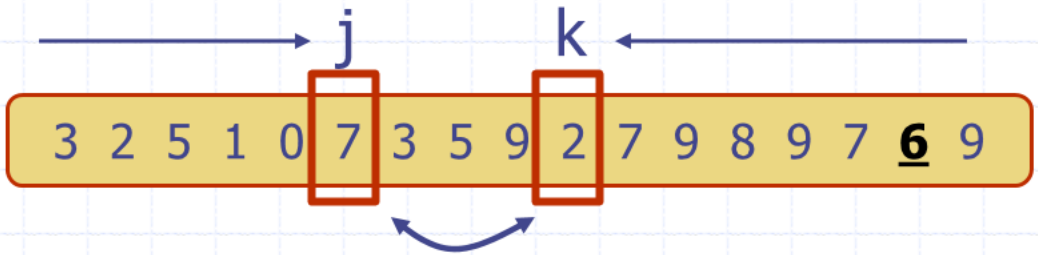
- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

j k

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

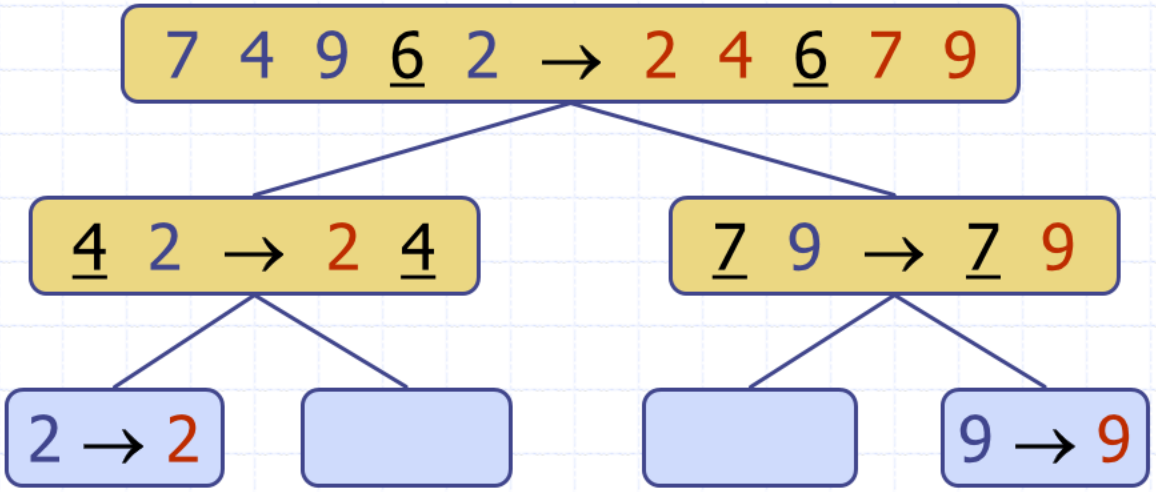
(pivot = 6)

- 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



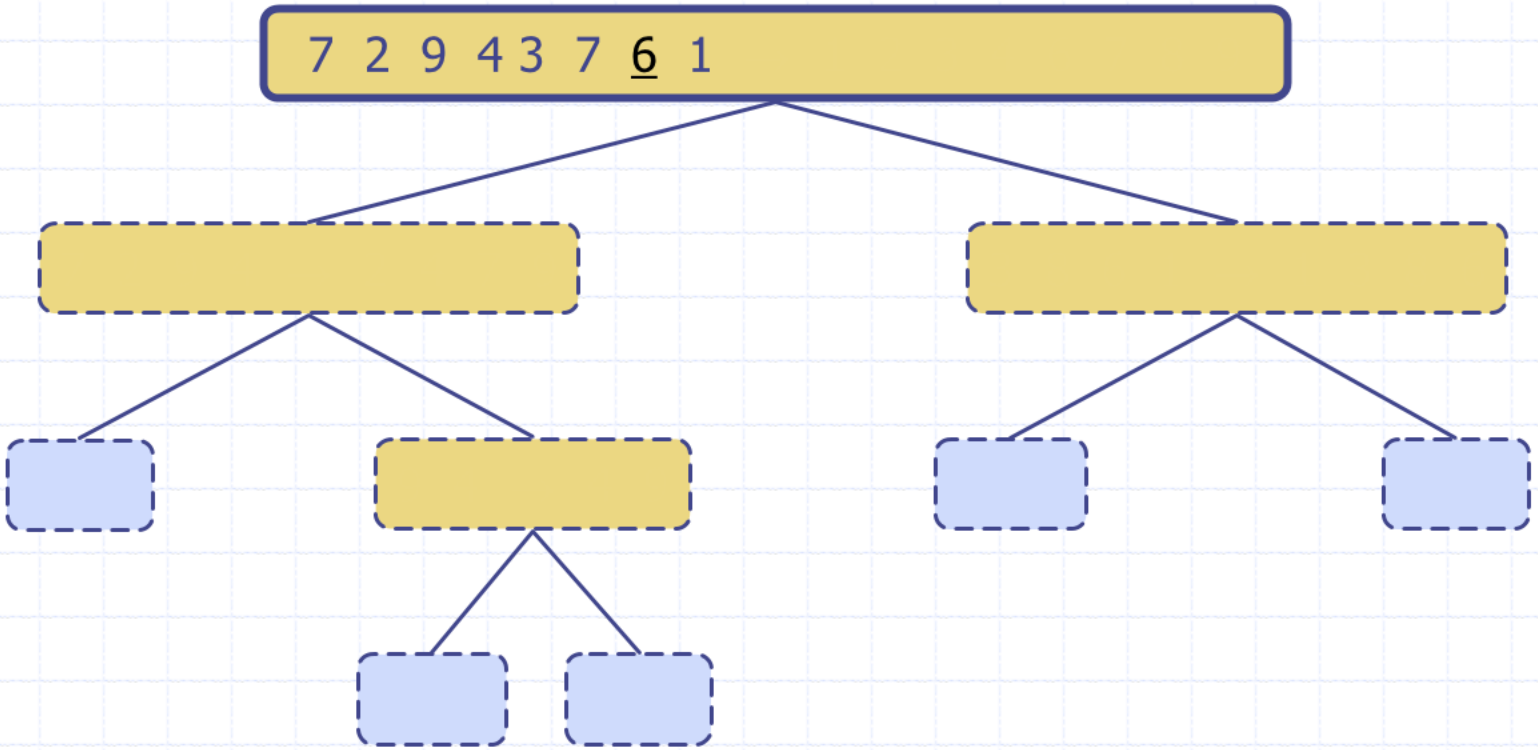
Quick-Sort Tree

- 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



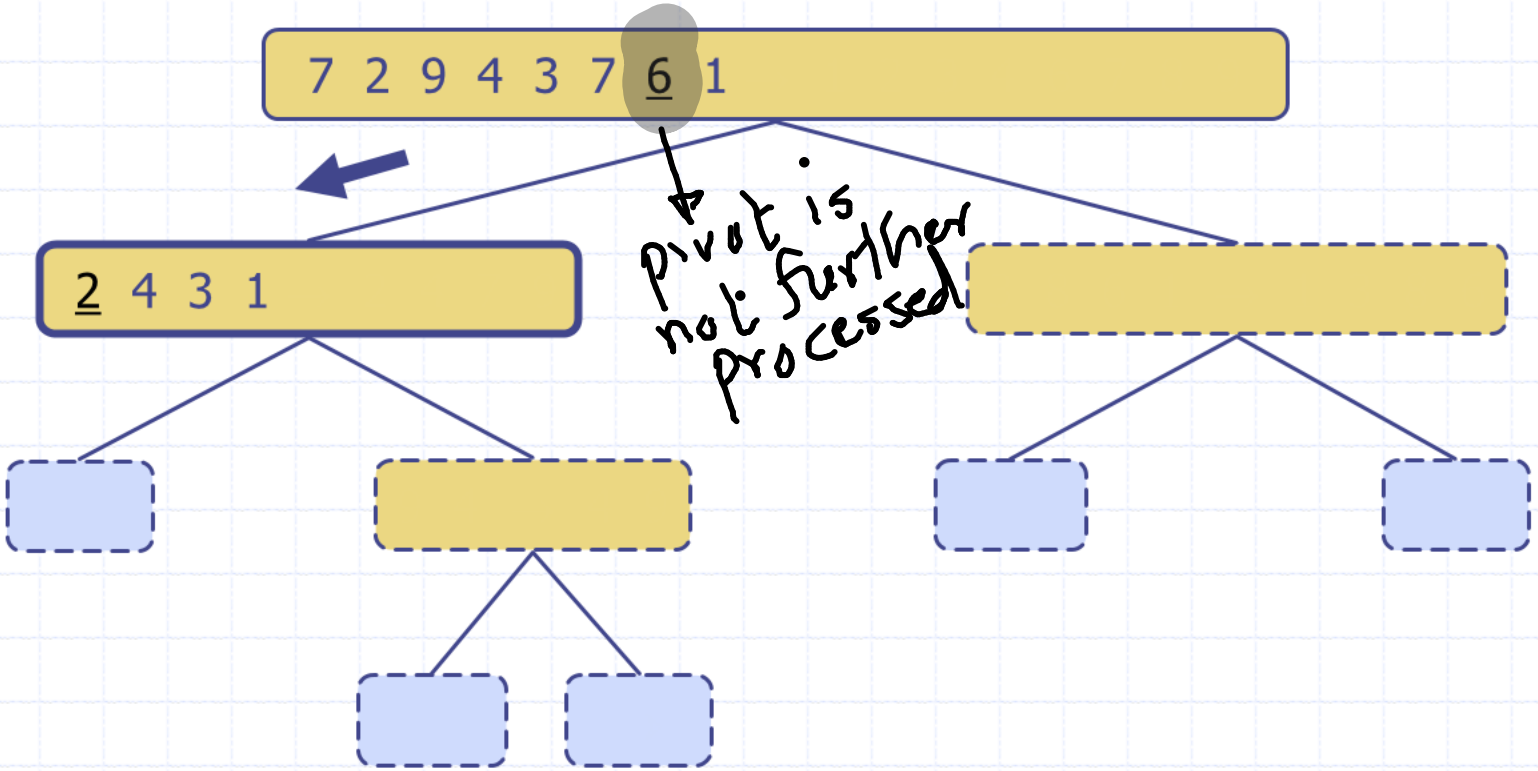
Execution Example

- Pivot selection



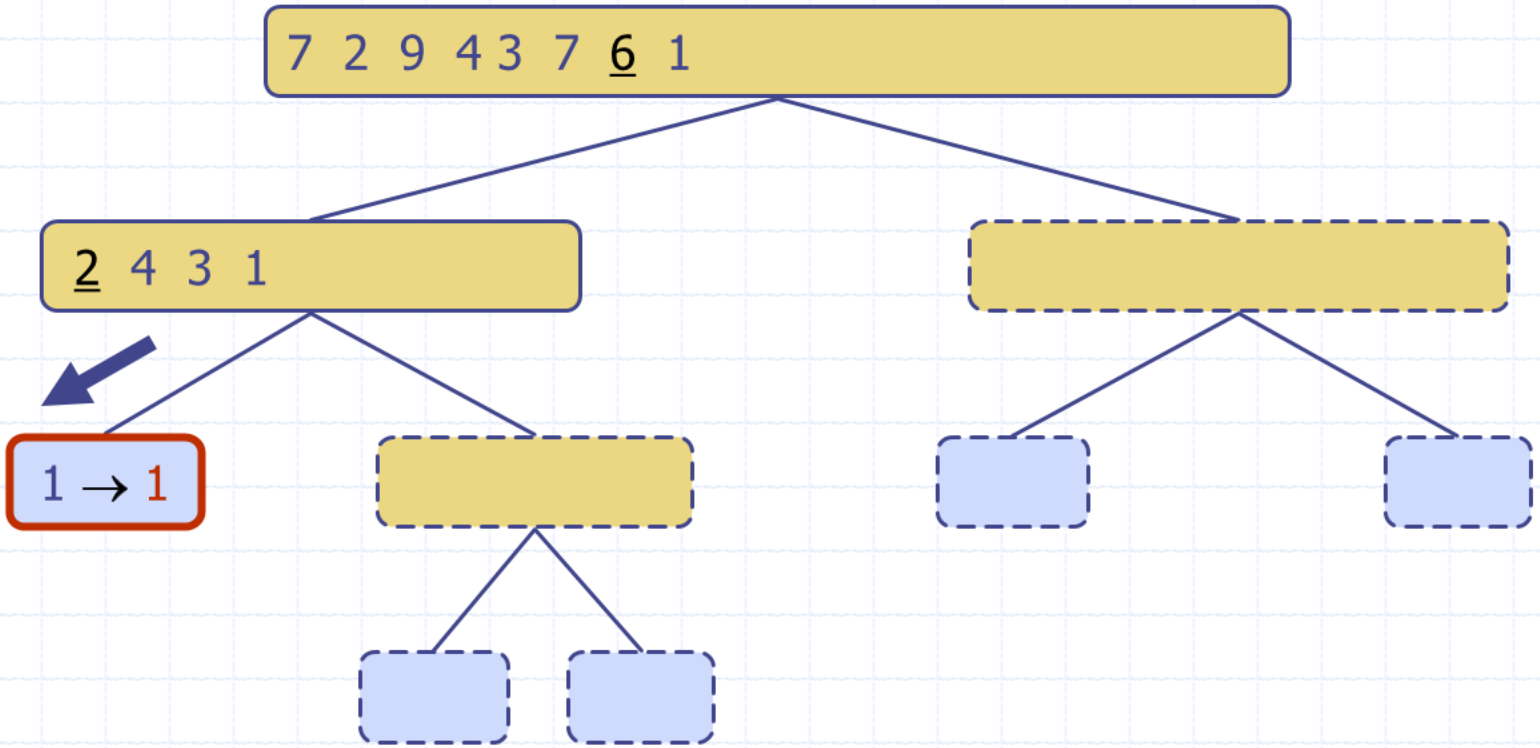
Execution Example (cont.)

- Partition, recursive call, pivot selection



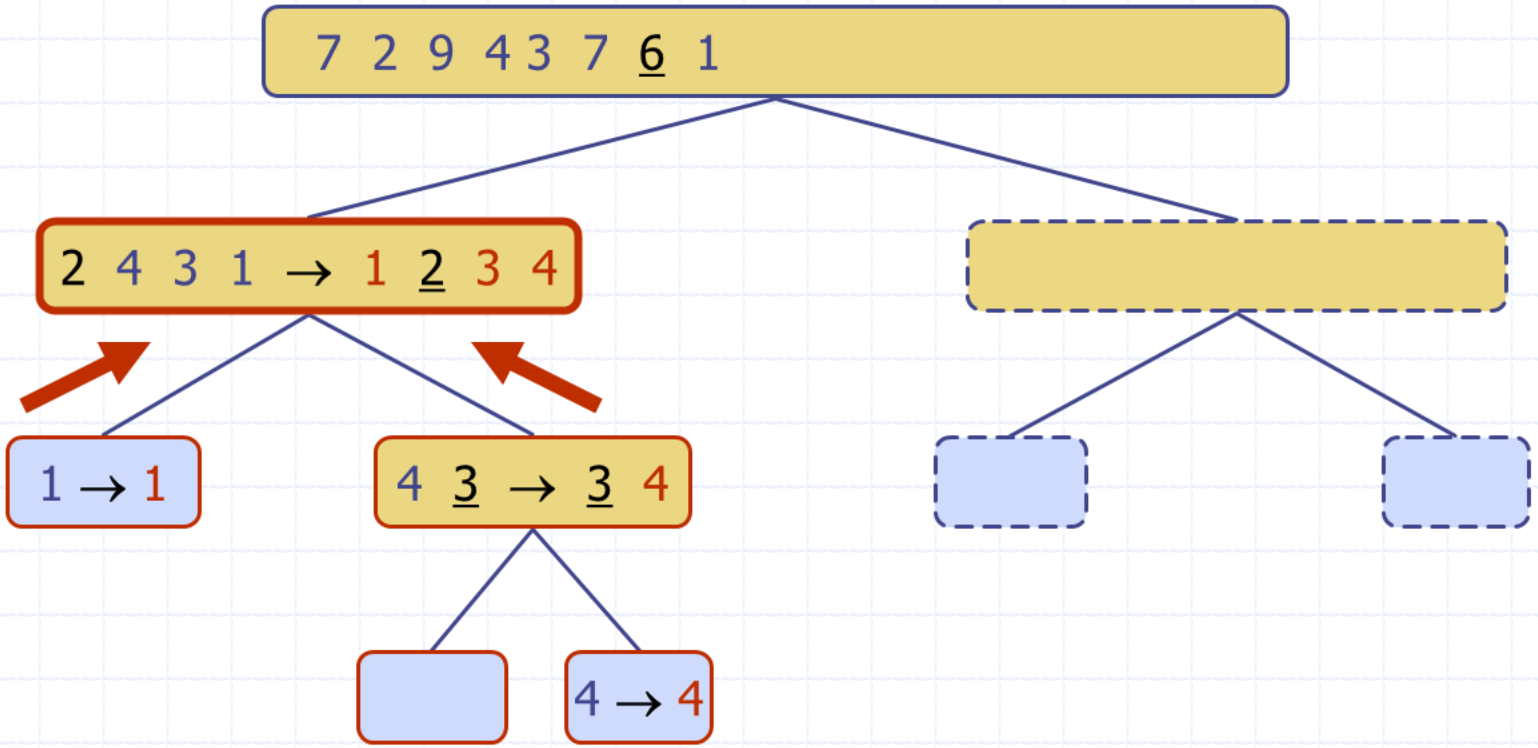
Execution Example (cont.)

- Partition, recursive call, base case



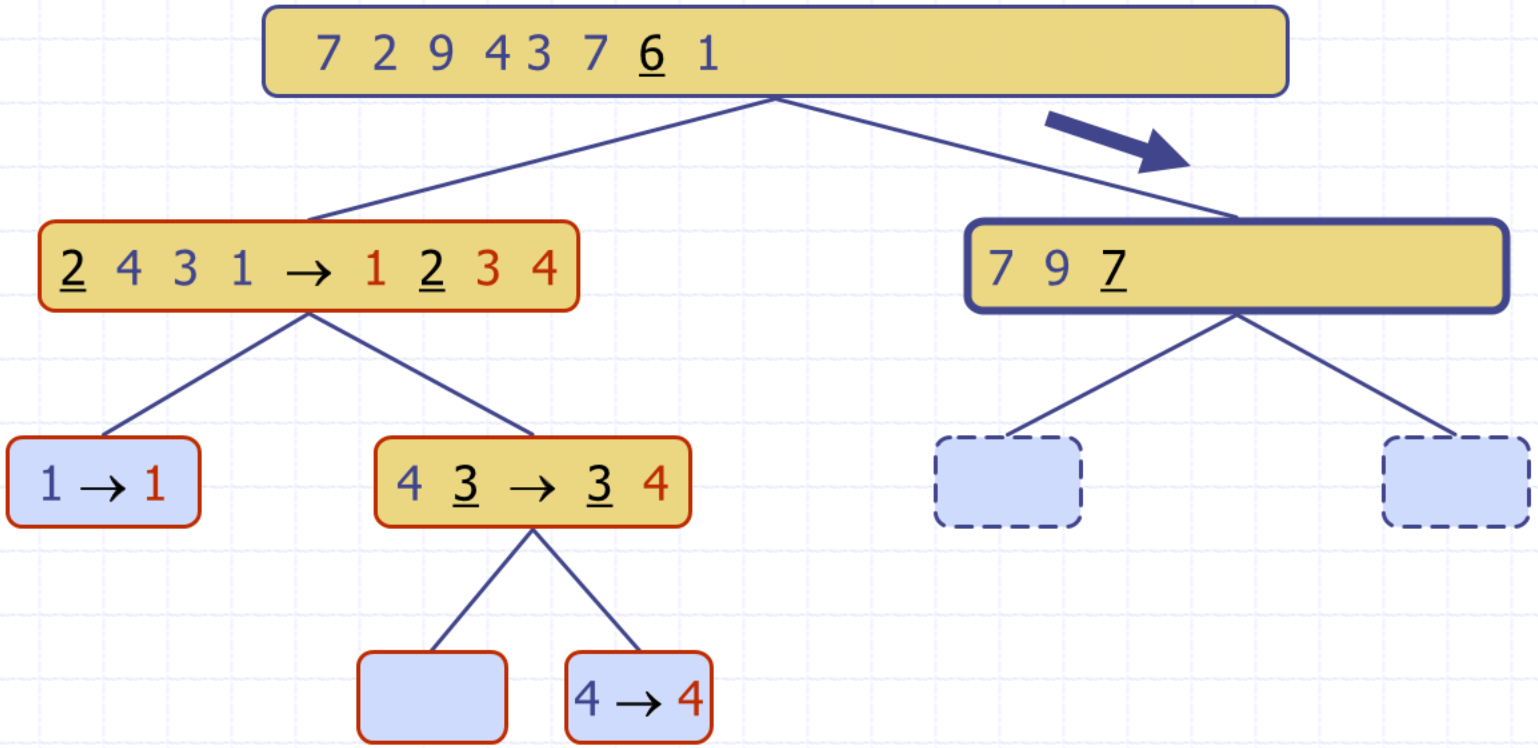
Execution Example (cont.)

- Recursive call, …, base case, join



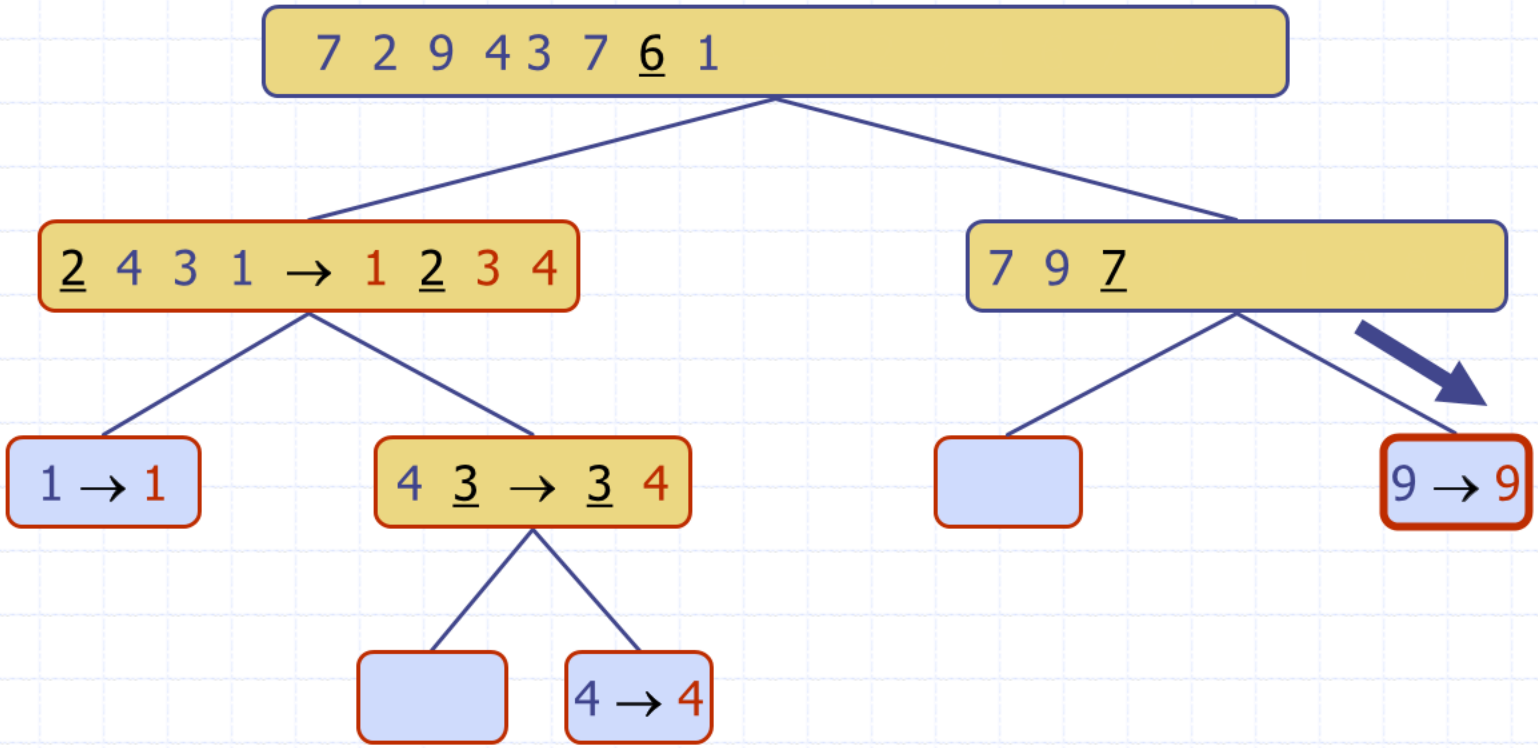
Execution Example (cont.)

- Recursive call, pivot selection



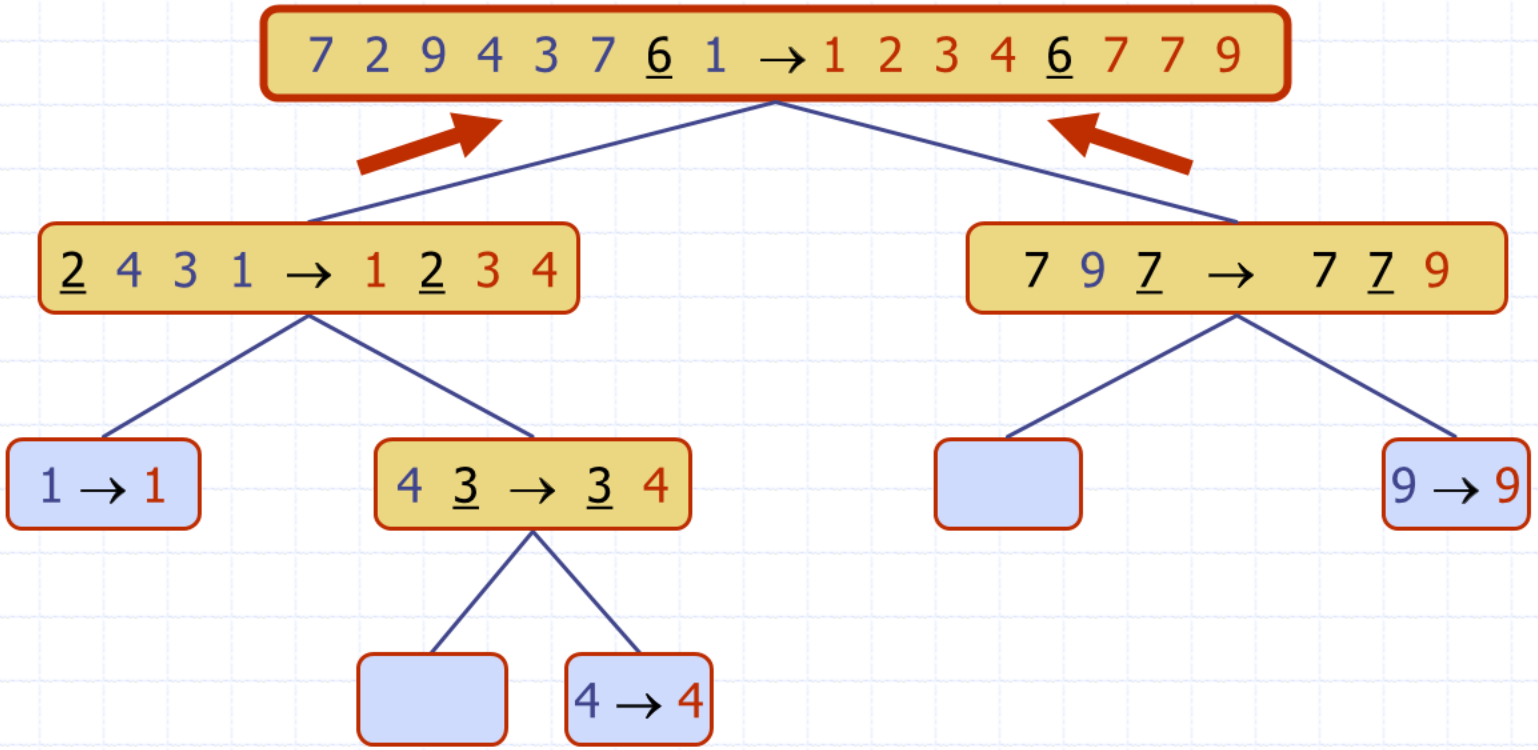
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



Worst-case Running Time

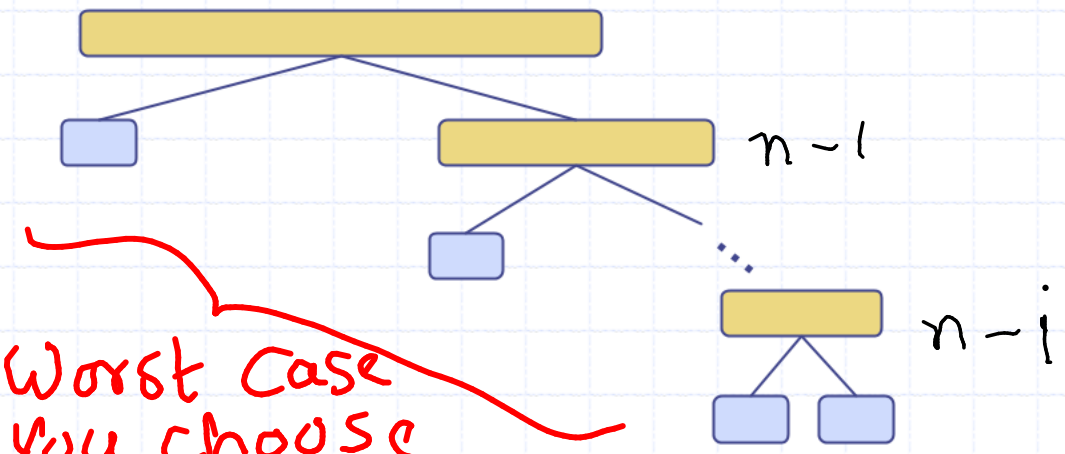
- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n-1) + \ldots + 2 + 1 \quad = O(n^2)$$

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

for worst case, functions like selection sort

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



Worst case you choose smallest element as pivot at each step