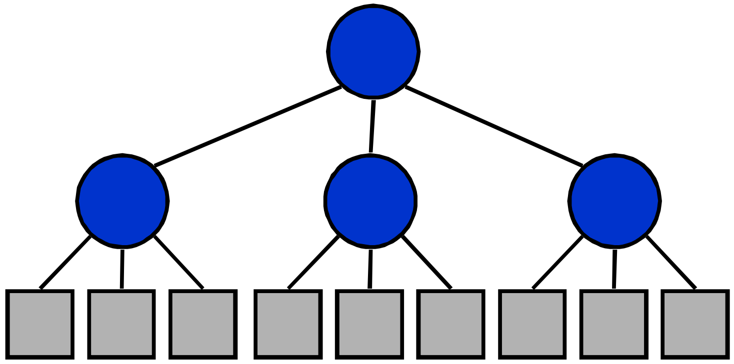
Divide-and-Conquer Technique: Merge Sort, Quick Sort

Dr. Md. Abul Kashem Mia, Professor, CSE Dept, BUET

Divide-and-Conquer

- Divide-and-Conquer is a general algorithm design paradigm:
  - Divide the problem into a number of subproblems that are smaller instances of the same problem
  - Conquer the subproblems by solving them recursively
  - Combine the solutions to the subproblems into the solution for the original problem
- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**



Merge Sort and Quick Sort

Two well-known sorting algorithms adopt this divide-and-conquer strategy

- Merge sort
  - Divide step is trivial – just split the list into two equal parts
  - Work is carried out in the conquer step by merging two sorted lists
- Quick sort
  - Work is carried out in the divide step using a pivot element
  - Conquer step is trivial

Merge Sort: Algorithm

MERGE-SORT(A, p, r)

if $p < r$

  then $q \leftarrow \lfloor (p + r)/2 \rfloor$

    MERGE-SORT(A, p, q)

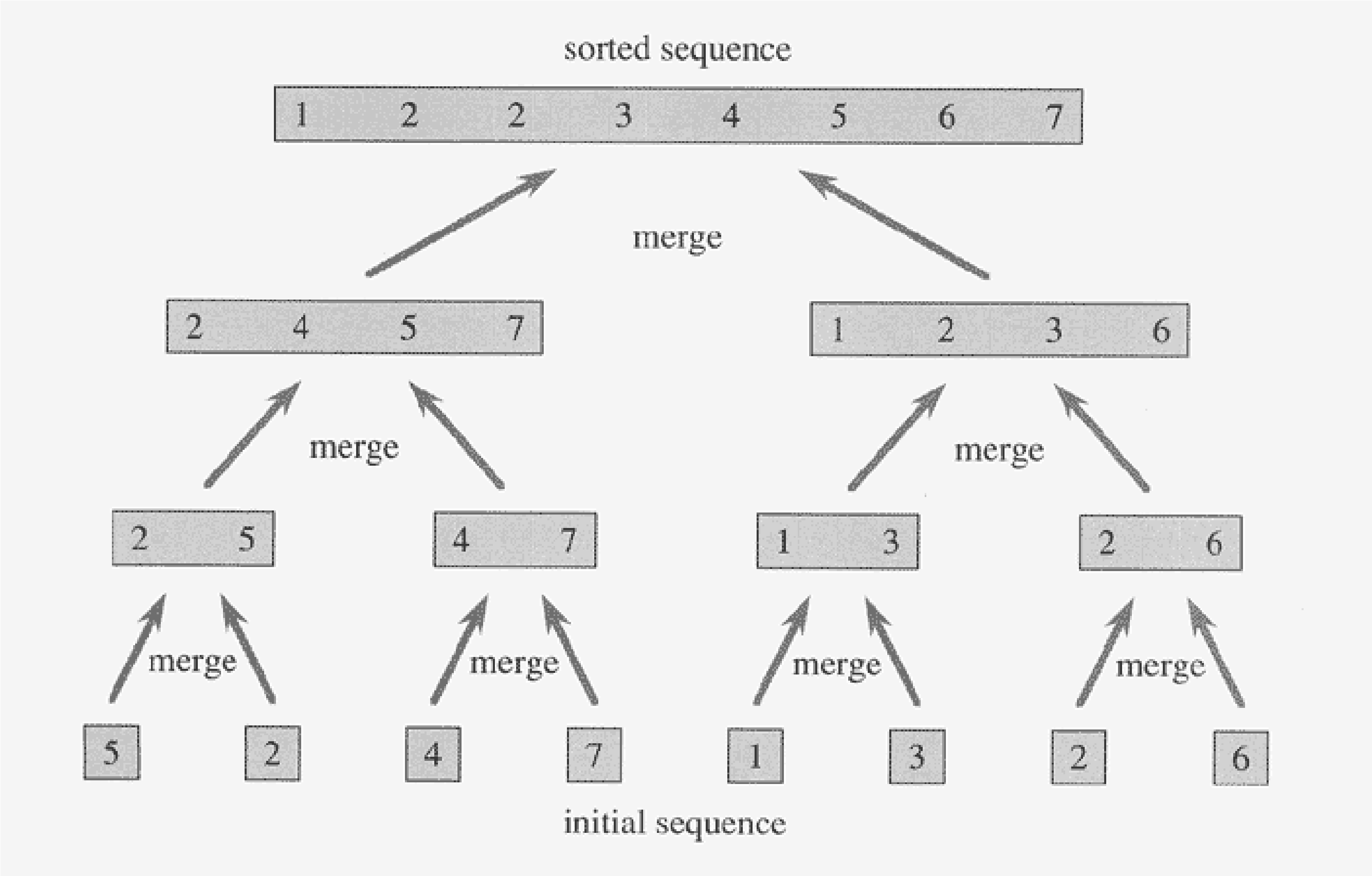
    MERGE-SORT$(A, q + 1, r)$

    MERGE(A, p, q, r)

Merge Sort: Algorithm

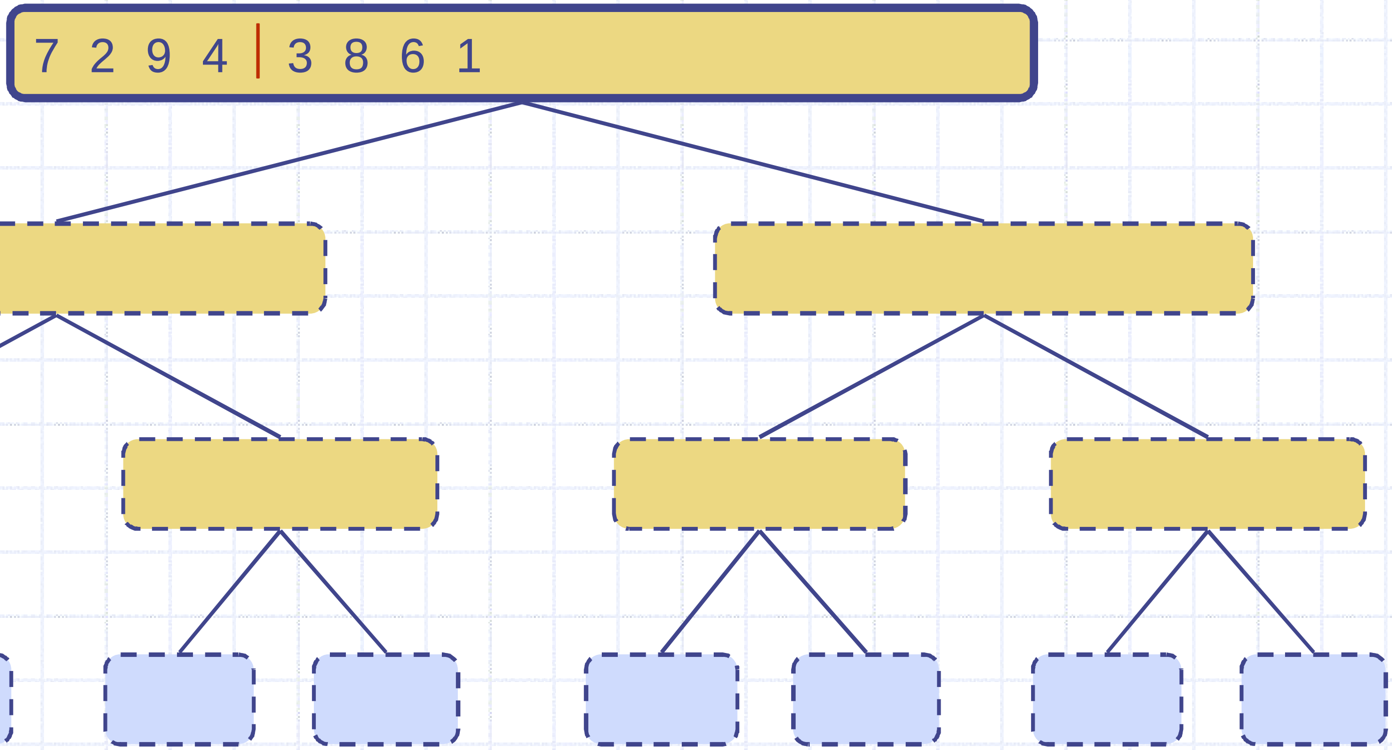
```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
for i ← 1 to n1
        do L[i] ← A[p + i − 1]
for j ← 1 to n2
        do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
        do if L[i] ≤ R[j]
              then A[k] ← L[i]
                   i ← i + 1
              else A[k] ← R[j]
                   j ← j + 1
```

Merge Sort: Example



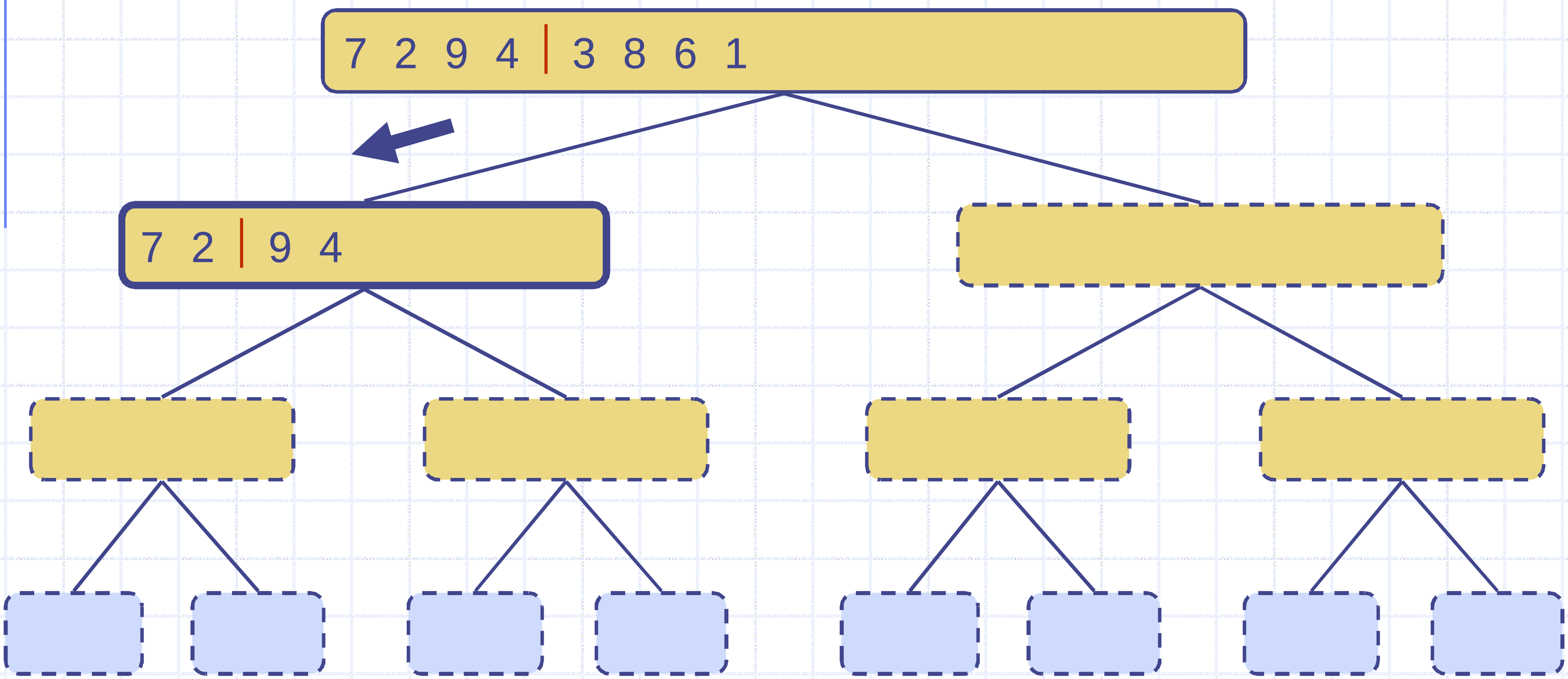
Execution Example

- Partition



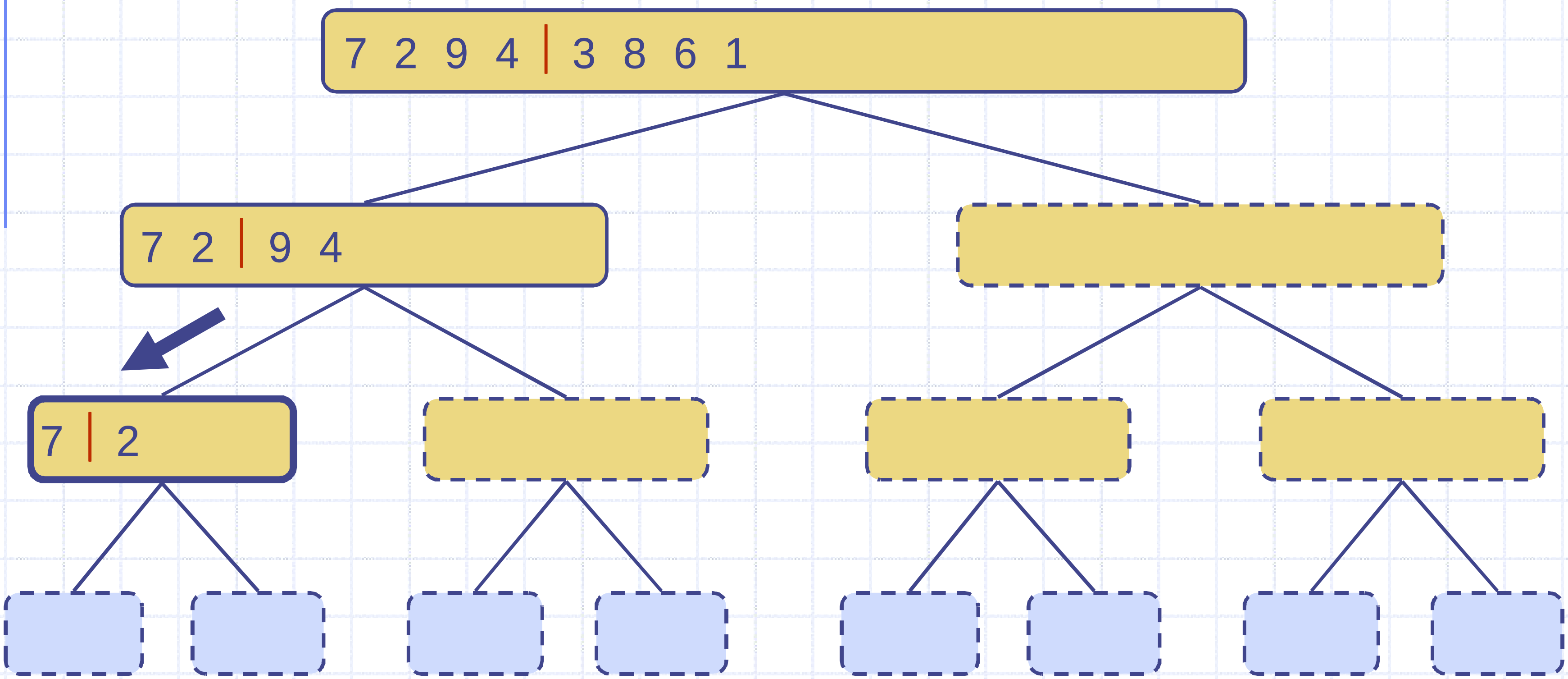
Execution Example (cont.)

- Recursive call, partition



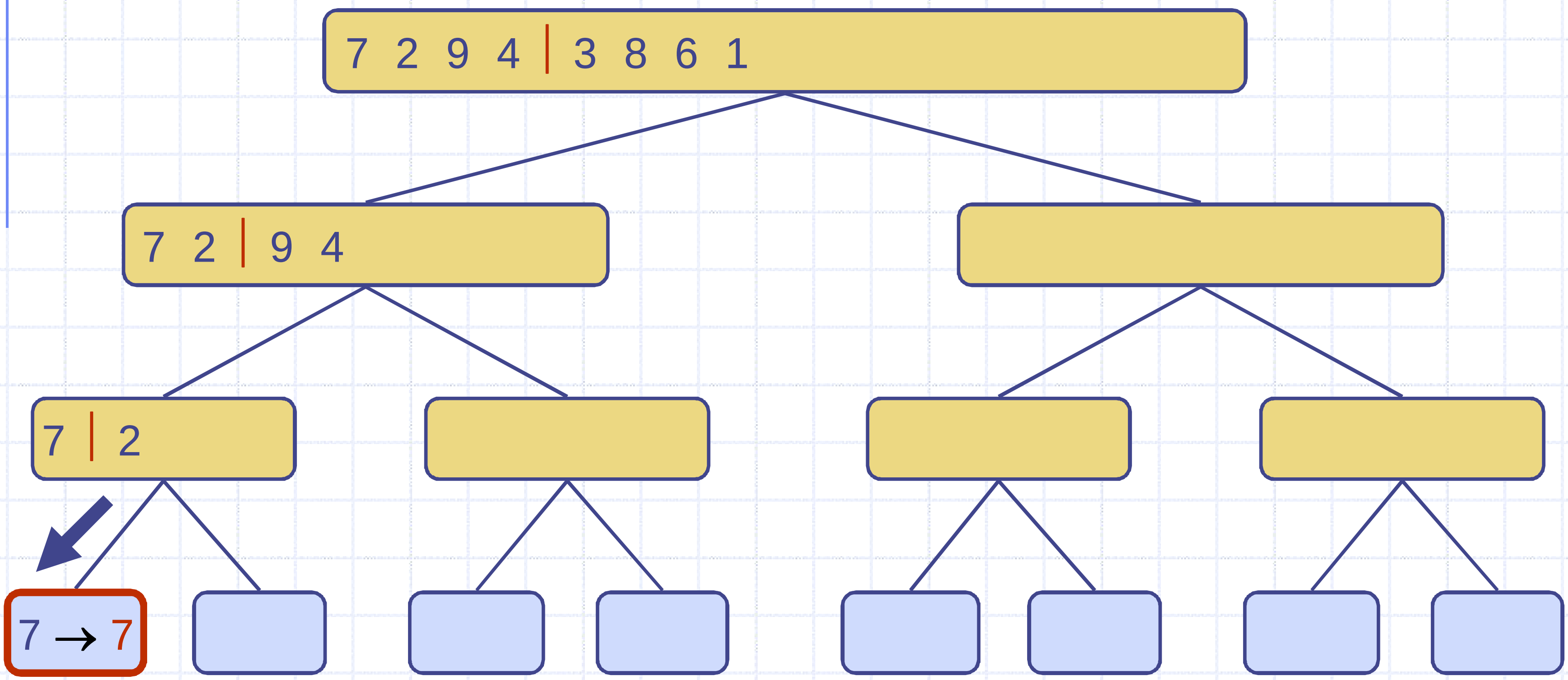
Execution Example (cont.)

- Recursive call, partition



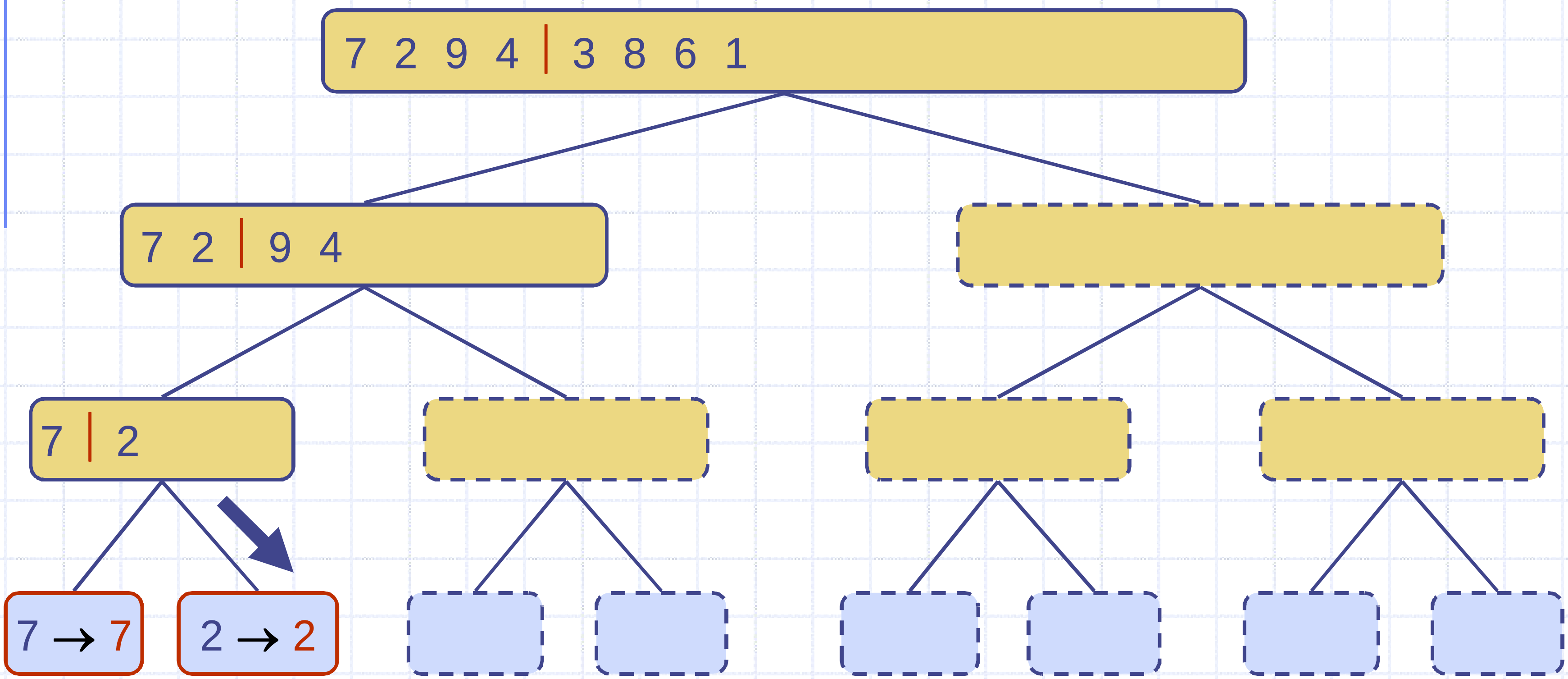
Execution Example (cont.)

- Recursive call, base case



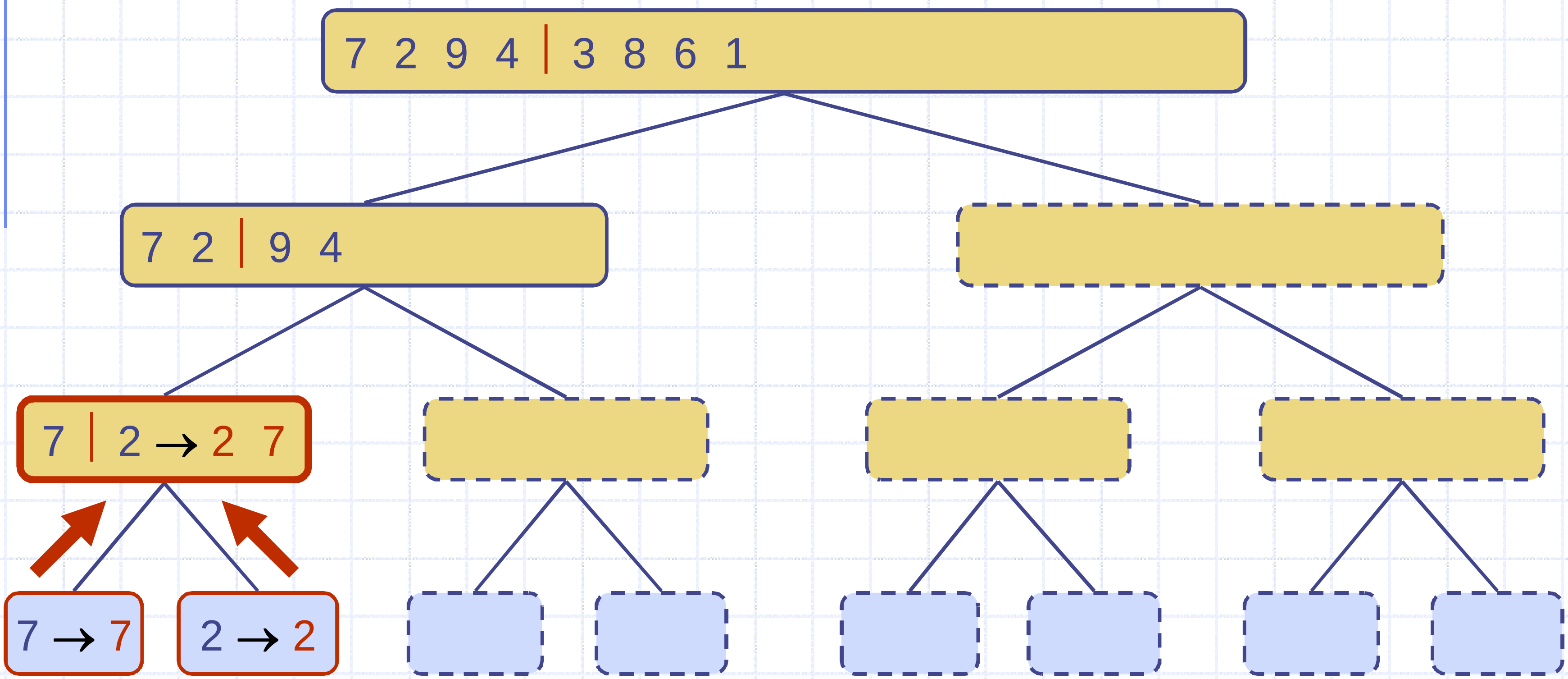
Execution Example (cont.)

- Recursive call, base case



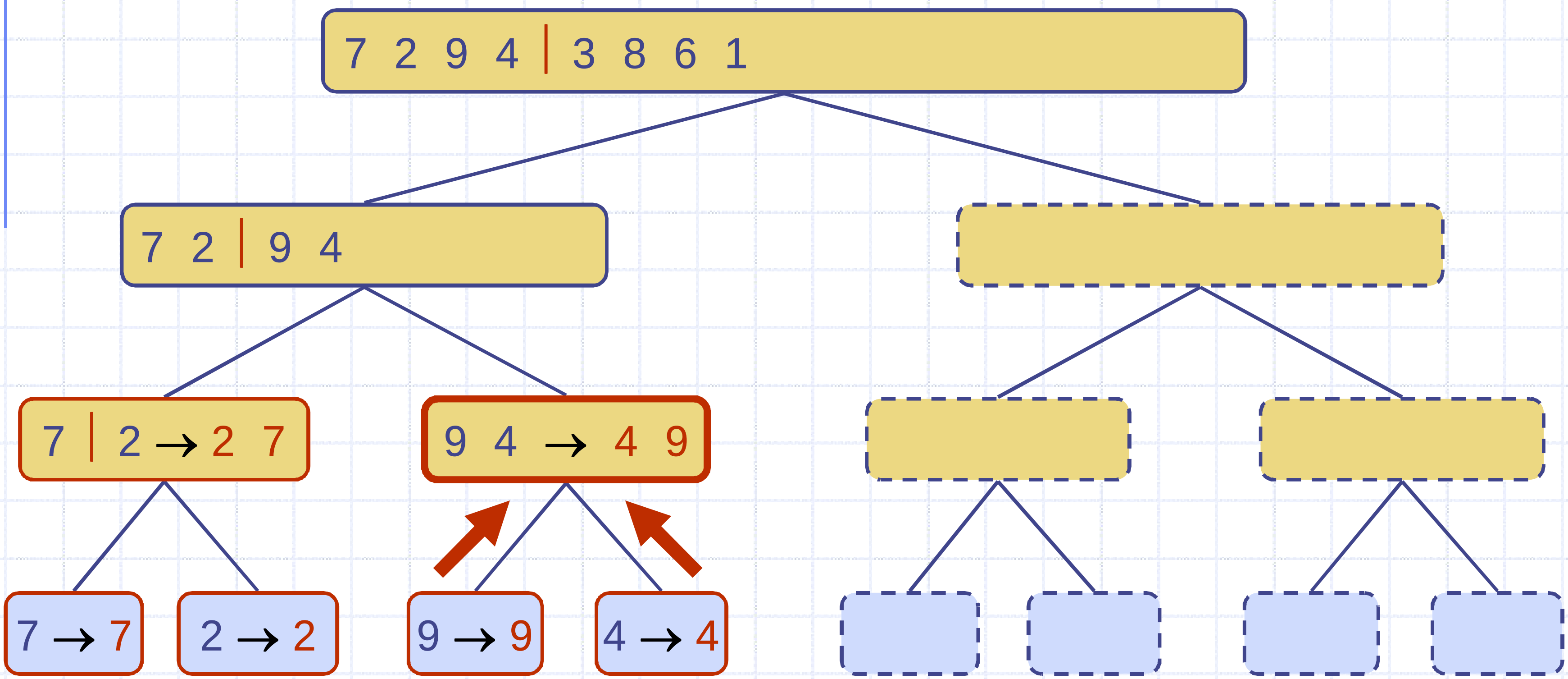
Execution Example (cont.)

- Merge



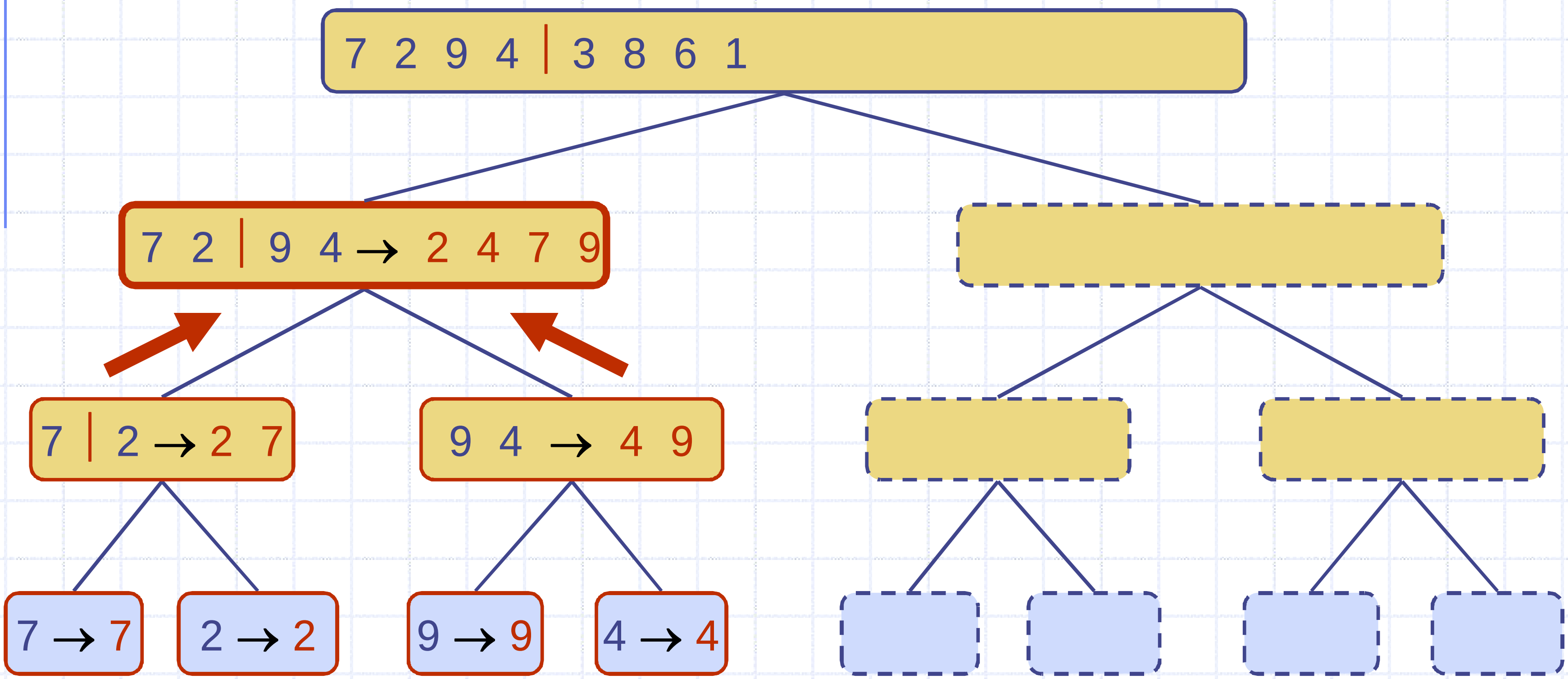
Execution Example (cont.)

- Recursive call, ..., base case, merge



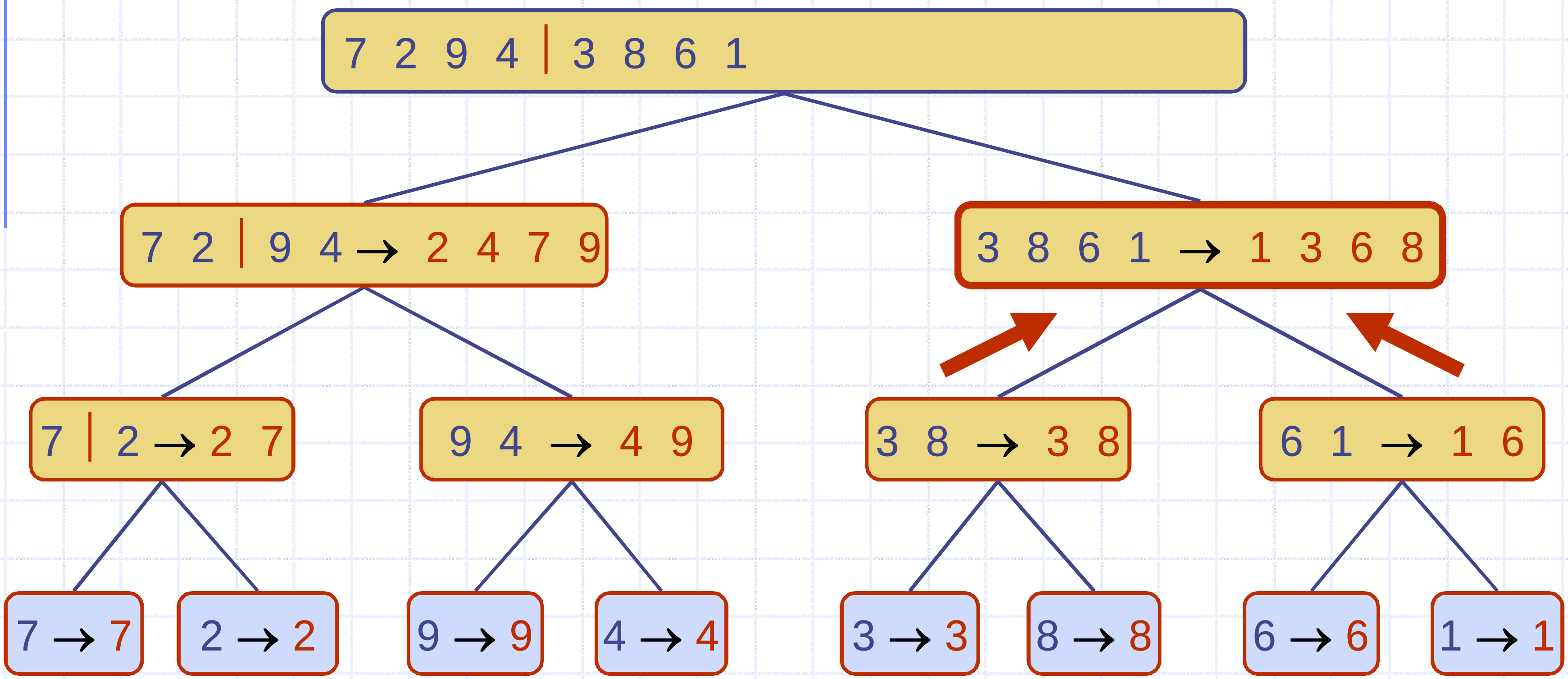
Execution Example (cont.)

- Merge



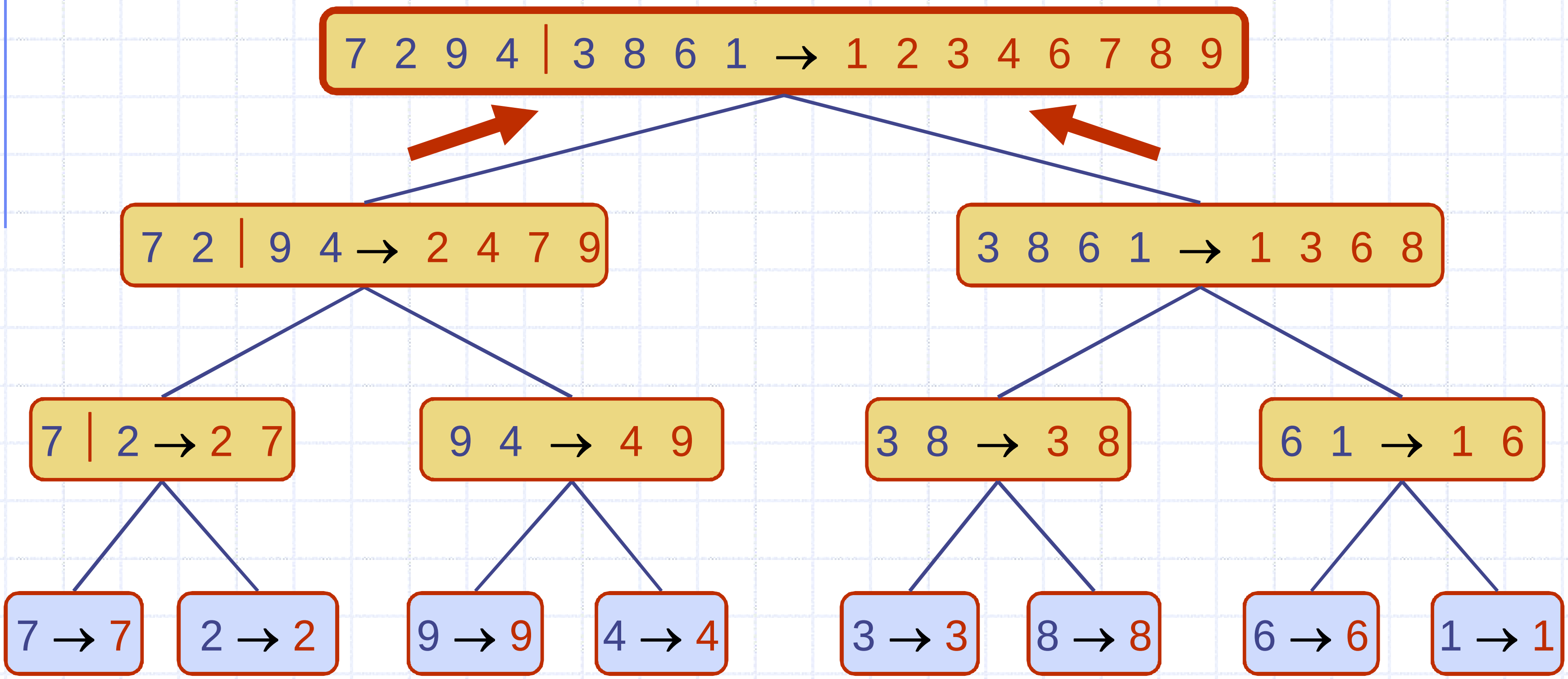
Execution Example (cont.)

- Recursive call, ..., merge, merge



Execution Example (cont.)

- Merge



Merge Sort: Running Time

The recurrence for the worst-case running time T(n) is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

equivalently

$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n > 1 \end{cases}$$

Solve this recurrence by

(1) iteratively expansion

(2) using the recursion tree

Merge Sort: Running Time (Iterative Expansion)

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2)) + b(n/2)) + bn \\
&= 2^2T(n/2^2) + 2bn \\
&= 2^3T(n/2^3) + 3bn \\
&= 2^4T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^iT(n/2^i) + ibn
\end{aligned}
$$

- Note that base, $T(n) = b$, case occurs when $2^i = n$.

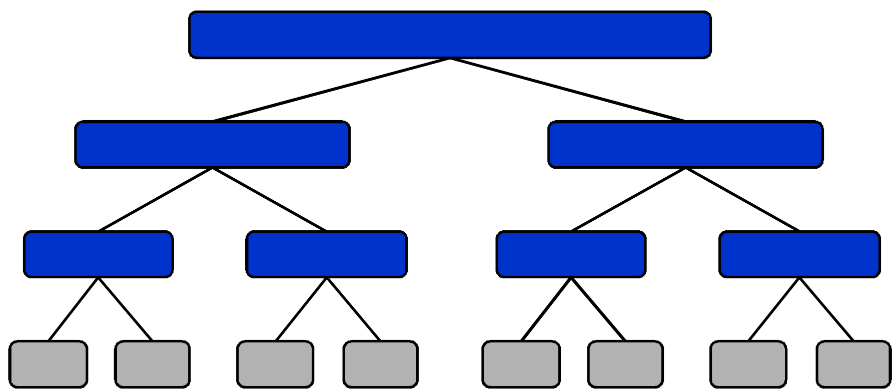
  That is, $i = \log n$.
- So, $T(n) = bn + bn\log n$
- Thus, $T(n)$ is $O(n \log n)$.

Merge Sort: Running Time (Recursion Tree)

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

$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

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



Total time = $\mathbf{bn} + \mathbf{bn} \log \mathbf{n}$

(last level plus all previous levels)

Quick Sort: Algorithm

- Another divide-and-conquer algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort
  - Unlike merge sort, no combining step: two subarrays form an already-sorted array

Quick Sort: Algorithm

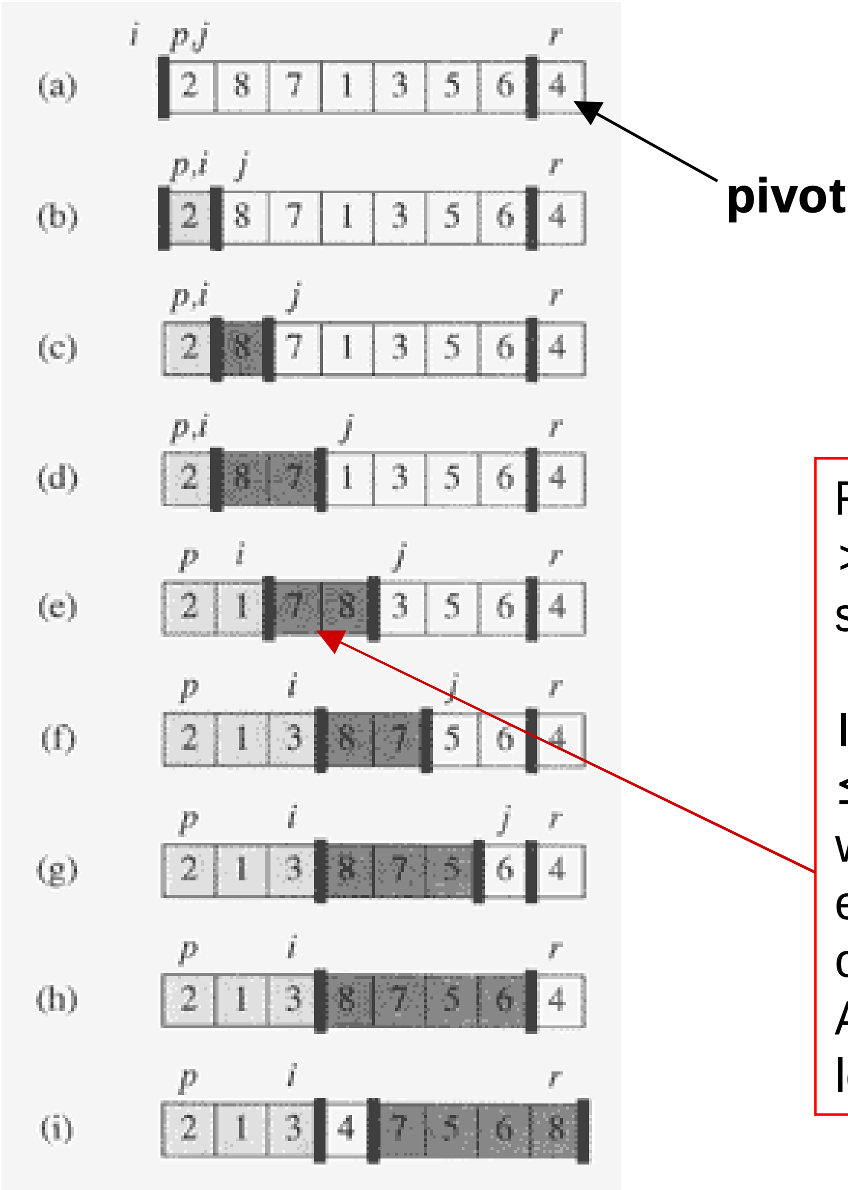
```
QUICKSORT(A, p, r)
if p < r
    then q ← PARTITION(A, p, r)
         QUICKSORT(A, p, q − 1)
         QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
    do if A[j] ≤ x
        then i ← i + 1
             exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Quick Sort: Algorithm (Partition)

- Clearly, all the actions take place in the **`partition()`** function
  - Rearranges the subarrays in place
  - End result:
    - Two subarrays
    - All values in first subarray $\leq$ all values in the second
  - Returns the index of the “pivot” element separating the two subarrays

Quick Sort: Algorithm



From i +1 to j is a window of elements > A[r]. The cursor j moves right one step at a time.

If the cursor j "discovers" an element ≤ A[r], then this element is swapped with the front element of the window, effectively moving the window right one step; if it discovers an element > A[r], then the window simply becomes longer one unit.

Quick Sort: Algorithm

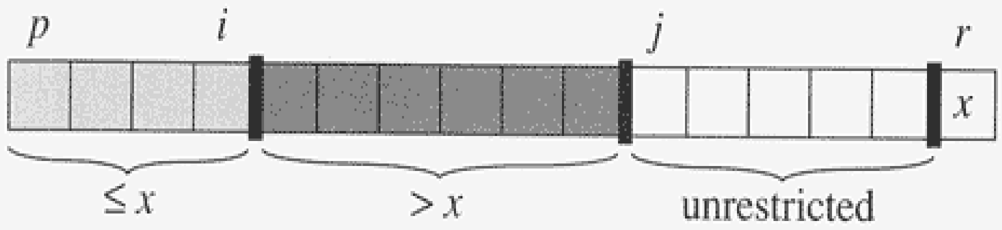


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p \mathinner{.\,.} r]$. The values in $A[p \mathinner{.\,.} i]$ are all less than or equal to x, the values in $A[i+1 \mathinner{.\,.} j-1]$ are all greater than x, and $A[r] = x$. The values in $A[j \mathinner{.\,.} r-1]$ can take on any values.

Quick Sort: Analysis