Data Structure - Spring 2022
8. Queue and Sorting

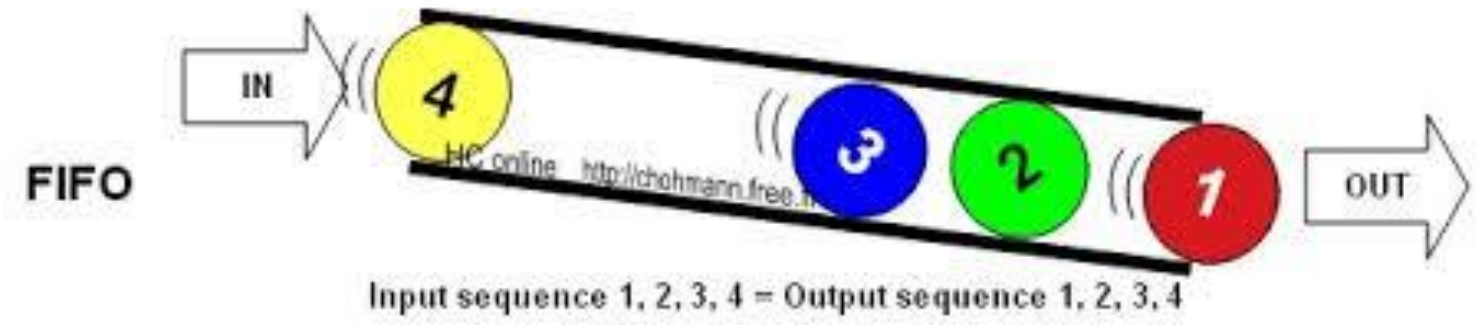
Walid Abdullah Al
Computer and Electronic Systems Engineering
Hankuk University of Foreign Studies

Based on:
Goodrich, Chapter 6
Karumanchi, Chapter 5
Slides by Prof. Yung Yi, KAIST
Slides by Prof. Chansu Shin, HUFS

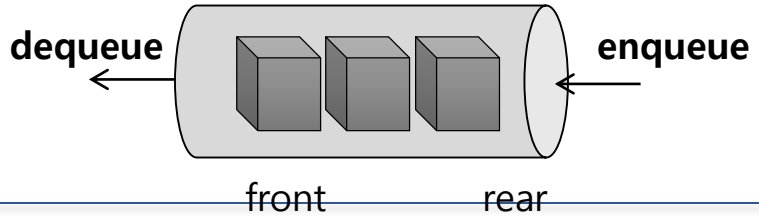
Computer Vision Lab
Hankuk University of Foreign Studies

Queue

- **First-in-first-out (FIFO) data structure**



Queue



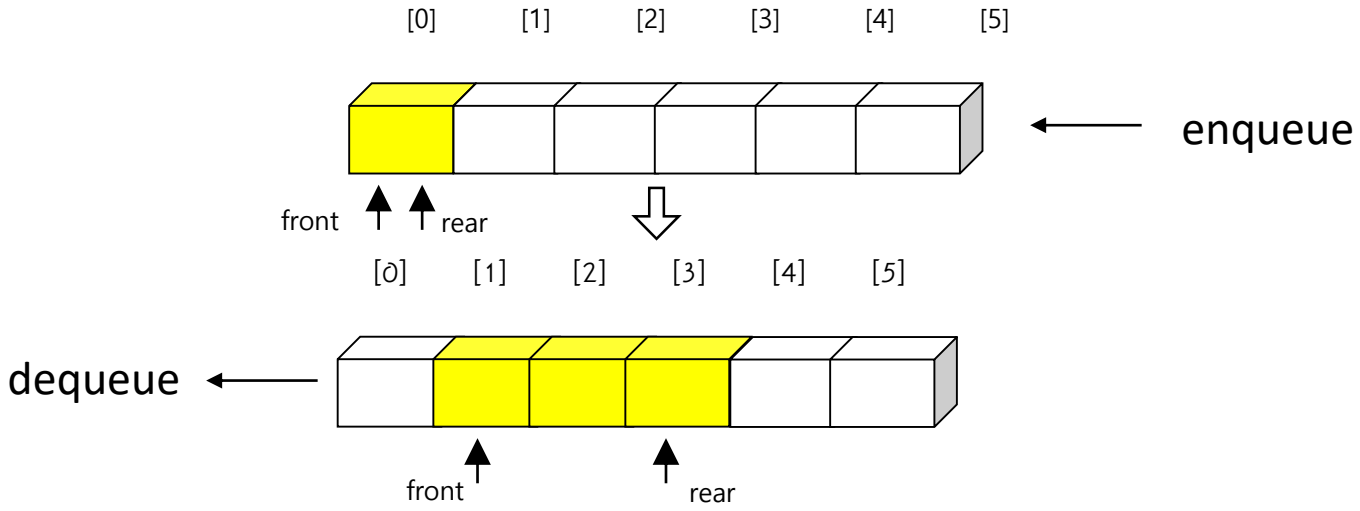
- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Main queue operations:
  - enqueue(object): inserts an element at the **back**
  - object dequeue(): removes and returns the **first** element
- Auxiliary queue operations:
  - object first(): returns the first element without removing it
  - integer size(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored
  - boolean is_full()

Queue operation

Operation	Return Value	first $\leftarrow Q \leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue( )	7	[9, 4]

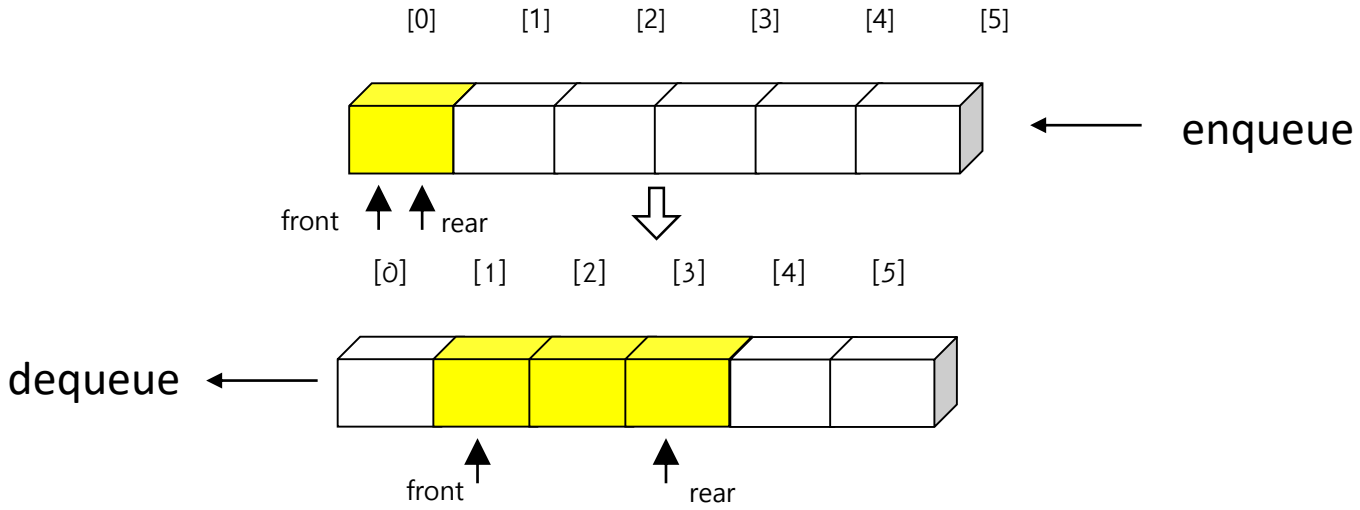
Queue implementation

- **Fixed size array/list**
- **Simplest way: linear structure**
  - Front, rear identifiers
  - enqueue: increment rear, insert at the updated rear
  - dequeue: remove front element, increment front



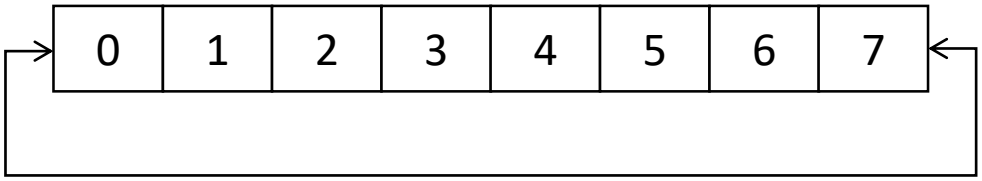
Queue: problem with linear structure

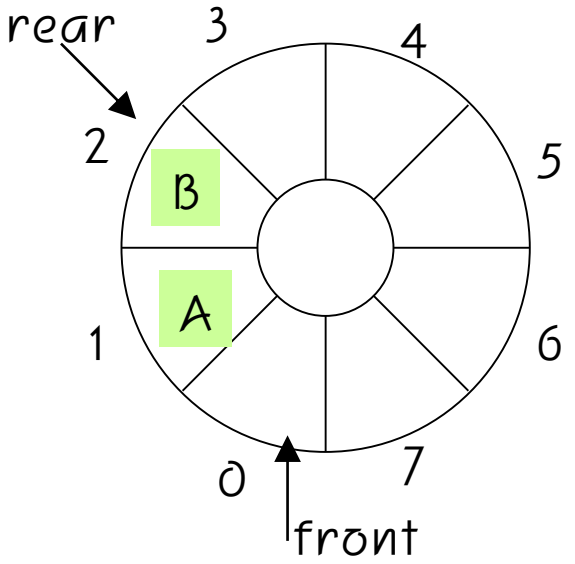
- **Enqueue 5 times, dequeue 5 times**
  - Queue is full, cannot increment rear
  - However, the leftmost slots are empty
- **How to solve?**
  - Shift left



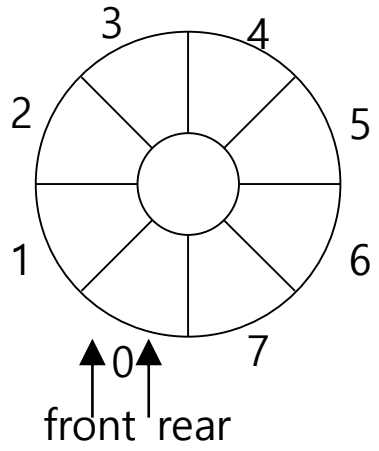
Queue: implement with circular structure

- **Circular indexing**
  - 0→1→2→3→…→6→7→0→1→…
  - How to increment front and rear?

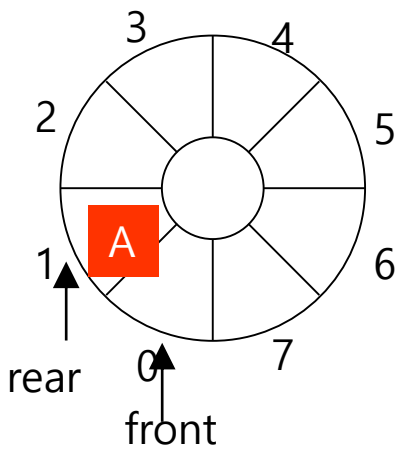




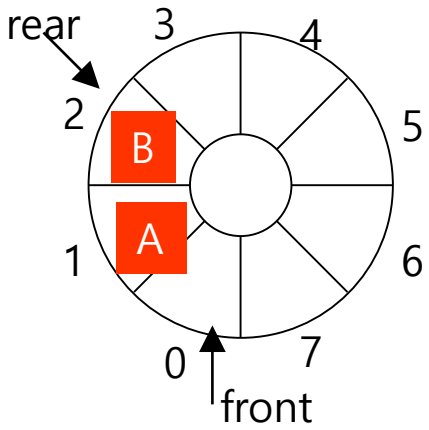
Circular Queue: operation



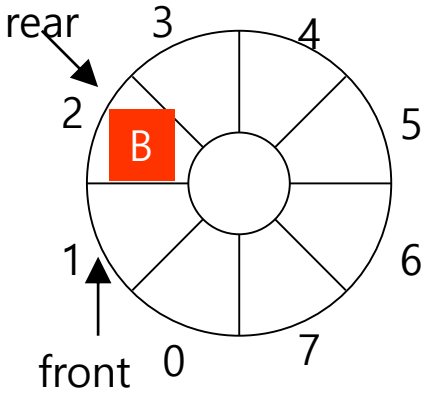
(a) Initial condition



(b) Insert A



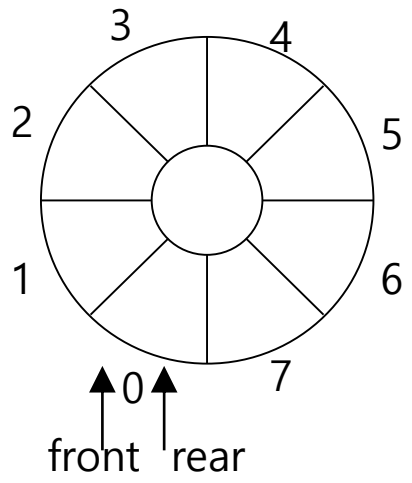
(c) Insert B



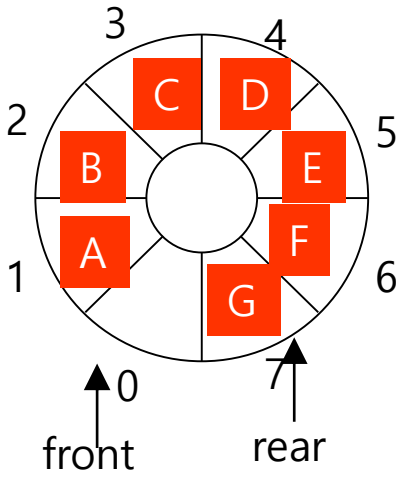
(d) Delete A

Circular Queue: problem

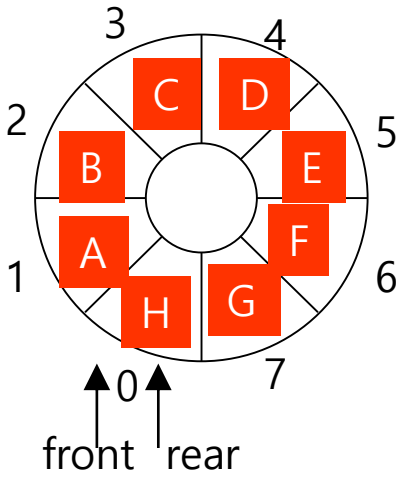
- **Empty and full state indistinguishable**



Empty



Just before Full



Full

- **How to solve?**
  - Use additional variable to track the size
  - Or, just maintain one empty space

Goorm Practice

- **Build a circular queue and verify its operations**
  - You may refer to Chapter 6 of the main textbook
- **Python shortcut using list**
  - enqueue(e): list.append(e)
  - dequeue(): list.pop(0)

Dequeue (double-ended queue)

- **Dequeue:**
  - Can insert at both front and rear
  - Can delete from both front and rear

D.add_first(e):	Add element e to the front of deque D.
D.add_last(e):	Add element e to the back of deque D.
D.delete_first( ):	Remove and return the first element from deque D; an error occurs if the deque is empty.
D.delete_last( ):	Remove and return the last element from deque D; an error occurs if the deque is empty.

  - Auxiliary functions:
    - D.first(), D.last(), D.is_empty(), D.size()

Dequeue: operation

Operation	Return Value	Deque
D.add_last(5)	–	[5]
D.add_first(3)	–	[3, 5]
D.add_first(7)	–	[7, 3, 5]
D.first( )	7	[7, 3, 5]
D.delete_last( )	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last( )	3	[7]
D.delete_last( )	7	[ ]
D.add_first(6)	–	[6]
D.last( )	6	[6]
D.add_first(8)	–	[8, 6]
D.is_empty( )	False	[8, 6]
D.last( )	6	[8, 6]

Dequeue- implementation

- from **collections** import **dequeue**

Our Deque ADT	collections.deque	Description
len(D)	len(D)	number of elements
D.add_first( )	D.appendleft( )	add to beginning
D.add_last( )	D.append( )	add to end
D.delete_first( )	D.popleft( )	remove from beginning
D.delete_last( )	D.pop( )	remove from end
D.first( )	D[0]	access first element
D.last( )	D[−1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear( )	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

Sorting

- **Ascending & descending order**
- **Fundamental element in searching**

- **Algorithm Evaluation:**
  - Number of comparisons, number of moves

Insertion sort

Algorithm InsertionSort(A):

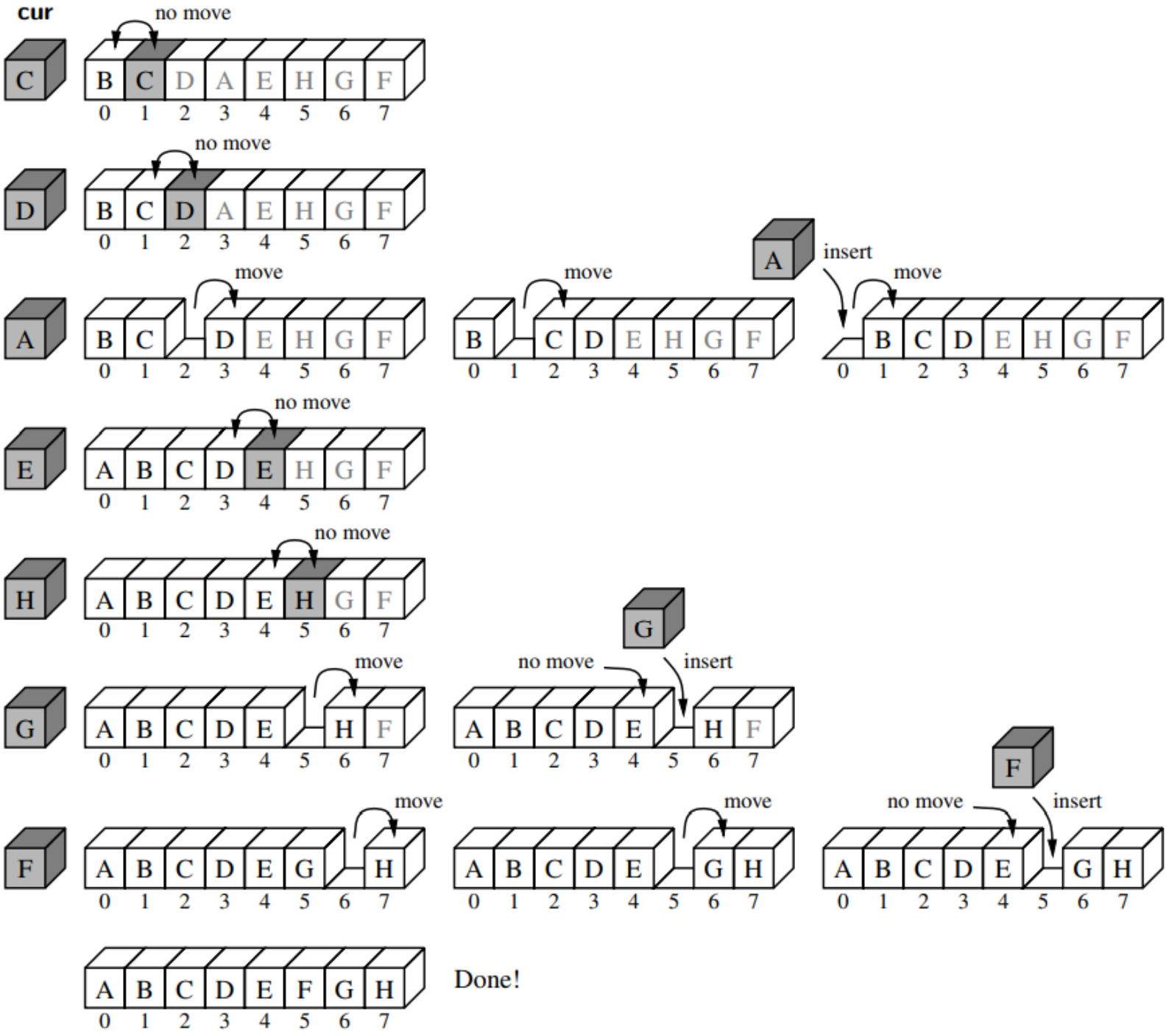
Input: An array A of n comparable elements

Output: The array A with elements rearranged in nondecreasing order

for k from 1 to n − 1 **do**

Insert A[k] at its proper location within A[0], A[1], . . ., A[k].

```
1  def insertion_sort(A):
2    """Sort list of comparable elements into nondecreasing order."""
3    for k in range(1, len(A)):                # from 1 to n-1
4      cur = A[k]                              # current element to be inserted
5      j = k                                   # find correct index j for current
6      while j > 0 and A[j−1] > cur:           # element A[j-1] must be after current
7        A[j] = A[j−1]
8        j −= 1
9      A[j] = cur                              # cur is now in the right place
```


cur
no move
C
B C D A E H G F
0 1 2 3 4 5 6 7
no move
D
B C D A E H G F
0 1 2 3 4 5 6 7
move
A
B C D E H G F
0 1 2 3 4 5 6 7
move
A
insert
B C D E H G F
0 1 2 3 4 5 6 7
move
B C D E H G F
0 1 2 3 4 5 6 7
no move
E
A B C D E H G F
0 1 2 3 4 5 6 7
no move
H
A B C D E H G F
0 1 2 3 4 5 6 7
move
G
A B C D E H F
0 1 2 3 4 5 6 7
G
no move
insert
A B C D E H F
0 1 2 3 4 5 6 7
move
F
A B C D E G H
0 1 2 3 4 5 6 7
move
A B C D E G H
0 1 2 3 4 5 6 7
F
no move
insert
A B C D E G H
0 1 2 3 4 5 6 7
A B C D E F G H
0 1 2 3 4 5 6 7
Done!

Insertion sort: running time

- $O(n^2)$
- **How?**

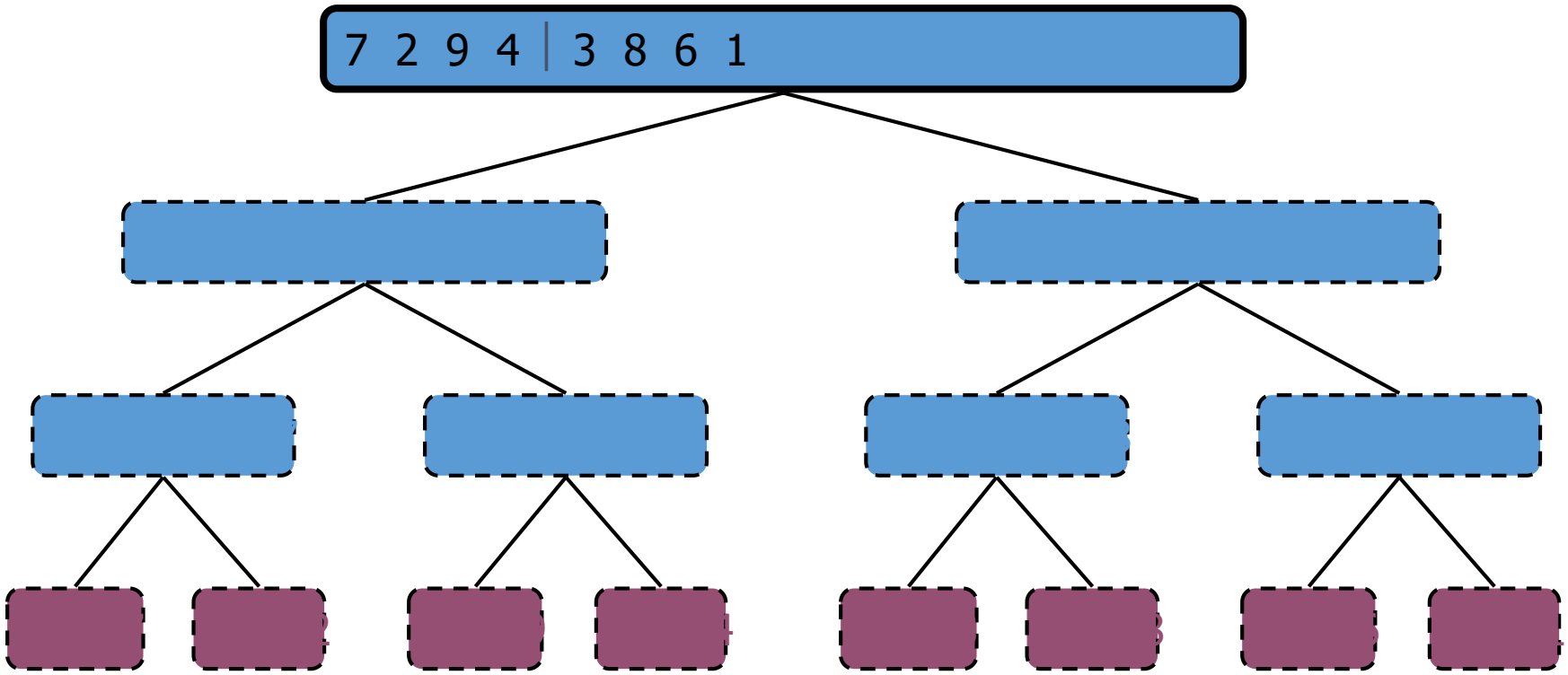
Merge-sort: divide and conquer (DnC) method

- **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**

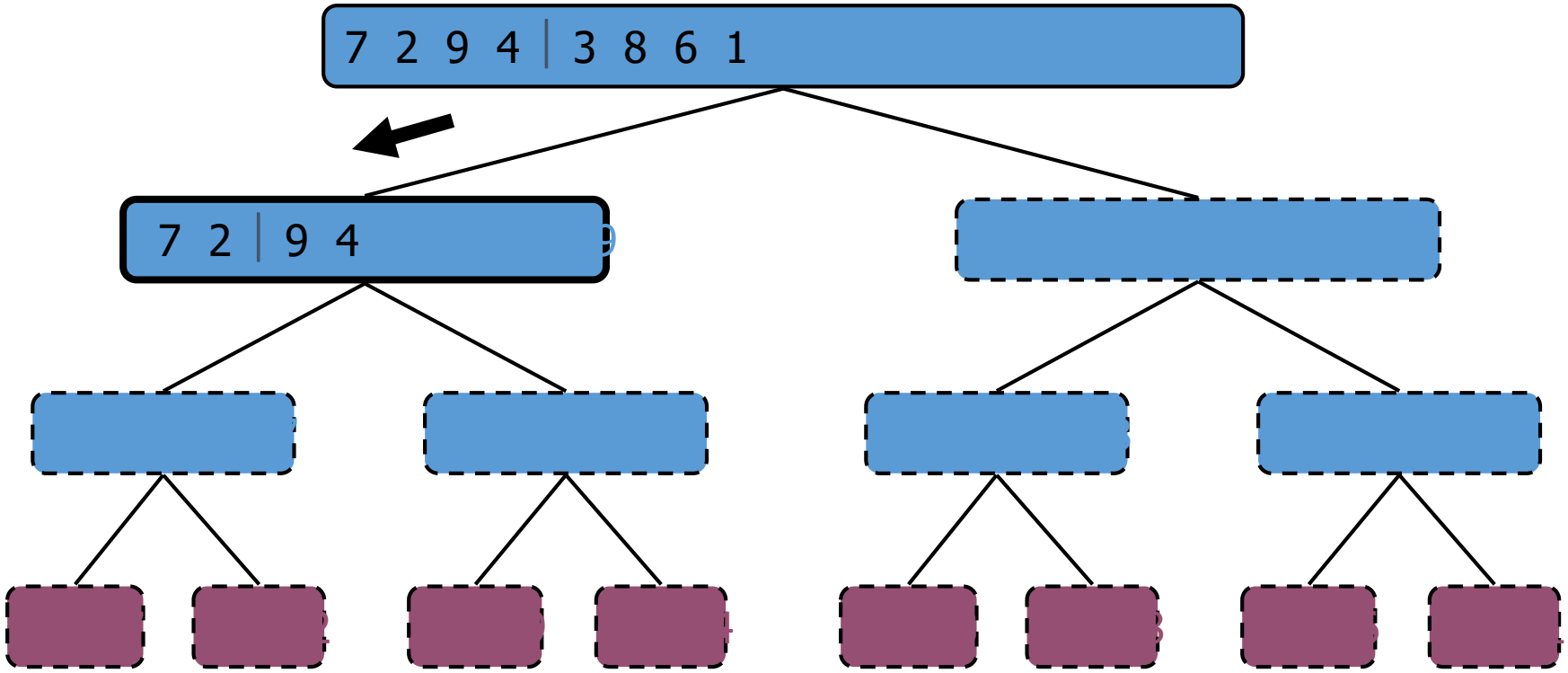
Execution Example

- **Partition**



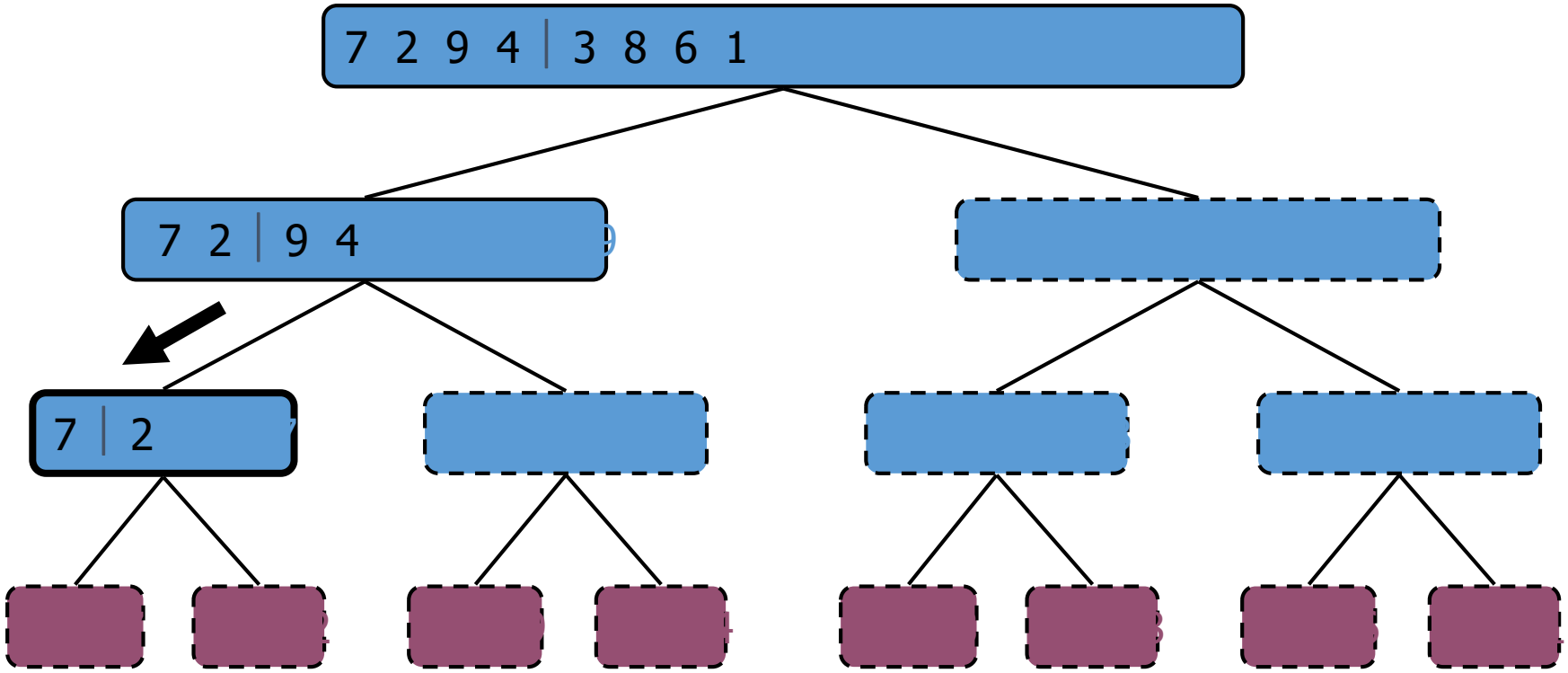
Execution Example (cont.)

- **Recursive call, partition**



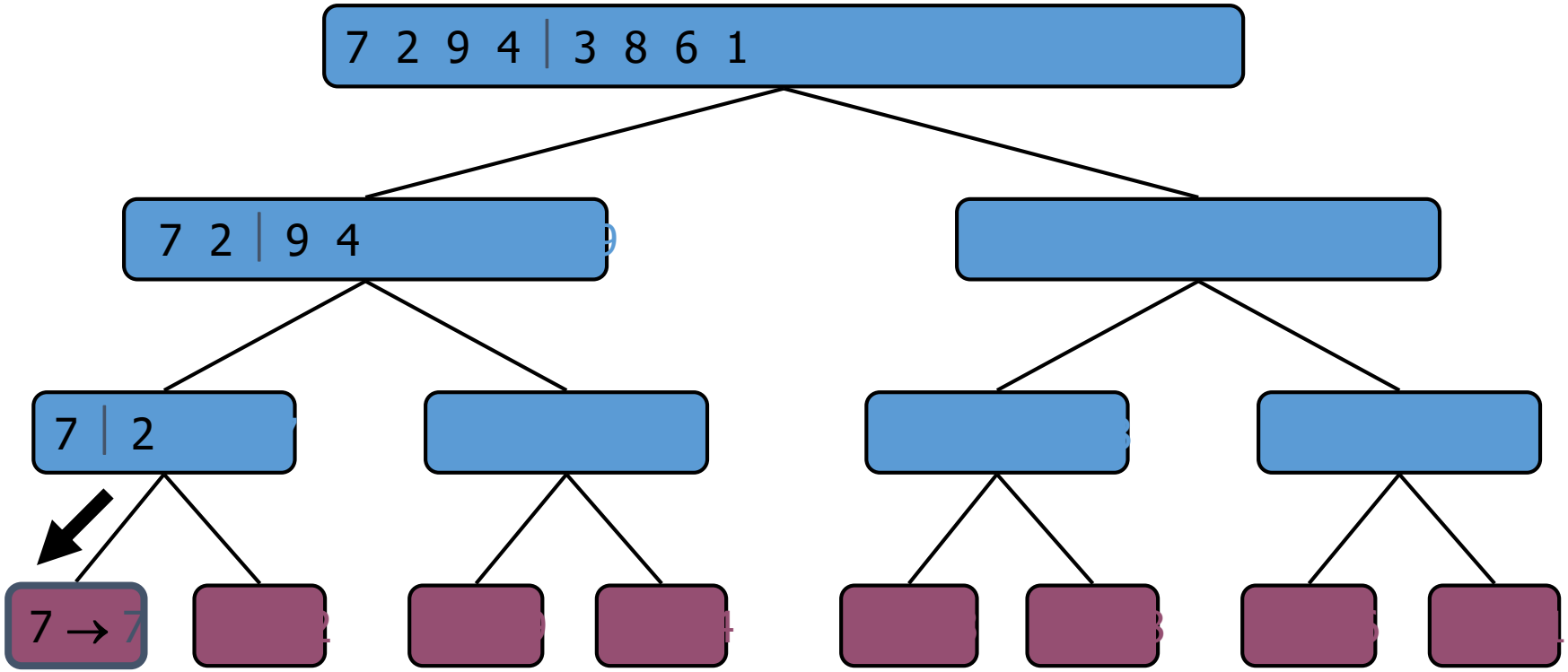
Execution Example (cont.)

- **Recursive call, partition**



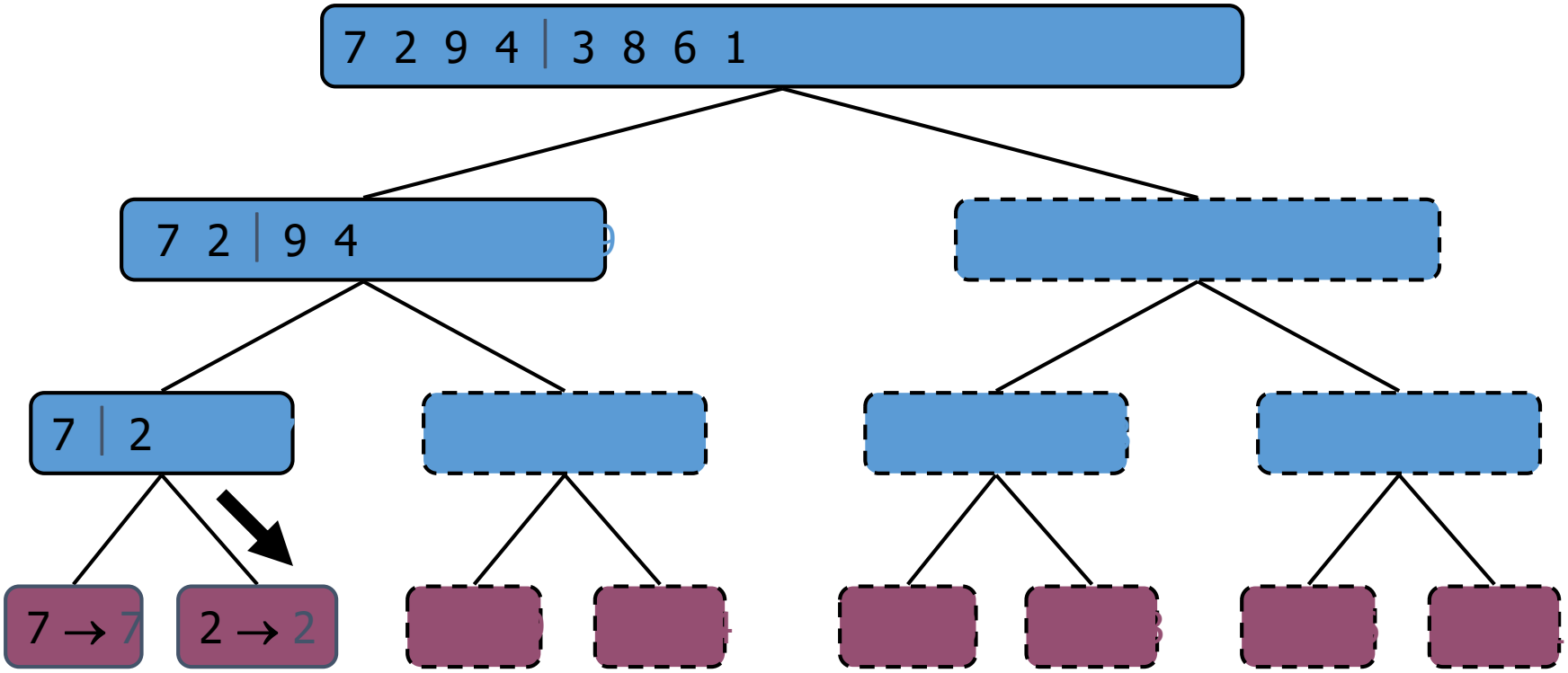
Execution Example (cont.)

- **Recursive call, base case**



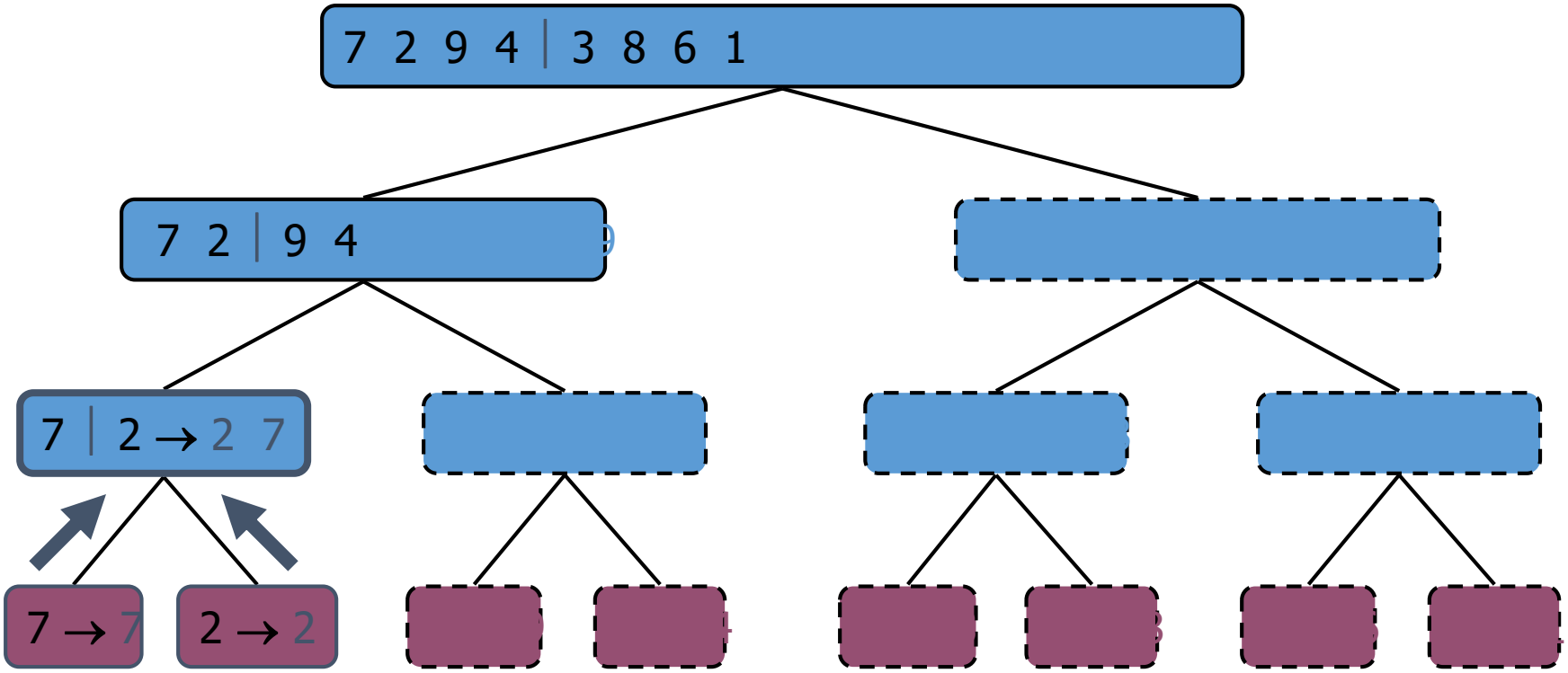
Execution Example (cont.)

- **Recursive call, base case**



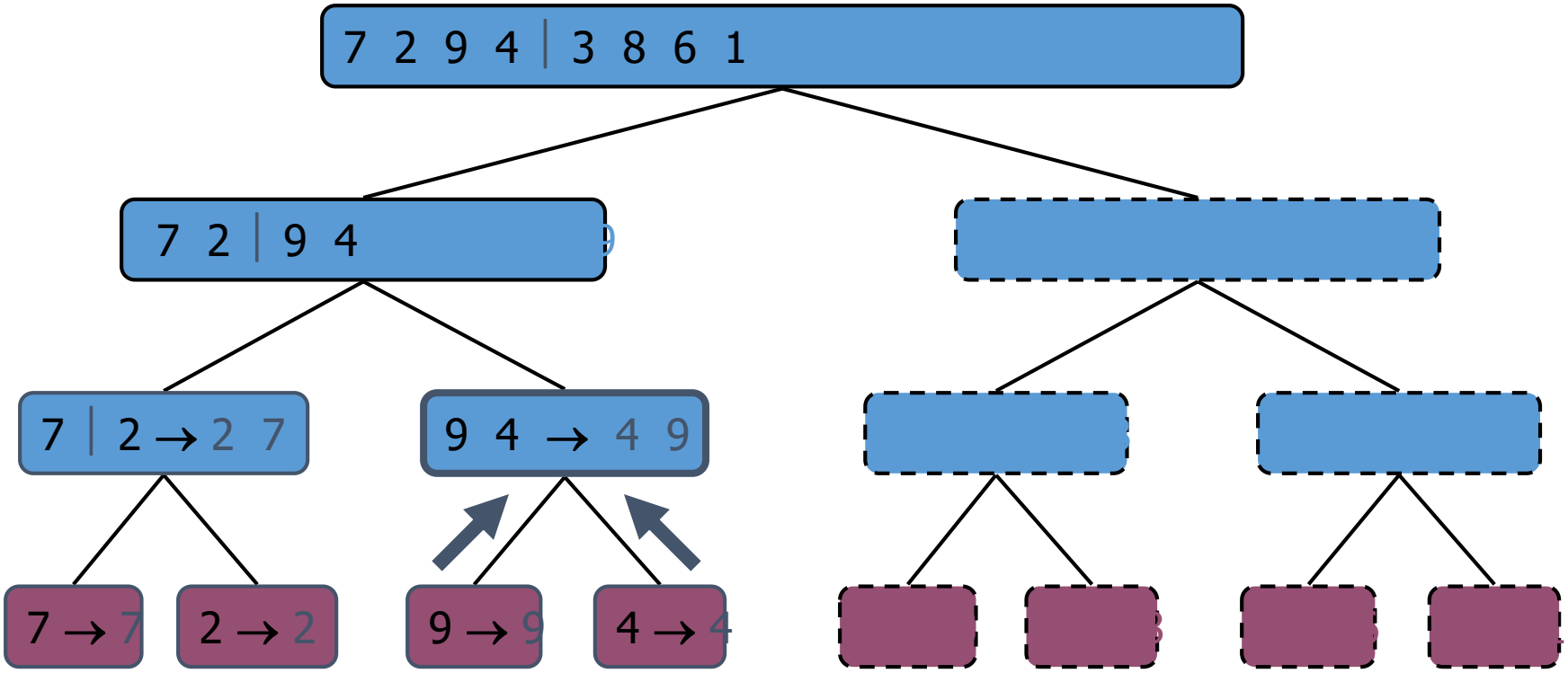
Execution Example (cont.)

- **Merge**



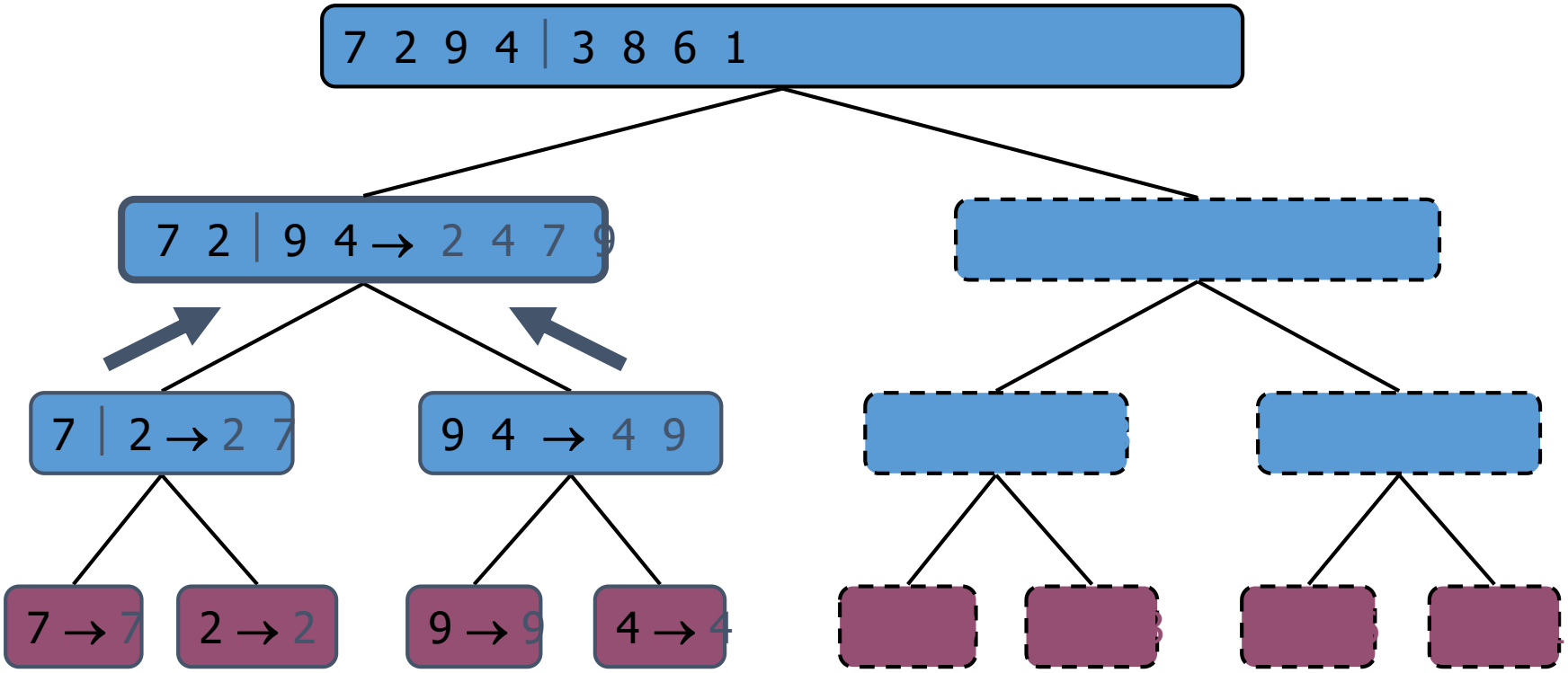
Execution Example (cont.)

- **Recursive call, …, base case, merge**



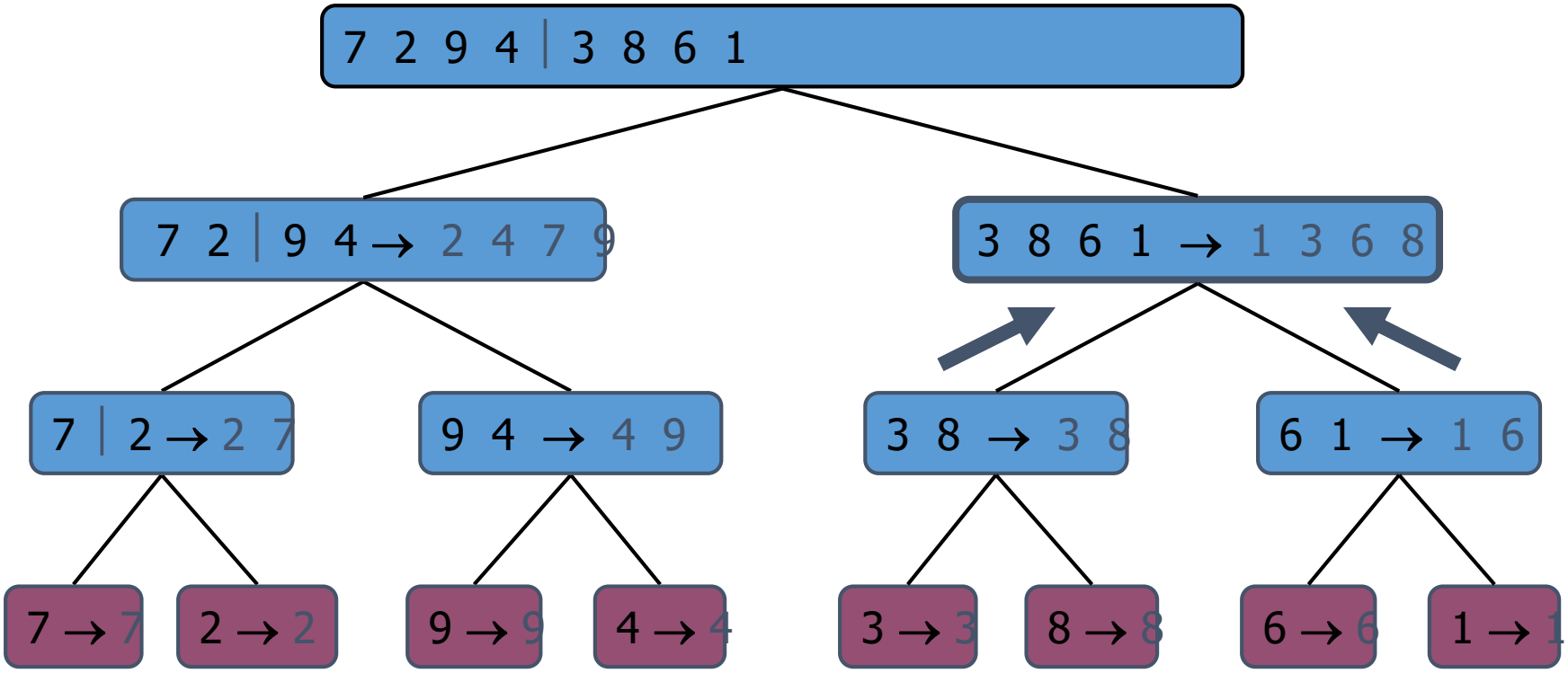
Execution Example (cont.)

- **Merge**



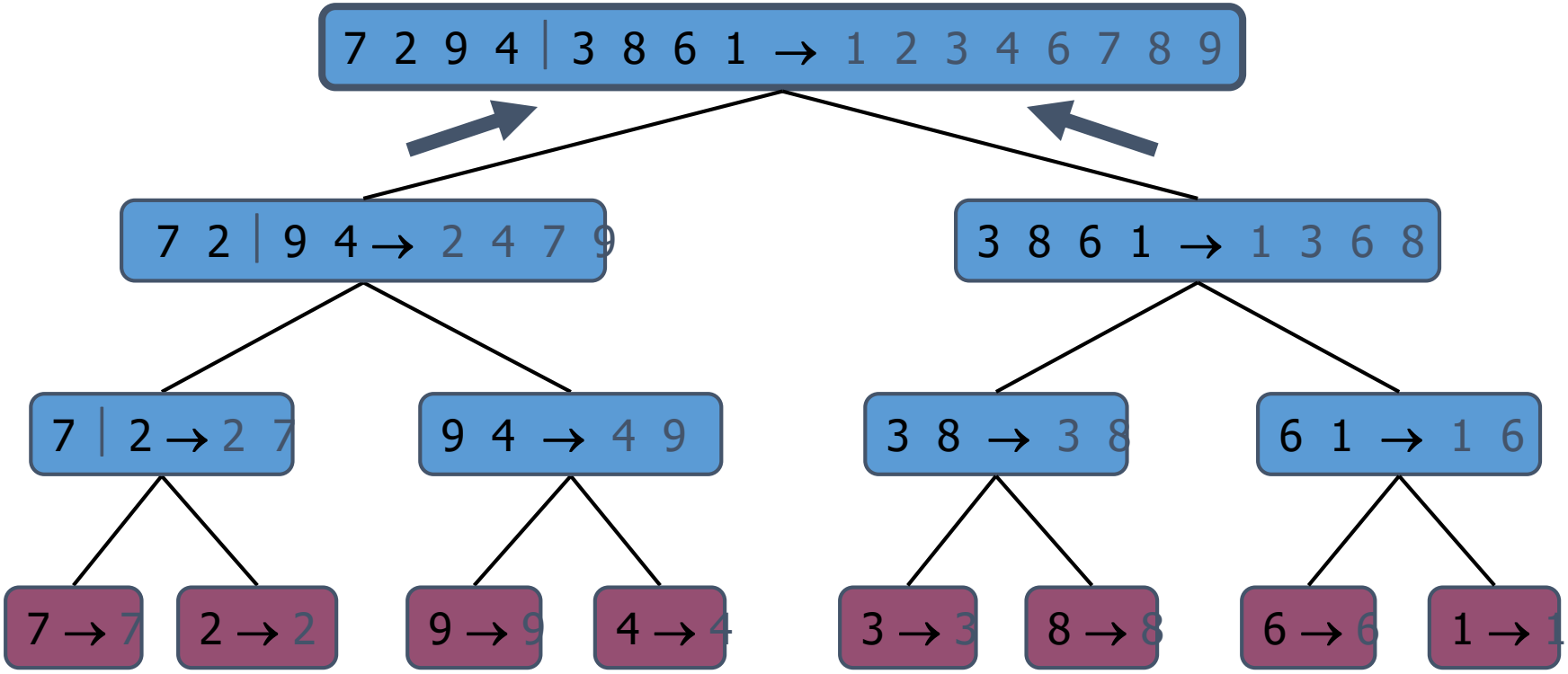
Execution Example (cont.)

- **Recursive call, …, merge, merge**



Execution Example (cont.)

- **Merge**

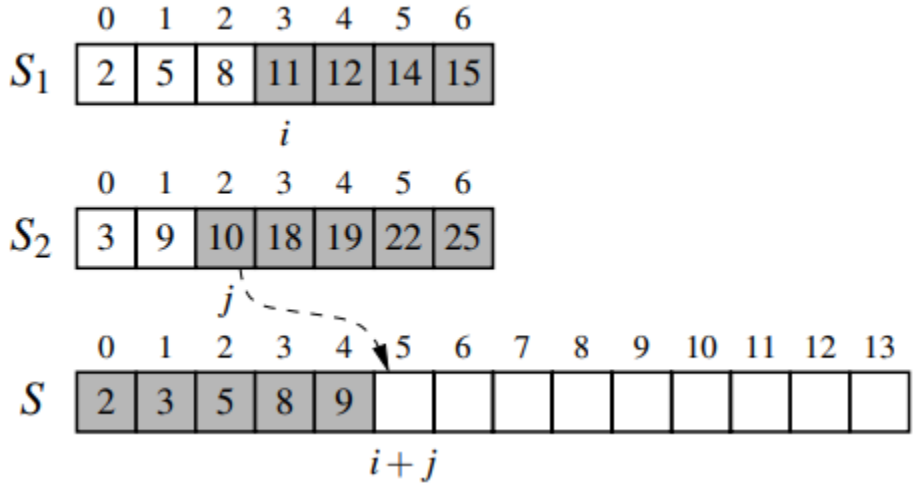


Merge-sort

1. ***Divide:*** If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), 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; that is, S_1 contains the first $\lfloor n/2 \rfloor$ elements of S, and S_2 contains the remaining $\lceil n/2 \rceil$ elements.
2. ***Conquer:*** Recursively sort sequences S_1 and S_2.
3. ***Combine:*** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

How to merge S1, S2 into S?

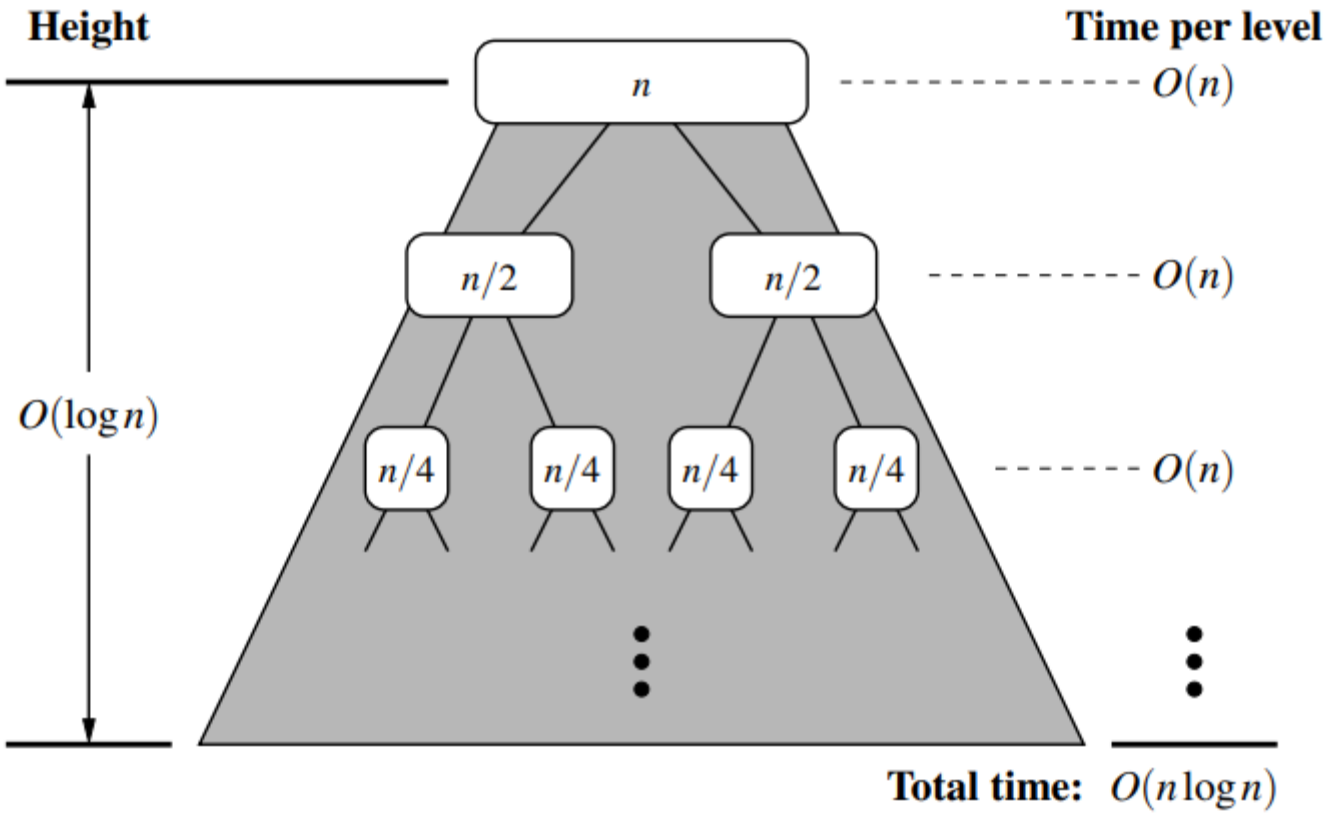
```
def merge(S1, S2, S):
  """Merge two sorted Python lists S1 and S2 into properly sized list S."""
  i = j = 0
  while i + j < len(S):
    if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
      S[i+j] = S1[i]                  # copy ith element of S1 as next item of S
      i += 1
    else:
      S[i+j] = S2[j]                  # copy jth element of S2 as next item of S
      j += 1
```



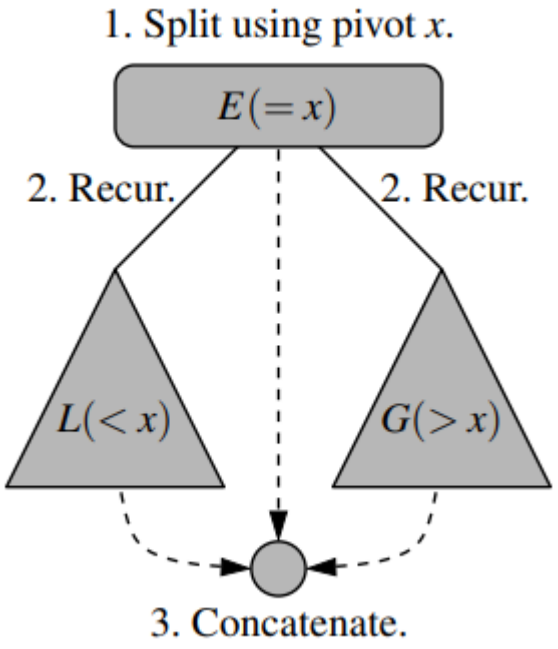
Merge-sort algorithm

```
1  def merge_sort(S):
2    """Sort the elements of Python list S using the merge-sort algorithm."""
3    n = len(S)
4    if n < 2:
5      return                    # list is already sorted
6    # divide
7    mid = n // 2
8    S1 = S[0:mid]               # copy of first half
9    S2 = S[mid:n]               # copy of second half
10   # conquer (with recursion)
11   merge_sort(S1)              # sort copy of first half
12   merge_sort(S2)              # sort copy of second half
13   # merge results
14   merge(S1, S2, S)            # merge sorted halves back into S
```

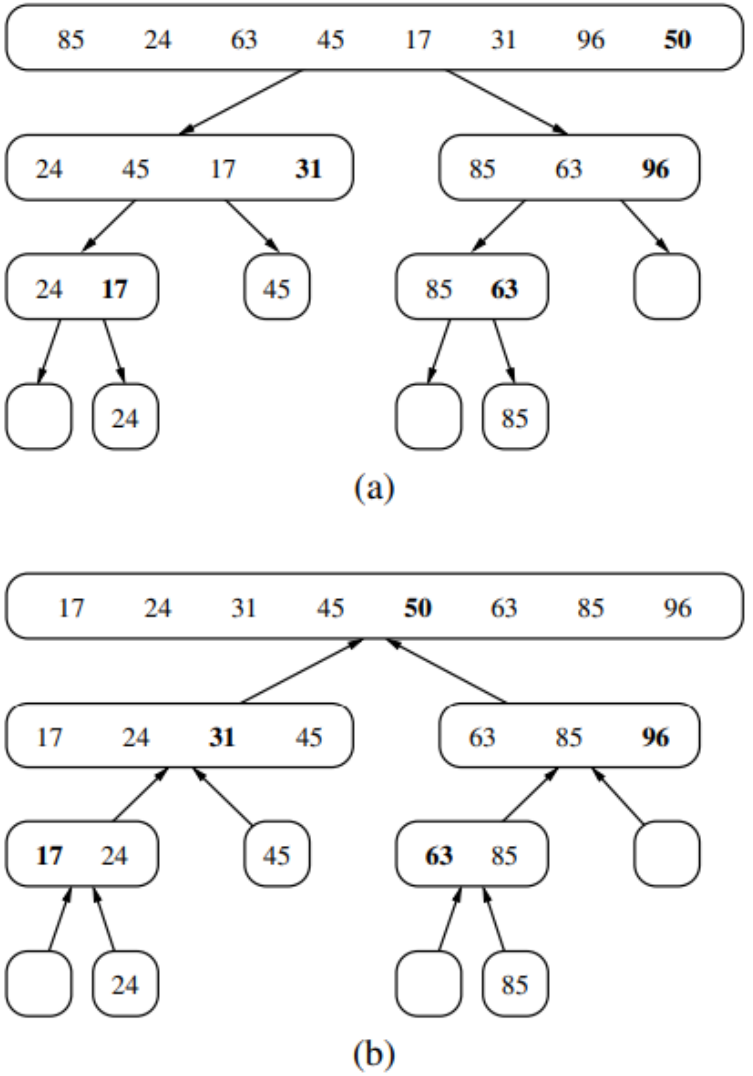
Merge-sort: running time



Quick-sort: another DnC method



Also, O(nlogn)



Quick-sort: another DnC method

1. ***Divide:*** If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S, which is called the ***pivot***. As is common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences:
   - L, storing the elements in S less than x
   - E, storing the elements in S equal to x
   - G, storing the elements in S greater than x

   Of course, if the elements of S are distinct, then E holds just one element—the pivot itself.
2. ***Conquer:*** Recursively sort sequences L and G.
3. ***Combine:*** Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.