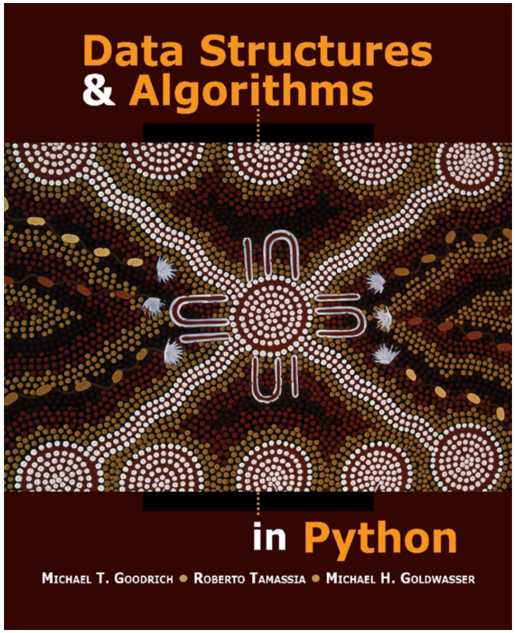
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
HUMANITIES
Department of General and Computational Linguistics

# Sorting: Insertion Sort & Quick Sort

Data Structures and Algorithms for CL III, WS 2019-2020

**Corina Dima**
corina.dima@uni-tuebingen.de




## 12. Sorting and Selection

- Why study sorting algorithms?
- Insertion sort (S. 5.5.2)
- Quick-sort
- Optimizations for quick-sort

# Why Study Sorting Algorithms?

- Sorting is among the most important and well studied computing problems
- Data is often stored in sorted order, to allow for efficient searching (i.e. with binary search)
- Many algorithms rely on sorting as a subroutine
- Programming languages have highly optimized, built-in sorting functions – which should be used
  - Python: `sorted()`, `sort()` from the `list` class
  - Java: `Arrays.sort()`

# Why Study Sorting Algorithms? (cont'd)

- Understand
  - what sorting algorithms do
  - what can be expected in terms of efficiency
  - how the efficiency of a sorting algorithm can depend on the initial ordering of the data or the type of objects being sorted

# Sorting

- Given a collection, rearrange its elements such that they are ordered from smallest to largest – or produce a new copy of the sequence with such an order
- We assume that such a consistent order exists
- In Python, natural order is typically defined using the < operator, having two properties:
  - Irreflexive property: $k \nless k$
  - Transitive property: if $k_1 < k_2$ and $k_2 < k_3$ then $k_1 < k_3$

# Sorting Algorithms

- Many different sorting algorithms available
  - Insertion Sort
  - Quick-sort
  - Bucket-sort
  - Radix-sort
  - Merge-sort (recap)
  - Selection Sort
  - Heap-sort (next lecture)
  - Timsort (next lecture)

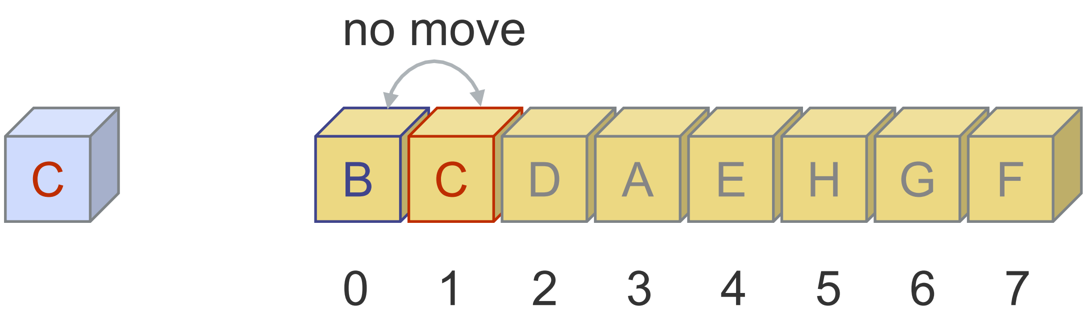
# Insertion Sort

# Insertion Sort

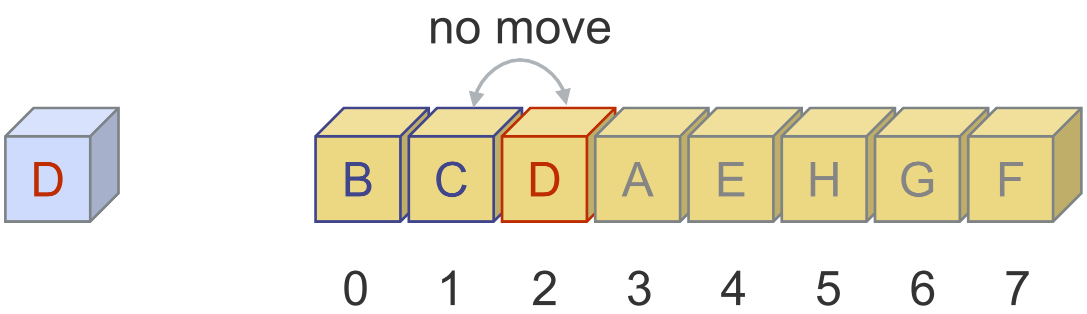
- Insertion sort is a simple algorithm for sorting an array
- Start with the first element of the array
  - one element by itself is already sorted
- Consider the next element of the array
  - if smaller than the first, swap them
- Consider the third element
  - Swap it leftward, until it is in proper order with the first two elements
- Continue in this manner with the fourth, fifth, etc. until the whole array is sorted

# Insertion Sort - Example

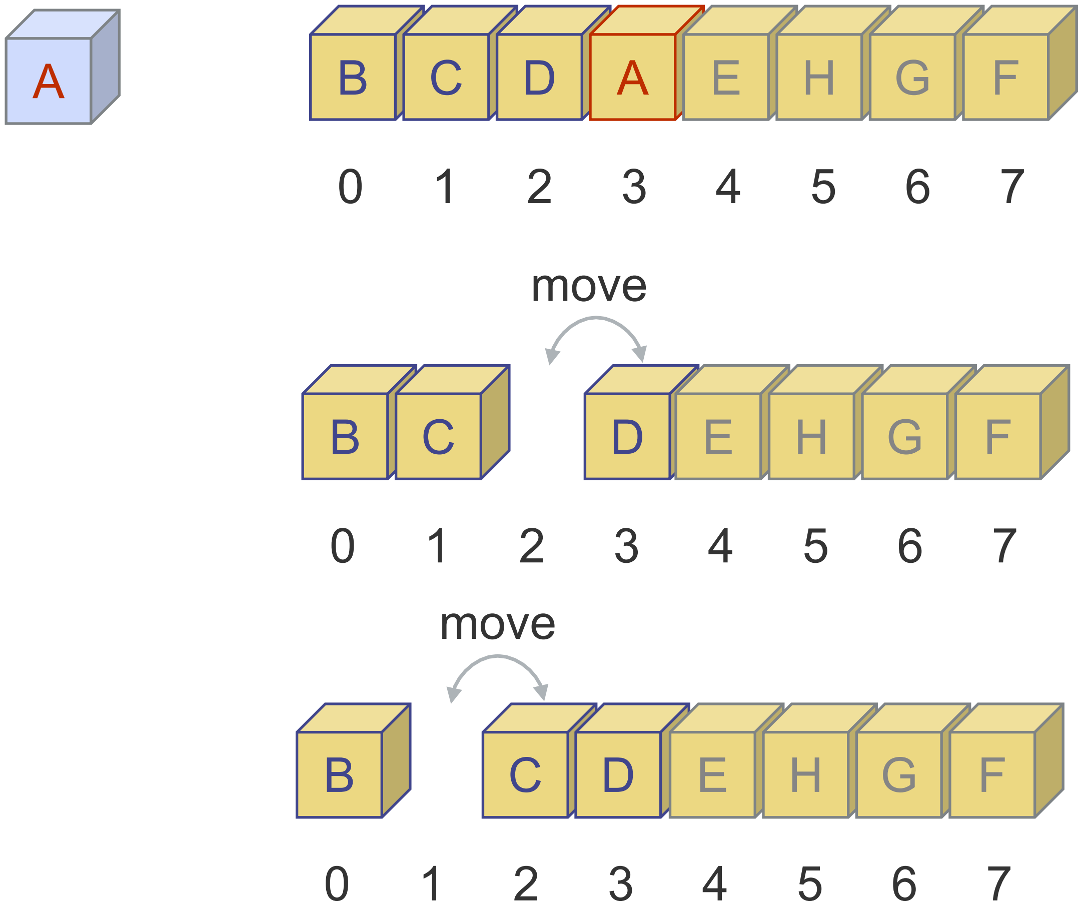
# Insertion Sort – Example (cont'd)



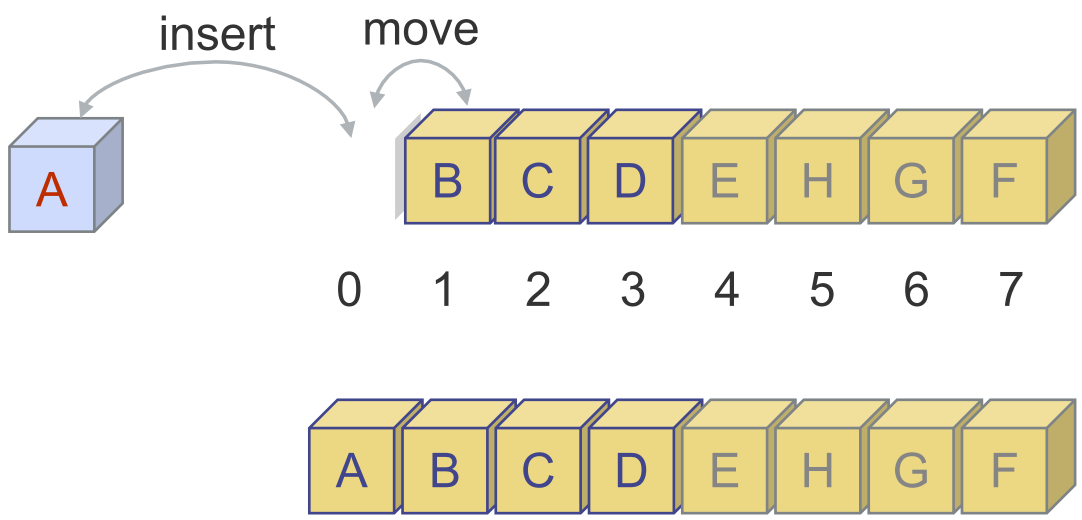
# Insertion Sort – Example (cont'd)



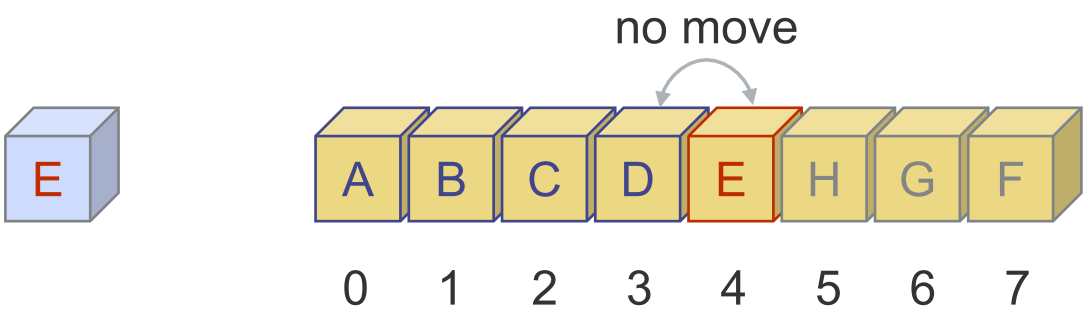
# Insertion Sort – Example (cont’d)



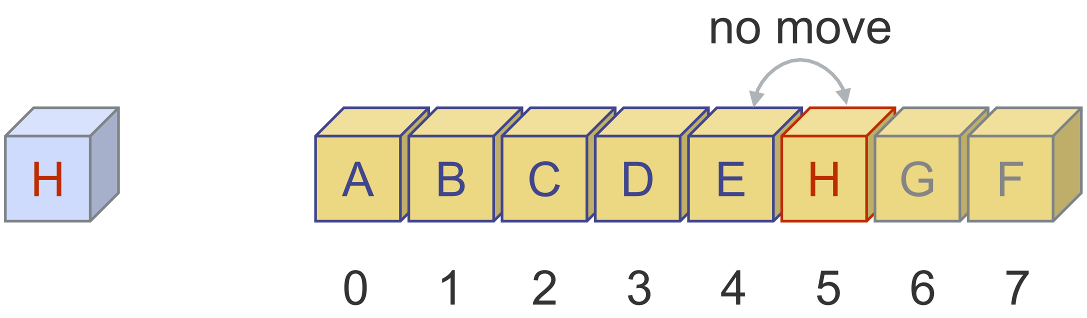
# Insertion Sort – Example (cont'd)



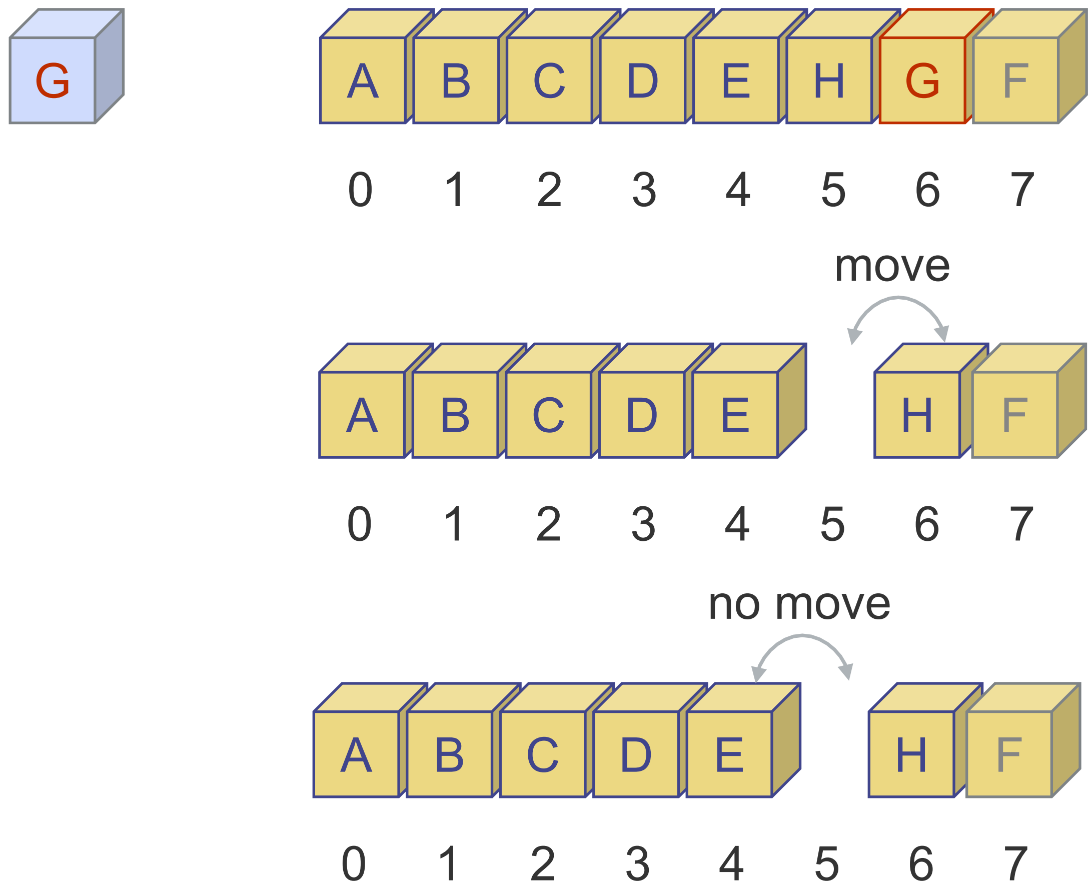
# Insertion Sort – Example (cont'd)



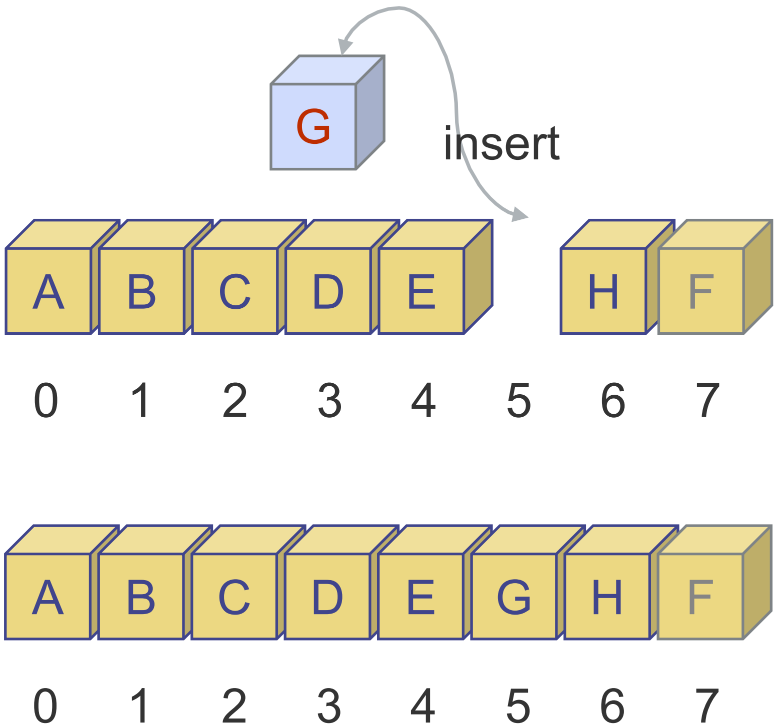
# Insertion Sort – Example (cont’d)



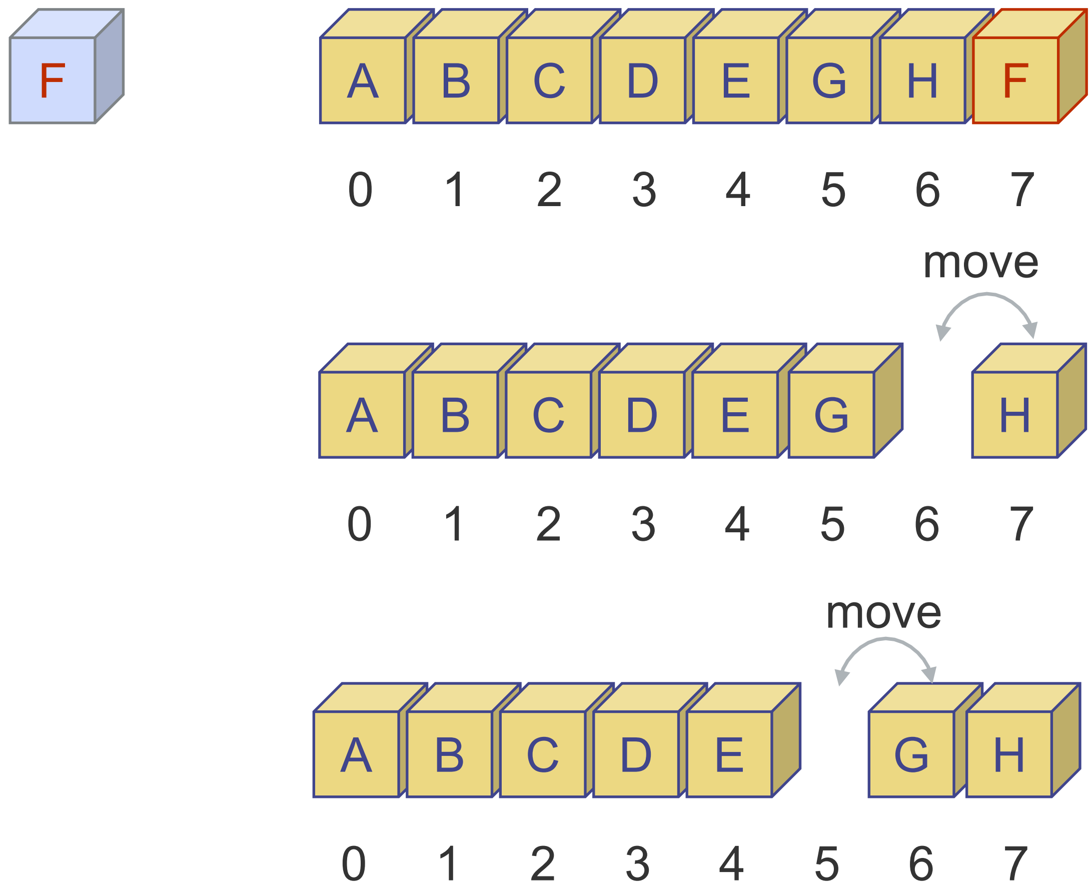
# Insertion Sort – Example (cont'd)



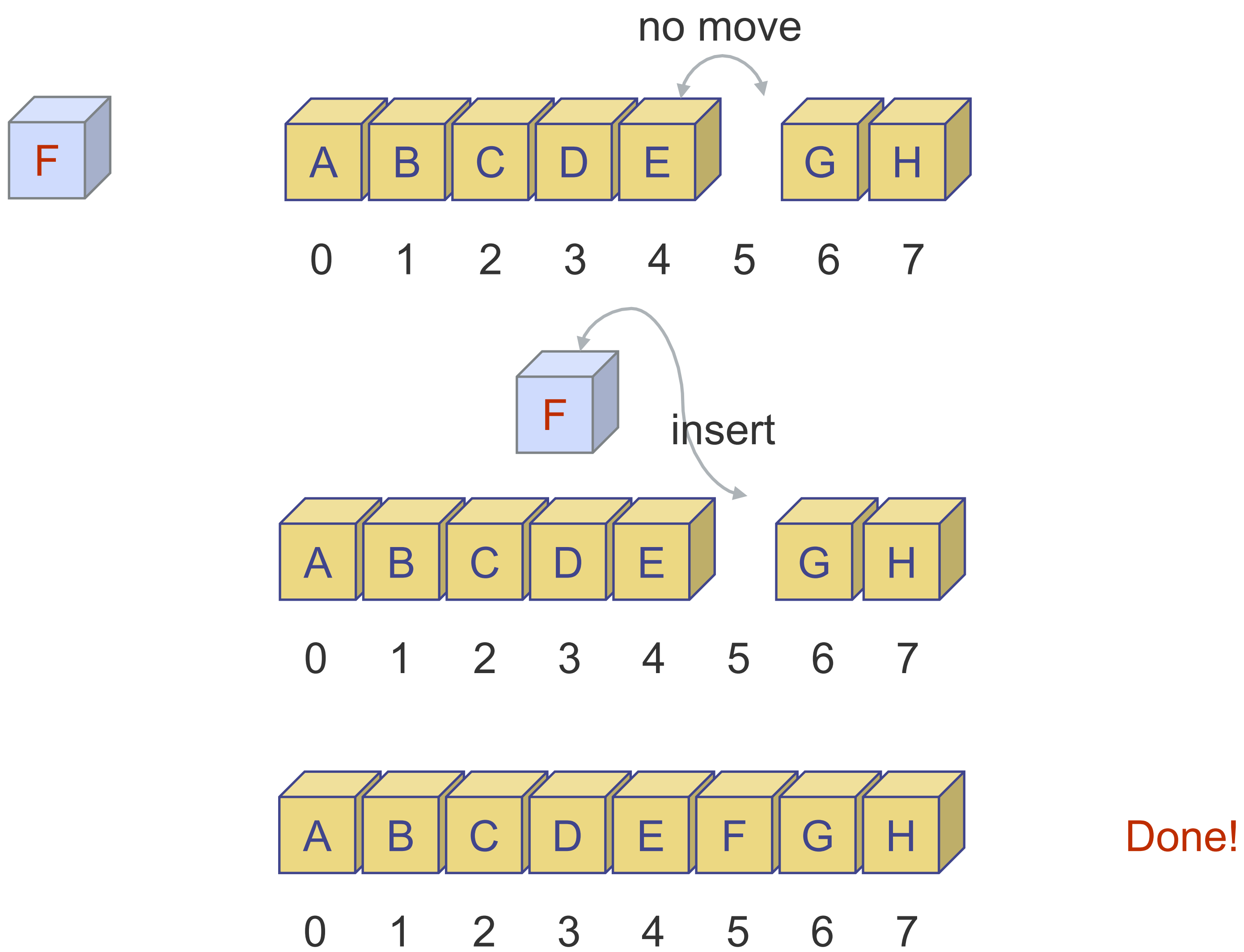
# Insertion Sort – Example (cont'd)



# Insertion Sort – Example (cont'd)



# Insertion Sort – Example (cont’d)



# Insertion Sort – Python code

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

# Insertion Sort - Complexity

- Worst case complexity?
  - Array in reverse order
  - Outer loop executes $n-1$ times
  - Inner loop executes $1 + 2 + \dots (n - 1) = ?$
- Best case complexity?
  - Array is sorted or almost sorted
  - Outer loop executes $n-1$ times
  - Inner loop executes few times, or doesn't execute

# Insertion Sort - Complexity

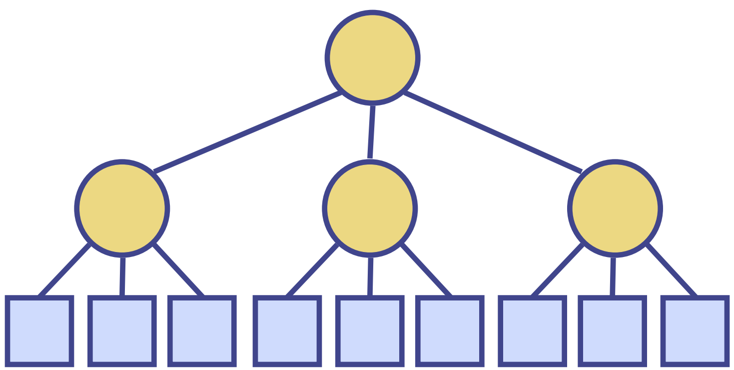
- Worst case complexity: $O(n^2)$
  - Array in reverse order
  - Outer loop executes $n-1$ times
  - Inner loop executes $1 + 2 + \ldots (n - 1) = \frac{(n-1)n}{2}$ times

- Best case complexity: $O(n)$
  - Array is sorted or almost sorted
  - Outer loop executes $n-1$ times
  - Inner loop executes few times, or doesn't execute

# Quick-Sort

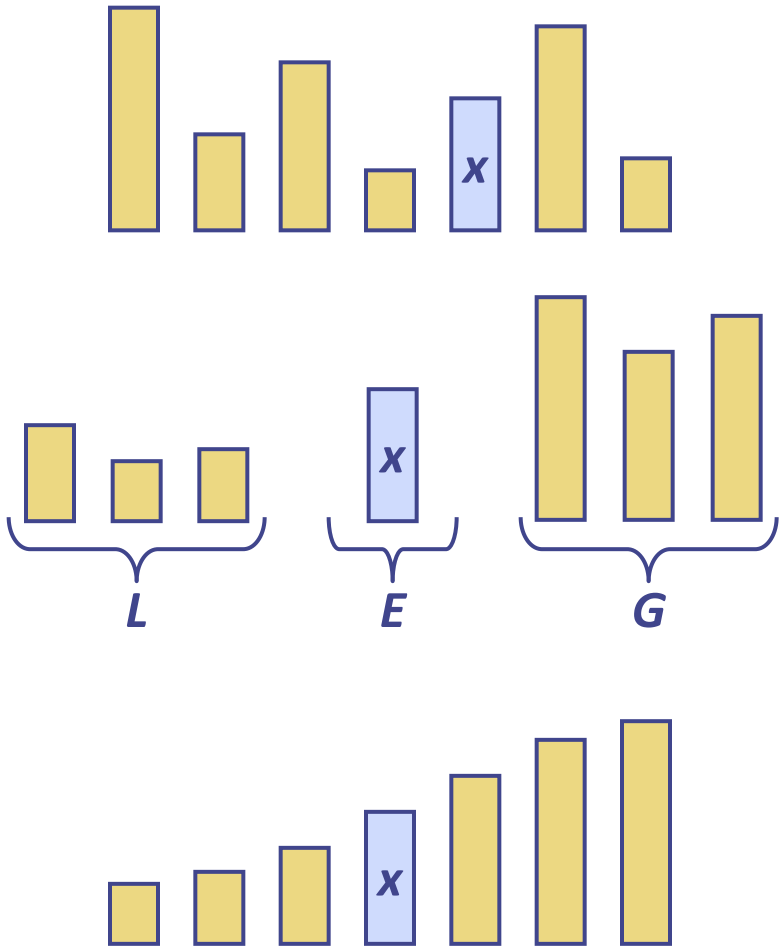
# Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  1. Divide divide the input data $S$ in two or more disjoint subsets $S_1, S_2, \ldots$
  2. Recur: solve the subproblems recursively
  3. Conquer: combine the solutions for $S_1, S_2, \ldots,$ into a solution for $S$
- The base case for the recursion are subproblems of constant size
- Analysis can be done using recurrence equations



# Quick-Sort

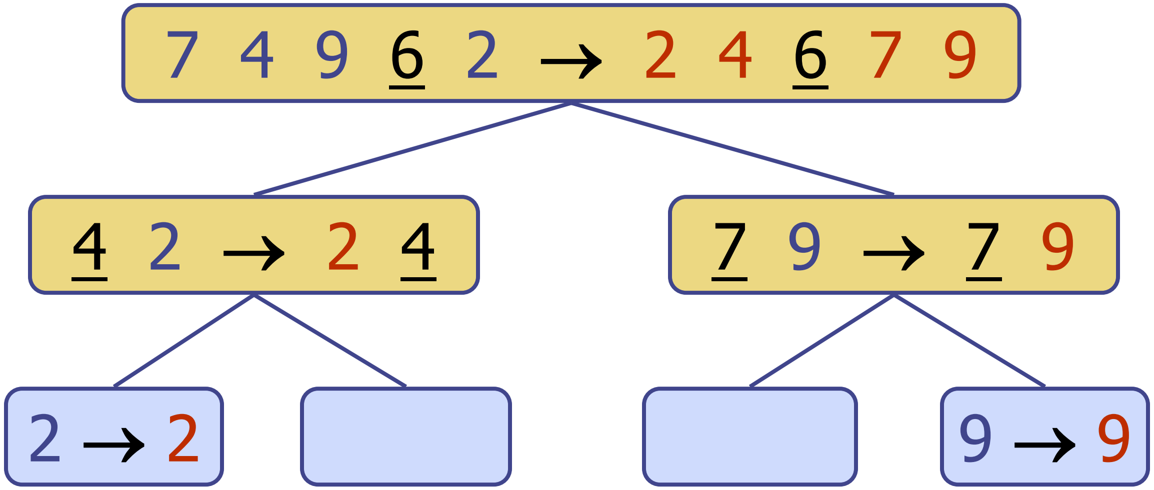
- Quick-sort is a sorting algorithm based on the divide-and-conquer paradigm; consists of three steps:
  - Divide: pick a element $x$ - called pivot, typically the last element - and partition $S$ into
    - $L$ elements less than $x$
    - $E$ elements equal $x$
    - $G$ elements greater than $x$
  - Recur: sort ***L*** and ***G***
  - Conquer: join ***L***, ***E*** and ***G***



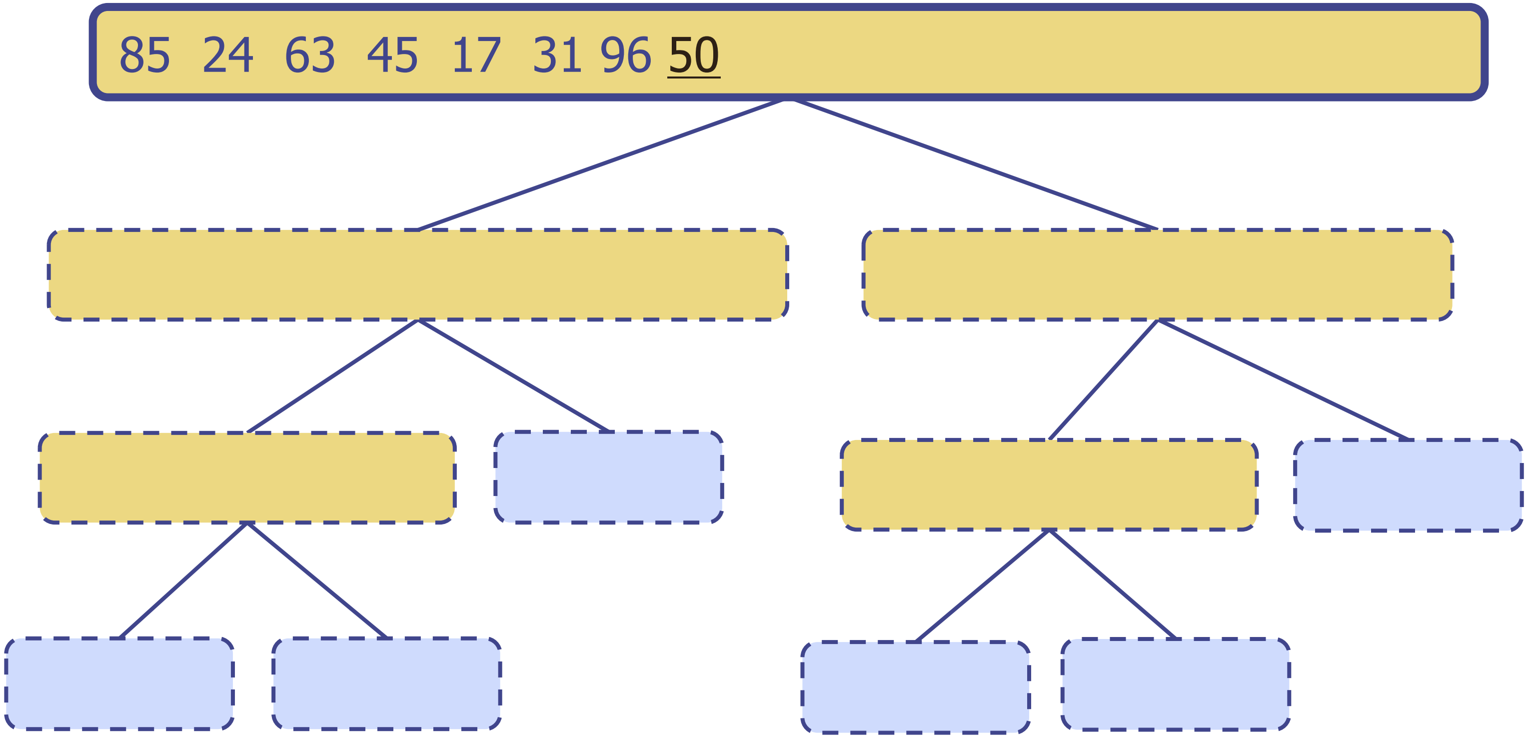

UNIVERSITÄT TÜBINGEN

# Quick-Sort Tree

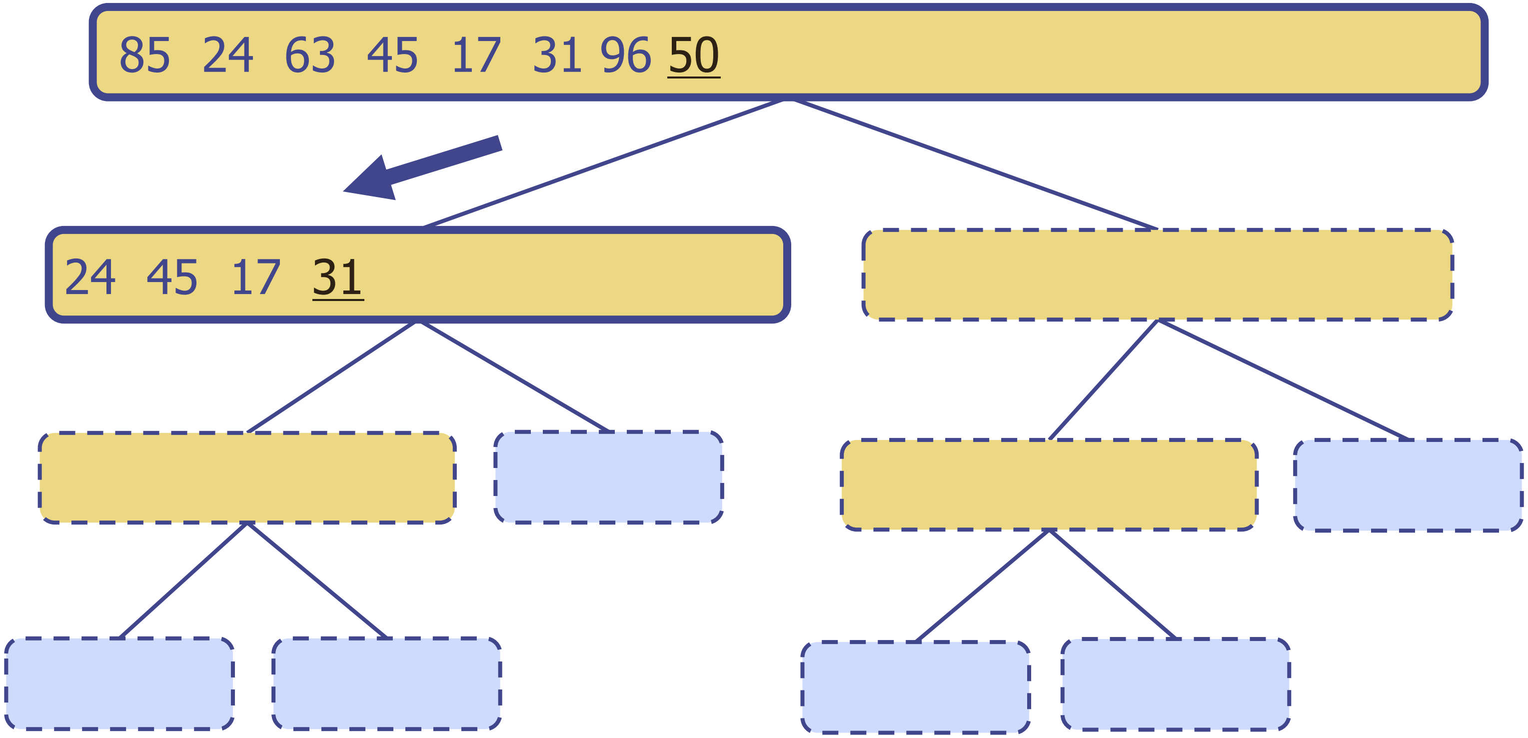
- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



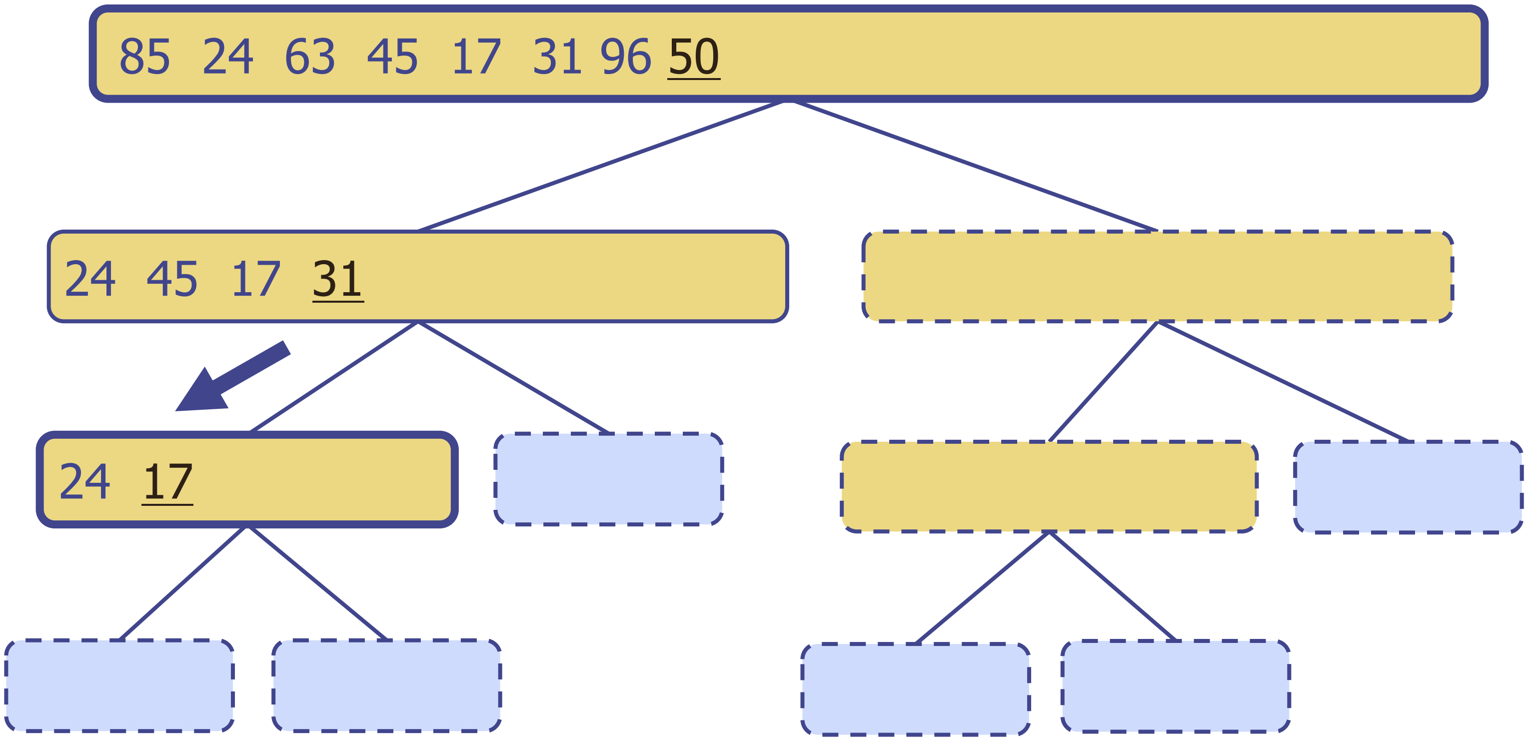
# Execution Example



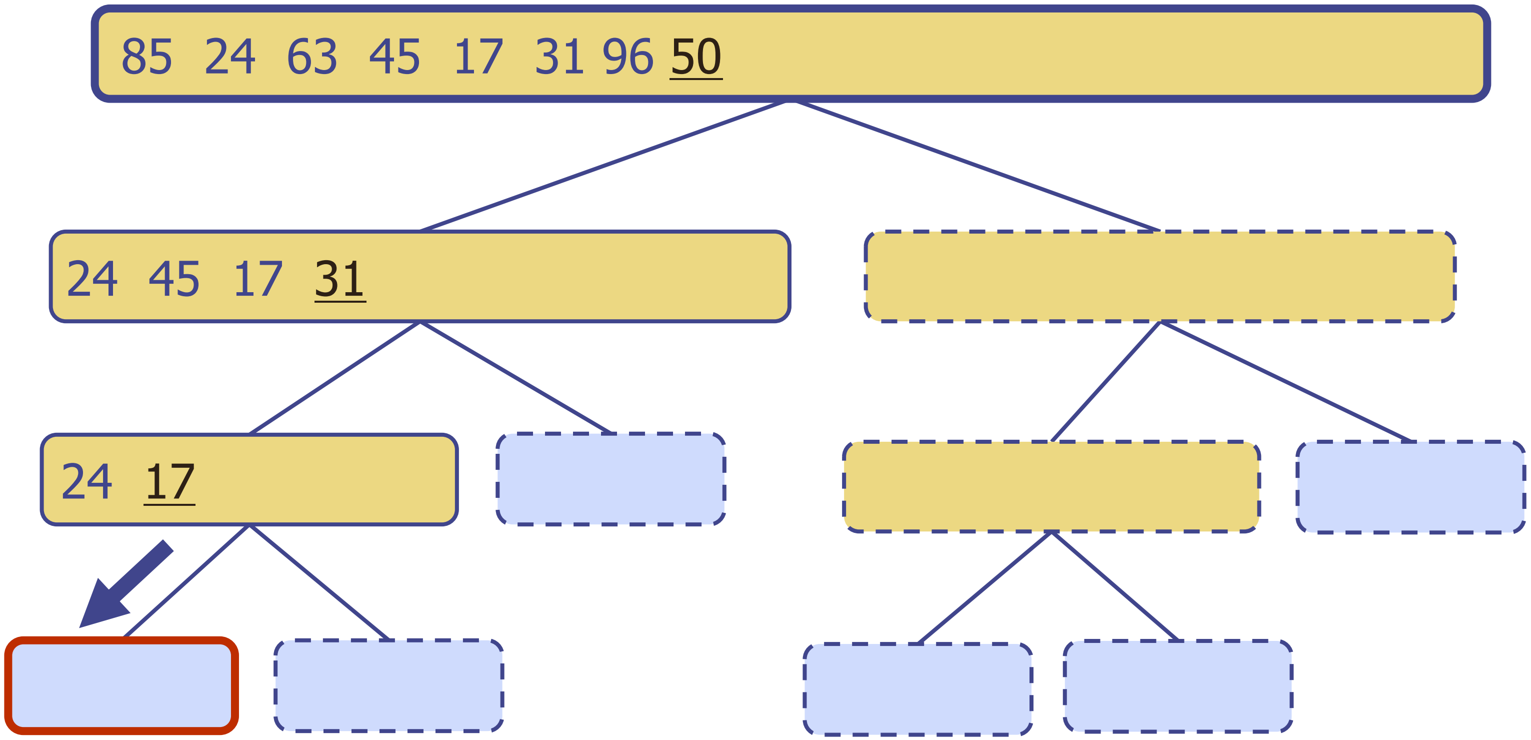
# Execution Example (cont'd)



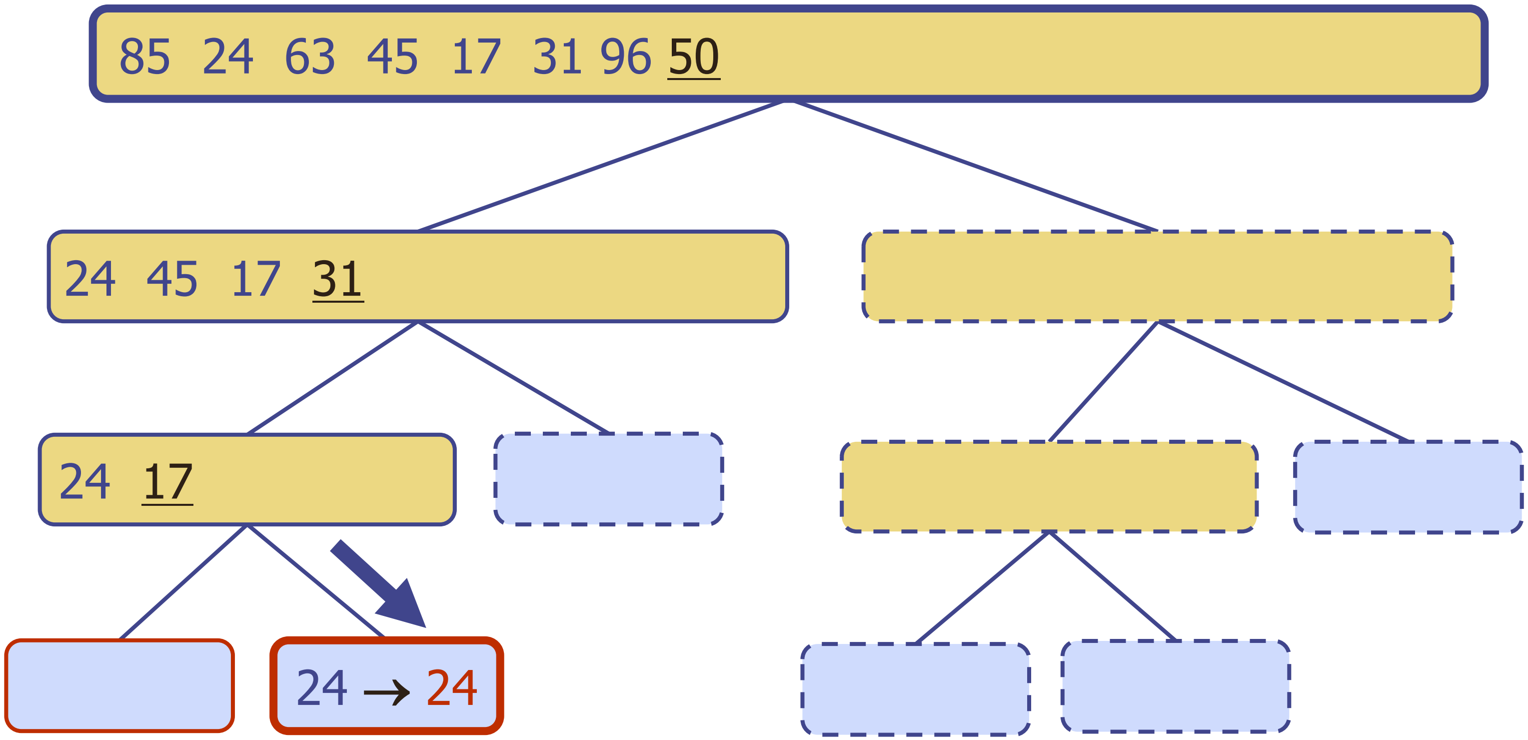
# Execution Example (cont'd)



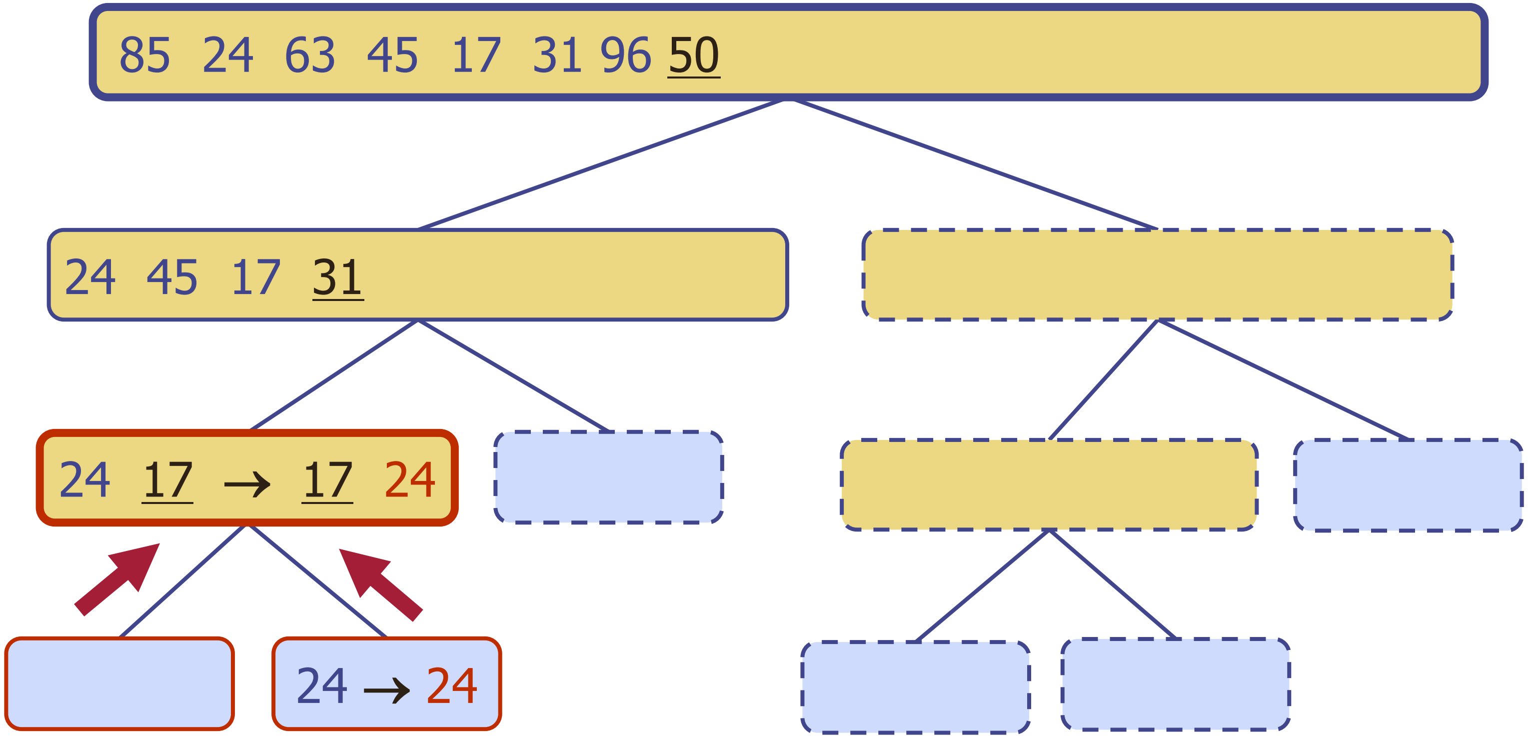
# Execution Example (cont'd)



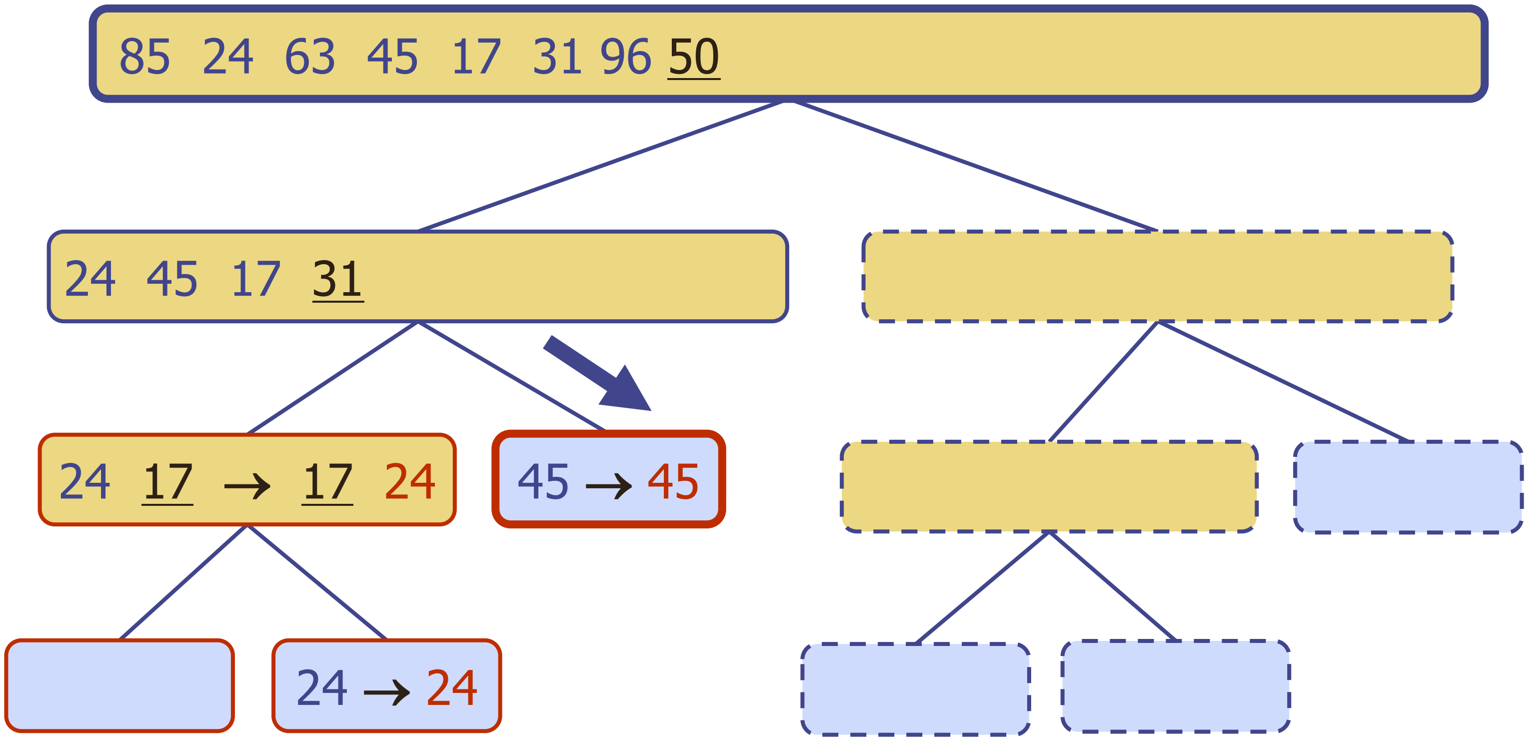
# Execution Example (cont'd)



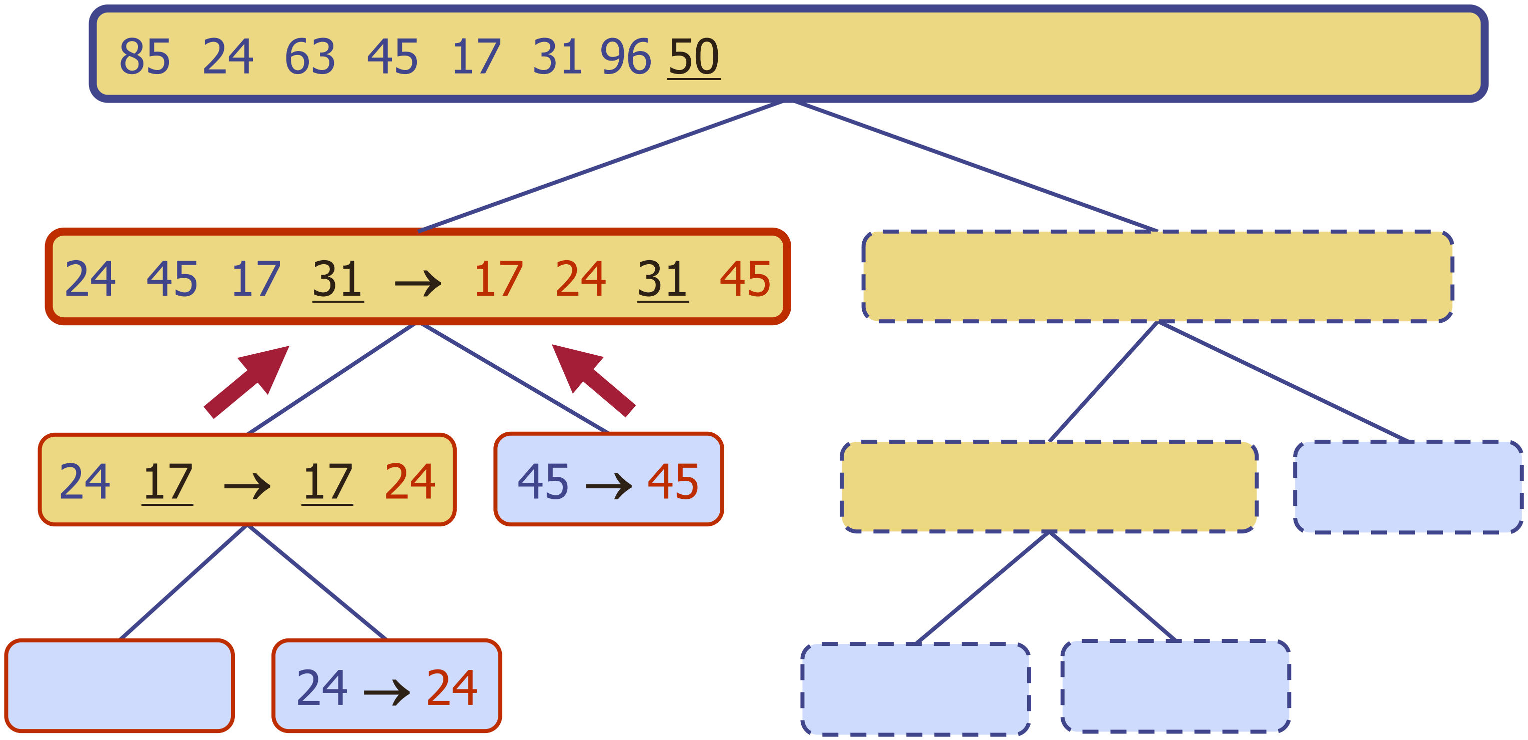
# Execution Example (cont'd)



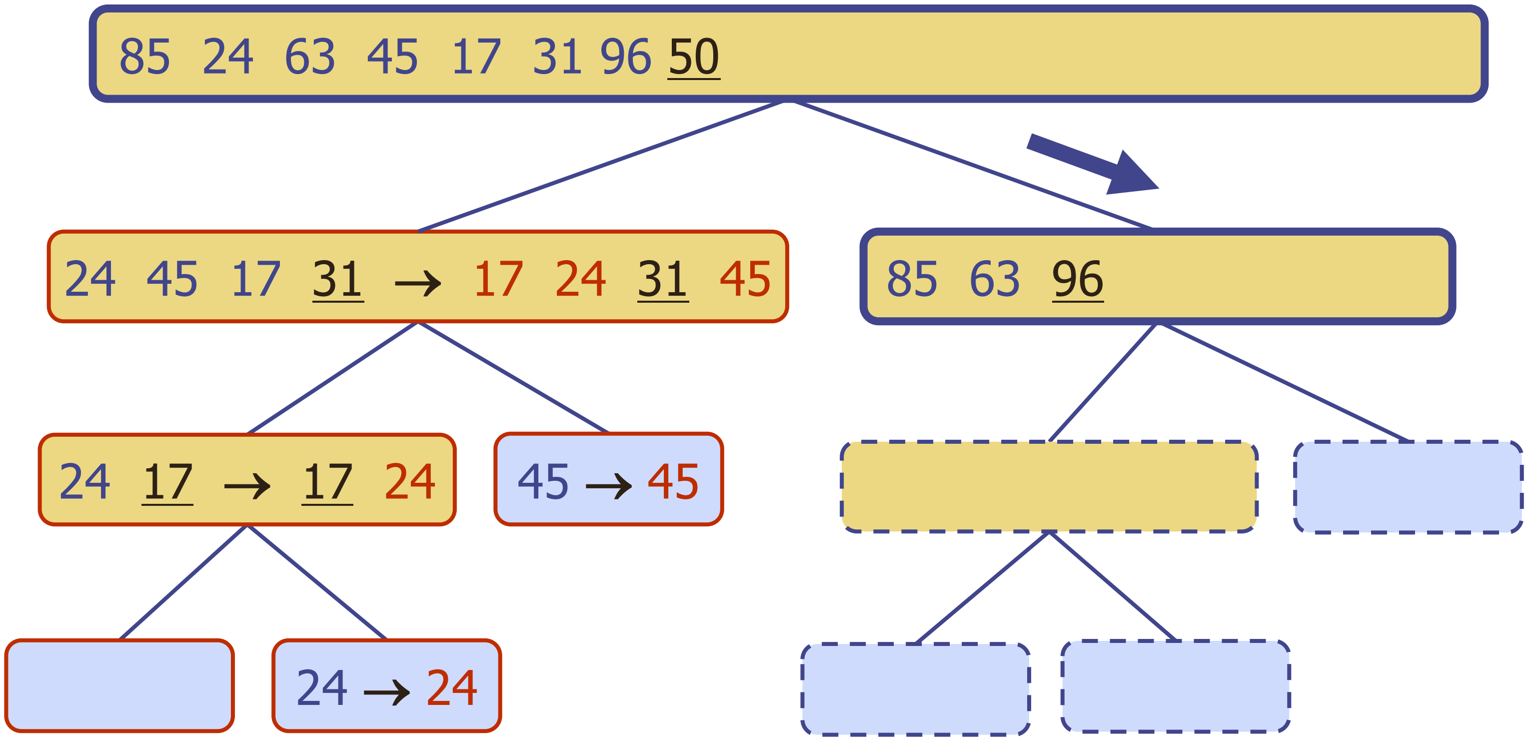
# Execution Example (cont'd)



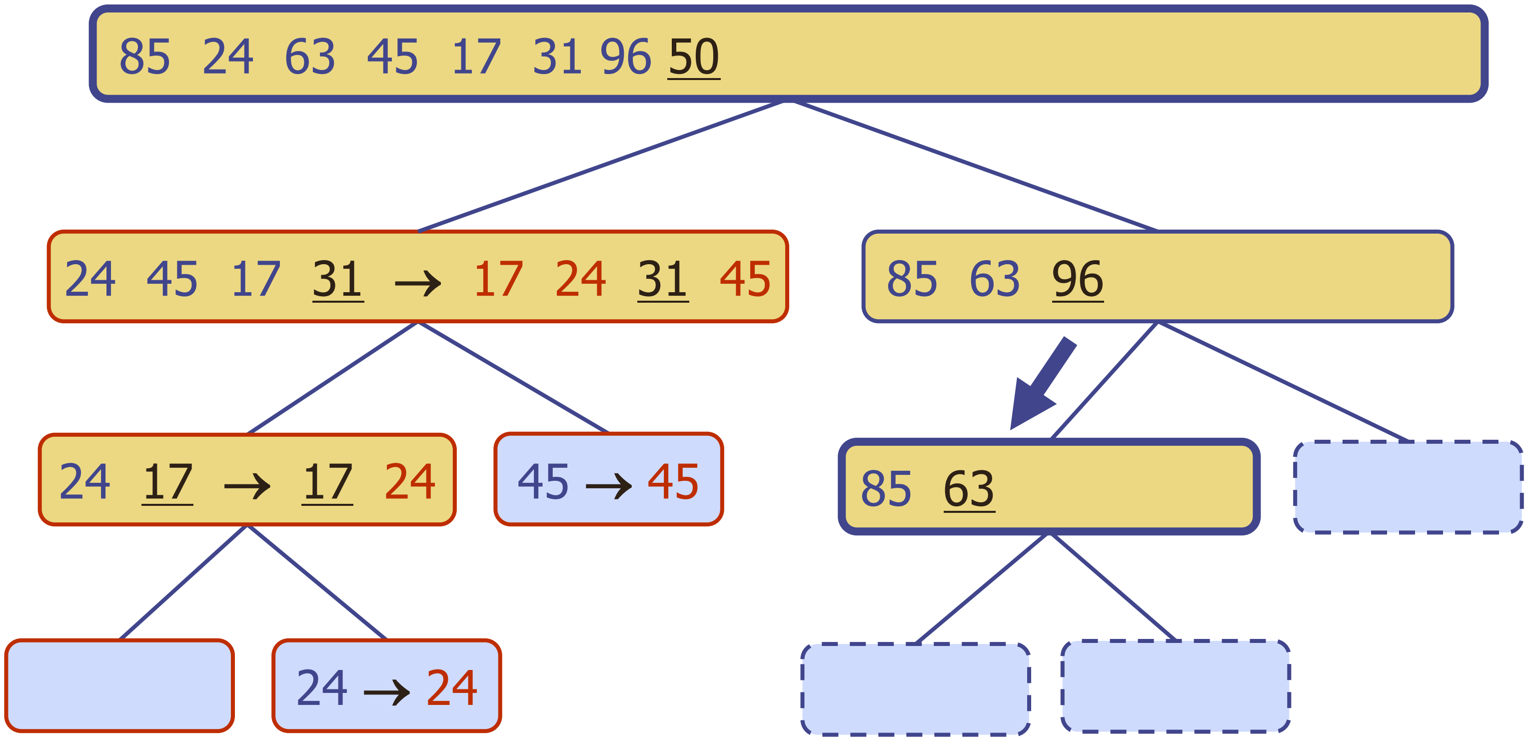
# Execution Example (cont'd)



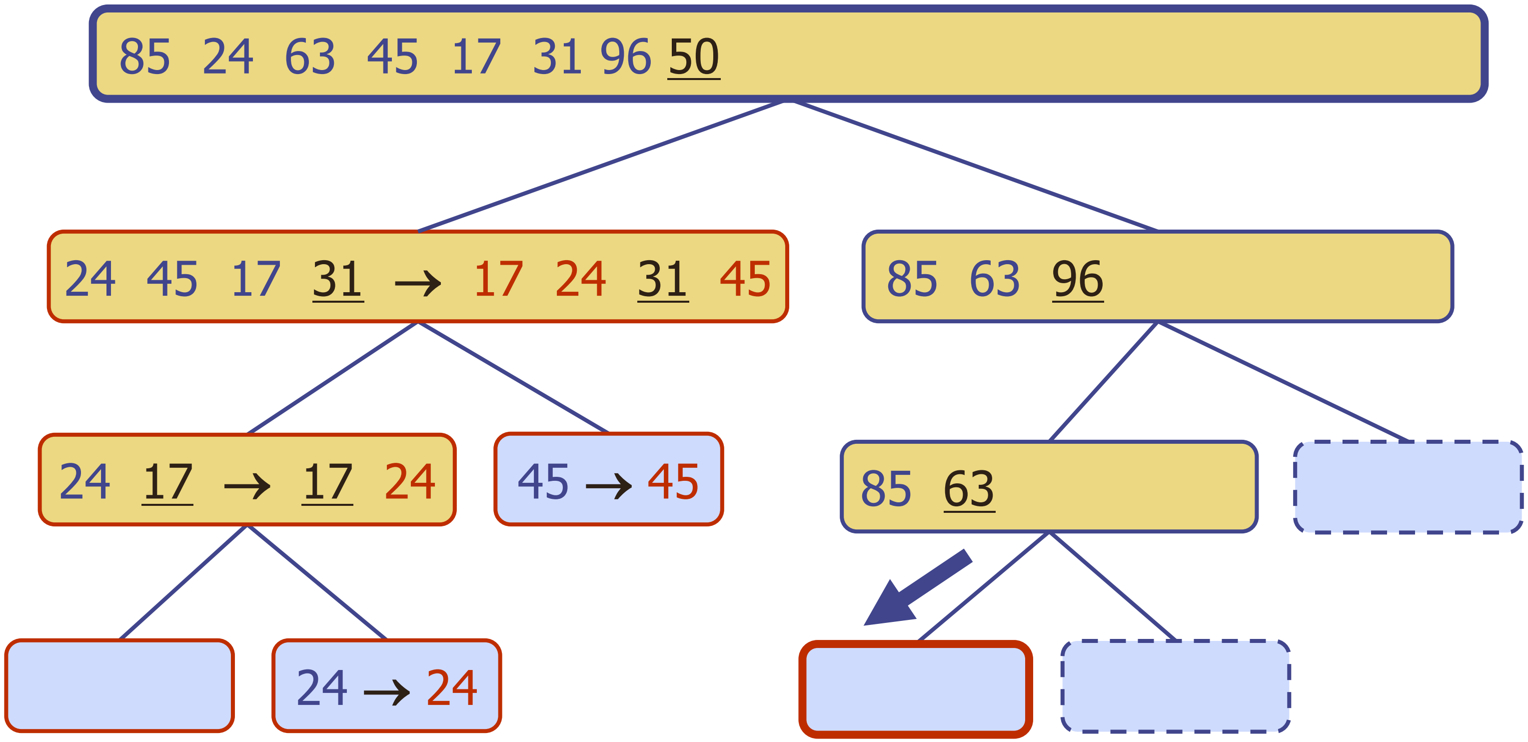
# Execution Example (cont'd)



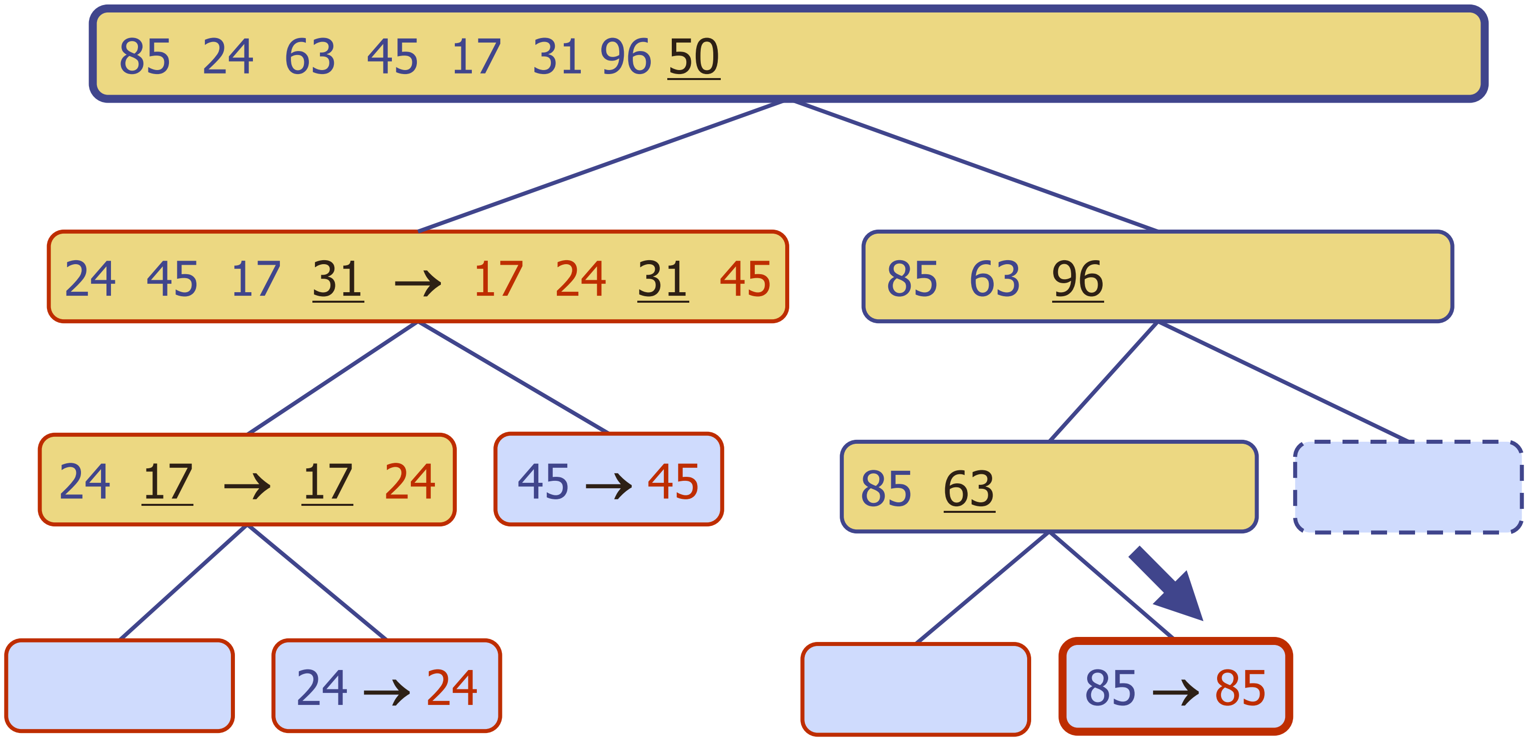
# Execution Example (cont'd)



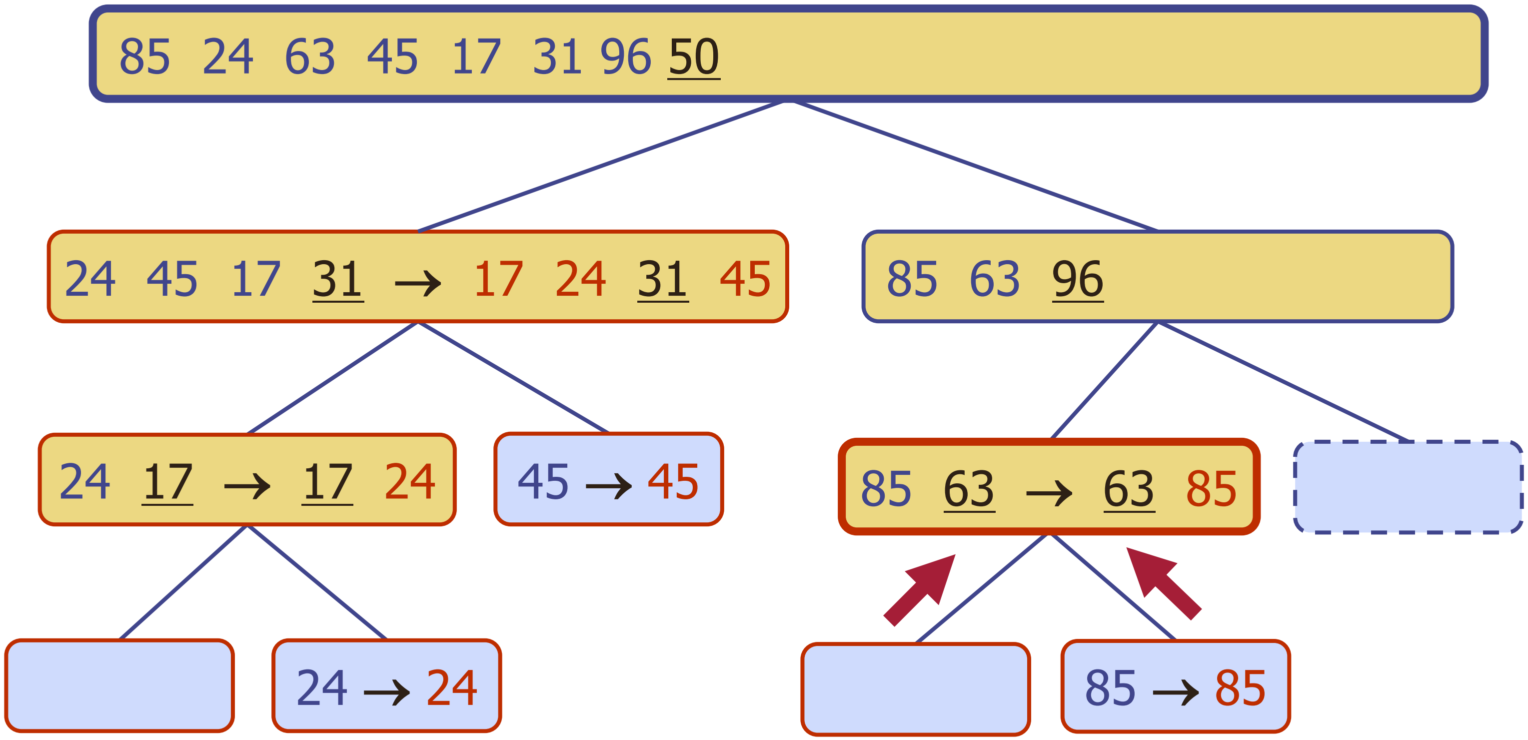
# Execution Example (cont'd)



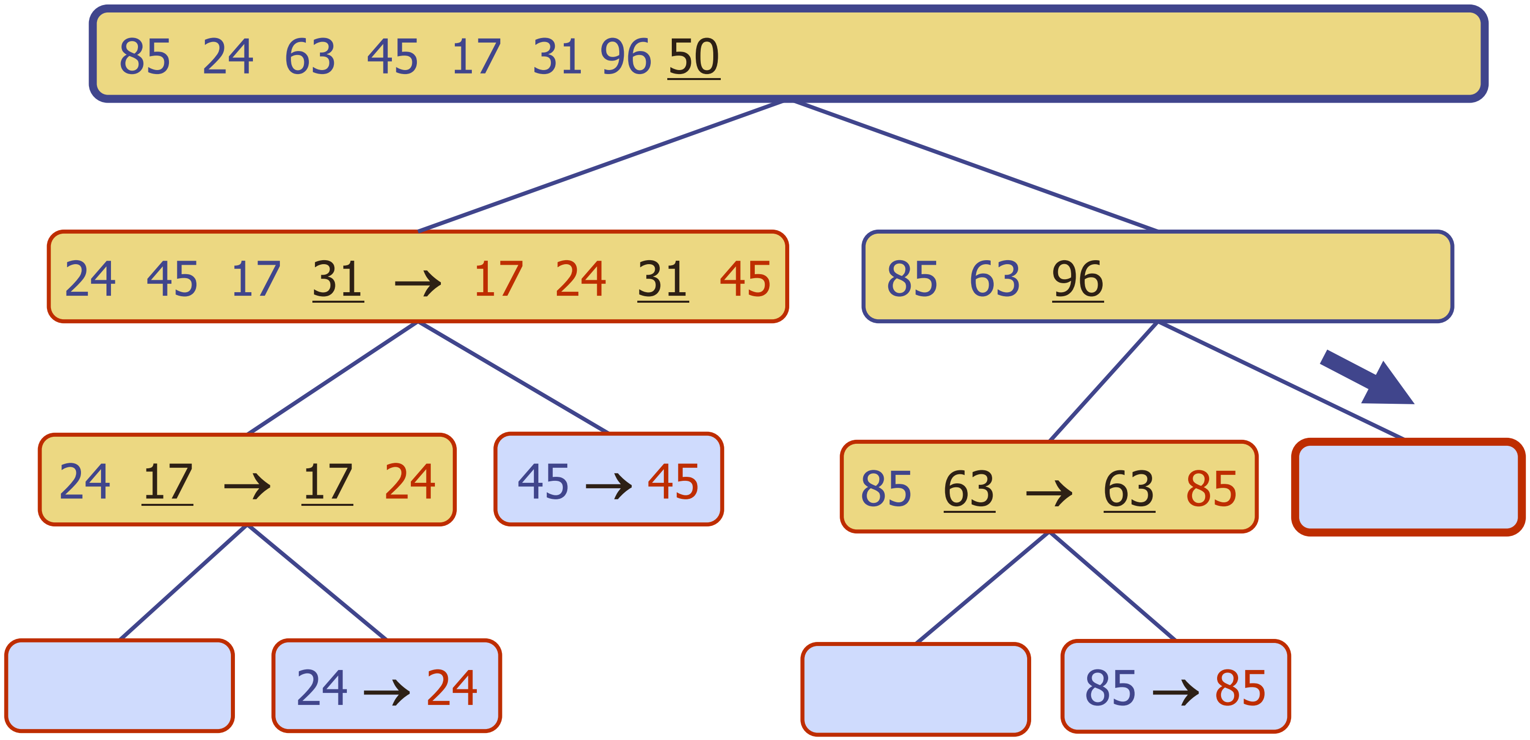
# Execution Example (cont'd)



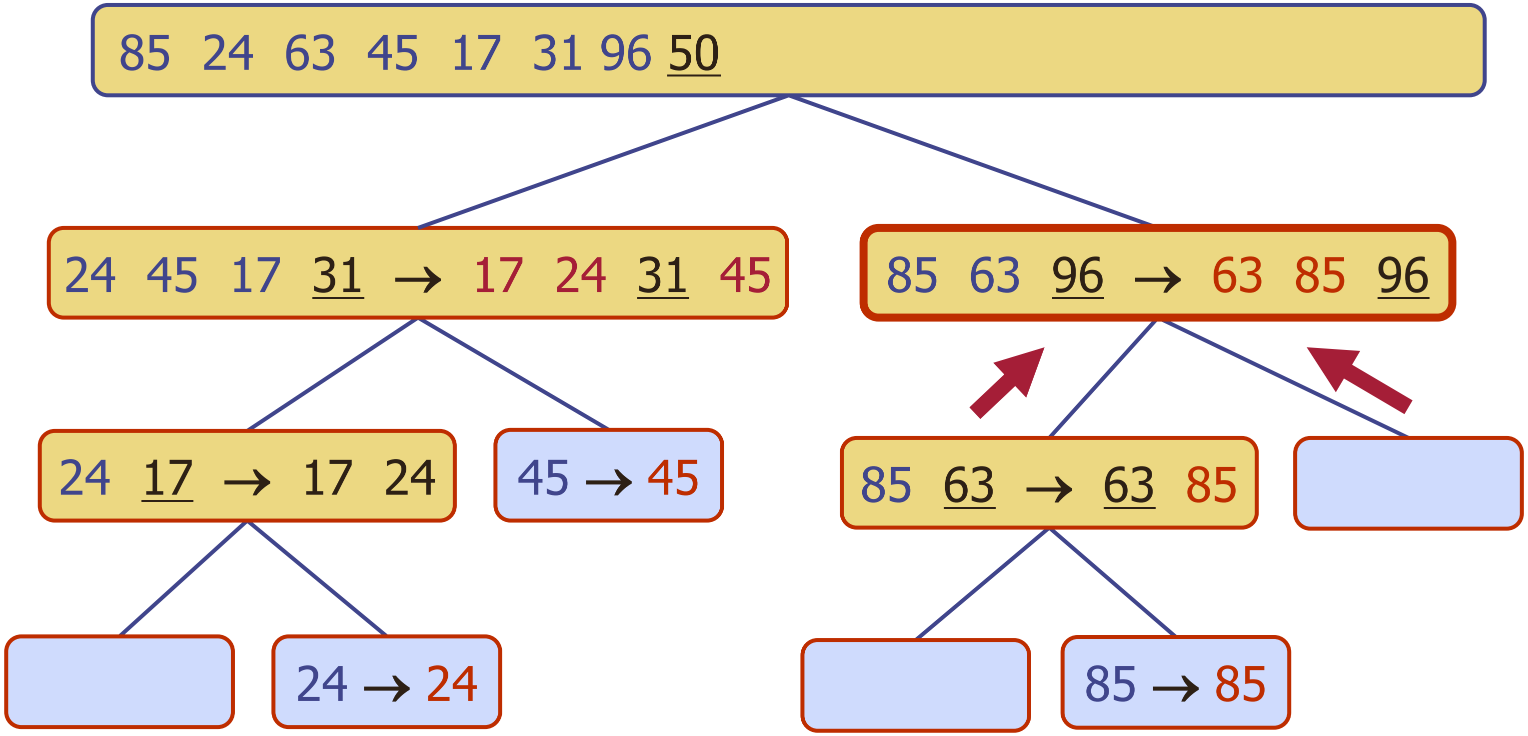
# Execution Example (cont'd)



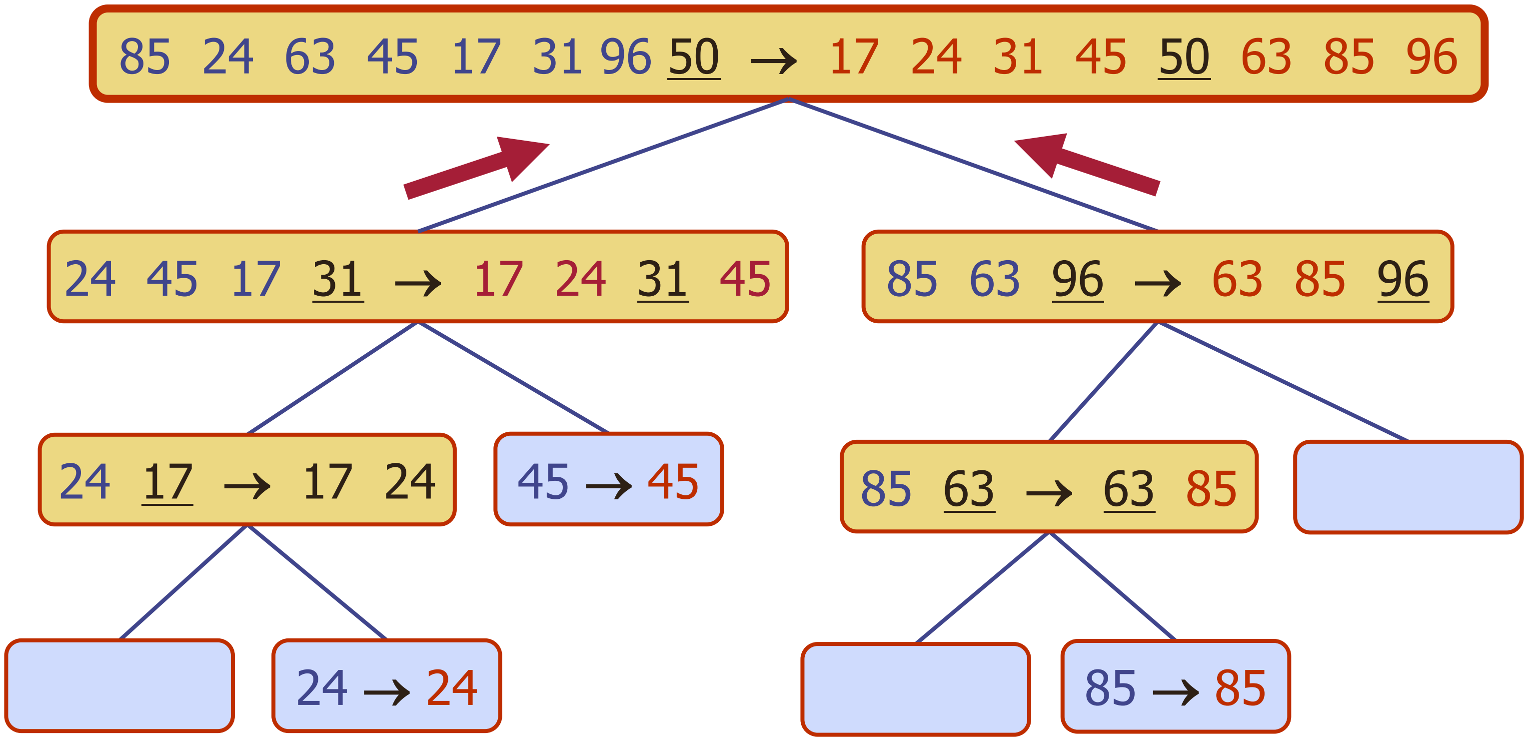
# Execution Example (cont'd)



# Execution Example (cont'd)



# Execution Example (cont'd)



# In-place Quick-Sort

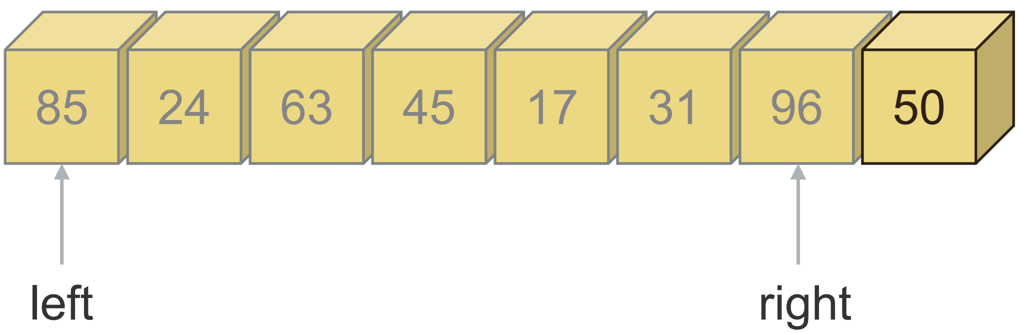
- An algorithm is in-place if it uses only a small amount of memory in addition to that needed for the original input
- If we use additional containers $L, E$ and $G$, as described before, then quick-sort is not in place
- But the algorithm can be modified to run in-place, by using the input sequence to store the subsequences for all the recursive calls
- The input sequence is modified during quick sort using element swapping, without ever explicitly creating subsequences
- The subsequence is implicitly specified by a leftmost index, $a$, and a rightmost index, $b$

# In-place Quick-Sort (cont'd)

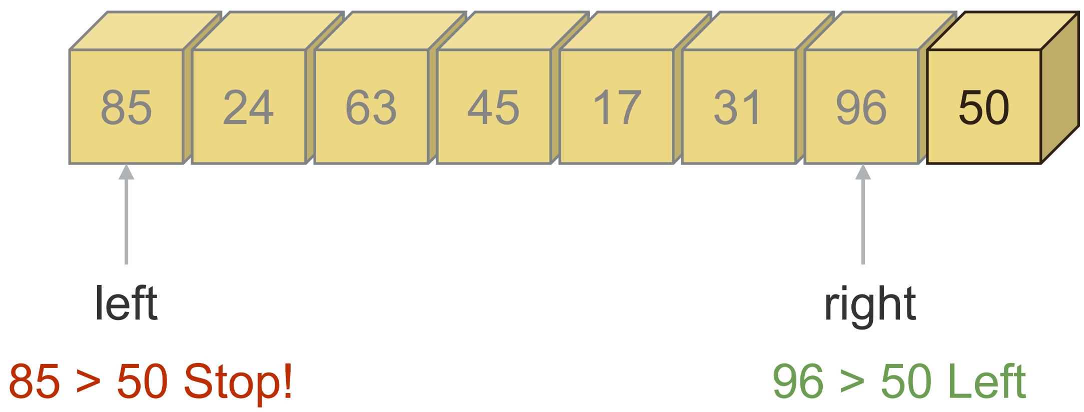
```
def inplace_quick_sort(S, a, b):
  """Sort the list from S[a] to S[b] inclusive using the quick-sort algorithm."""
  if a >= b: return                                  # range is trivially sorted
  pivot = S[b]                                       # last element of range is pivot
  left = a                                           # will scan rightward
  right = b-1                                        # will scan leftward
  while left <= right:
    # scan until reaching value equal or larger than pivot (or right marker)
    while left <= right and S[left] < pivot:
      left += 1
    # scan until reaching value equal or smaller than pivot (or left marker)
    while left <= right and pivot < S[right]:
      right -= 1
    if left <= right:                                # scans did not strictly cross
      S[left], S[right] = S[right], S[left]                    # swap values
      left, right = left + 1, right - 1                        # shrink range

  # put pivot into its final place (currently marked by left index)
  S[left], S[b] = S[b], S[left]
  # make recursive calls
  inplace_quick_sort(S, a, left - 1)
  inplace_quick_sort(S, left + 1, b)
```

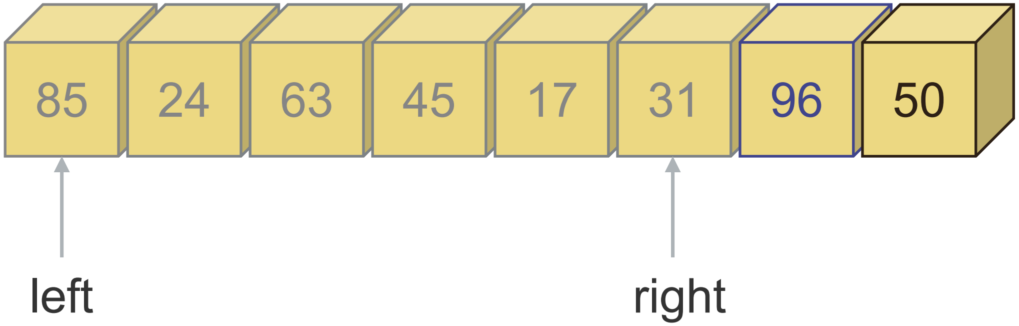
# In-place Quick-sort - Example



# In-place Quick-sort – Example (cont'd)



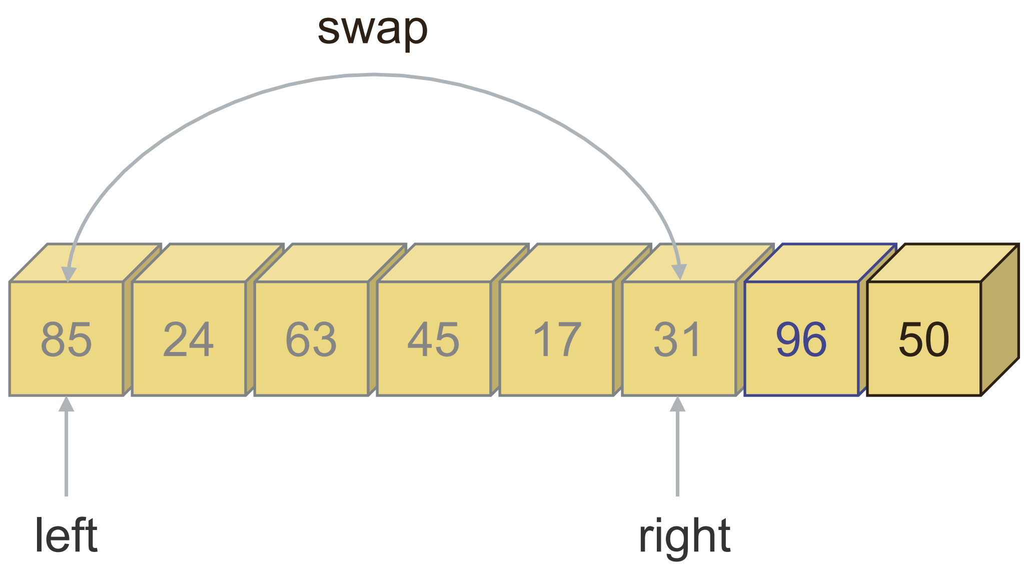
# In-place Quick-sort – Example (cont'd)




85 > 50 Stop!

31 < 50 Stop!

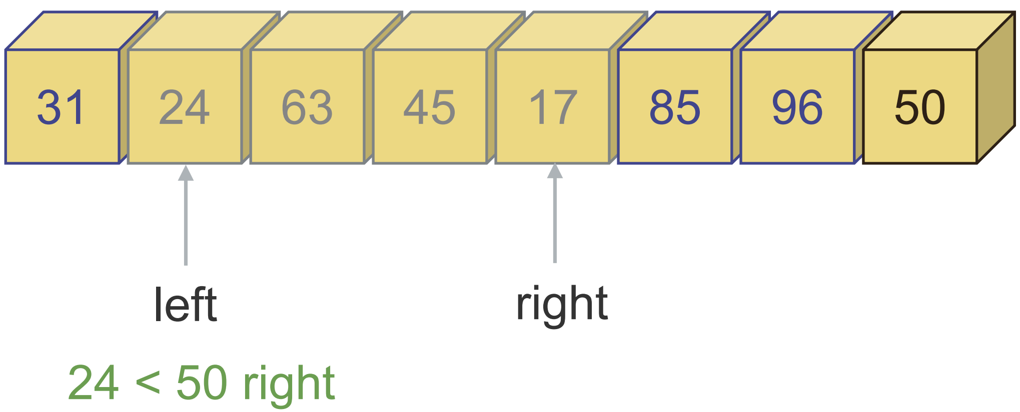
# In-place Quick-sort – Example (cont'd)



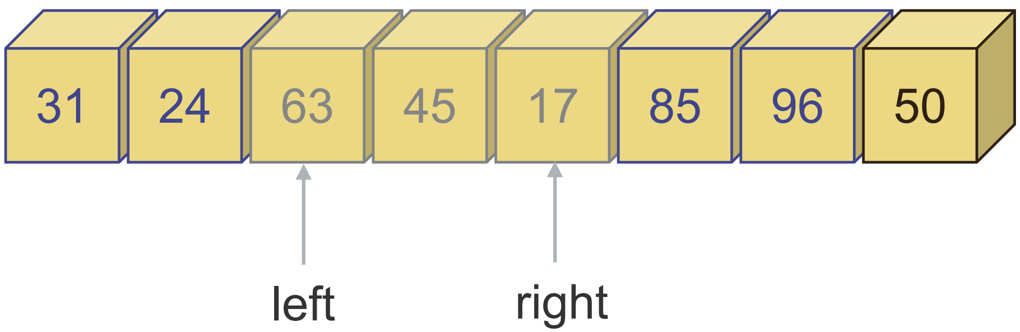

85 > 50 Stop!

31 < 50 Stop!

# In-place Quick-sort – Example (cont'd)

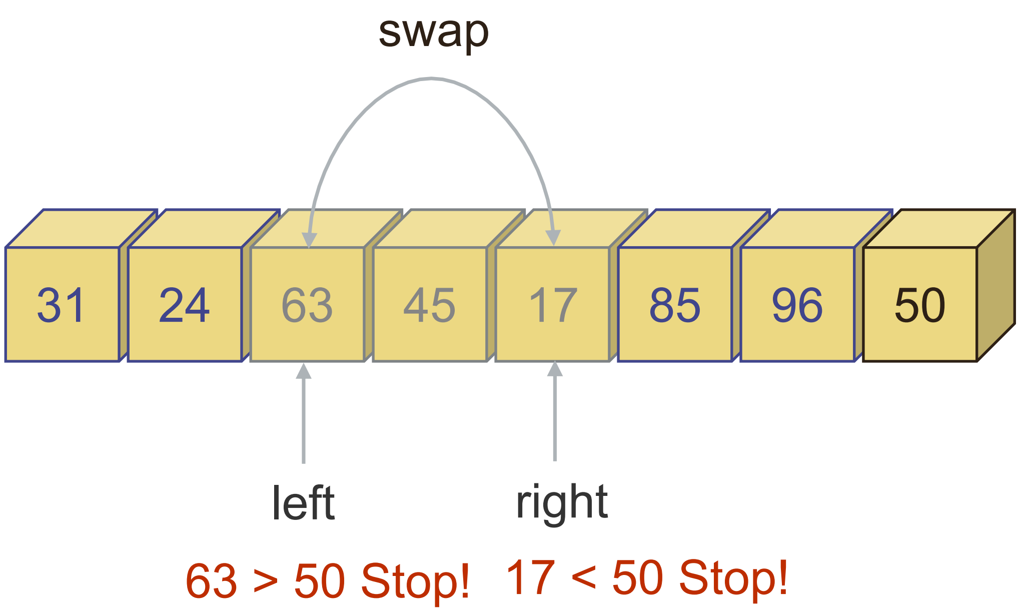


# In-place Quick-sort – Example (cont'd)

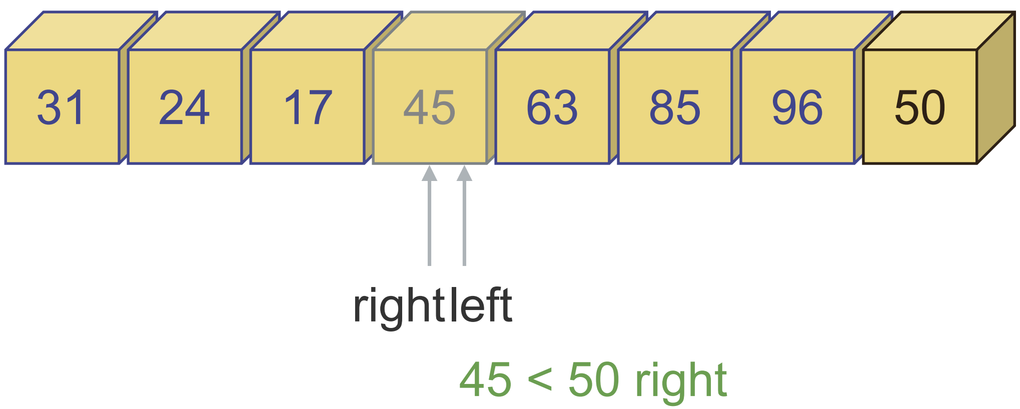



63 > 50 Stop!  17 < 50 Stop!

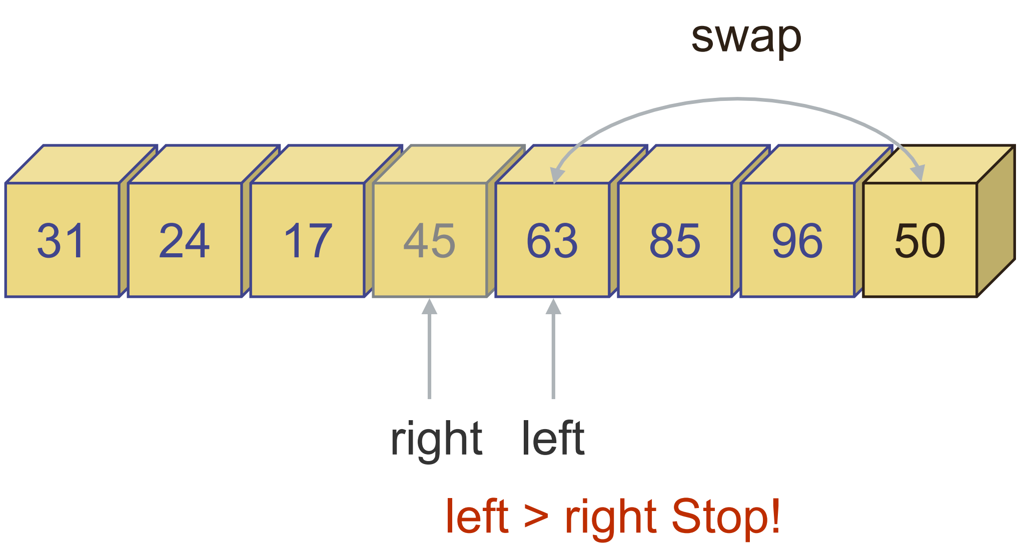
# In-place Quick-sort – Example (cont'd)



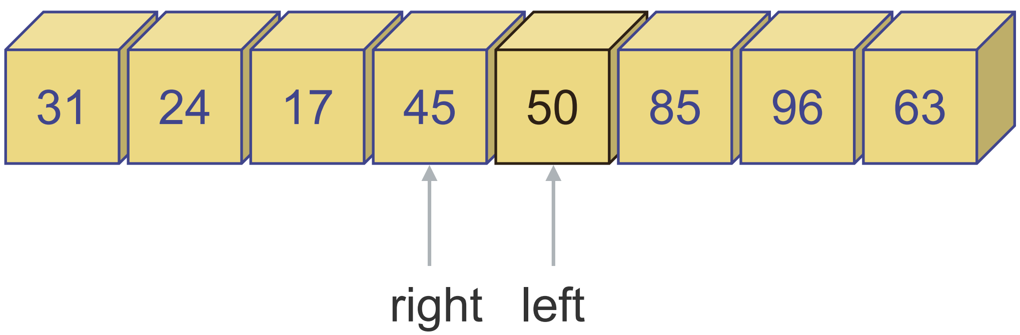
# In-place Quick-sort – Example (cont'd)



# In-place Quick-sort – Example (cont'd)

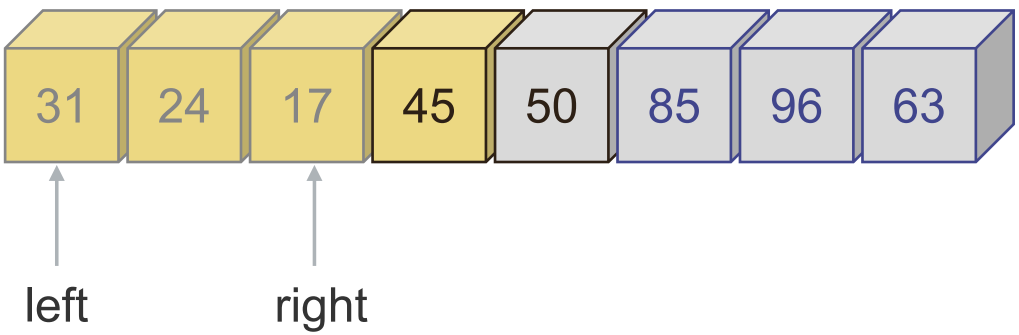


# In-place Quick-sort – Example (cont'd)

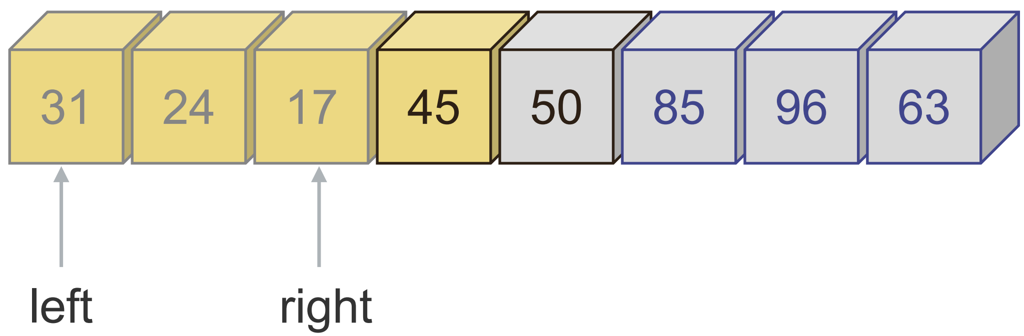



recur first on the left subsequence, up to the pivot

# In-place Quick-sort – Example (cont’d)

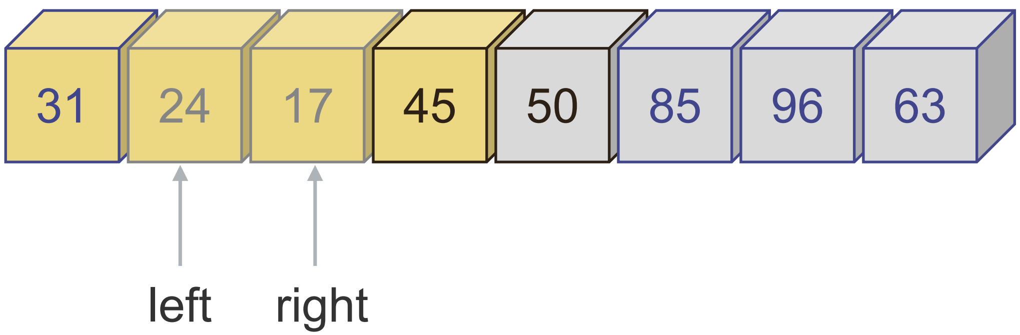


# In-place Quick-sort – Example (cont’d)



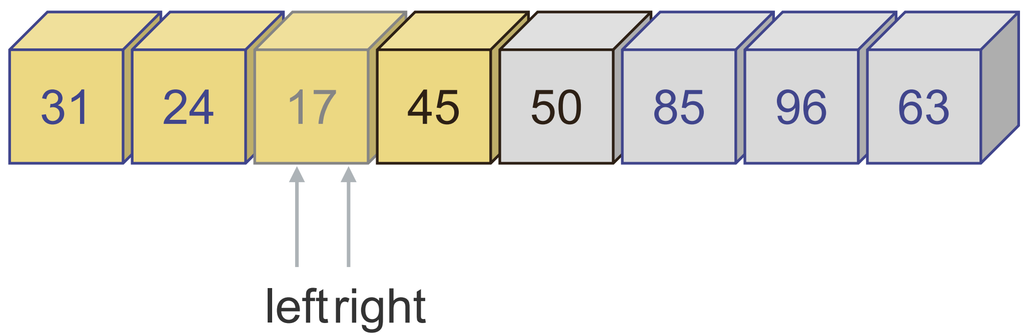

31 < 45 right

# In-place Quick-sort – Example (cont'd)



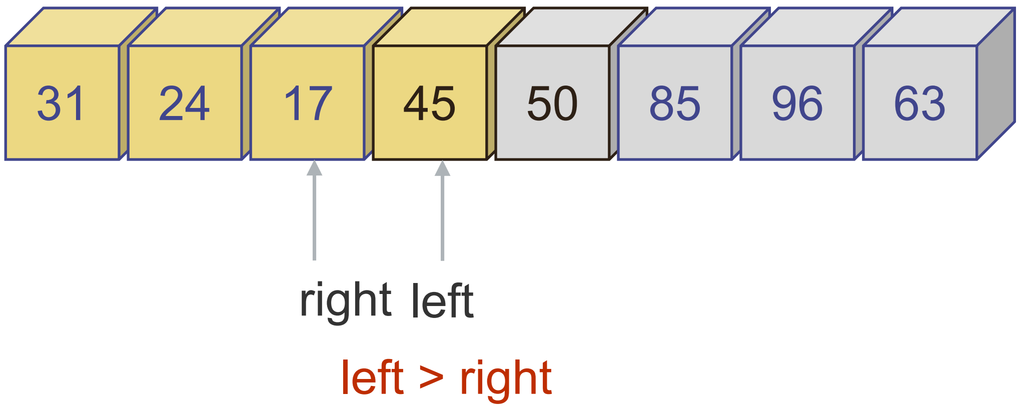

24 < 45 right

# In-place Quick-sort – Example (cont'd)



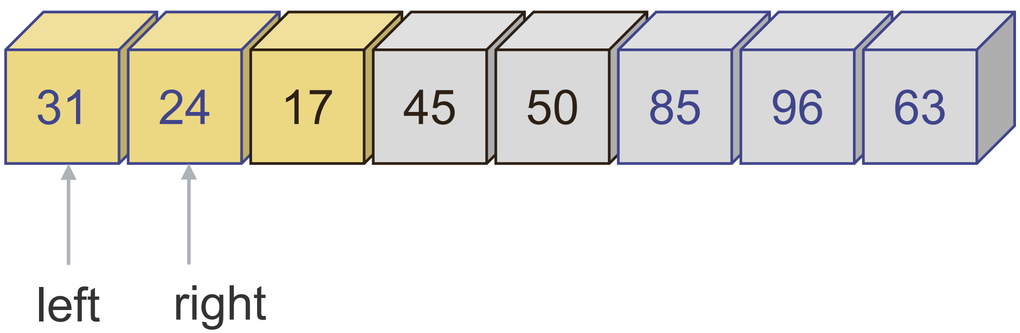

17 < 45 right

# In-place Quick-sort – Example (cont'd)

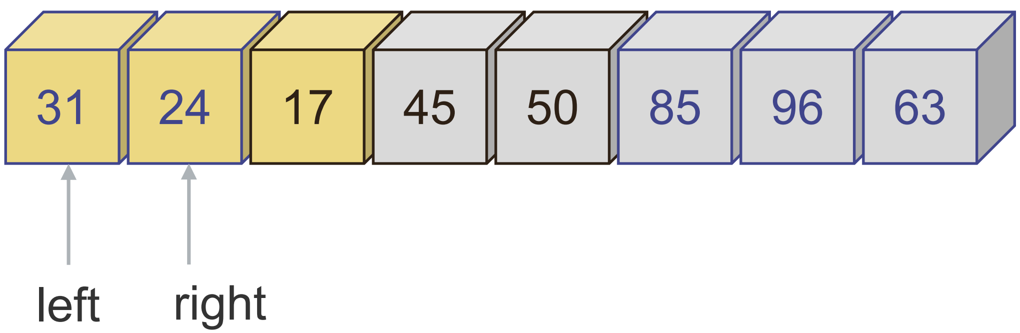



- pivot already in place
- recur first on the left subsequence, up to the pivot

# In-place Quick-sort – Example (cont’d)

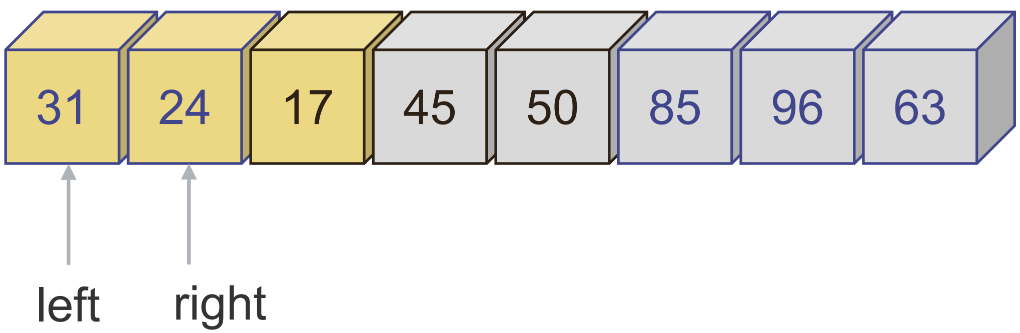


# In-place Quick-sort – Example (cont'd)



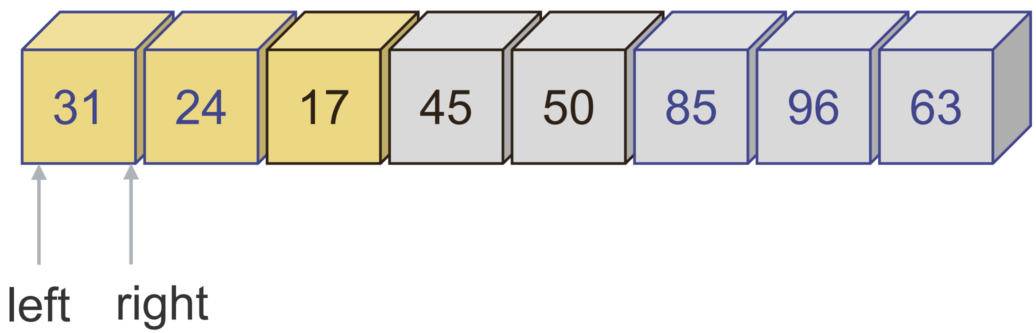

31 > 17 Stop!

# In-place Quick-sort – Example (cont'd)



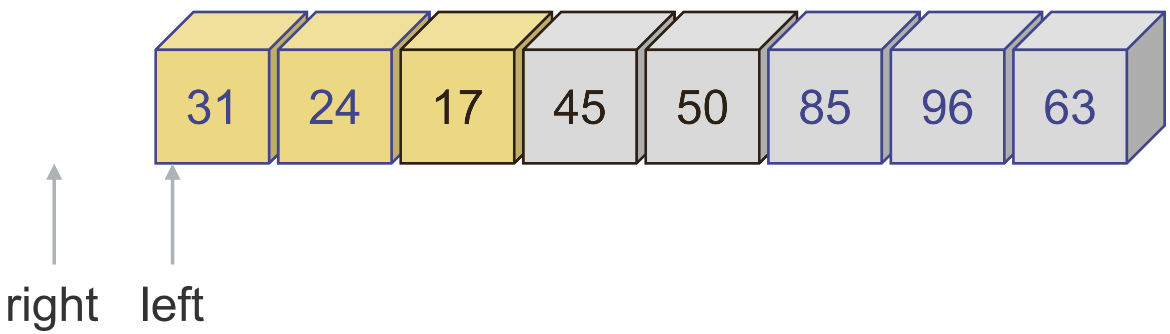

31 > 17 Stop!   24 > 17 left

# In-place Quick-sort – Example (cont'd)



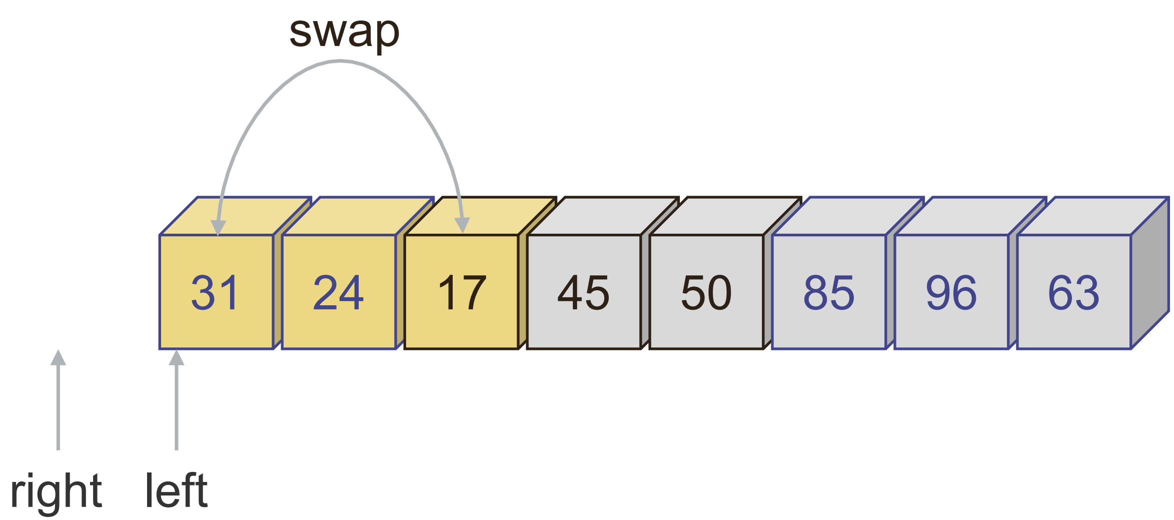

31 > 17 Stop!    31 > 17 left

# In-place Quick-sort – Example (cont'd)



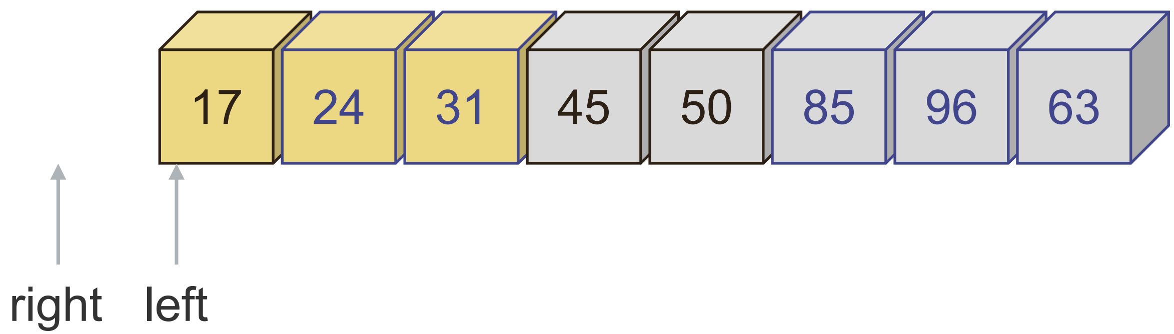

left > right Stop!   31 > 17 Stop!

# In-place Quick-sort – Example (cont'd)

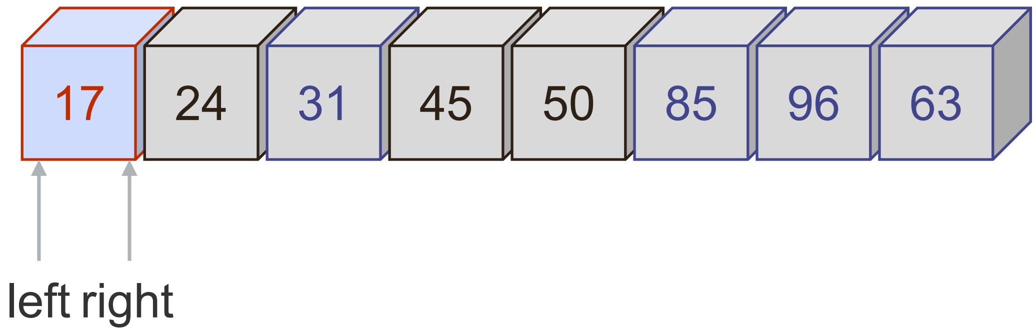



left > right Stop!   31 > 17 Stop!

# In-place Quick-sort – Example (cont’d)

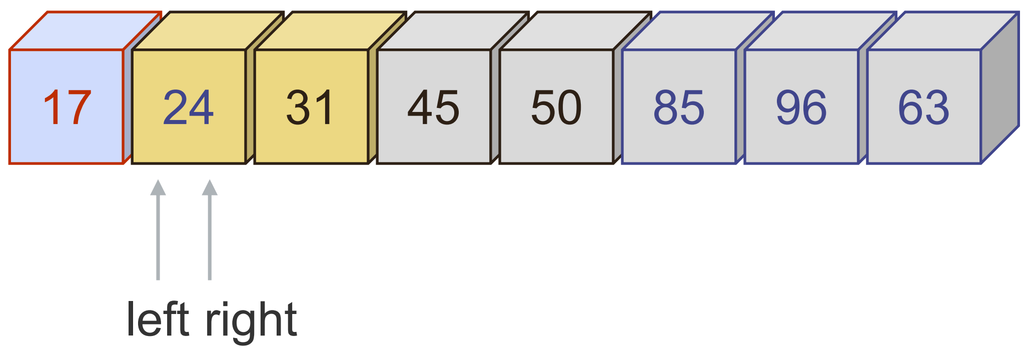


# In-place Quick-sort – Example (cont'd)



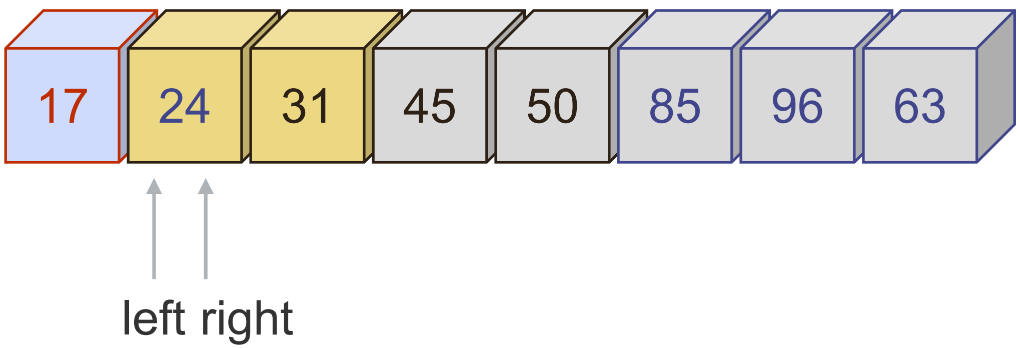

- pivot 17
- left subsequence empty
- recur on the right subsequence

# In-place Quick-sort – Example (cont'd)



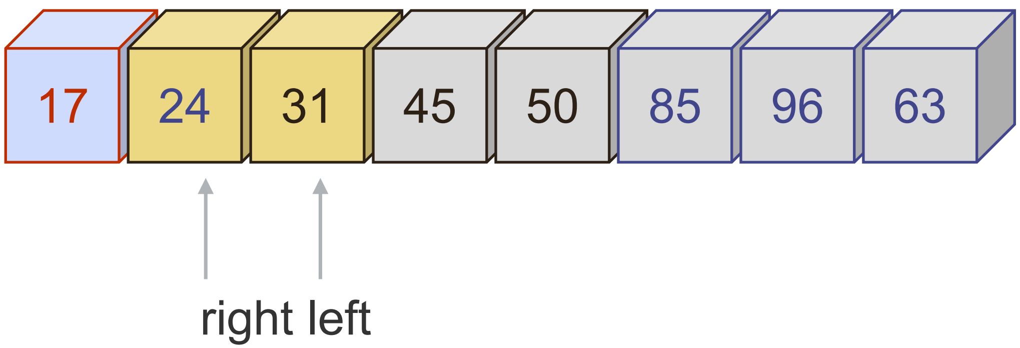

- recur on the right subsequence
- pivot 31

# In-place Quick-sort – Example (cont'd)




24 < 31 move left

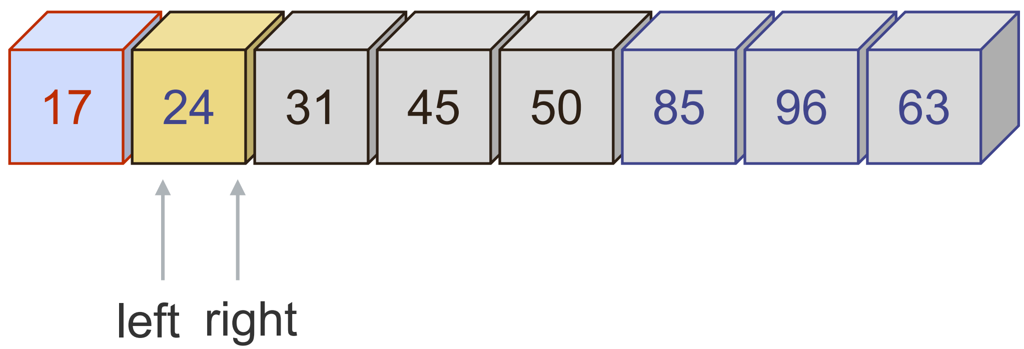
# In-place Quick-sort – Example (cont'd)




left > right Stop!

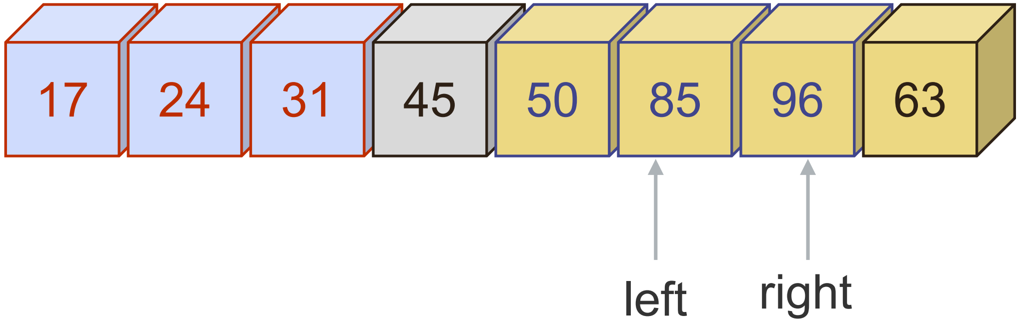
- pivot already in the correct position, no need to swap
- recur on left subsequence

# In-place Quick-sort – Example (cont'd)



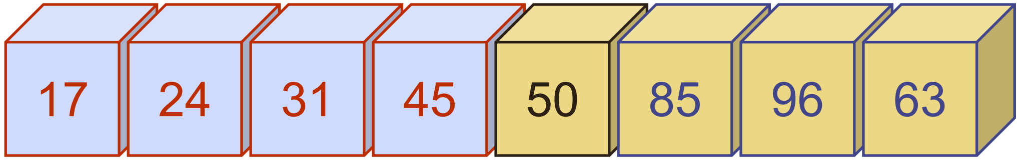

- one element (24), trivially sorted
- right subsequence empty, return

# In-place Quick-sort – Example (cont'd)



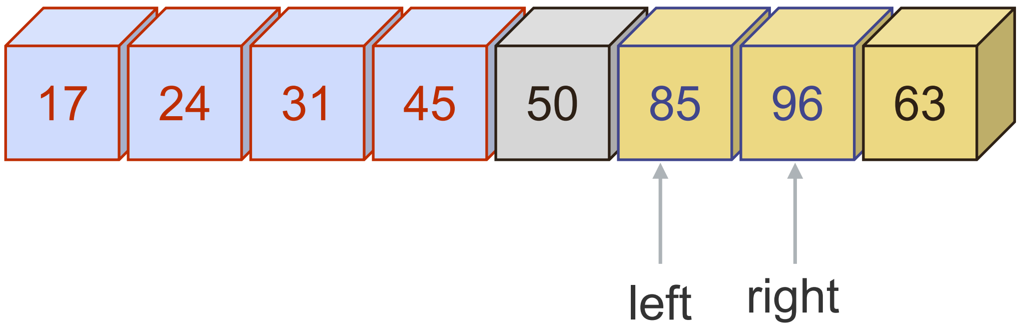

- finished recurring on the left, pivot is 45, right subsequence empty - return

# In-place Quick-sort – Example (cont'd)



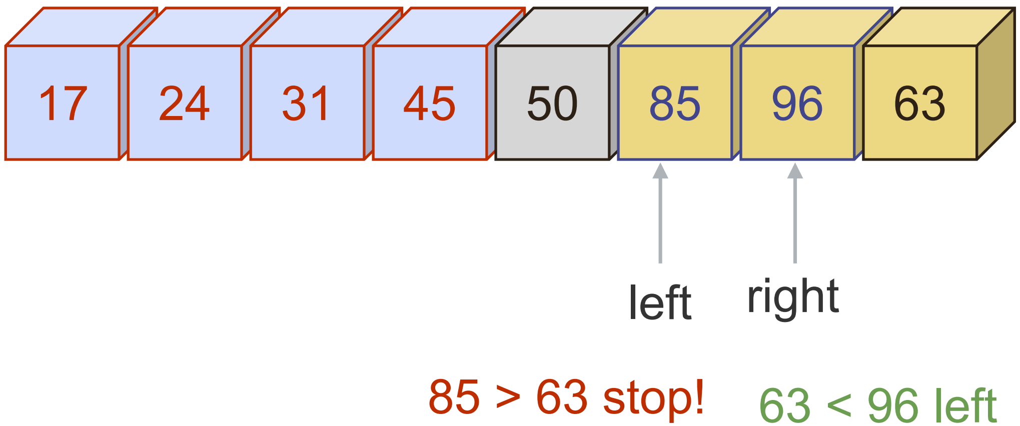

- return to top call with pivot 50
- recur on the right sequence

# In-place Quick-sort – Example (cont'd)

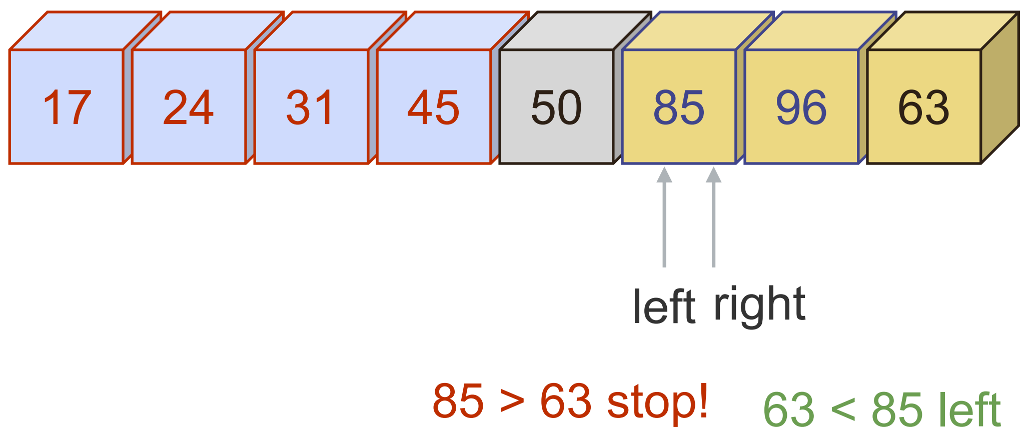



85 > 63 stop!

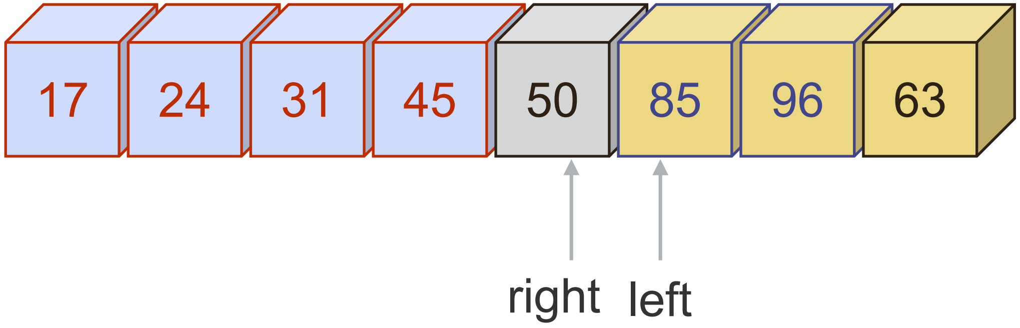
# In-place Quick-sort – Example (cont’d)



# In-place Quick-sort – Example (cont'd)

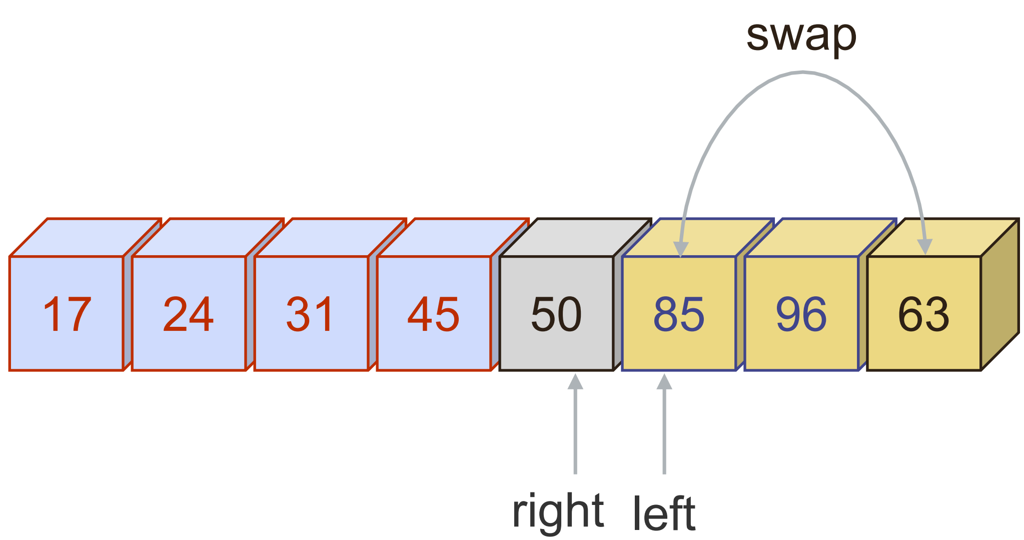


# In-place Quick-sort – Example (cont'd)



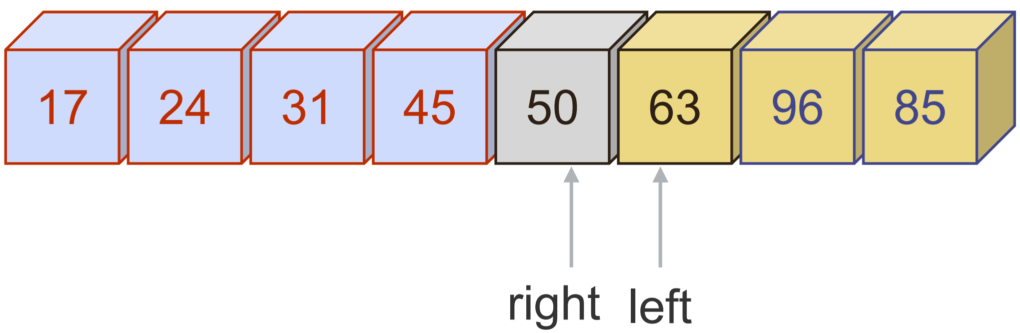

85 > 63 stop! left > right Stop!

# In-place Quick-sort – Example (cont'd)



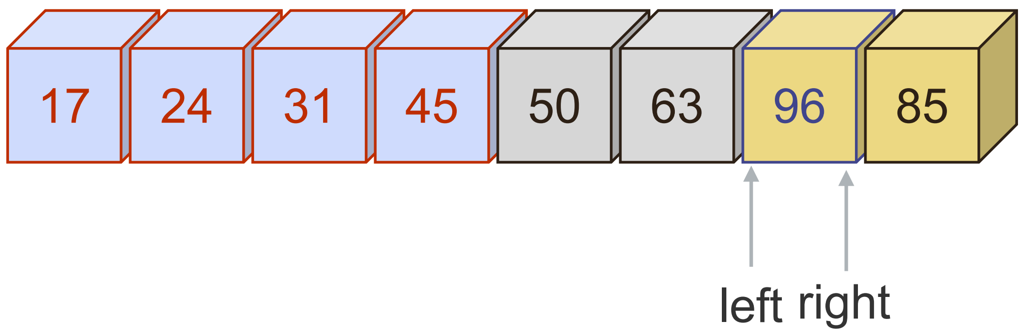

85 > 63 stop! left > right Stop!

# In-place Quick-sort – Example (cont'd)



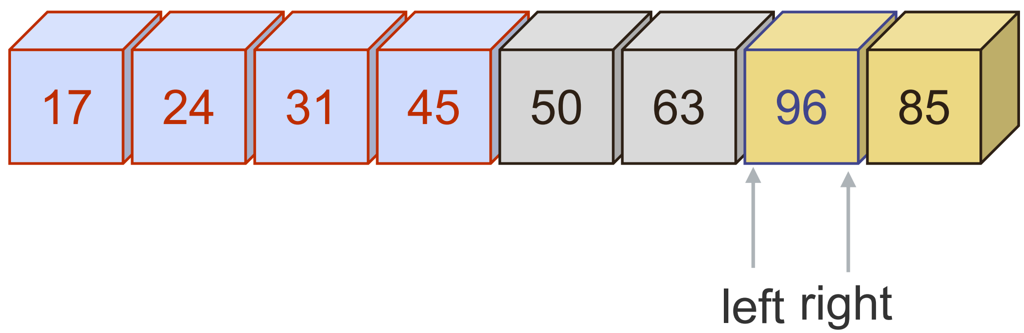

- left subsequence empty, recur on the right subsequence

# In-place Quick-sort – Example (cont’d)



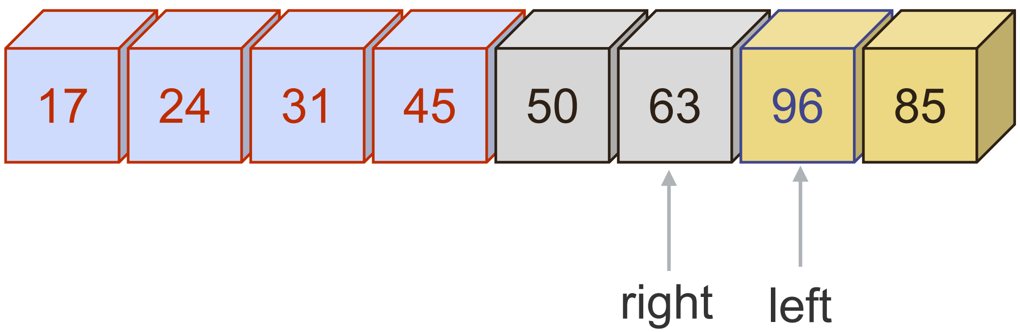

96 > 85 Stop!

# In-place Quick-sort – Example (cont'd)




96 > 85 Stop!   96 < 85 left

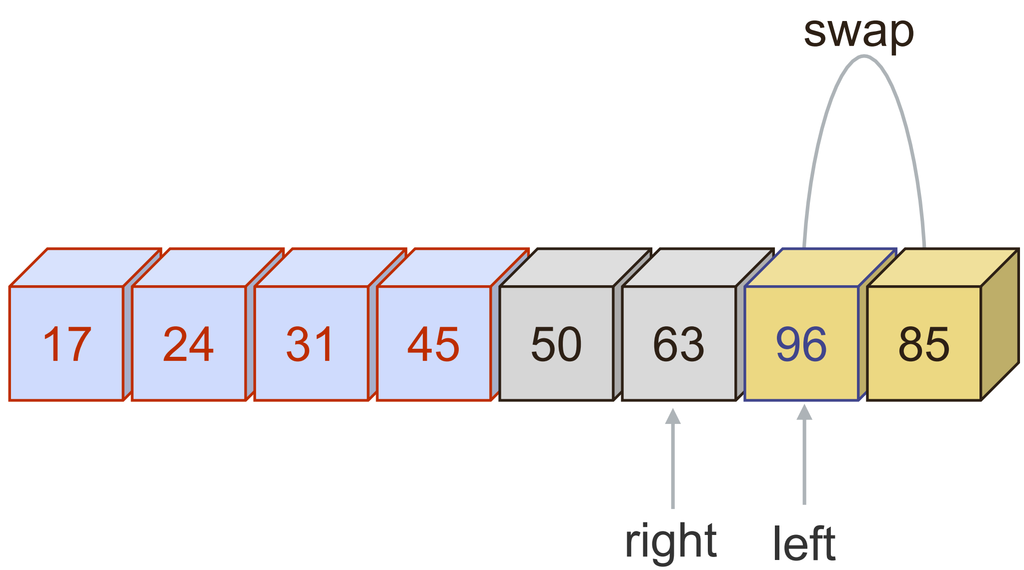
# In-place Quick-sort – Example (cont'd)




left > right Stop!

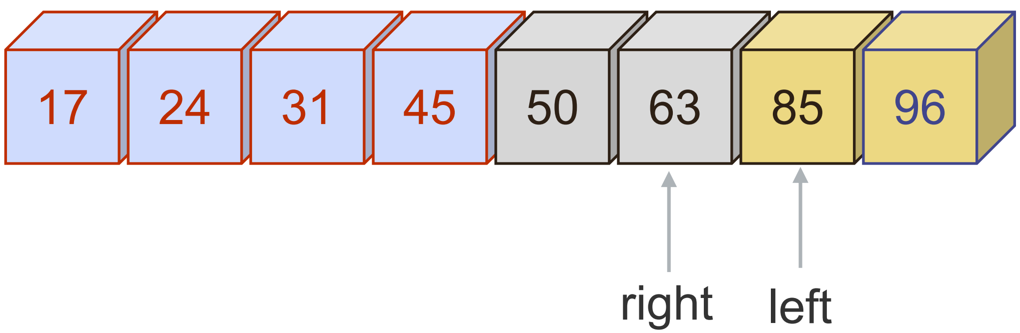
96 > 85 Stop!

# In-place Quick-sort – Example (cont'd)



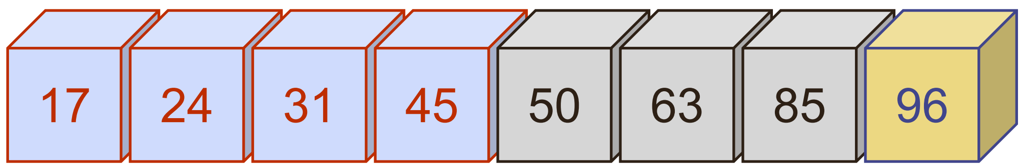

left > right Stop!    96 > 85 Stop!

# In-place Quick-sort – Example (cont'd)



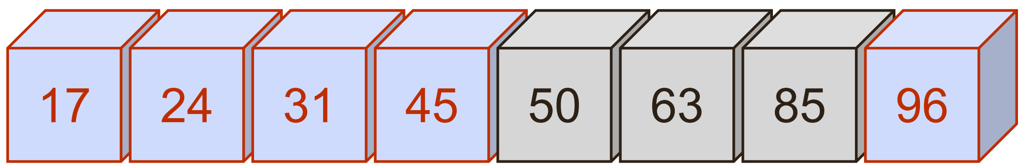

left > right Stop!    96 > 85 Stop!

# In-place Quick-sort – Example (cont'd)



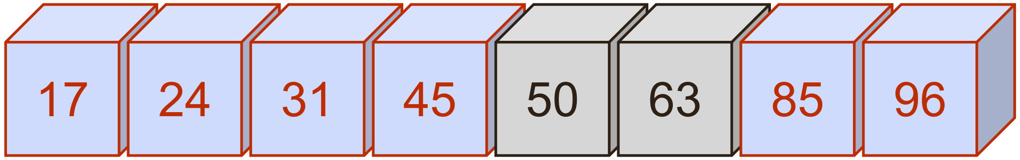

- pivot 85
- left subsequence empty
- recur on the right subsequence, trivial, one element

# In-place Quick-sort – Example (cont'd)



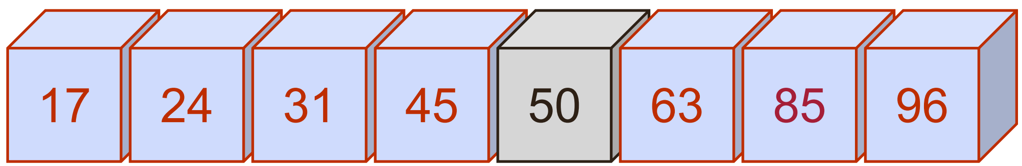

- pivot 85
- left subsequence empty
- recur on the right subsequence, trivial, one element

# In-place Quick-sort – Example (cont'd)



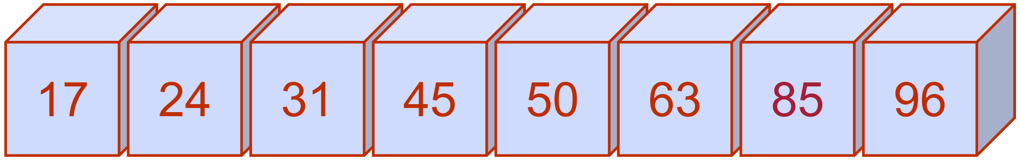

- return to pivot 63

# In-place Quick-sort – Example (cont'd)




- end recursion on the right subsequence

# In-place Quick-sort – Example (cont'd)




- end recursion on the full sequence

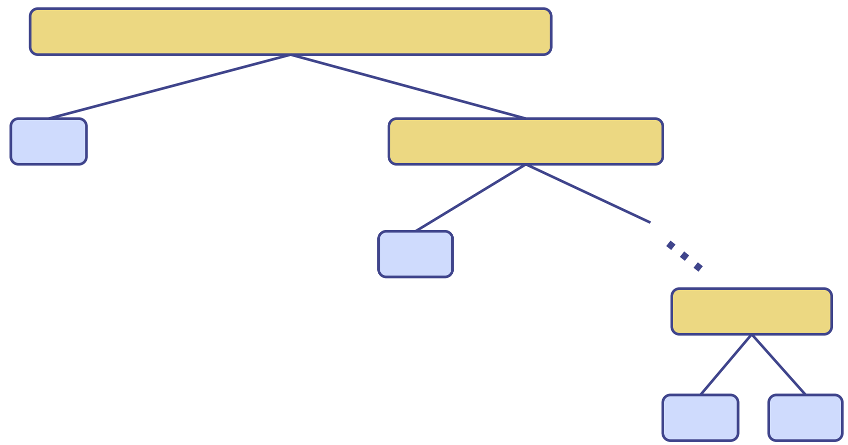
# Worst-case Running Time

- Worst-case behavior occurs when sorting should be easy – when the sequence is already sorted
- The worst case for quick-sort occurs when the pivot is the unique maximum or minimum element
- One of $L$ or $G$ has size $n - 1$, the other has size $0$
- The running time is proportional to the sum

$$n + (n-1) + \ldots + 2 + 1 = \frac{n(n+1)}{2}$$

# Worst-case Running Time (cont'd)

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



- Thus the worst-case running time of quick-sort is $O(n^2)$
- The overall running time is $O(n \cdot h)$, where $h$ is the overall height of the quicksort tree

# Running Time of Quick Sort

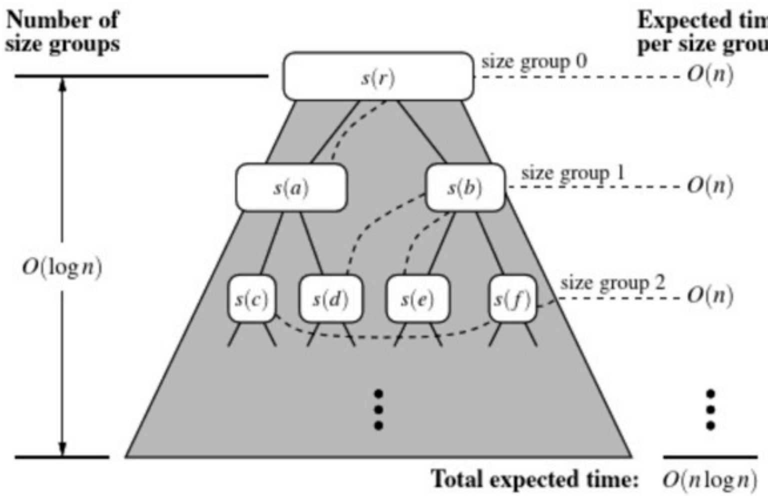
- $O(n^2)$ worst-case behavior – why is quick-sort so slow?
- Actually, it *is* quick, in practice:
  - The best case for quick-sort is when subsequences $L$ and $G$ have roughly the same size
  - By introducing randomization in the choice of the pivot, quick-sort can have an expected running time of $O(n \log n)$

# Randomized Quick-Sort

- Instead of picking the pivot as the last element of the sequence, use a random element as the pivot
- Some heuristics for pivot selection
  - Pick a random element of the array
  - Median-of-three: take the median of three values chosen from the front, middle or tail of the array
- In this case the height of the quick-sort tree is $\log n$, and the overall running time is $O(n \log n)$

# Randomized Quick-Sort – $O(n\ log\ n)$

- Why? If the pivot is well-chosen, it splits the input sequence in approx. half
- E.g. 128 elements in the sequence:
  - Partition 1: 64 elements
  - Partition 2: 32 elements
  - Partition 3: 16 elements
  - Partition 4: 8 elements
  - Partition 5: 4 elements
  - Partition 6: 2 elements
  - Partition 7: 1 element
- 7 levels of partitioning: $\log_2 128 = 7$
- Each of the 7 ($\log n$) partitioning levels visits all 128 ($n$) inputs



# Summary

| Algorithm | Time | Notes |
|---|---|---|
| insertion sort | $O(n^2)$ | • in-place<br>• slow, but good for small inputs |
| quick sort | $O(n \log n)$ | • in-place, randomized<br>• fastest, good for large inputs |

Thank you.