

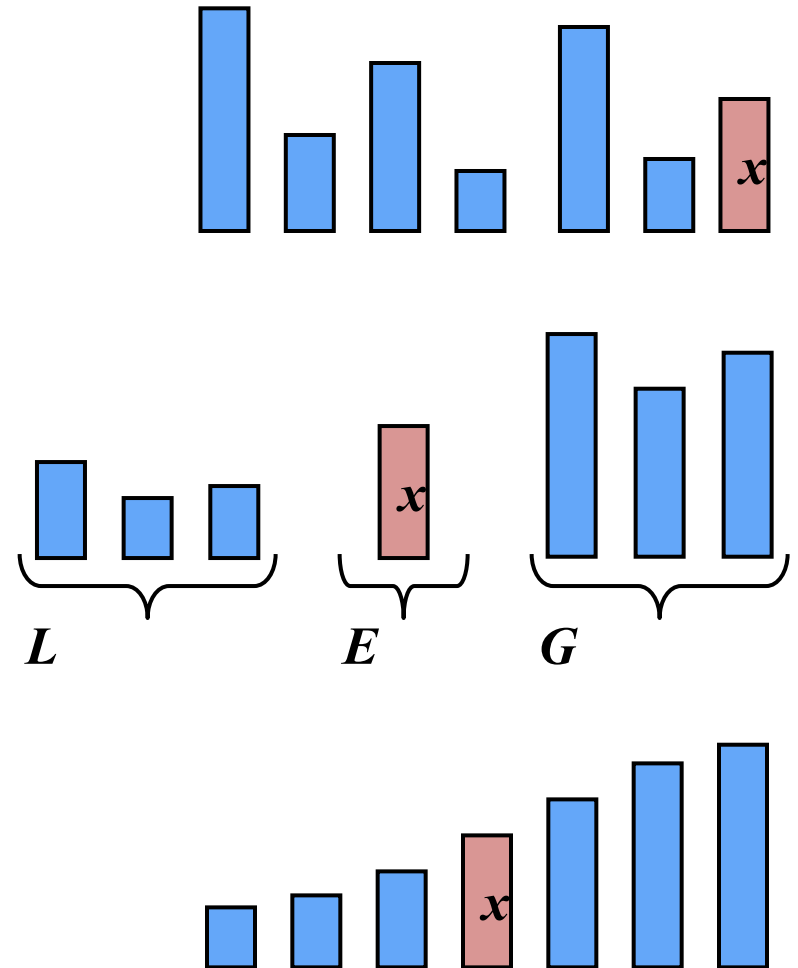
# Quick Sort

# Quick Sort

A sorting algorithm based on the divide-and-conquer paradigm

- **Divide**: pick a **pivot** element  $x$  and partition  $S$  into
  - $L$  elements less than  $x$
  - $E$  elements equal to  $x$
  - $G$  elements greater than  $x$
- **Recur**: sort  $L$  and  $G$
- **Conquer**: join  $L$ ,  $E$  and  $G$

The choice of the pivot affects the algorithm's performance.



# Partition

1. Remove each element  $y$  from  $S$
  2. Insert  $y$  into  $L$ ,  $E$  or  $G$ , depending on the result of the comparison with the pivot  $x$
- Each insert/remove takes  $O(1)$  time.
  - Thus, the partition step of quick-sort takes  $O(n)$  time.



**Algorithm** *partition*( $S, x$ )

**Input** sequence  $S$ , pivot element  $x$

**Output** subsequences  $L, E, G$

$L, E, G \leftarrow$  empty sequences

**while**  $\neg S.isEmpty()$

$y \leftarrow S.remove(S.first())$

**if**  $y < x$

$L.insertLast(y)$

**else if**  $y = x$

$E.insertLast(y)$

**else**  $\{ y > x \}$

$G.insertLast(y)$

**return**  $L, E, G$

The choice of the pivot affects the performance of Quick Sort.

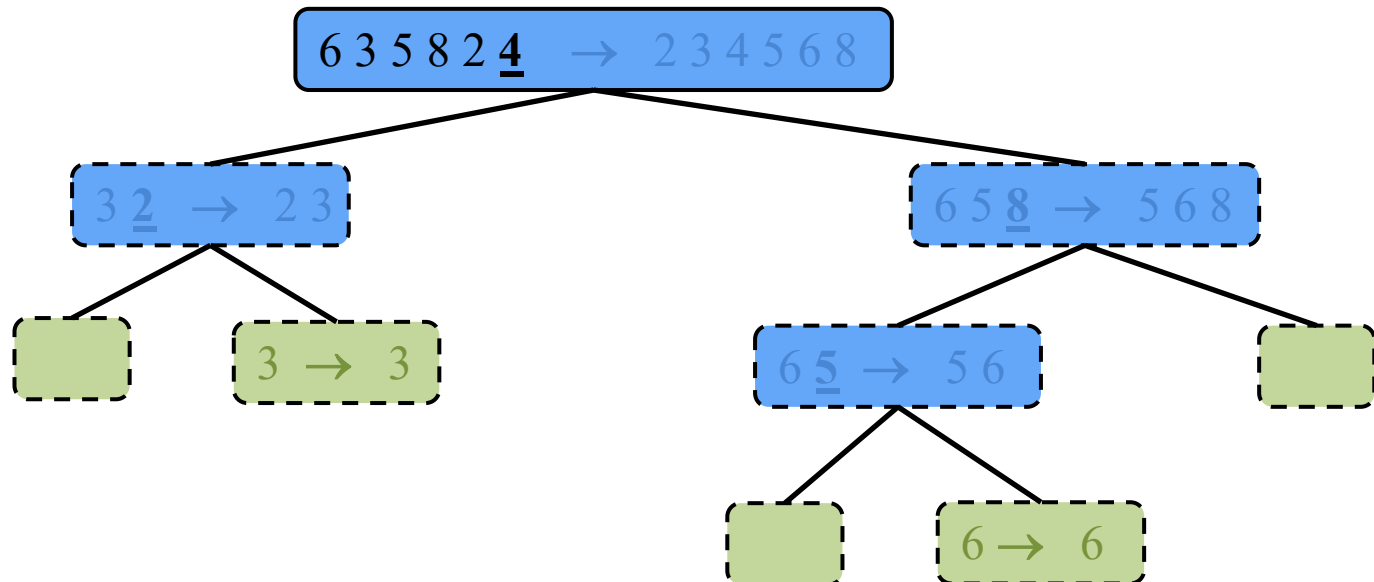
# Quick-Sort Tree

An execution of quick-sort 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

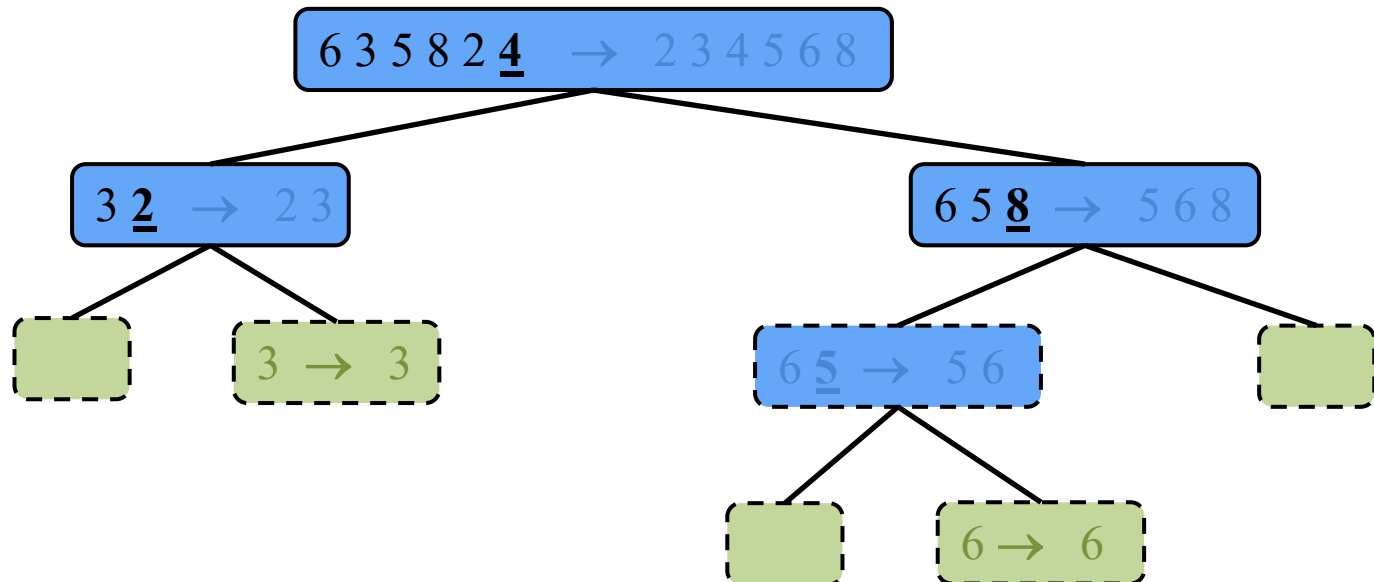
# Quick Sort Execution

- Strategy: Select the last element as the pivot



# Quick Sort Execution

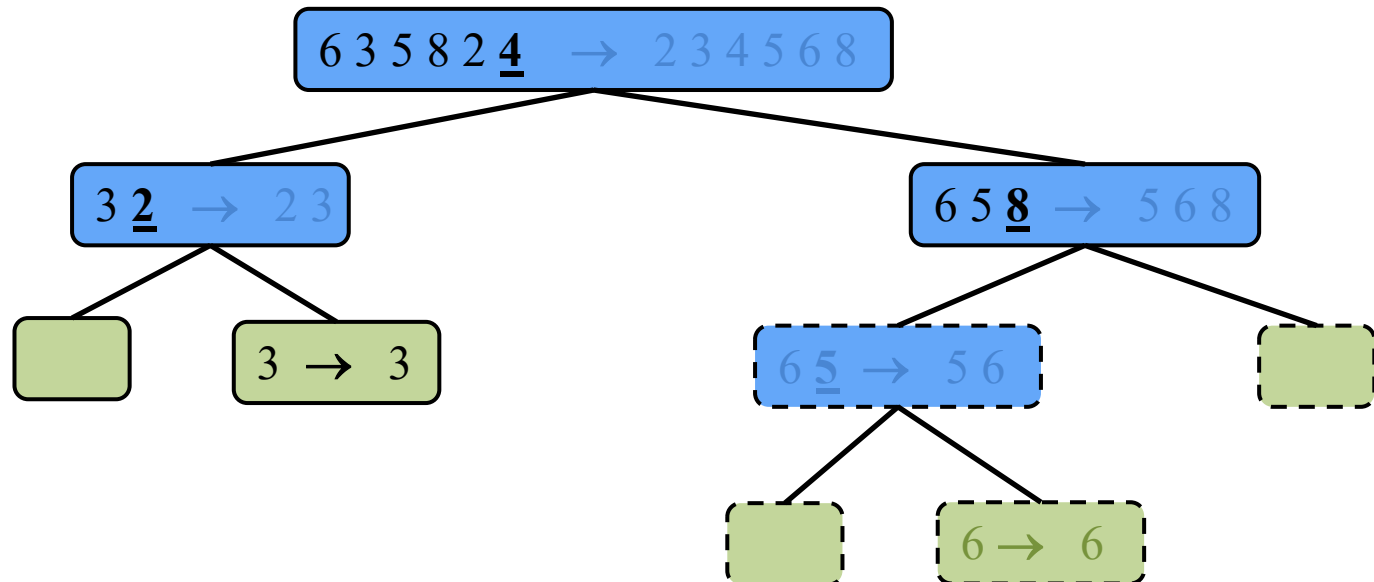
- Strategy: Select the last element as the pivot



- Select pivot, partition, recursive call

# Quick Sort Execution

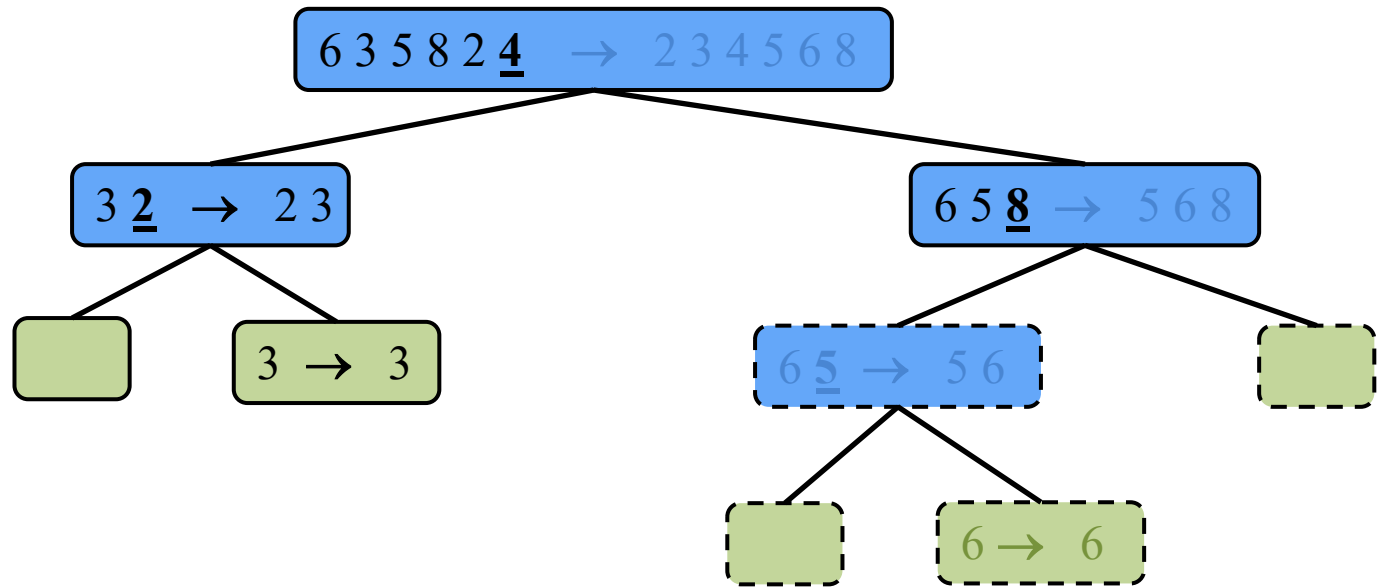
- Strategy: Select the last element as the pivot



- Select pivot, partition, recursive call

# Quick Sort Execution

- Strategy: Select the last element as the pivot

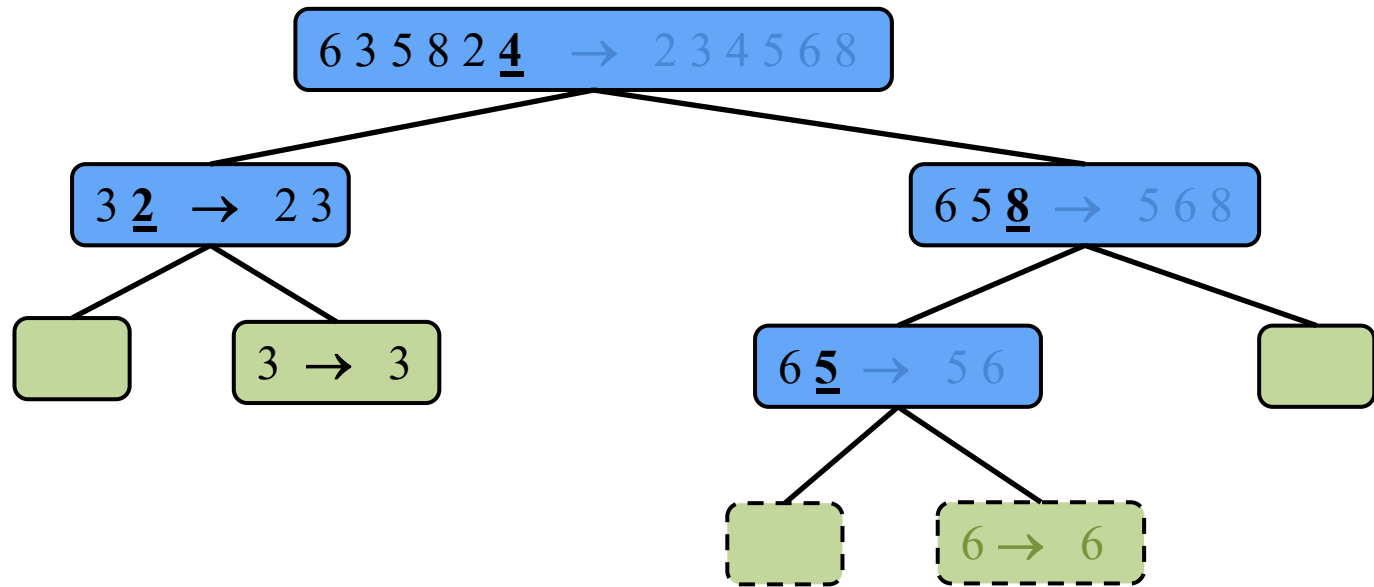


- Join



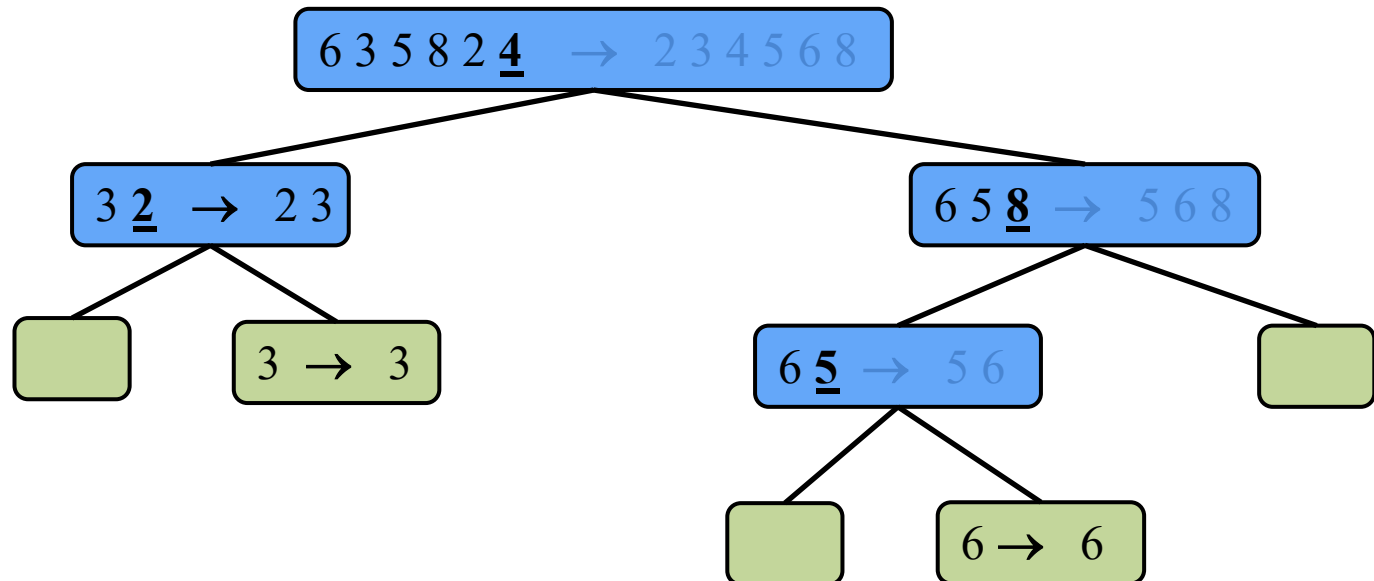
# Quick Sort Execution

- Strategy: Select the last element as the pivot



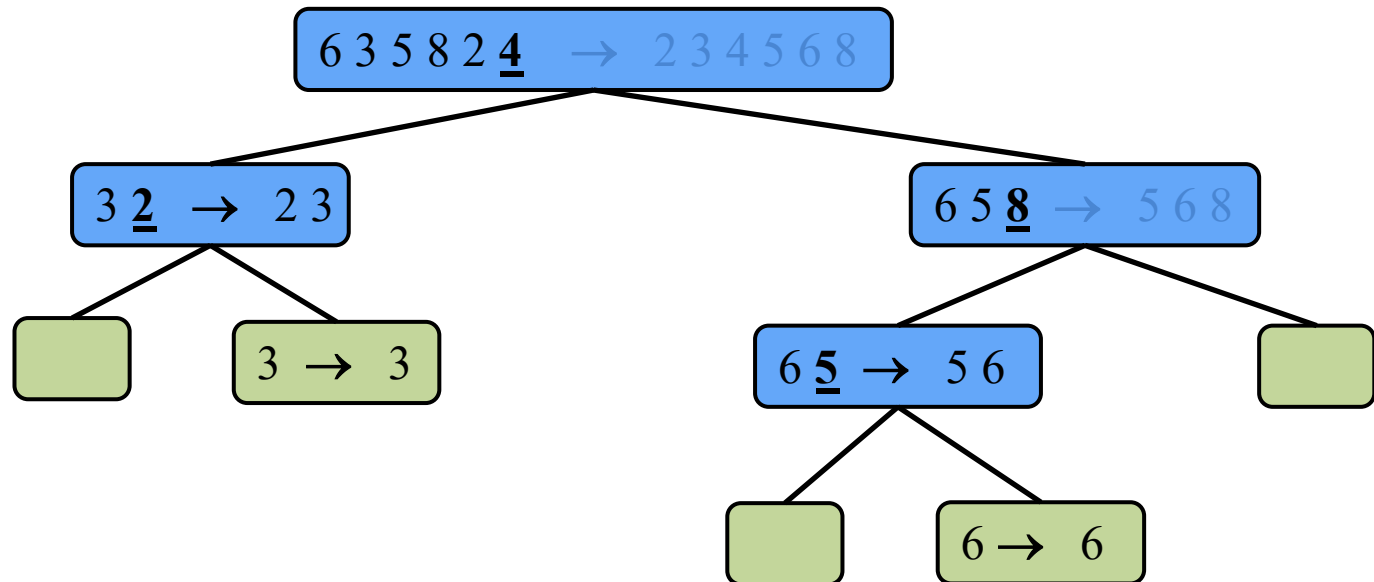
# Quick Sort Execution

- Strategy: Select the last element as the pivot



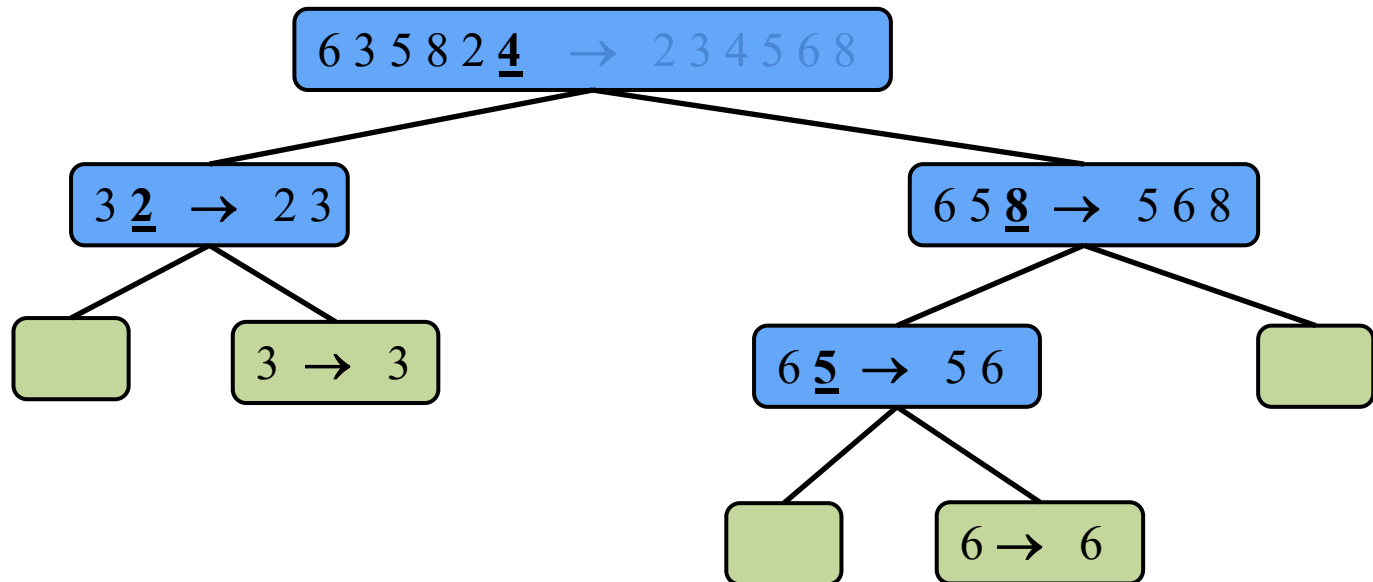
# Quick Sort Execution

- Strategy: Select the last element as the pivot



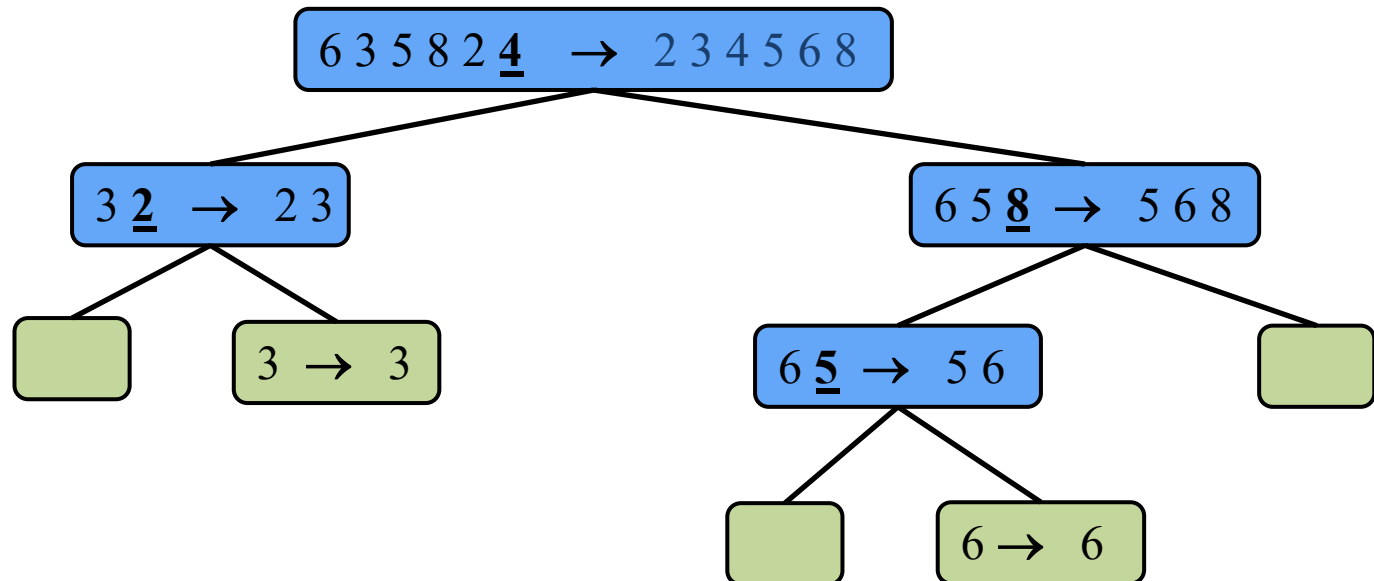
# Quick Sort Execution

- Strategy: Select the last element as the pivot



# Quick Sort Execution

- Strategy: Select the last element as the pivot

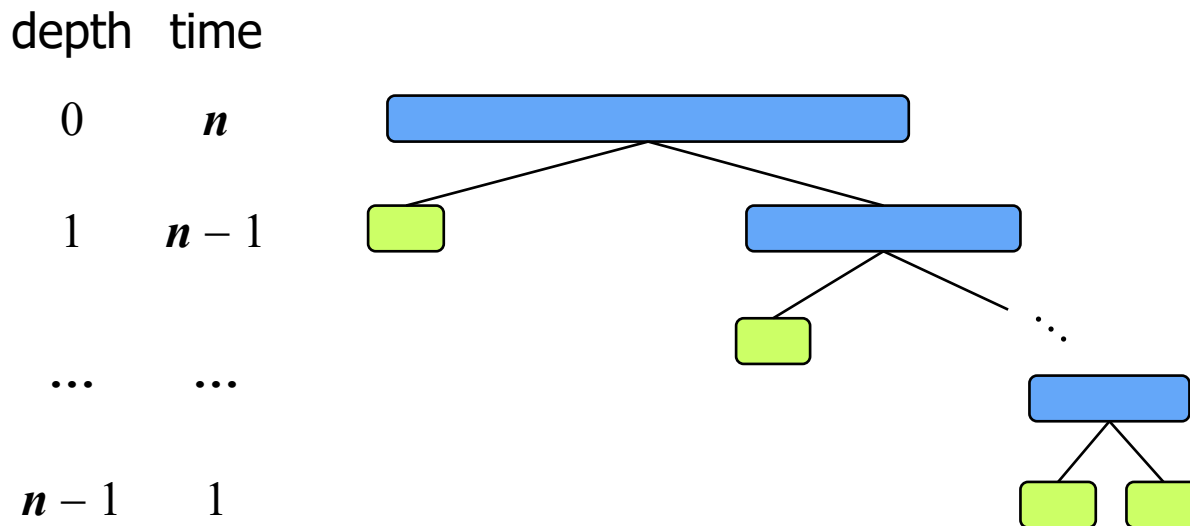


# Worst-case Running Time

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:  $n + (n - 1) + \dots + 2 + 1$
- If we use the strategy of selecting the **last element** as the pivot, this happens when the list is already sorted!

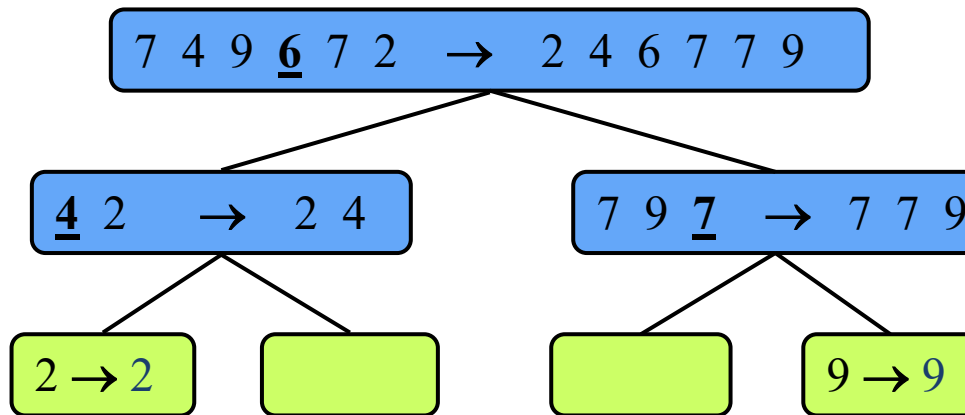
Thus, the worst-case running time of quick-sort is  $O(n^2)$



# Randomized Quick Sort

Pivot selection strategy: choose a **random** element as the pivot

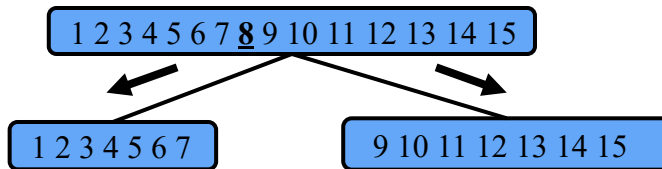
- Still has worst-case running time  $O(n^2)$ 
  - Due to random selection, this case is highly unlikely
- Expected running time is  $O(n \log n)$



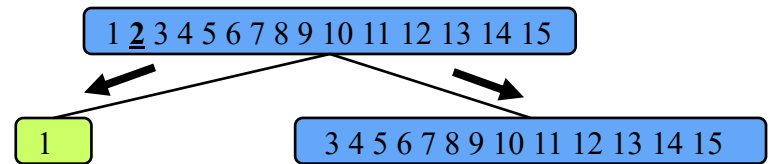
# Expected Running Time

Consider a recursive call of quick-sort on a sequence of size  $s$

- **Good call:** the sizes of  $L$  and  $G$  are each less than  $3s/4$
- **Bad call:** one of  $L$  and  $G$  has size greater than  $3s/4$



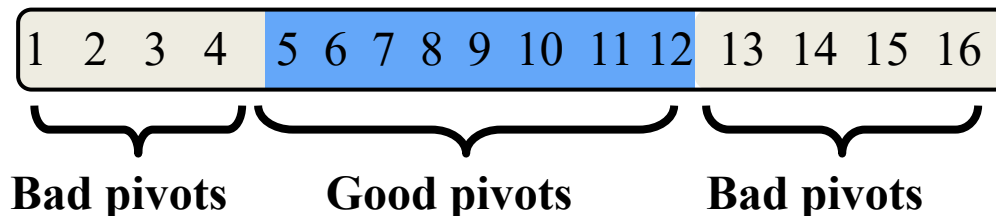
**Good call**



**Bad call**

A call is **good** with probability  $1/2$

- $1/2$  of the possible pivots cause good calls:





# Expected Running Time (continued)

**Probabilistic Fact:** The expected number of coin tosses required in order to get  $k$  heads is  $2k$ .

For a node of depth  $i$ , we expect

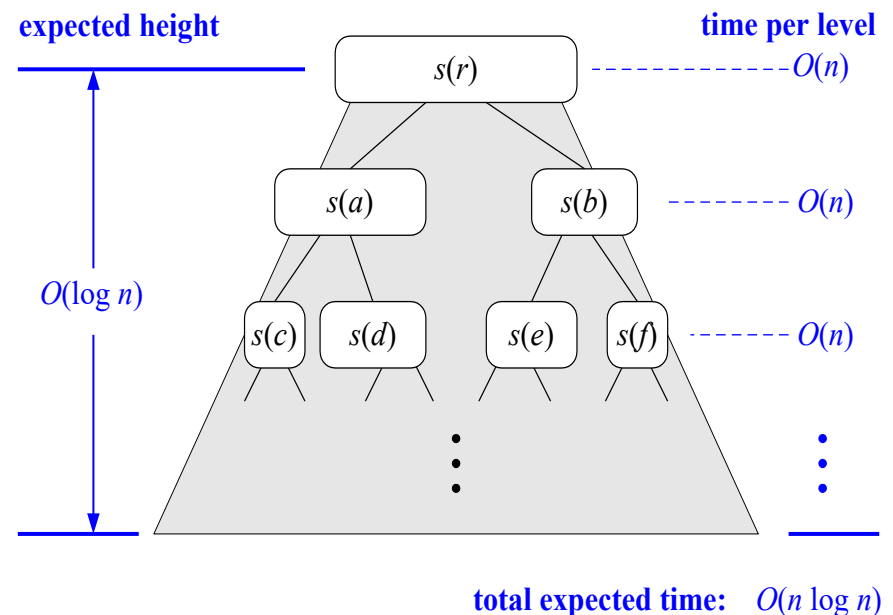
- $i/2$  ancestors are good calls
- size of the input sequence for the current call is at most  $(3/4)^{i/2}n$

For a node of depth  $2\log_{4/3}n$

- the expected input size is one
- the expected height of the quick-sort tree is  $O(\log n)$

The amount of work done at the nodes of the same depth is  $O(n)$

Thus, the **expected running time of quick-sort is  $O(n \log n)$**



# In-Place Quick-Sort

During the partition step, use replace operations to rearrange elements of the input sequences such that:

- elements less than pivot have rank  $< h$
- elements equal to pivot have rank between  $[h, k]$
- elements greater than pivot have rank  $> k$

**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  $r$

$x \leftarrow S.\text{elemAtRank}(i)$

$(h, k) \leftarrow \text{inPlacePartition}(x)$

*inPlaceQuickSort*( $S, l, h - 1$ )

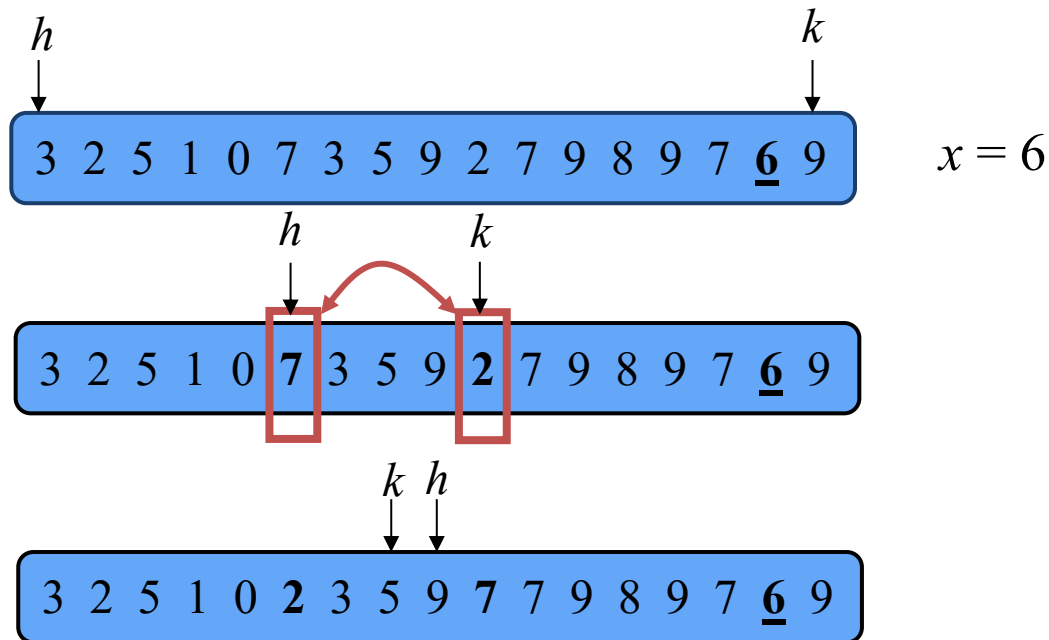
*inPlaceQuickSort*( $S, k + 1, r$ )

# In-Place Partition

Performs a partitioning using two indices to split  $S$  into  $L$  and  $E \cup G$  (a similar method can split  $E \cup G$  into  $E$  and  $G$ ).

Repeat until  $h$  and  $k$  cross:

- Scan  $h$  to the right until it finds an element  $\geq x$
- Scan  $k$  to the left until it finds an element  $< x$
- Swap elements at indices  $h$  and  $k$



# Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	<ul style="list-style-type: none"><li>◆ in-place, not stable</li><li>◆ slow (good for small inputs)</li></ul>
insertion-sort	$O(n^2)$	<ul style="list-style-type: none"><li>◆ in-place, stable</li><li>◆ slow (good for small inputs)</li></ul>
quick-sort	$O(n \log n)$ expected	<ul style="list-style-type: none"><li>◆ in-place, not stable</li><li>◆ randomized</li><li>◆ fast (good for large inputs)</li></ul>
heap-sort	$O(n \log n)$	<ul style="list-style-type: none"><li>◆ in-place, not stable</li><li>◆ fast (good for large inputs)</li></ul>
merge-sort	$O(n \log n)$	<ul style="list-style-type: none"><li>◆ not in-place, stable</li><li>◆ sequential data access</li><li>◆ fast (good for huge inputs)</li></ul>

# Other: Nuts and Bolts



You are given a collection of  $n$  bolts of different widths, and  $n$  corresponding nuts.

- You can test whether a given nut and bolt fit together, from which you learn whether the nut is too large, too small, or an exact match for the bolt.
- The differences in size between pairs of nuts or bolts are too small to see by eye, so you cannot compare the sizes of two nuts or two bolts directly.
- You are to match each bolt to each nut.

Give an efficient algorithm to solve the nuts and bolts problem.