Merge Sort

Divide-and-Conquer

Divide-and-conquer is a general algorithm design paradigm:

- Divide: divide the input data $\boldsymbol{S}$ in two disjoint subsets $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
- Recur: solve the subproblems associated with $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - the base case for the recursion are subproblems of size 0 or 1
- Conquer: combine the solutions for $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a solution for $\boldsymbol{S}$

Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

- Like heap-sort
  - Uses a comparator
  - Has $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$ running time
- Unlike heap-sort
  - Does not use an auxiliary priority queue
  - Accesses data in a sequential manner (suitable to sort data on a disk)

Merge Sort

Merge-sort on an input sequence $\boldsymbol{S}$ with $\boldsymbol{n}$ elements consists of three steps:

- Divide: partition $\boldsymbol{S}$ into two sequences $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ of about $\boldsymbol{n}/2$ elements each
- Recur: recursively sort $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
- Conquer: merge $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a unique sorted sequence

Algorithm ***mergeSort***(***S***, ***C***)

  Input sequence $\boldsymbol{S}$ with $\boldsymbol{n}$ elements, comparator $\boldsymbol{C}$

  Output sequence $\boldsymbol{S}$ sorted according to $\boldsymbol{C}$

  if ***S.size***() > 1

    $(\boldsymbol{S}_1, \boldsymbol{S}_2) \leftarrow \boldsymbol{partition}(\boldsymbol{S}, \boldsymbol{n}/2)$

    $\boldsymbol{mergeSort}(\boldsymbol{S}_1, \boldsymbol{C})$

    $\boldsymbol{mergeSort}(\boldsymbol{S}_2, \boldsymbol{C})$

    $\boldsymbol{S} \leftarrow \boldsymbol{merge}(\boldsymbol{S}_1, \boldsymbol{S}_2)$

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, takes $O(n)$ time

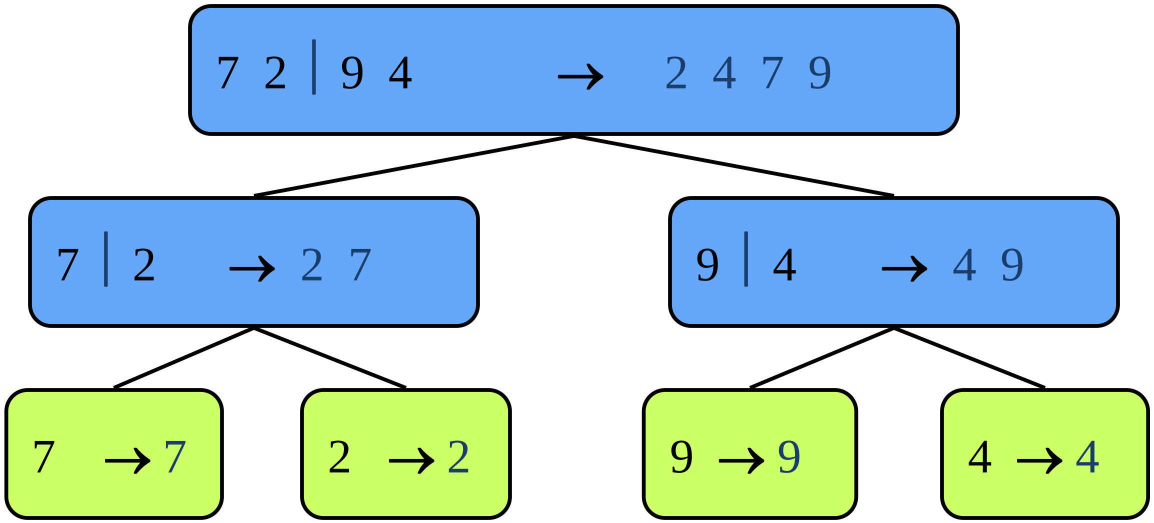
Algorithm *merge*(A, B)
  Input sequences A and B with $n/2$ elements each
  Output sorted sequence of $A \cup B$

  $S \leftarrow$ empty sequence
  while $\neg$*A.isEmpty*() $\wedge$ $\neg$*B.isEmpty*()
    if *A.first*().*element*() < *B.first*().*element*()
      S.insertLast(*A.remove*(*A.first*()))
    else
      S.insertLast(*B.remove*(*B.first*()))
  while $\neg$*A.isEmpty*()
    S.insertLast(*A.remove*(*A.first*()))
  while $\neg$*B.isEmpty*()
    S.insertLast(*B.remove*(*B.first*()))
  return S

Merge-Sort Tree

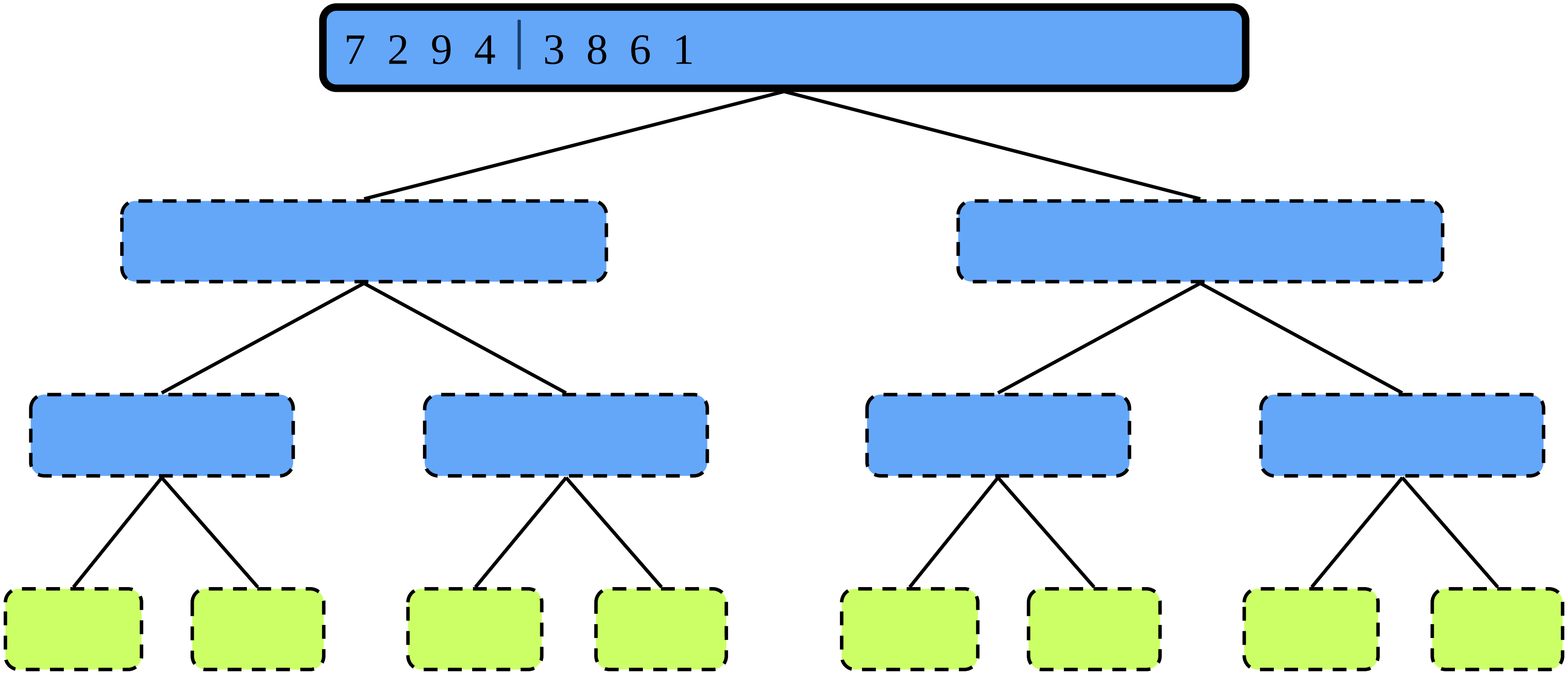
An execution of merge-sort is depicted by a binary tree

- 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 1



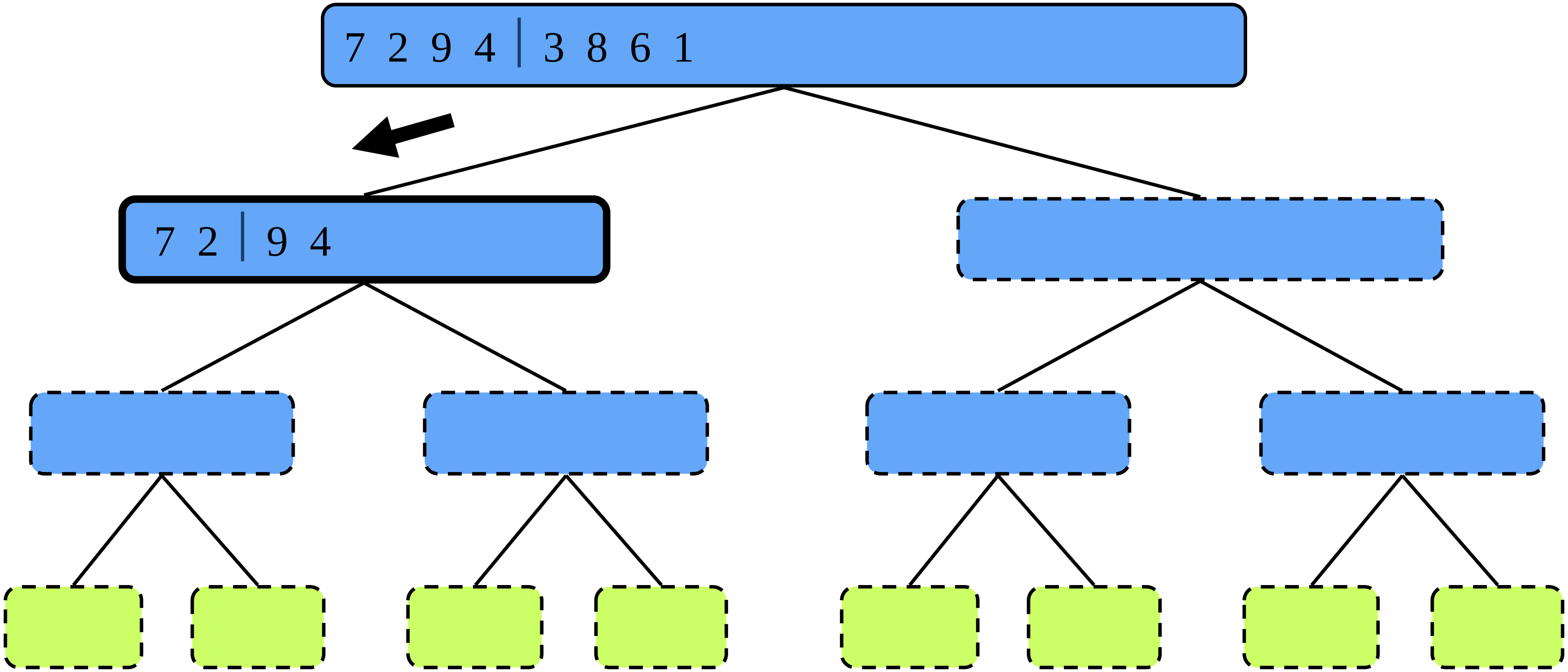
Execution Example

- Partition



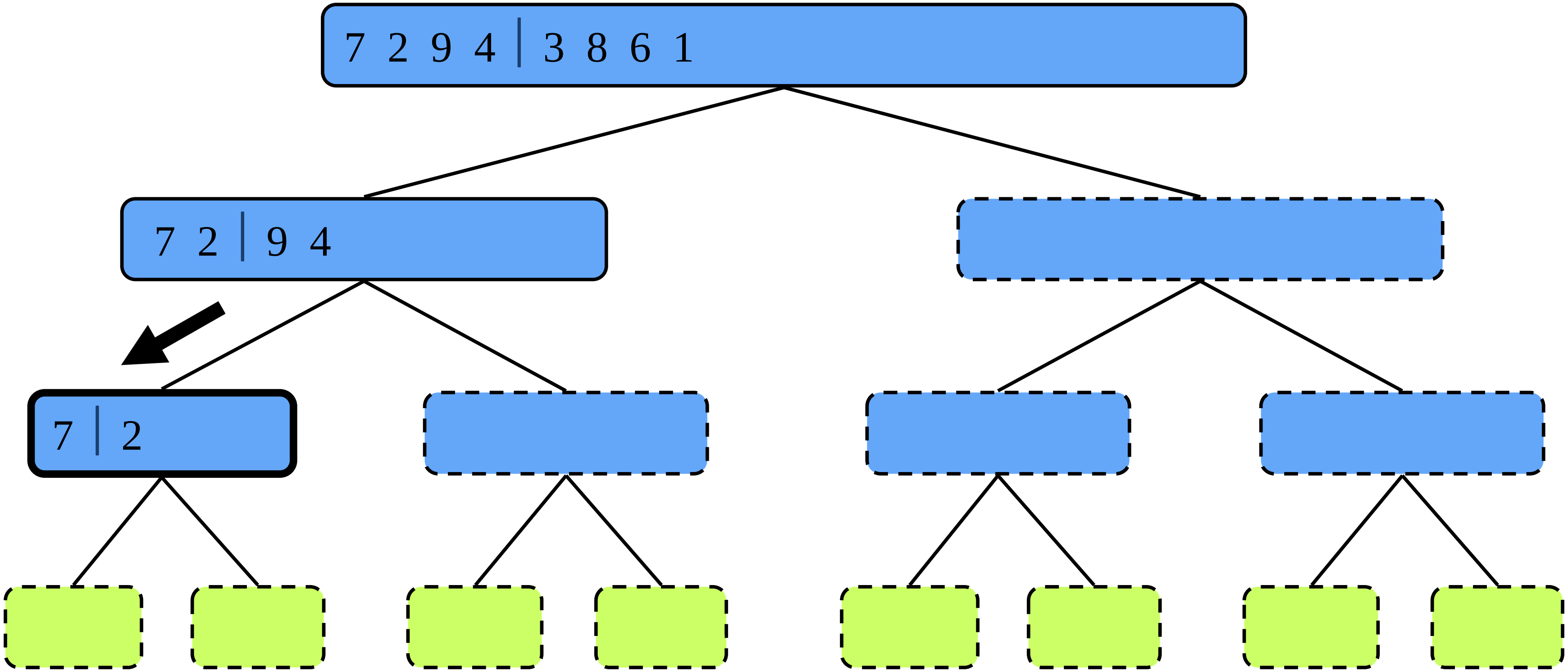
Execution Example (cont.)

- Recursive call, partition



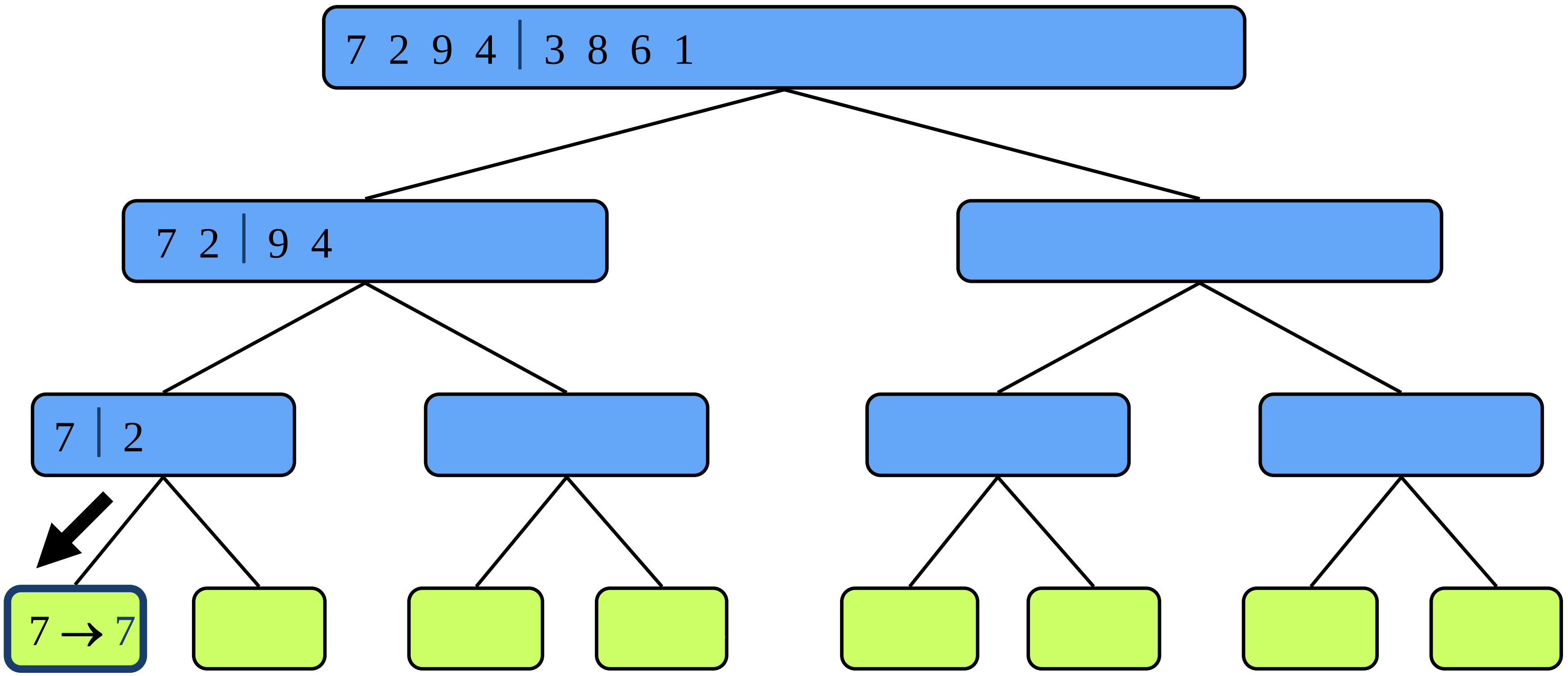
Execution Example (cont.)

- Recursive call, partition



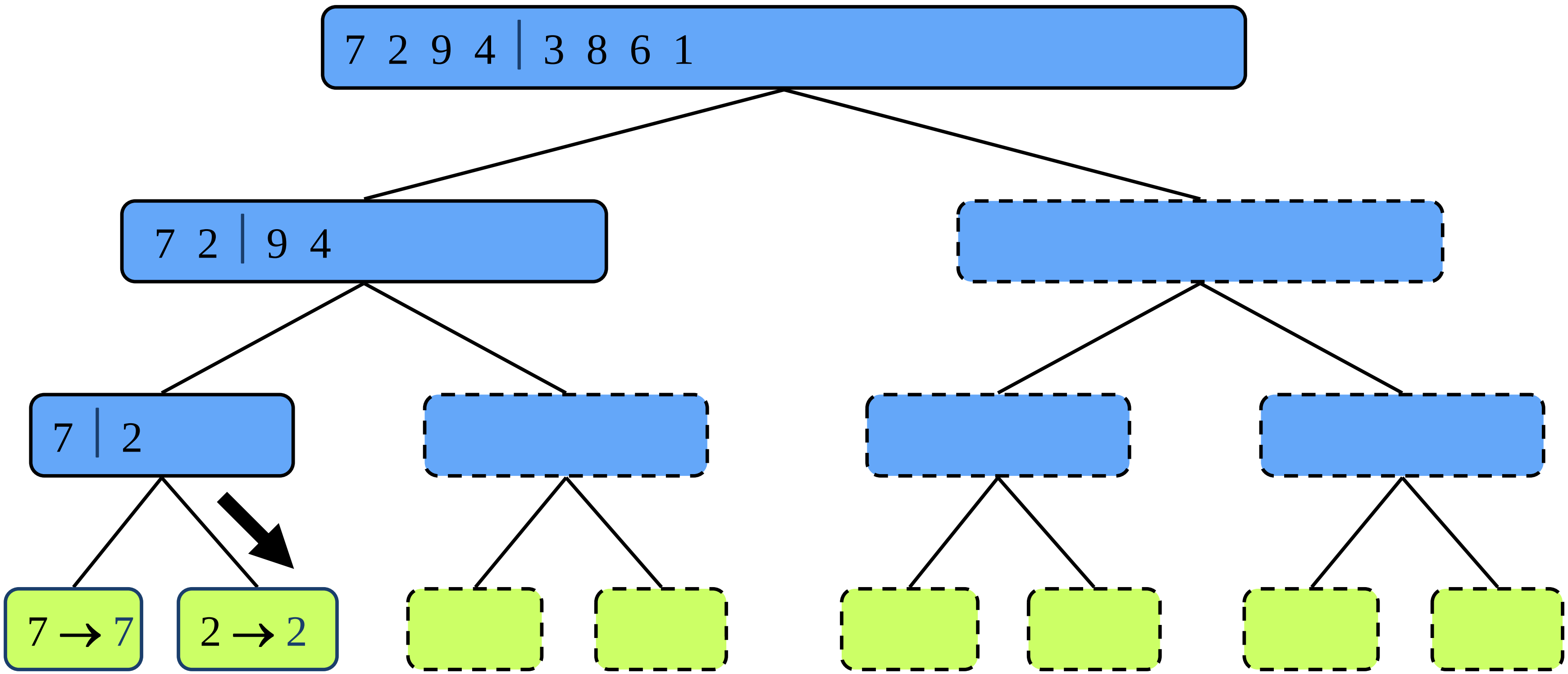
Execution Example (cont.)

- Recursive call, base case



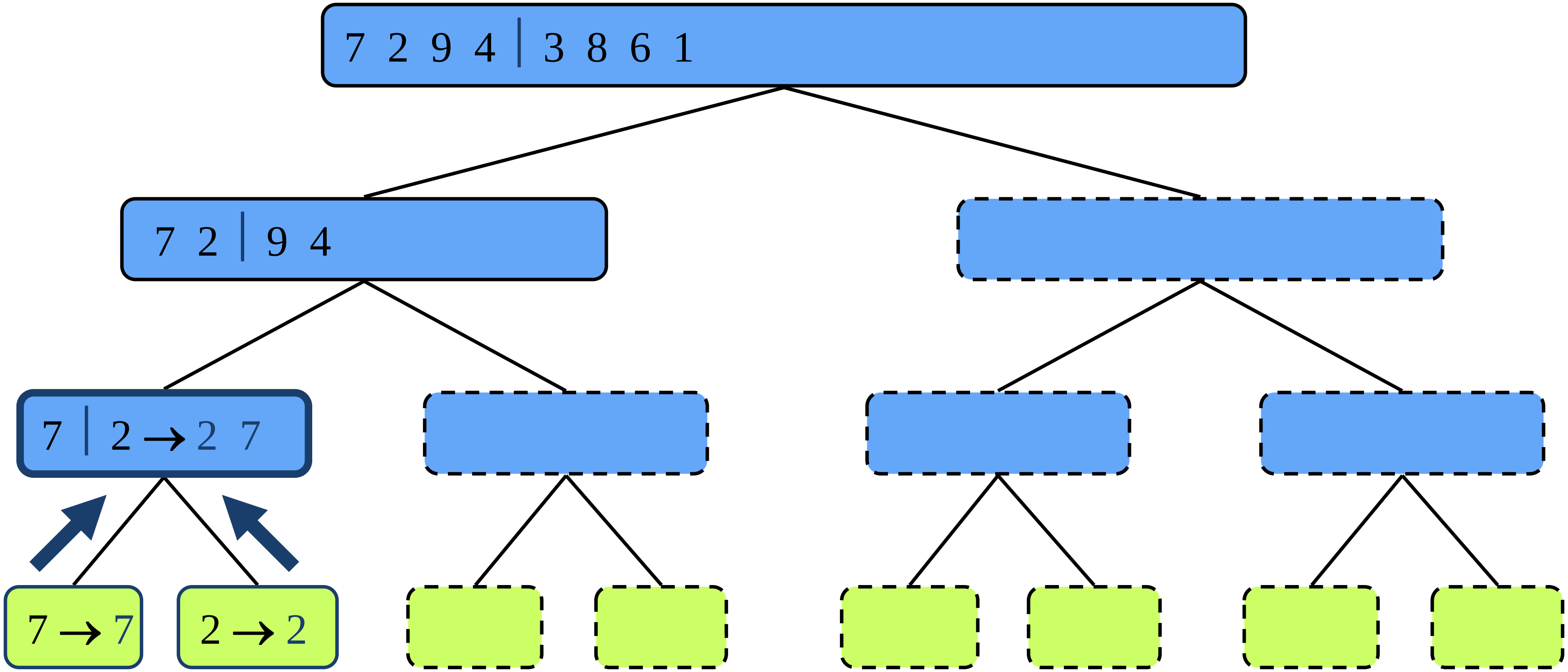
Execution Example (cont.)

- Recursive call, base case



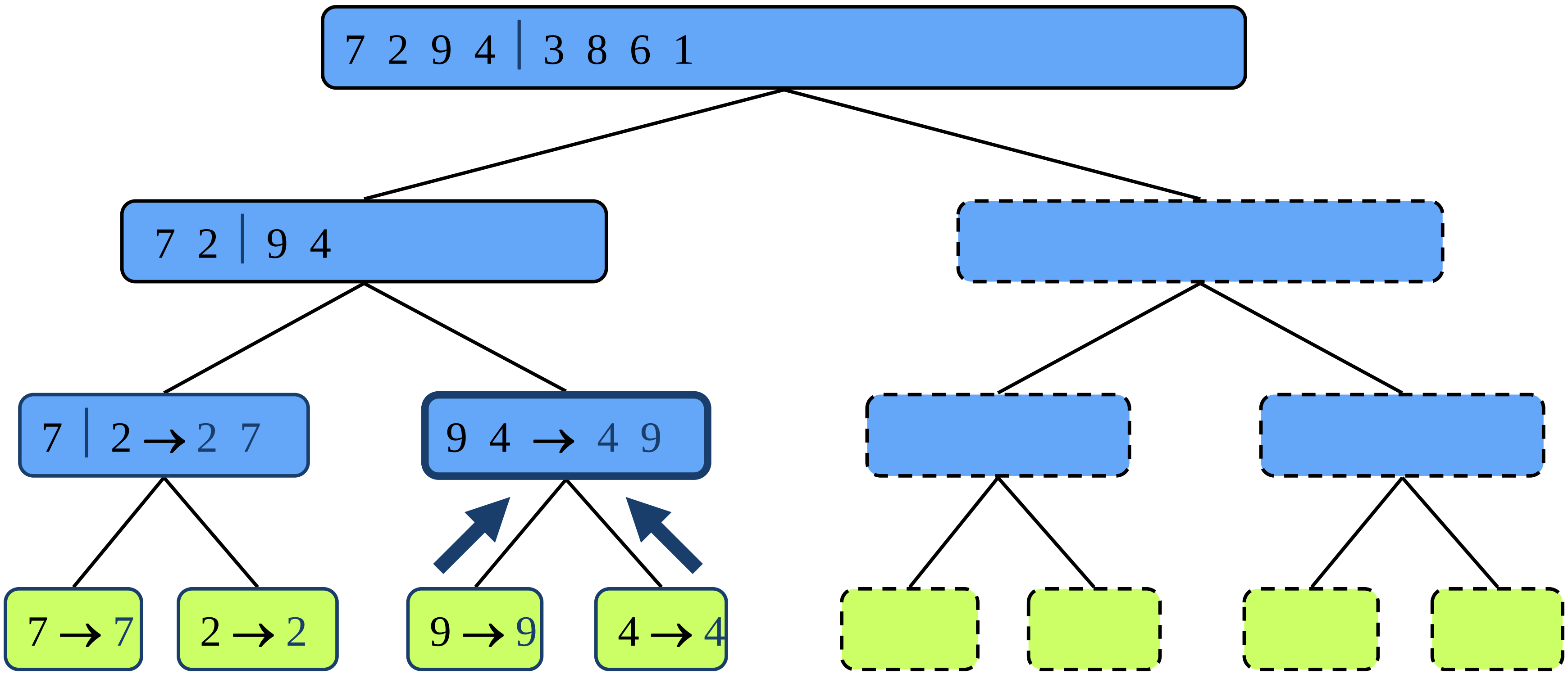
Execution Example (cont.)

- Merge



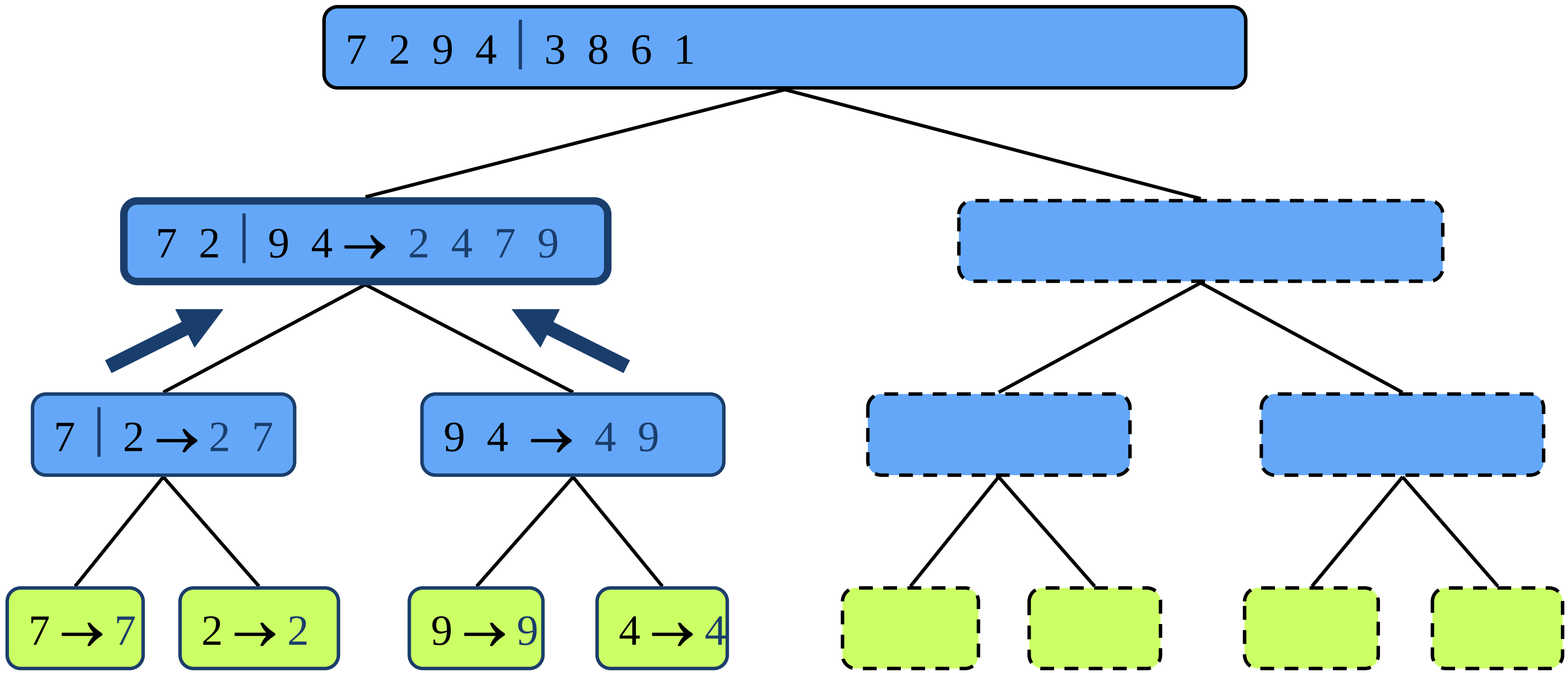
Execution Example (cont.)

- Recursive call, …, base case, merge

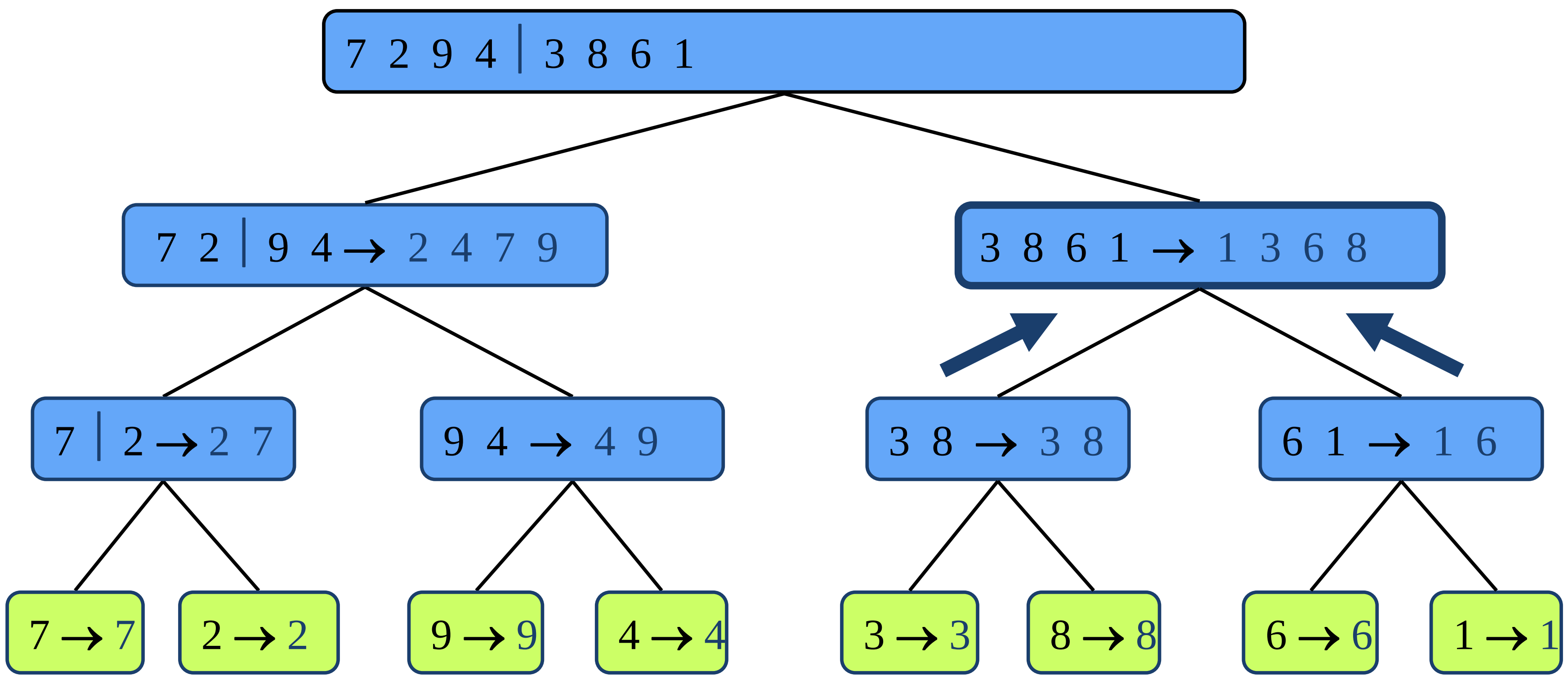


Execution Example (cont.)

- Merge

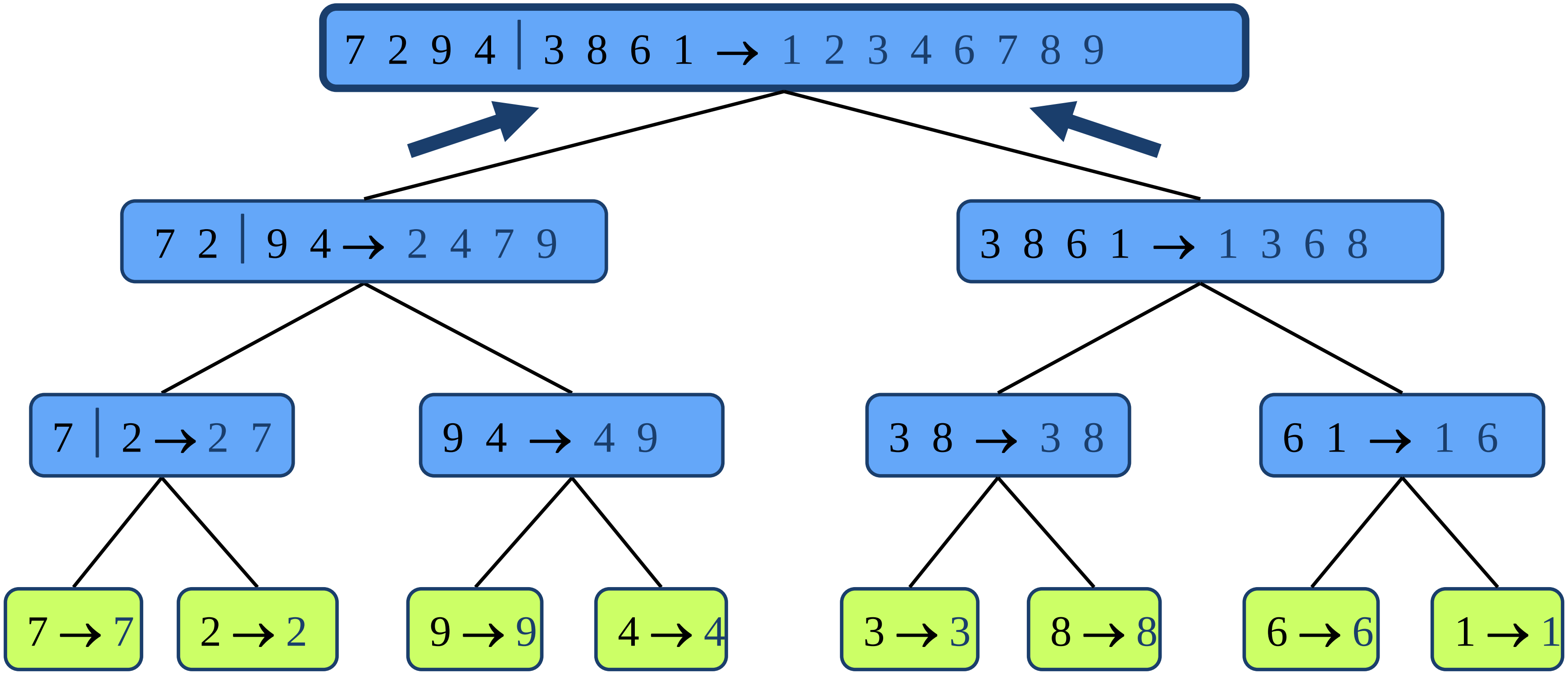


Execution Example (cont.)



Execution Example (cont.)

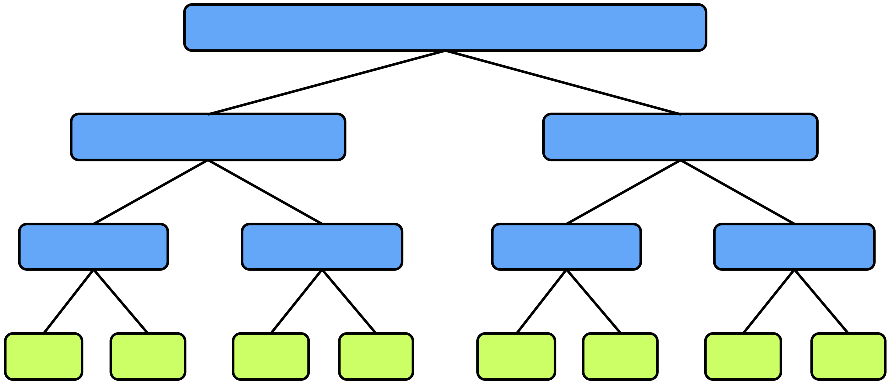
- Merge



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide the sequence in half
- 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	$n/2^i$
…	…	…



Comparing sorting algorithms

Consider the following when evaluating a sorting algorithm:

- Time complexity
- Space complexity
  - An in-place algorithm requires only $n + O(1)$ space, using the already given space for the n elements and $O(1)$ additional space
- Stability
  - A sorting algorithm is stable if it preserves the original relative ordering of elements with equal value
  - Ex: Unsorted sequence (**B**, **b**, a, c). Suppose B = b and a < b < c.
    - Stable sorted: (a, **B, b**, c)
    - Unstable sorted: (a, **b, B**, c)
  - Necessary if we want to sort repeatedly by different attributes (i.e., sort by first name, then sort again by last name)

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	◈ in-place ◈ not stable ◈ for small data sets (< 1K)
insertion-sort	$O(n^2)$	◈ in-place ◈ stable ◈ for small data sets (< 1K)
heap-sort	$O(n \log n)$	◈not stable ◈ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	◈ not in-place ◈ stable ◈ sequential data access ◈ for huge data sets (> 1M)

Sets

Set ADT

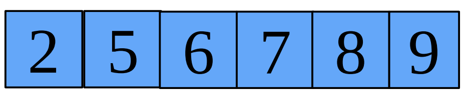
- A collection of unordered distinct objects
  - there is no inherent ordering of elements in a set, but keeping the elements sorted can lead to more efficient set operations

- Main operations
  - union(B): executes $A \leftarrow A \cup B$
  - intersect(B): executes $A \leftarrow A \cap B$
  - subtract(B): executes $A \leftarrow A - B$
  - implemented using a generic version of the merge algorithm

- Running time of an operation should be at most $O(n_A + n_B)$

Storing a Set in a List

- We can implement a set with a list
- Elements are sorted according to some canonical ordering
- Space used is $O(n)$



Generic Merging

- Generalized merge of two sorted lists *A* and *B*
- Auxiliary methods aIsLess, bIsLess, bothAreEqual decide whether to add the element to list *S* based on what main operation is performed

```
Algorithm genericMerge(A, B)
    S ← empty sequence
    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        a ← A.first().element();  b ← B.first().element()
        if a < b
            aIsLess(a, S);  A.remove(A.first())
        else if b < a
            bIsLess(b, S);  B.remove(B.first())
        else { b = a }
            bothAreEqual(a, b, S)
            A.remove(A.first());  B.remove(B.first())
    while ¬A.isEmpty()
        aIsLess(a, S);  A.remove(A.first())
    while ¬B.isEmpty()
        bIsLess(b, S);  B.remove(B.first())
    return S
```

Example: Union

- if $a < b$, copy a to output sequence and go to next element of A
- if $a = b$, copy a to output sequence and go to next element of A and B
- if $a > b$, copy b to output sequence and go to next element of B

A	2	5	6	7	9

B	2	7	8	10

$S = A \cup B$

Example: Union

- if $a < b$, copy a to output sequence and go to next element of A
- if $a = b$, copy a to output sequence and go to next element of A and B
- if $a > b$, copy b to output sequence and go to next element of B

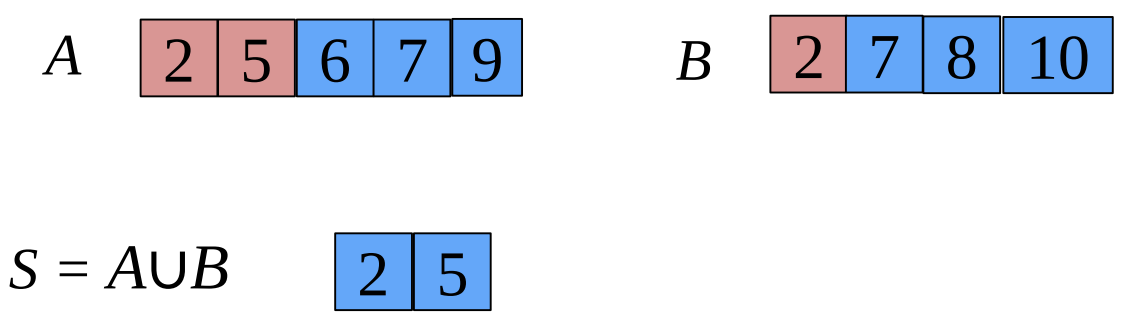
A	2	5	6	7	9

B	2	7	8	10

$S = A \cup B$	2

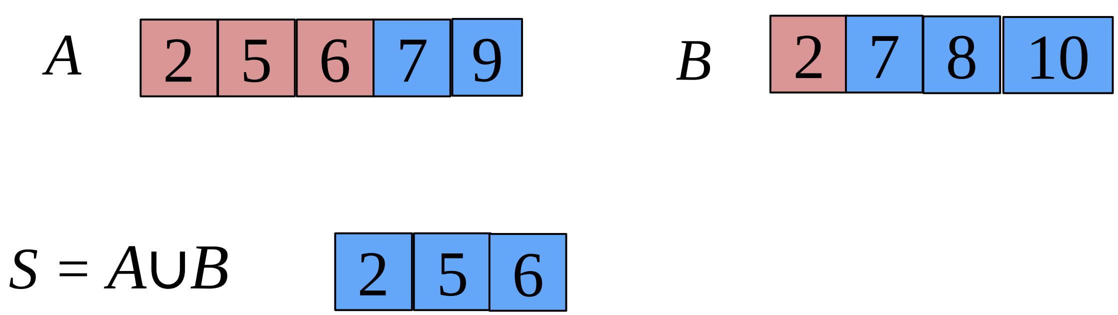
Example: Union

- if $a < b$, copy a to output sequence and go to next element of A
- if $a = b$, copy a to output sequence and go to next element of A and B
- if $a > b$, copy b to output sequence and go to next element of B



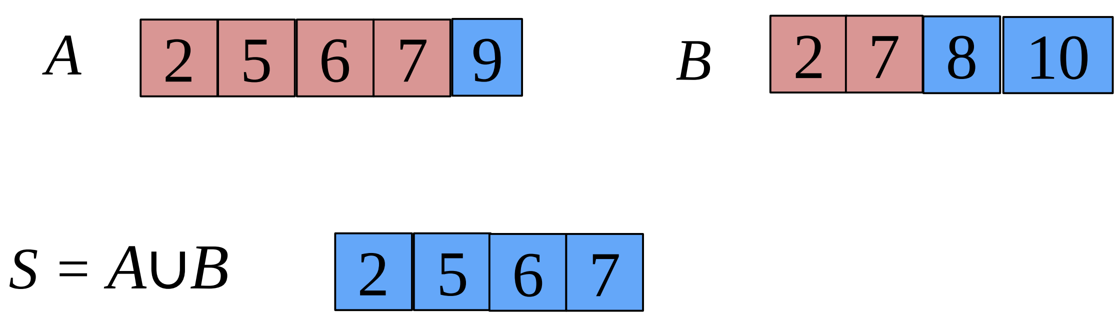
Example: Union

- if $a < b$, copy a to output sequence and go to next element of A
- if $a = b$, copy a to output sequence and go to next element of A and B
- if $a > b$, copy b to output sequence and go to next element of B



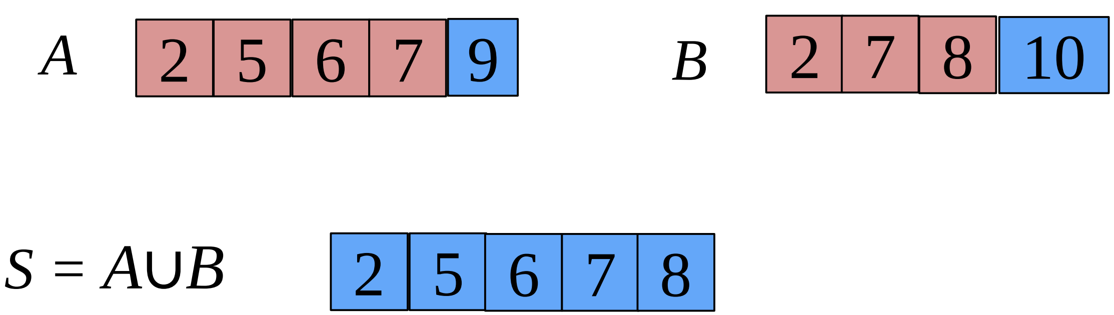
Example: Union

- if $a < b$, copy a to output sequence and go to next element of A
- if $a = b$, copy a to output sequence and go to next element of A and B
- if $a > b$, copy b to output sequence and go to next element of B



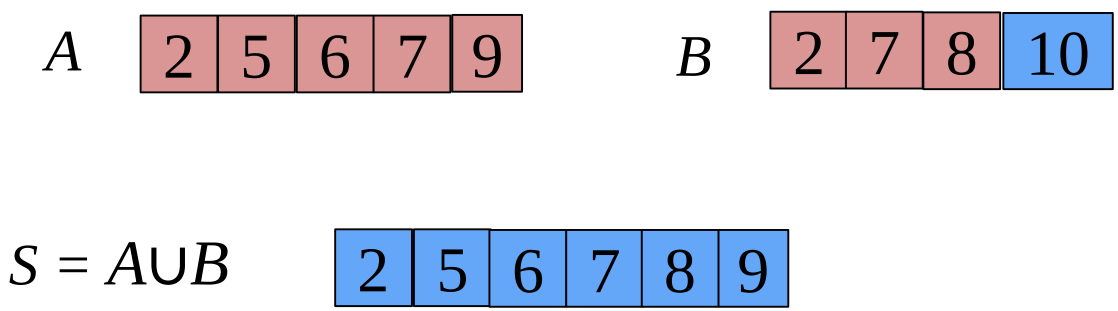
Example: Union

- if $a < b$, copy a to output sequence and go to next element of A
- if $a = b$, copy a to output sequence and go to next element of A and B
- if $a > b$, copy b to output sequence and go to next element of B



Example: Union

- if $a < b$, copy a to output sequence and go to next element of A
- if $a = b$, copy a to output sequence and go to next element of A and B
- if $a > b$, copy b to output sequence and go to next element of B



Example: Union

- if $a < b$, copy a to output sequence and go to next element of A
- if $a = b$, copy a to output sequence and go to next element of A and B
- if $a > b$, copy b to output sequence and go to next element of B

A	2	5	6	7	9

B	2	7	8	10

$S = A \cup B$	2	5	6	7	8	9	10

Using Generic Merge for Set Operations

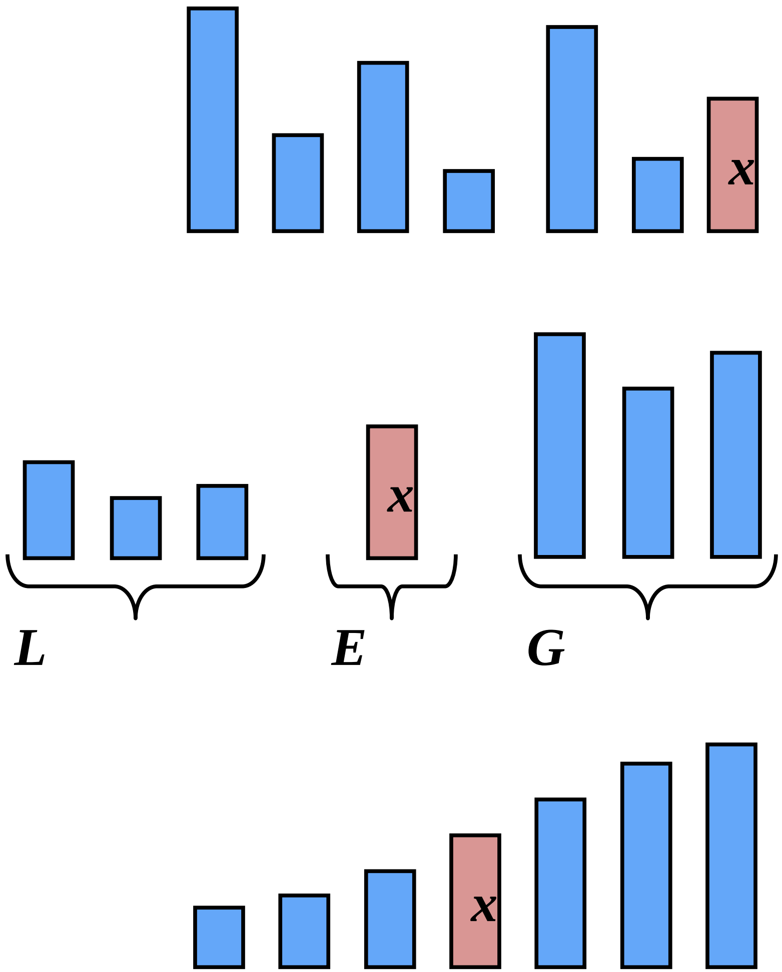
- Any of the set operations can be implemented using a generic merge
- For example:
  - intersection: only copy elements that are duplicated in both lists
  - subtraction: only copy elements from *A* that are not equal to those in *B*
- All methods run in linear time.

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***

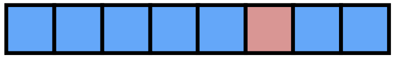


The choice of the pivot affects the algorithm's performance.

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.

S 

```
Algorithm partition(S, x)
    Input sequence S, pivot element x
    Output subsequences L, E, G
    L, E, G ← empty sequences
    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
```

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

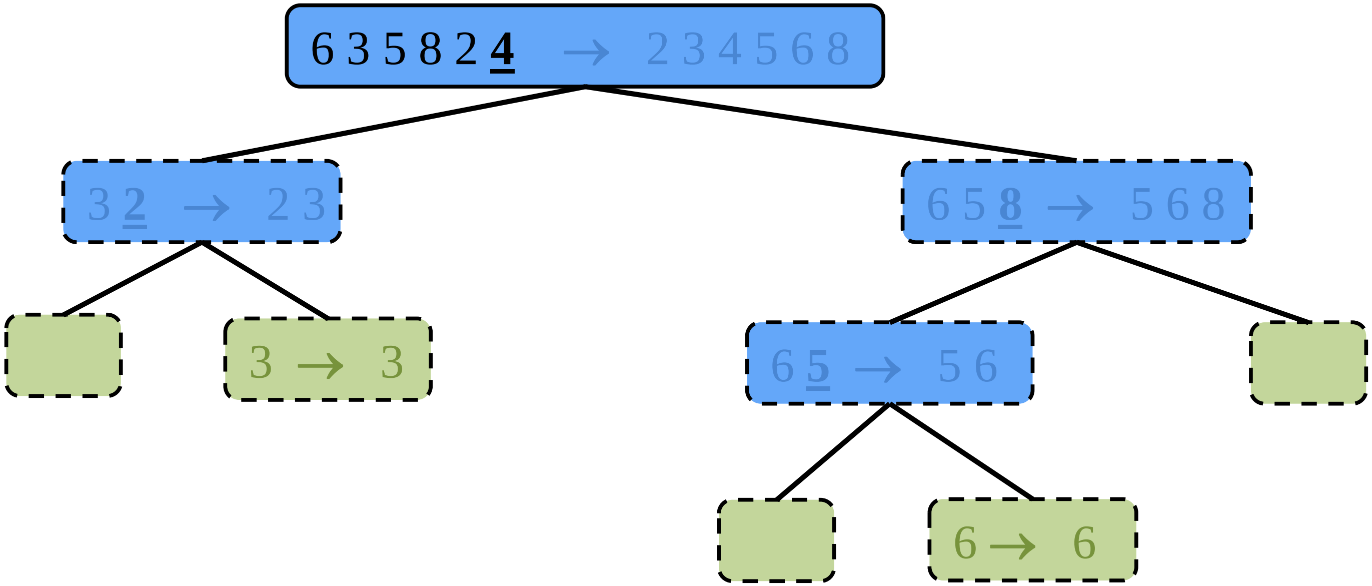
Quick-Sort Tree

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

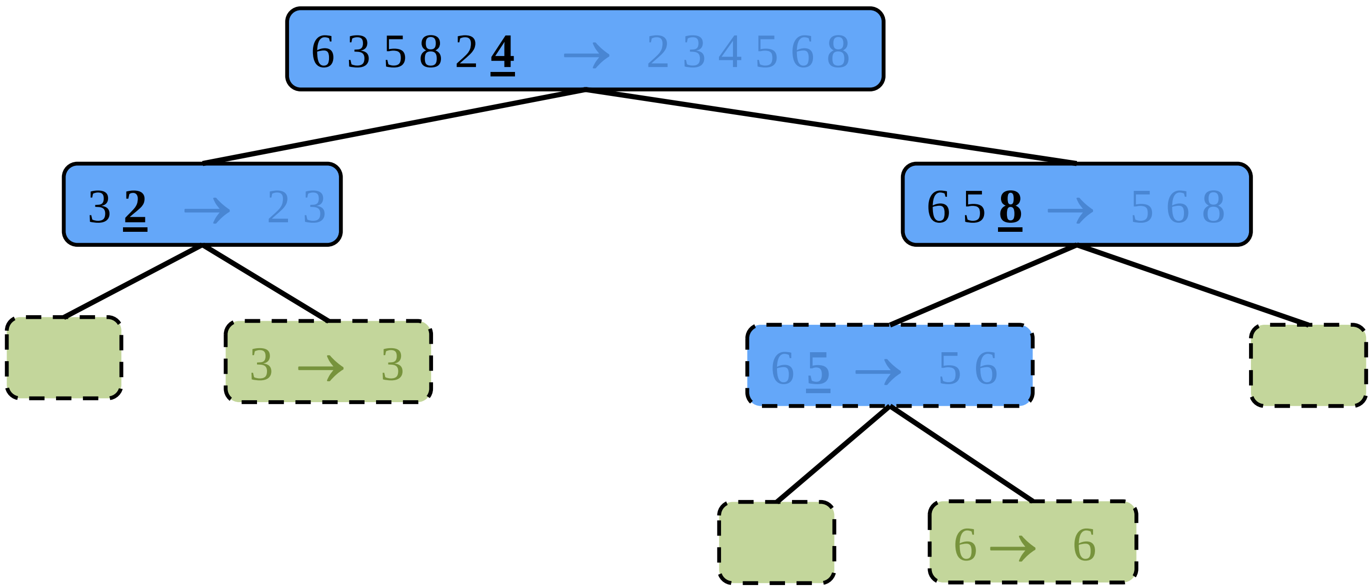
Quick Sort Execution

- Strategy: Select the last element as the pivot



Quick Sort Execution

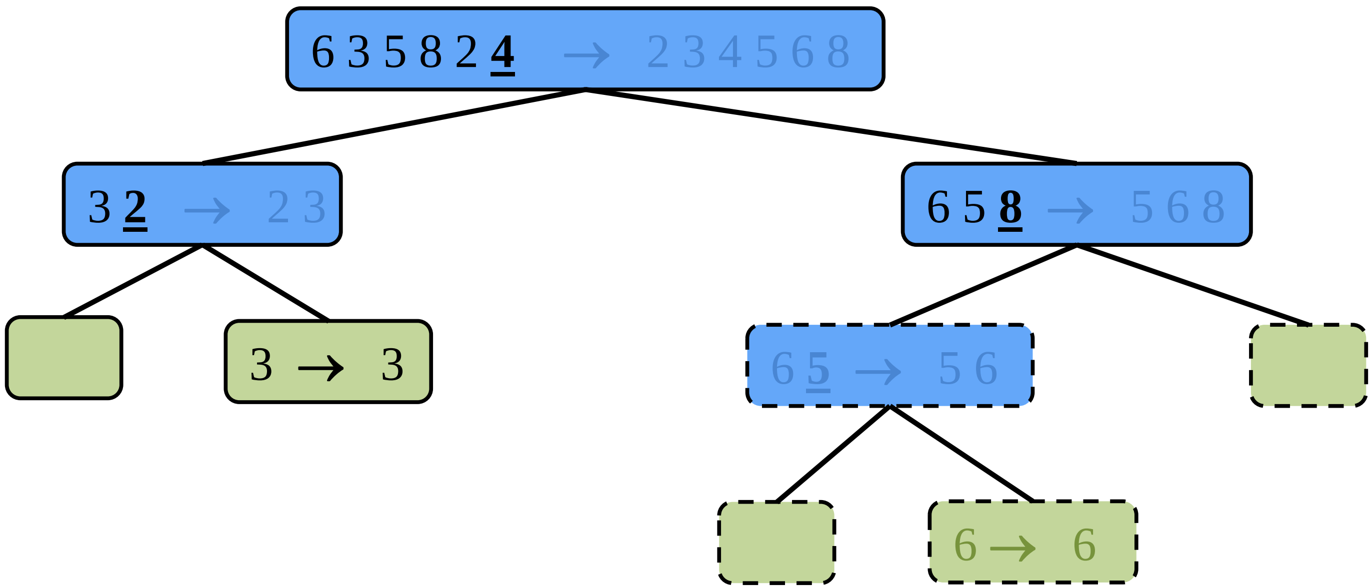
- Strategy: Select the last element as the pivot



- Select pivot, partition, recursive call

Quick Sort Execution

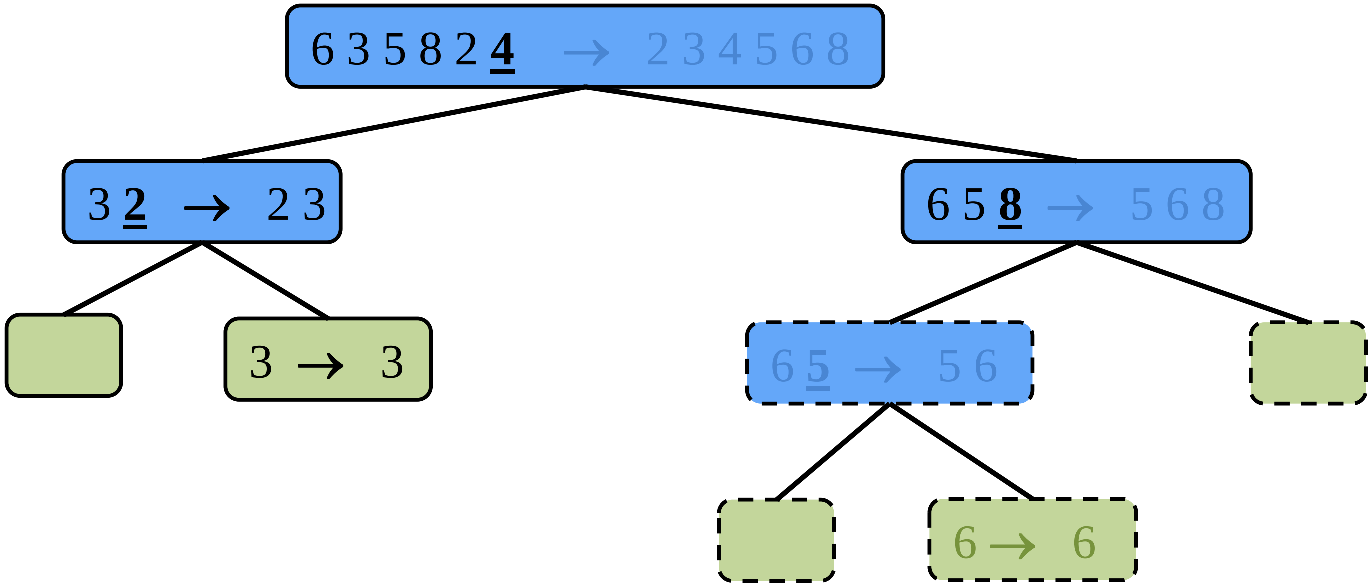
- Strategy: Select the last element as the pivot



- Select pivot, partition, recursive call

Quick Sort Execution

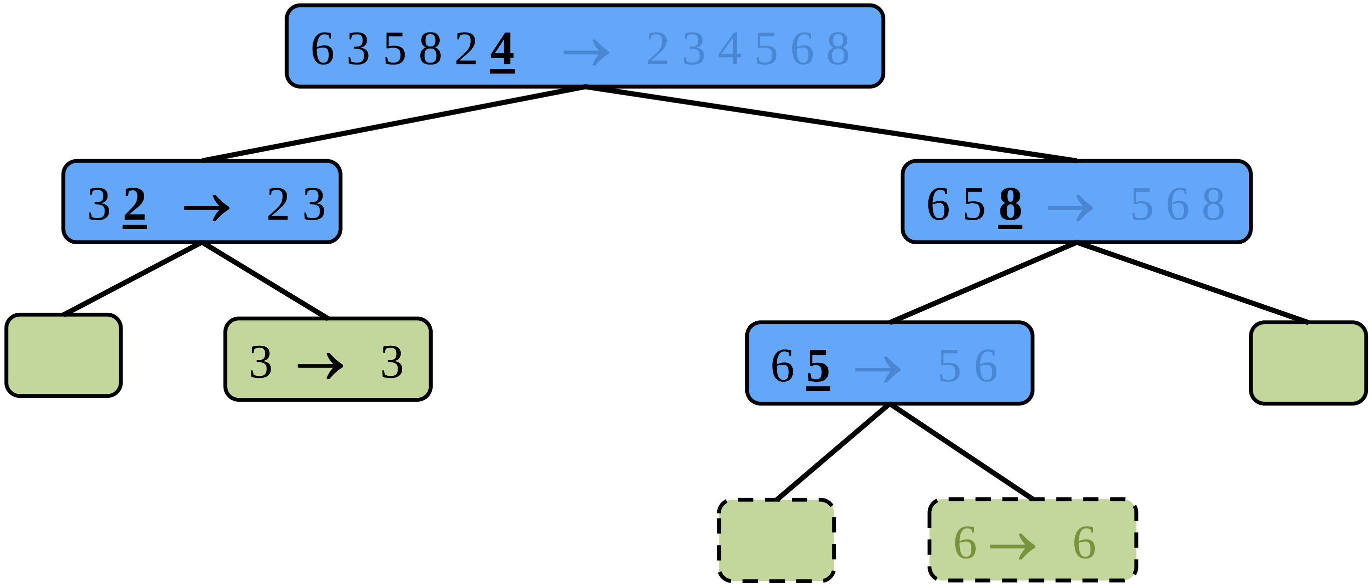
- Strategy: Select the last element as the pivot



- Join

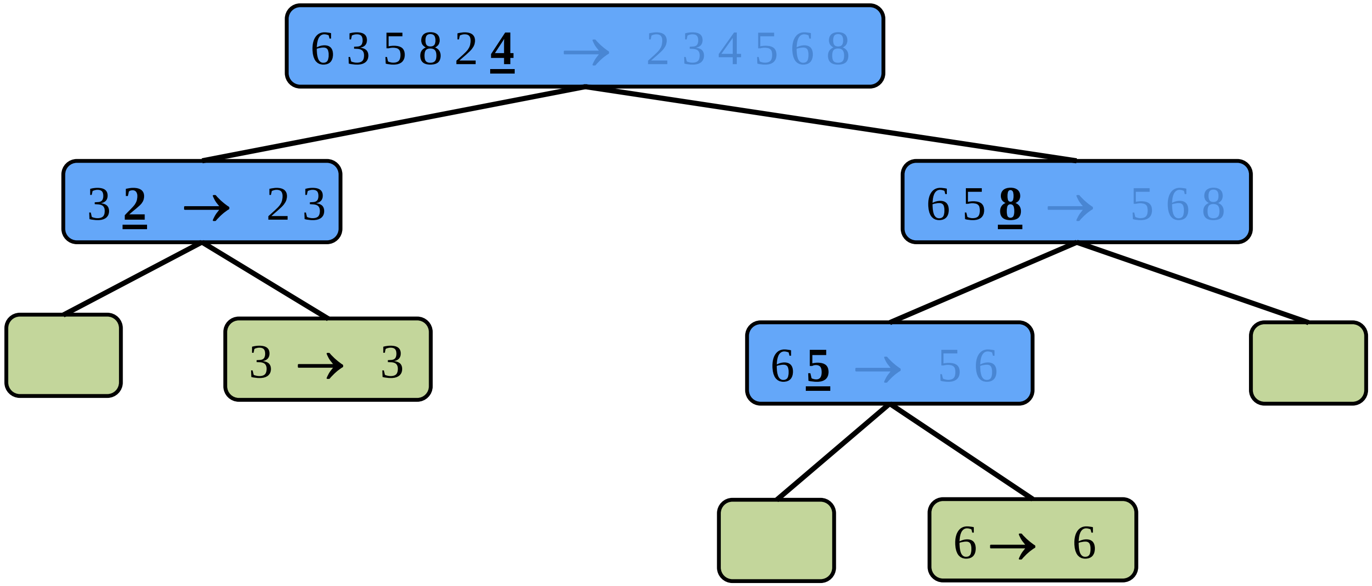
Quick Sort Execution

- Strategy: Select the last element as the pivot



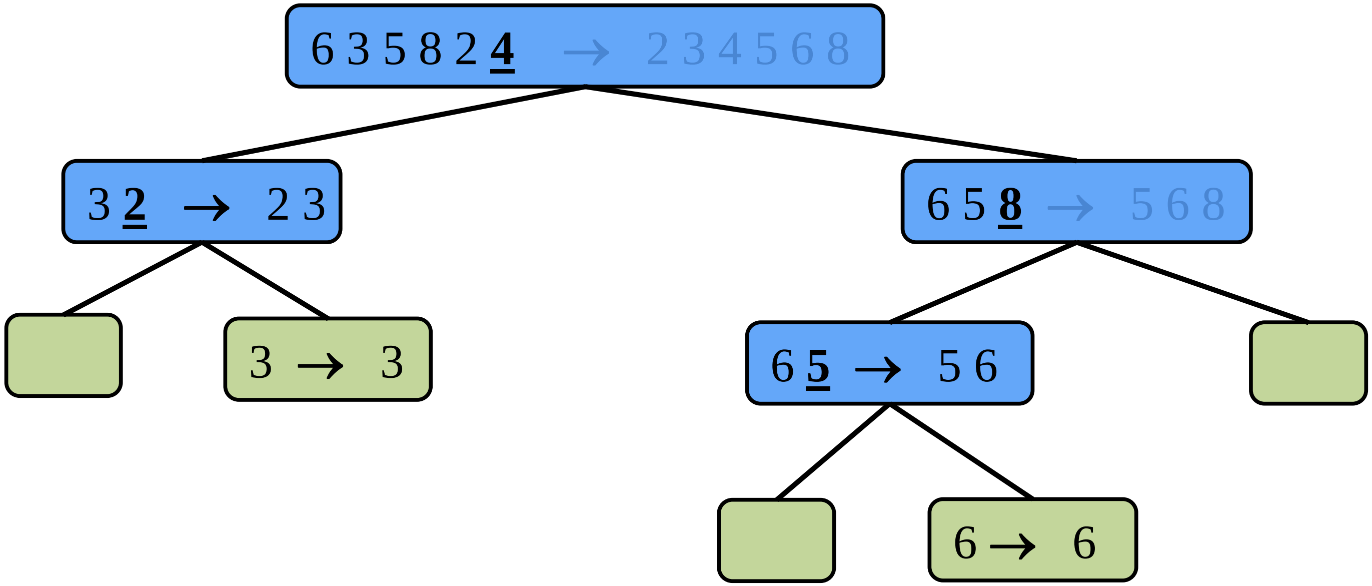
Quick Sort Execution

- Strategy: Select the last element as the pivot



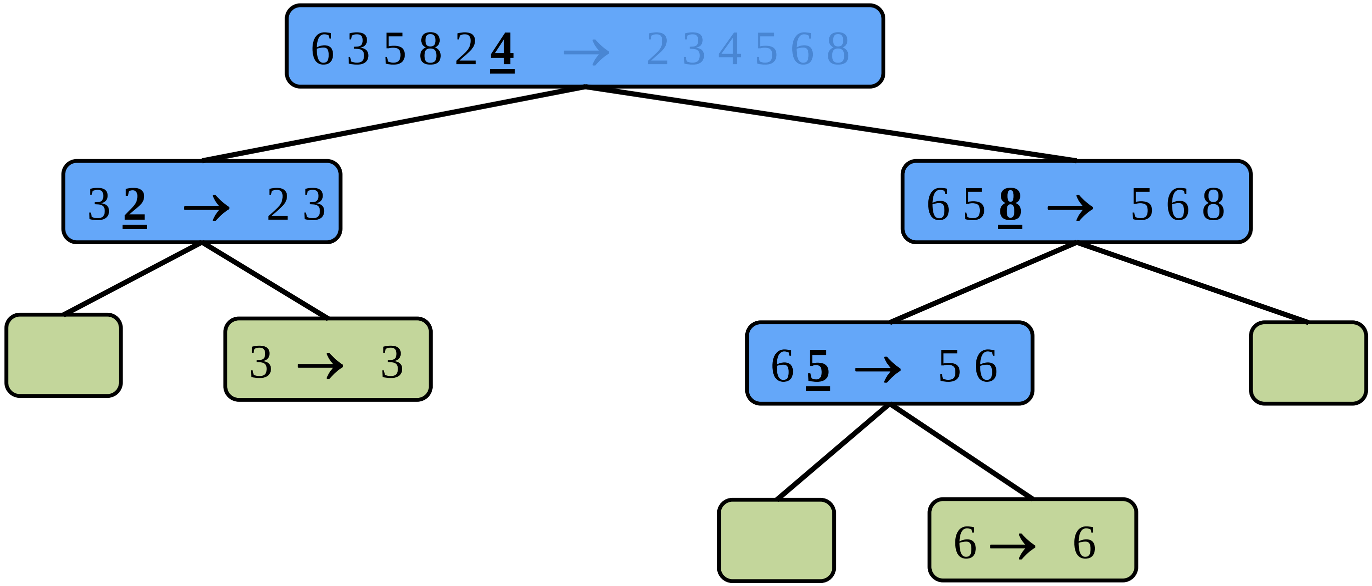
Quick Sort Execution

- Strategy: Select the last element as the pivot



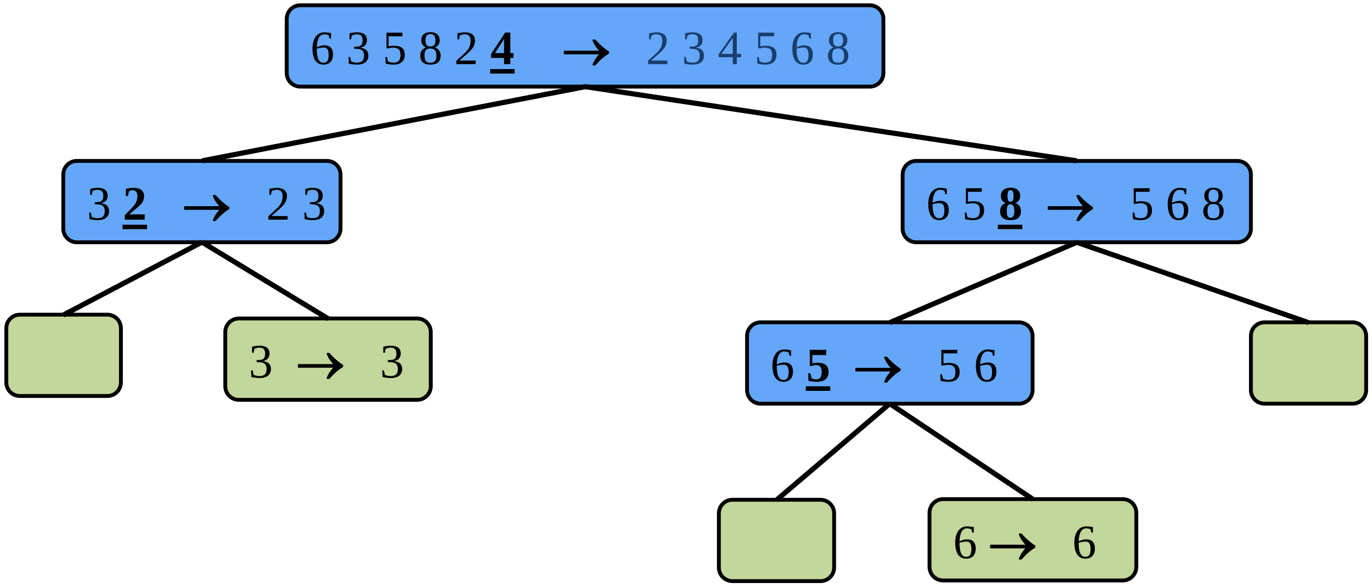
Quick Sort Execution

- Strategy: Select the last element as the pivot



Quick Sort Execution

- Strategy: Select the last element as the pivot



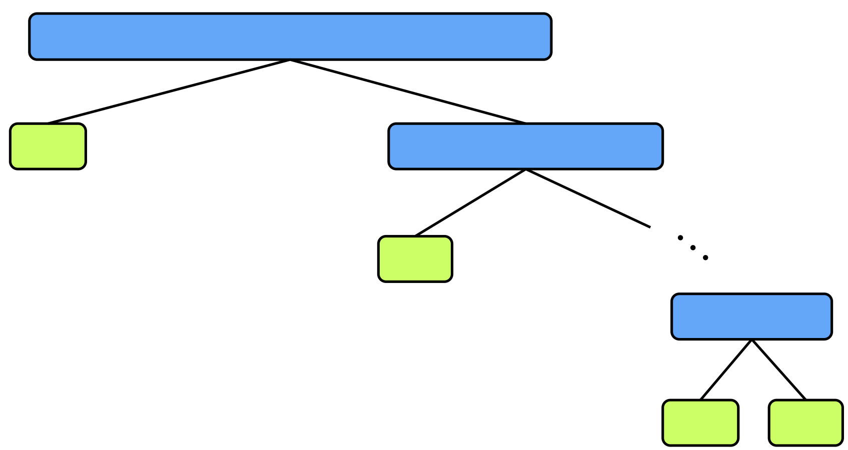
Worst-case Running Time

Occurs when the pivot is the unique minimum or maximum element

- One of $\boldsymbol{L}$ and $\boldsymbol{G}$ has size $n - 1$ and the other has size 0
- The running time is proportional to the sum: $n + (n - 1) + \ldots + 2 + 1$
- 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 $O(n^2)$

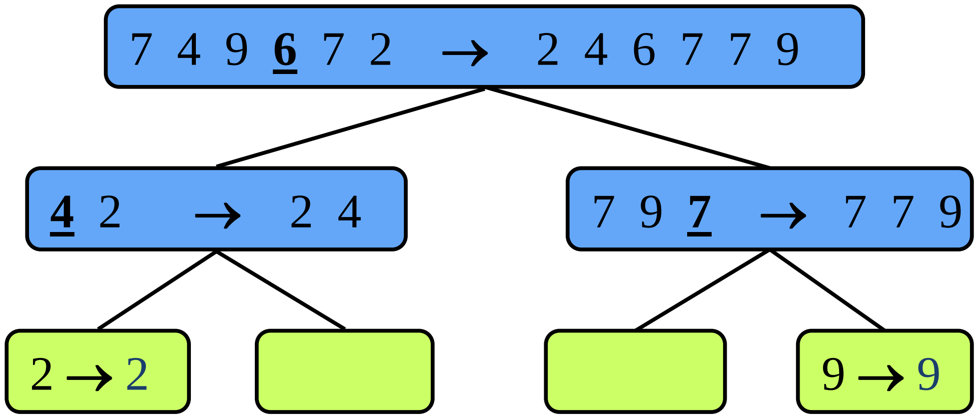
depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



Randomized Quick Sort

Pivot selection strategy: choose a random element as the pivot

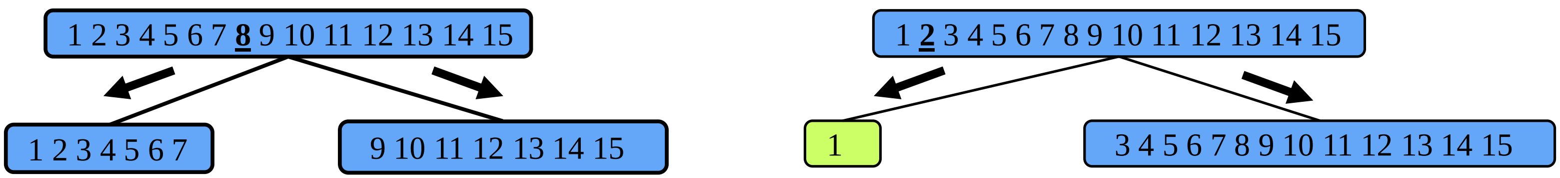
- Still has worst-case running time $O(n^2)$
  - Due to random selection, this case is highly unlikely
- Expected running time is $O(n \log n)$



Expected Running Time

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

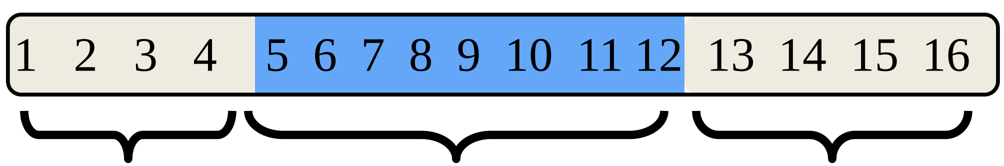
- **Good call:** the sizes of L and G are each less than $3s/4$
- **Bad call:** one of L and G has size greater than $3s/4$



Good call **Bad call**

A call is good with probability 1/2

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



Bad pivots **Good pivots** **Bad pivots**

Quick Sort Pseudocode

The following procedure implements quicksort:

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

To sort an entire array A, the initial call is QUICKSORT(A, 1, $A.length$).

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,..\, r]$ in place.

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	◈ in-place, not stable ◈ slow (good for small inputs)
insertion-sort	$O(n^2)$	◈ in-place, stable ◈ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	◈ in-place, not stable ◈ randomized ◈ fast (good for large inputs)
heap-sort	$O(n \log n)$	◈not stable ◈ fast (good for large inputs)
merge-sort	$O(n \log n)$	◈ not in-place, stable ◈ sequential data access ◈ fast (good for huge inputs)

Exercise
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.

Exercise

- How would you modify QUICKSORT to sort into nonincreasing order?

Sorting Lower Bound

Comparison Based Sorting

Recall - Sorting

- input: A sequence of n values $x_1, x_2, \ldots, x_n$
- output: A permutation $y_1, y_2, \ldots, y_n$ such that $y_1 \leq y_2 \leq \ldots \leq y_n$

Many algorithms are comparison based

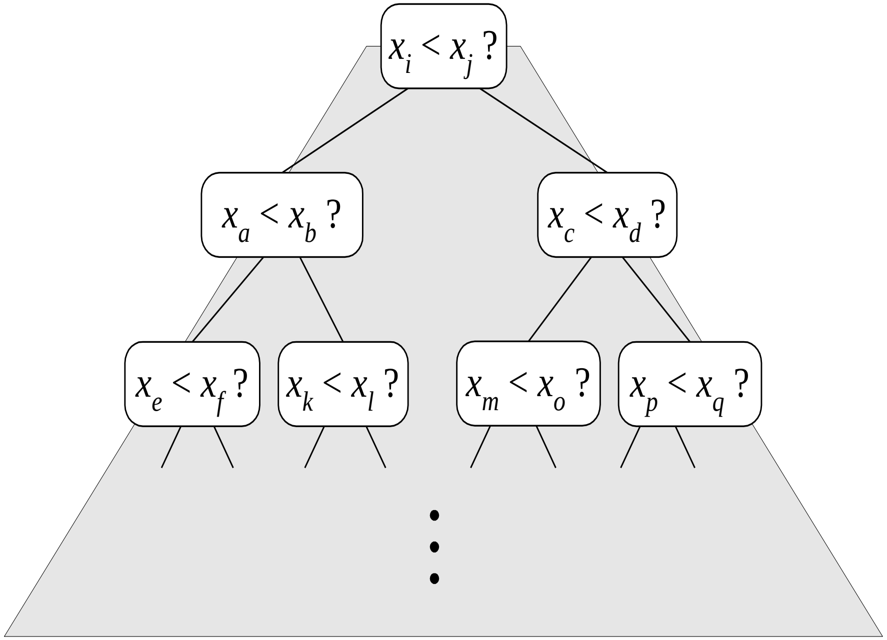
- they sort by making comparisons between pairs of objects
- ex: selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, …
- best so far runs in $O(n\log n)$ time… can we do better?

Let's derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements $x_1, x_2, \ldots, x_n$

Counting Comparisons

A decision tree represents every sequence of comparisons that an algorithm might make on an input of size n

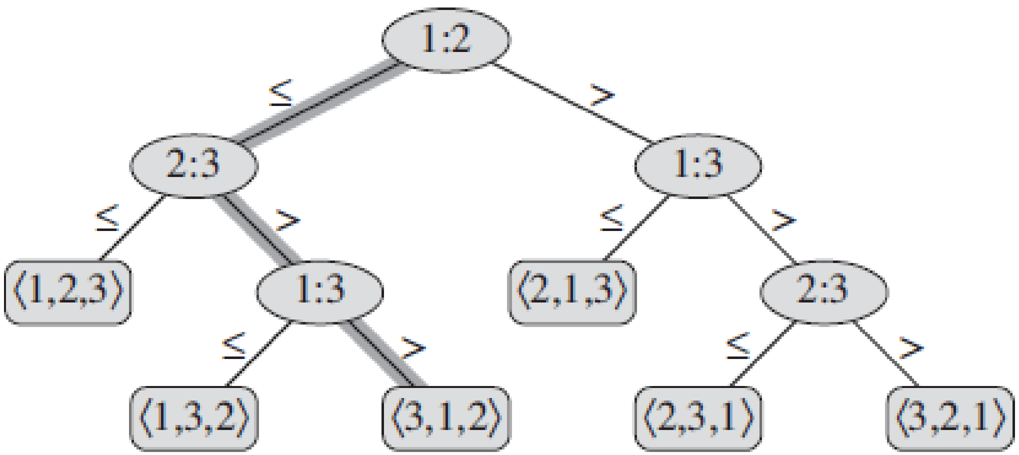
- each possible run of the algorithm corresponds to a root-to-leaf path
- at each internal node a comparison $x_i < x_j$ is performed and branching made
- nodes annotated with the orderings consistent with the comparisons made so far
- leaf contains result of computation (a total order of elements)



Decision Tree Example

Algorithm: insertion sort

Instance (n = 3): the numbers *1,2,3*



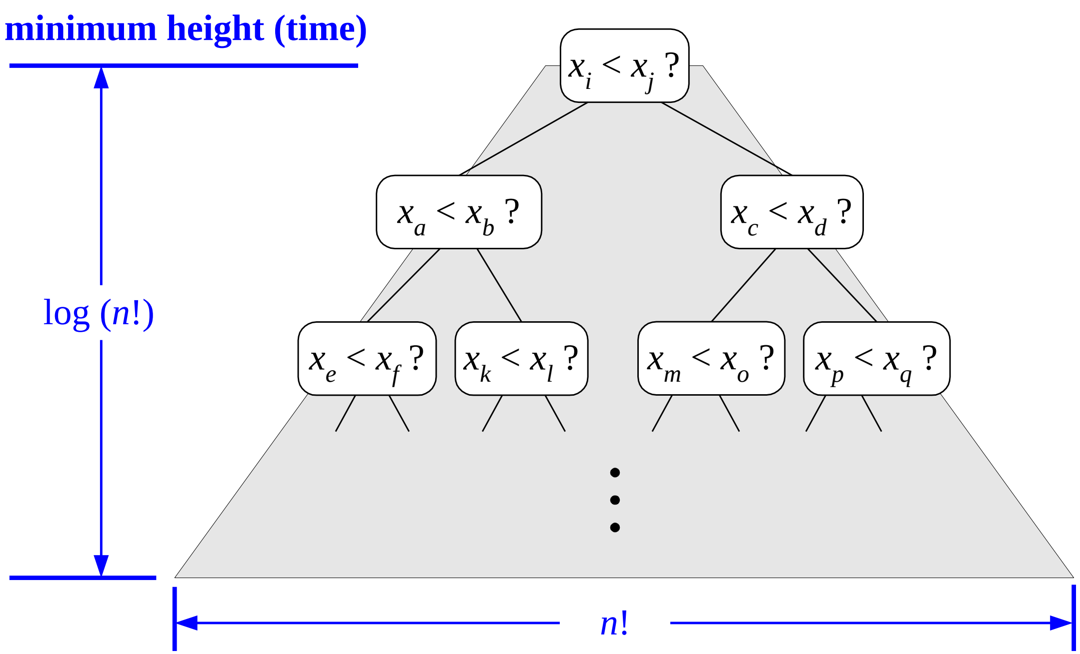
Height of a Decision Tree

Claim: The height of a decision tree is $\Omega(n\log n)$.

Proof: There are n! leaves. A tree of height h has at most 2^h leaves. So

$$\begin{aligned} 2^h &\geq n! \\ h &\geq \log_2(n!) \\ &\geq c\cdot\log_2(n^n) \\ &= c\cdot n\log_2 n. \end{aligned}$$

Thus, $h \in \Omega(n\log n)$.



Lower Bound

Theorem: Every comparison sort requires $\Omega(n\log n)$ in the worst-case.

Proof: Given a comparison sort, we look at the decision tree it generates on an input of size n.

- Each path from root to leaf is one possible sequence of comparisons
- Length of the path is the number of comparisons for that instance
- Height of the tree is the worst-case path length (number of comparisons)

Height of the tree is $\Omega(n\log n)$ by the previous claim. Hence, every comparison sort requires $\Omega(n\log n)$ comparisons.