SE274 Data Structure

Lecture 7: Sorting and Selection
(textbook: Chapter 12)

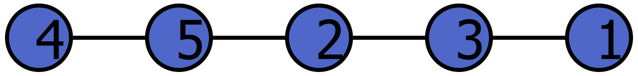
Apr 06, 2020

Instructor: Sunjun Kim

Information&Communication Engineering, DGIST

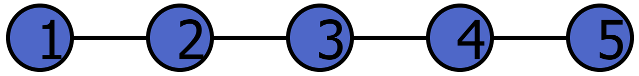
Recap: Insertion sort, Selection Sort

- **Selection sort**: Priority Queue Implementation with an unsorted list



- Performance:
  - add takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - Remove_min and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

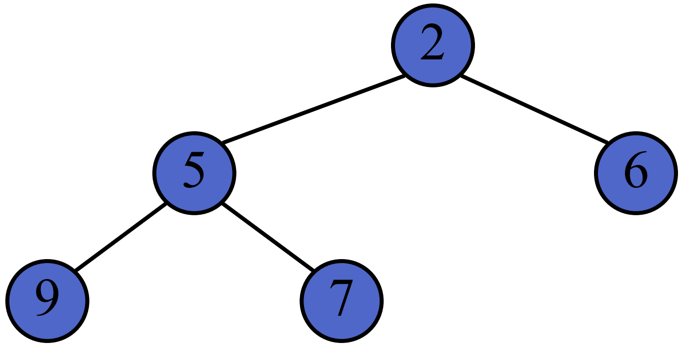
- **Selection sort**: Priority Queue Implementation with a sorted list



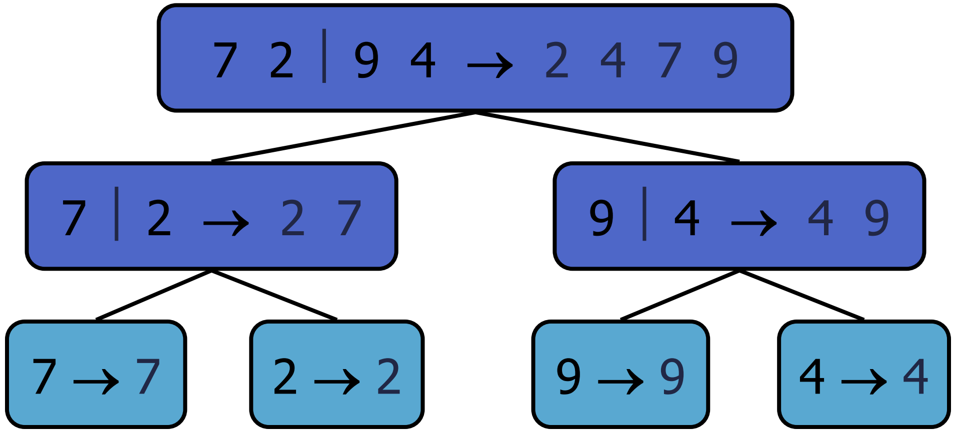
- Performance:
  - add takes $O(n)$ time since we have to find the place where to insert the item
  - remove_min and min take $O(1)$ time, since the smallest key is at the beginning

Recap: heapsort

- ***Add*** all the items to a heap.
- Repeat ***Remove_min*** until the heap is empty.
- ***Add*** takes $\boldsymbol{O}(log\ \boldsymbol{n})$ time, remove_min take $\boldsymbol{O}(log\ \boldsymbol{n})$ time.
- Heapsort takes $\boldsymbol{O}(\boldsymbol{n}\ log\ \boldsymbol{n})$



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 $\boldsymbol{S}_2$
  - Conquer: recursively solve the subproblems associated with $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - Combine: combine the solutions for $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a solution for $\boldsymbol{S}$
- The base case for the recursion are subproblems of size 0 or 1

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It has $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Conquer: recursively sort S_1 and S_2
  - Combine: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)
  Input sequence S with n elements
  Output sequence S sorted according to C
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1)$
    $mergeSort(S_2)$
    $S \leftarrow merge(S_1, S_2)$

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 and implemented by means of a doubly linked list, 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 $\neg A.isEmpty() \land \neg B.isEmpty()$
    if $A.first().element() < B.first().element()$
      $S.addLast(A.remove(A.first()))$
    else
      $S.addLast(B.remove(B.first()))$
  while $\neg A.isEmpty()$
    $S.addLast(A.remove(A.first()))$
  while $\neg B.isEmpty()$
    $S.addLast(B.remove(B.first()))$
  return S

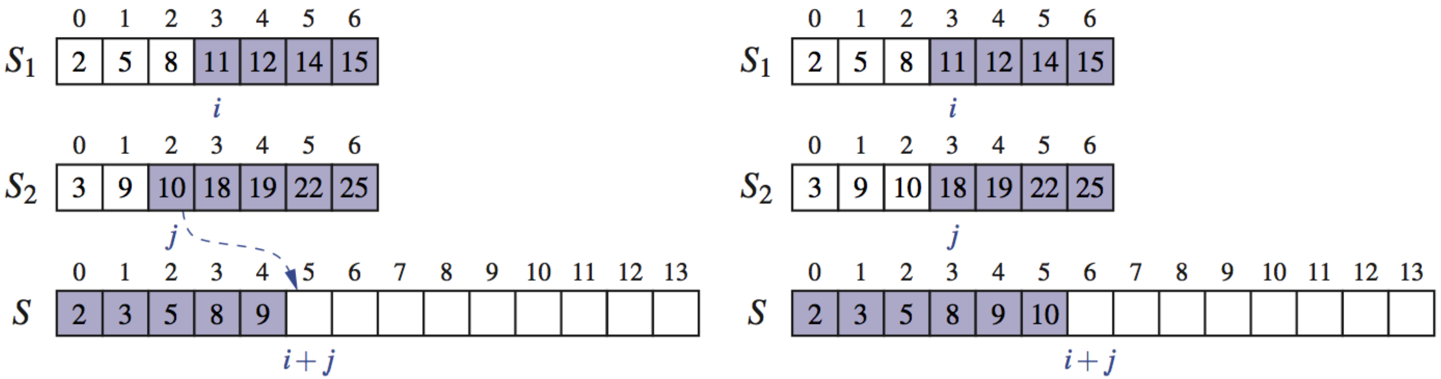
Recap: Generic Merging – set union

- Generalized merge of two sorted lists A and B
- Template method genericMerge
- Auxiliary methods
  - aIsLess => add a
  - bIsLess => add b
  - bothAreEqual => (add b)
- Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

Algorithm *genericMerge*(A, B)
    $S \leftarrow$ empty sequence
    while $\neg A.isEmpty() \wedge \neg B.isEmpty()$
        $a \leftarrow A.first().element()$; $b \leftarrow B.first().element()$
        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 $\neg A.isEmpty()$
        $aIsLess(a, S)$; $A.remove(A.first())$
    while $\neg B.isEmpty()$
        $bIsLess(b, S)$; $B.remove(B.first())$
    return S

Python Merge Implementation

```
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
```

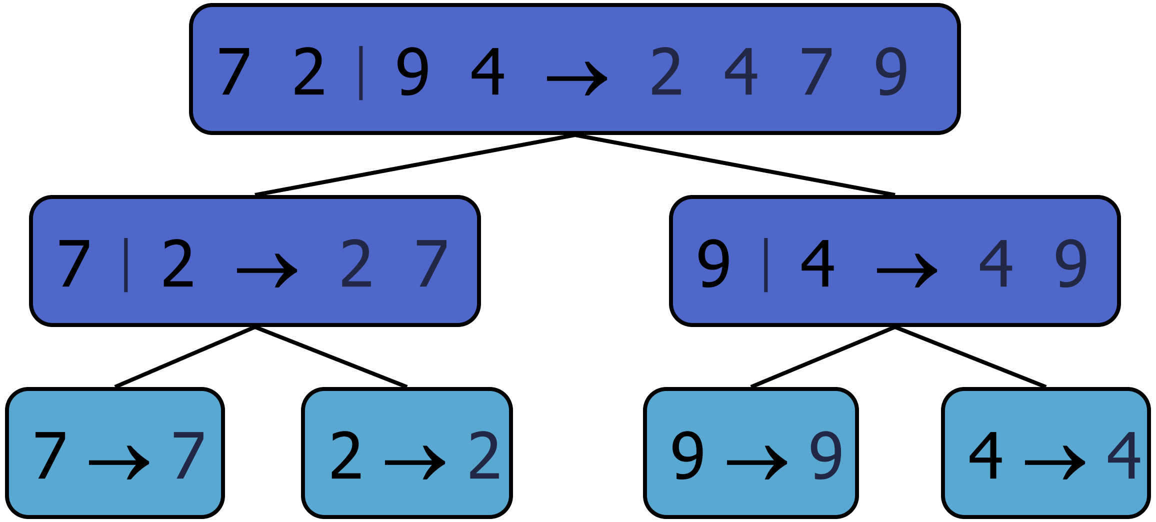


Python Merge-Sort Implementation

```
def merge_sort(S):
  """Sort the elements of Python list S using the merge-sort algorithm."""
  n = len(S)
  if n < 2:
    return                        # list is already sorted
  # divide
  mid = n // 2
  S1 = S[0:mid]                   # copy of first half
  S2 = S[mid:n]                   # copy of second half
  # conquer (with recursion)
  merge_sort(S1)                  # sort copy of first half
  merge_sort(S2)                  # sort copy of second half
  # merge results
  merge(S1, S2, S)                # merge sorted halves back into S
```

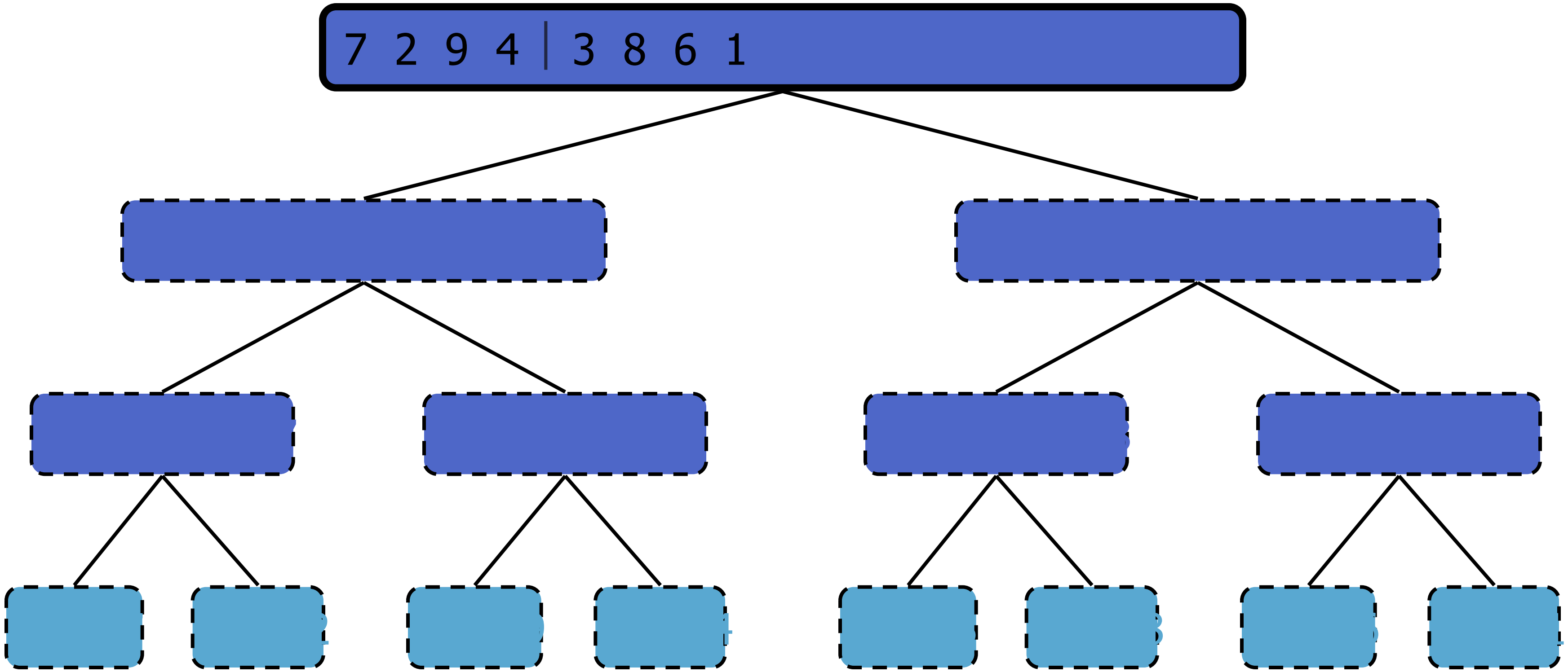
Merge-Sort Tree

- 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 0 or 1



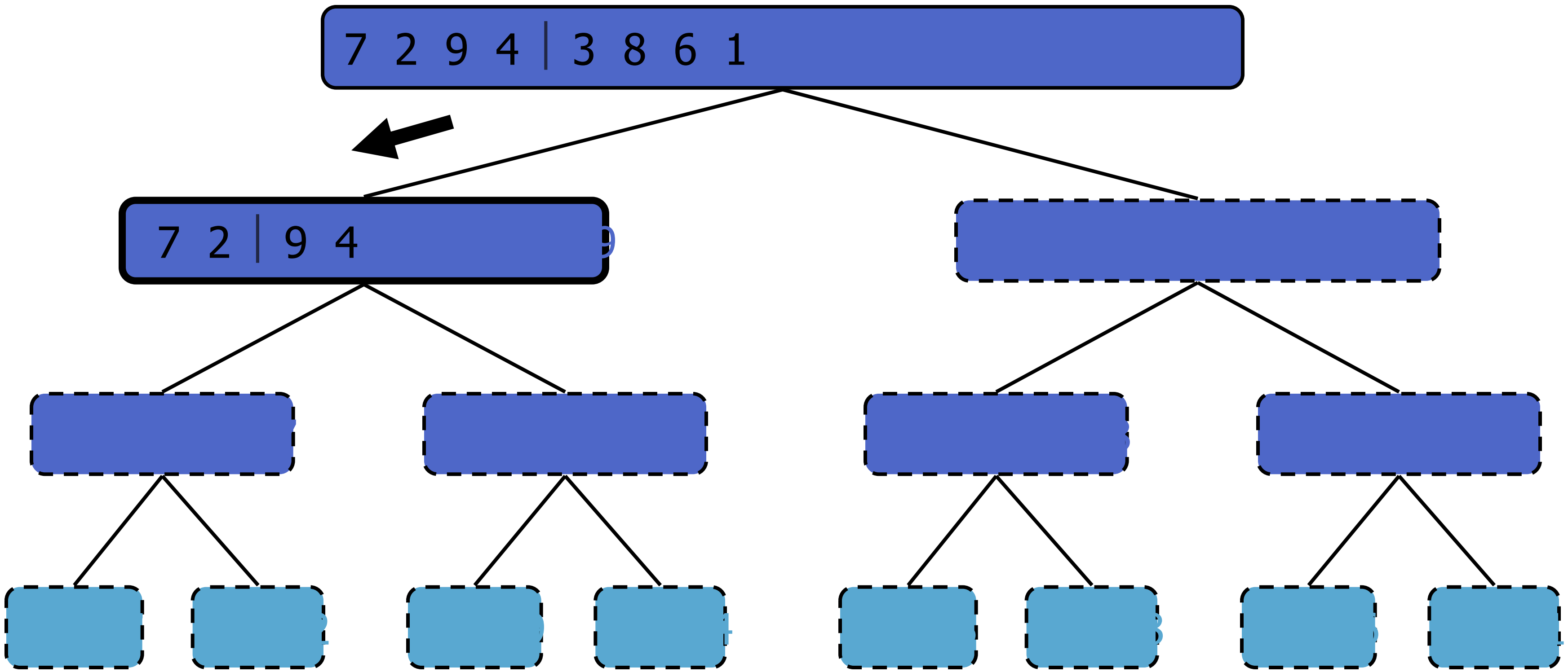
Execution Example

- Partition



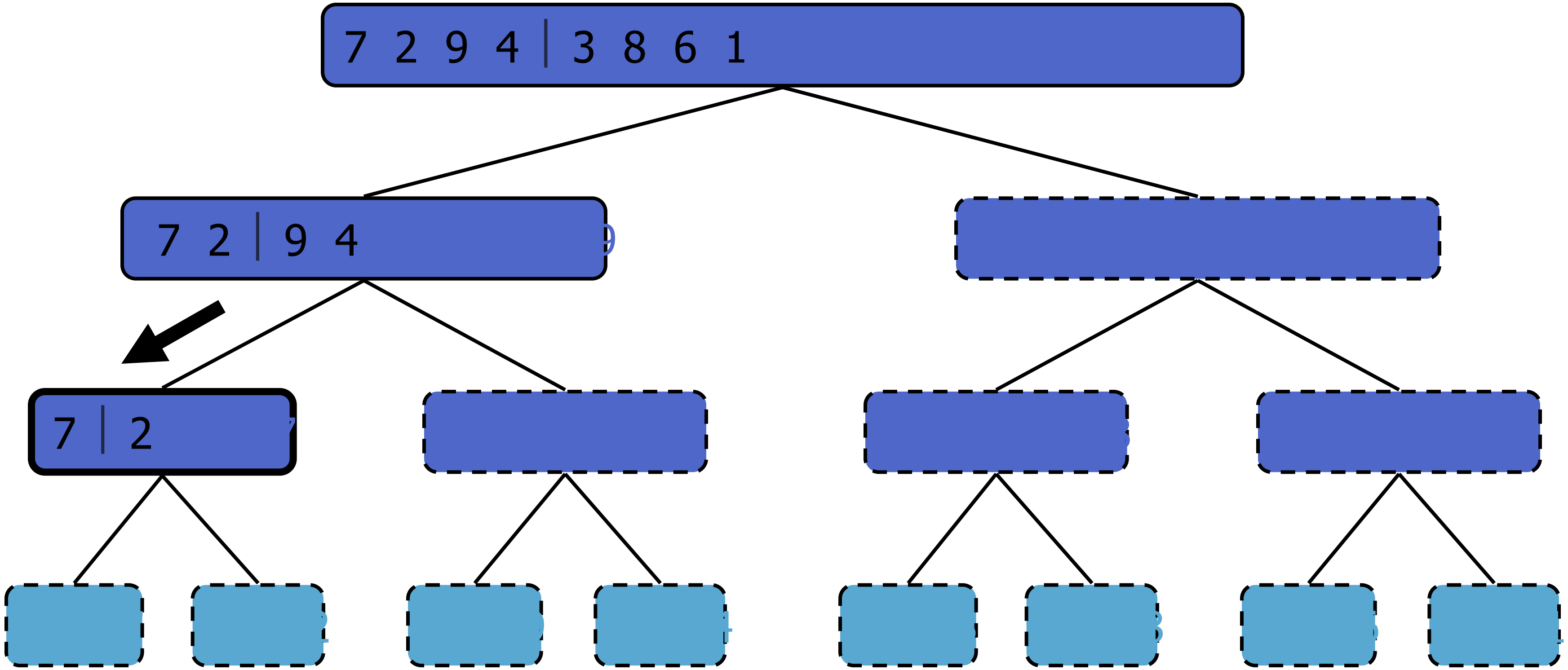
Execution Example (cont.)

- Recursive call, partition



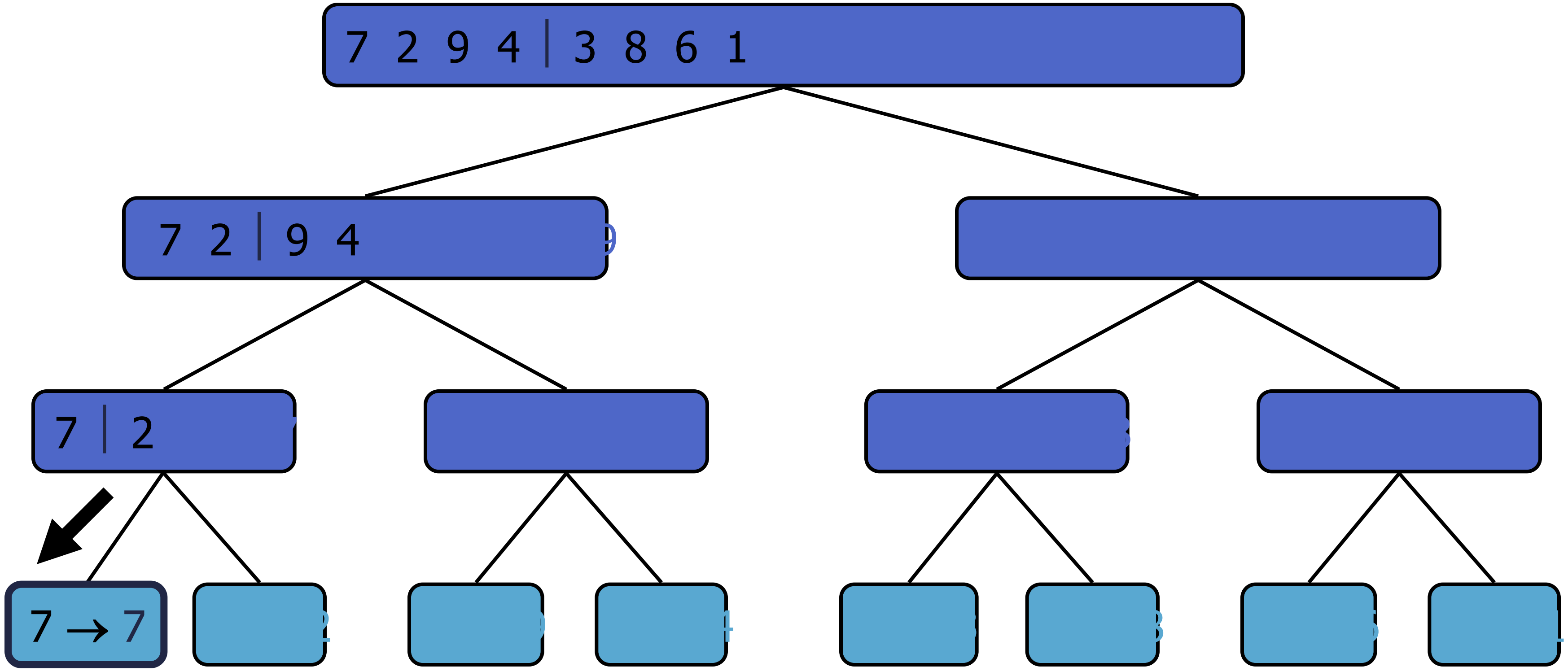
Execution Example (cont.)

- Recursive call, partition



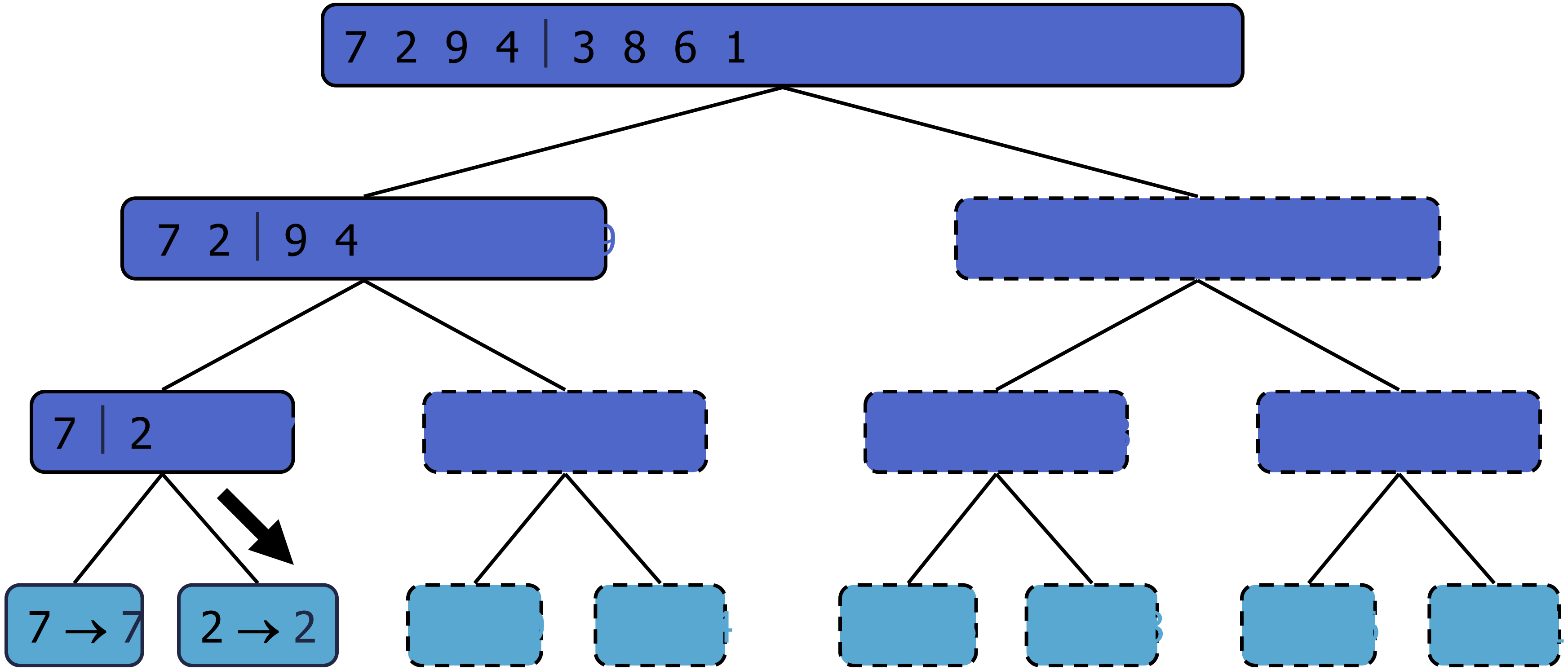
Execution Example (cont.)

- Recursive call, base case



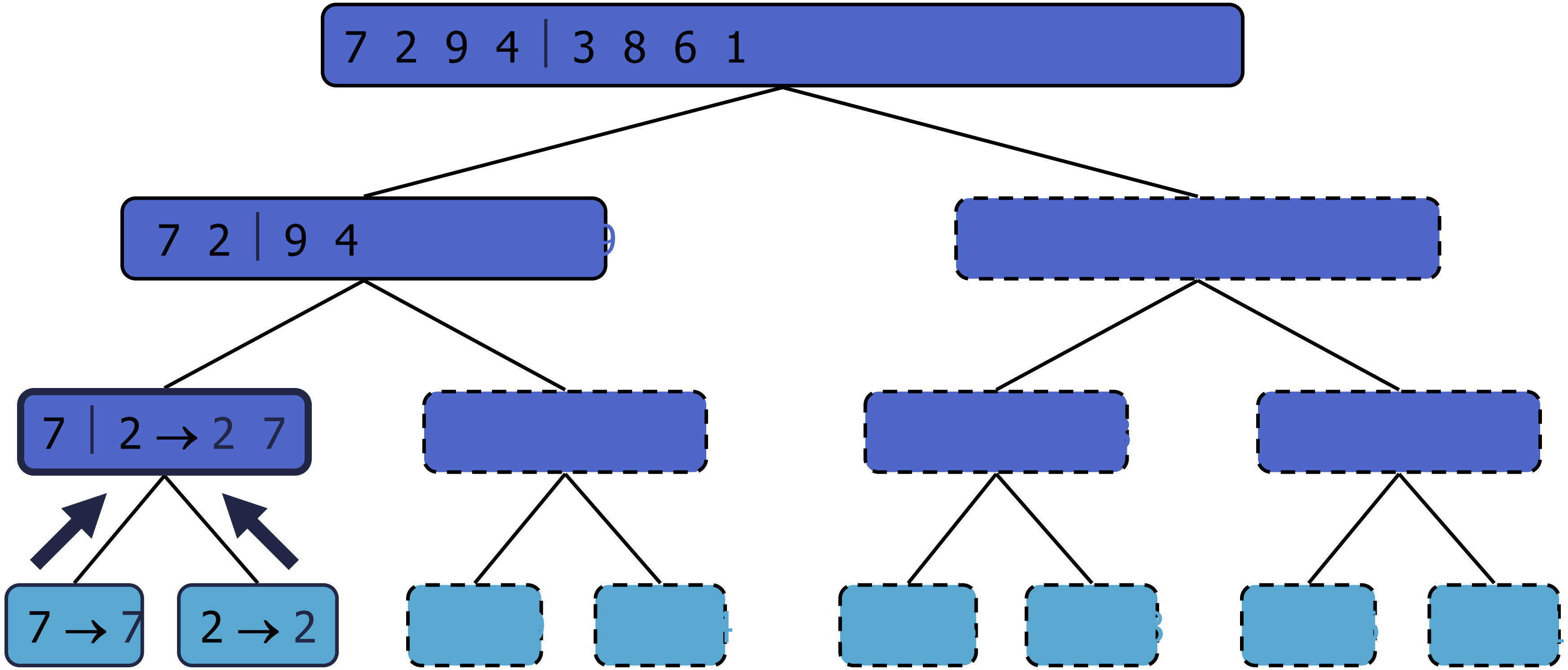
Execution Example (cont.)

- Recursive call, base case



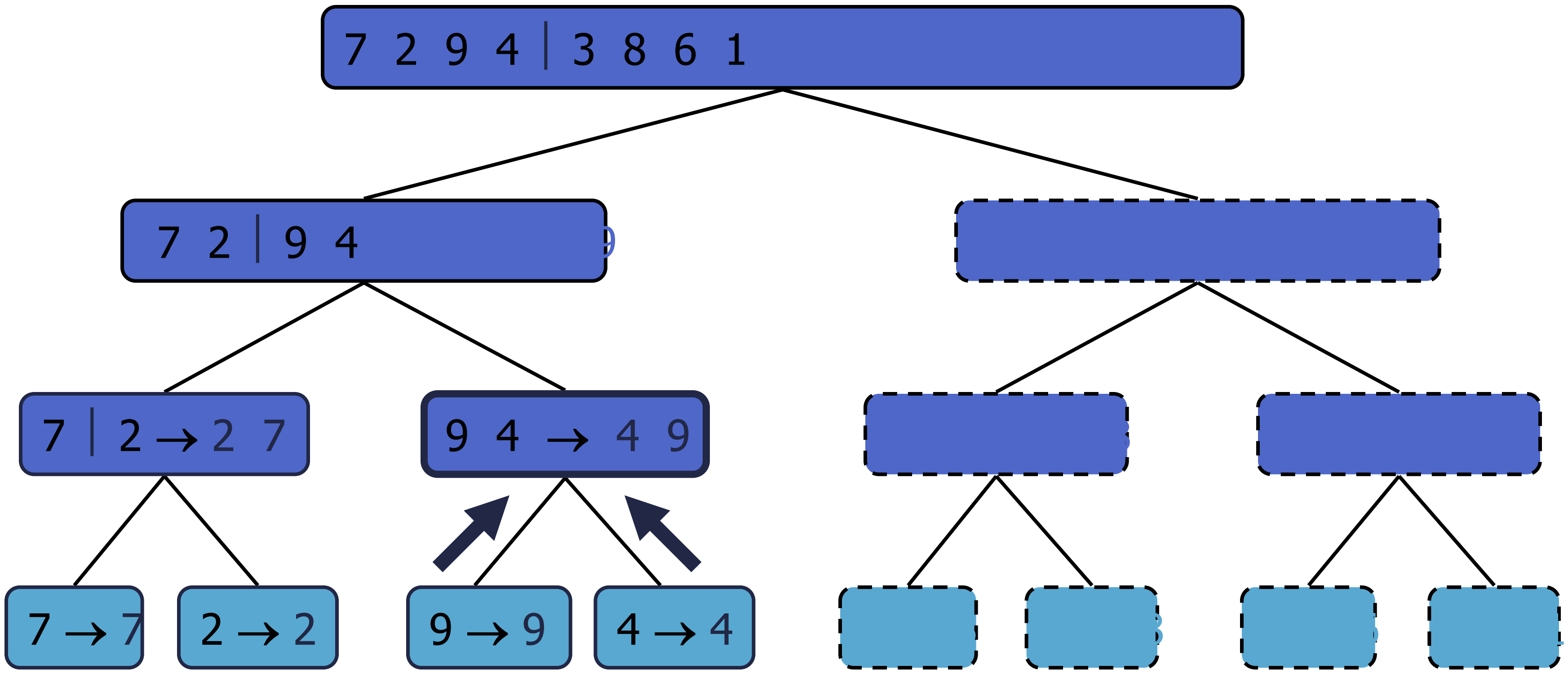
Execution Example (cont.)

- Merge



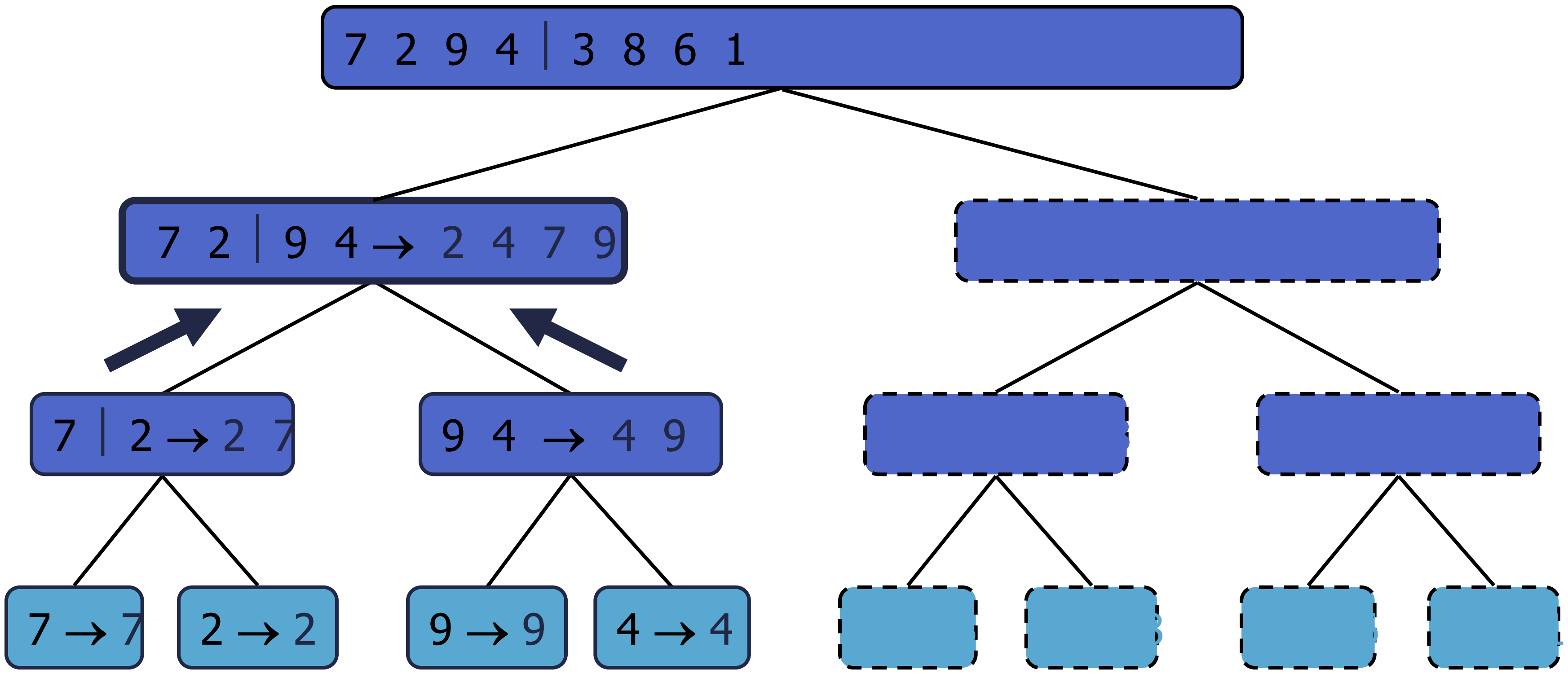
Execution Example (cont.)

- Recursive call, …, base case, merge



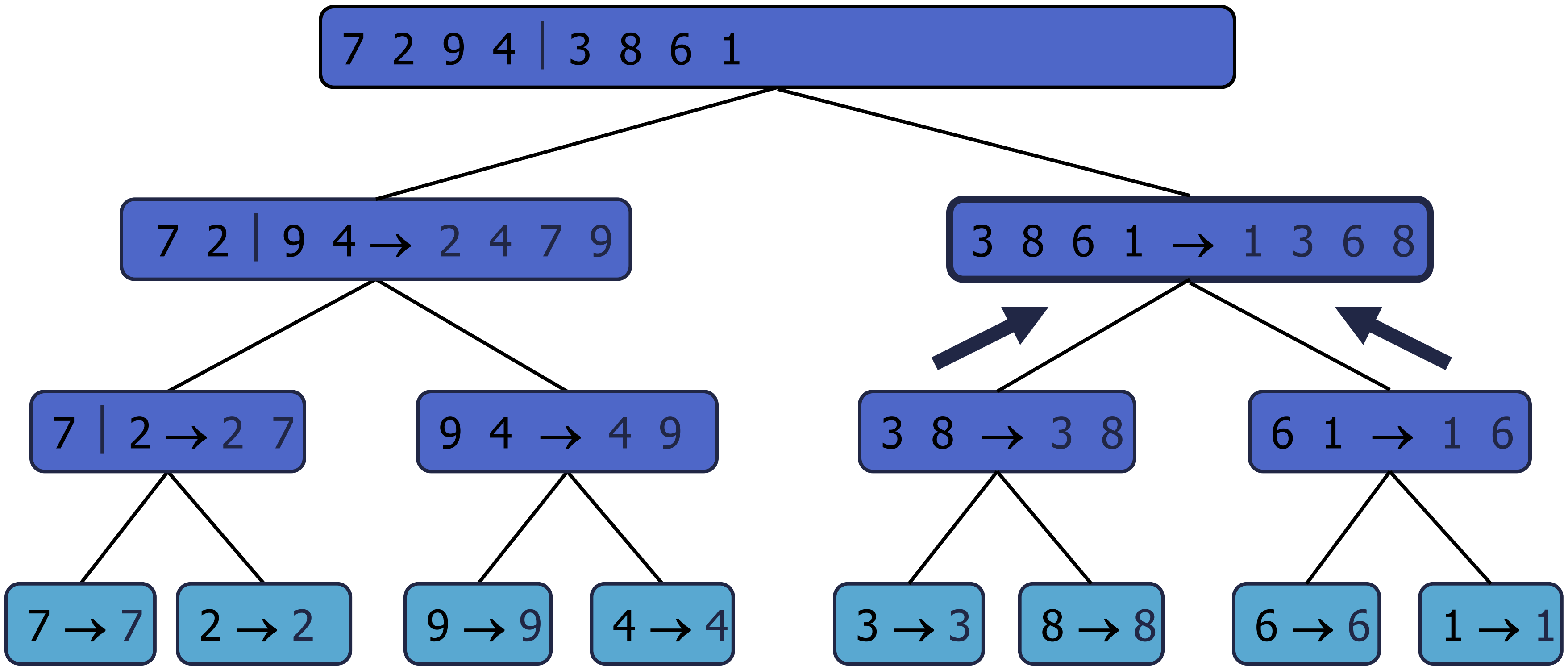
Execution Example (cont.)

- Merge



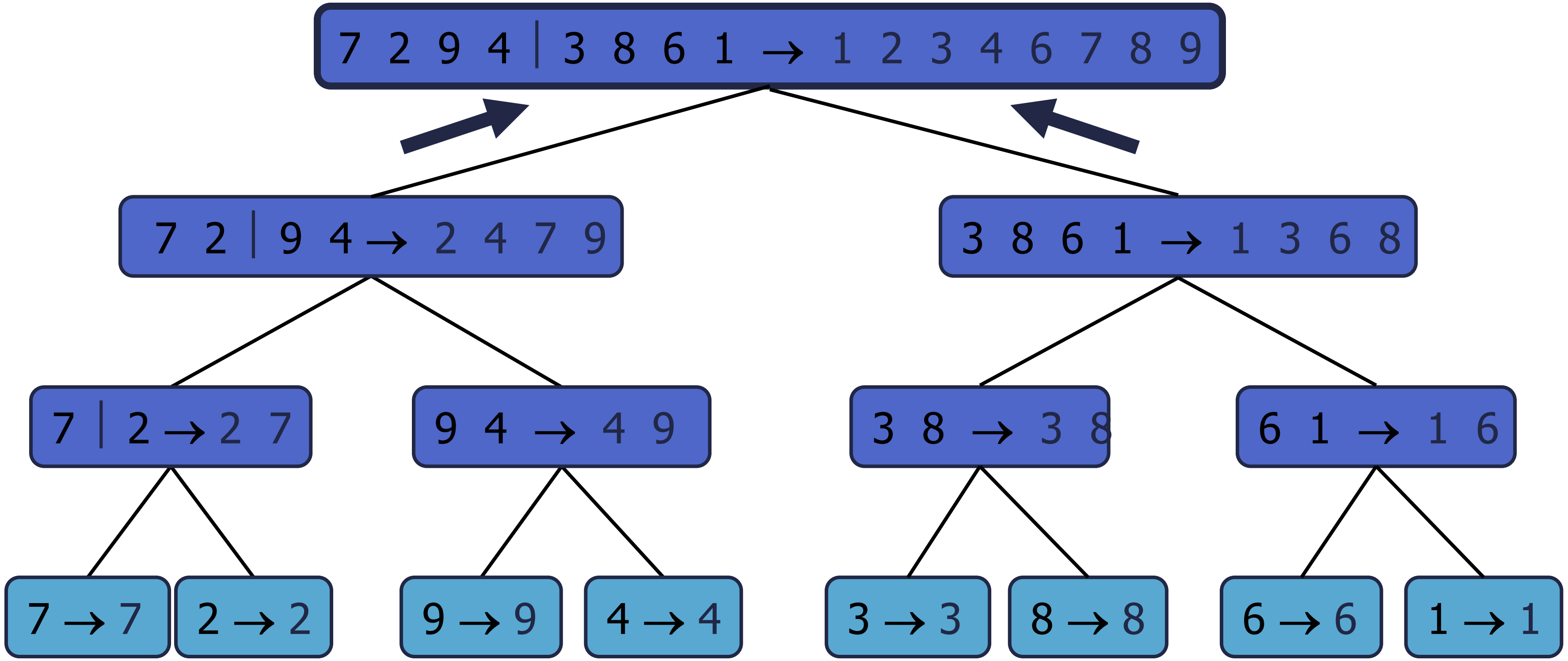
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

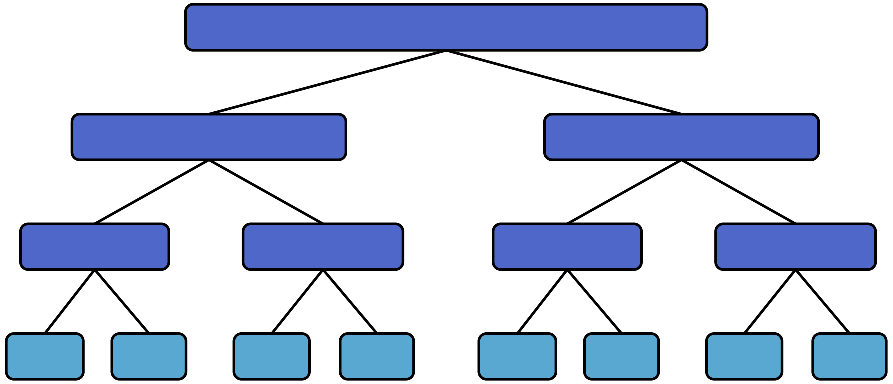
- Merge



Analysis of Merge-Sort

- The height $\boldsymbol{h}$ of the merge-sort tree is $\boldsymbol{O}(\log \boldsymbol{n})$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth $\boldsymbol{i}$ is $\boldsymbol{O}(\boldsymbol{n})$
  - we partition and merge $2^{\boldsymbol{i}}$ sequences of size $\boldsymbol{n}/2^{\boldsymbol{i}}$
  - we make $2^{\boldsymbol{i}+1}$ recursive calls
  - When $\boldsymbol{h} = \log \boldsymbol{n}$, $2^{log\boldsymbol{n}+1} = \boldsymbol{O}(\boldsymbol{n})$
- Thus, the total running time of merge-sort is $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$

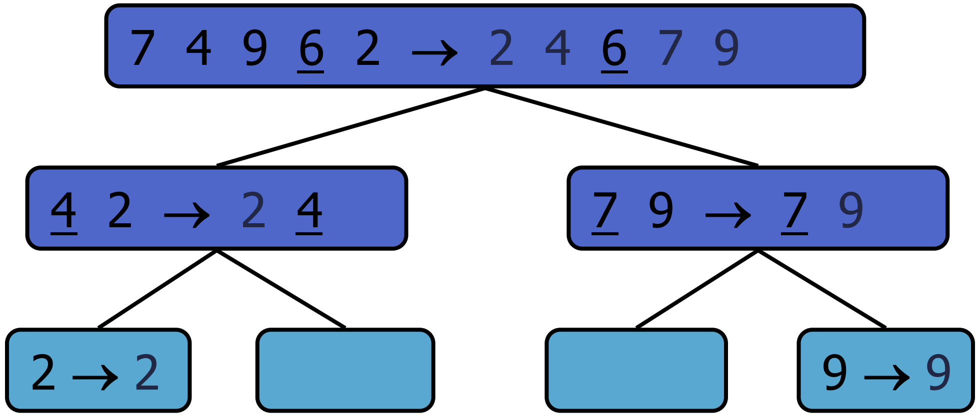
depth	#seqs	size
0	1	$\boldsymbol{n}$
1	2	$\boldsymbol{n}/2$
$\boldsymbol{i}$	$2^{\boldsymbol{i}}$	$\boldsymbol{n}/2^{\boldsymbol{i}}$
…	…	…



Summary of Sorting Algorithms

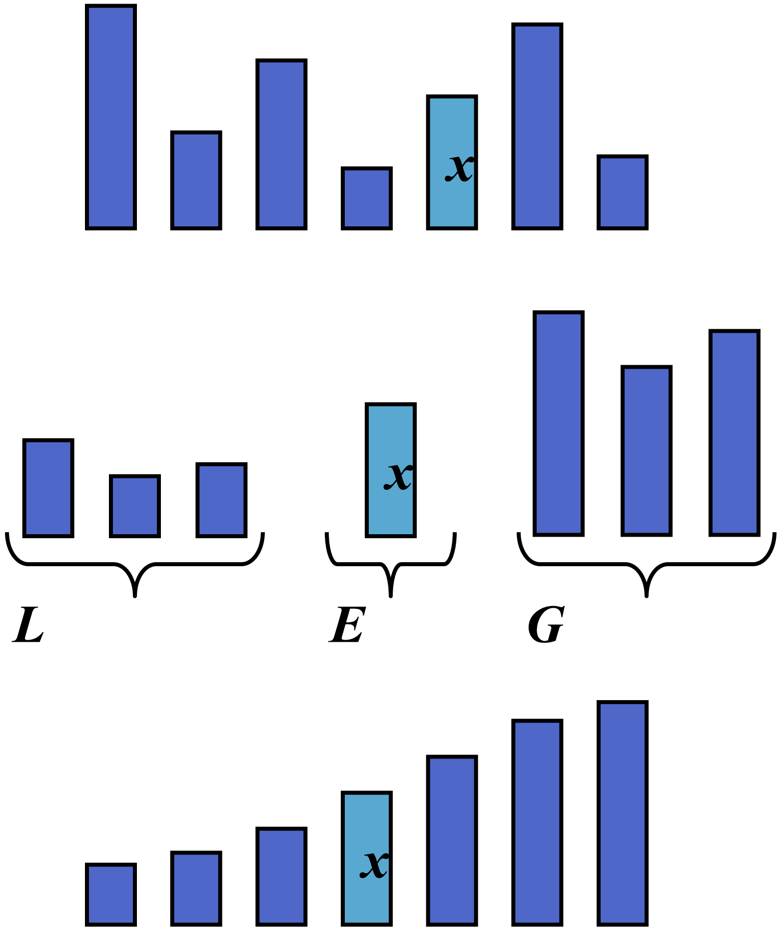
Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	▪ fast ▪ in-place ▪ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	▪ fast ▪ sequential data access ▪ for huge data sets (> 1M)

Quick-Sort

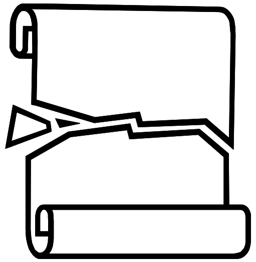


Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Conquer: sort L and G
  - Combine: join L, E and G



Partition



- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$
- Thus, the partition step of quick-sort takes $O(n)$

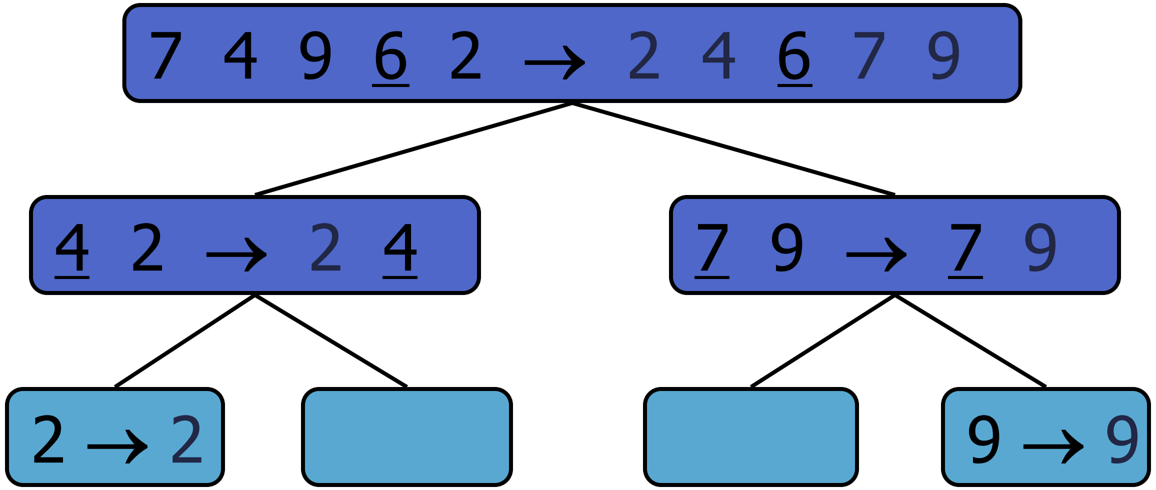
```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.addLast(y)
        else if y = x
            E.addLast(y)
        else { y > x }
            G.addLast(y)
    return L, E, G
```

Python Implementation

```
def quick_sort(S):
  """Sort the elements of queue S using the quick-sort algorithm."""
  n = len(S)
  if n < 2:
    return                              # list is already sorted
  # divide
  p = S.first( )                        # using first as arbitrary pivot
  L = LinkedQueue( )
  E = LinkedQueue( )
  G = LinkedQueue( )
  while not S.is_empty( ):              # divide S into L, E, and G
    if S.first( ) < p:
      L.enqueue(S.dequeue( ))
    elif p < S.first( ):
      G.enqueue(S.dequeue( ))
    else:                               # S.first() must equal pivot
      E.enqueue(S.dequeue( ))
  # conquer (with recursion)
  quick_sort(L)                         # sort elements less than p
  quick_sort(G)                         # sort elements greater than p
  # concatenate results
  while not L.is_empty( ):
    S.enqueue(L.dequeue( ))
  while not E.is_empty( ):
    S.enqueue(E.dequeue( ))
  while not G.is_empty( ):
    S.enqueue(G.dequeue( ))
```

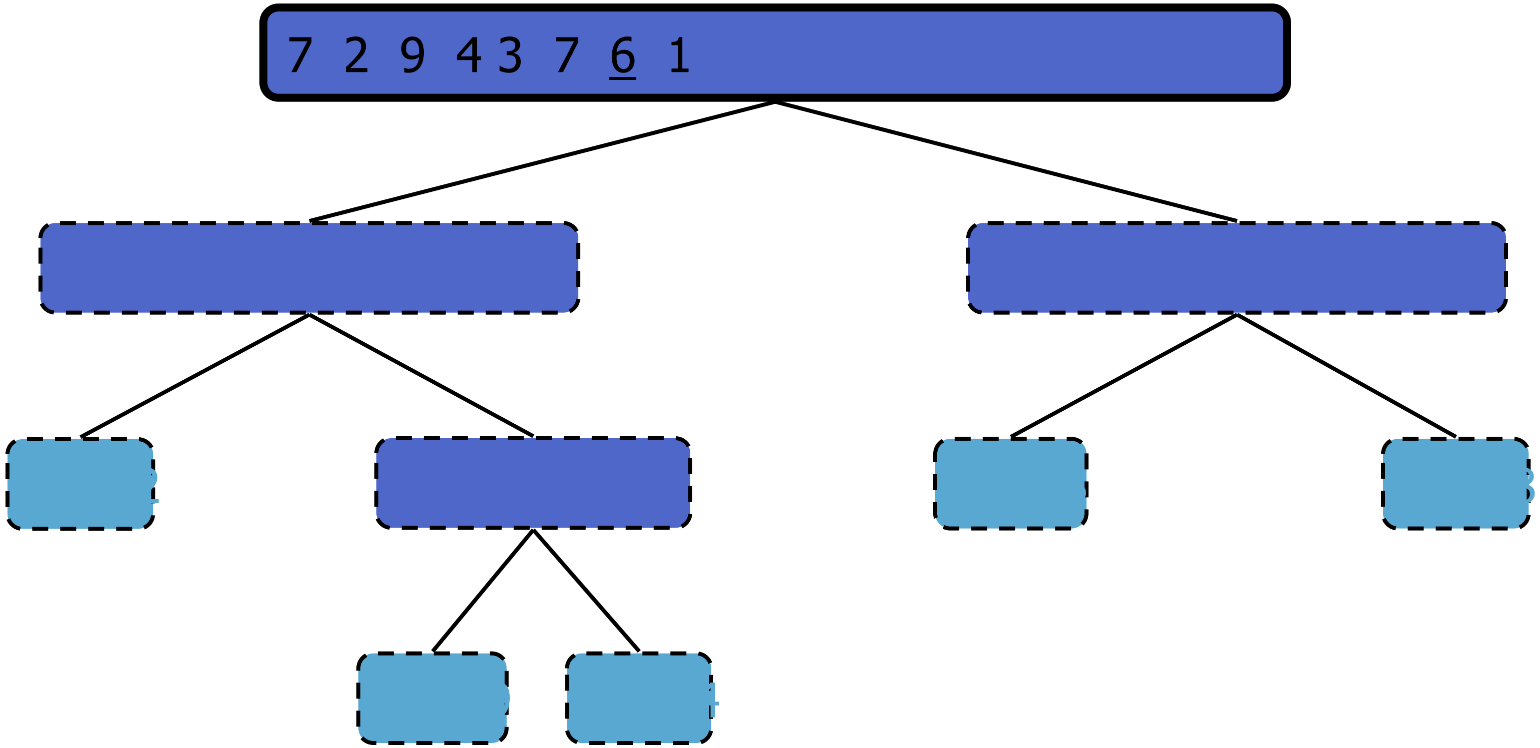
Quick-Sort Tree

- 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



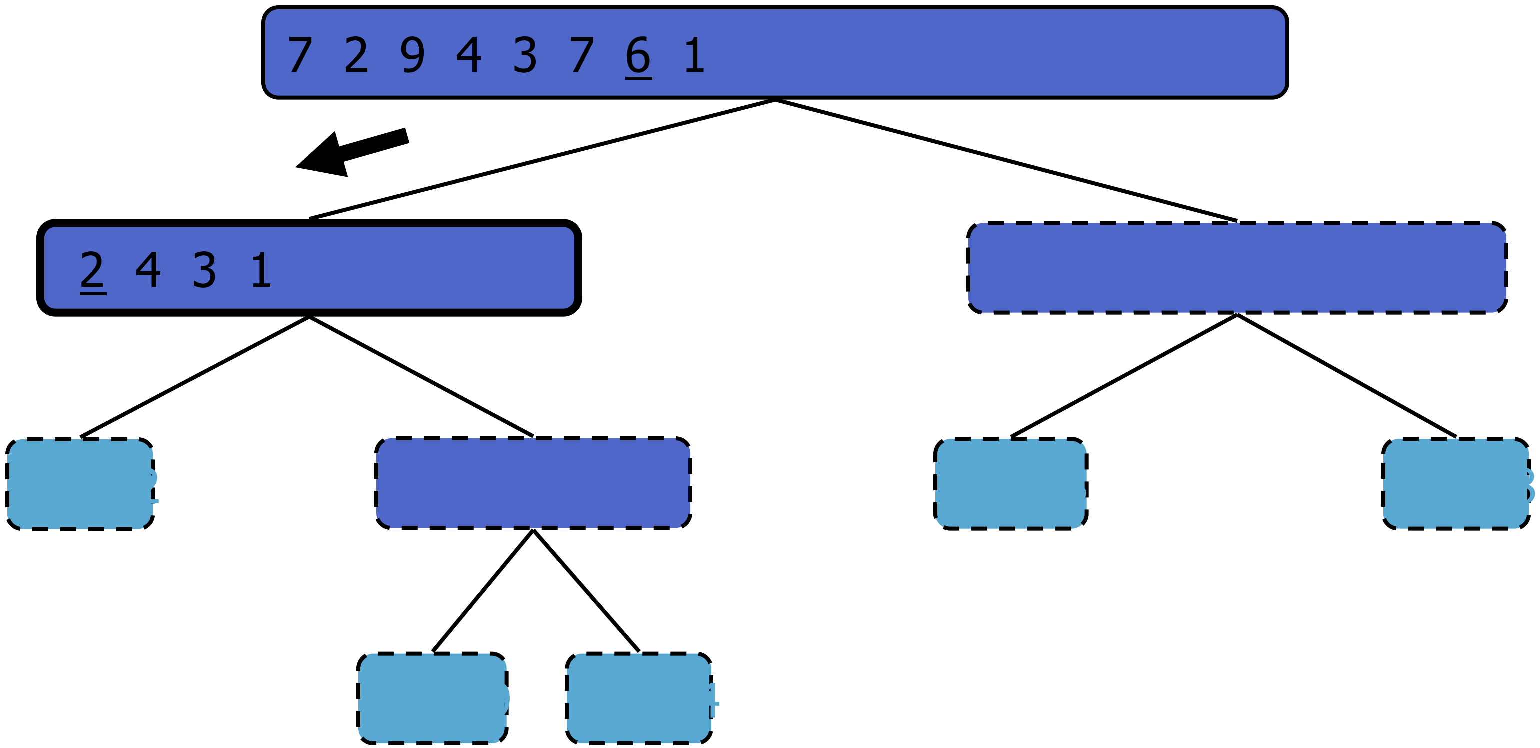
Execution Example

- Pivot selection



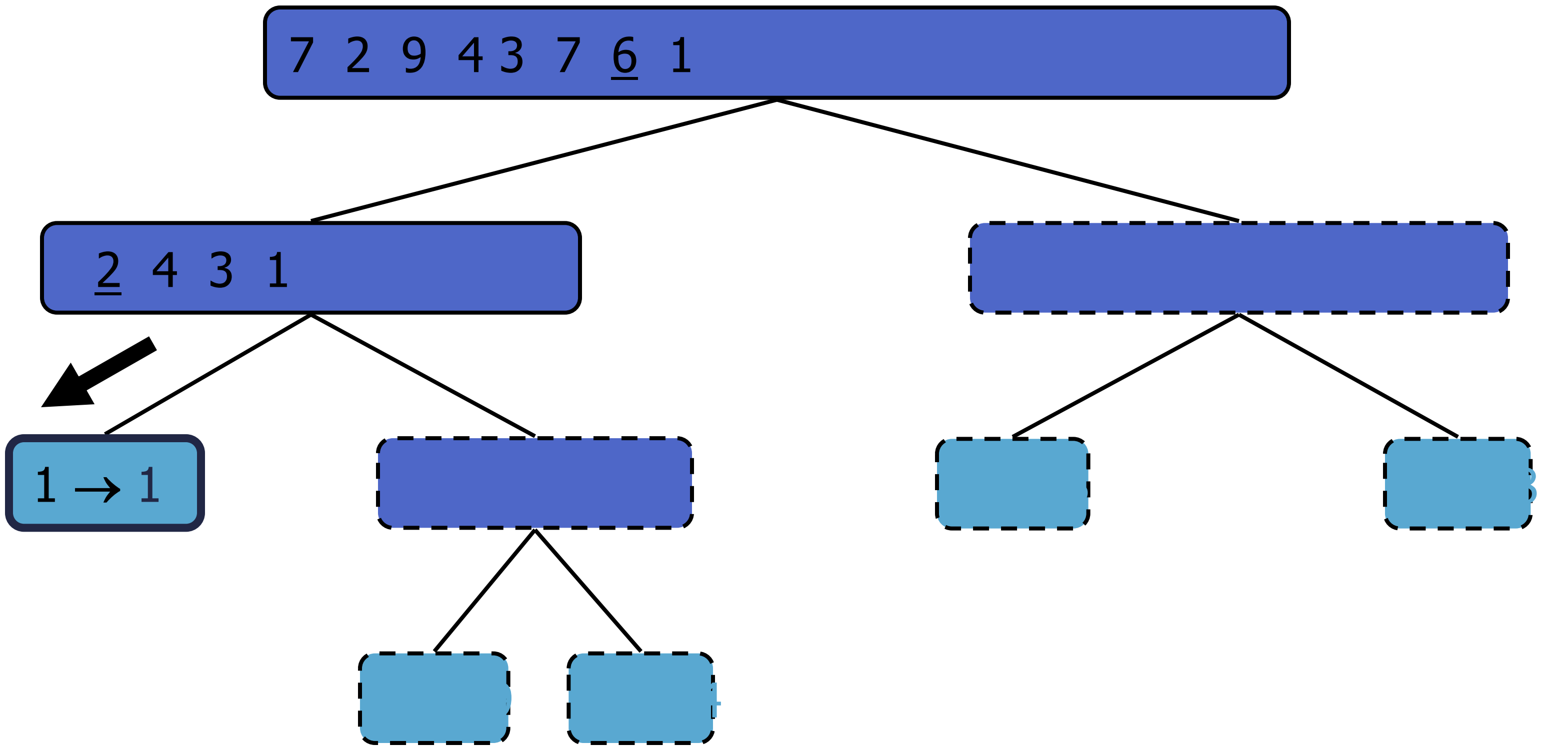
Execution Example (cont.)

- Partition, recursive call, pivot selection



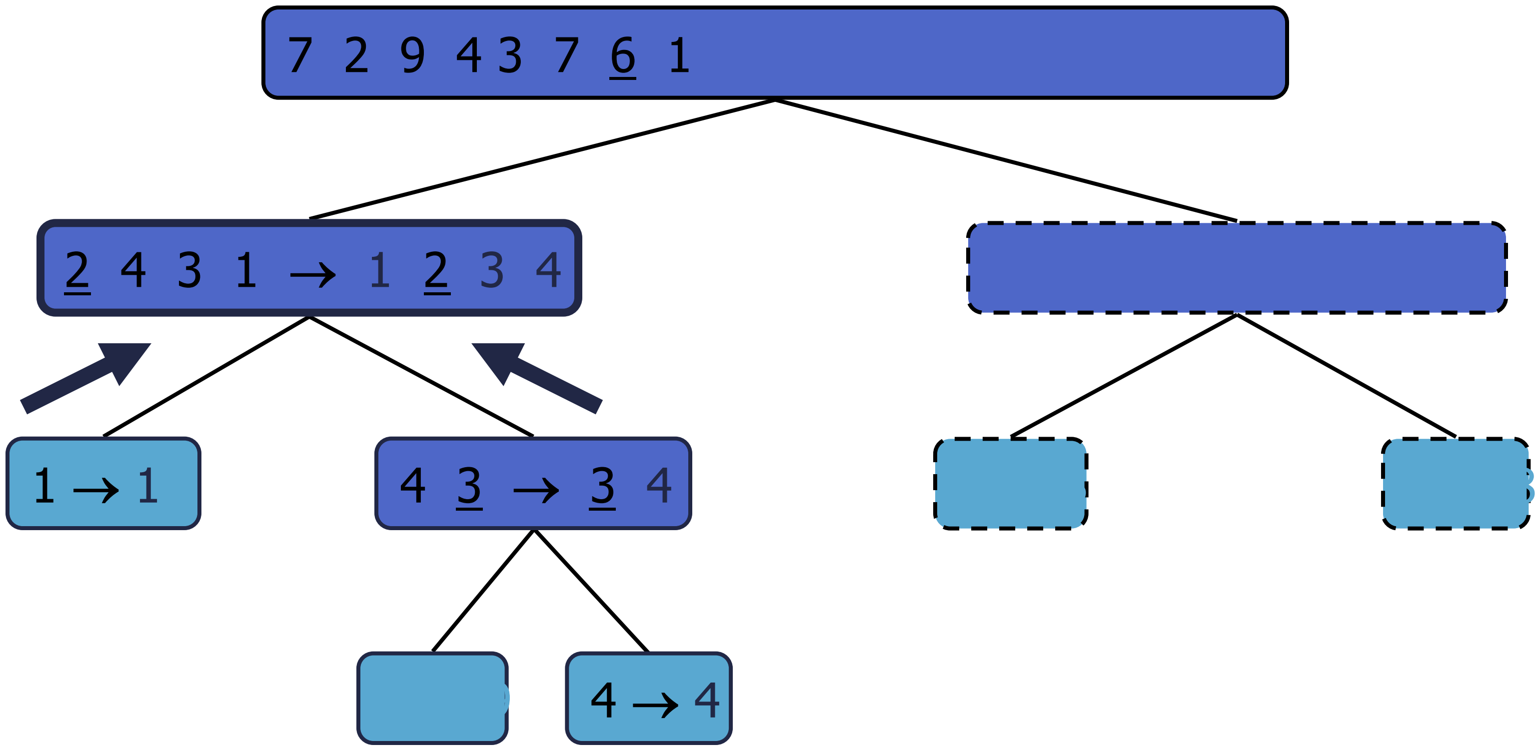
Execution Example (cont.)

- Partition, recursive call, base case



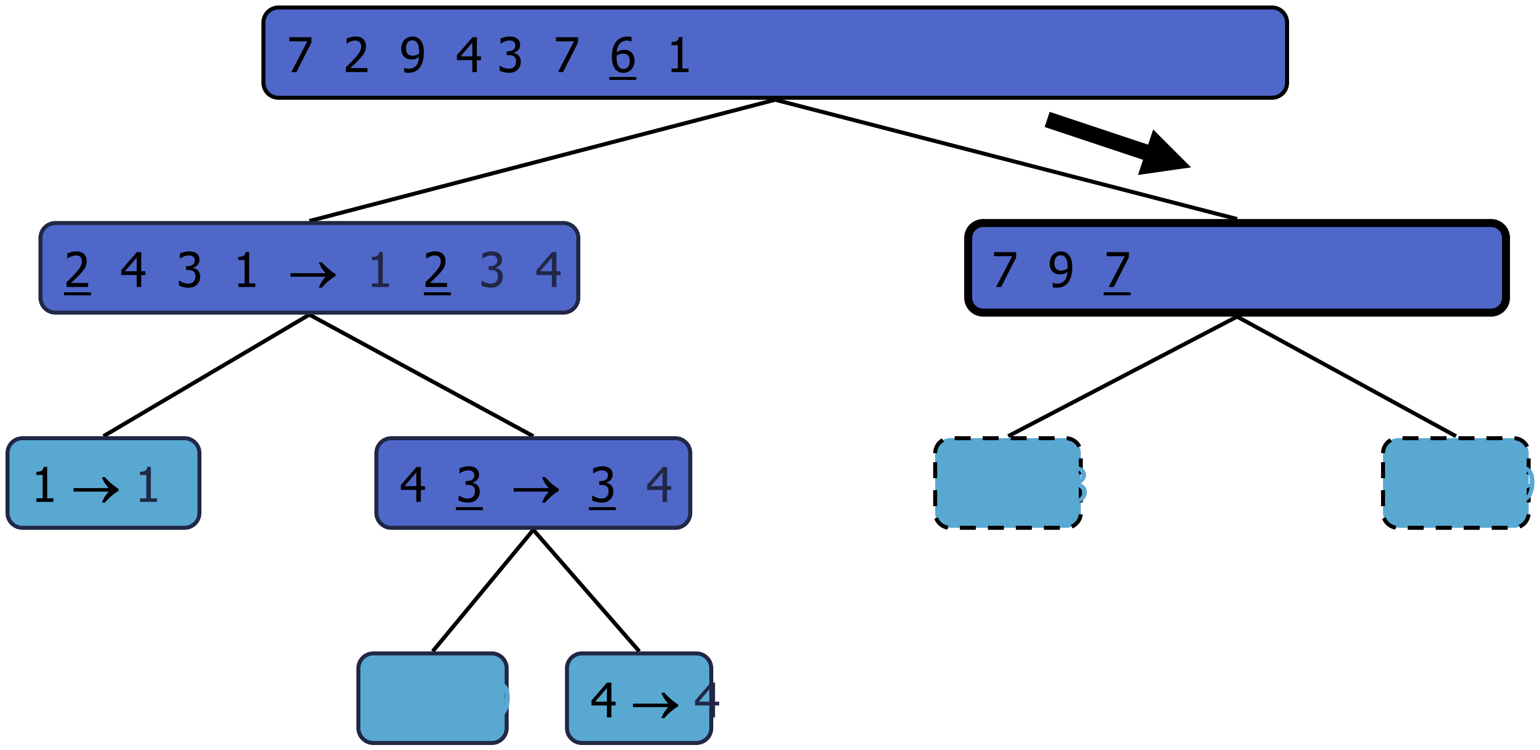
Execution Example (cont.)

- Recursive call, …, base case, join



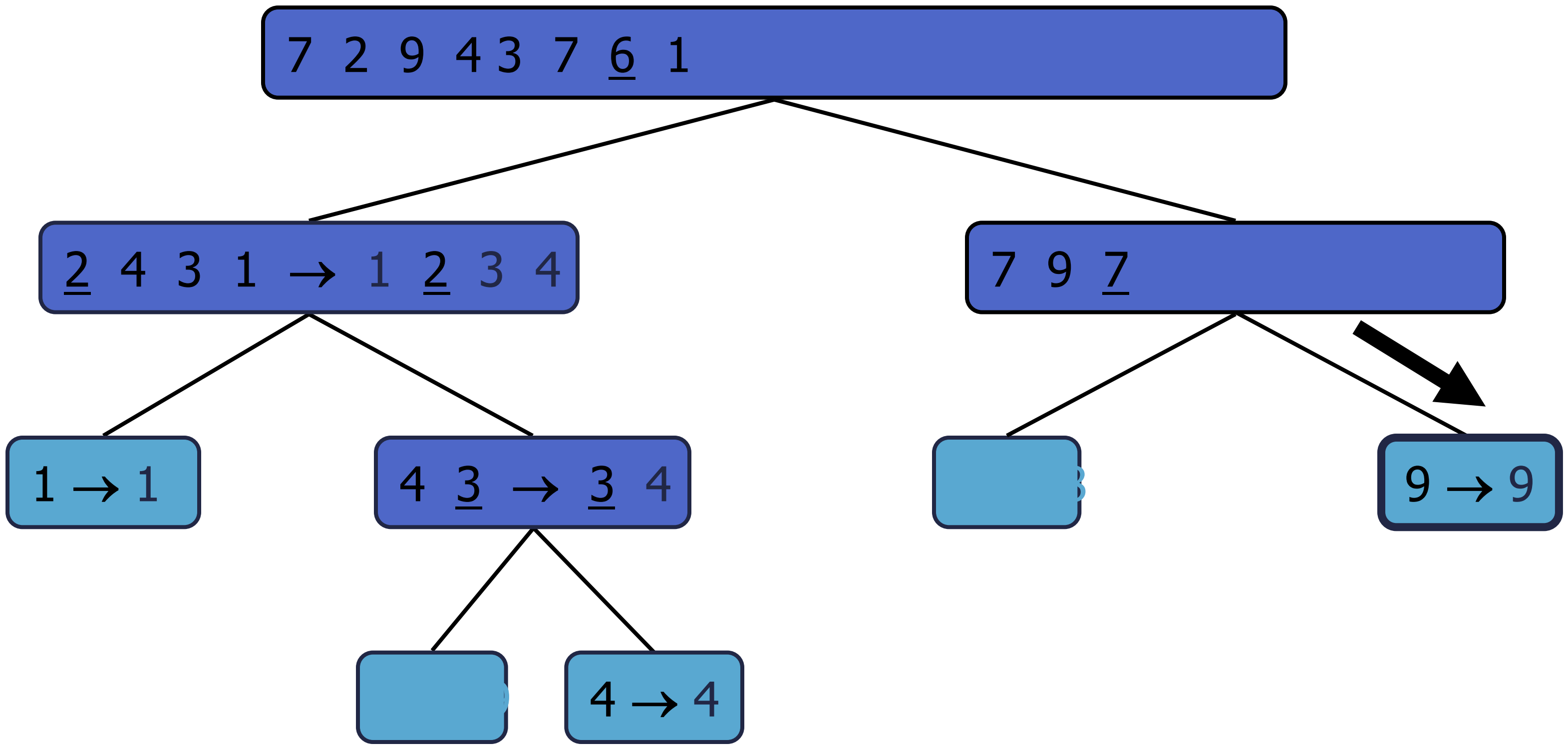
Execution Example (cont.)

- Recursive call, pivot selection



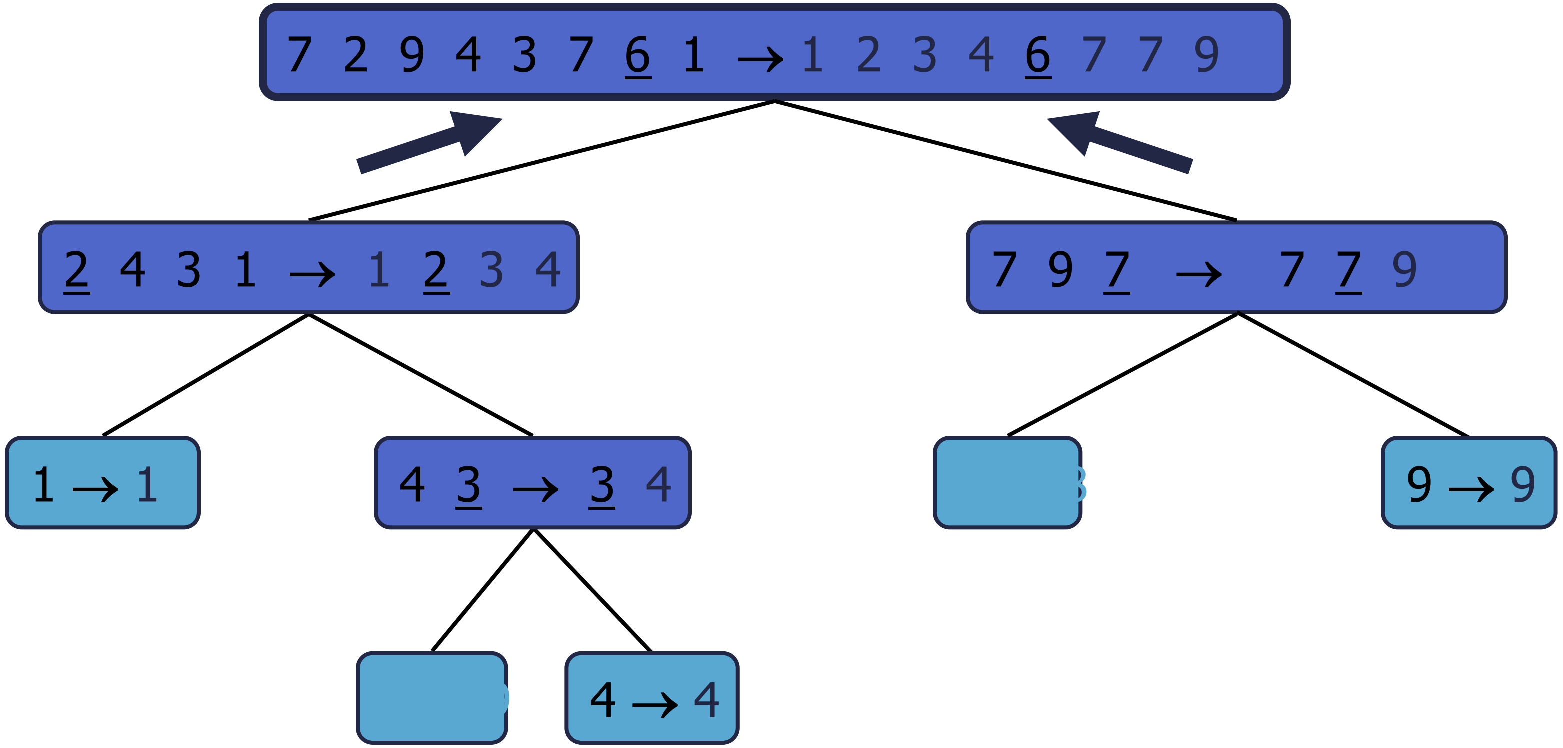
Execution Example (cont.)

- Partition, ..., recursive call, base case



Execution Example (cont.)

- Join, join



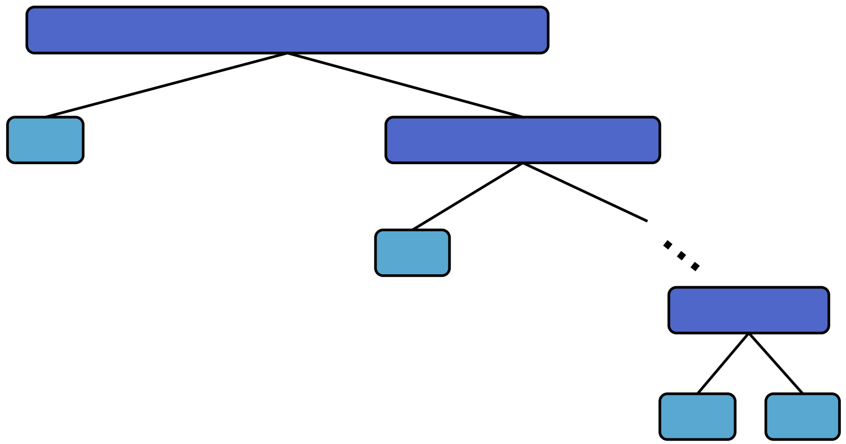
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of $\boldsymbol{L}$ and $\boldsymbol{G}$ has size $\boldsymbol{n} - 1$ and the other has size 0
- The running time is proportional to the sum

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

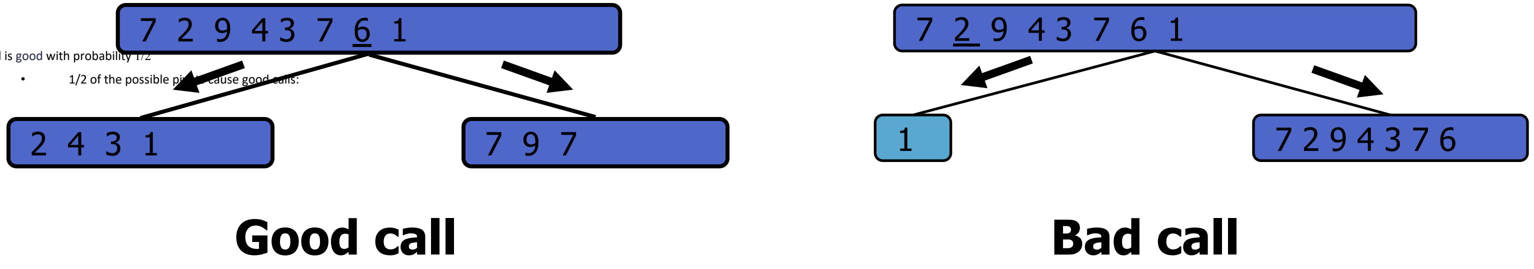
- Thus, the worst-case running time of quick-sort is $\boldsymbol{O}(\boldsymbol{n}^2)$

depth	time
0	$\boldsymbol{n}$
1	$\boldsymbol{n} - 1$
…	…
$\boldsymbol{n} - 1$	1

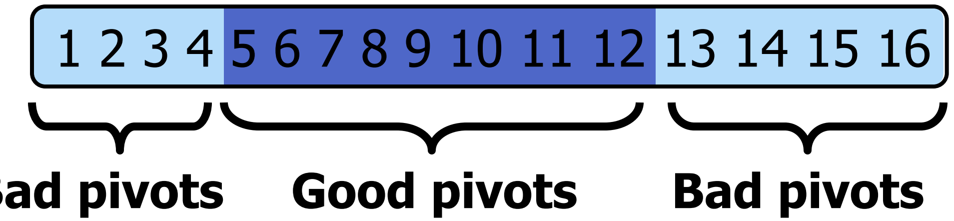


Expected Running Time

- 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 $3s/4$



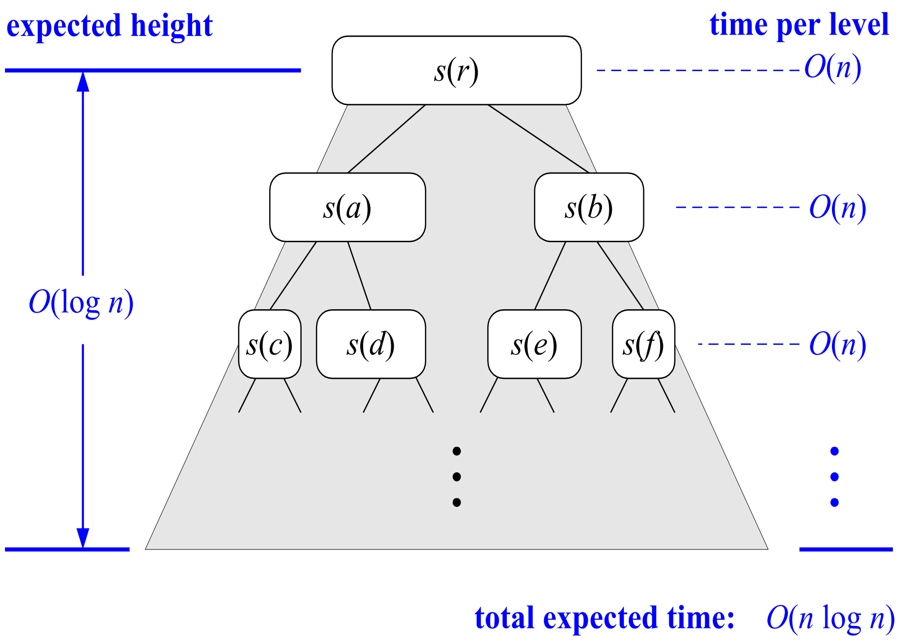
- A call is good with probability 1/2
  - 1/2 of the possible pivots cause good calls:



Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get $\boldsymbol{k}$ heads is $2\boldsymbol{k}$
- For a node of depth $\boldsymbol{i}$, we expect
  - $\boldsymbol{i}/2$ ancestors are good calls (divide the sequence better than 1/4 & 3/4)
  - The size of the input sequence for the current call is at most $(3/4)^{\boldsymbol{i}/2}\boldsymbol{n}$

- Therefore, we have
  - For a node of depth $2\log_{4/3}\boldsymbol{n}$, the expected input size is one
  - The expected height of the quick-sort tree is $\boldsymbol{O}(\log \boldsymbol{n})$
- The amount or work done at the nodes of the same depth is $\boldsymbol{O}(\boldsymbol{n})$
- Thus, the expected running time of quick-sort is $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$



In-Place Quick-Sort

- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than h
  - the elements equal to the pivot have rank between h and k
  - the elements greater than the pivot have rank greater than k
- The recursive calls consider
  - elements with rank less than h
  - elements with rank greater than 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
    $x \leftarrow S.elemAtRank(i)$
    $(h, k) \leftarrow inPlacePartition(x)$
    $inPlaceQuickSort(S, l, h - 1)$
    $inPlaceQuickSort(S, k + 1, r)$

In-Place Partitioning

- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element ≥ x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k

→ j k ←

3 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 **6** 9

Python Implementation

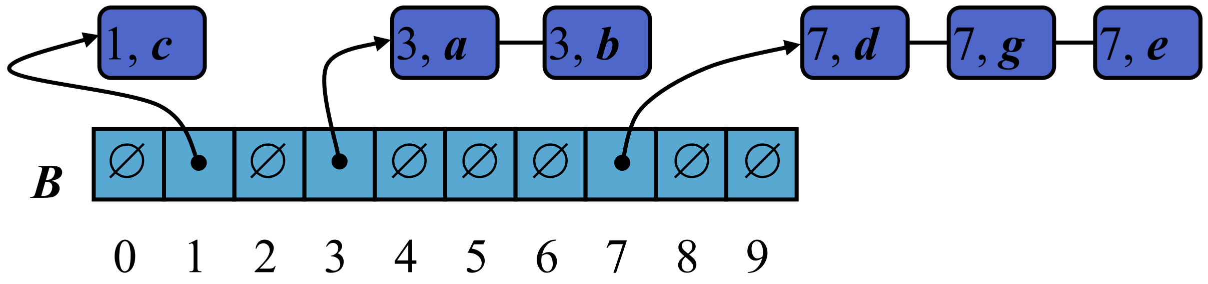
```python
def inplace_quick_sort(S, a, b):
  """Sort the list from S[a] to S[b] inclusive using the quick-sort algorithm."""
  if a >= b: return                                # range is trivially sorted
  pivot = S[b]                                     # last element of range is pivot
  left = a                                         # 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)
```

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)

Bucket-Sort and Radix-Sort



Bucket-Sort

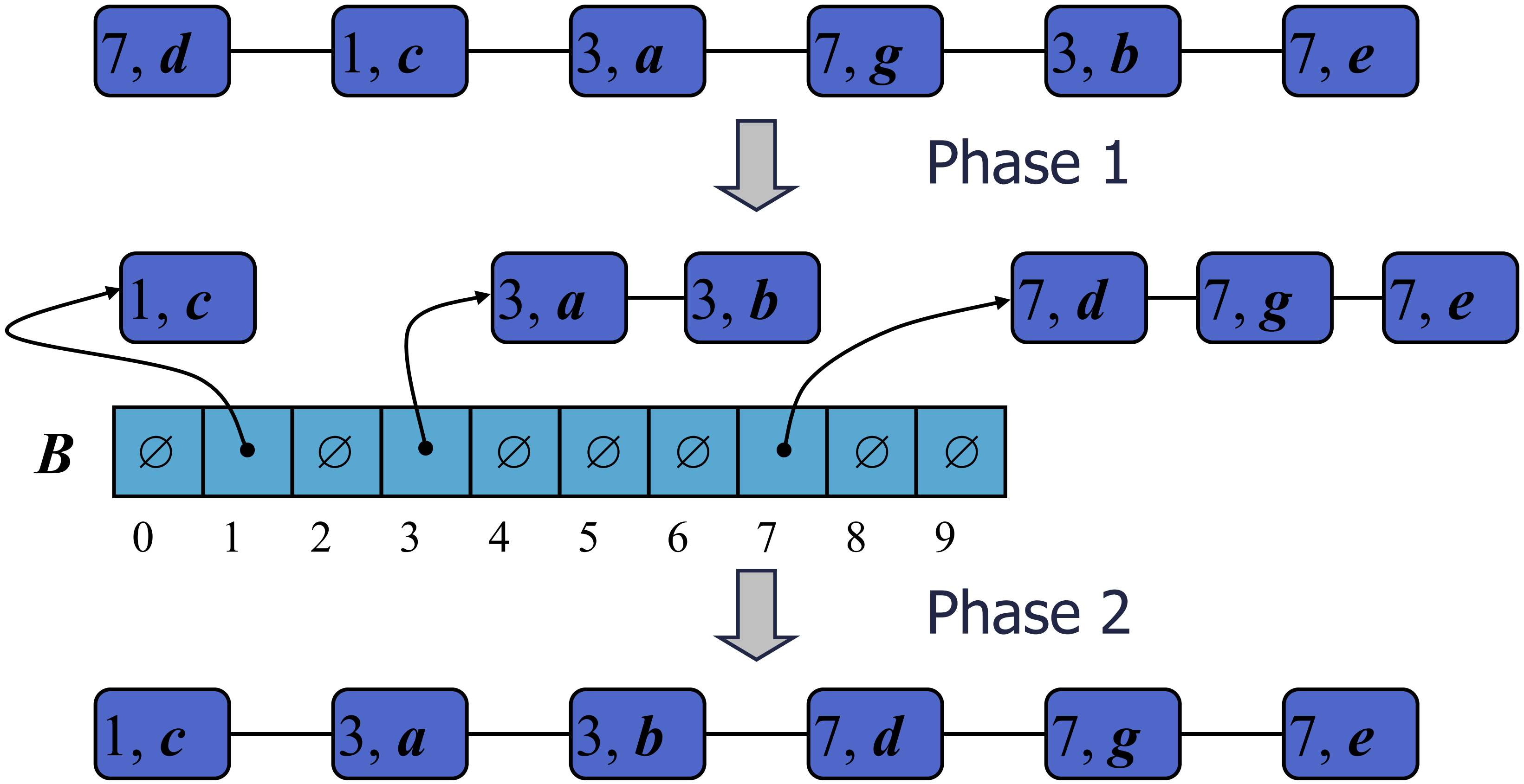
- Let be S be a sequence of n (key, element) items with keys in the range $[0, N-1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
  - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$, move the entries of bucket $B[i]$ to the end of sequence S
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n+N)$ time

  Bucket-sort takes $O(n+N)$ time

Algorithm bucketSort(S):
Input: Sequence S of entries with integer keys in the range [0, N − 1]
Output: Sequence S sorted in nondecreasing order of the keys
let B be an array of N sequences, each of which is initially empty
for each entry e in S **do**
  k = the key of e
  remove e from S
  insert e at the end of bucket B[k]
for i = 0 to N−1 **do**
  for each entry e in B[i] **do**
    remove e from B[i]
    insert e at the end of S

Example

- Key range [0, 9]



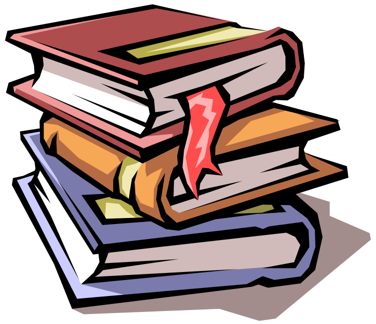
Properties and Extensions

- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator
- Stable Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm

Extensions

- Integer keys in the range $[a, b]$
  - Put entry (k, o) into bucket $B[k - a]$
- String keys from a set D of possible strings, where D has constant size (e.g., names of the 50 U.S. states)
  - Sort D and compute the rank $r(k)$ of each string k of D in the sorted sequence
  - Put entry (k, o) into bucket $B[r(k)]$

Lexicographic Order



- A $\boldsymbol{d}$-tuple is a sequence of $\boldsymbol{d}$ keys $(\boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_d)$, where key $\boldsymbol{k}_i$ is said to be the $\boldsymbol{i}$-th dimension of the tuple
- Example:
  - The Cartesian coordinates of a point in space are a 3-tuple
- The lexicographic order of two $\boldsymbol{d}$-tuples is recursively defined as follows

$$(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_d) < (\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_d)$$

$$\Leftrightarrow$$

$$\boldsymbol{x}_1 < \boldsymbol{y}_1 \ \vee \ \boldsymbol{x}_1 = \boldsymbol{y}_1 \wedge (\boldsymbol{x}_2, \ldots, \boldsymbol{x}_d) < (\boldsymbol{y}_2, \ldots, \boldsymbol{y}_d)$$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

Lexicographic-Sort

- Let C_i be the comparator that compares two tuples by their i-th dimension
- Let $stableSort(S, C)$ be a stable sorting algorithm that uses comparator C
- Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm *stableSort*, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of *stableSort*

Algorithm *lexicographicSort*(S)
  Input sequence S of d-tuples
  Output sequence S sorted in lexicographic order

  for $i \leftarrow d$ **downto** 1
    $stableSort(S, C_i)$

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

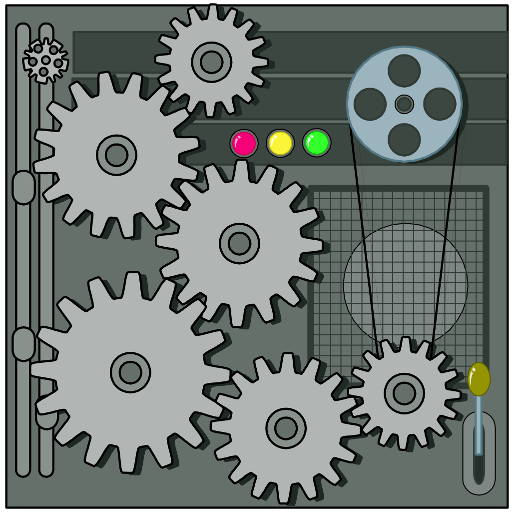
(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

Radix-Sort

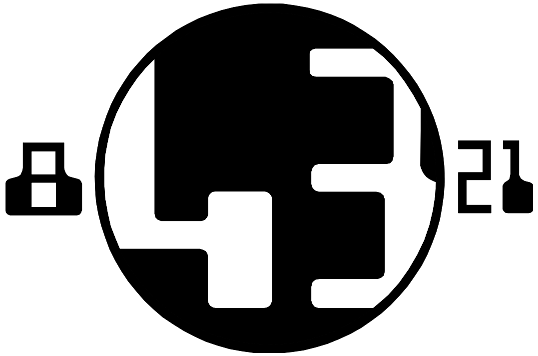
- Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range $[0, N-1]$
- Radix-sort runs in time $O(d(n+N))$



Algorithm *radixSort*(S, N)
    Input sequence S of d-tuples such that $(0, \dots, 0) \le (x_1, \dots, x_d)$ and $(x_1, \dots, x_d) \le (N-1, \dots, N-1)$ for each tuple $(x_1, \dots, x_d)$ in S
    Output sequence S sorted in lexicographic order

    for $i \leftarrow d$ **downto** 1
        bucketSort(S, N)

Radix-Sort for Binary Numbers

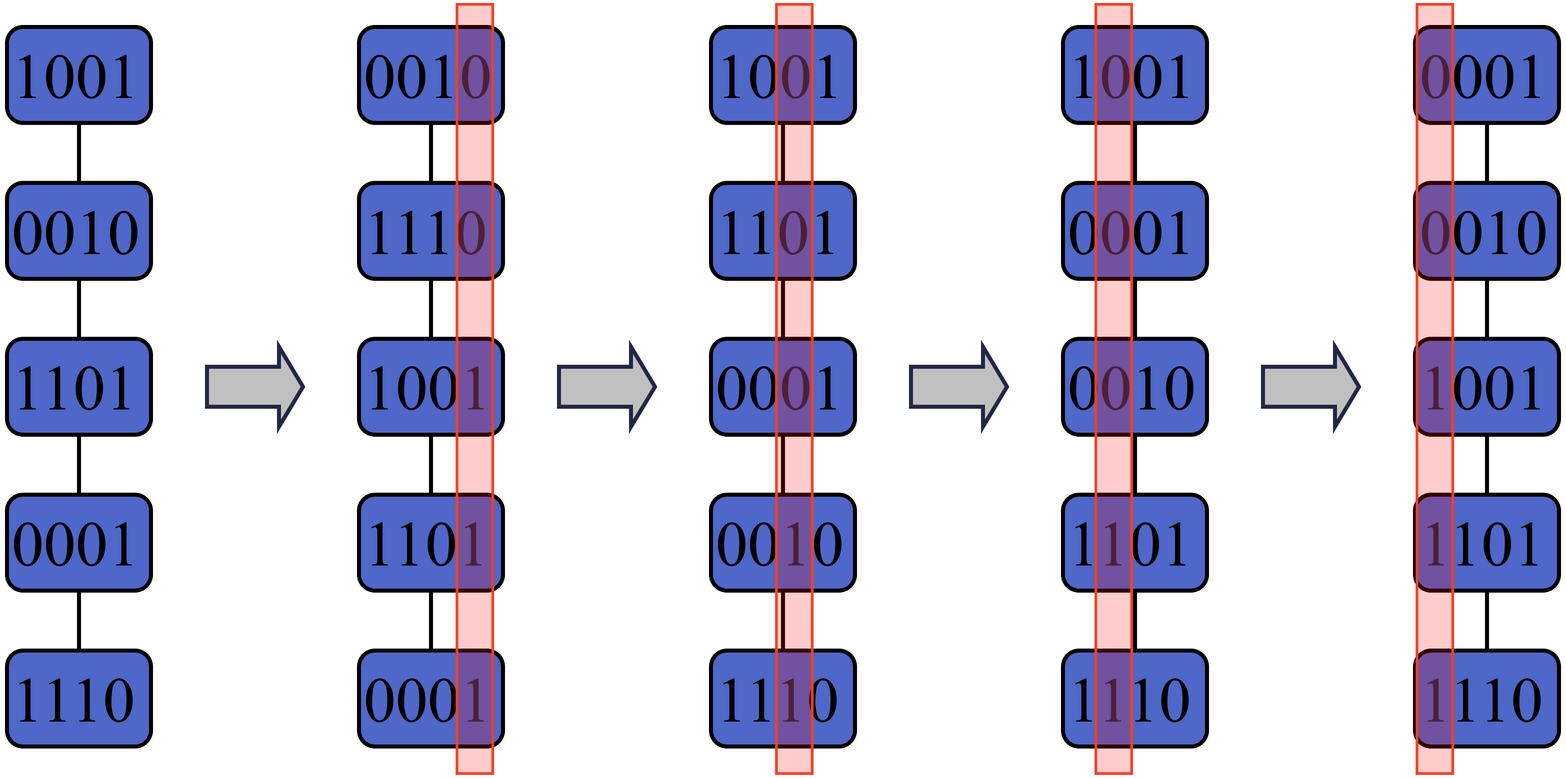


- Consider a sequence of $\boldsymbol{n}$ $\boldsymbol{b}$-bit integers
  $$\boldsymbol{x} = \boldsymbol{x}_{b-1} \ldots \boldsymbol{x}_1\boldsymbol{x}_0$$
- We represent each element as a $\boldsymbol{b}$-tuple of integers in the range $[0, 1]$ and apply radix-sort with $N = 2$
- This application of the radix-sort algorithm runs in $\boldsymbol{O}(\boldsymbol{bn})$ time
- For example, we can sort a sequence of 32-bit integers in linear time

Algorithm ***binaryRadixSort***(***S***)
    Input sequence $\boldsymbol{S}$ of $\boldsymbol{b}$-bit integers
    Output sequence $\boldsymbol{S}$ sorted
    replace each element $\boldsymbol{x}$ of $\boldsymbol{S}$ with the item $(0, \boldsymbol{x})$
    for $\boldsymbol{i} \leftarrow 0$ **to** $\boldsymbol{b} - 1$
        replace the key $\boldsymbol{k}$ of each item $(\boldsymbol{k}, \boldsymbol{x})$ of $\boldsymbol{S}$ with bit $\boldsymbol{x}_i$ of $\boldsymbol{x}$
        bucketSort(***S***, 2)

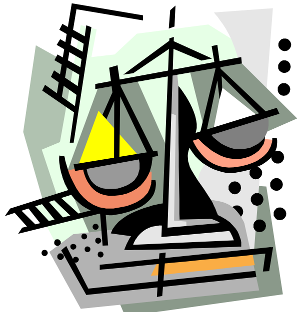
Example

- Sorting a sequence of 4-bit integers



Selection

The Selection Problem



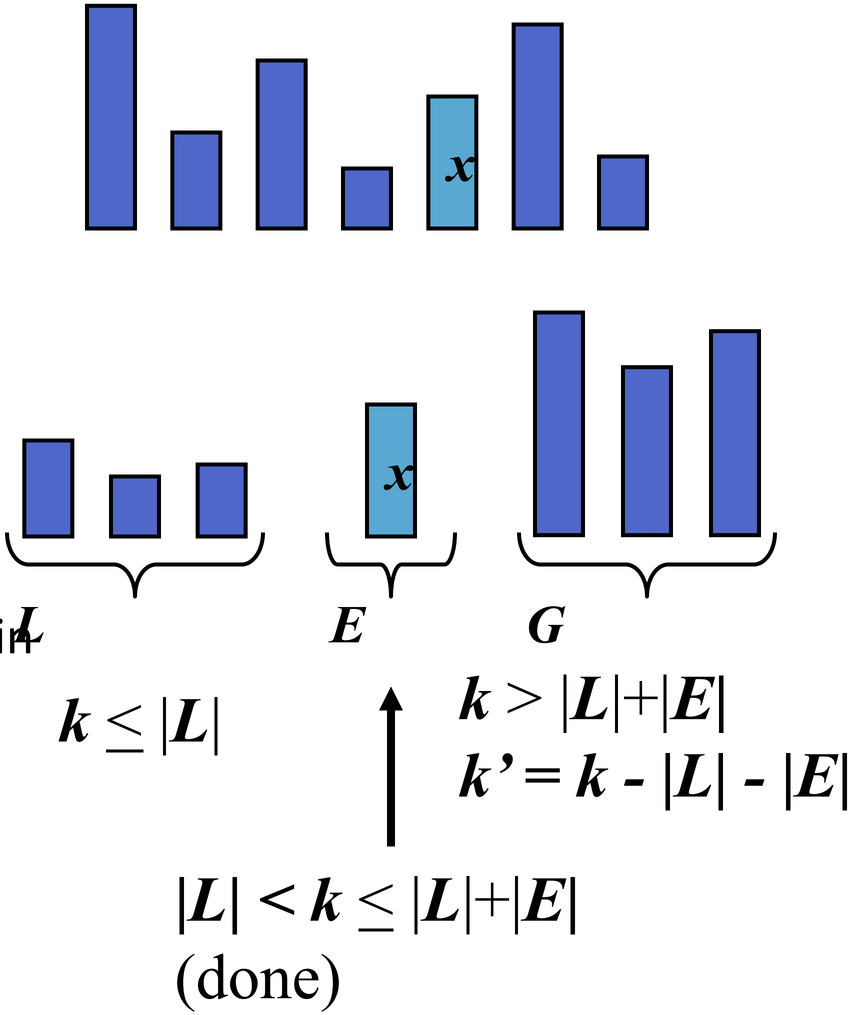
- Given an integer k and n elements $x_1, x_2, \ldots, x_n$, taken from a total order, find the k-th smallest element in this set.
- Of course, we can sort the set in O(n log n) time and then index the k-th element.

k=3 7 4 9 6 2 $\rightarrow$ 2 4 6 7 9

- Can we solve the selection problem faster?

Quick-Select

- Quick-select is a randomized selection algorithm based on the prune-and-search paradigm:
  - Prune: pick a random element x (called pivot) and partition S into
    - L: elements less than x
    - E: elements equal x
    - G: elements greater than x
  - Search: depending on k, either answer is in E, or we need to recur in either L or G



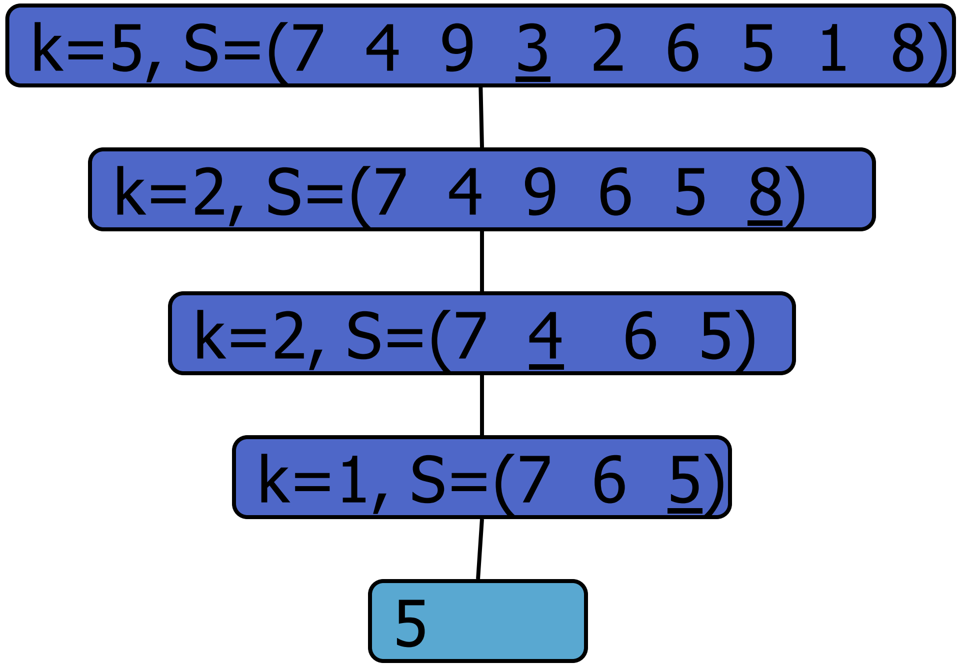
Partition

- We partition an input sequence as in the quick-sort algorithm:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-select takes $O(n)$ time

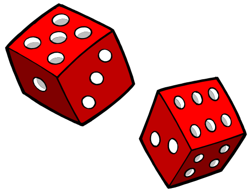
```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.addLast(y)
        else if y = x
            E.addLast(y)
        else { y > x }
            G.addLast(y)
    return L, E, G
```

Quick-Select Visualization

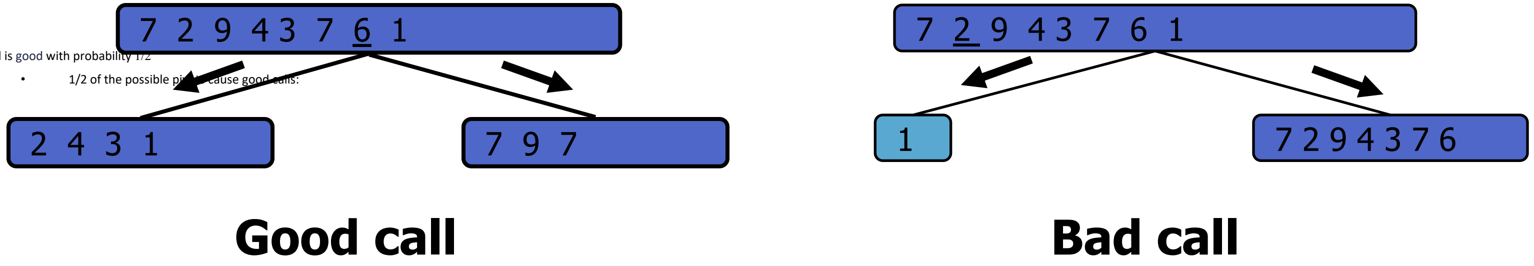
- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence



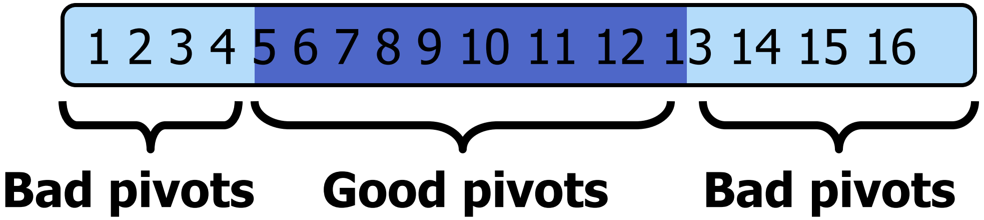
Expected Running Time



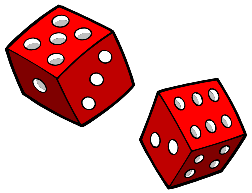
- Consider a recursive call of quick-select on a sequence of size s
  - Good call: the sizes of L and G are each less than $3s/4$
  - Bad call: one of L and G has size greater than $3s/4$



- A call is good with probability 1/2
  - 1/2 of the possible pivots cause good calls:



Expected Running Time, Part 2

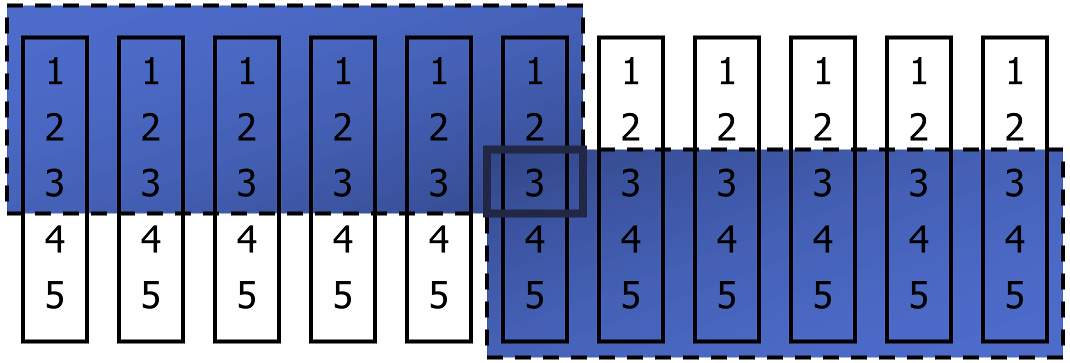


- Probabilistic Fact #1: The expected number of coin tosses required in order to get one head is two
- Probabilistic Fact #2: Expectation is a linear function:
  - $E(X+Y) = E(X) + E(Y)$
  - $E(cX) = cE(X)$
- Let T(n) denote the expected running time of quick-select.
- By Fact #2,
  - $T(n) \leq T(3n/4) + bn$*(expected # of calls before a good call)
- By Fact #1,
  - $T(n) \leq T(3n/4) + 2bn$
- That is, T(n) is a geometric series:
  - $T(n) \leq 2bn + 2b(3/4)n + 2b(3/4)^2 n + 2b(3/4)^3 n + \ldots$
- So T(n) is O(n).
- We can solve the selection problem in O(n) expected time.

Deterministic Selection

- We can do selection in O(n) worst-case time.
- Main idea: recursively use the selection algorithm itself to find a good pivot for quick-select:
  - Divide S into n/5 sets of 5 each
  - Find a median in each set
  - Recursively find the median of the "baby" medians.

Min size for L



Min size for G