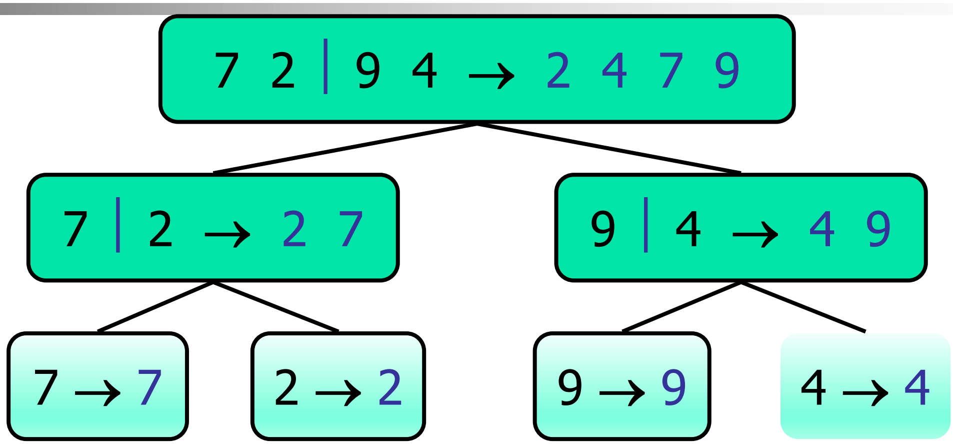
COMP9024: Data Structures and Algorithms

Sorting Algorithms

Outline

- Merge Sort
- Quick Sort
- Bucket-Sort
- Radix Sort
- Sorting Lower Bound

Merge Sort



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - 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
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

```
Algorithm mergeSort(S)
    Input sequence S with n
    elements
    Output sorted sequence S
  {
    if ( S.size() > 1 )
      {
        (S1, S2) = partition(S, n/2);
        mergeSort(S1);
        mergeSort(S2);
        S = merge(S1, S2);
      }
  }
```

Merging Two Sorted Sequences

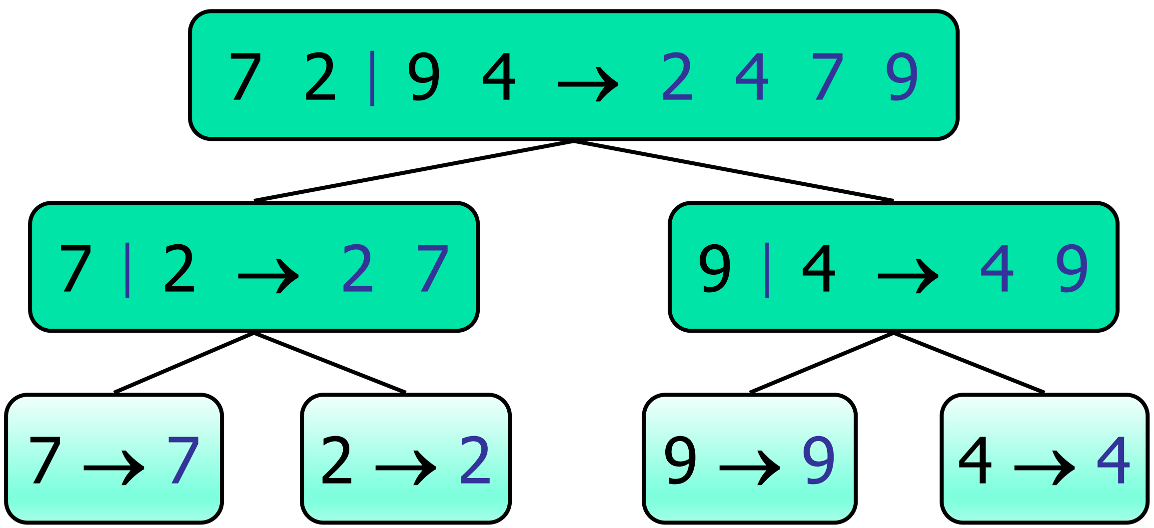
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

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

```
{
 S = empty sequence;
 while ( ¬isEmpty(A)  ∧ ¬ isEmpty(B) )
    if  ( firstElement(A) < firstElement(B) )
       insertLast(S, removeFirst(A));
    else
       insertLast(S, removeFirst(B));
 while  (¬ isEmpty(A))
    insertLast(S, removeFirst(A));
 while  ( ¬ isEmpty(B) )
    insertLast(S, removeFirst(B));
 return S;
}
```

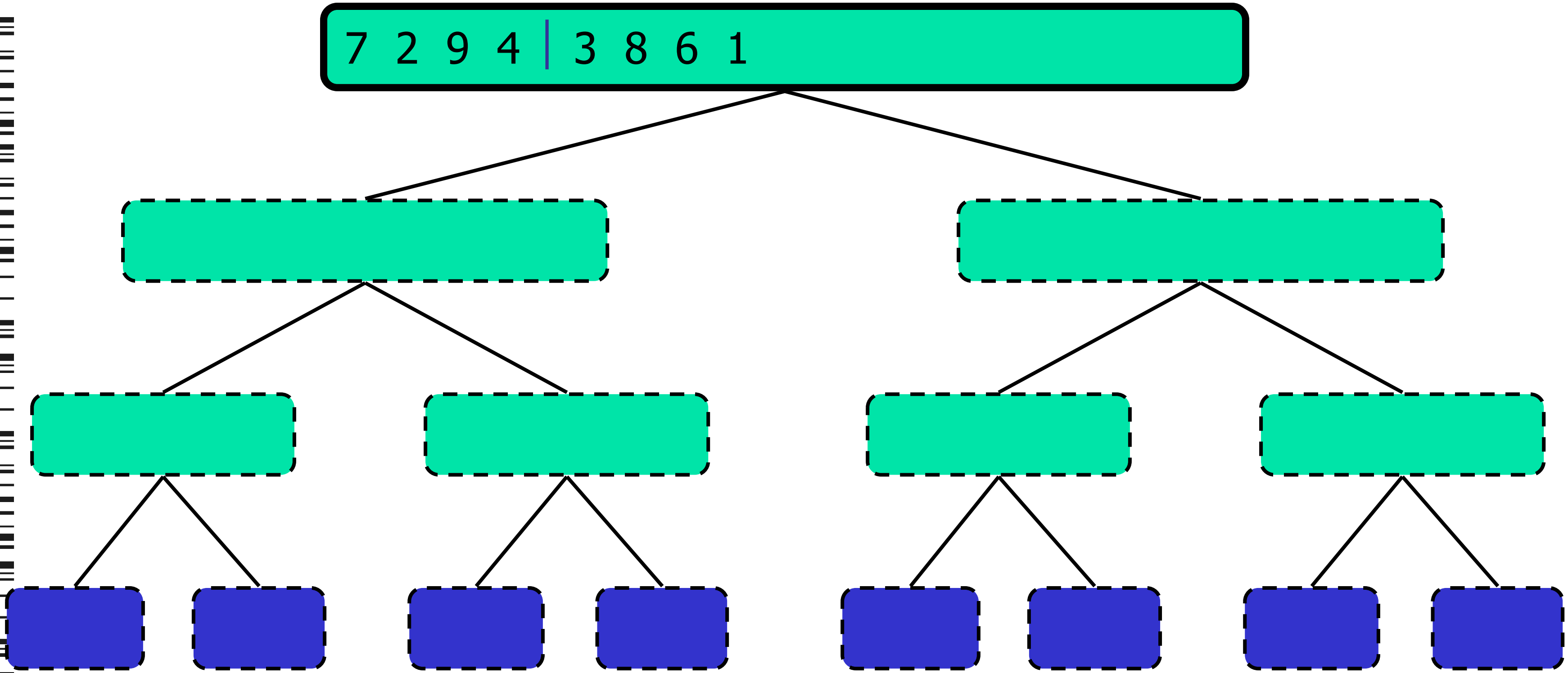
Merge-Sort Tree

- 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 0 or 1



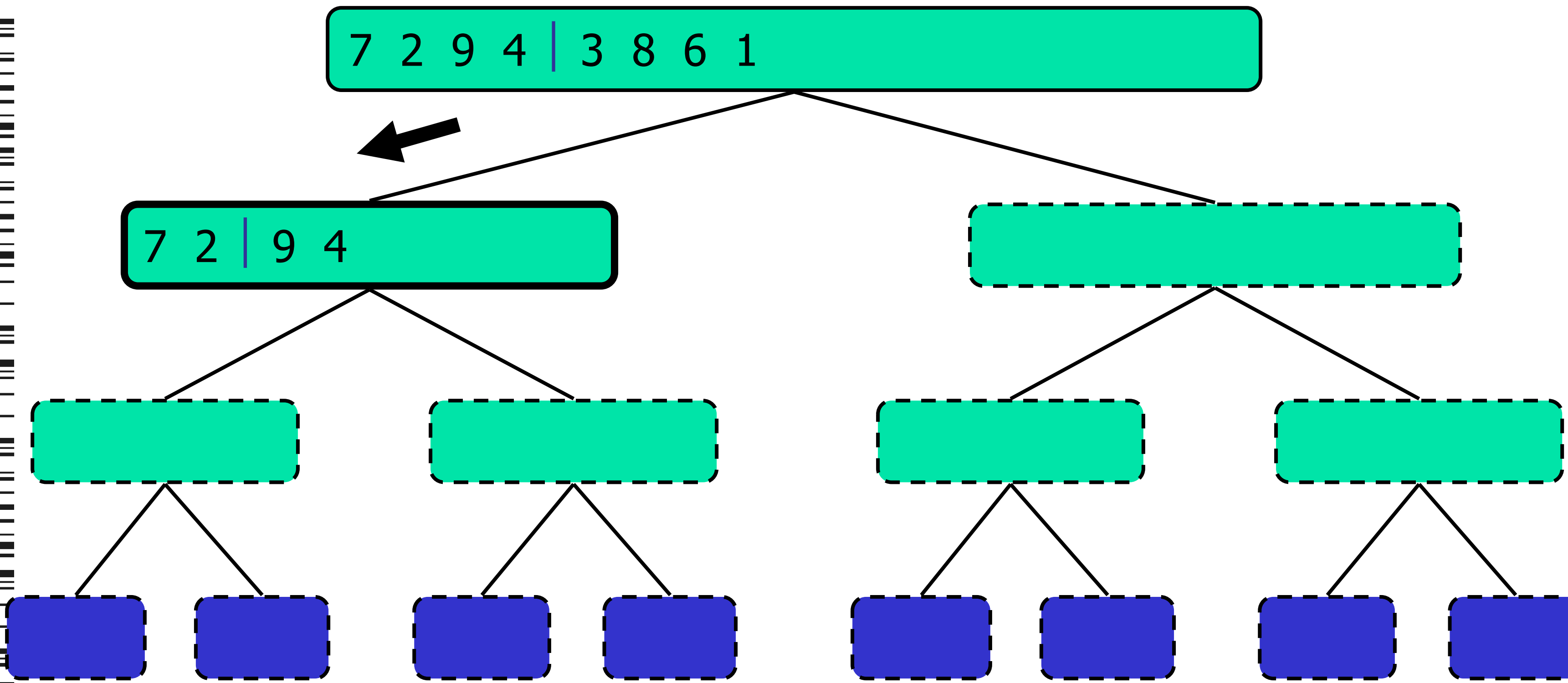
Execution Example (1/10)

- Partition



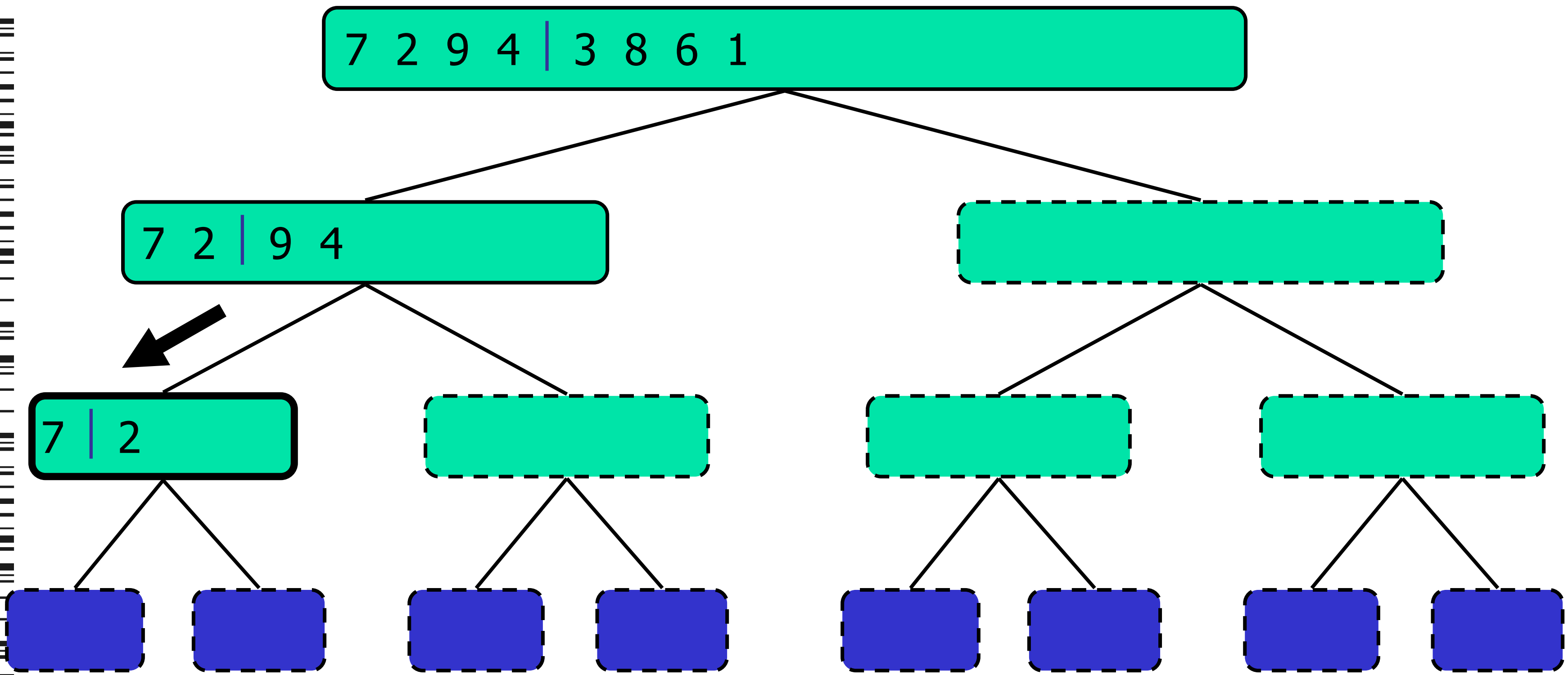
Execution Example (2/10)

- Recursive call, partition



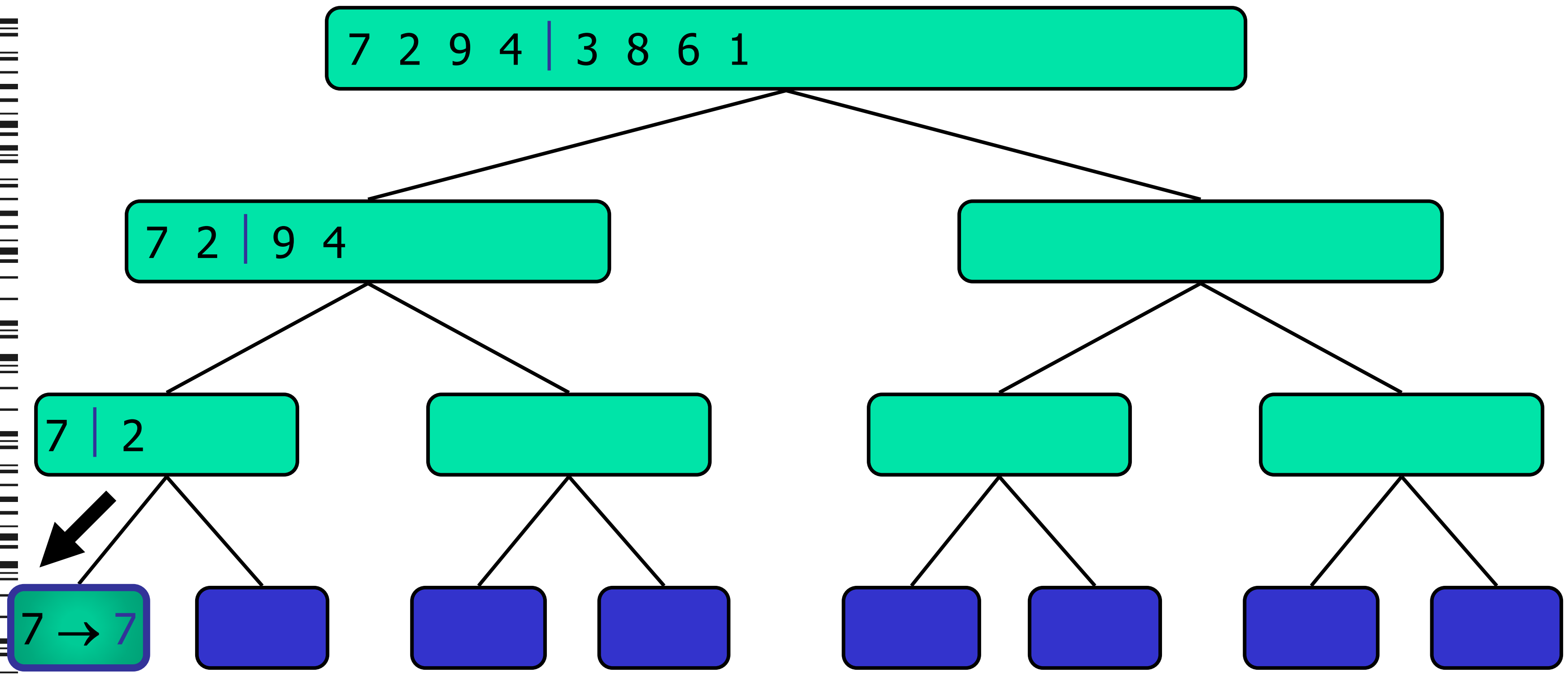
Execution Example (3/10)

- Recursive call, partition



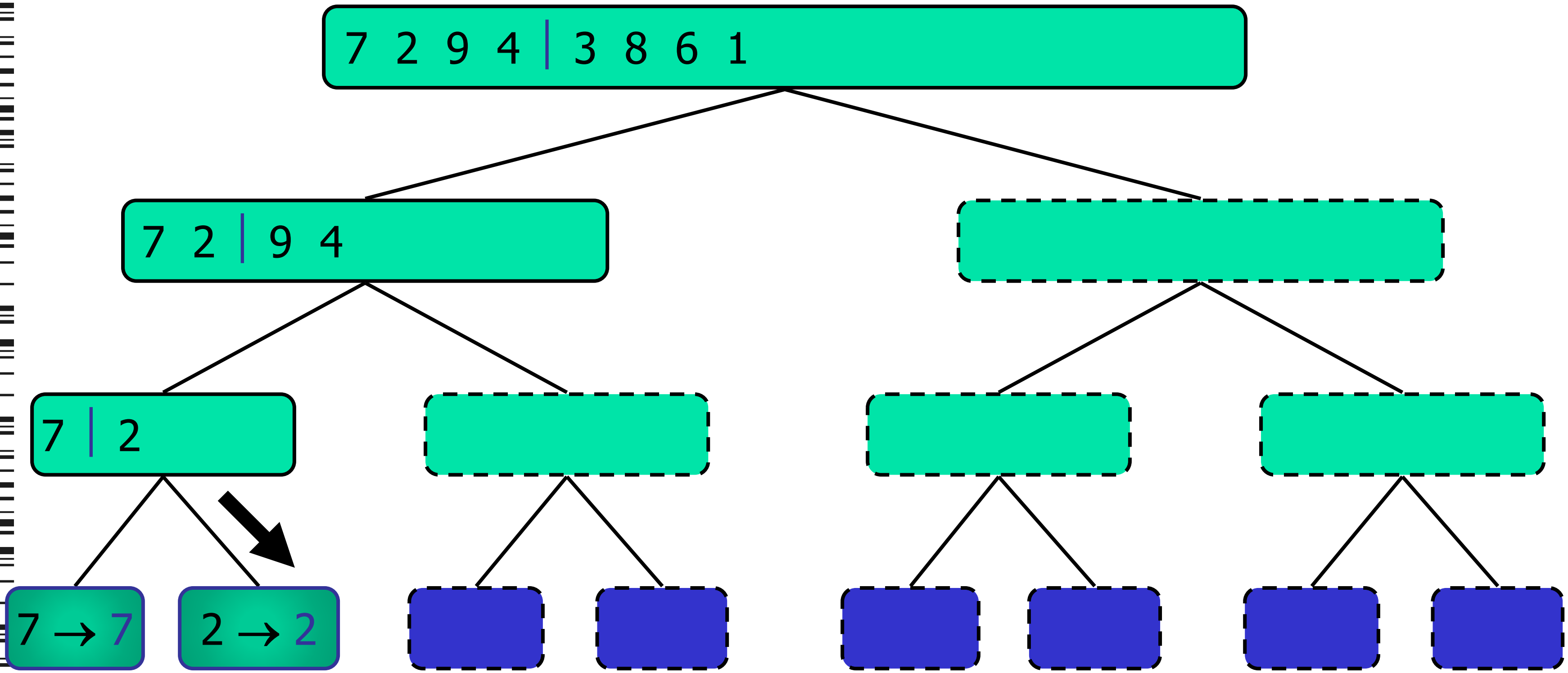
Execution Example (4/10)

- Recursive call, base case



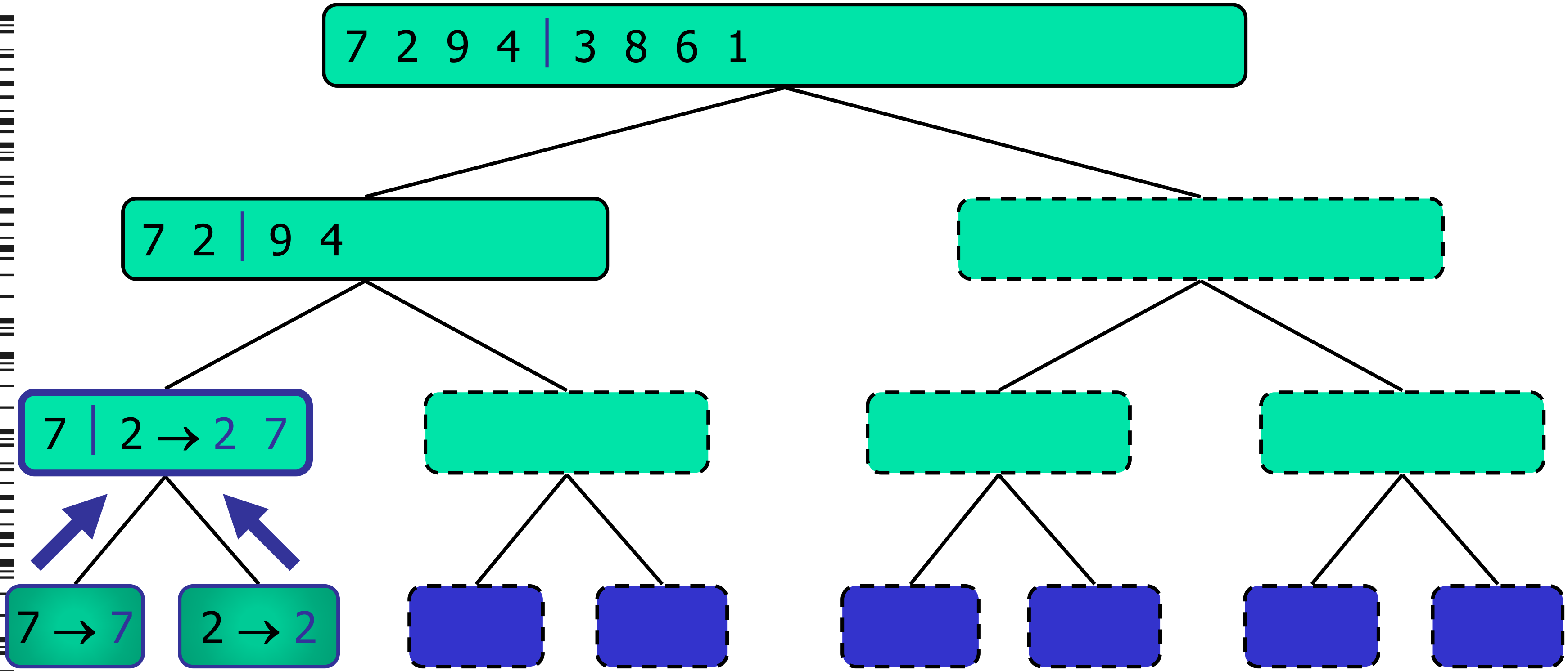
Execution Example (5/10)

- Recursive call, base case



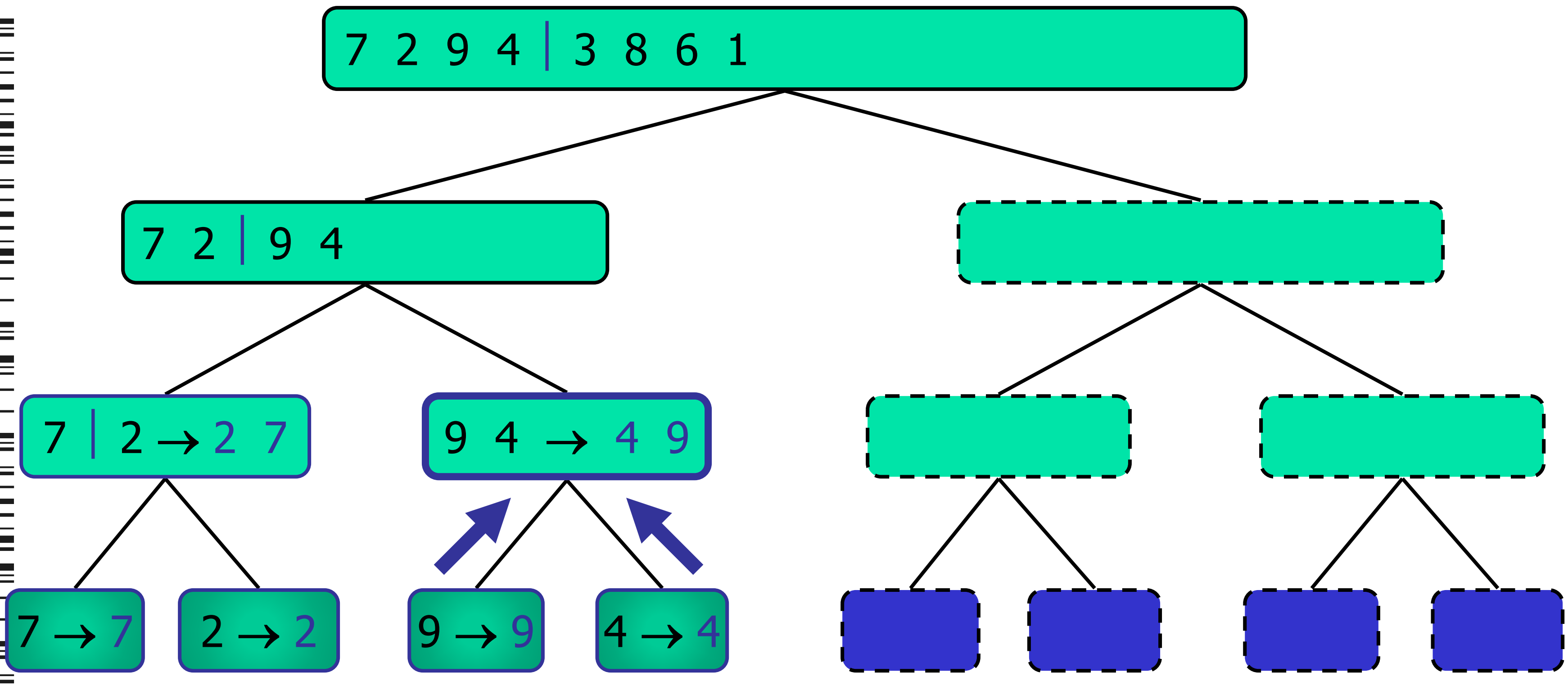
Execution Example (6/10)

- Merge



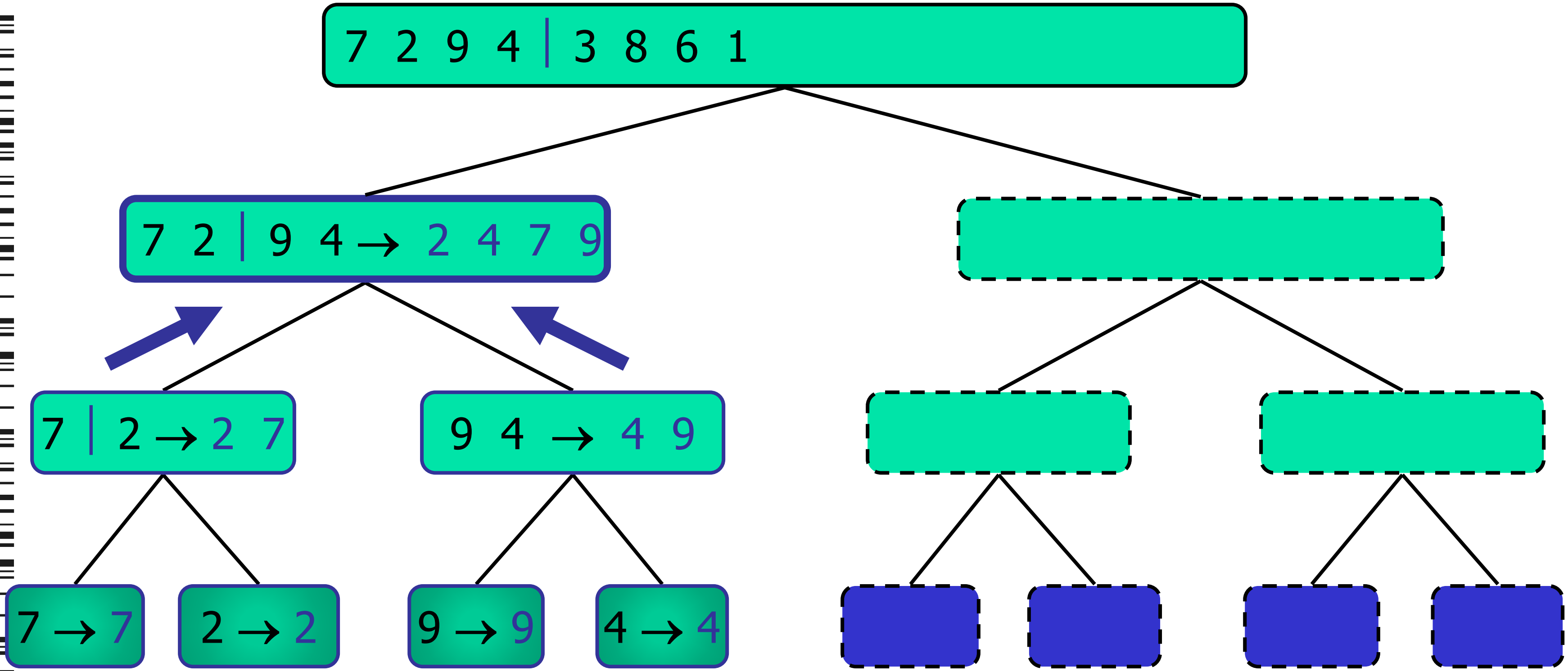
Execution Example (7/10)

- Recursive call, …, base case, merge



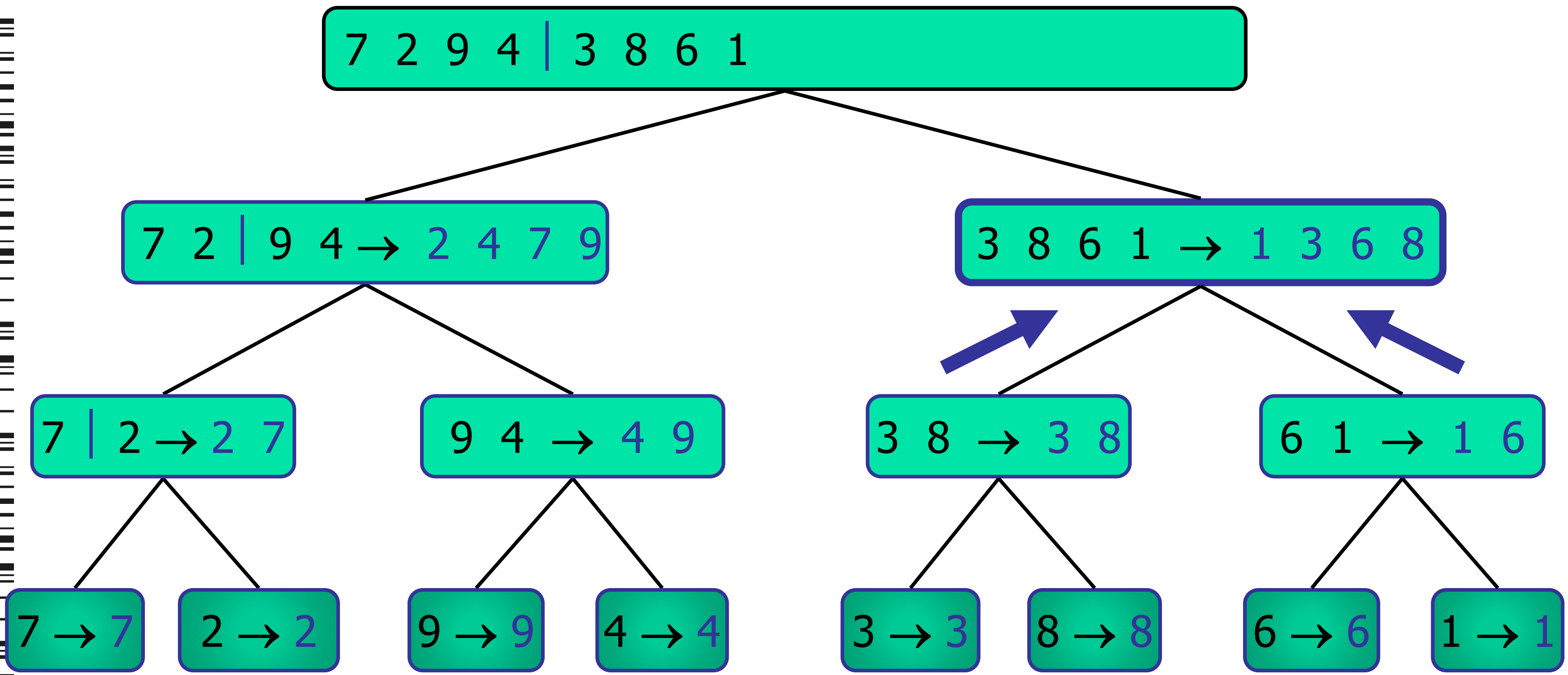
Execution Example (8/10)

- Merge



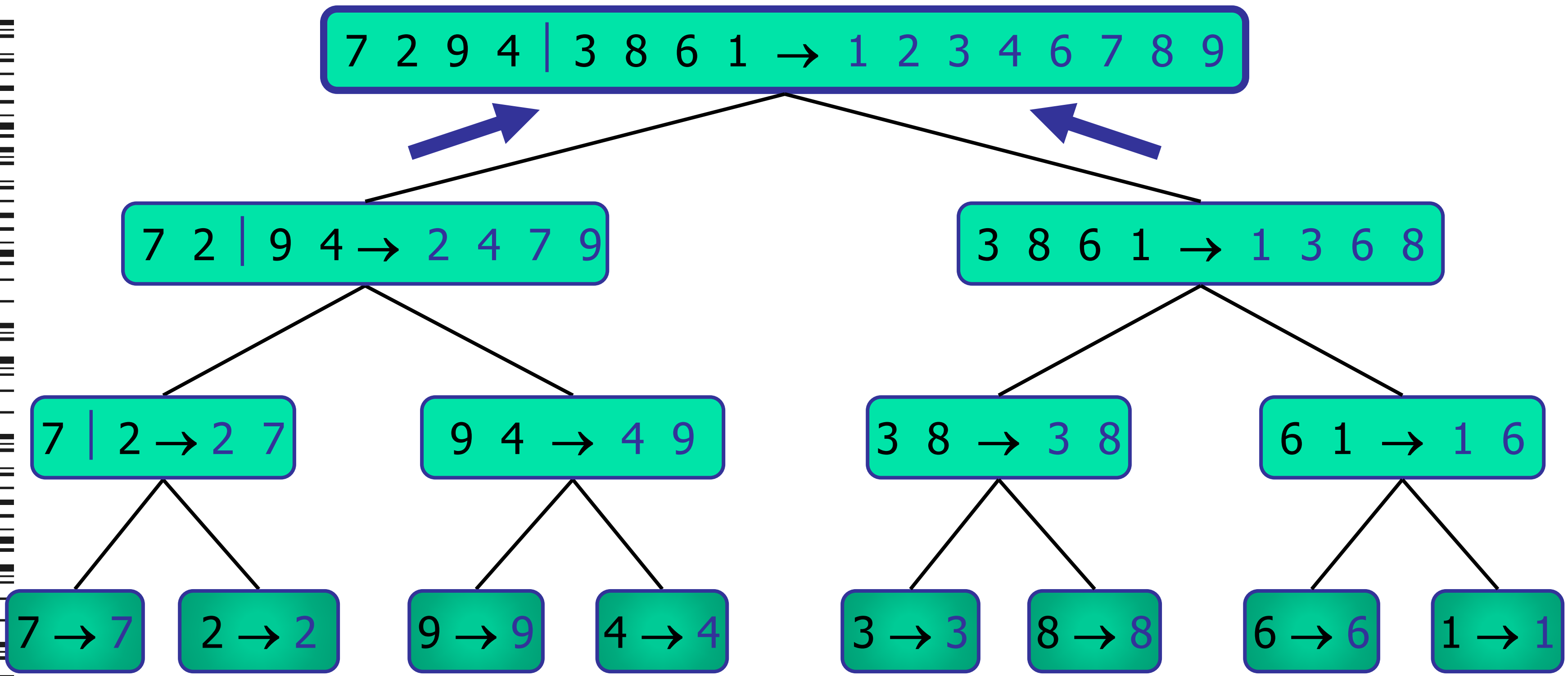
Execution Example (9/10)

- Recursive call, …, merge, merge



Execution Example (10/10)

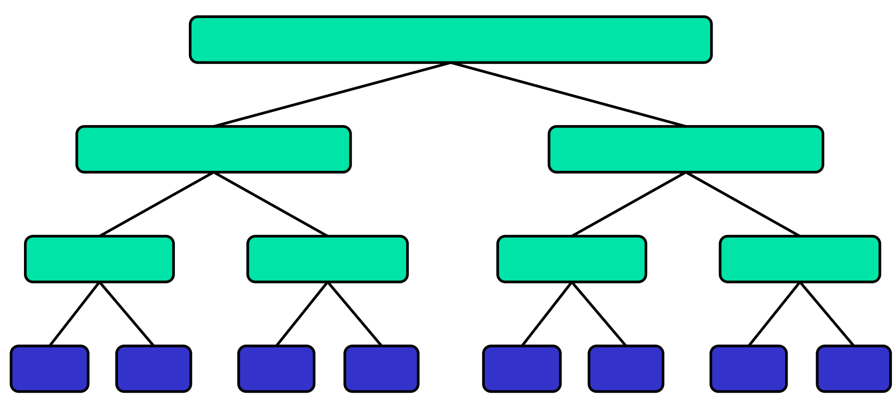
- Merge



Analysis of Merge-Sort

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



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)

Recursive Merge-Sort on Array (1/3)

```
// Array A[] is the source array to sort; array B[] is a temporary array.
void TopDownMergeSort(A[], B[], n)
{
   CopyArray(A, 0, n, B);      // one time copy of A[] to B[]
   TopDownSplitMerge(B, 0, n, A);   // sort data from B[] into A[]
}

void CopyArray(A[], iBegin, iEnd, B[])
{
   for(k = iBegin; k < iEnd; k++)
      B[k] = A[k];
}
```

Recursive Merge-Sort on Array (2/3)

```
// Sort the given run of array A[] using array B[] as a source.
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
void TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
   if(iEnd - iBegin < 2)              // if run size == 1
      return;                         //   consider it sorted
   // split the run longer than 1 item into halves
   iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
   // recursively sort both runs from array A[] into B[]
   TopDownSplitMerge(A, iBegin,  iMiddle, B);  // sort the left  run
   TopDownSplitMerge(A, iMiddle,    iEnd, B);  // sort the right run
   // merge the resulting runs from array B[] into A[]
   TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}
```

Recursive Merge-Sort on Array (3/3)

```
// Left source half is A[ iBegin:iMiddle-1].
// Right source half is A[iMiddle:iEnd-1   ].
// Result is B[iBegin:iEnd-1].
void TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{  i = iBegin, j = iMiddle;
   // while there are elements in the left or right runs
   for (k = iBegin; k < iEnd; k++) {
      // If left run head exists and is <= existing right run head.
      if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) {
         B[k] = A[i];
         i = i + 1;
      } else {
         B[k] = A[j];
         j = j + 1;
      }
   }
}
```

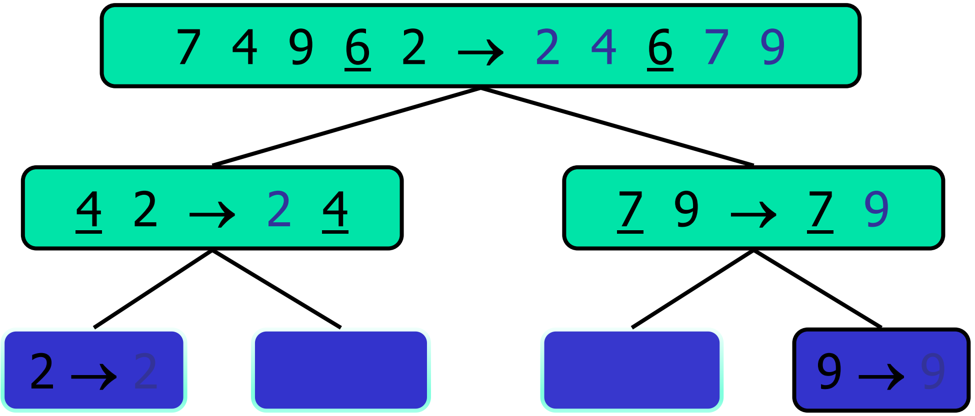
Nonrecursive Merge-Sort on Array (1/2)

```
// array A[] has the items to sort; array B[] is a work array
void BottomUpMergeSort(A[], B[], n)
{  // Each 1-element run in A is already "sorted".
   // Make successively longer sorted runs of length 2, 4, 8, 16... until whole array is sorted.
   for (width = 1; width < n; width = 2 * width)
   {  // Array A is full of runs of length width.
      for (i = 0; i < n; i = i + 2 * width)
      {  // Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1] to B[]
         // or copy A[i:n-1] to B[] ( if(i+width >= n) )
         BottomUpMerge(A, i, min(i+width, n), min(i+2*width, n), B);
      }
      // Now work array B is full of runs of length 2*width.
      // Copy array B to array A for next iteration.
      // A more efficient implementation would swap the roles of A and B.
      CopyArray(B, 0, n, A);
      // Now array A is full of runs of length 2*width.
   }
}
```

Nonrecursive Merge-Sort on Array (2/2)

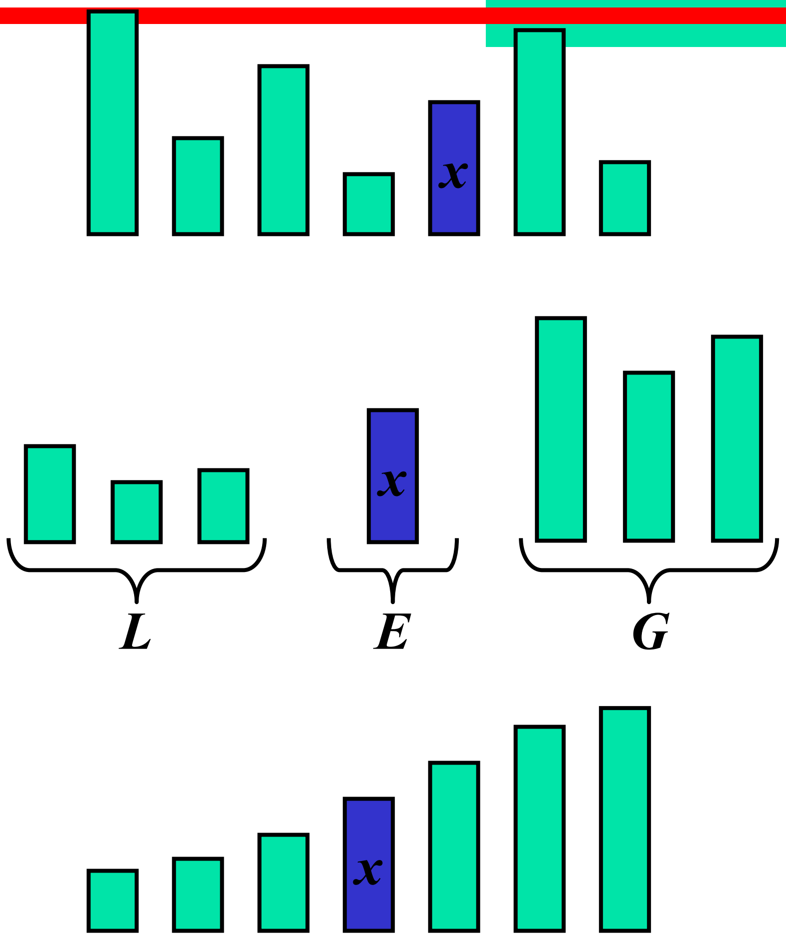
```
// Left run is A[iLeft :iRight-1].
// Right run is A[iRight:iEnd-1  ].
void BottomUpMerge(A[], iLeft, iRight, iEnd, B[])
{
    i = iLeft, j = iRight;
    // While there are elements in the left or right runs...
    for (k = iLeft; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iRight && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}
```

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
  - Recur: sort L and G
  - Conquer: 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 $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

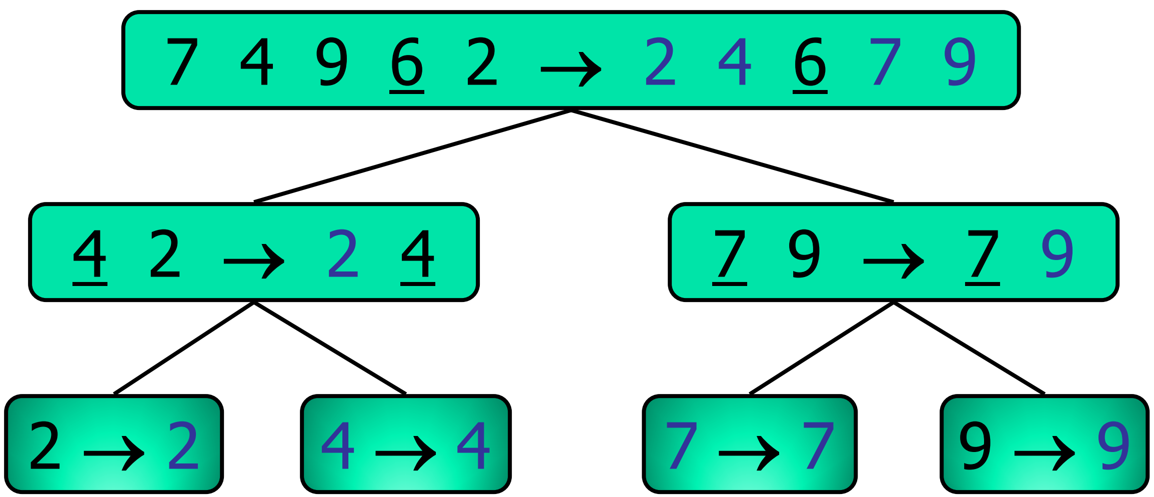
Algorithm *partition*(*S*, *p*)

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.insertLast(y);
      else if ( y = x )
         E.insertLast(y);
      else  // y > x
         G.insertLast(y);}
   return L, E, G;
}
```

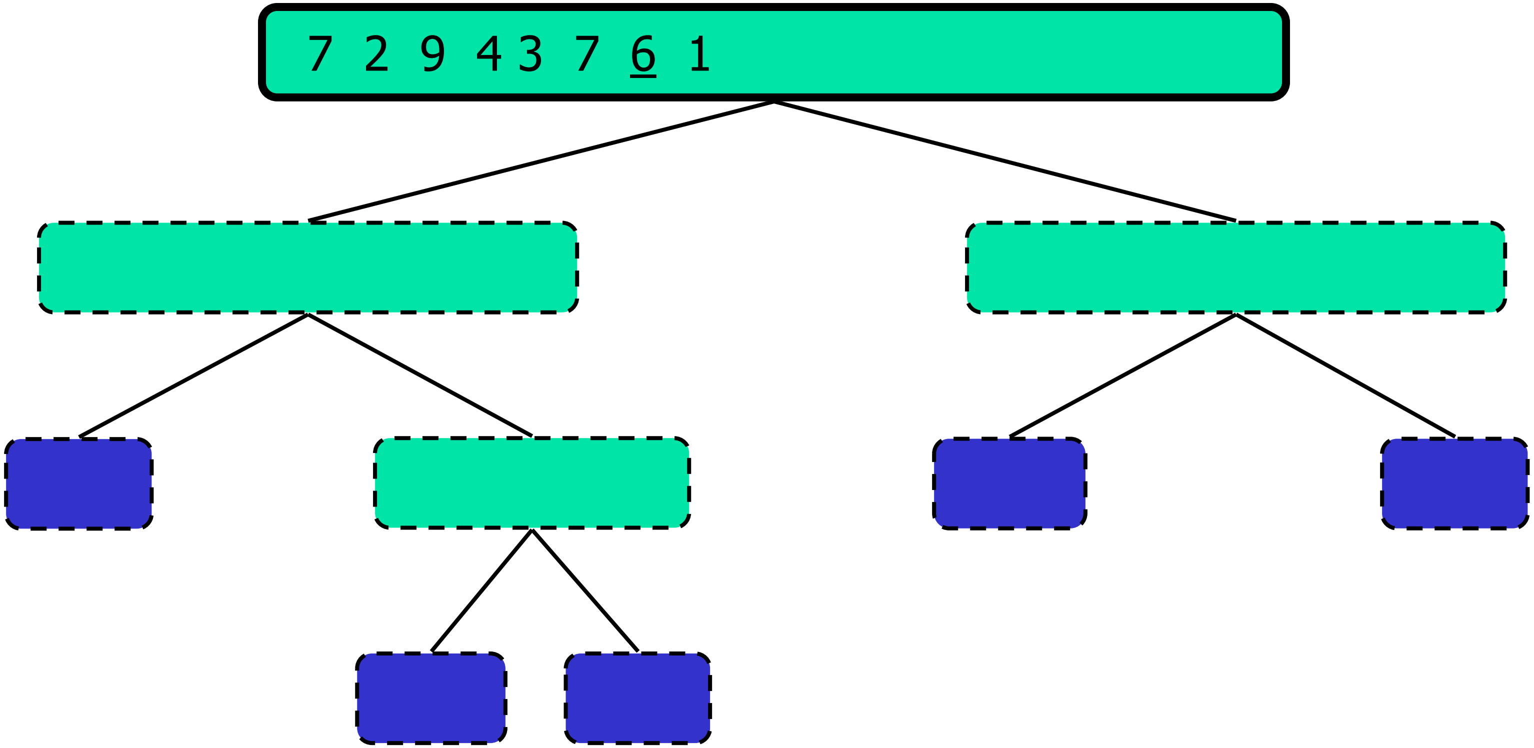
Quick-Sort Tree

- 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



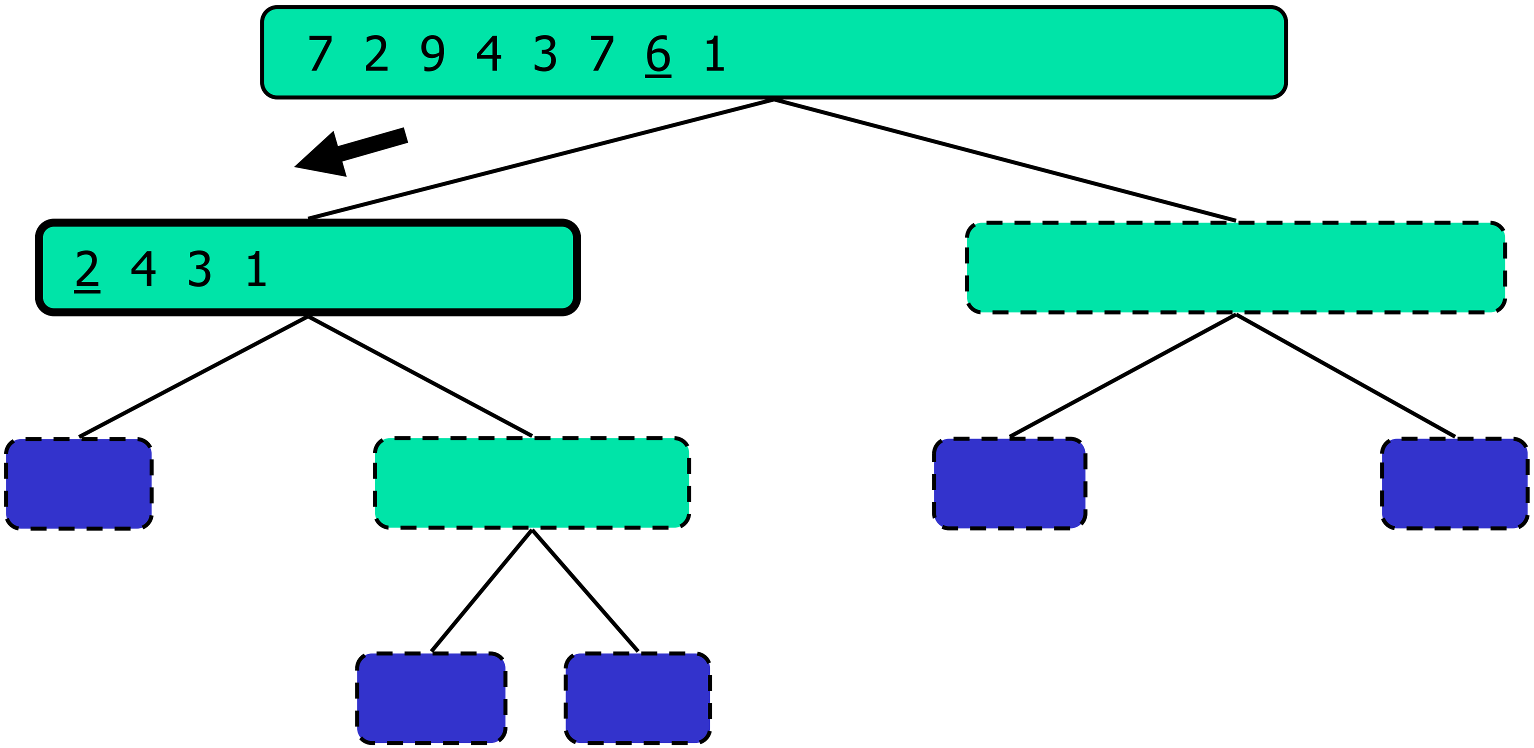
Execution Example (1/7)

- Pivot selection



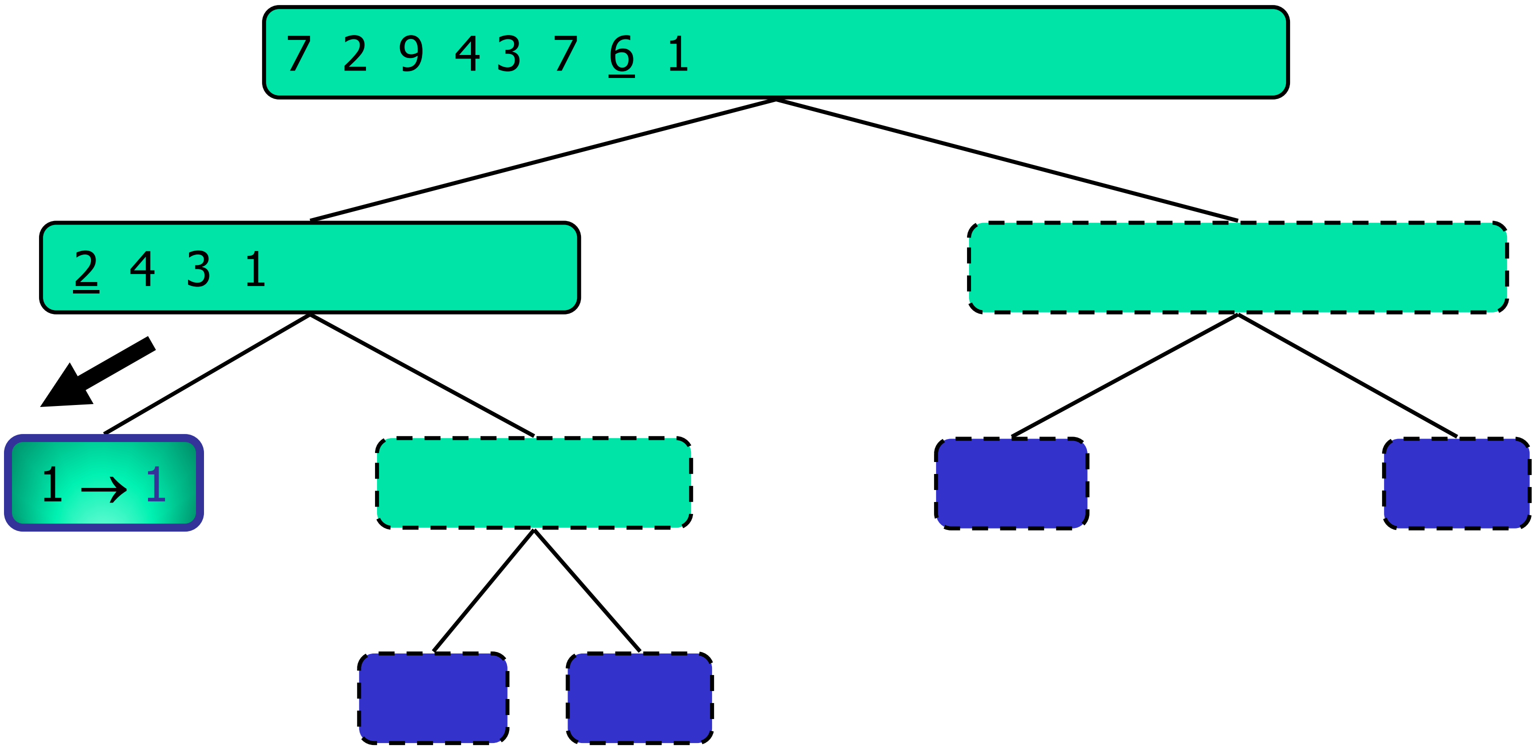
Execution Example (2/7)

- Partition, recursive call, pivot selection



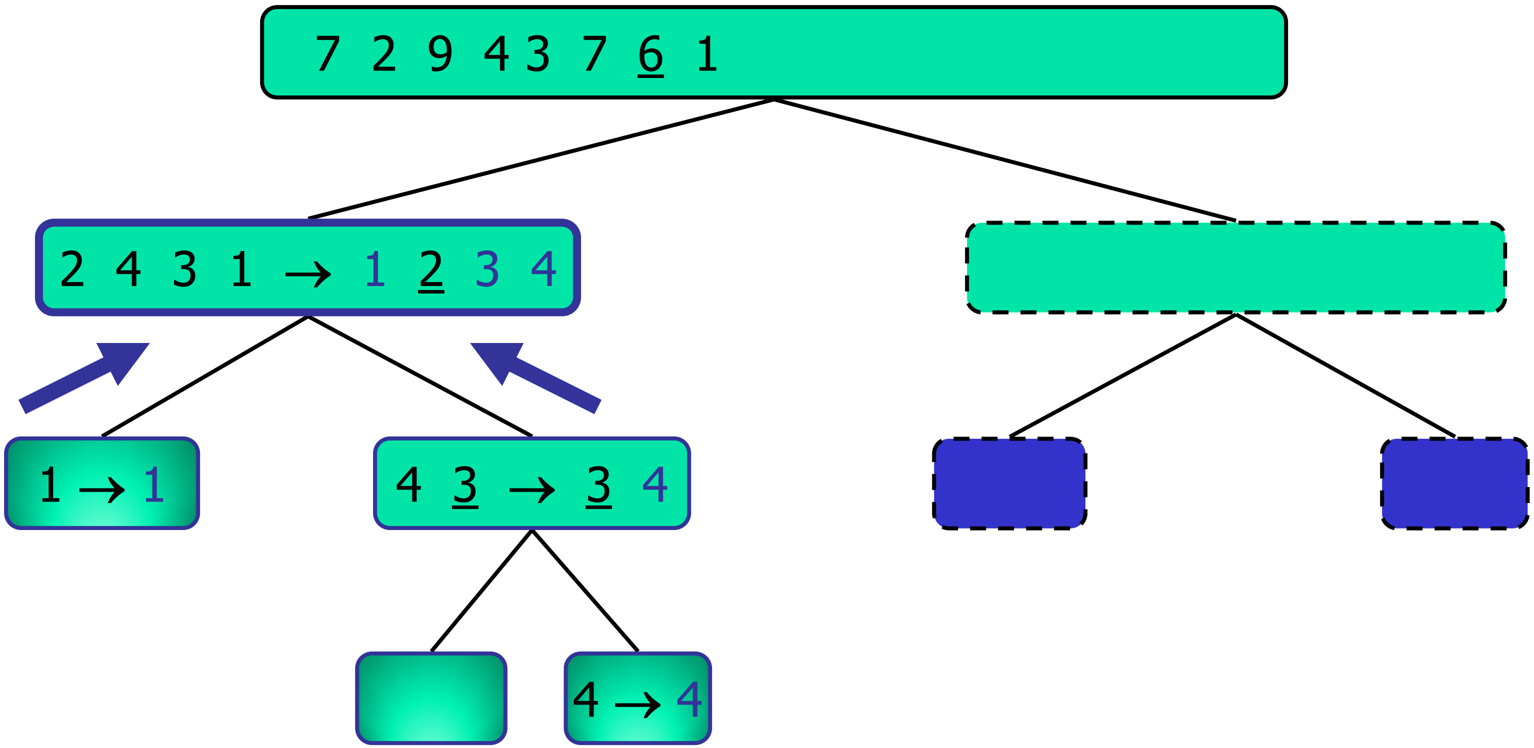
Execution Example (3/7)

- Partition, recursive call, base case



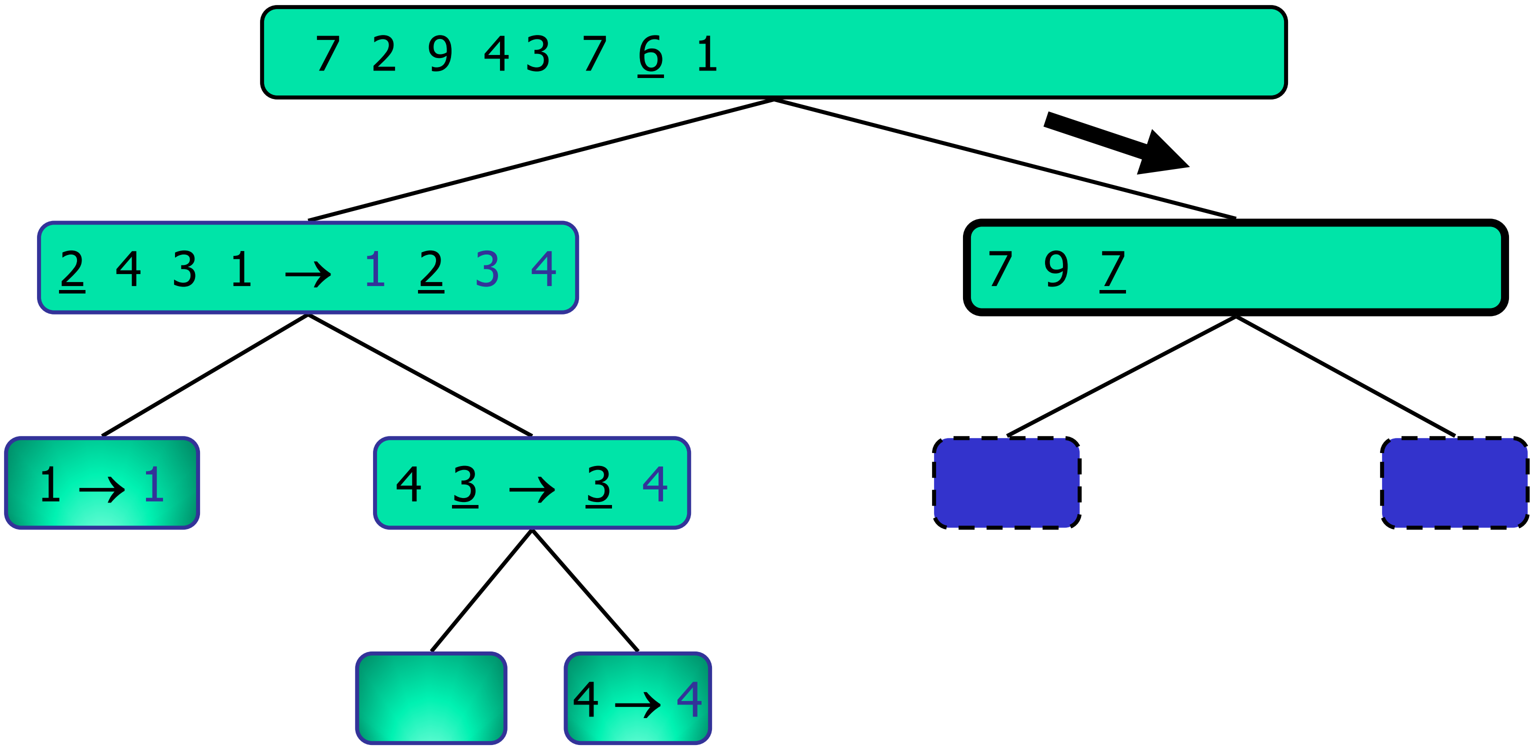
Execution Example (4/7)

- Recursive call, …, base case, join



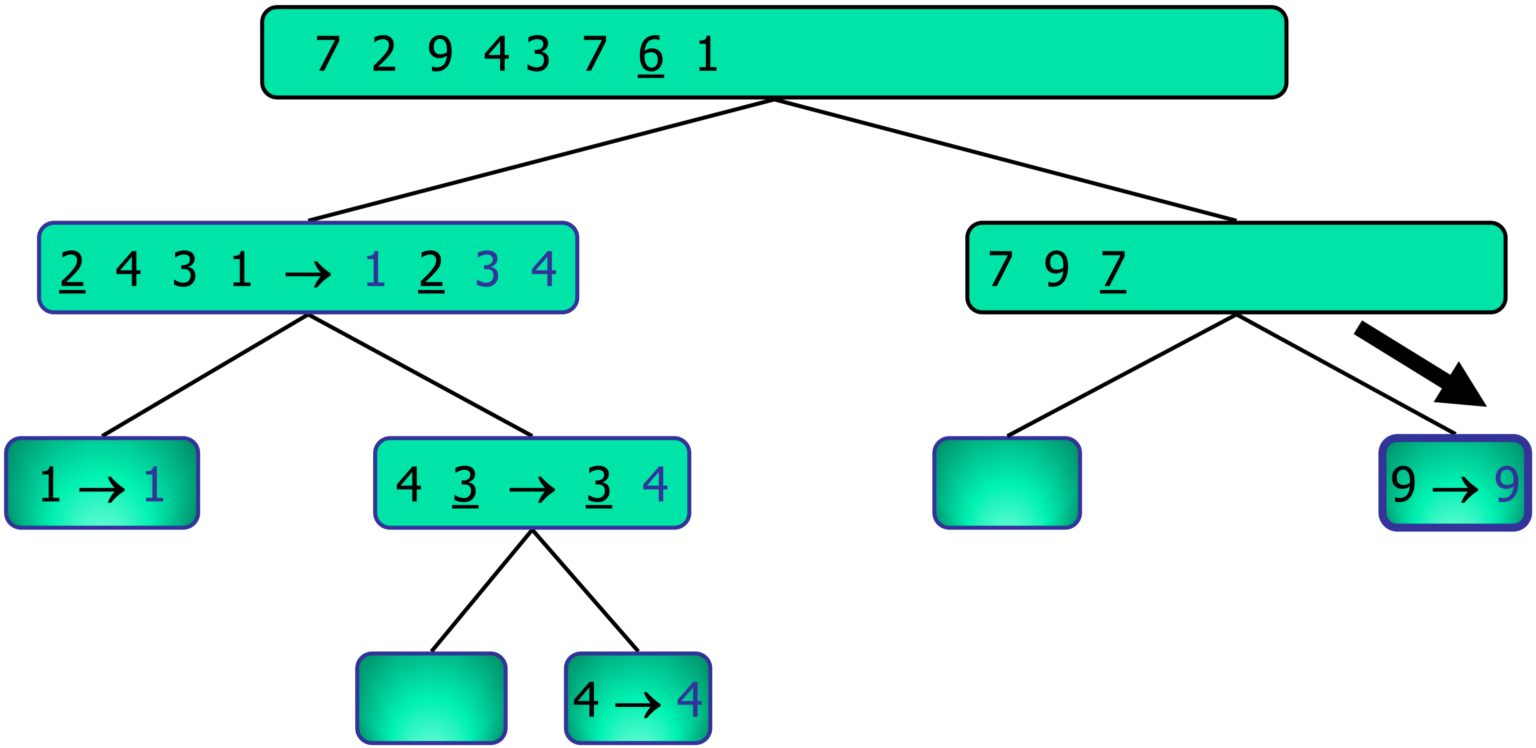
Execution Example (5/7)

- Recursive call, pivot selection



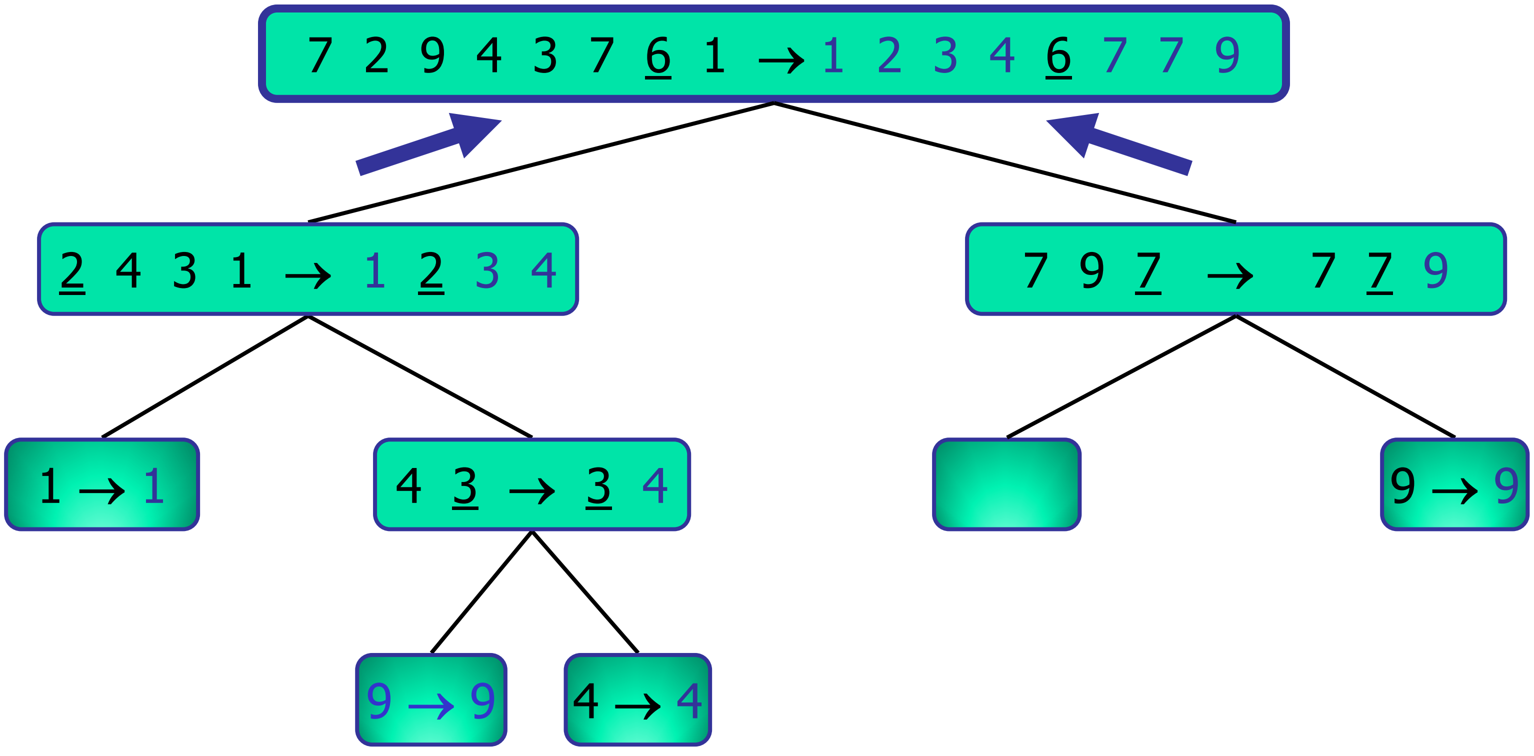
Execution Example (6/7)

- Partition, …, recursive call, base case



Execution Example (7/7)

- Join, join



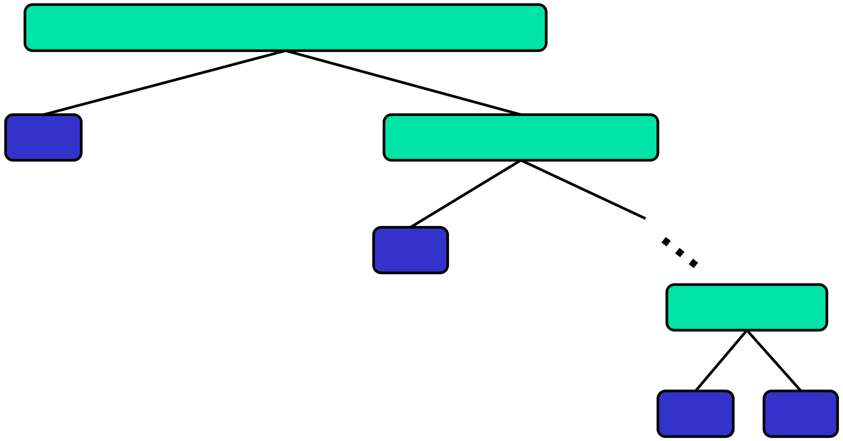
Worst-case Running Time

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

$$n + (n - 1) + \ldots + 2 + 1$$

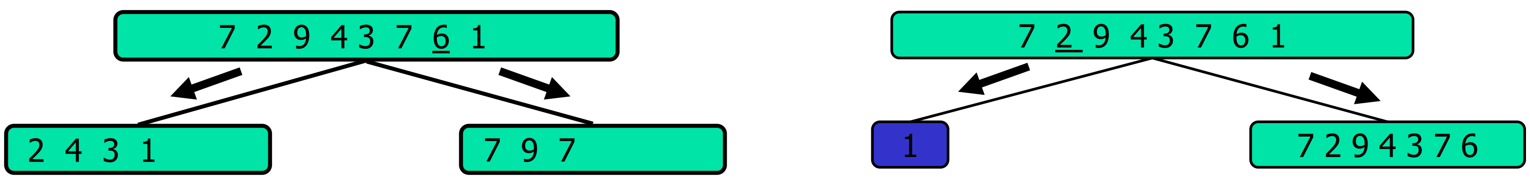
- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n - 1$
...	...
$n - 1$	1



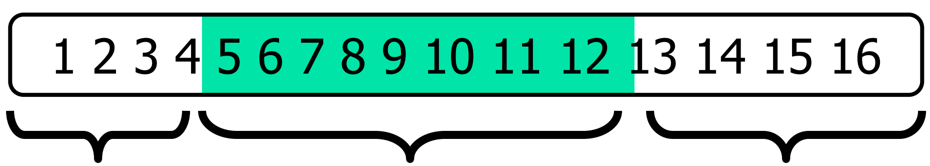
Expected Running Time (1/2)

- 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**

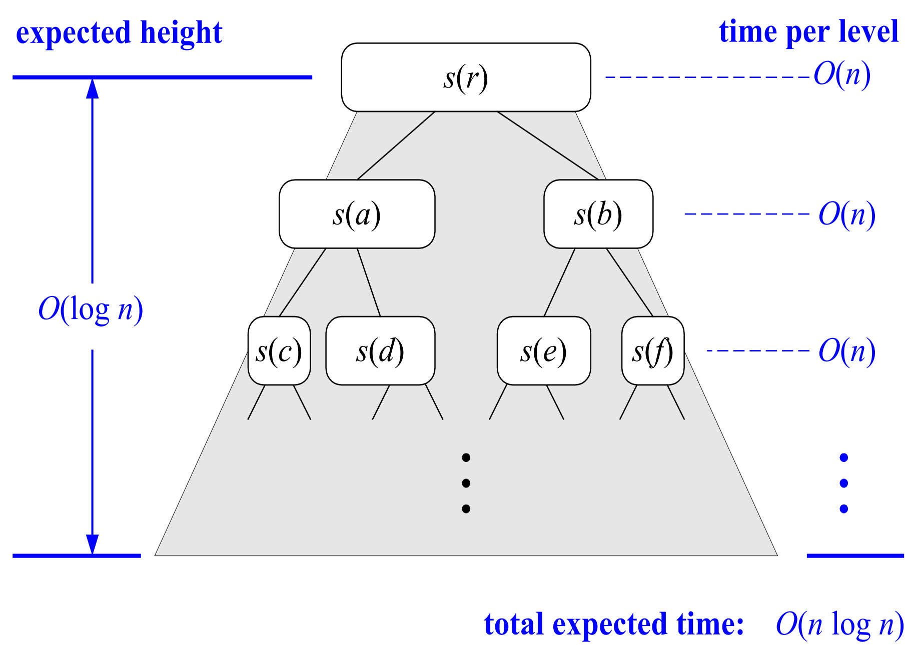
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



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

Expected Running Time (2/2)

- 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
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Therefore, we have
  - 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 or 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

- Quick-sort can be implemented to run in-place
- In the partition step, we rearrange the elements of the input sequence such that
  - the elements less than the pivot have indices less than p
  - the elements equal to or greater than the pivot have indices between $p+1$ and r
- The recursive calls consider
  - elements with indices less than p.
  - elements with indices greater than p

Algorithm ***inPlaceQuickSort***(S, l, r)

Input sequence S, indices l and r of the first and last elements

Output sequence S with the elements of indices between l and r rearranged in increasing order

{ **if** $l \geq r$

  return;

 k = a random integer between l and r;

 pivot=$S[k]$;

 Swap pivot and S[r];

 // pivot is the last element now

 p = ***inPlacePartition***();

 inPlaceQuickSort(S, l, p-1);

 inPlaceQuickSort(S, p+1, r);

}

In-Place Partitioning (1/3)

```
inPlacePartition()
 { left = l;                // scan right to locate the end of G U E
   right = r;              // scan left to locate the end of L
   while left < right
      {
        while left  <  right and S[left] < pivot
           left ++;           // insert into L
        while left < right  and S[right] ≥ pivot
           right --;         // insert into G
        if  left < right
           Swap S[left] and S[right];
       }
   Swap S[left] and pivot;
   return left;
 }
```

In-Place Partitioning (2/3)

- 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).

left *right*

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 9 **<u>6</u>** (pivot = 6)

- Repeat until *left* and *right* meet:
  - Scan left to the right until finding an element $\geq$ pivot.
  - Scan right to the left until finding an element < pivot.
  - Swap elements at indices *left* and *right*

$\longrightarrow$ *left* *right* $\longleftarrow$

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 9 **<u>6</u>**

In-Place Partitioning (2/3)

- Swap the element at *left* and pivot

right

↓

3 2 5 1 0 2 3 5 9 7 7 9 8 9 7 9 **<u>6</u>**

↑

left

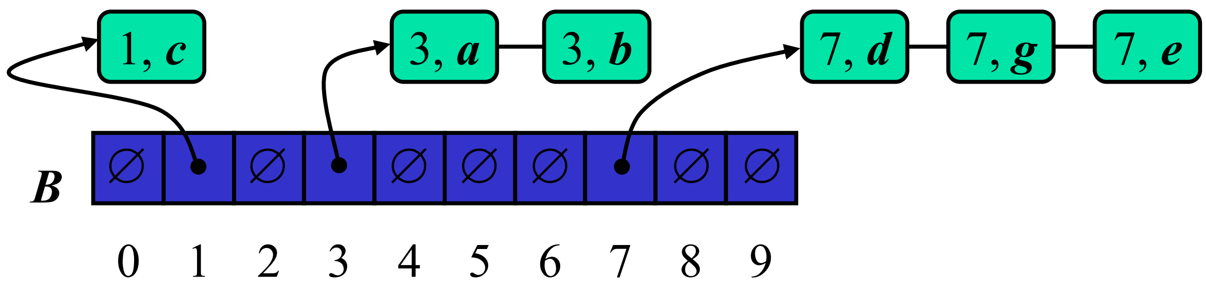
⇓

3 2 5 1 0 2 3 5 **<u>6</u>** 7 7 9 8 9 7 9 9

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, value) entries 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

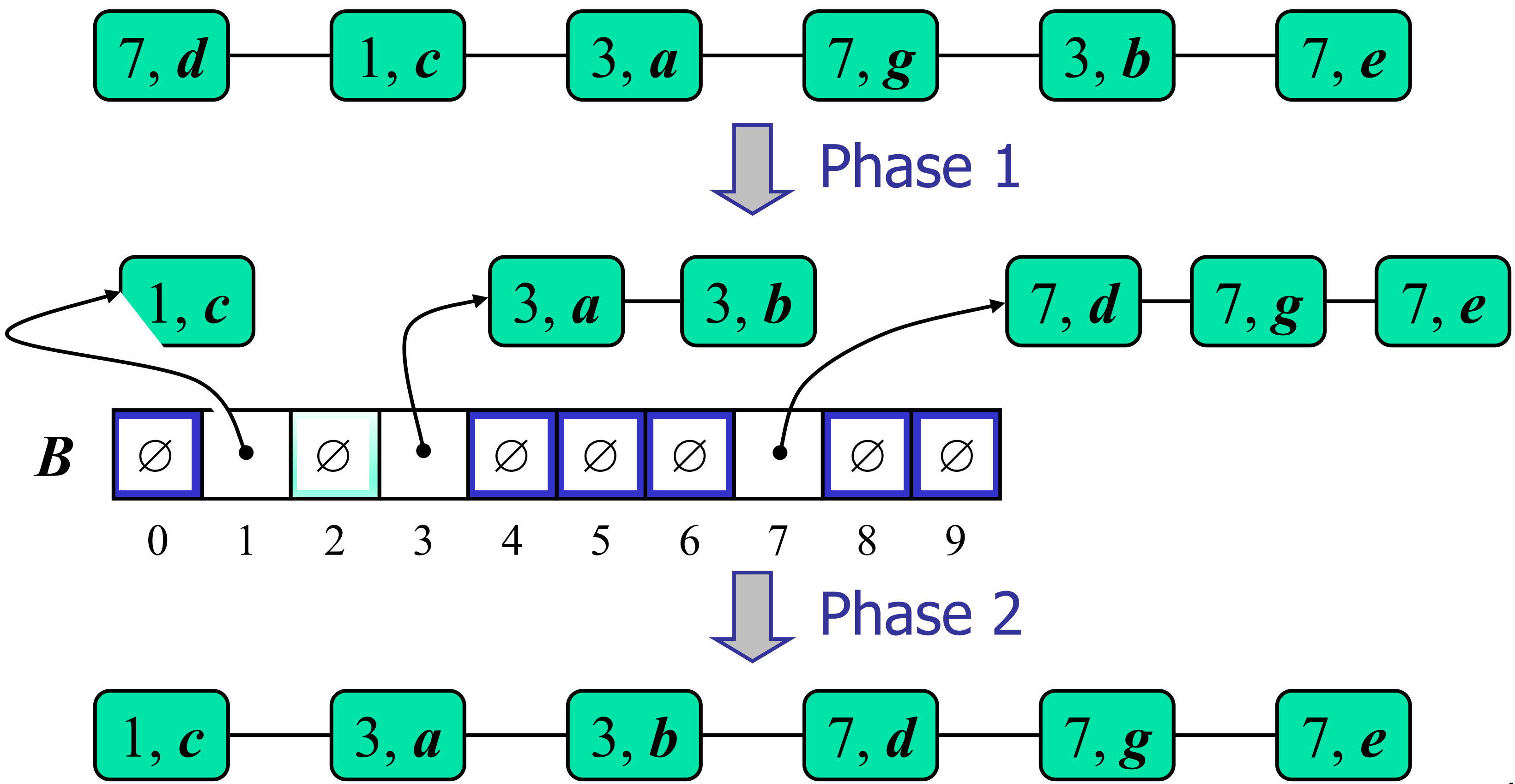
Algorithm *bucketSort(S, N)*

Input sequence S of (key, value) items with keys in the range $[0, N-1]$
Output sequence S sorted by increasing keys

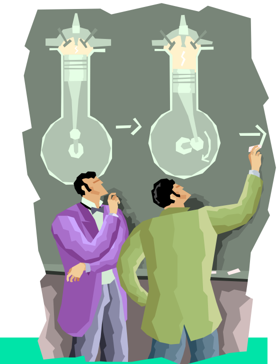
```
{ B = array of N empty sequences;
  while ( ¬ isEmpty(S ) )
     { f = first(S);
       (k, o) = remove(S, f);
       insertLast(B[k], (k, o)); }
   for ( i = 0; i++; i <=N – 1)
      while ( ¬ isEmpty(B[i]))
         {  f = first(B[i]);
            (k, o) = remove(B[i], f);
            insertLast(S, (k, o)); }
}
```

Example

- Key range [0, 9]



Properties and Extensions

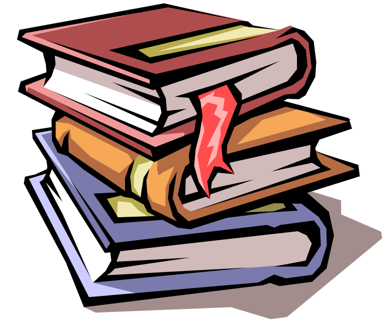


- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
- **Stable** Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm

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 d-tuple is a sequence of d keys $(k_1, k_2, \ldots, k_d)$, where key k_i is said to be the i-th dimension of the tuple
- Example:
  - The Cartesian coordinates of a point in space are a 3-tuple
- The lexicographic order of two d-tuples is recursively defined as follows

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$

$$\Leftrightarrow$$

$$x_1 < y_1 \ \vee \ x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d)$$

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

Lexicographic-Sort

- Let ***stableSort***(***S***) be a stable sorting algorithm
- Lexicographic-sort sorts a sequence of ***d***-tuples in lexicographic order by executing ***d*** times algorithm ***stableSort***, one per dimension, from least significant element to most significant element
- 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 = d; i >=1; i- -; )
       stableSort(S, i);
}
```

Example:

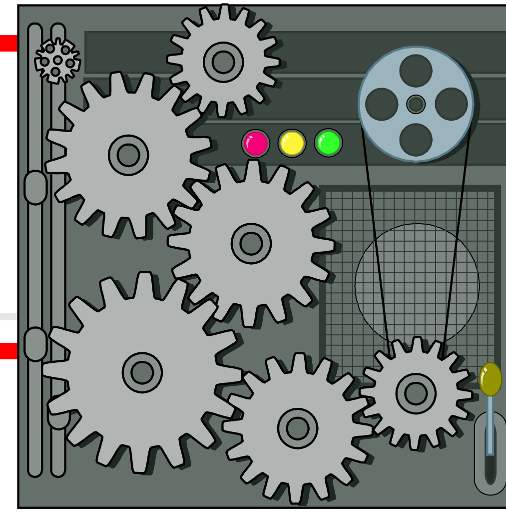
(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

(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 $(0, \ldots, 0) \leq (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \leq (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S

Output sequence S sorted in lexicographic order

```
{  for ( i = d; i>=1; i-- )
      bucketSort(S, N);
}
```

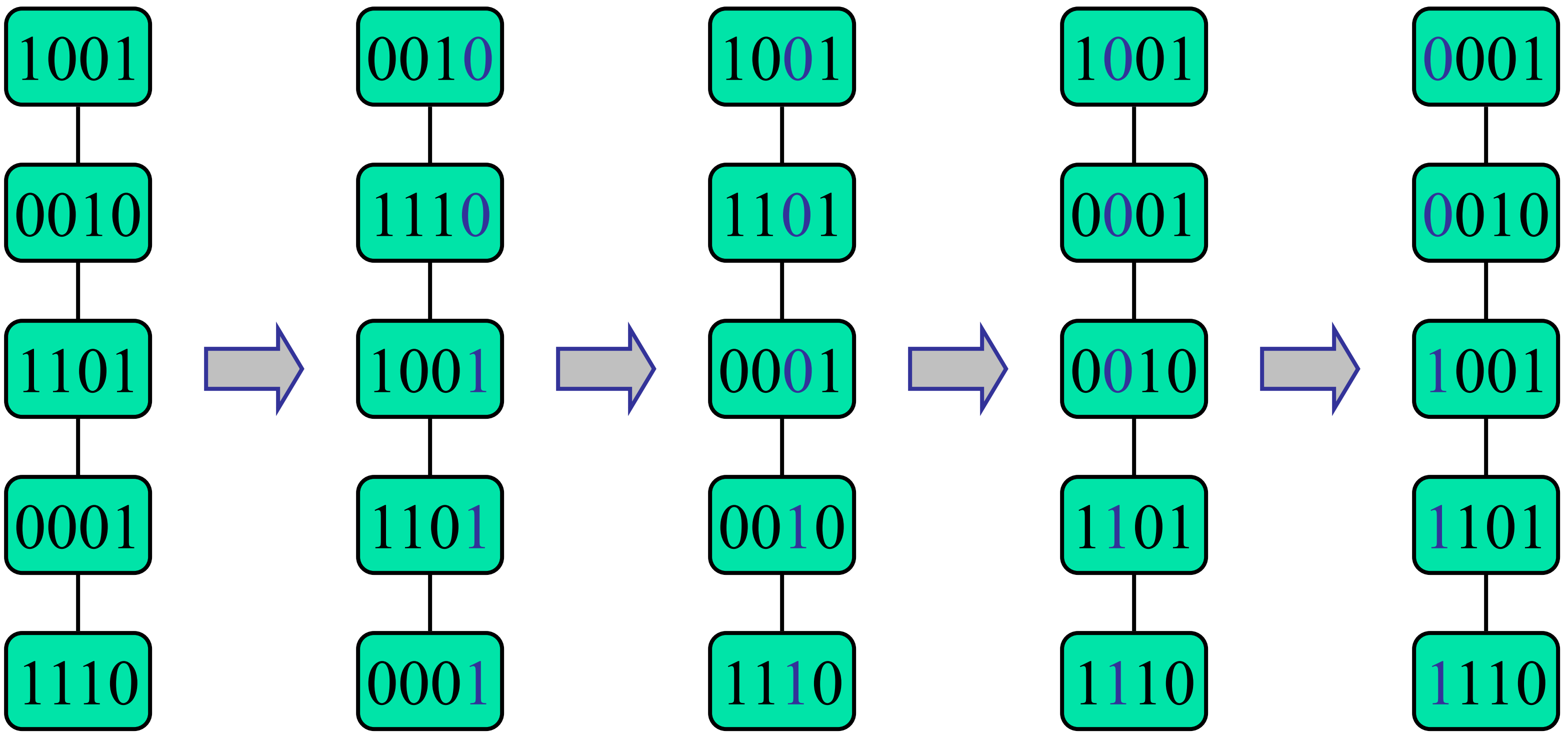
Radix-Sort for Binary Numbers

- Consider a sequence of n b-bit integers
  $$x = x_{b-1} \ldots x_1 x_0$$
- We represent each element as a 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 $O(bn)$ time
- For example, we can sort a sequence of 32-bit integers in linear time

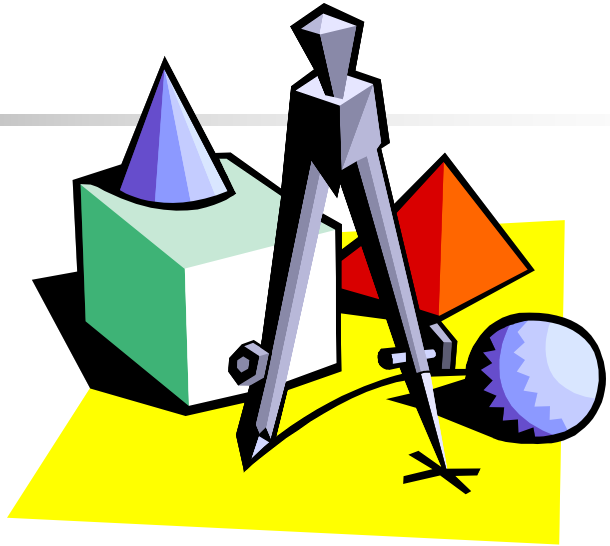
Algorithm *binaryRadixSort*(S)
  Input sequence S of b-bit integers
  Output sequence S sorted
  replace each element x of S with the item $(0, x)$
{ **for** ($i = 0$; i<= b–1; i++)
  { replace the key k of each item (k, x) of S with bit x_i of x;
    bucketSort(S, 2);
  }
}

Example

- Sorting a sequence of 4-bit integers

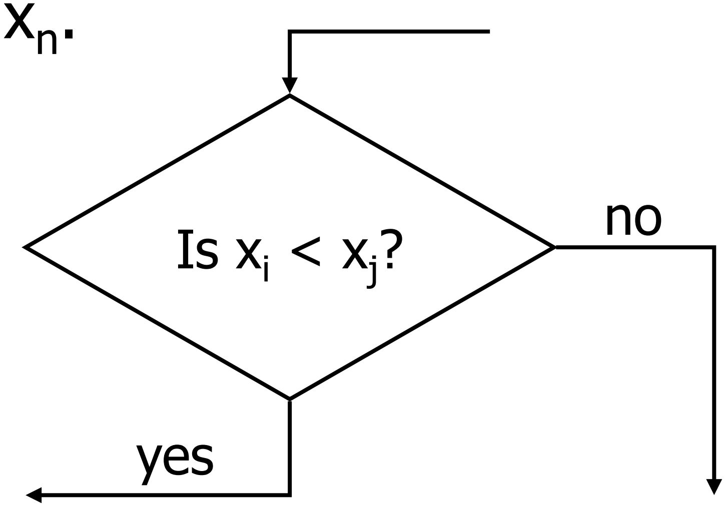


Sorting Lower Bound



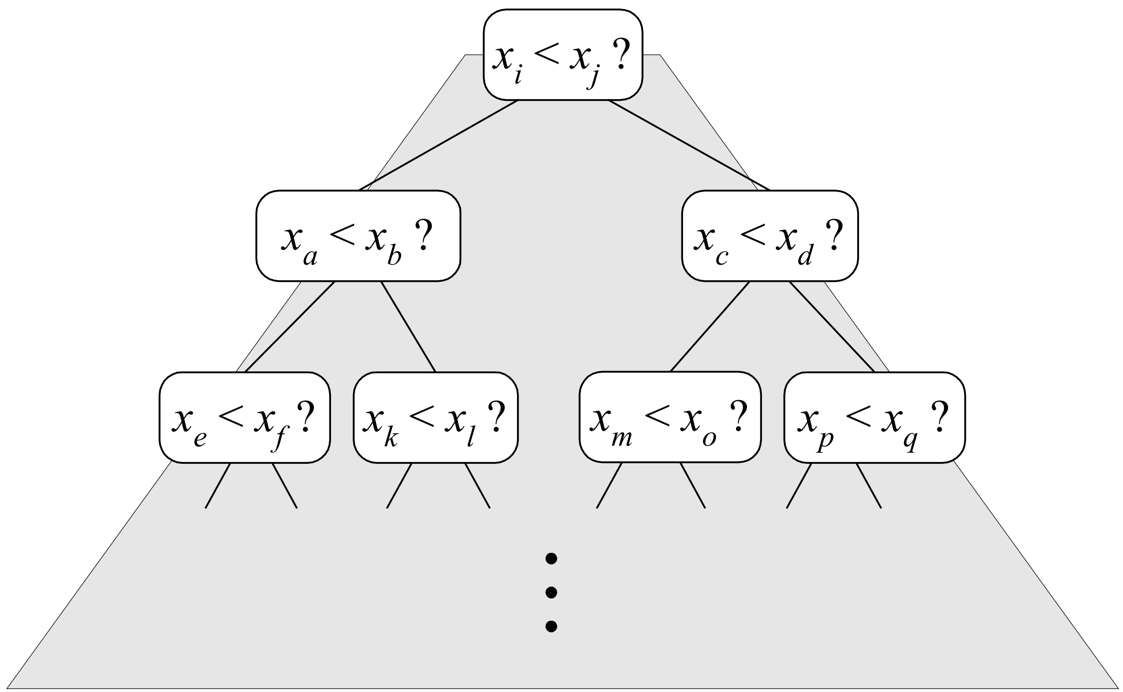
Comparison-Based Sorting

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.



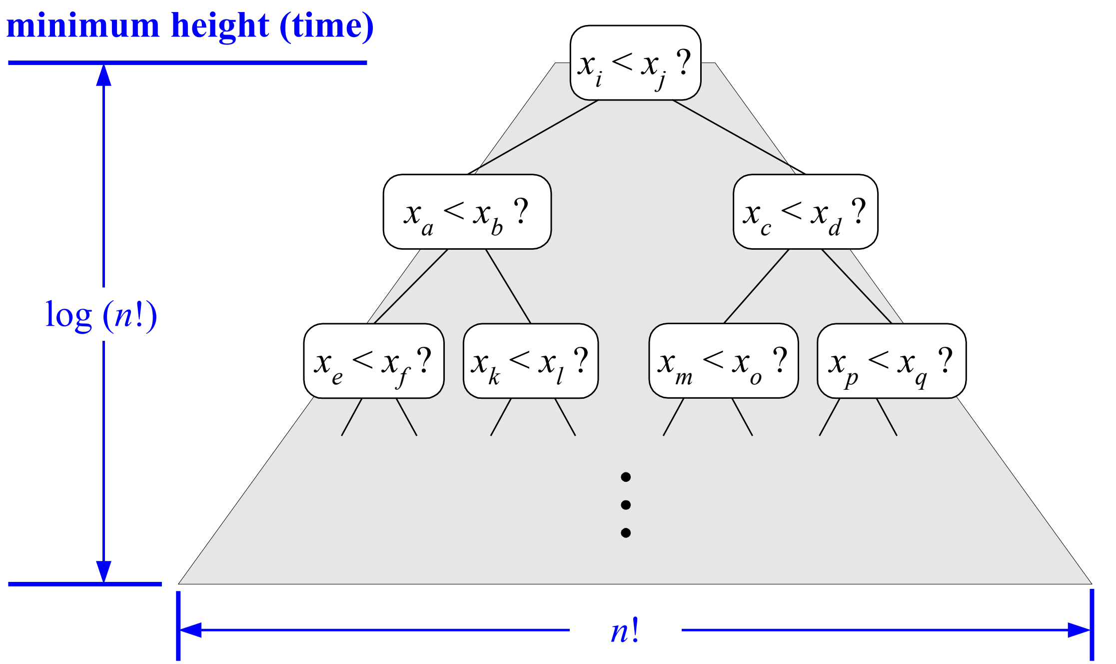
Counting Comparisons

- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a **decision tree**



Decision Tree Height

- The height of this decision tree is a lower bound on the running time
- Every possible input permutation must lead to a separate leaf output.
  - If not, some input …4…5… would have same output ordering as …5…4…, which would be wrong.
- Since there are n!=1*2*…*n leaves, the height is at least log (n!)



The Lower Bound

- Any comparison-based sorting algorithms takes at least log (n!) time
- Therefore, any such algorithm takes time at least

$$\log\,(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2).$$

- That is, any comparison-based sorting algorithm must run in Ω(n log n) time.

Summary

- Merge sort
- Quick sort
- Lexicographic sort
- Bucket sort
- Radix sort

Suggested reading:

Sedgewick, Chapters 7, 8, 10