Data Structures and Algorithms

- sort -

School of Electrical Engineering
Korea University

Sorting

- Sorting problem (non-decreasing)

  Input:

  a sequence of n numbers $(a_1, a_2, \ldots ,a_n)$

  Output:

  a permutation of the input sequence

  $(a'_1, a'_2 ,\ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Sorting

- Simple sort
  - Bubble sort : $O_w(n^2)$ $O_a(n^2)$
  - Insertion sort : $O_w(n^2)$ $O_a(n^2)$
  - Shell sort : $O_w(n^{3/2})$ $O_a(n^{5/4})$
- Complex sort
  - Merge sort : $O_w(nlogn)$ $O_a(nlogn)$
  - Heap sort : $O_w(nlogn)$ $O_a(nlogn)$
  - Quick sort : $O_w(n^2)$ $O_a(nlogn)$

Insertion Sort

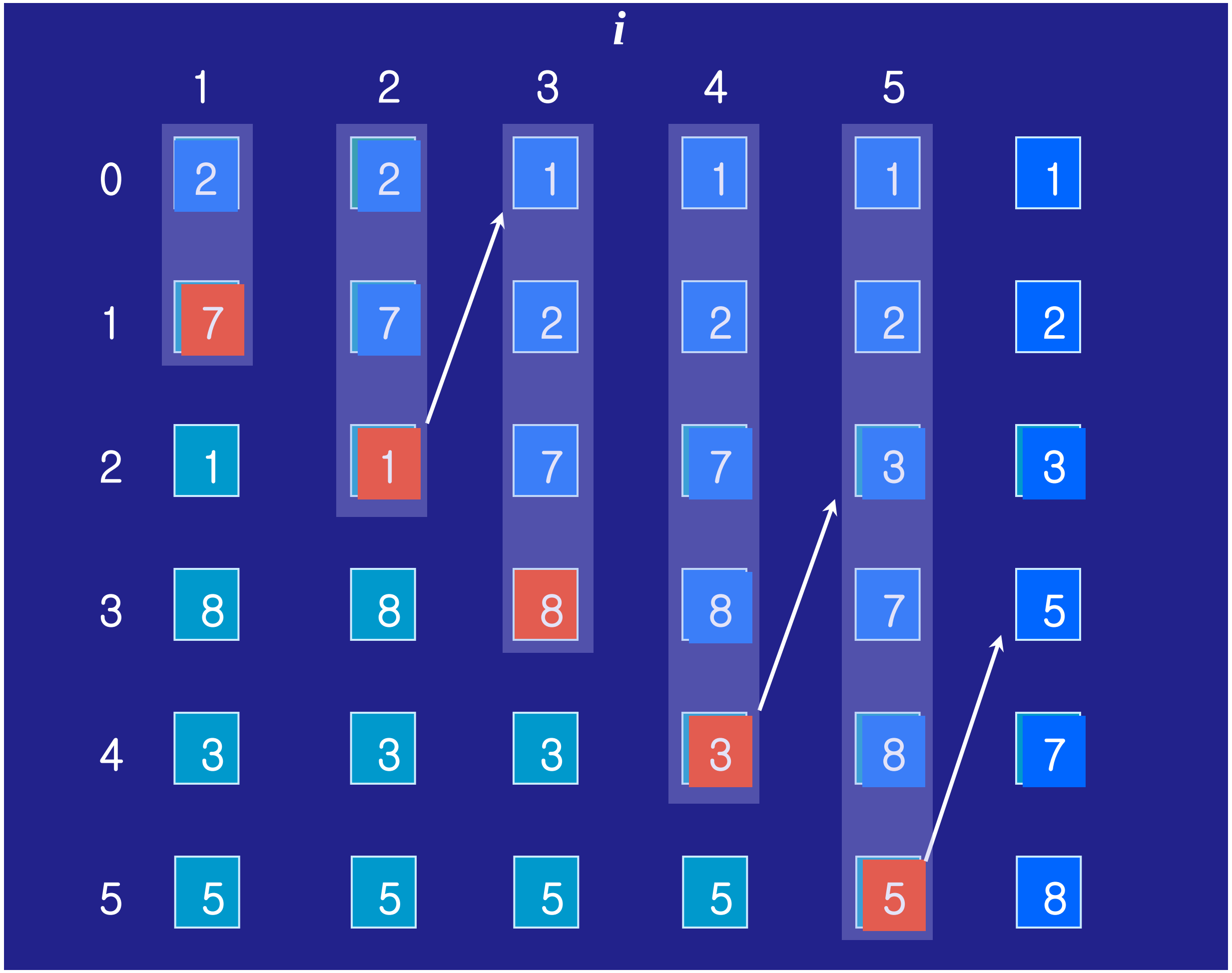
- One of the simplest sorting algorithm
- Consists of N-1 passes
- For pass P = 1 through N-1, the elements in positions 0 through P-1 are already known to be in sorted order
- Two extremes:
  - Input in reverse order
  - Input in presorted order

Insertion Sort

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Figure 7.1 Insertion sort after each pass

Insertion Sort



Insertion Sort Routine

```
        void
        InsertionSort( ElementType A[ ], int N )
        {
            int j, P;

            Element Type Tmp;
/* 1*/      for( P = 1; P < N; P++ )
            {
/* 2*/          Tmp = A[ P ];
/* 3*/          for( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )
/* 4*/              A[ j ] = A[ j - 1 ];
/* 5*/          A[ j ] = Tmp;
            }
        }
```

Shellsort

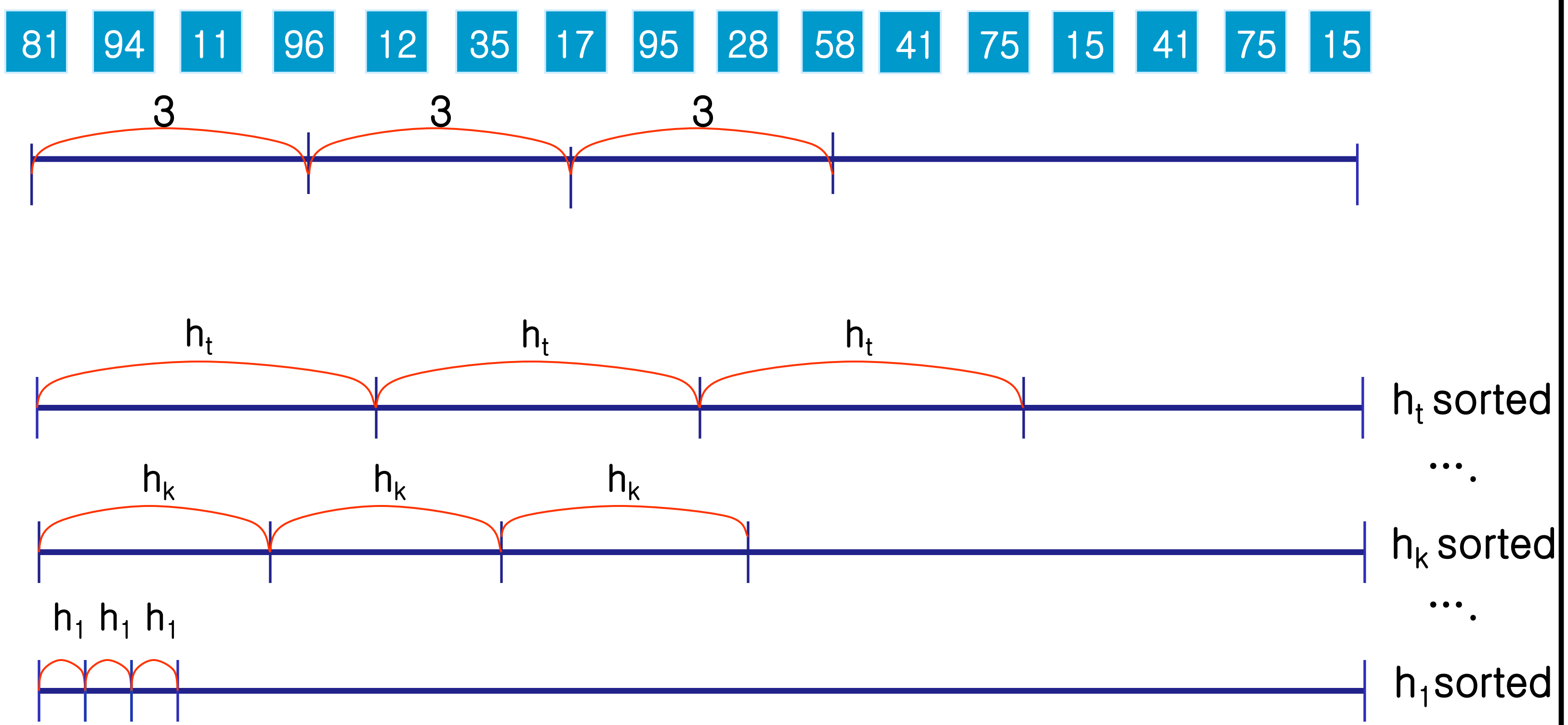
- Named after its inventor, Donald Shell
- It works by comparing distant elements
- The distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared.
- A file is $h_{\mathbf{k}}$–sorted when all elements spaced $h_{\mathbf{k}}$ apart are sorted.

Shellsort

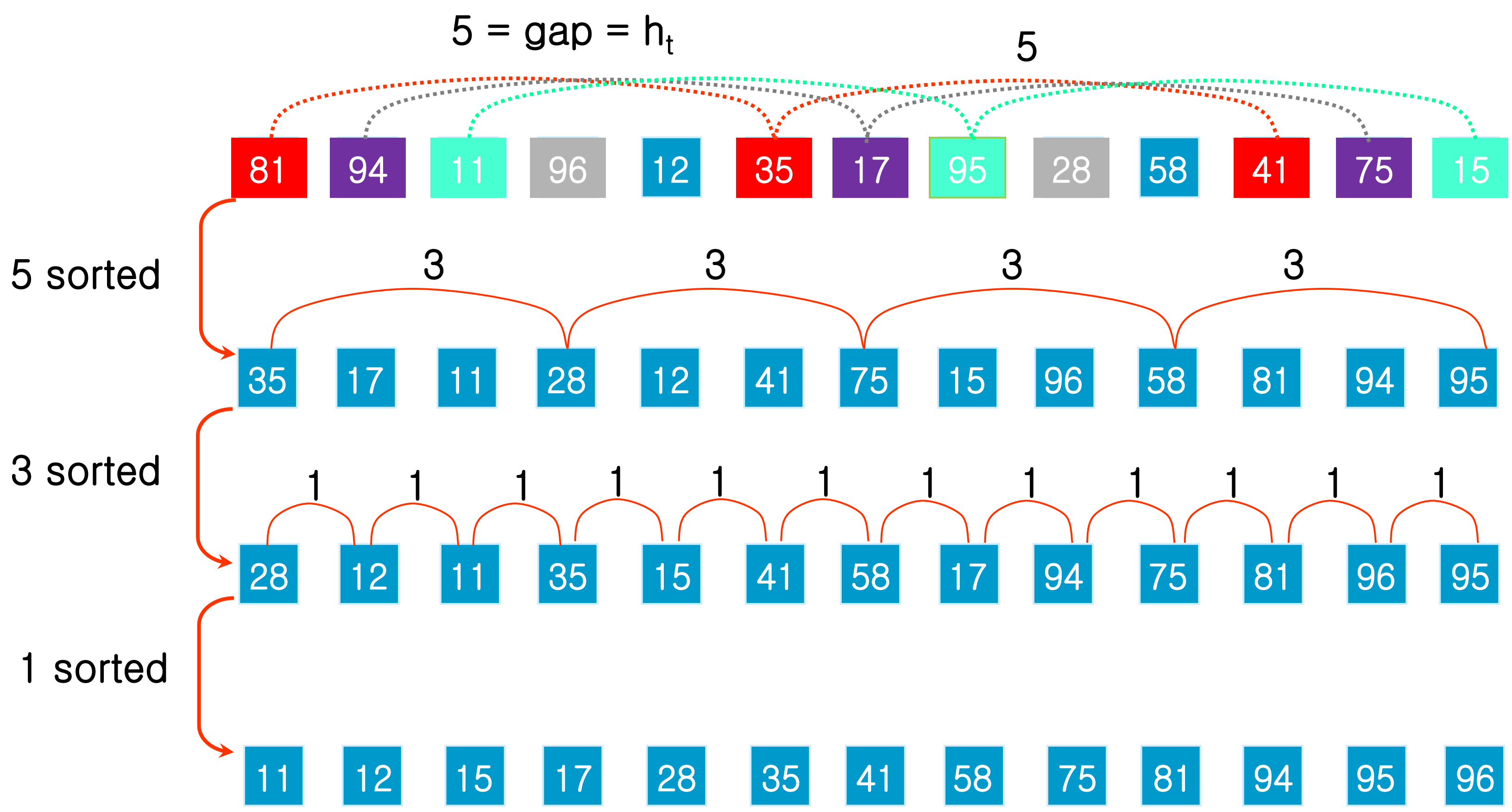
- Use increment sequence, h_1, h_2,⋯,h_t, where h_1 = 1
- Perform an insertion sort on $h_{\mathbf{k}}$ independent subarrays
- After a phase of using h_k, A[i] ≤ A[$i + h_k$] for every i.
- An h_k–sorted file remains h_k–sorted after h_{k-1}–sorting

Shellsort

- Define an increment sequence $h_1, h_2, \ldots, h_t$
- h_k sorted: all elements spaced h_k apart (n/ h_k elements) are sorted



Shellsort



5 = gap = h_t
5
81 94 11 96 12 35 17 95 28 58 41 75 15
5 sorted
3 3 3 3
35 17 11 28 12 41 75 15 96 58 81 94 95
3 sorted
1 1 1 1 1 1 1 1 1 1 1 1
28 12 11 35 15 41 58 17 94 75 81 96 95
1 sorted
11 12 15 17 28 35 41 58 75 81 94 95 96

Shellsort

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

Shellsort Algorithm

```
        void
        Shellsort( ElementType A[ ], int N )
        {
            int i, j, Increment;
            ElementType Tmp;

/* 1*/      for( Increment = N / 2; Increment > 0; Increment /= 2 )
/* 2*/          for( i = Increment; i < N; i++ )
                {
/* 3*/              Tmp = A[ i ];
/* 4*/              for( j = i; j >= Increment; j -= Increment )
/* 5*/                  if( Tmp < A[ j - Increment ] )
/* 6*/                      A[ j ] = A[ j - Increment ];
                        else
/* 7*/                      break;
/* 8*/              A[ j ] = Tmp;
                }
        }
```

Shellsort Algorithm

Start	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 7.5 Bad case for Shellsort with Shell's increments

Heapsort

- Based on the priority queue
- Build a binary heap of N elements and perform N *DeleteMin* operations
- Smaller one first. Where to put?
  - Use extra array
  - In-place method, whose result is in
- Use (*max*)heap for increasing sorted order

Heapsort

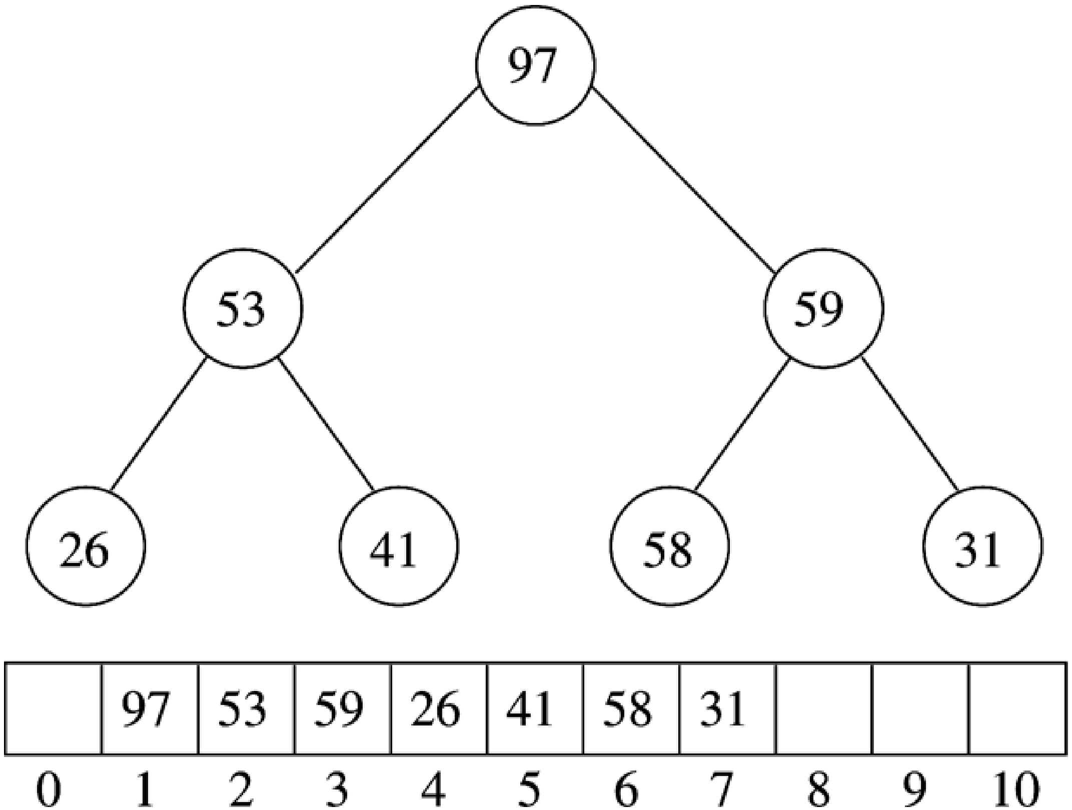


Figure 7.6 (*Max*) heap after *BuildHeap* phase

Heapsort

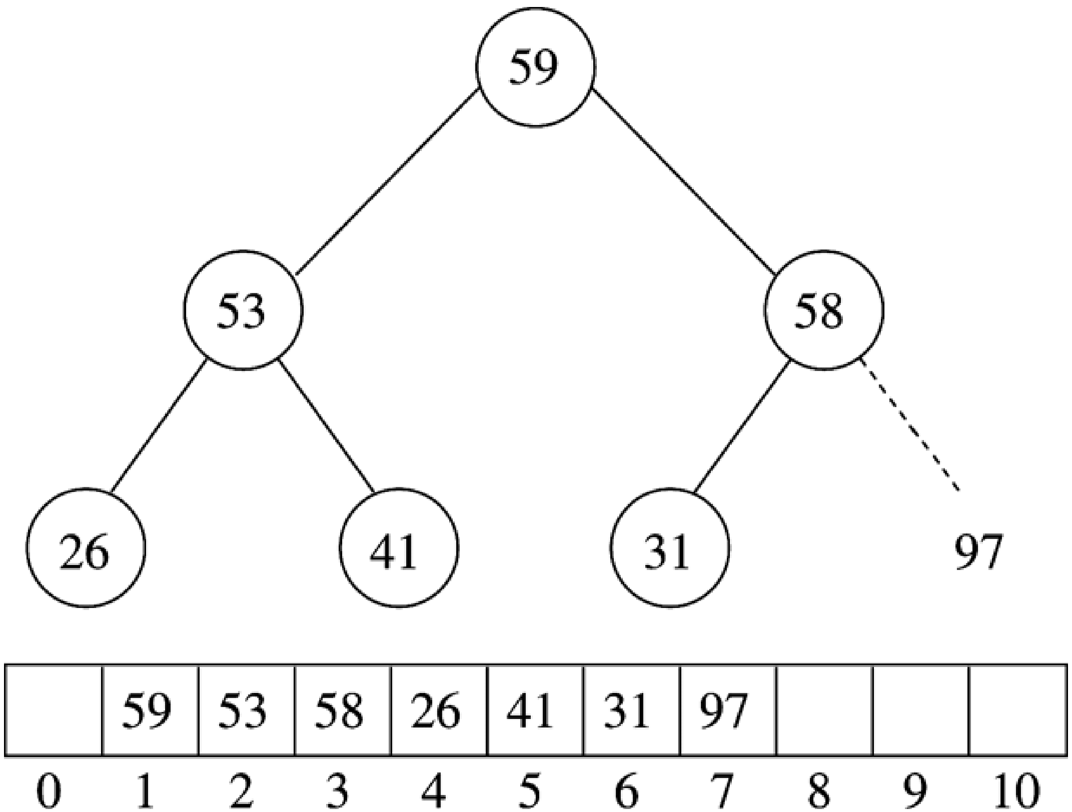


Figure 7.7 Heap after first *DeleteMax*

Heapsort algorithm

```
        void
        Heapsort( ElementType A[ ], int N )
        {
            int i;

/* 1*/      for( i = N / 2; i >= 0; i-- ) /* BuildHeap */
/* 2*/          PercDown( A, i, N );
/* 3*/      for( i = N - 1; i > 0; i-- )
            {
/* 4*/          Swap( &A[ 0 ], &A[ i ] ); /* DeleteMax */
/* 5*/          PercDown( A, 0, i );
            }
        }
```

PercDown routine

```
        void
        PercDown( ElementType A[ ], int i, int N )
        {
            int Child;
            ElementType Tmp;

/* 1*/      for( Tmp = A[ i ]; LeftChild( i ) < N; i = Child )
            {
/* 2*/          Child = LeftChild( i );
/* 3*/          if( Child != N - 1 && A[ Child + 1 ] > A[ Child ] )
/* 4*/              Child++;
/* 5*/          if( Tmp < A[ Child ] )
/* 6*/              A[ i ] = A[ Child ];
                else
/* 7*/              break;
            }
/* 8*/      A[ i ] = Tmp;
        }
```

Bubble sort

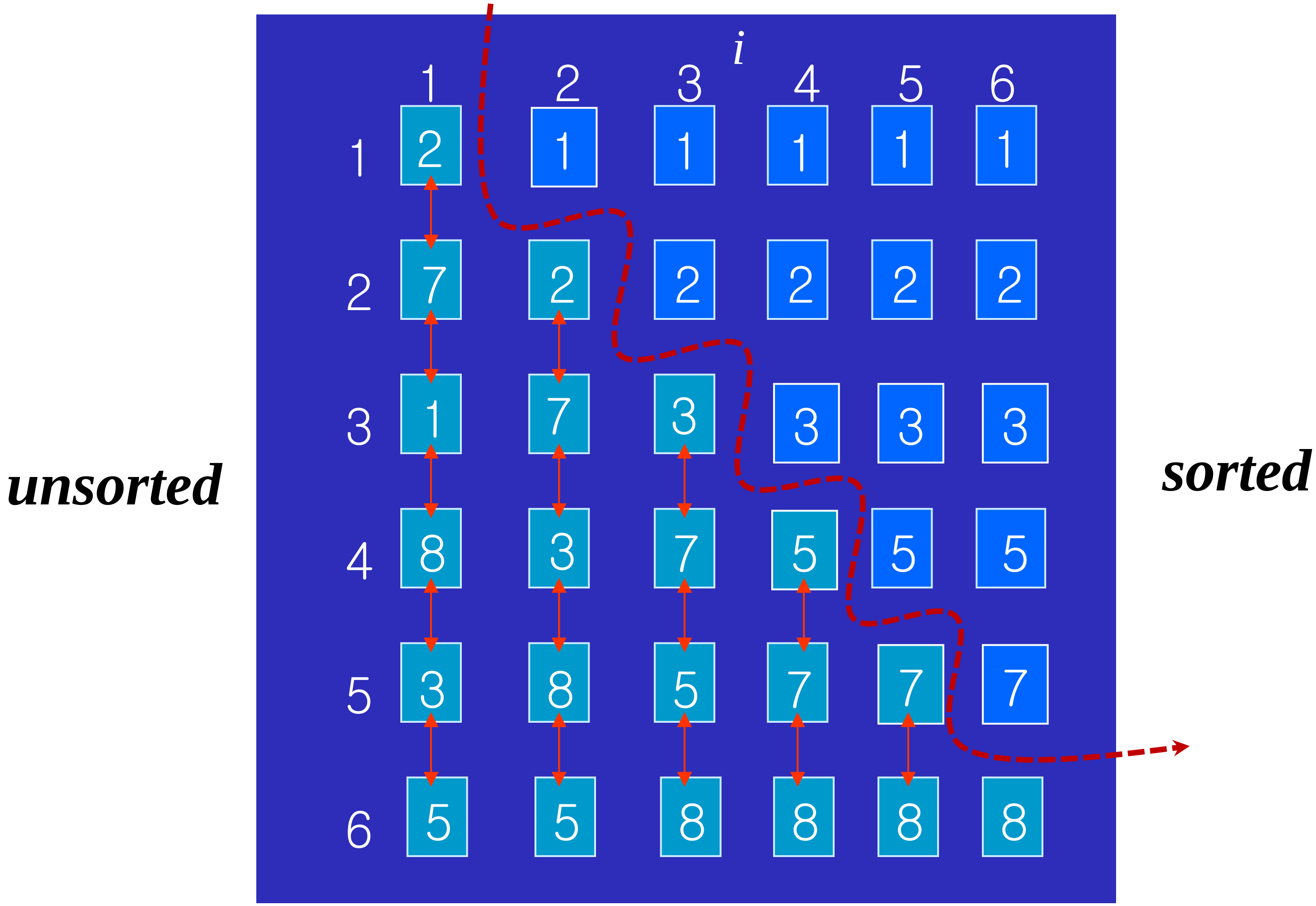
- Smallest data in its place sequentially
- Requires N-1 passes for partially sorted remaining data
- After K passes, K smallest data in their places

Bubble sort

```
for ( i = 1; i <= n-1; i++ ) do
   place the smallest element from
   A[i] to A[n] into A[i]
```

```
for (i=1; i ≤ n-1; i++)
       for (j=n; j ≥ i+1; j--)
            if (A[j-1] > A[j])
              swap A[j-1]& A[j]
```

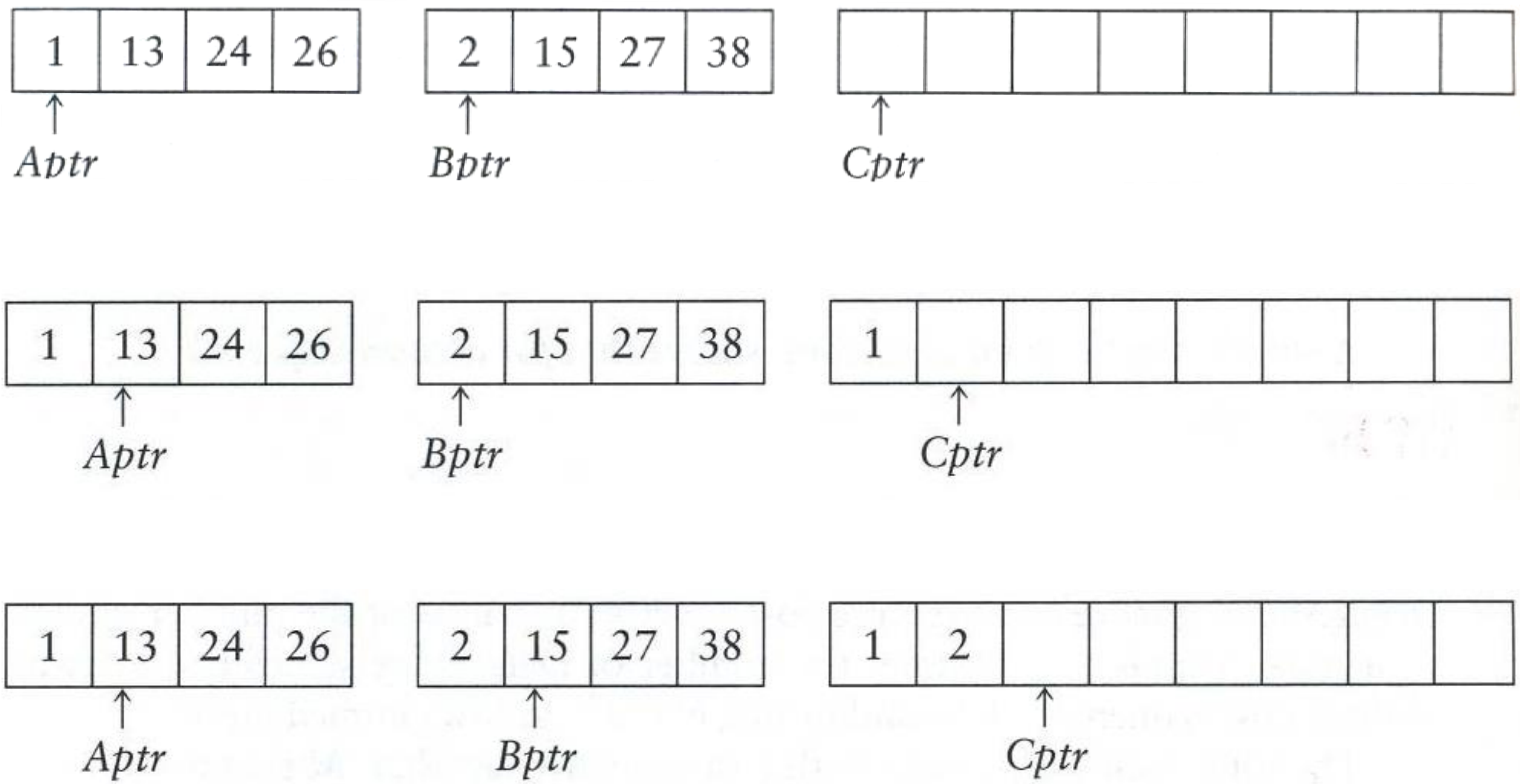
Bubble sort


i
1 2 3 4 5 6
1 2 1 1 1 1 1
2 7 2 2 2 2 2
3 1 7 3 3 3 3
4 8 3 7 5 5 5
5 3 8 5 7 7 7
6 5 5 8 8 8 8
unsorted
sorted

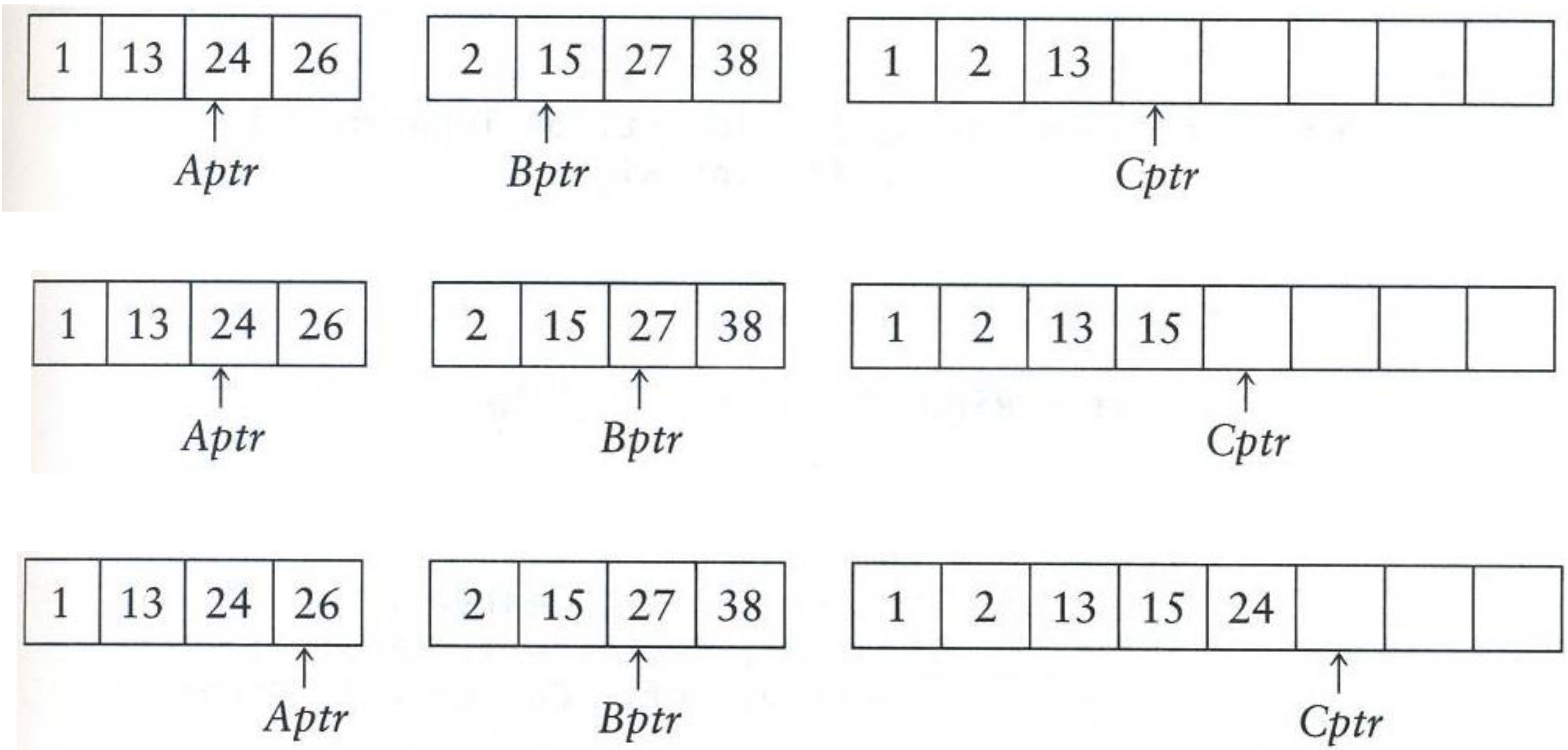
Mergesort

- Runs in $O(N \log N)$ worst-case running time
- The fundamental operation is merging two *sorted* lists.
- Uses a classic divide-and-conquer strategy

Mergesort

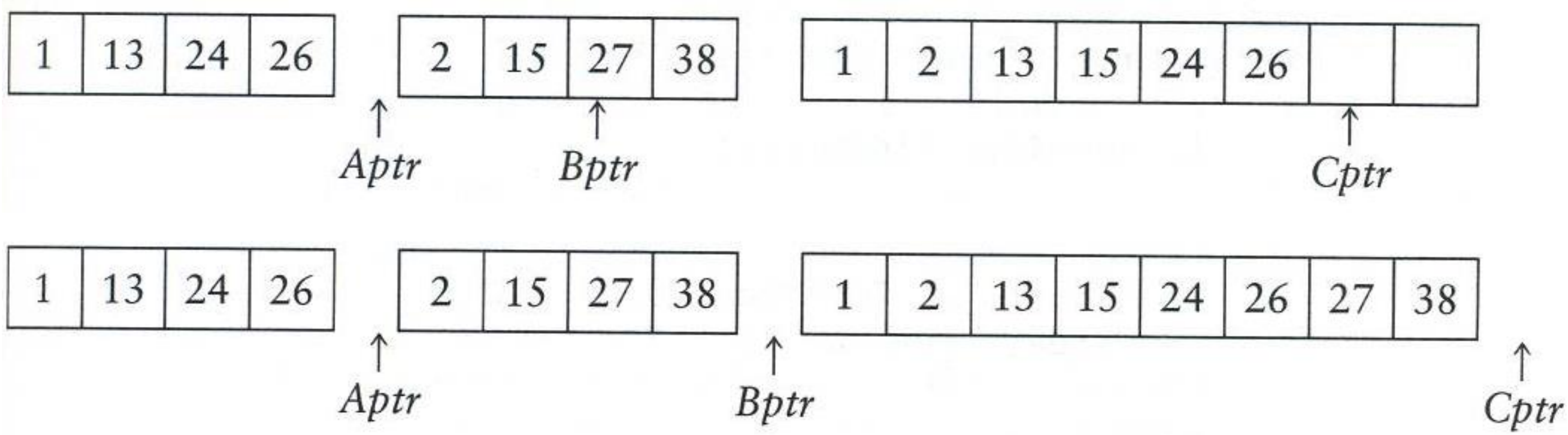

1 13 24 26
Aptr
2 15 27 38
Bptr
Cptr
1 13 24 26
Aptr
2 15 27 38
Bptr
1
Cptr
1 13 24 26
Aptr
2 15 27 38
Bptr
1 2
Cptr

Mergesort

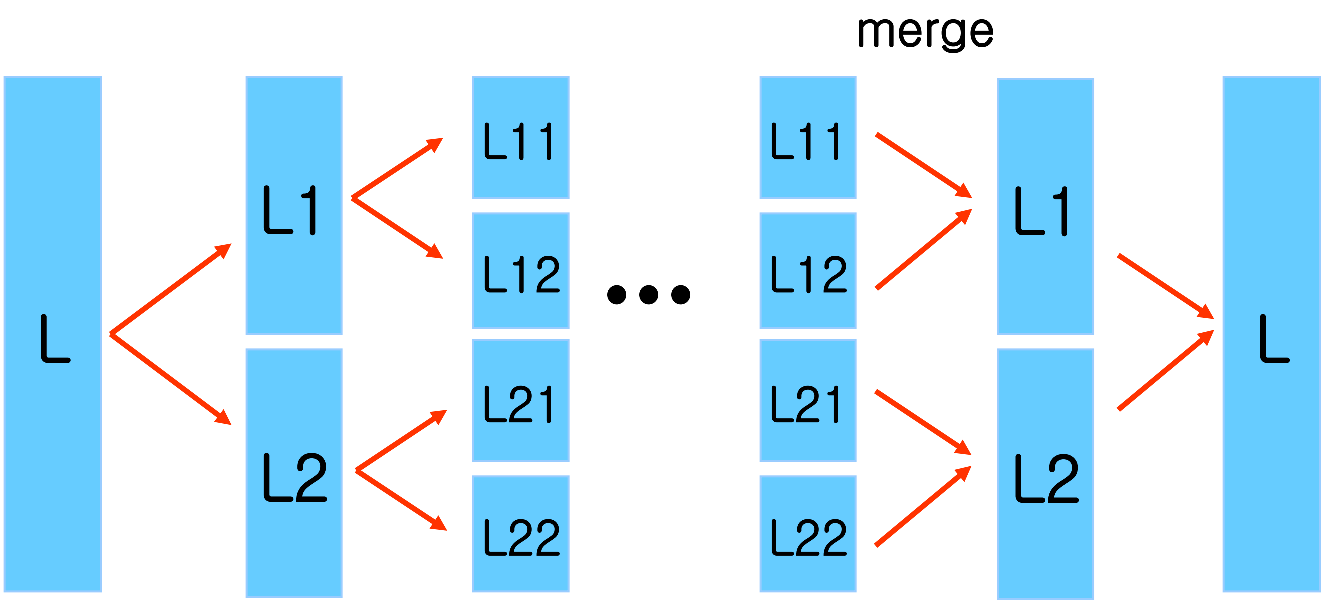


1 13 24 26
Aptr
2 15 27 38
Bptr
1 2 13
Cptr
1 13 24 26
Aptr
2 15 27 38
Bptr
1 2 13 15
Cptr
1 13 24 26
Aptr
2 15 27 38
Bptr
1 2 13 15 24
Cptr

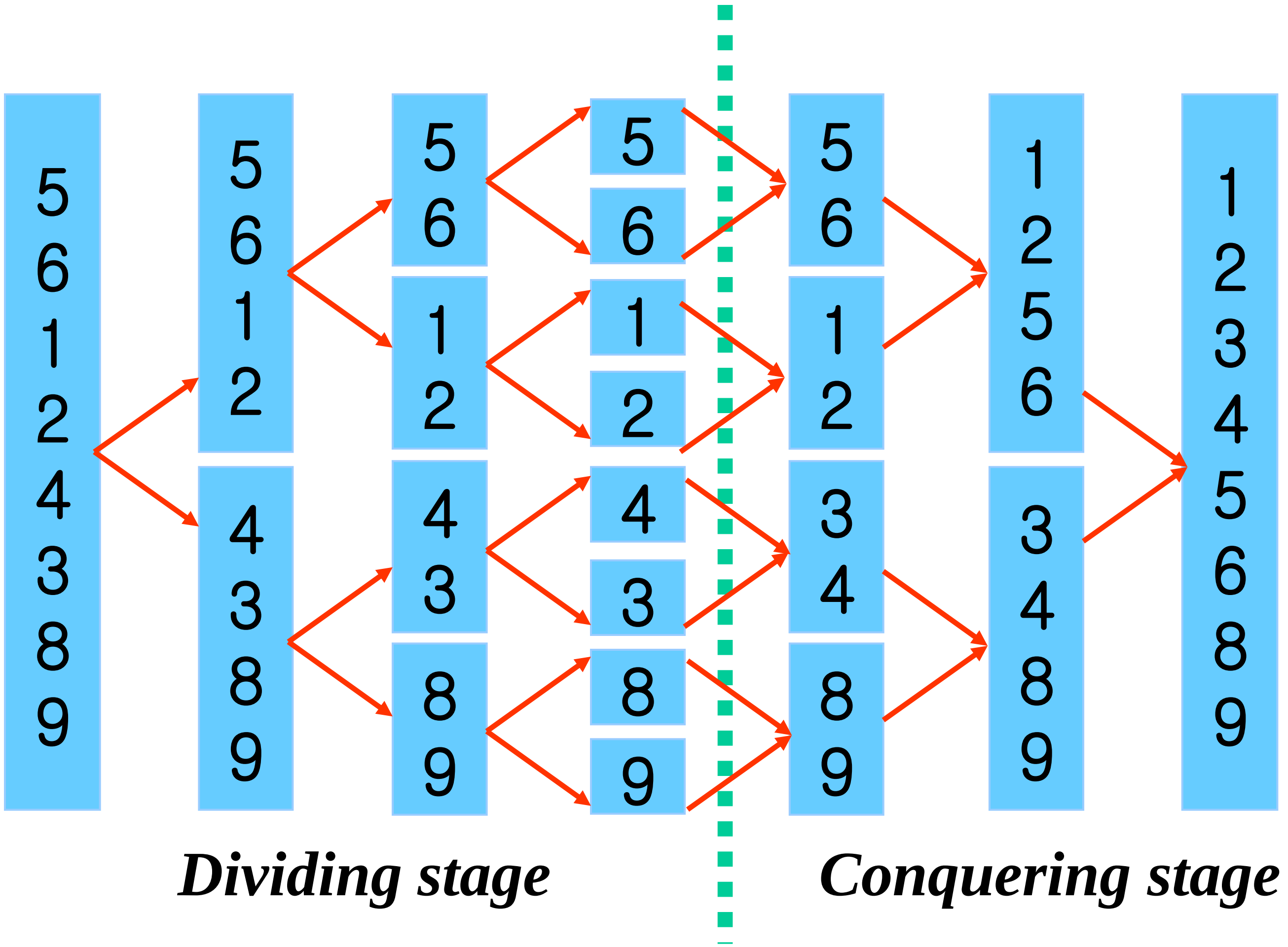
Mergesort


1 13 24 26
2 15 27 38
1 2 13 15 24 26
Aptr
Bptr
Cptr
1 13 24 26
2 15 27 38
1 2 13 15 24 26 27 38
Aptr
Bptr
Cptr

Mergesort



Mergesort



5 6 1 2 4 3 8 9
5 6 1 2
4 3 8 9
5 6
1 2
4 3
8 9
5
6
1
2
4
3
8
9
5 6
1 2
3 4
8 9
1 2 5 6
3 4 8 9
1 2 3 4 5 6 8 9
Dividing stage
Conquering stage

Mergesort algorithm

```
void
Mergesort( ElementType A[ ], int N )
{
    ElementType *TmpArray;

    TmpArray = malloc( N * sizeof( ElementType ) );
    if( TmpArray != NULL )
    {
        MSort( A, TmpArray, 0, N - 1 );
        free( TmpArray );
    }
    else
        FatalError( "No space for tmp array!!!" );
}
```

MSort routine

```
void
MSort( ElementType A[ ], ElementType TmpArray[ ],
                int Left, int Right )
{
    int Center;

    if( Left < Right )
    {
        Center = ( Left + Right ) / 2;
        MSort( A, TmpArray, Left, Center );
        MSort( A, TmpArray, Center + 1, Right );
        Merge( A, TmpArray, Left, Center + 1, Right );
    }
}
```

Merge algorithm

```
/* Lpos = start of left half, Rpos = start of right half */

void
Merge( ElementType A[ ], ElementType TmpArray[ ],
                int Lpos, int Rpos, int RightEnd )
{
    int i, LeftEnd, NumElements, TmpPos;

    LeftEnd = Rpos - 1;
    TmpPos = Lpos;
    NumElements = RightEnd - Lpos + 1;

    /* main loop */
    while( Lpos <= LeftEnd && Rpos <= RightEnd )
        if( A[ Lpos ] <= A[ Rpos ] )
            TmpArray[ TmpPos++ ] = A[ Lpos++ ];
        else
            TmpArray[ TmpPos++ ] = A[ Rpos++ ];

    while( Lpos <= LeftEnd )  /* Copy rest of first half */
        TmpArray[ TmpPos++ ] = A[ Lpos++ ];
    while( Rpos <= RightEnd ) /* Copy rest of second half */
        TmpArray[ TmpPos++ ] = A[ Rpos++ ];

    /* Copy TmpArray back */
    for( i = 0; i < NumElements; i++, RightEnd-- )
        A[ RightEnd ] = TmpArray[ RightEnd ];
}
```

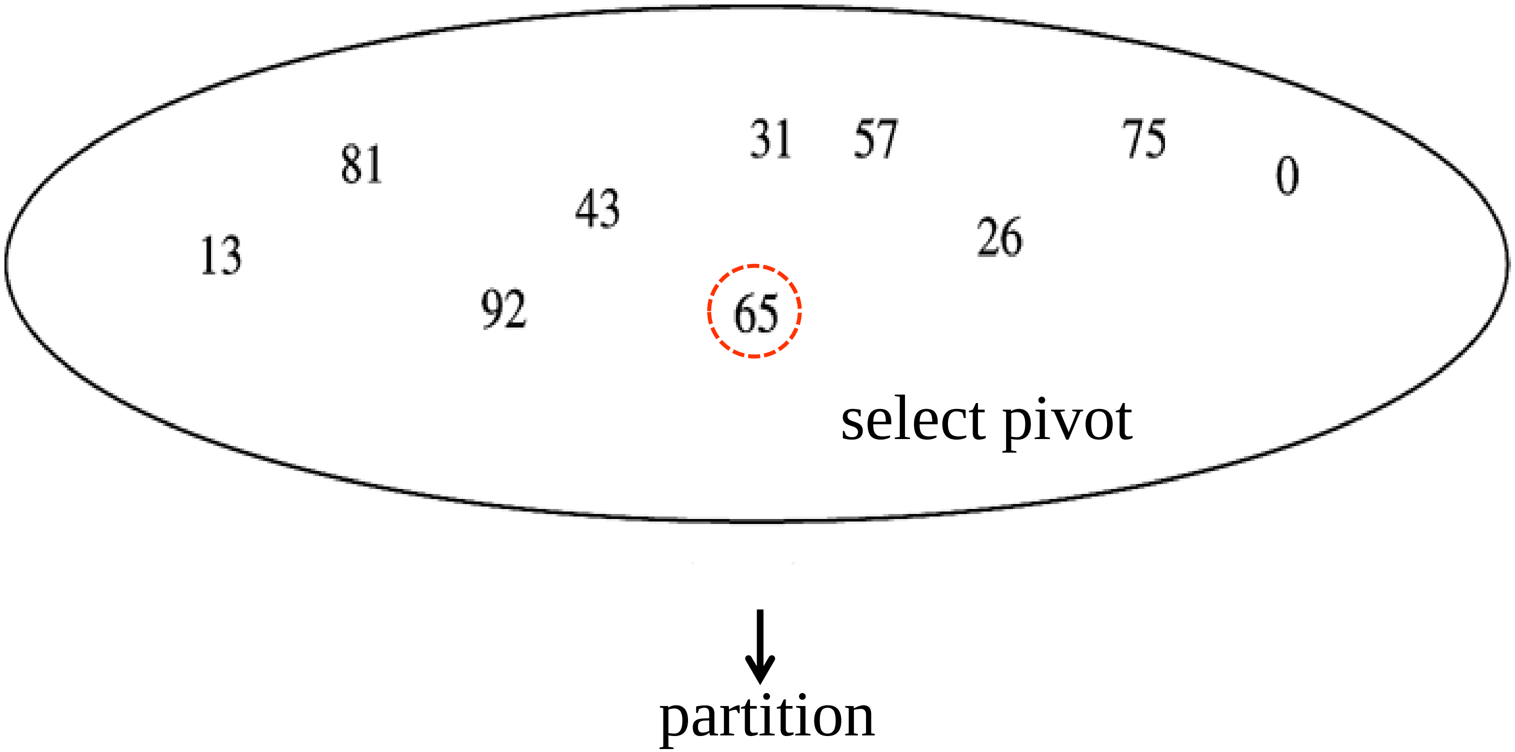
Quicksort algorithm

- The fastest known sorting algorithm in practice.
- Average running time is $O(N \log N)$
- $O(N^2)$ worst-case performance
- A divide-and-conquer like mergesrot

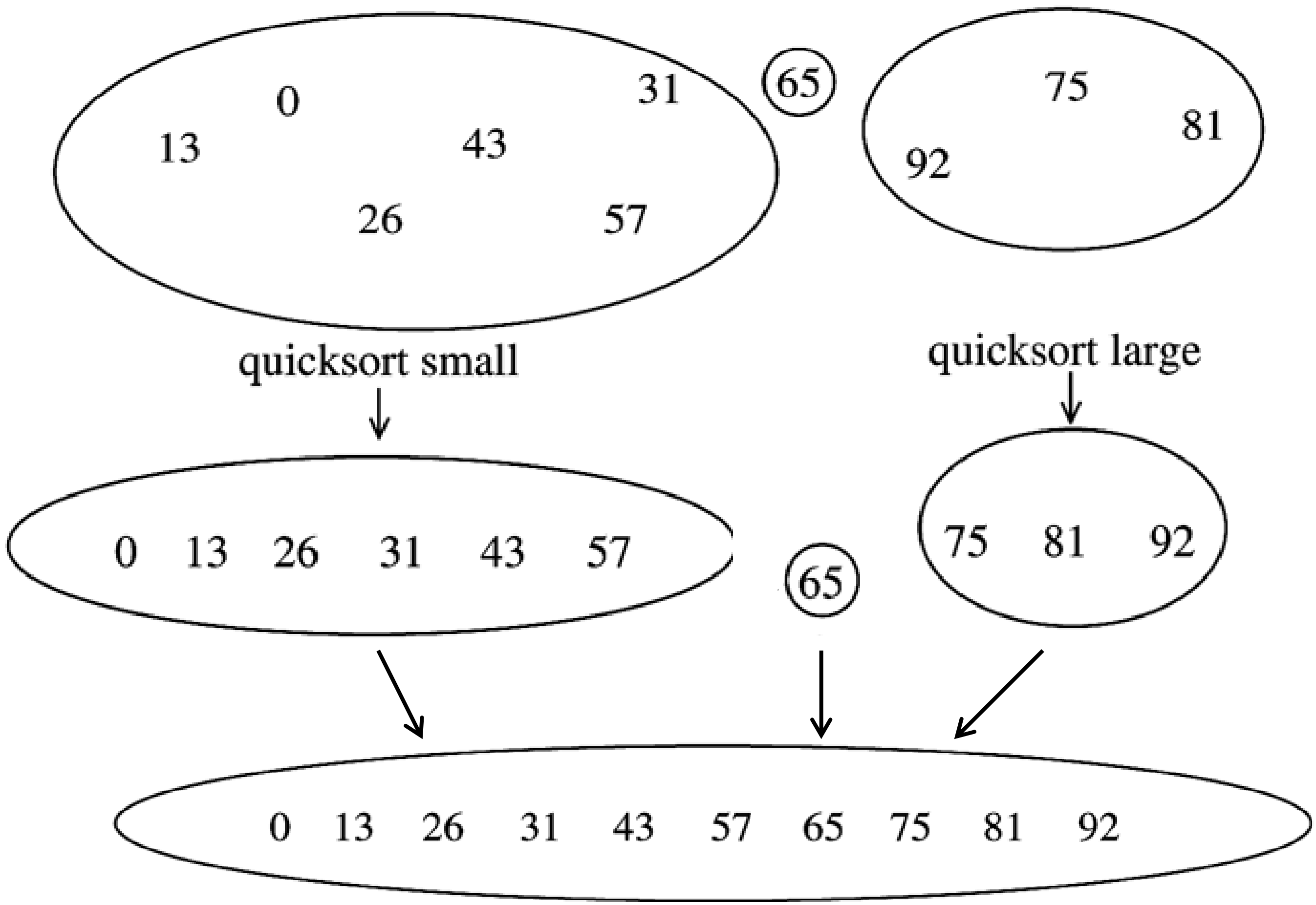
To quicksort an array S

1. If the number of elements in S is 0 or 1, then return.
2. Pick any element v in S called *pivot.*
3. Partition $S - \{v\}$ into two disjoint groups:
   $S_1 = \{x \in S - \{v\} \mid x \leq v\}$, and
   $S_2 = \{x \in S - \{v\} \mid x \geq v\}$.
4. Return {quicksort(S_1 followed by v followed by quicksort(S_2)}.

Quicksort algorithm



Quicksort algorithm



Quicksort algorithm

pivot

8	1	4	9	0	3	5	2	7	6
↑								↑	
i								*j*	

8	1	4	9	0	3	5	2	7	6
↑							↑		
i							*j*		

After First Swap

2	1	4	9	0	3	5	8	7	6
↑							↑		
i							*j*		

Quicksort algorithm

After First Swap									
2	1	4	9	0	3	5	8	7	6
↑							↑		
i							*j*		

Before Second Swap									
2	1	4	9	0	3	5	8	7	6
			↑			↑			
			i			*j*			

After Second Swap									
2	1	4	5	0	3	9	8	7	6
			↑			↑			
			i			*j*			

Quicksort algorithm

After Second Swap

2	1	4	5	0	3	9	8	7	6
			↑			↑			
			i			*j*			

Before Third Swap

2	1	4	5	0	3	9	8	7	6
					↑	↑			
					j	*i*			

After Swap with Pivot

2	1	4	5	0	3	6	8	7	9
						↑			↑
						i			pivot

Quicksort algorithm

Figure 7.12 Driver for quicksort

```
void
Quicksort( ElementType A[ ], int N )
{
    Qsort( A, 0, N - 1 );
}
```

Picking the pivot

1. Use the first element as pivot: popular and uninformed choice. Acceptable if the input is random, but poor if the input is presorted.
2. Choose the pivot randomly. Safe unless the random number generator has a flaw, which is quite common
3. Median-of-Three Partitioning: substitute for the best choice of the median of the array, which is expensive to calculate

Small arrays

- For very small arrays (N <= 20), quicksort does not perform as well as insertion sort.
- Do not use quicksort recursively for small arrays, but instead use a insertion sort.
- Save about 15 % in the running time.
- A good cutoff range is N = 10, although any cutoff between 5 and 20 is likely to produce similar results.

Quicksort algorithm

```
/* Return median of Left, Center, and Right */
/* Order these and hide the pivot */

ElementType
Median3( ElementType A[ ], int Left, int Right )
{
    int Center = ( Left + Right ) / 2;

    if( A[ Left ] > A[ Center ] )
        Swap( &A[ Left ], &A[ Center ] );
    if( A[ Left ] > A[ Right ] )
        Swap( &A[ Left ], &A[ Right ] );
    if( A[ Center ] > A[ Right ] )
        Swap( &A[ Center ], &A[ Right ] );

    /* Invariant: A[ Left ] <= A[ Center ] <= A[ Right ] */

    Swap( &A[ Center ], &A[ Right - 1 ] );  /* Hide pivot */
    return A[ Right - 1 ];                 /* Return pivot */
}
```

Actual quicksort routines

- For pivot selection, sort A[Left], A[Right], and A[Center] in place
- Place A[center] into A[Right – 1] as pivot.
- Hence, we can initialize *i* to Left + 1, *j* to Right – 2, which gives marginal improvement.

Quicksort algorithm

```
        Qsort( ElementType A[ ], int Left, int Right )
        {
            int i, j;
            ElementType Pivot;

/* 1*/      if( Left + Cutoff <= Right )
            {
/* 2*/          Pivot = Median3( A, Left, Right );
/* 3*/          i = Left; j = Right - 1;
/* 4*/          for( ; ; )
                {
/* 5*/              while( A[ ++i ] < Pivot ){ }
/* 6*/              while( A[ --j ] > Pivot ){ }
/* 7*/              if( i < j )
/* 8*/                  Swap( &A[ i ], &A[ j ] );
                    else
/* 9*/                  break;
                }
/*10*/          Swap( &A[ i ], &A[ Right - 1 ] ); /* Restore

/*11*/          Qsort( A, Left, i - 1 );
/*12*/          Qsort( A, i + 1, Right );
            }
            else /* Do an insertion sort on the subarray */
/*13*/          InsertionSort( A + Left, Right - Left + 1 );
        }
```

```
/* 3*/  i = Left + 1; j = Right - 2;
/* 4*/  for( ; ; )
        {
/* 5*/      while( A[ i ] < Pivot ) i++;
/* 6*/      while( A[ j ] > Pivot ) j--;
/* 7*/      if( i < j )
/* 8*/          Swap( &A[ i ], &A[ j ] );
            else
/* 9*/          break;
        }
```

Figure 7.15 A small change to quicksort, which breaks the algorithm

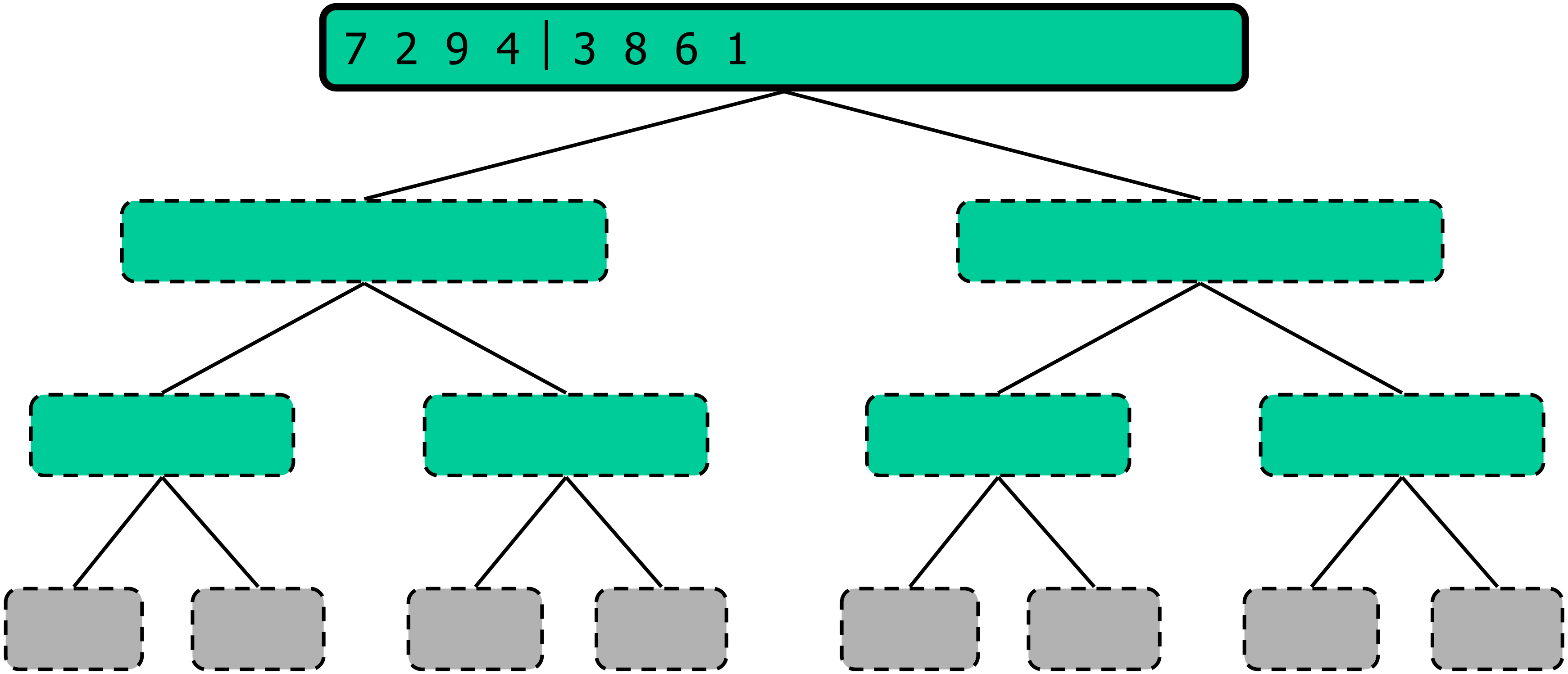
Homework

- Exercises for chap 7.
  2, 4, 7, 9, 11, 12.a, 15, 17, 18
  Due Nov. 11.

- Quiz on Nov. 11. Example on the next slide.

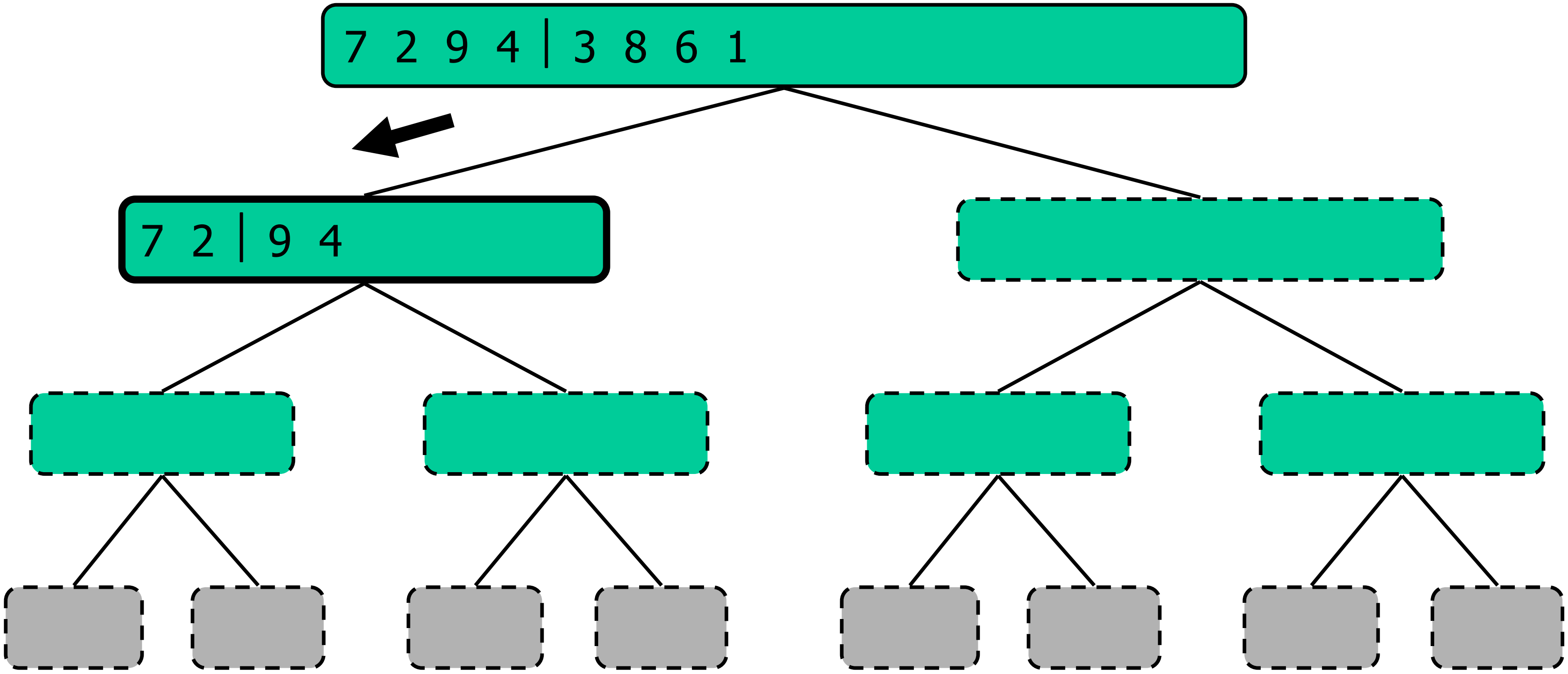
Mergesort: Execution Example

- Partition



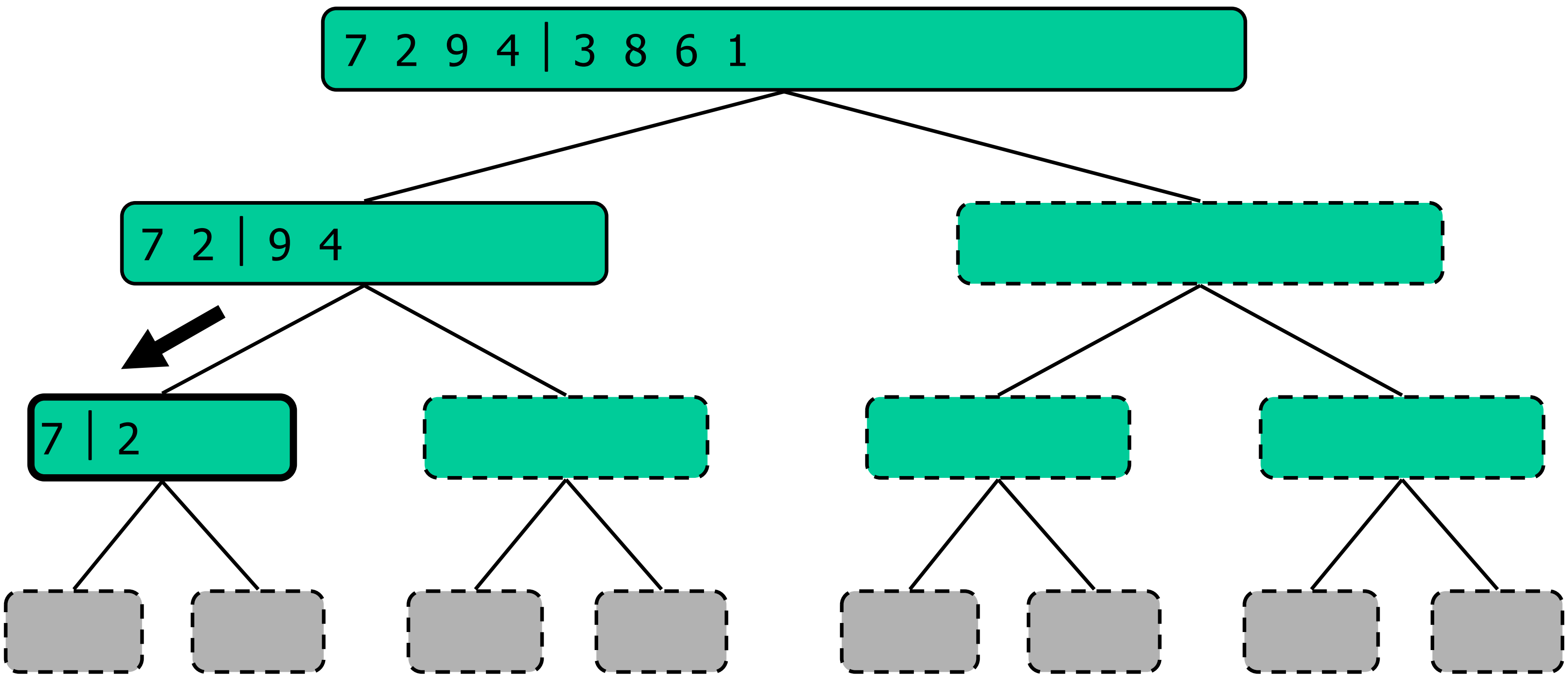
Execution Example (cont.)

- Recursive call, partition



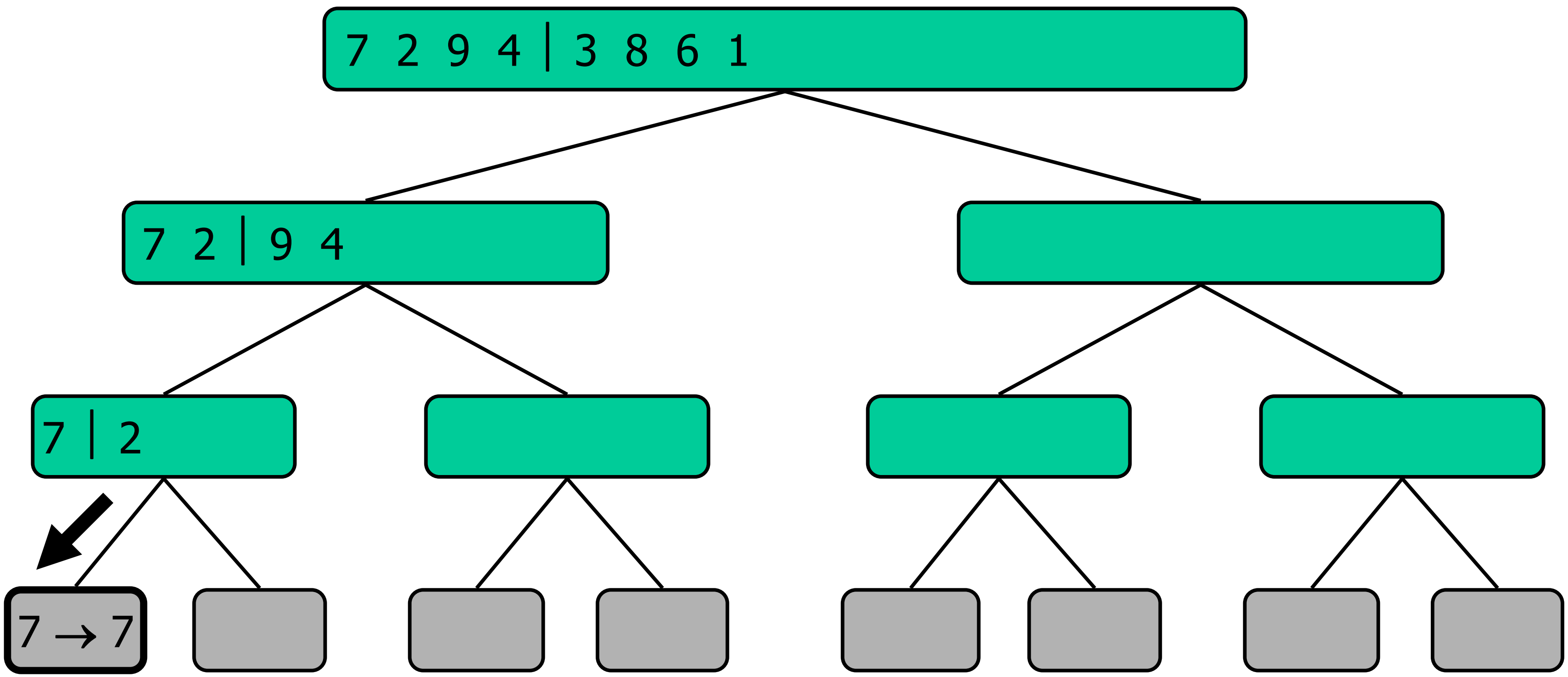
Execution Example (cont.)

- Recursive call, partition



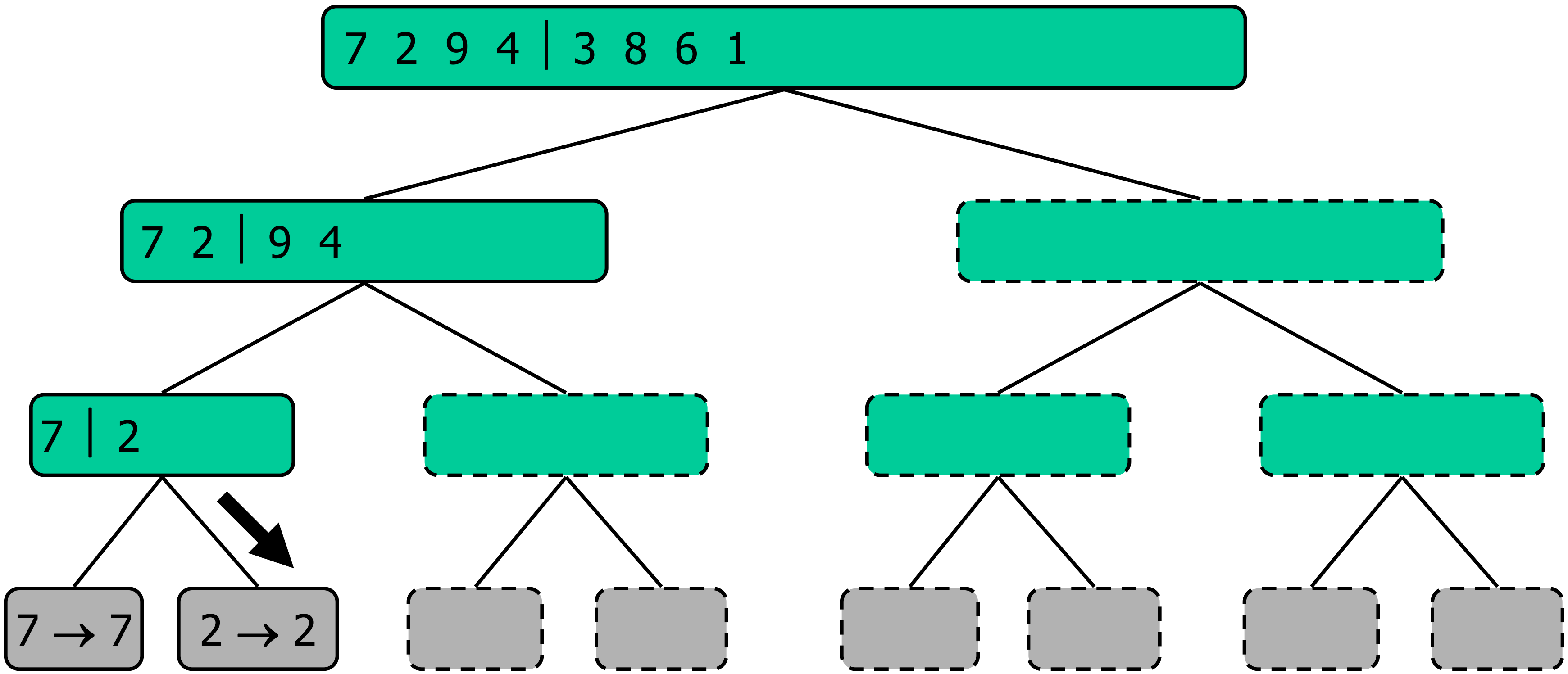
Execution Example (cont.)

- Recursive call, base case



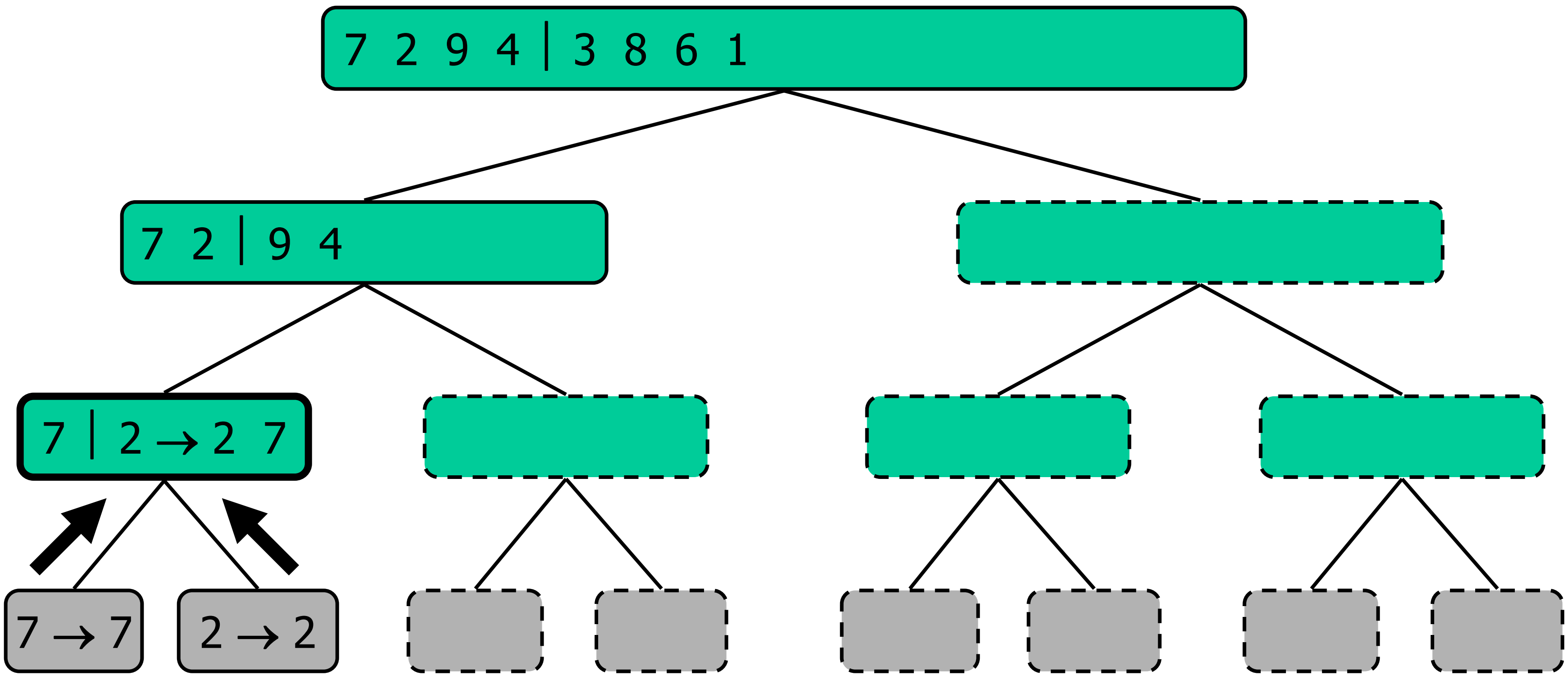
Execution Example (cont.)

- Recursive call, base case



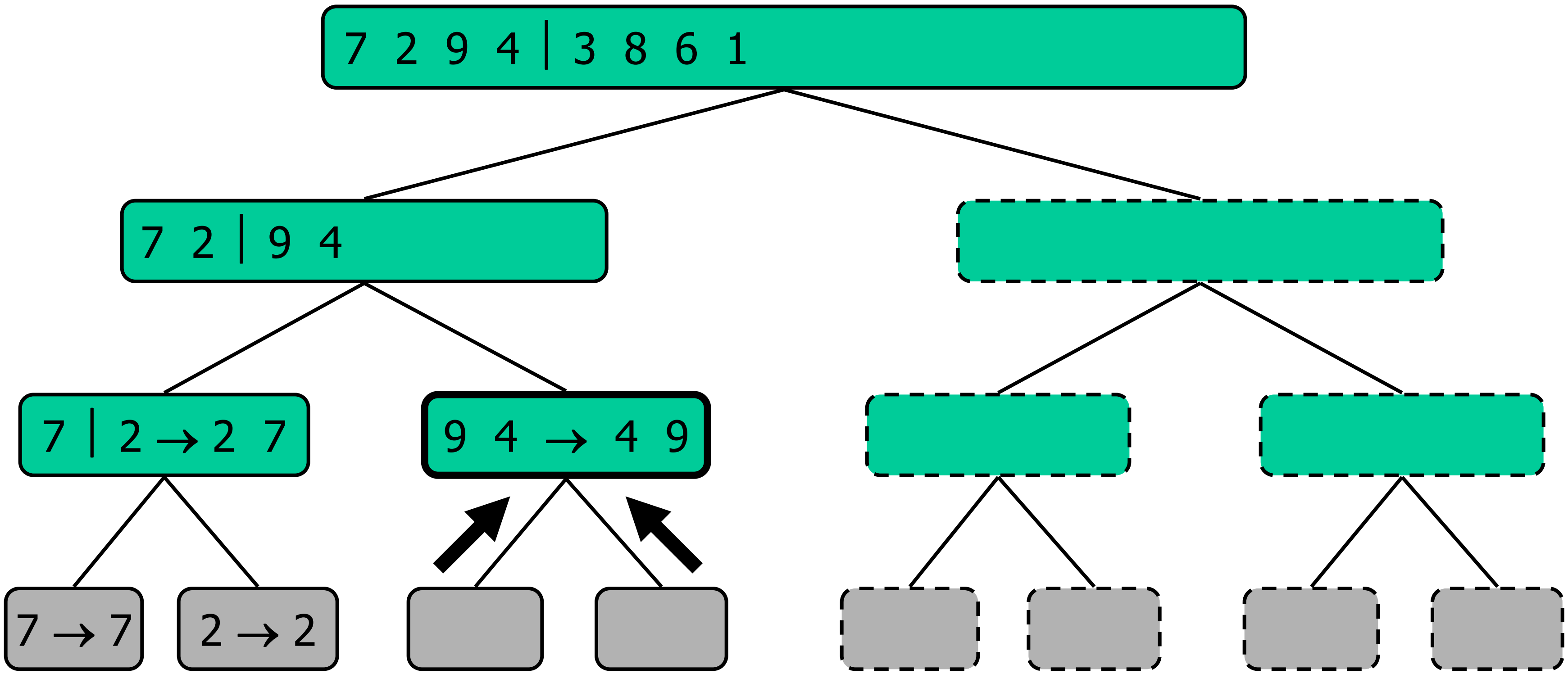
Execution Example (cont.)

- Merge



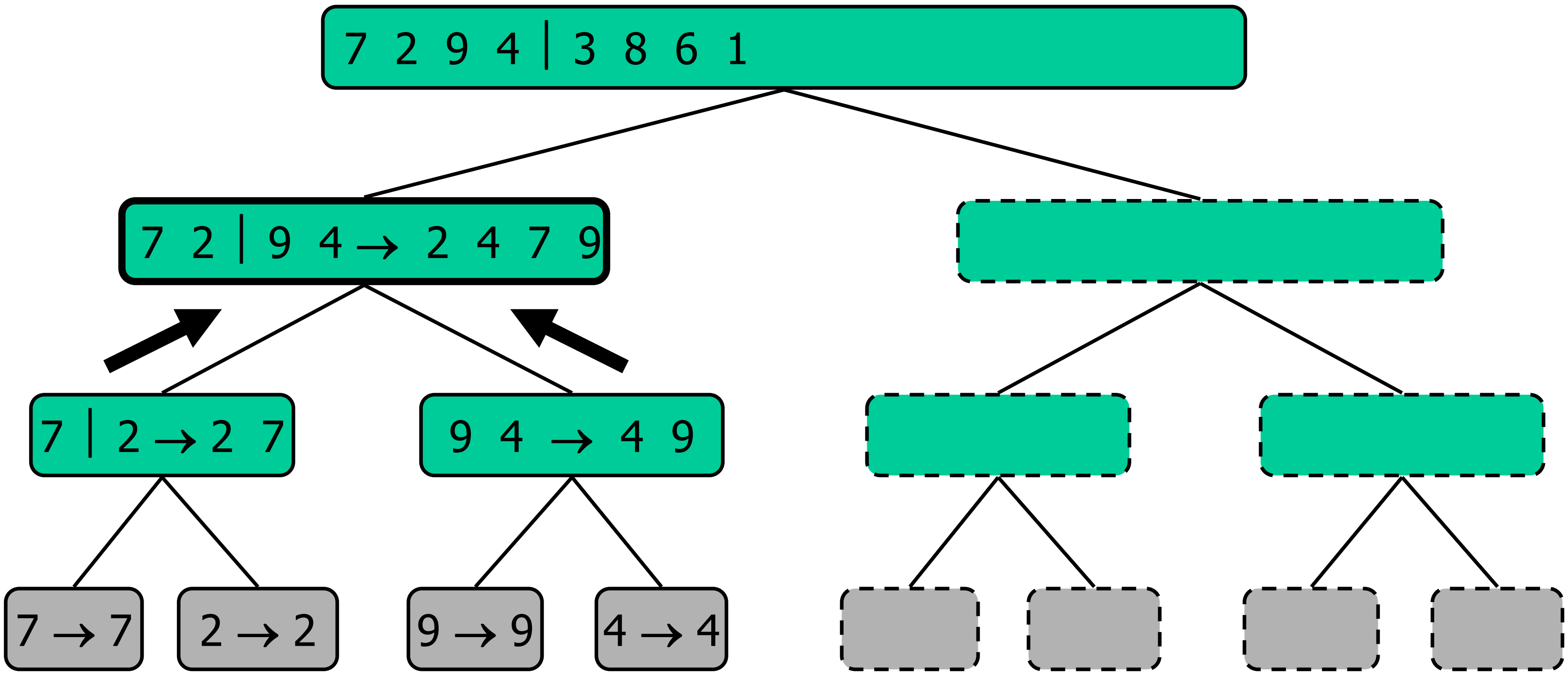
Execution Example (cont.)

- Recursive call, ..., base case, merge



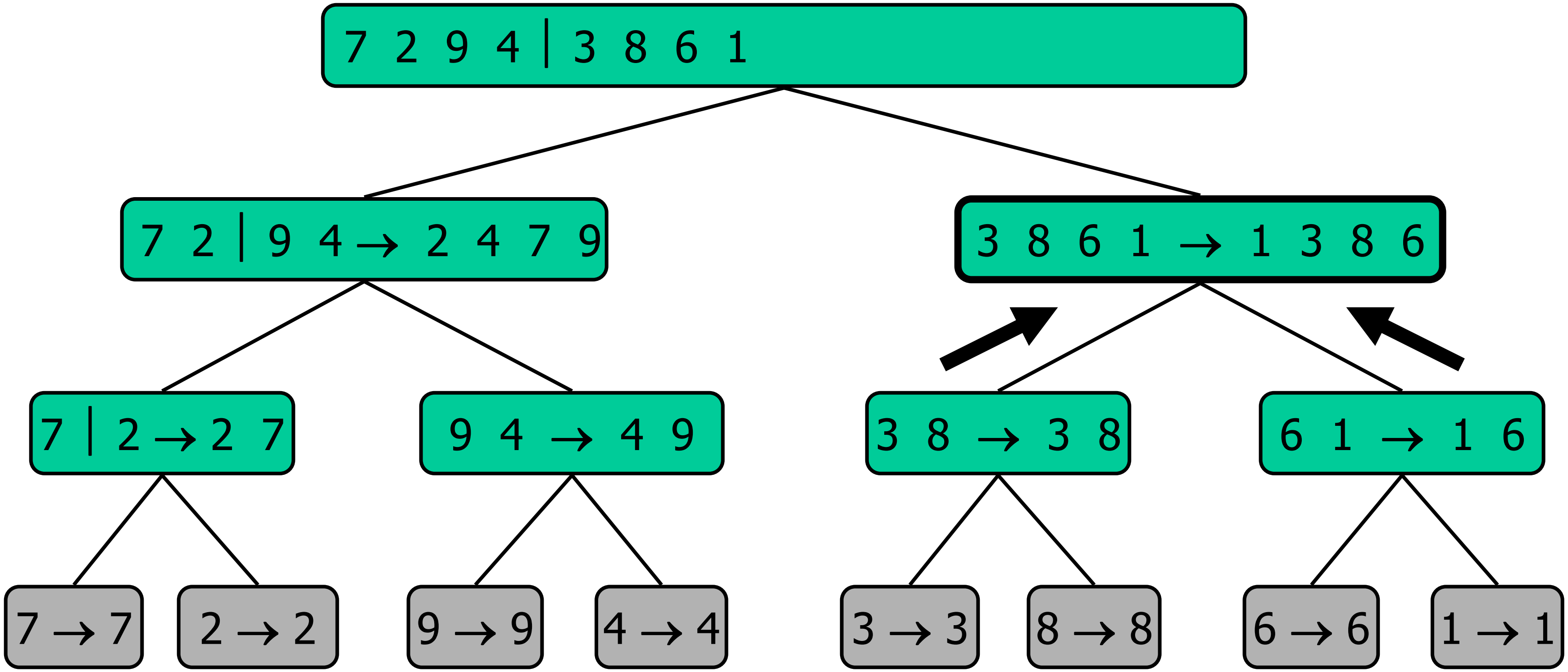
Execution Example (cont.)

- Merge



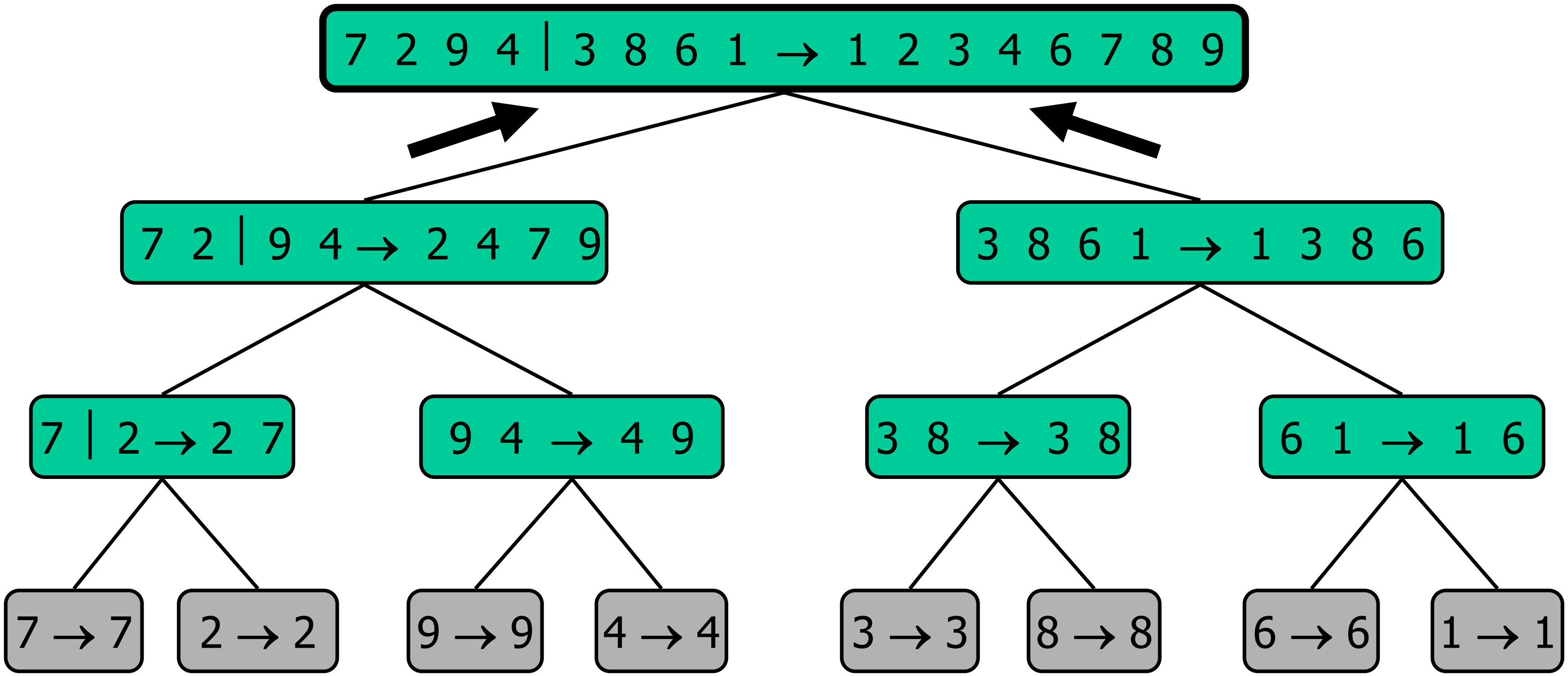
Execution Example (cont.)

- Recursive call, ..., merge, merge



Execution Example (cont.)

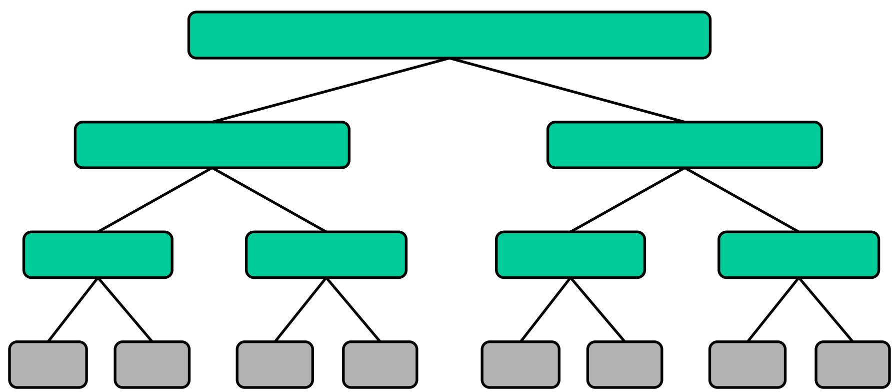
- Merge



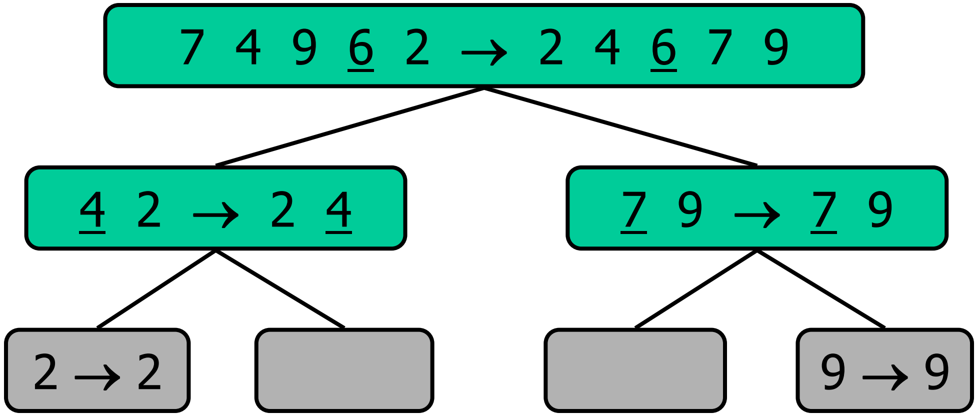
Analysis of Mergesort

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

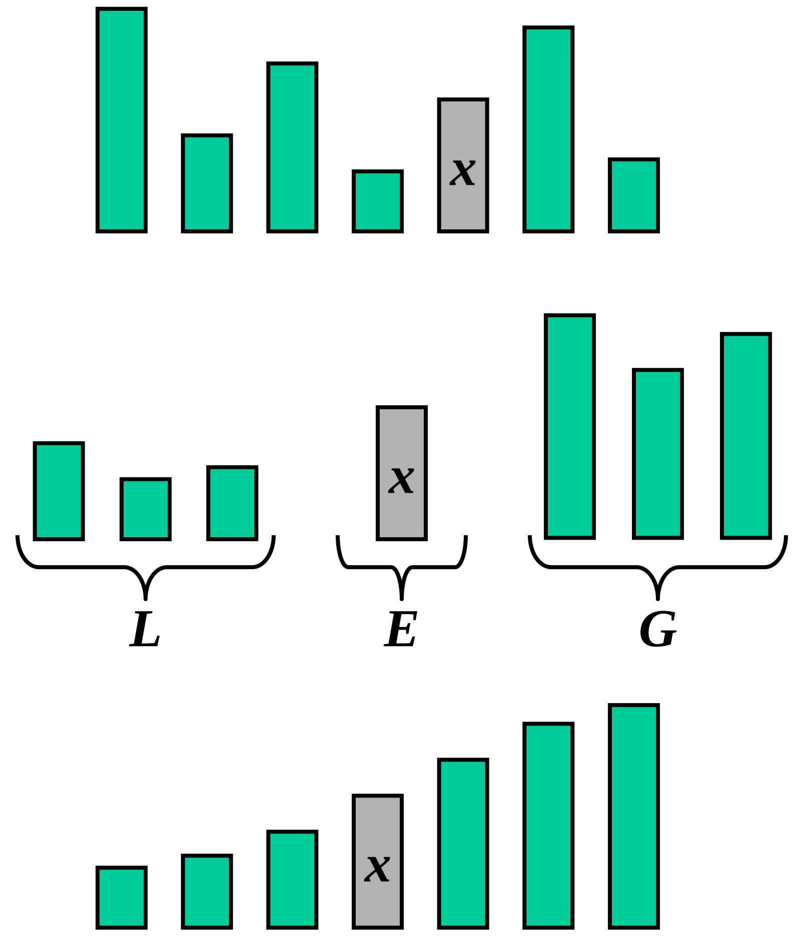


Quicksort



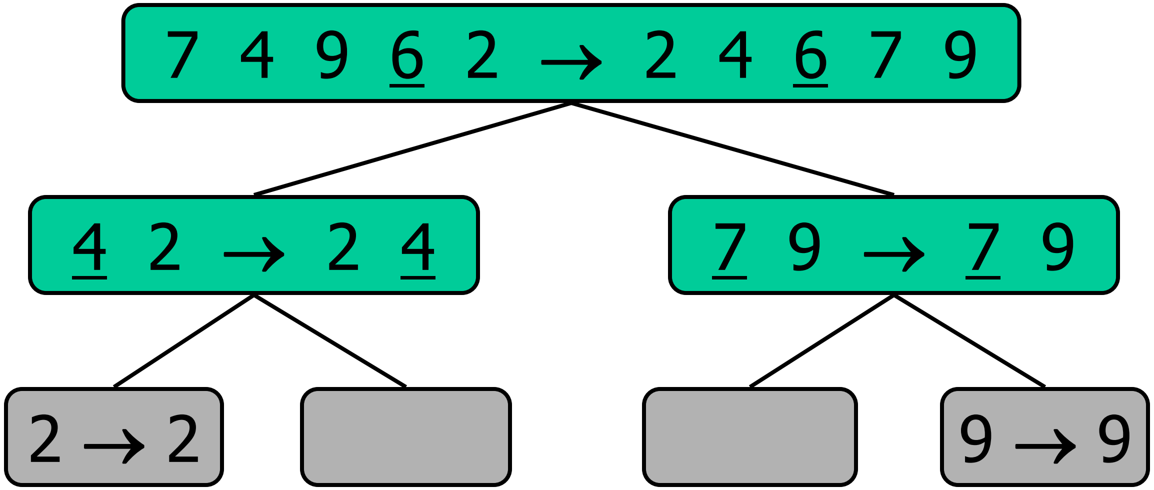
Quicksort

- Divide: pick a random element ***v*** (called pivot) and partition ***S*** into
  - ***L*** elements $\leq$ ***x***
  - ***E*** elements = ***x***
  - ***G*** elements $\geq$ ***x***
- Recur:
  sort ***L*** and ***G***
- Conquer:
  join ***L***, ***E*** and ***G***



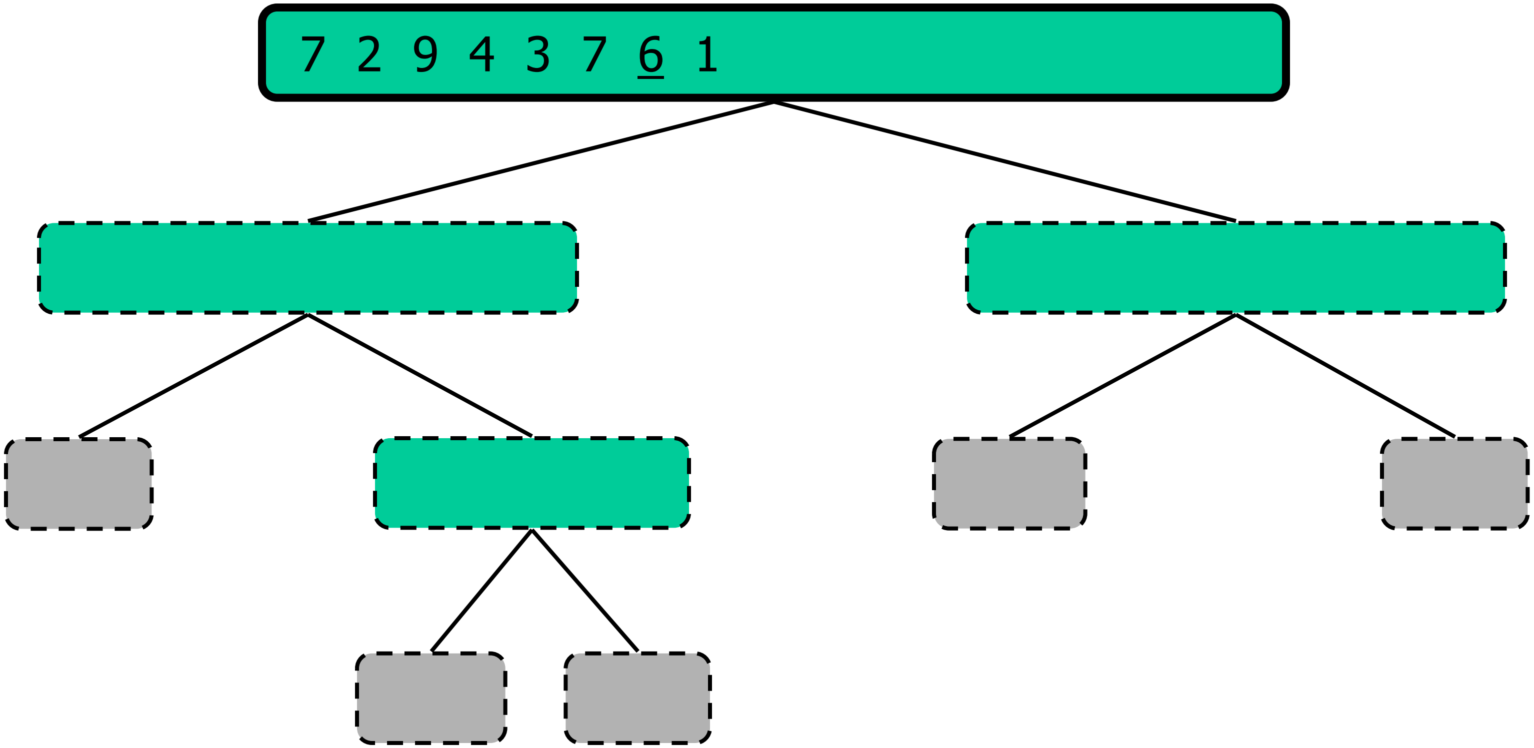
Quicksort ExecutionTree

- Each node represents a recursive call of quicksort 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



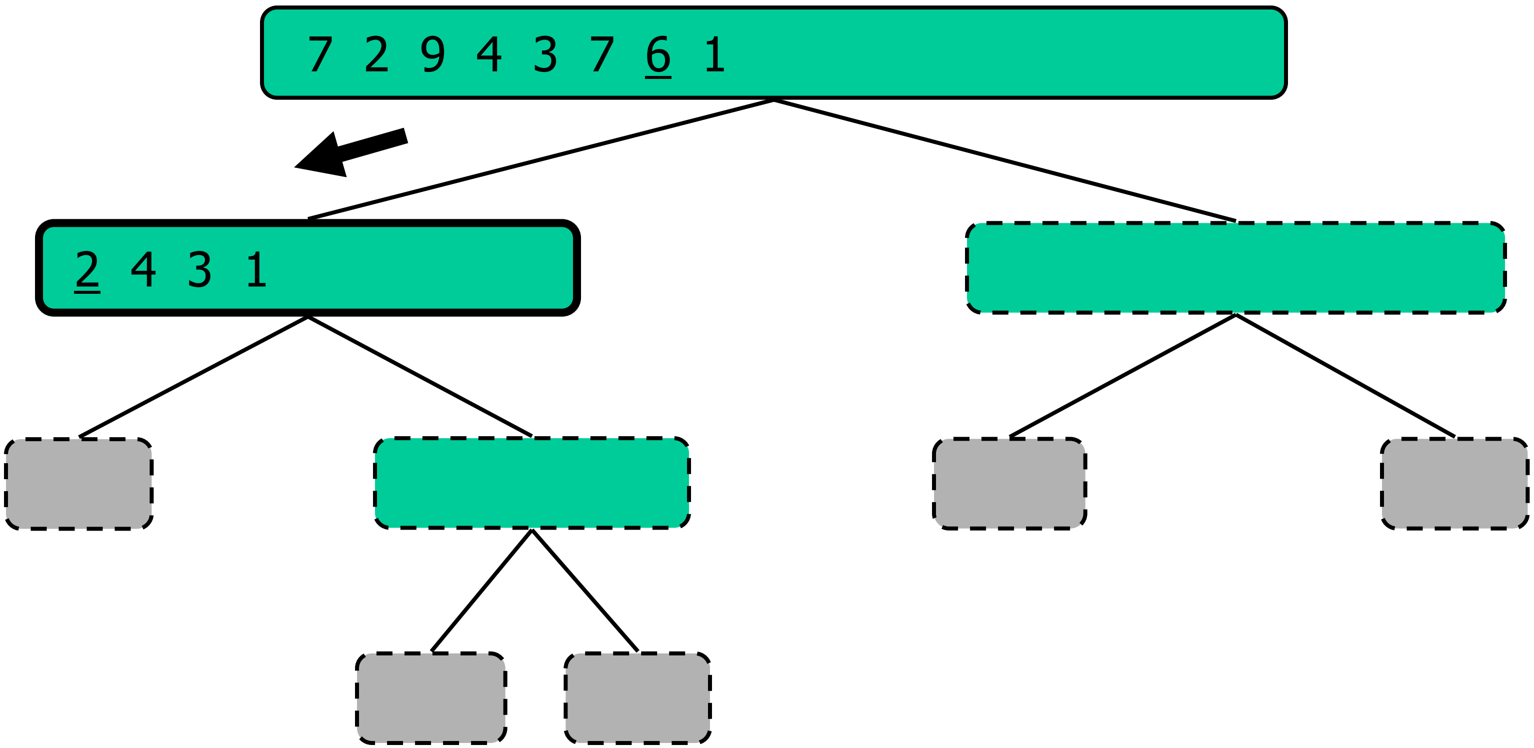
Quicksort: Execution Example

- Pivot selection



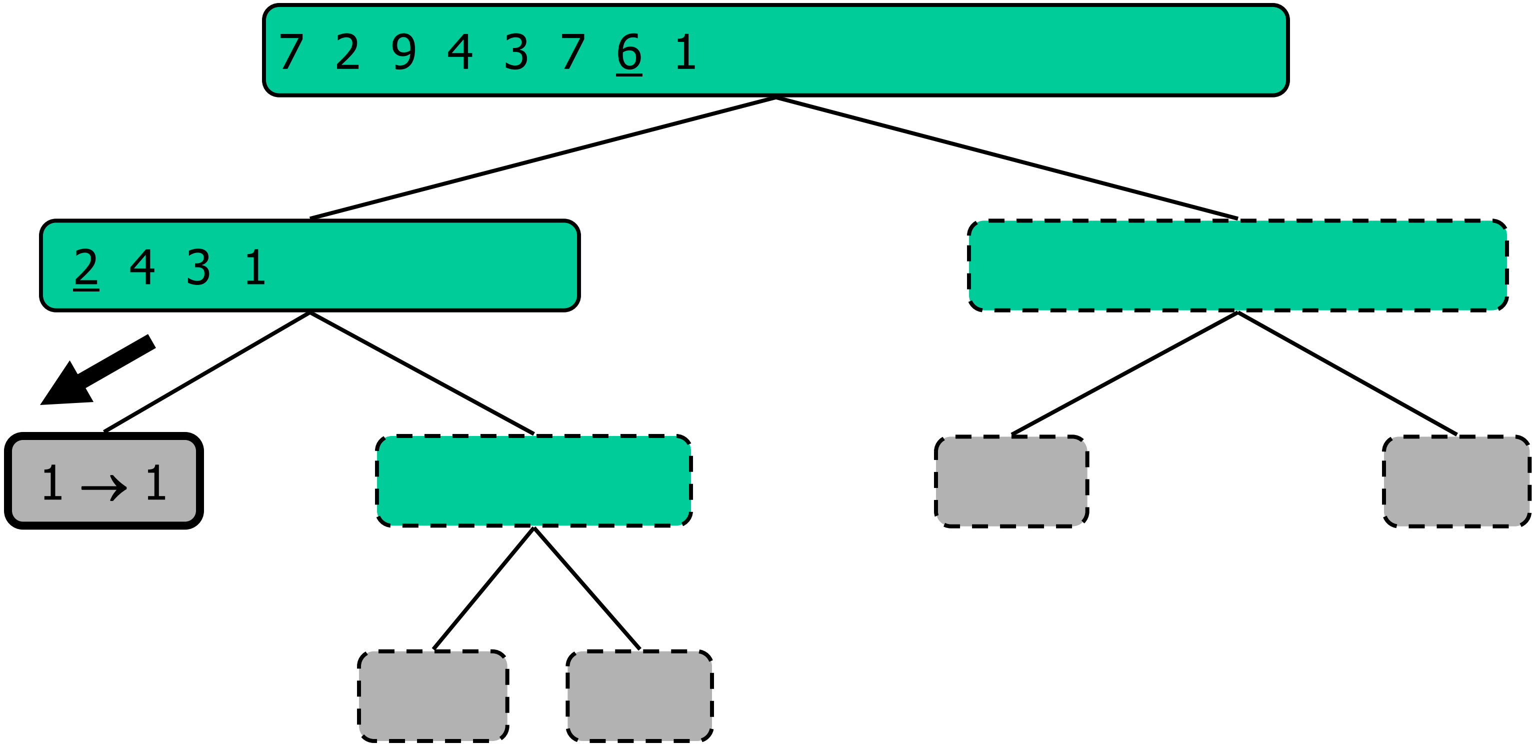
Execution Example (cont.)

- Partition, recursive call, pivot selection



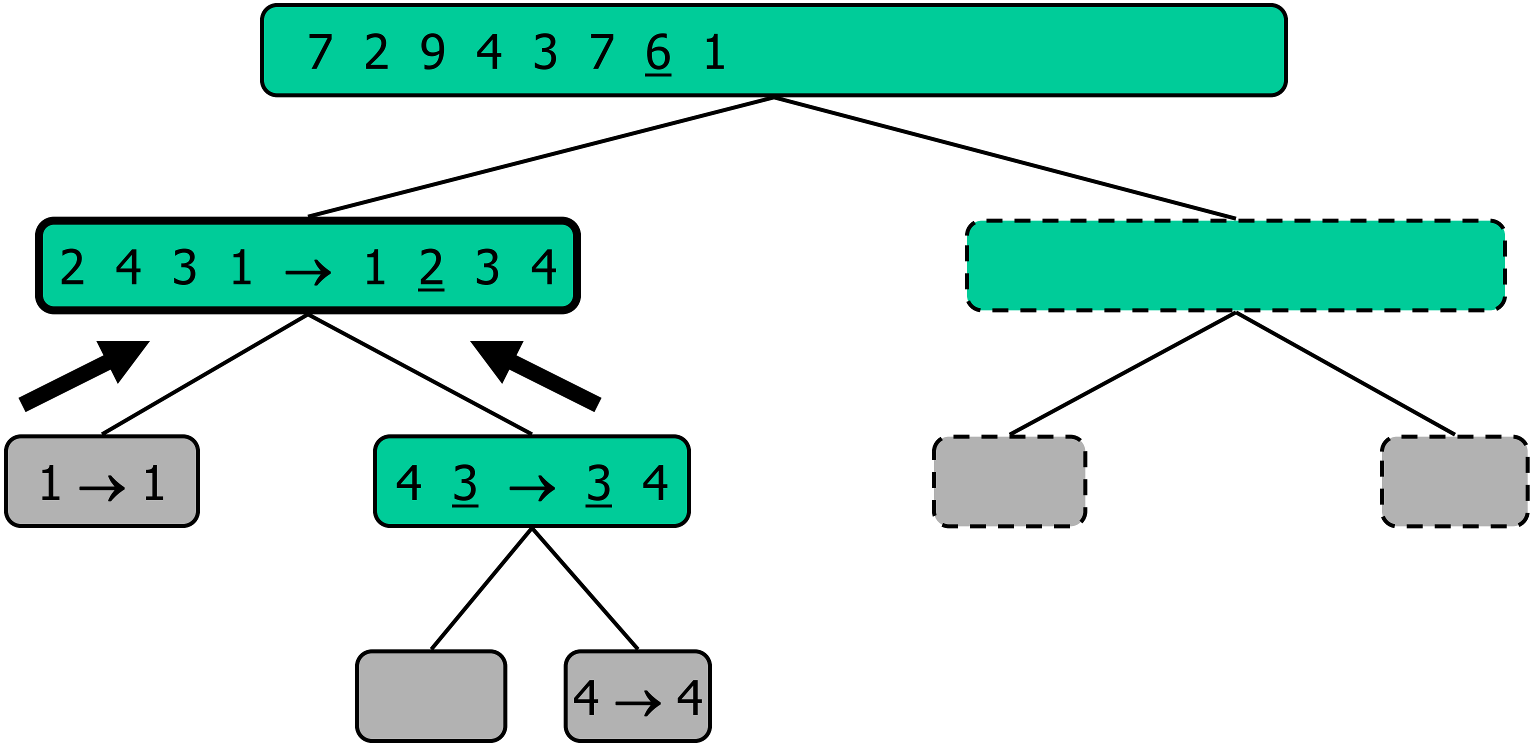
Execution Example (cont.)

- Partition, recursive call, base case



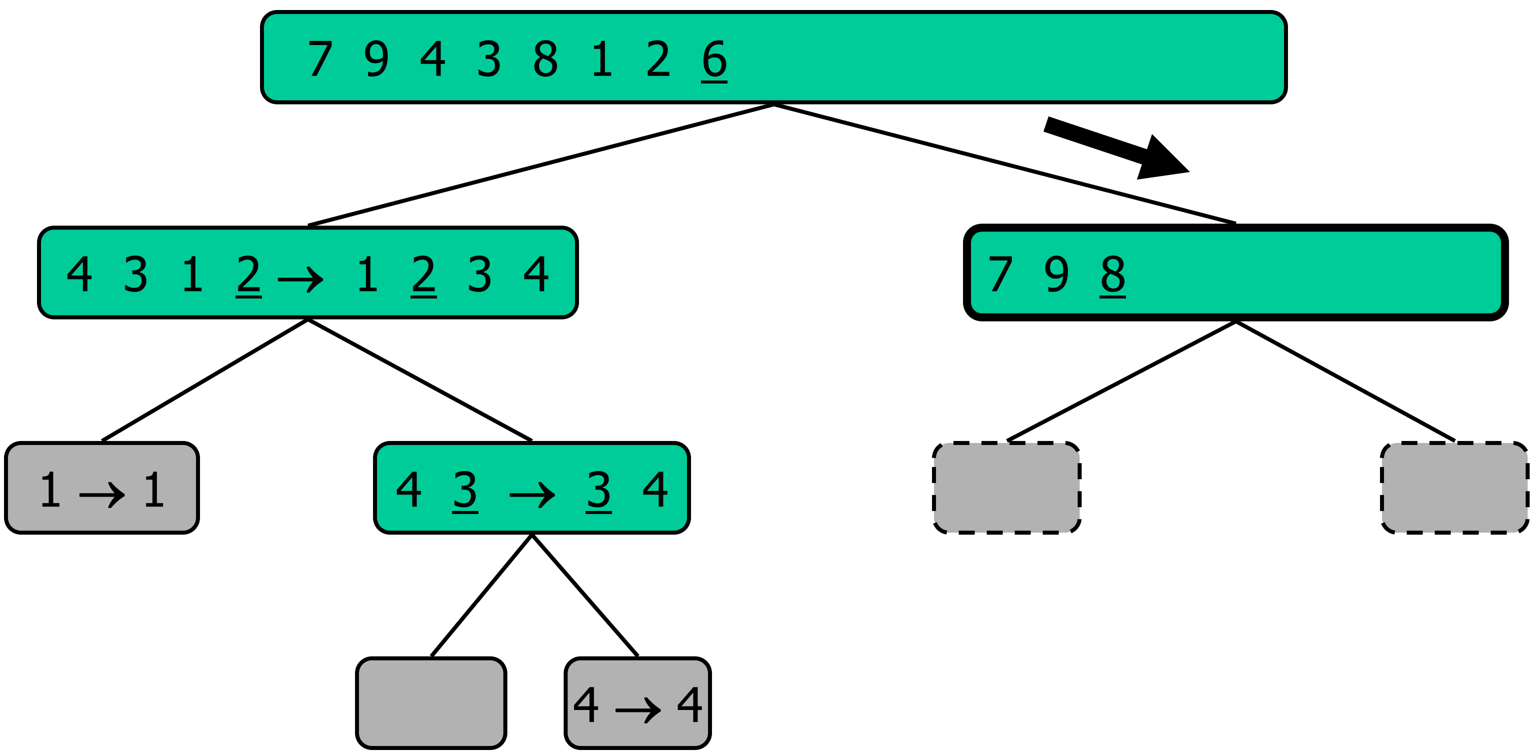
Execution Example (cont.)

- Recursive call, **…**, base case, join



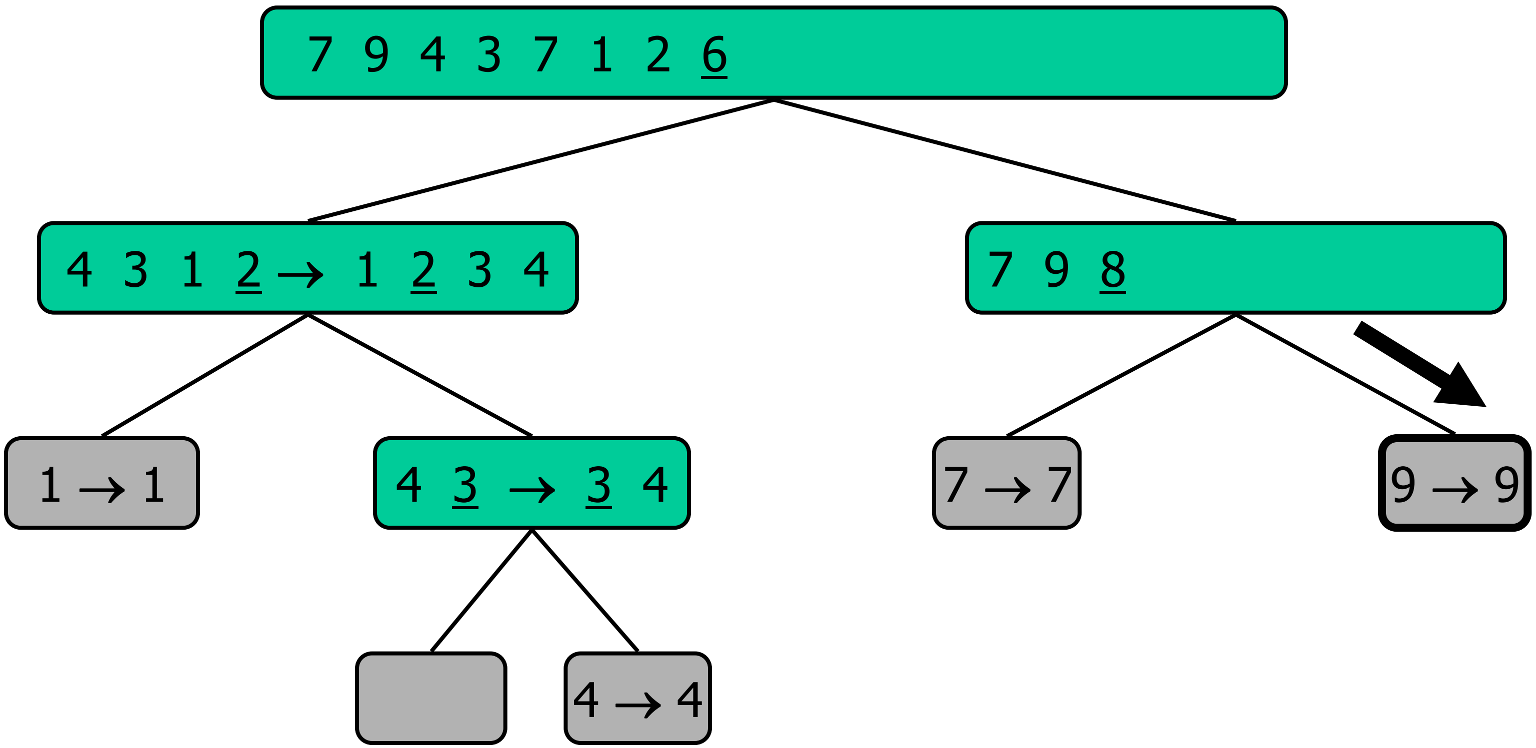
Execution Example (cont.)

- Recursive call, pivot selection



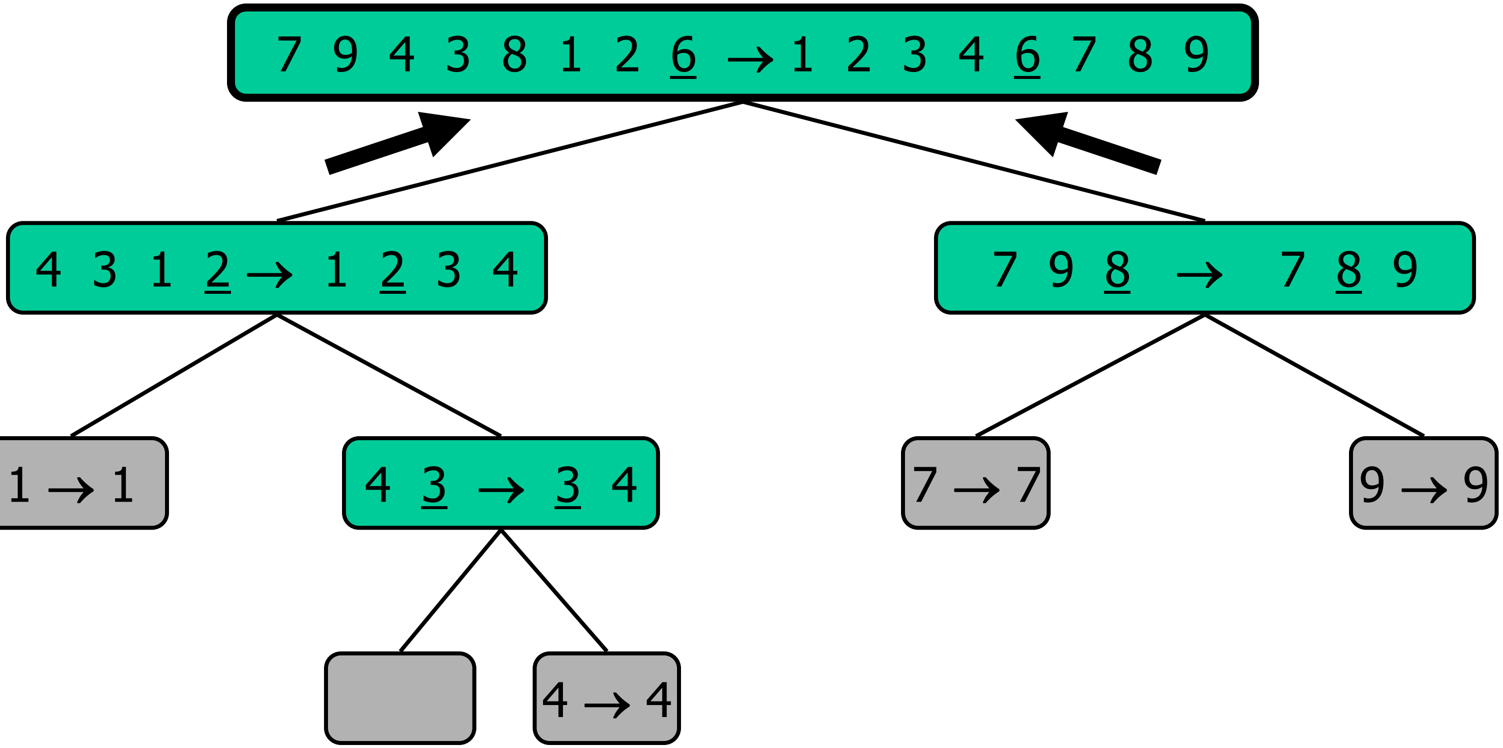
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

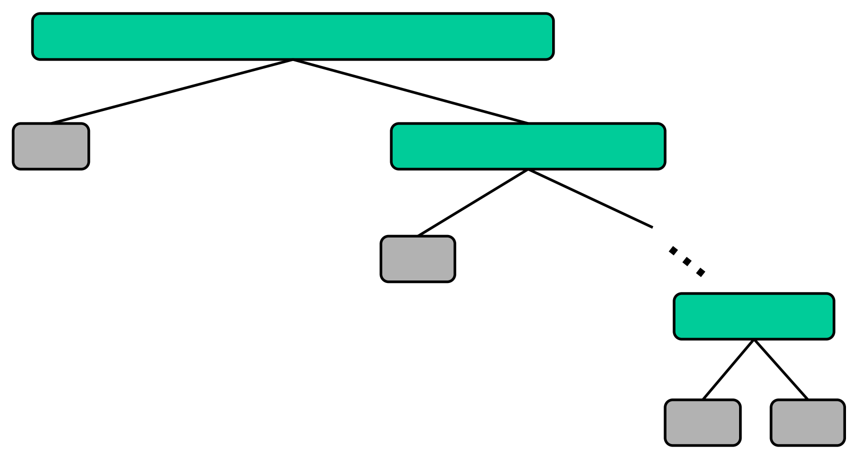
- Join, join



Worst-case Running Time

- The worst case is when the pivot is the unique min or max: $\boldsymbol{L}$ and $\boldsymbol{G}$ are size of $\boldsymbol{n} - \mathbf{1}$ and $\mathbf{0}$
- The running time is proportional to $\boldsymbol{n} + (\boldsymbol{n} - \mathbf{1}) + \ldots + \mathbf{1}$
- Thus, the worst-case running time is $\boldsymbol{O}(\boldsymbol{n}^2)$

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



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)
Quicksort	$O(n \log n)$ expected	• in-place, randomized • fastest (good for large inputs)
Heapsort	$O(n \log n)$	• in-place • fast (good for large inputs)
Mergesort	$O(n \log n)$	• sequential data access • fast (good for huge inputs)