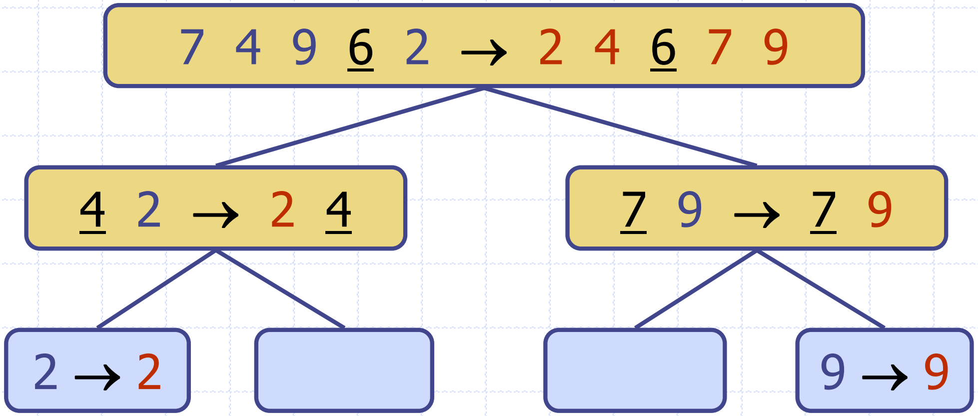
Sorting



Selection Sort

- Given is a list L of n value {L[0], … , L[n-1]}
- Divide list into unsorted (left) and sorted part (right – initially empty):

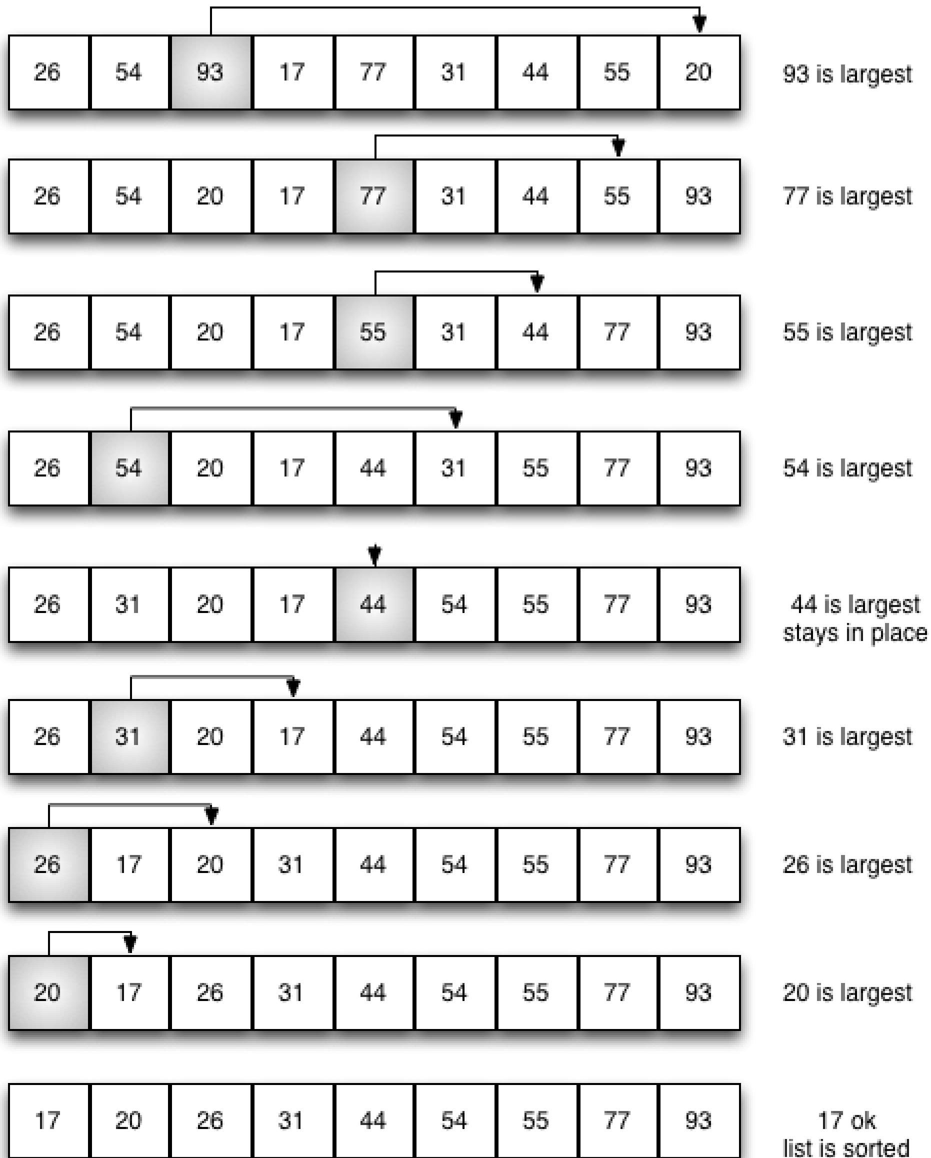
  Unsorted: {L[0], … , L[n-1]}   Sorted: {}

- In each pass find largest value and place it to the right of the unsorted part using a single swap (only one exchange for every pass through the list)
- Reduce size of unsorted part by one and increase size of sorted part by one. After i^{th} pass:

  Unsorted: {L[0], … , L[n-1-i]} Sorted: {L[n-i],…,L[n-1]}

- Repeat until unsorted part has a size of 1 – then all elements are sorted

Selection Sort



Insertion Sort

- Given is a list L of n value {L[0], … , L[n-1]}
- Divide list into sorted (left – initially only one element) and sorted part (right):

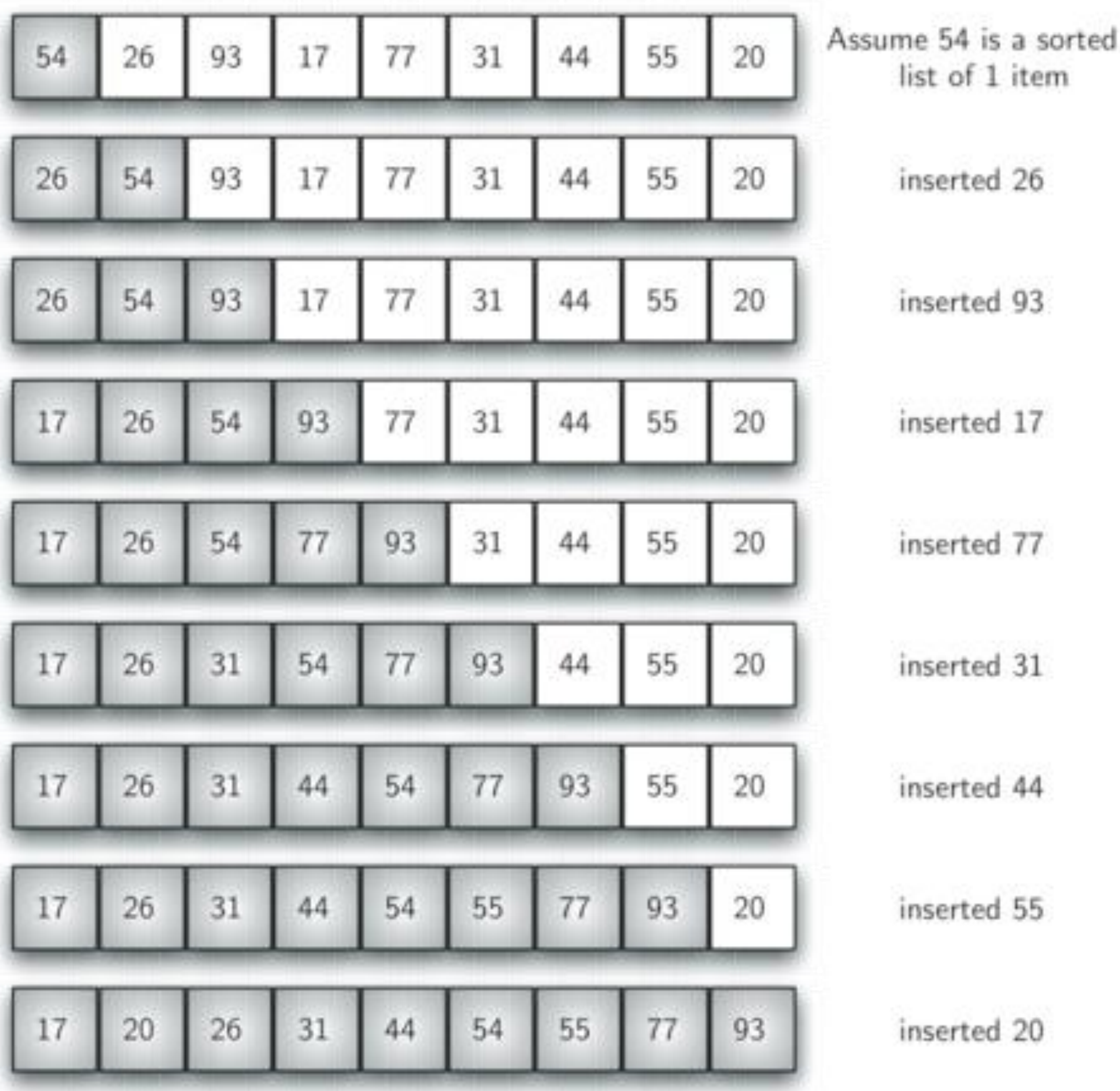
  Sorted: {L[0]} Unsorted: {L[1], … , L[n-1]}

- In each pass take left most element from unsorted part and place it into correct position of sorted part
- Reduce size of unsorted part by one and increase size of sorted part by one. After i^{th} pass::

  Sorted: {L[0],…,L[i]} Unsorted: {L[i+1], … , L[n-1-i]}

- Repeat until unsorted part is an empty list – then all elements are sorted

Insertion Sort



Shell Sort

Remember:

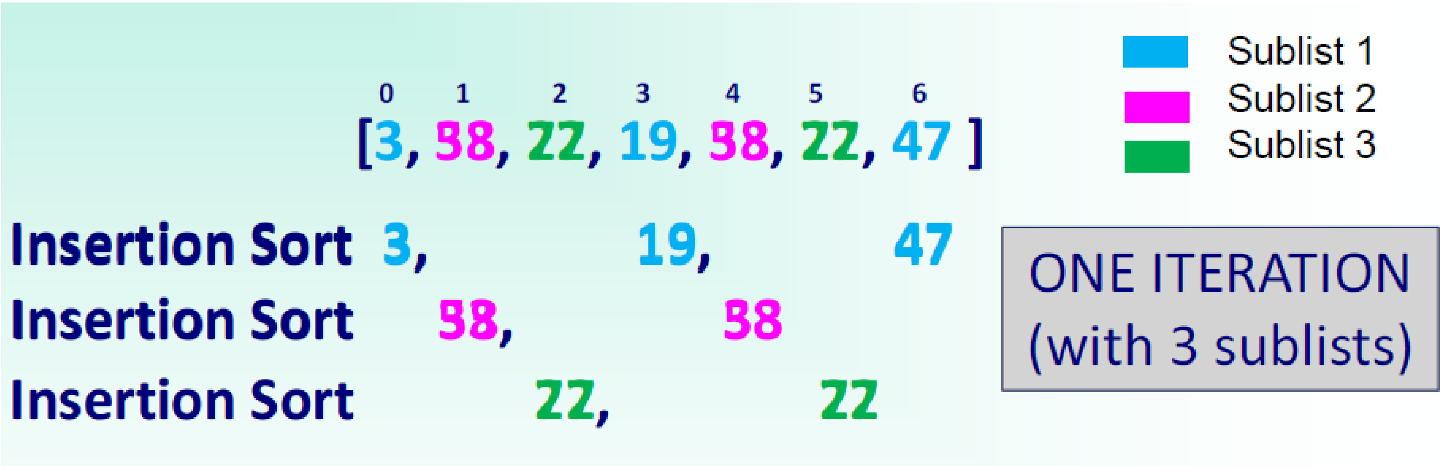
- Insertion sort has fewer comparisons than Selection sort
- Selection sort has fewer moves-swaps than Insertion sort
- => IDEA: compare/shift non-neighbouring elements

Shell Sort (diminishing increment sort)

- On average shell sort has fewer comparisons than Selection sort and Bubble sort, and fewer moves than Insertion sort
- Shell sort is based on the Insertion sort algorithm,
- BUT: instead of shifting elements many times by one step, it makes larger moves

Shell Sort

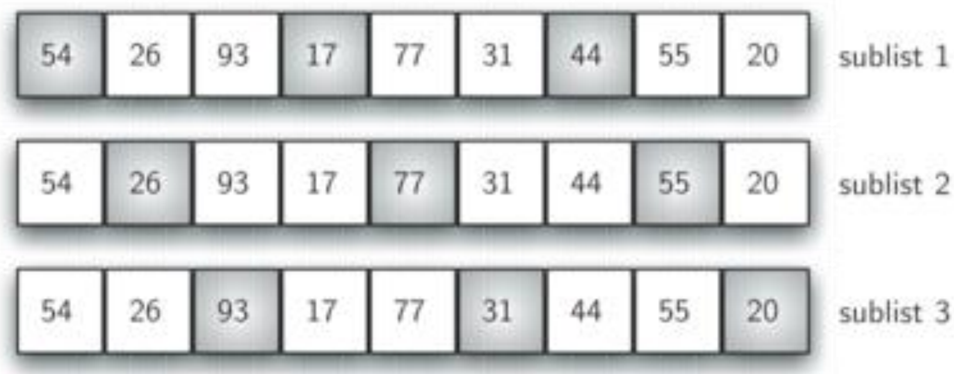
- Divide the list into lots of smaller sublists, e.g., gap (increment) used below is 3
- Instead of breaking the list into sublists of contiguous items, the Shell sort uses an increment i (gap) to create a sublist by choosing all items that are i items apart



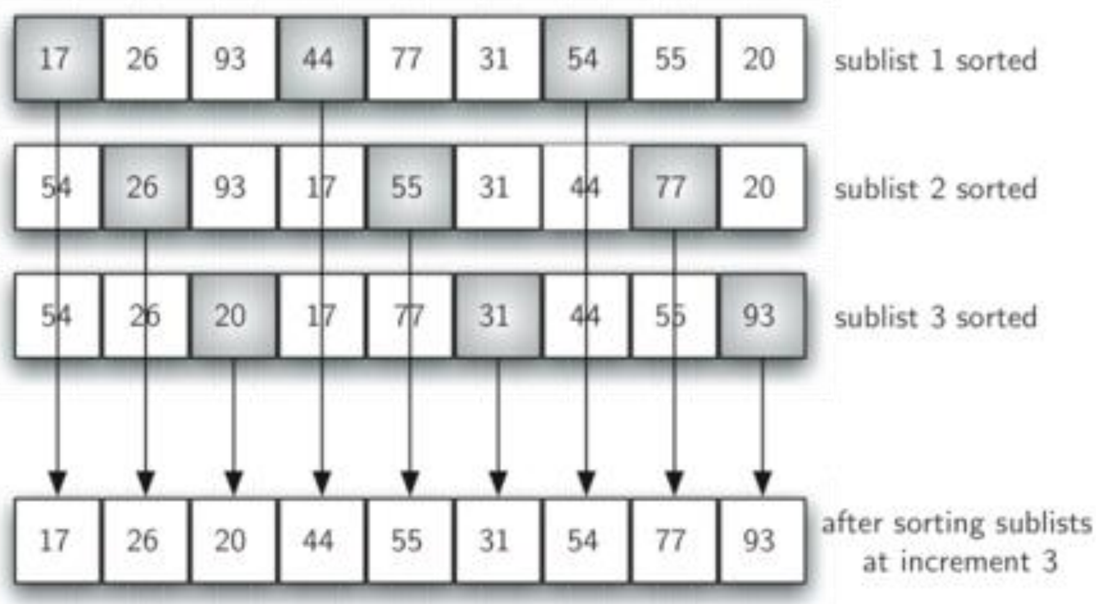
- Each of which is sorted using an insertion sort
- Then repeat sorting with reduced gap (=> fewer, but larger sublists) until gap is 1.

Shell Sort

- list of 9 items
- increment of 3
- 3 sublists
- Sort each by an Insertion sort

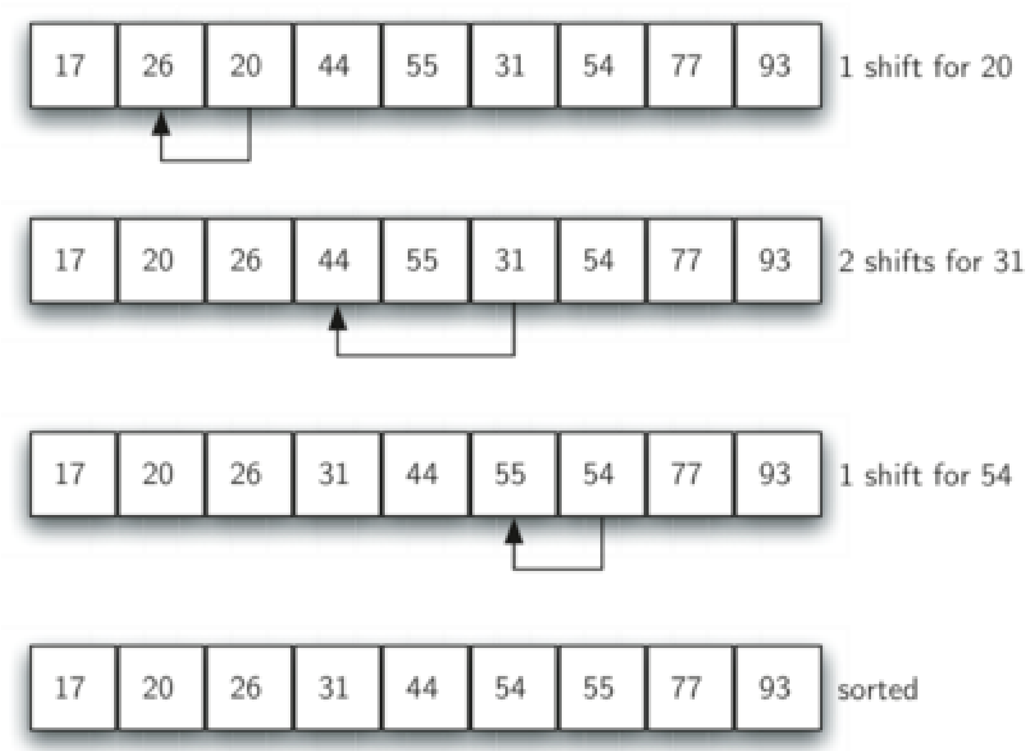


- list not completely sorted
- But by sorting sublists, we have moved items closer to where they actually belong



Shell Sort

- final insertion sort using an increment of one (standard Insertion sort)
- sorting with gap 1 very efficient because list almost sorted due to previous steps
- reduced number of shifting operations necessary to put list in its final order



Shell Sort

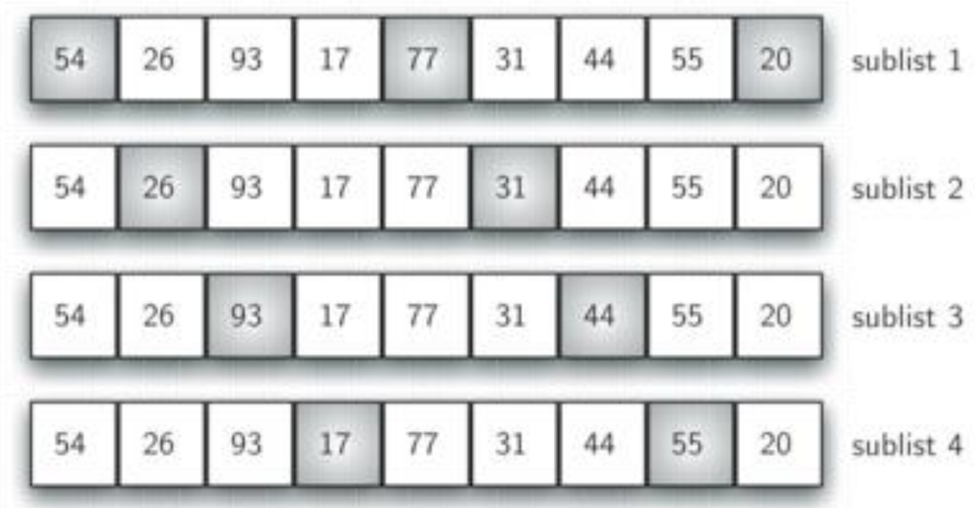
- Choose a gap size, do an Insertion sort on all sublists using this chosen gap size (this is a total of one pass of the collection), repeat using smaller gap sizes until finally gap size is one

- In practice, the final insertion sort needs to move few elements

- A default option for gap sizes is 2^k-1, i.e. [..., 31, 15, 7, 3,1]
- Research in the optimal gap sequence is ongoing
- A often quoted empirical derived gap sequence is [701, 301, 132, 57, 23, 10, 4, 1]

Shell Sort

- increments are chosen is unique feature of Shell sort
- we begin with n/2 sublists
- on next pass, n/4 sublists are sorted
- eventually, a single list is sorted with the basic Insertion sort



Shell Sort

Eg. 8//2 → 4 i.e gap of 4

```
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:

      for startposition in range(sublistcount):
        gapInsertionSort(alist,startposition,sublistcount)

      print("After increments of size",sublistcount,"The list is",alist)

      sublistcount = sublistcount // 2



def gapInsertionSort(alist,start,gap):
    for i in range(start+gap,len(alist),gap):

        currentvalue = alist[i]
        position = i

        while position>=gap and alist[position-gap]>currentvalue:
            alist[position]=alist[position-gap]
            position = position-gap

        alist[position]=currentvalue
```

Shell Sort

```
alist = [54,26,93,17,77,31,44,55,20]
shellSort(alist)
print(alist)
```

[54, 26, 93, 17, 77, 31, 44, 55, 20]

After increments of size 4 the list is [20, 26, 44, 17, 54, 31, 93, 55, 77]

[20, 26, 44, 17, 54, 31, 93, 55, 77]

After increments of size 2 the list is [20, 17, 44, 26, 54, 31, 77, 55, 93]

After increments of size 1 the list is [17, 20, 26, 31, 44, 54, 55, 77, 93]

Quiz

Suppose you have the following list of numbers to sort

[5, 16, 20, 12, 3, 8, 9, 17, 19, 7]

[5, 16, 20, 12, 3, 8, 9, 17, 19, 7]

Which answer illustrates the contents of the list after all swapping is complete for a gap size of 3?

- A. [5, 3, 8, 7, 16, 19, 9, 17, 20, 12] √
- B. [3, 7, 5, 8, 9, 12, 19, 16, 20, 17]
- C. [3, 5, 7, 8, 9, 12, 16, 17, 19, 20]
- D. [5, 16, 20, 3, 8, 12, 9, 17, 20, 7]

Shell Sort

- this is an improvement on all the previous sorting algorithms
- a general analysis of shell sort is beyond the scope of this module
- Big-O for Shell sort depends on the gap sequence and input values
- we can say that it tends to fall somewhere between $O(n)$ and $O(n^2)$

- Gap sequence n/2, n/4, … , 1 => worst case $O(n^2)$
- Gap sequence 2^k-1 (…, 31, 15, 7, 3, 1) => worst case $O(n^{1.5})$
- Gap sequence …, 109, 41, 19, 5, 1 => worst case $O(n^{1.333})$

Merge Sort

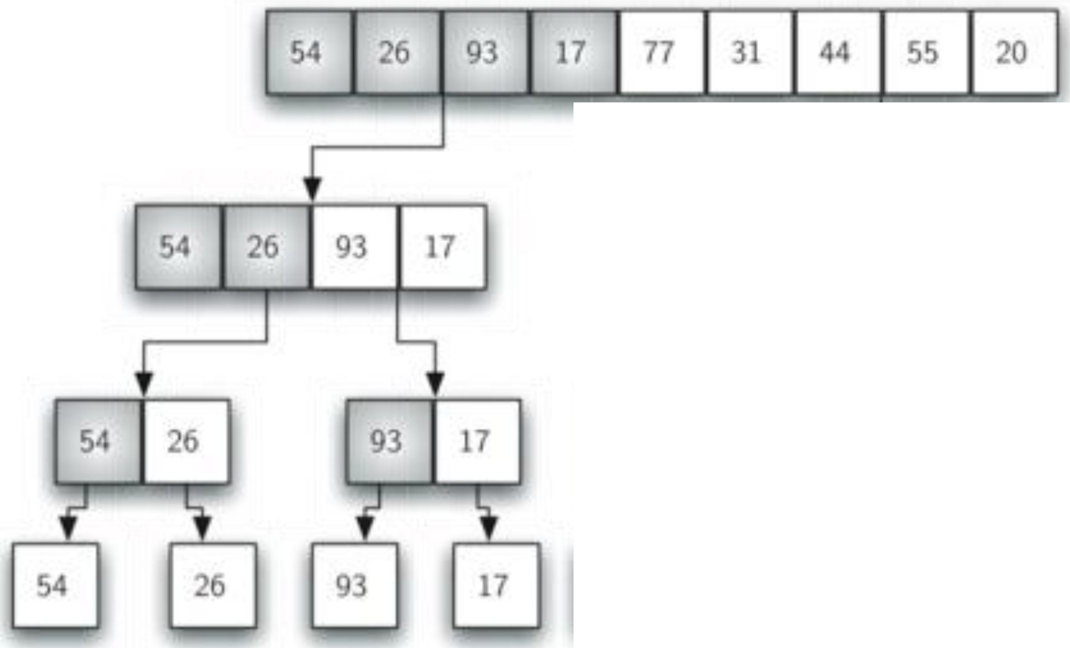
This is a divide and conquer algorithm:

- Cut the list in half
- Sort each half
- Merge the two sorted halves

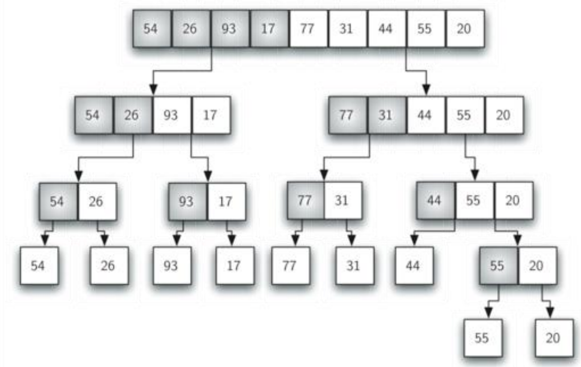
Merge sort is a recursive algorithm

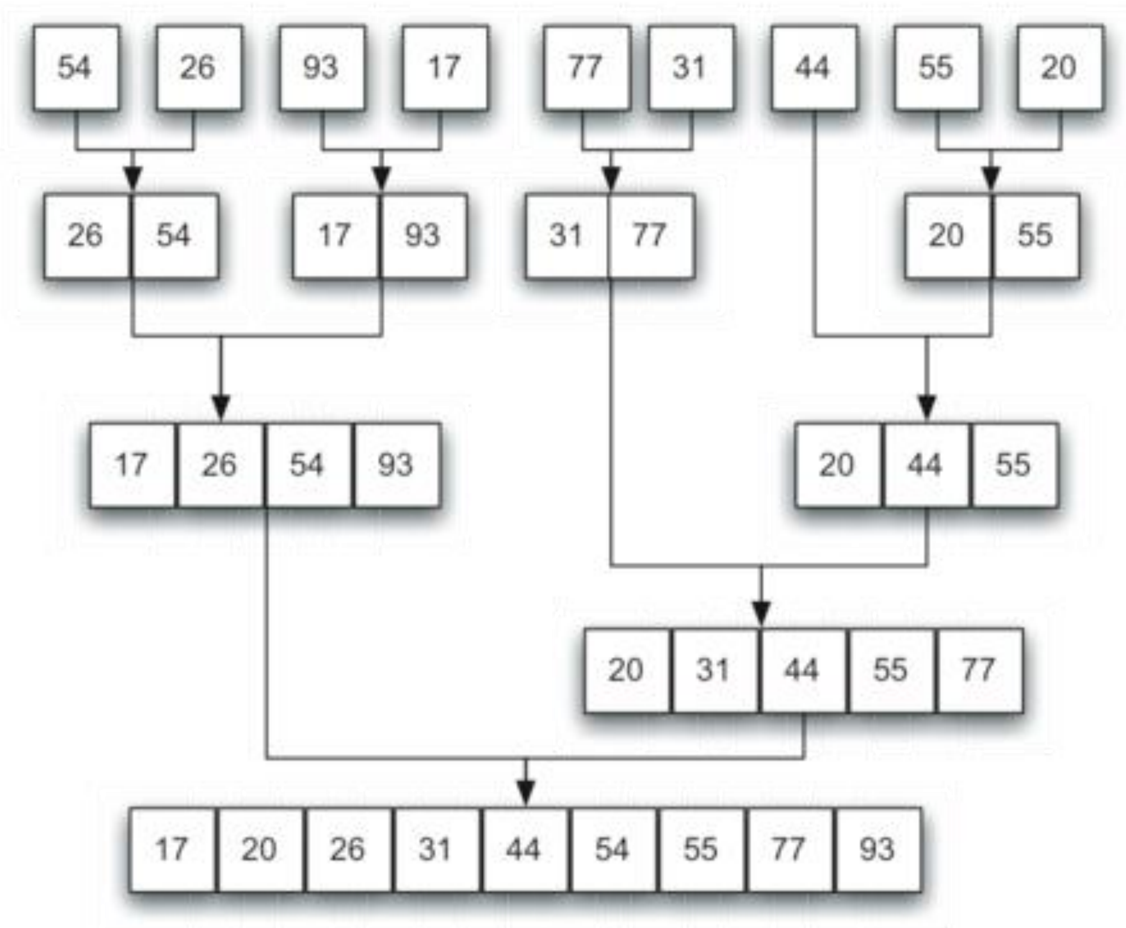
- continually splits a list in half
- If list is empty or has one item, it is sorted (base case)
- If list has more than one item, we split the list and recursively invoke a merge sort on both halves
- Once the two halves are sorted, the fundamental operation, called a merge, is performed
- Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list

Merge Sort



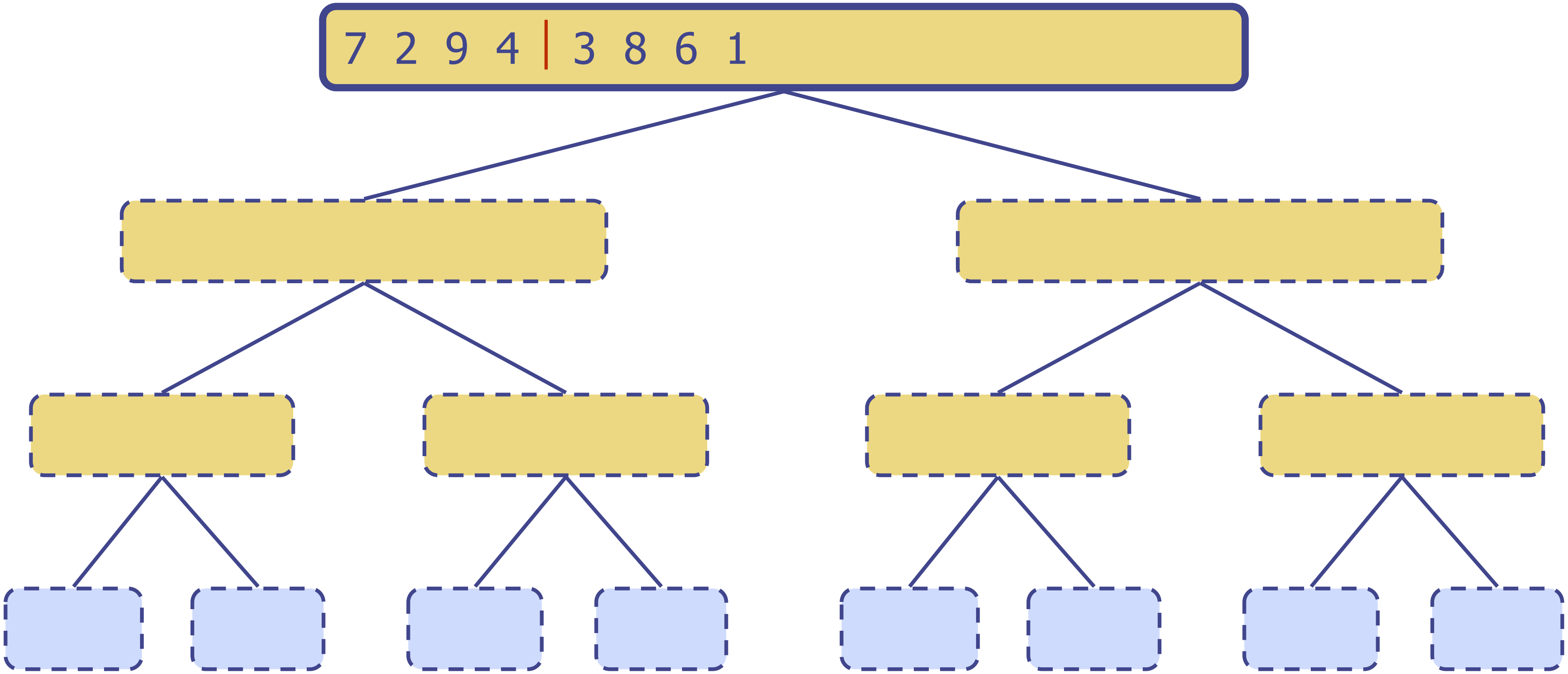
Merge Sort


54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20


54
26
93
17
77
31
44
55
20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

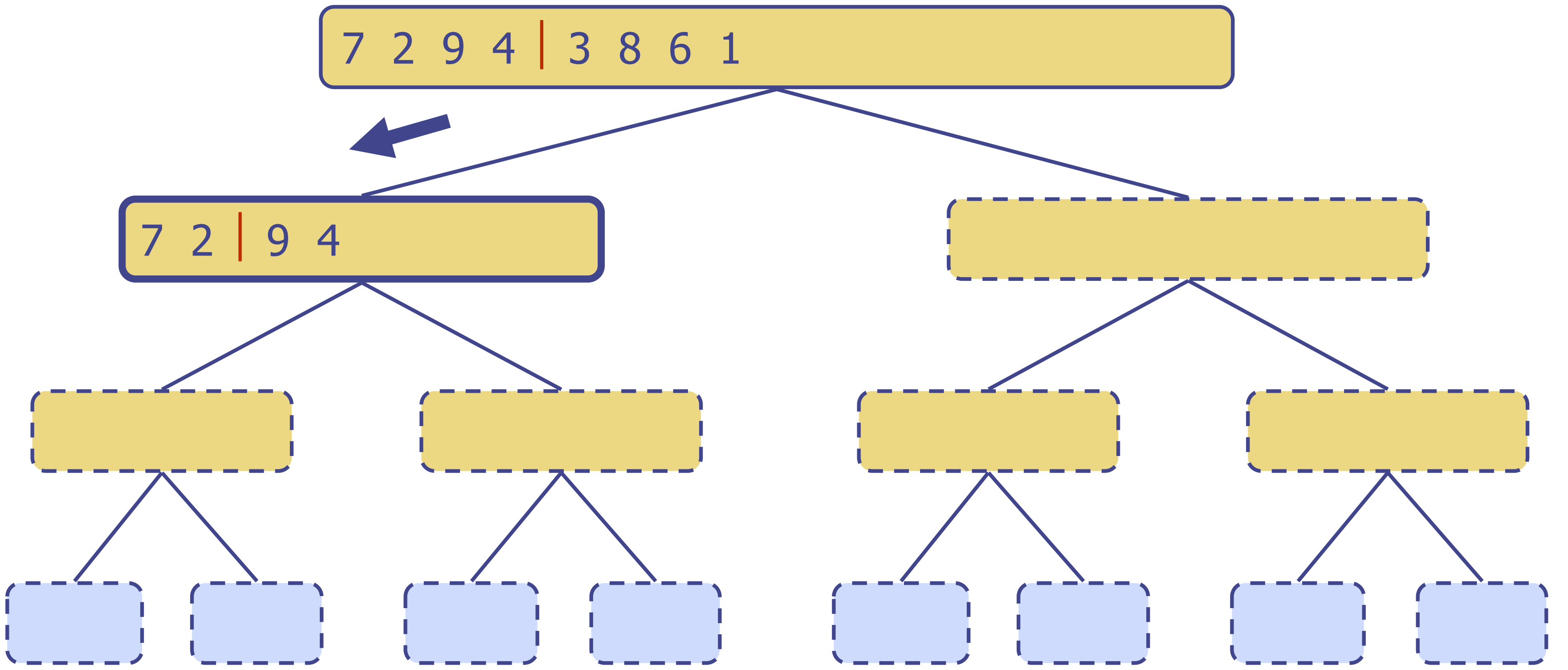
Execution Example

- Partition



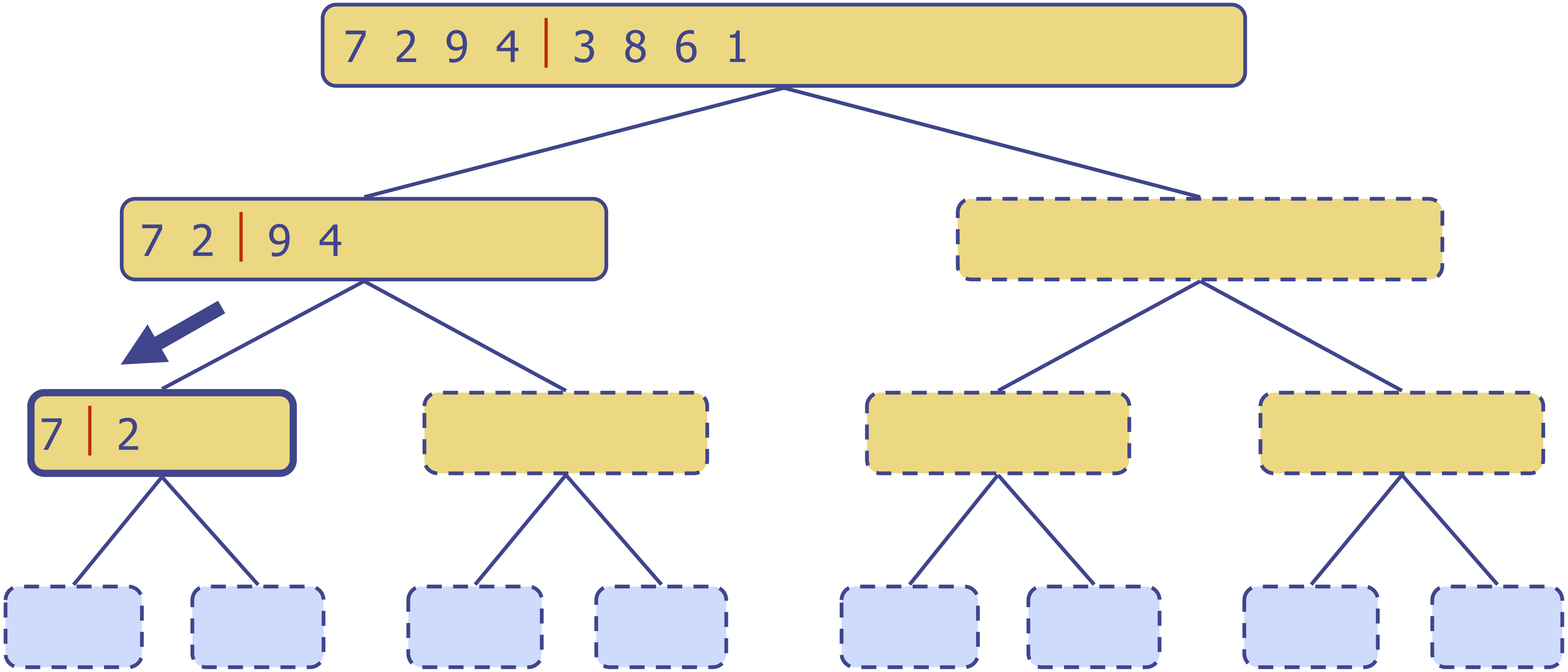
Execution Example (cont.)

- Recursive call, partition



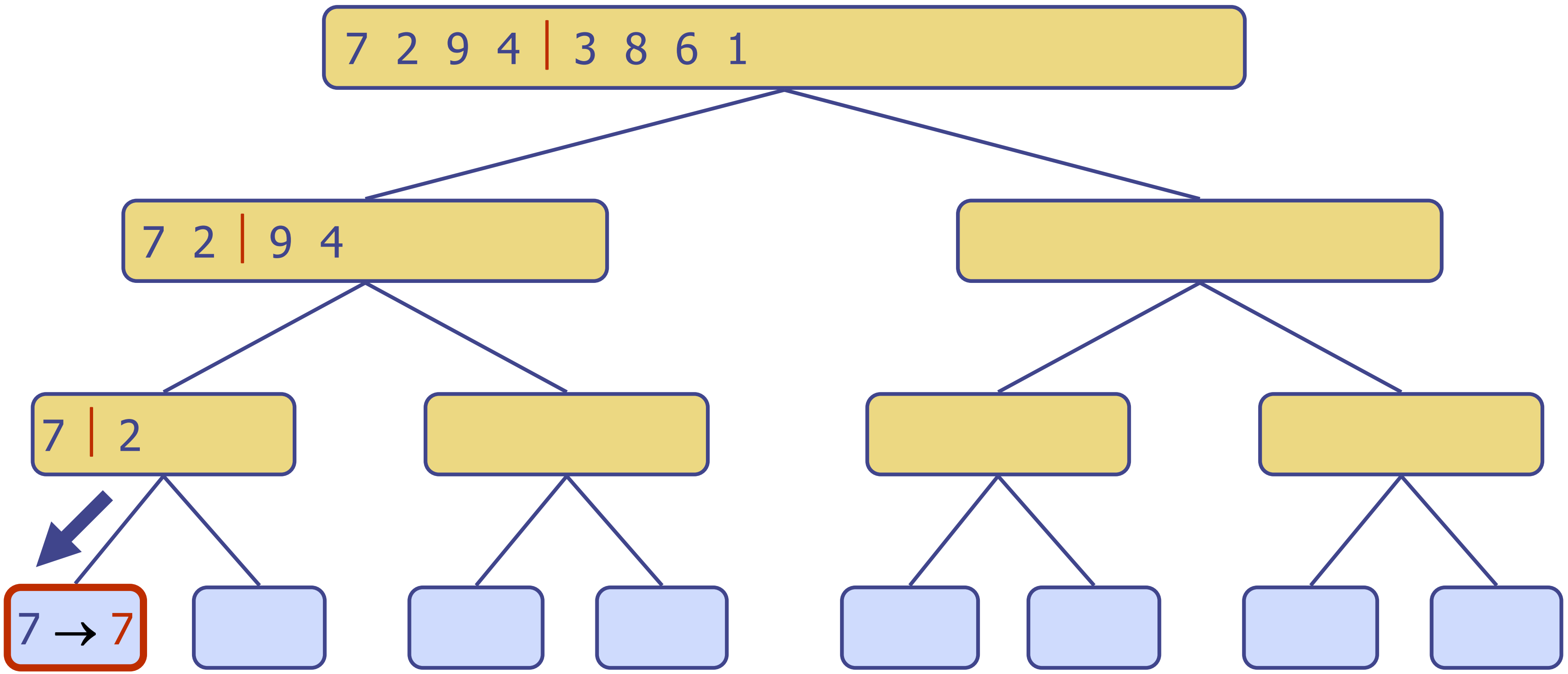
Execution Example (cont.)

- Recursive call, partition



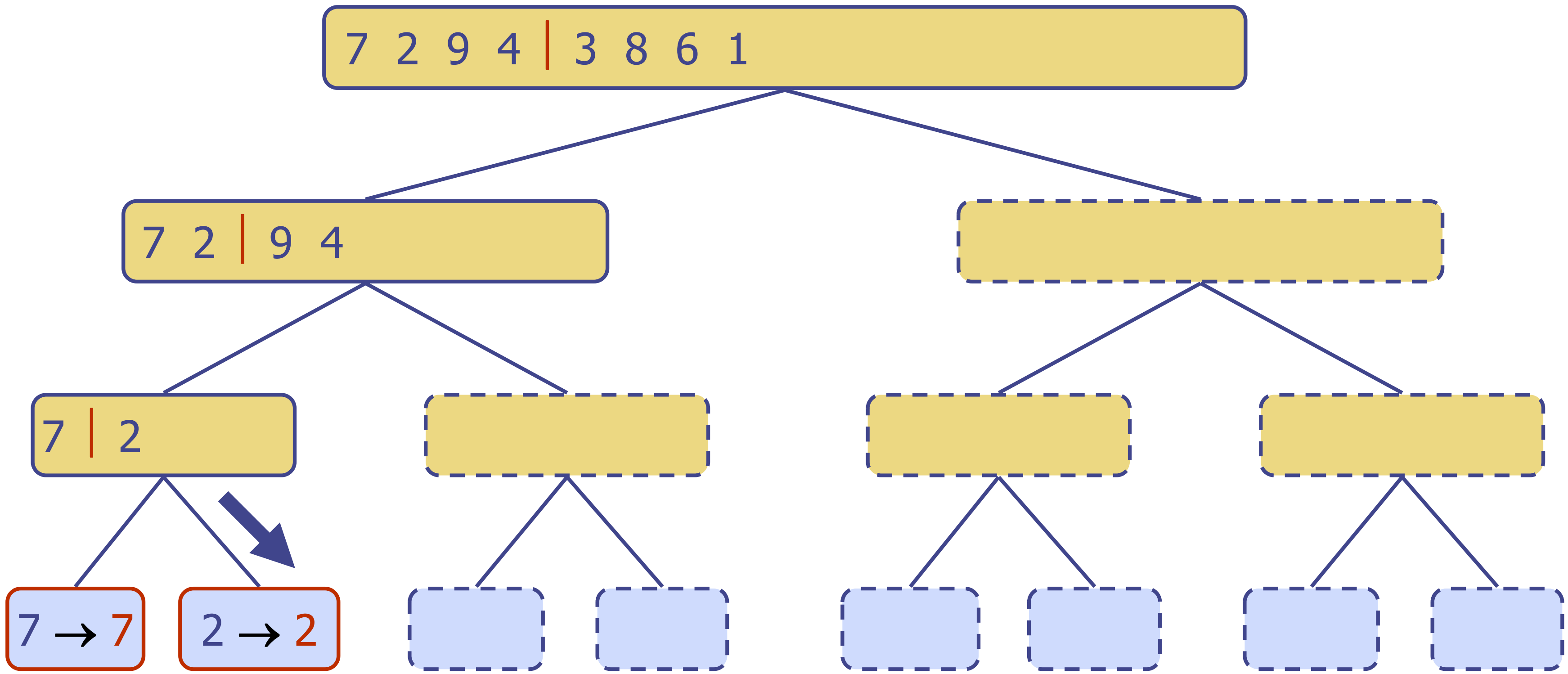
Execution Example (cont.)

- Recursive call, base case



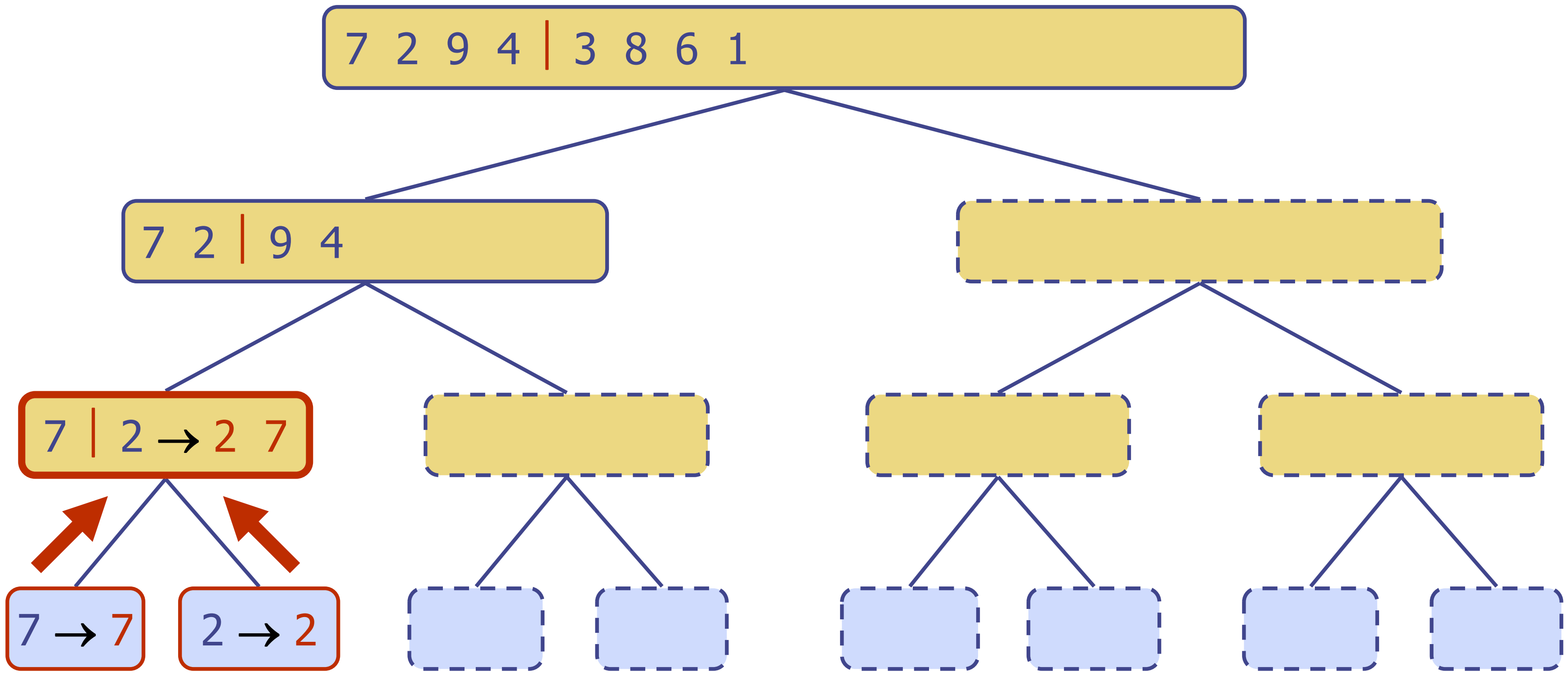
Execution Example (cont.)

- Recursive call, base case



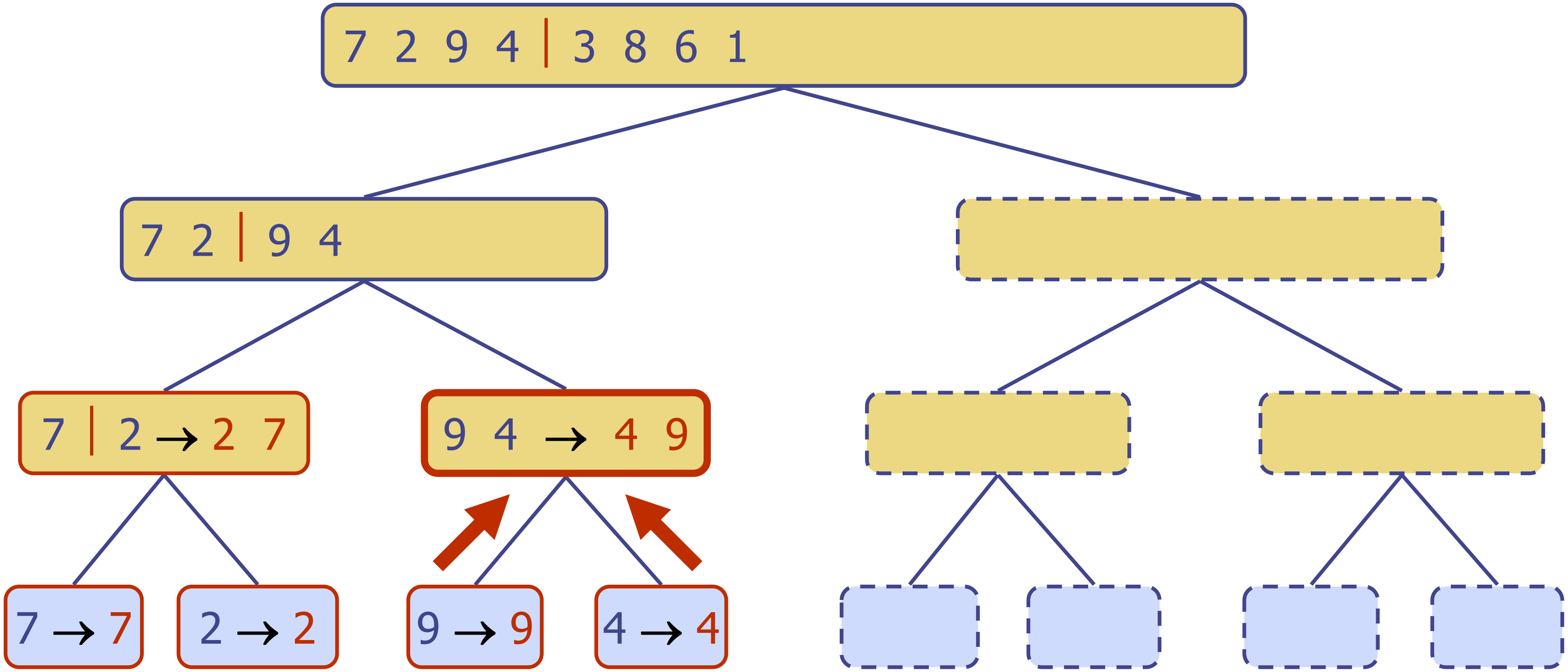
Execution Example (cont.)

- Merge



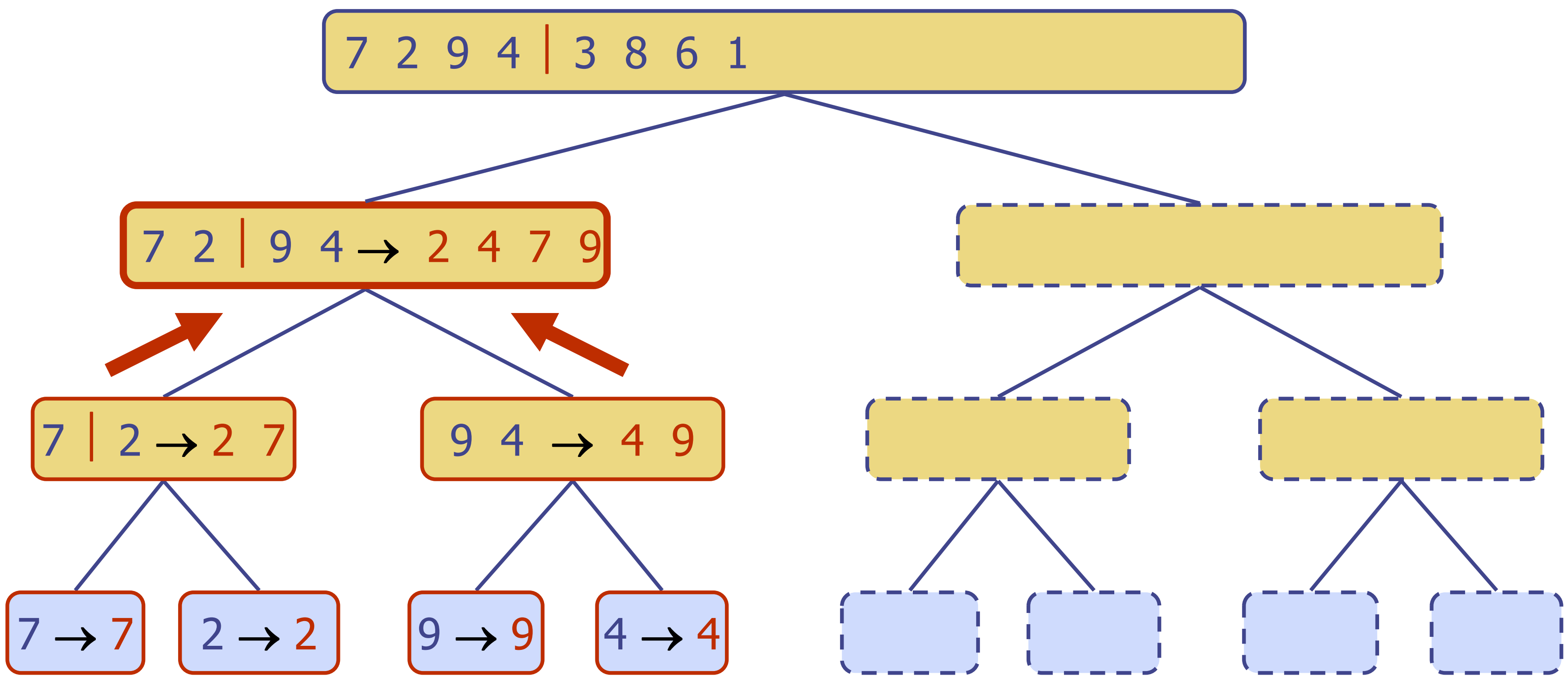
Execution Example (cont.)

- Recursive call, …, base case, merge



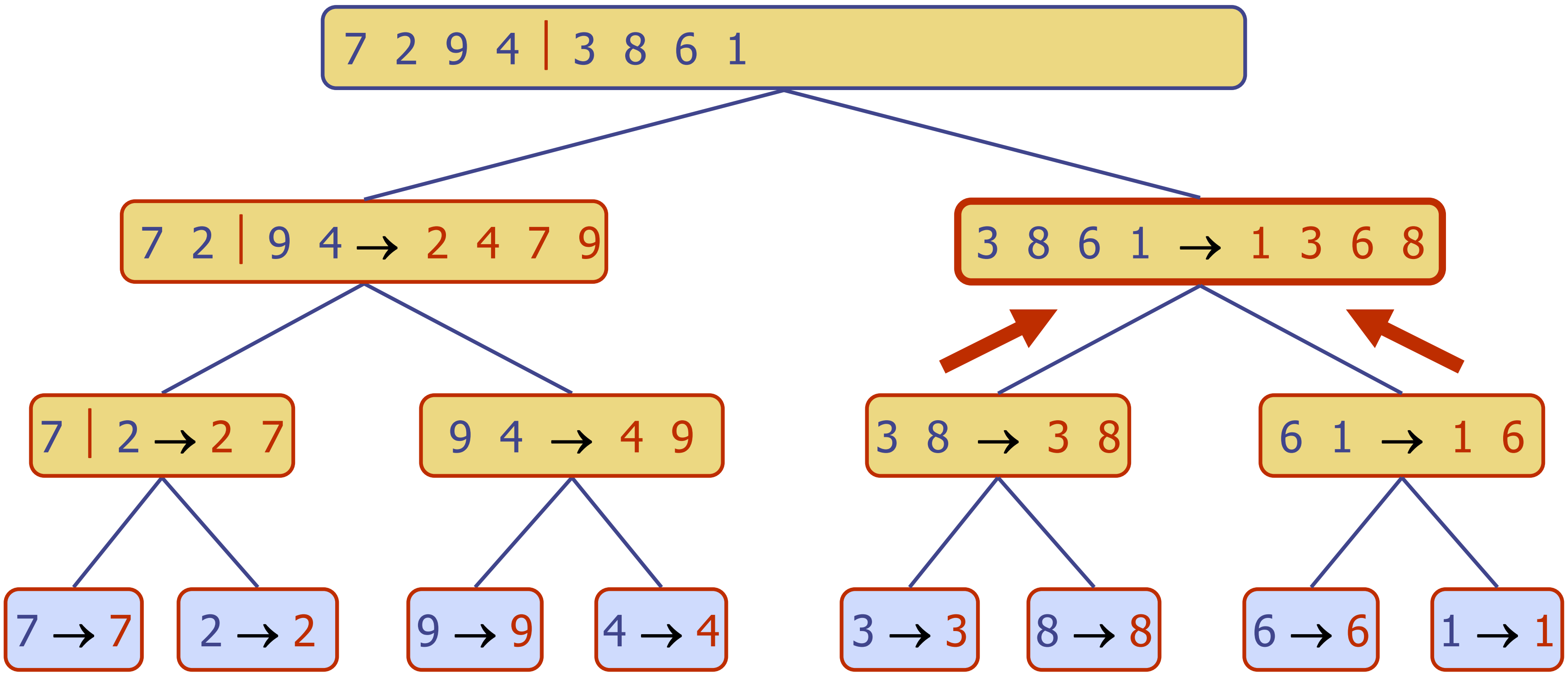
Execution Example (cont.)

- Merge



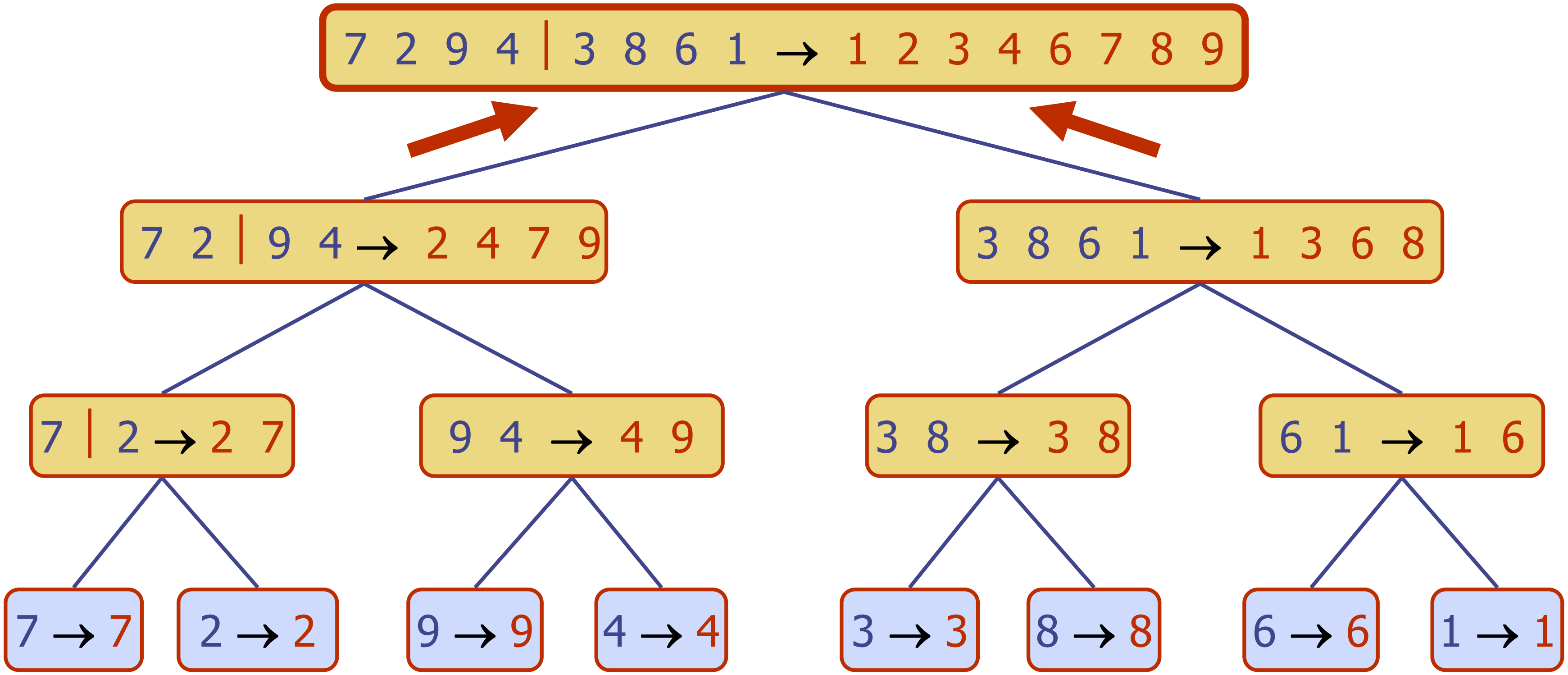
Execution Example (cont.)

- Recursive call, ..., merge, merge



Execution Example (cont.)

- Merge



Merge Sort

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] <= righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",alist)
```

mergeSort function is invoked on left half and right half

merging the two smaller sorted lists into a larger sorted list

Merge Sort

```
alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

Splitting [54, 26, 93, 17, 77, 31, 44, 55, 20]
Splitting [54, 26, 93, 17]
Splitting [54, 26]
Splitting [54]
Merging [54]
Splitting [26]
Merging [26]
Merging [26, 54]
Splitting [93, 17]
Splitting [93]
Merging [93]
Splitting [17]
Merging [17]
Merging [17, 93]
Merging [17, 26, 54, 93]
Splitting [77, 31, 44, 55, 20]
Splitting [77, 31]
Splitting [77]
Merging [77]
Splitting [31]
Merging [31]
Merging [31, 77]
Splitting [44, 55, 20]
Splitting [44]
Merging [44]
Splitting [55, 20]
Splitting [55]
Merging [55]
Splitting [20]
Merging [20]
Merging [20, 55]
Merging [20, 44, 55]
Merging [20, 31, 44, 55, 77]
Merging [17, 20, 26, 31, 44, 54, 55, 77, 93]
[17, 20, 26, 31, 44, 54, 55, 77, 93]

```
alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

```
Splitting [54, 26, 93, 17, 77, 31, 44, 55, 20]
Splitting [54, 26, 93, 17]
Splitting [54, 26]
Splitting [54]
Splitting [26]
Merging [26, 54]
Splitting [93, 17]
Splitting [93]
Splitting [17]
Merging [17, 93]
Merging [17, 26, 54, 93]
Splitting [77, 31, 44, 55, 20]
Splitting [77, 31]
Splitting [77]
Splitting [31]
Merging [31, 77]
Splitting [44, 55, 20]
Splitting [44]
Splitting [55, 20]
Splitting [55]
Splitting [20]
Merging [20, 55]
Merging [20, 44, 55]
Merging [20, 31, 44, 55, 77]
Merging [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Modified Output

Quiz

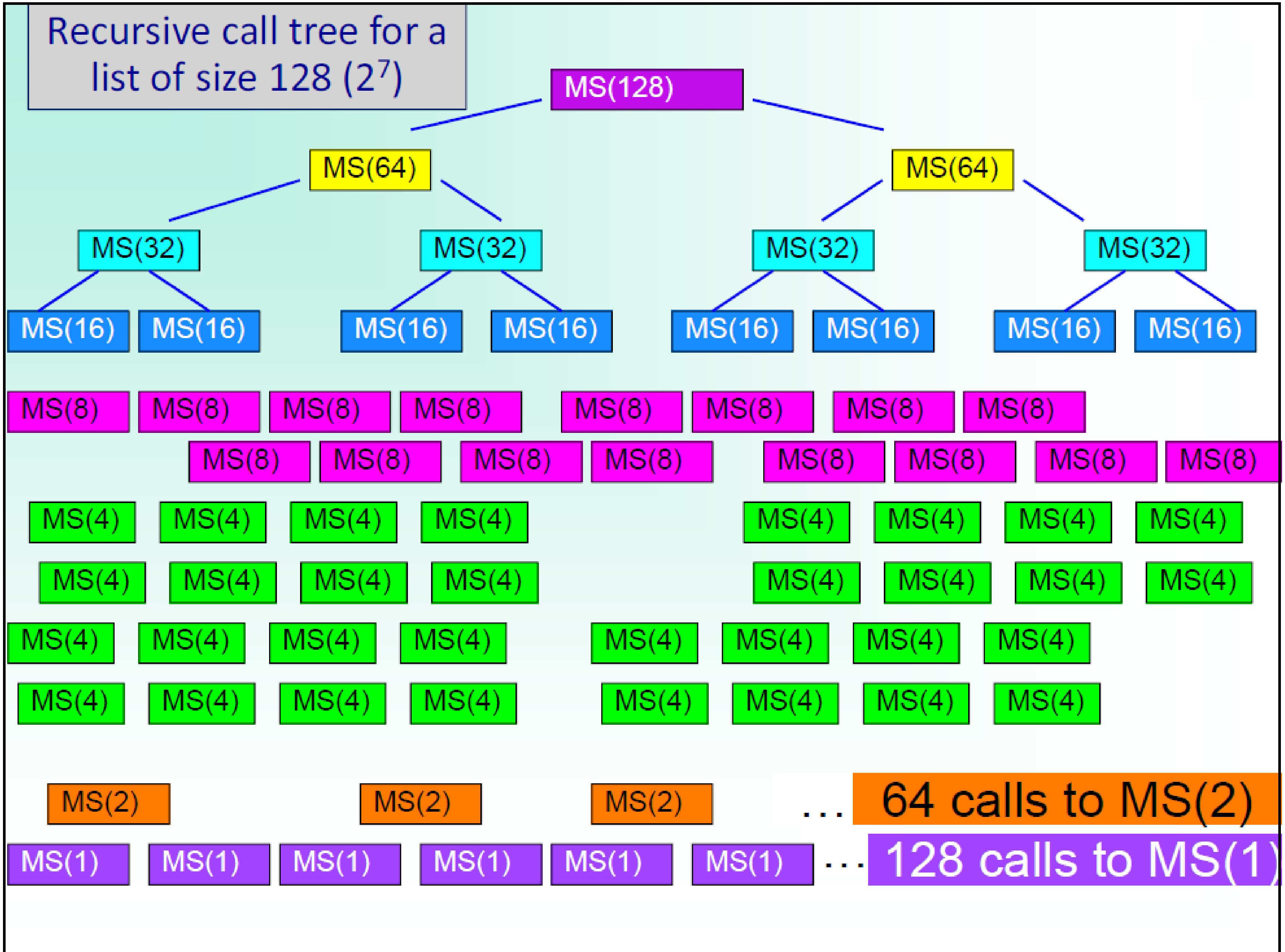
Suppose you have the following list of numbers to sort

[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

which answer illustrates the list to be sorted after 3 recursive calls to mergesort?

- A. [16, 49, 39, 27, 43, 34, 46, 40]
- B. [21,1] √
- C. [21, 1, 26, 45]
- D. [21]

Merge Sort


Recursive call tree for a list of size 128 (2^7)
MS(128)
MS(64) MS(64)
MS(32) MS(32) MS(32) MS(32)
MS(16) MS(16) MS(16) MS(16) MS(16) MS(16) MS(16) MS(16)
MS(8) MS(8) MS(8) MS(8) MS(8) MS(8) MS(8) MS(8)
MS(8) MS(8) MS(8) MS(8) MS(8) MS(8) MS(8) MS(8)
MS(4) MS(4) MS(4) MS(4) MS(4) MS(4) MS(4) MS(4)
MS(4) MS(4) MS(4) MS(4) MS(4) MS(4) MS(4) MS(4)
MS(4) MS(4) MS(4) MS(4) MS(4) MS(4) MS(4) MS(4)
MS(4) MS(4) MS(4) MS(4) MS(4) MS(4) MS(4) MS(4)
MS(2) MS(2) MS(2) . . .
64 calls to MS(2)
MS(1) MS(1) MS(1) MS(1) MS(1) MS(1) . . .
128 calls to MS(1)

Quiz

Suppose you have the following list of numbers to sort

[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

which answer illustrates the first two lists to be merged?

- A. [21, 1] and [26, 45]
- B. [[1, 2, 9, 21, 26, 28, 29, 45] and [16, 27, 34, 39, 40, 43, 46, 49]
- C. [21] and [1] √
- D. [9] and [16]

Merge Sort

- The time for sorting a list of size 1 is constant, i.e. $T(1)=1$
- The time for sorting a list of size n is the time of sorting the two halves plus the time for merging, i.e. $T(n) = 2*T(n/2)+n$
- Can prove: $T(n) = n + n \log n$
- => Big-O is $O(n \log(n))$

Summary

	Best	Worst
Bubble Sort (lecture)	O(*n*^2)	O(*n*^2)
Bubble Sort (optimised)	O(n)	O(*n*^2)
Selection Sort	O(*n*^2)	O(*n*^2)
Insertion Sort	O(*n*)	O(*n*^2)
Shell Sort (best gap sequence)	O(*n*)	O(*n* (log *n*)^2)
Merge Sort	O(*n* log *n*)	O(*n* log *n*)
Tim Sort (used in Python, hybrid of Merge Sort and Insertion Sort)	O(*n*)	O(*n* log *n*)

Note: A comparison based sorting algorithm can NOT be better than O(n log n) in the average and worst case