CSC311: Data Structures

Dr. Amlan Chatterjee

Computer Science Department
California State University, Dominguez Hills

November 1, 2016

Sorting

Merge Sort

Merge

- A *merge* is a common data processing operation performed on two sequences of data with the following characteristics
  - Both sequences contain items with a common `compareTo` method
  - The objects in both sequences are ordered in accordance with this `compareTo` method
- The result is a third sequence containing all the data from the first two sequences

Merge Algorithm

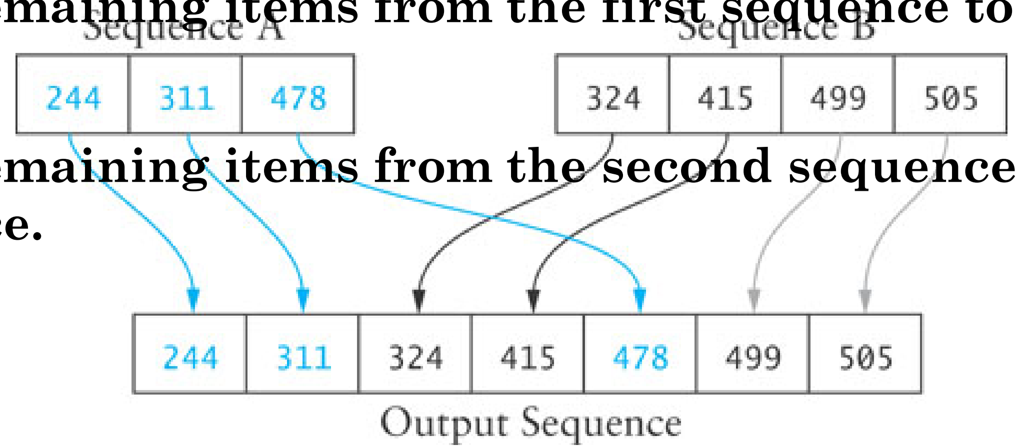
1. **Access the first item from both sequences.**

2. **`while` not finished with either sequence**

3. **Compare the current items from the two sequences, copy the smaller
   current item to the output sequence, and access the next item from the
   input sequence whose item was copied.**

4. **Copy any remaining items from the first sequence to the output sequence.**

5. **Copy any remaining items from the second sequence to the output sequence.**



Analysis of Merge

- For two input sequences each containing n elements, each element needs to move from its input sequence to the output sequence
- Merge time is $O(n)$
- Space requirements
  - The array cannot be merged in place
  - Additional space usage is $O(n)$

Merge Sort

- We can modify merging to sort a single, unsorted array
    1. Split the array into two halves
    2. Sort the left half
    3. Sort the right half
    4. Merge the two

- This algorithm can be written with a recursive step

(recursive) Algorithm for Merge Sort

Algorithm for Merge Sort

1. `if` the `tableSize` is > 1
2. Set `halfSize` to `tableSize` divided by 2.
3. Allocate a table called `leftTable` of size `halfSize`.
4. Allocate a table called `rightTable` of size `tableSize - halfSize`.
5. Copy the elements from `table[0 ... halfSize - 1]` into `leftTable`.
6. Copy the elements from `table[halfSize ... tableSize]` into `rightTable`.
7. Recursively apply the merge sort algorithm to `leftTable`.
8. Recursively apply the merge sort algorithm to `rightTable`.
9. Apply the merge method using `leftTable` and `rightTable` as the input and the original table as the output.

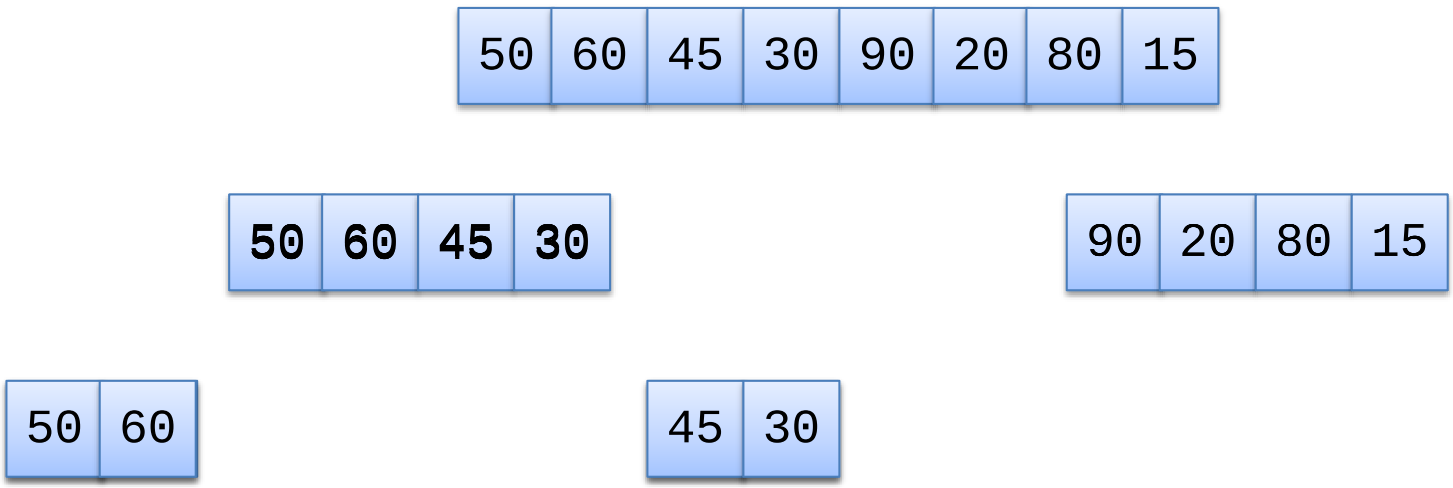
Trace of Merge Sort

50	**60**	**45**	**30**	**90**	**20**	**80**	**15**

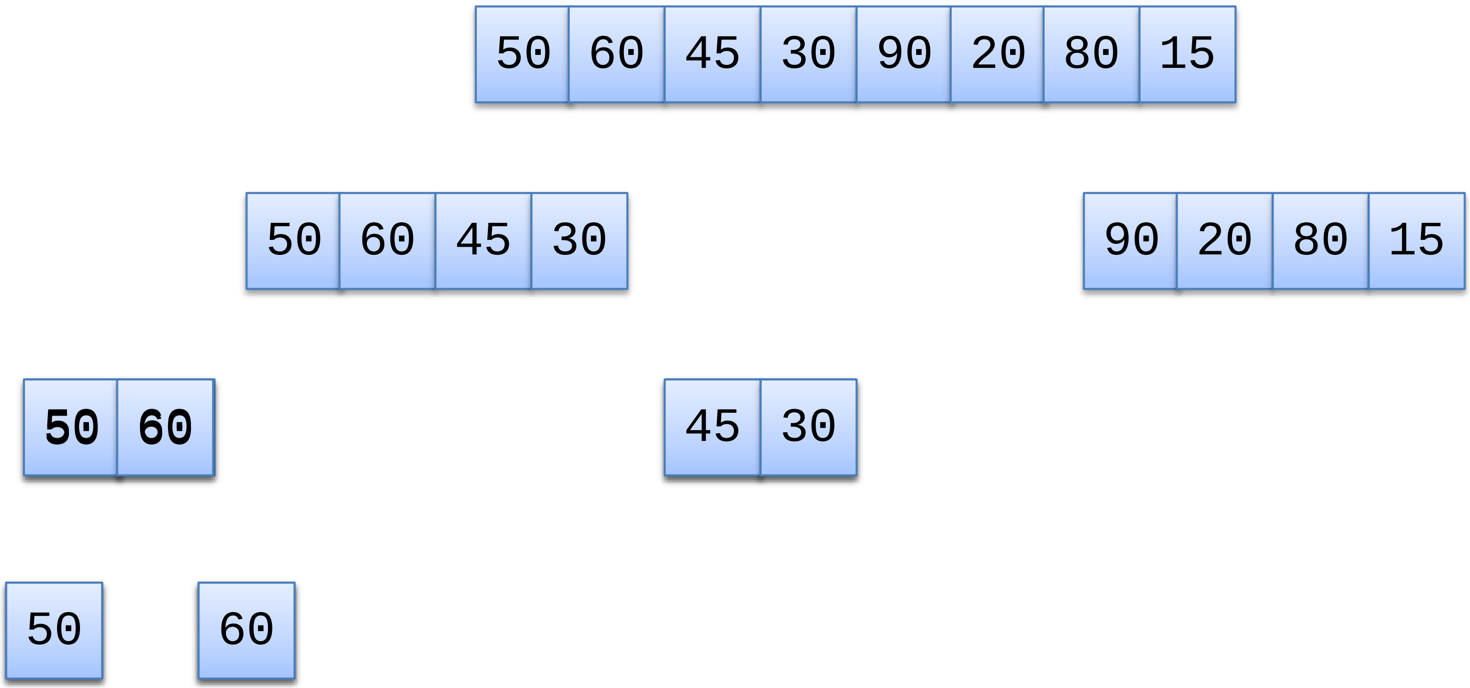
50	60	45	30

90	20	80	15

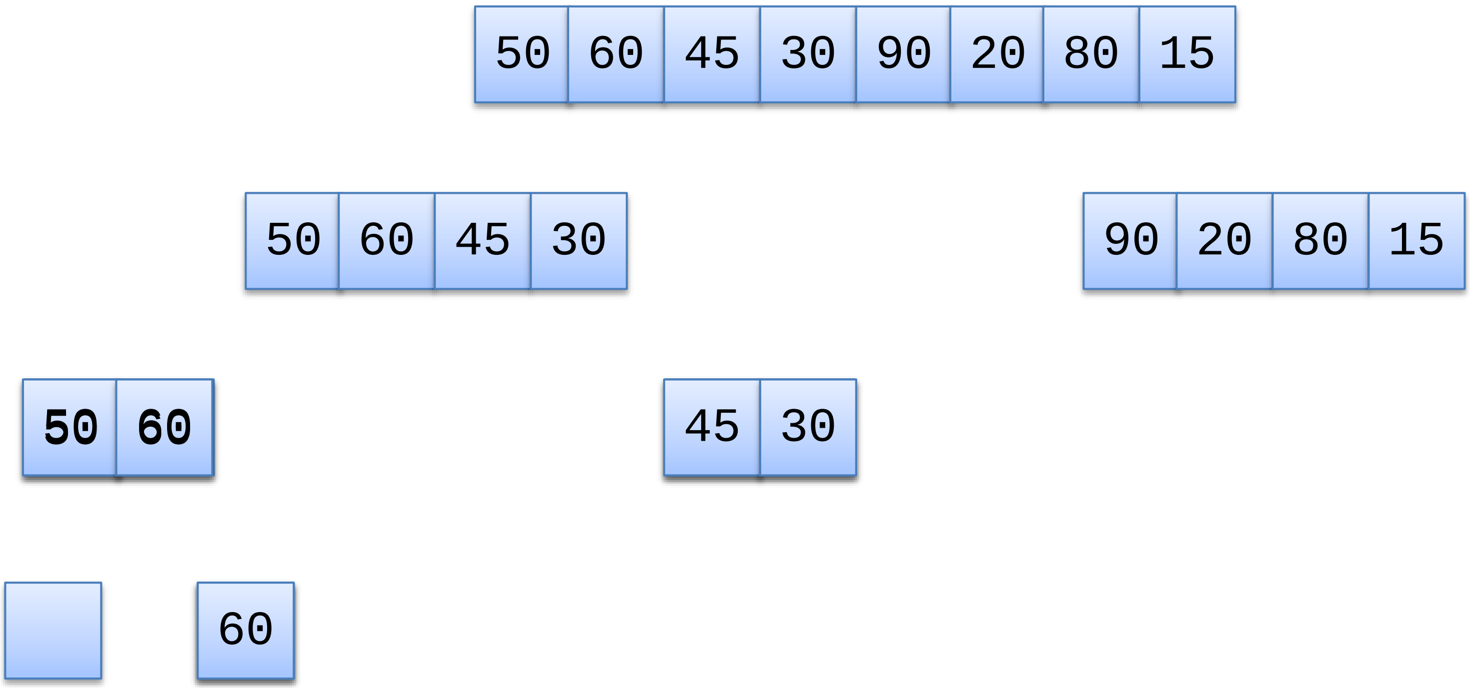
Trace of Merge Sort (cont.)



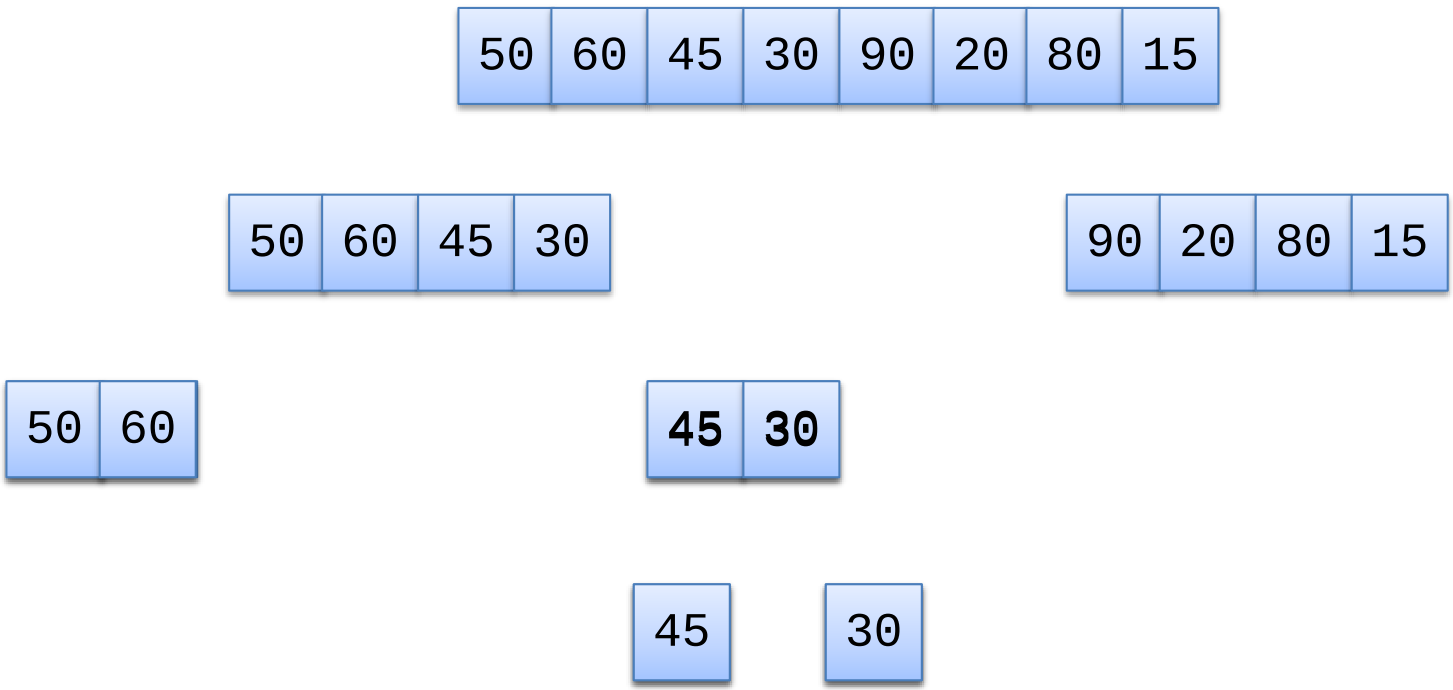
Trace of Merge Sort (cont.)



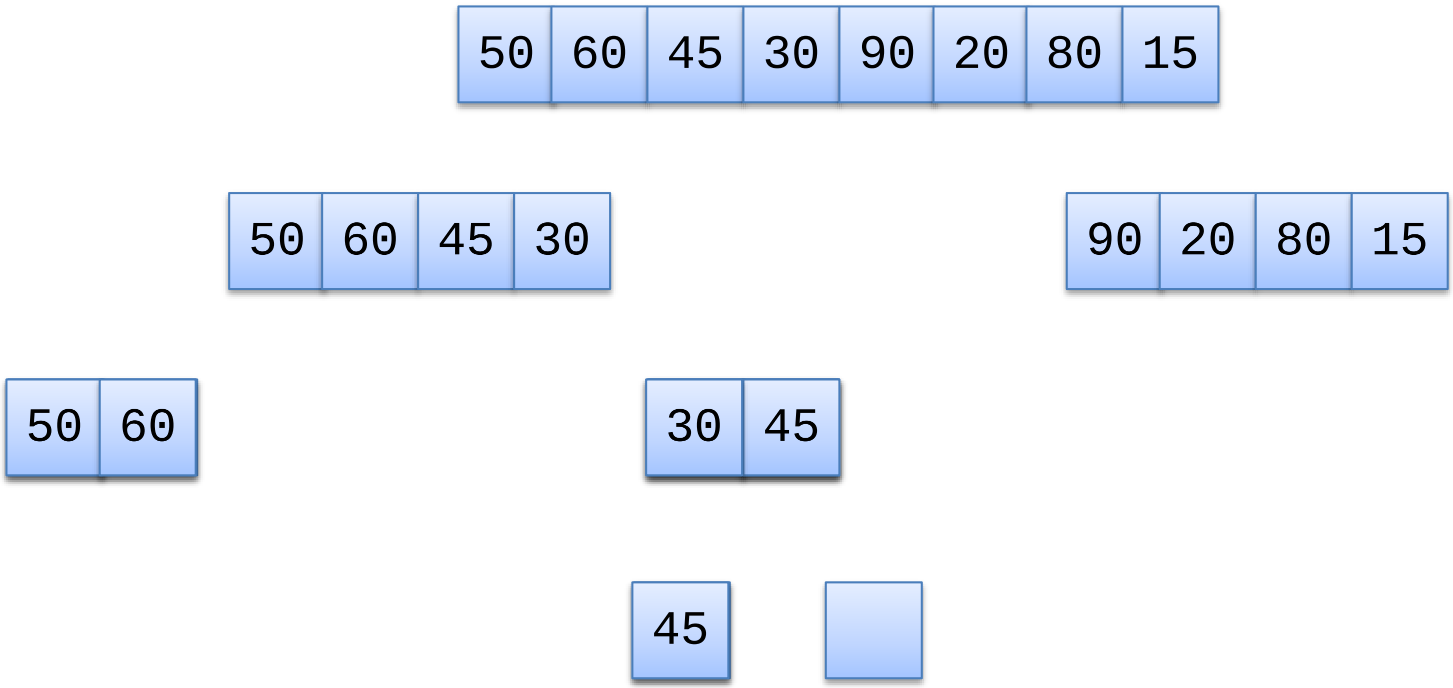
Trace of Merge Sort (cont.)



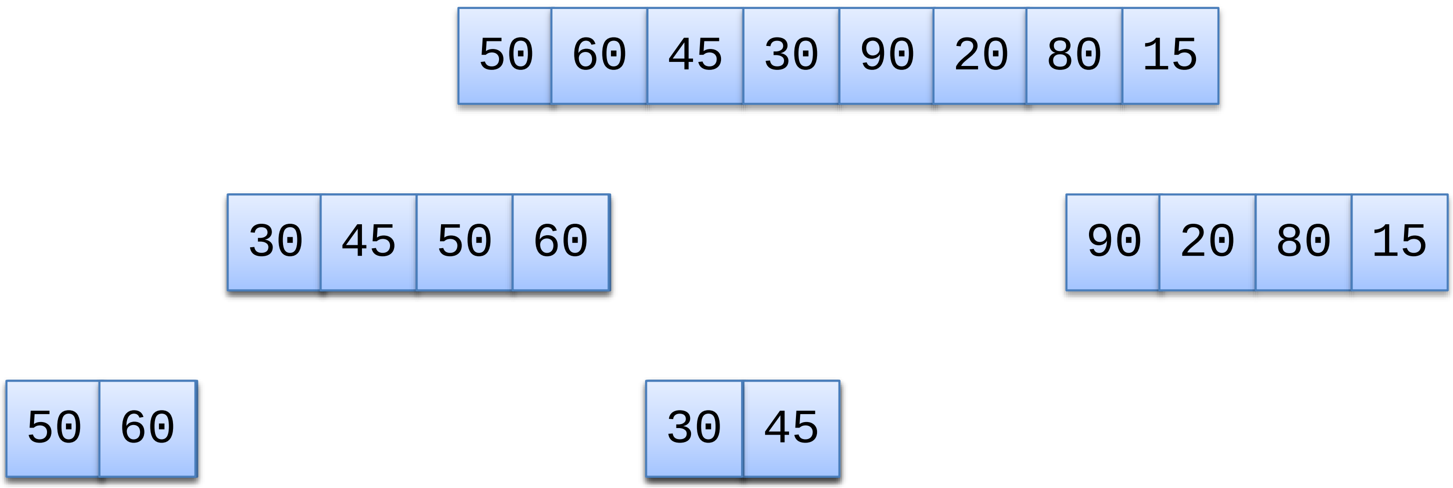
Trace of Merge Sort (cont.)



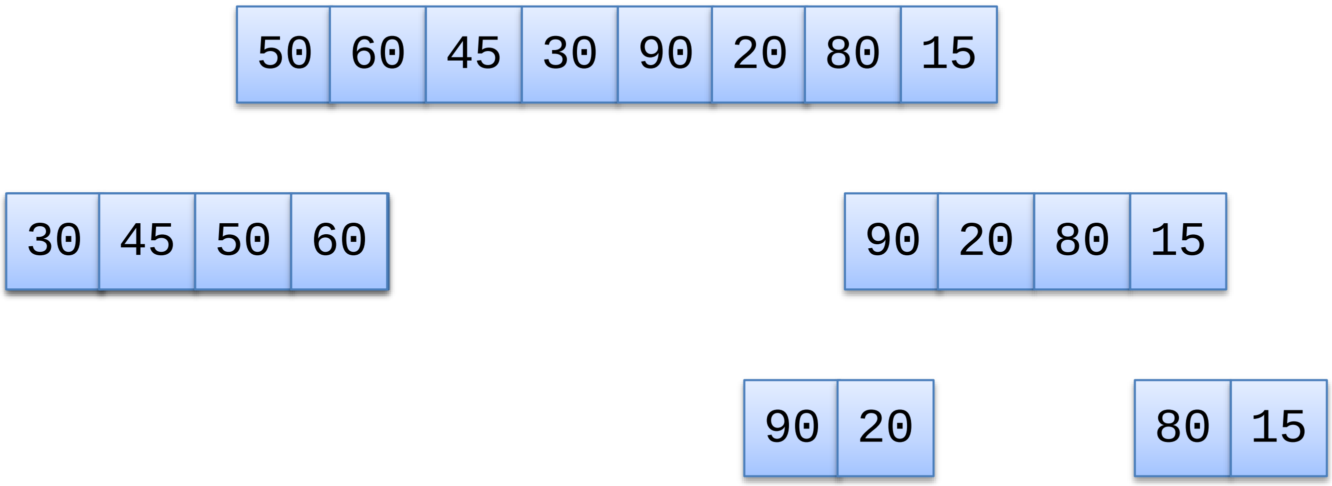
Trace of Merge Sort (cont.)



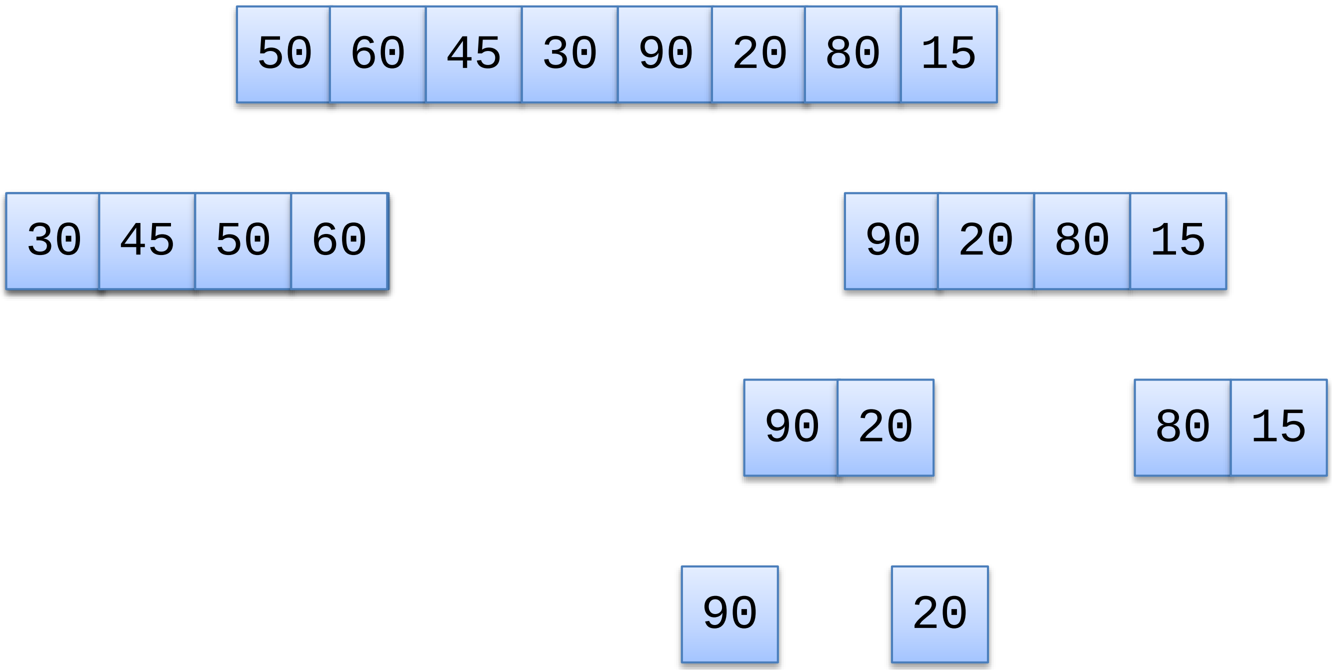
Trace of Merge Sort (cont.)



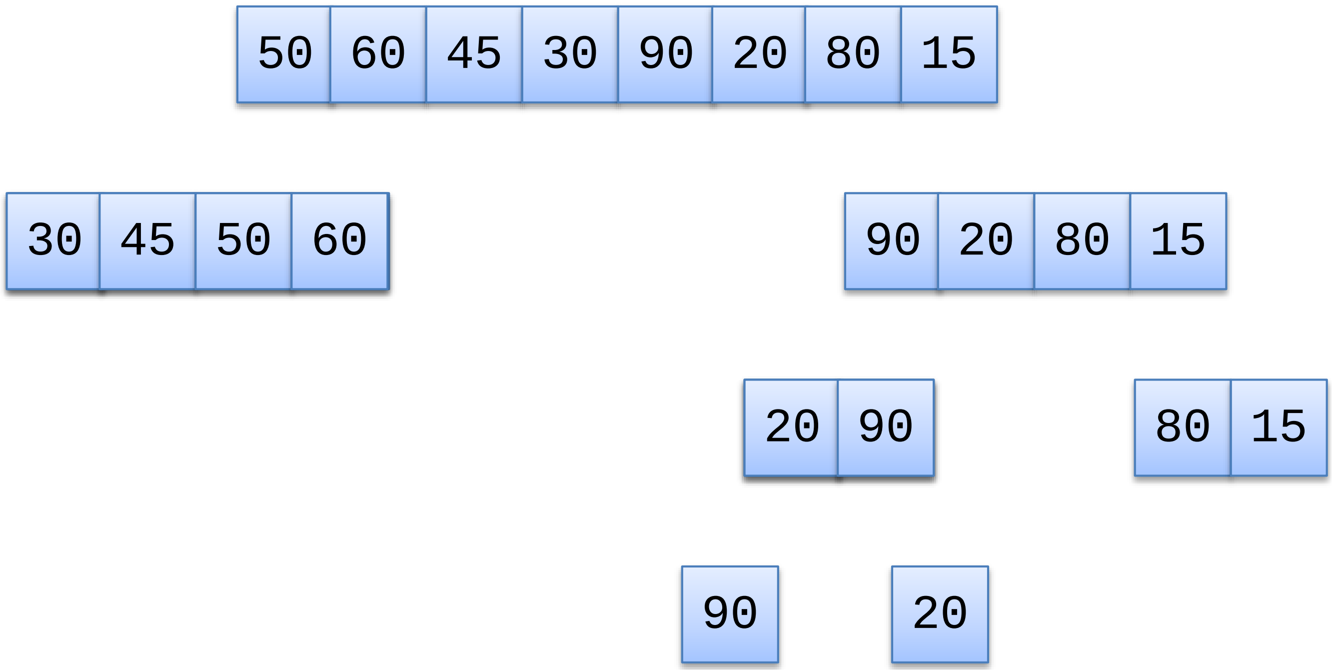
Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)

50	60	45	30	90	20	80	15

30	45	50	60

90	20	80	15

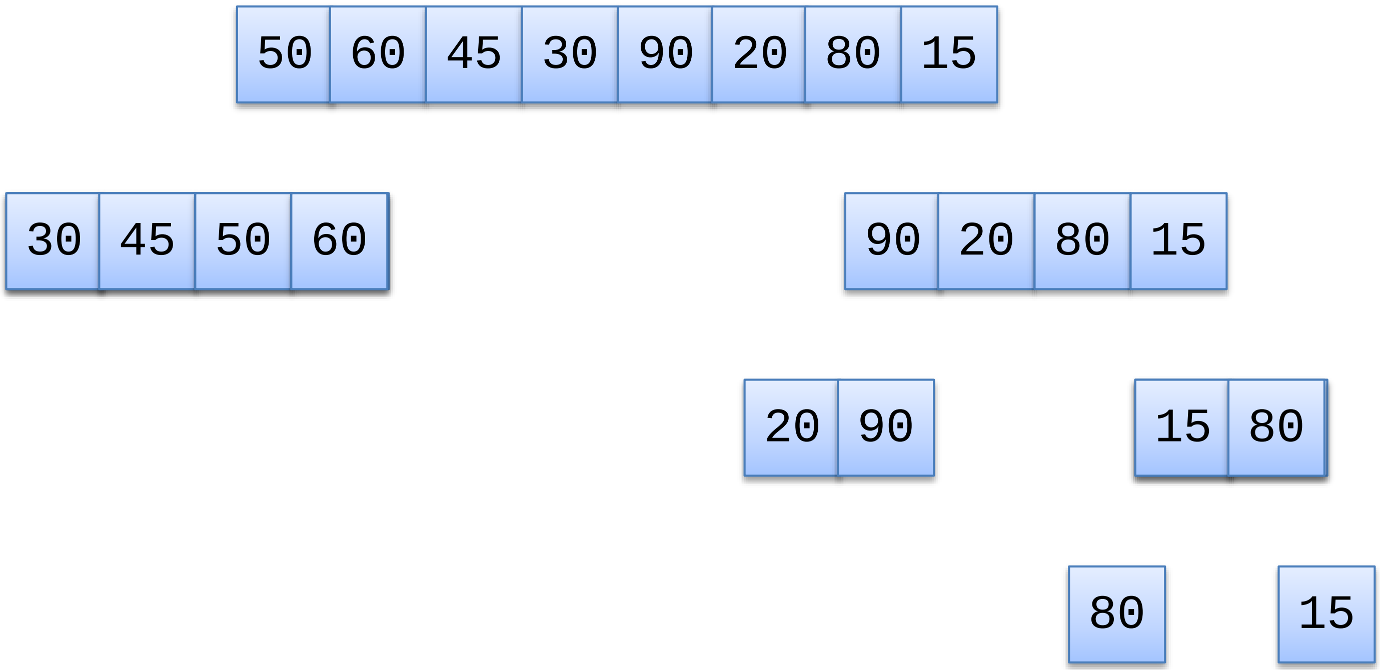
20	90

80	15

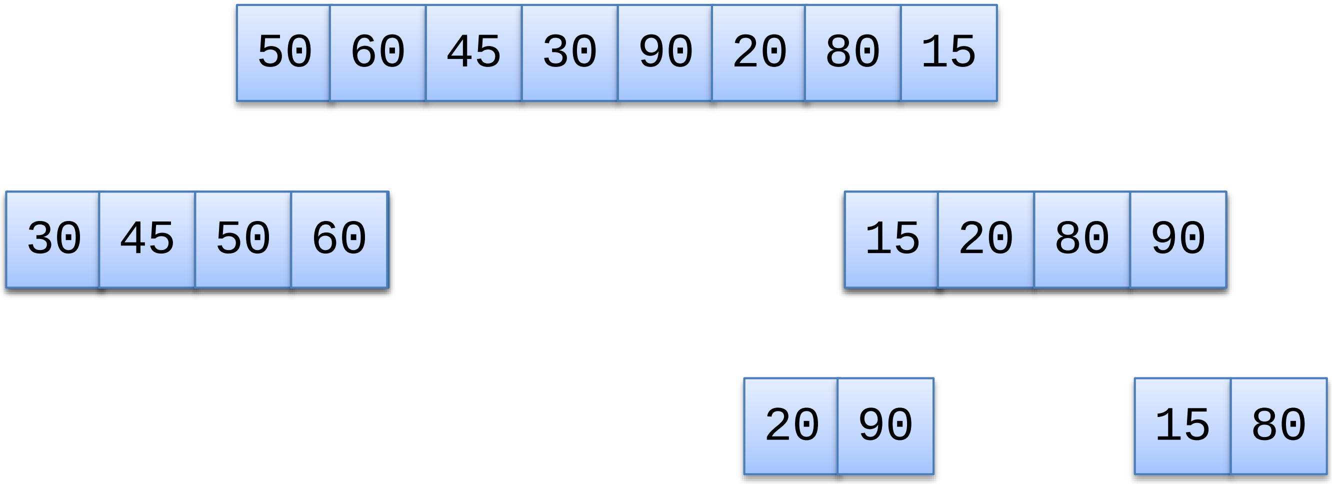
80

15

Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)

15	20	30	45	50	60	80	90

30	45	50	60

15	20	80	90

Analysis of Merge Sort

- Each backward step requires a movement of n elements from smaller-size arrays to larger arrays; the effort is $O(n)$
- The number of steps which require merging is $\log n$ because each recursive call splits the array in half
- The total effort to reconstruct the sorted array through merging is $O(n \log n)$

Quicksort

Quicksort

- Developed in 1962
- Quicksort selects a specific value called a pivot and rearranges the array into two parts (called *partioning*)
  - all the elements in the left subarray are less than or equal to the pivot
  - all the elements in the right subarray are larger than the pivot
  - The pivot is placed between the two subarrays
- The process is repeated until the array is sorted

Trace of Quicksort

44	75	23	43	55	12	64	77	33

Trace of Quicksort (cont.)

44	75	23	43	55	12	64	77	33

Arbitrarily select the first element as the pivot

Trace of Quicksort (cont.)

55	75	23	43	44	12	64	77	33

Swap the pivot with the element in the middle

Trace of Quicksort (cont.)

55	75	23	43	44	12	64	77	33

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right

Trace of Quicksort (cont.)

12	33	23	43	44	55	64	77	75

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right

Trace of Quicksort (cont.)

44 is now in its correct position

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

12	33	23	43	44	55	64	77	75

Now apply quicksort recursively to the two subarrays

Trace of Quicksort (cont.)

Pivot value = 12

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 12

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Left and right subarrays have single values; they are sorted

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Left and right subarrays have single values; they are sorted

Trace of Quicksort (cont.)

Pivot value = 55

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 64

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 77

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 77

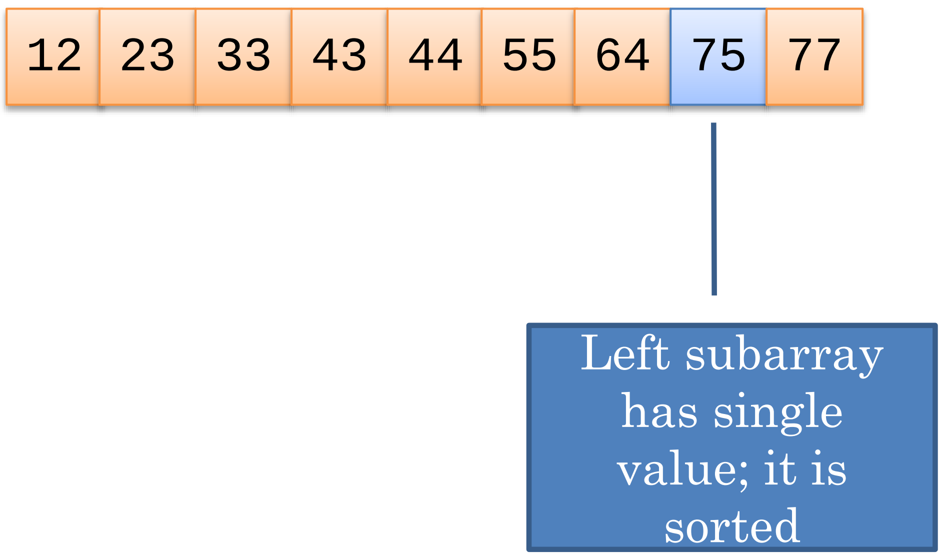
12	23	33	43	44	55	64	75	77

Trace of Quicksort (cont.)

Pivot value = 77

12	23	33	43	44	55	64	75	77

Trace of Quicksort (cont.)



Trace of Quicksort (cont.)

12	23	33	43	44	55	64	75	77

Algorithm for Quicksort

- The indexes `first` and `last` are the end points of the array being sorted
- The index of the pivot after partitioning is `pivIndex`

Algorithm for Quicksort

1. **`if first < last` then**
2. **Partition the elements in the subarray `first . . . last` so that the pivot value is in its correct place (subscript `pivIndex`)**
3. **Recursively apply quicksort to the subarray `first . . . pivIndex - 1`**
4. **Recursively apply quicksort to the subarray `pivIndex + 1 . . . last`**

Analysis of Quicksort

- If the pivot value is a random value selected from the current subarray,
  - then statistically half of the items in the subarray will be less than the pivot and half will be greater
- If both subarrays have the same number of elements (best case), there will be log n levels of recursion
- At each recursion level, the partitioning process involves moving every element to its correct position—n moves
- Quicksort is O(n log n), just like merge sort

Analysis of Quicksort (cont.)

- The array split may not be the best case, i.e. 50-50
- An exact analysis is difficult (and beyond the scope of this class), but, the running time will be bounded by a constant x n log n

Analysis of Quicksort (cont.)

- A quicksort will give very poor behavior if, each time the array is partitioned, a subarray is empty.
- In that case, the sort will be O(n^2)
- Under these circumstances, the overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of quicksort a poor performer relative to the quadratic sorts
  - Use good partition techniques

Testing the Sort Algorithms

- Use a variety of test cases
  - small and large arrays
  - arrays in random order
  - arrays that are already sorted
  - arrays with duplicate values
- Compare performance on each type of array