Sorting

Reference: "Data Structures and Algorithm Analysis", C. Shaffer, pp. 223–244; skip sections 7.2.2 and 7.3.

Introduction

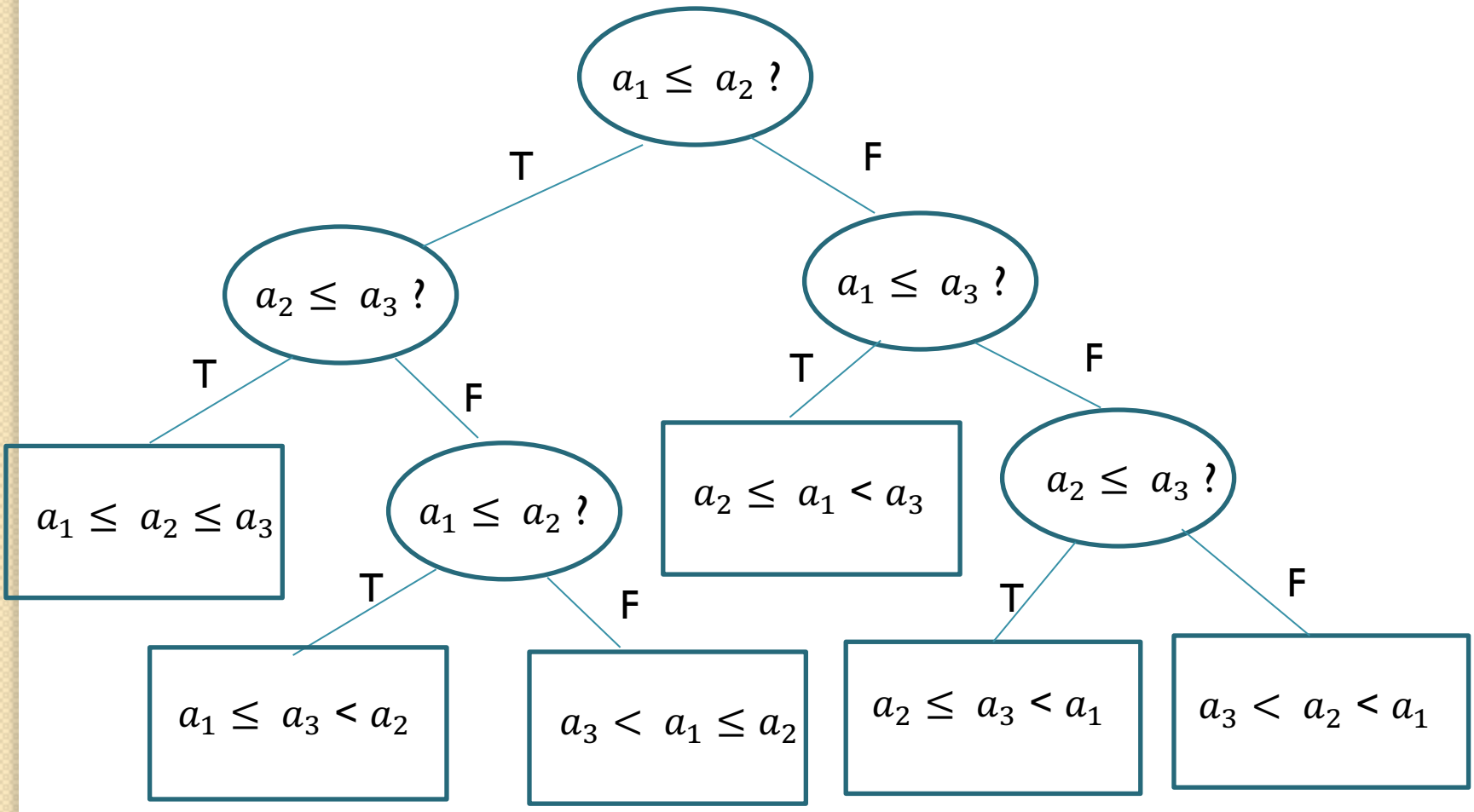
- Sorting entails arranging data in order
- Familiarity with sorting algorithms is an important programming skill
- The study of sorting algorithms provides insight
  - into problem solving techniques such as *divide and conquer*
  - into the analysis and comparison of algorithms which perform the same task

Using Java Sorting Methods

- The Java API provides a class `Arrays` with several overloaded sort methods for different array types
- The `Collections` class provides similar sorting methods for `Lists`
- Sorting methods for arrays of primitive types are based on the quicksort algorithm
- Sorting methods for arrays of objects and `Lists` are based on the merge sort algorithm
- Both algorithms are $O(n \log n)$

Sorting lower bound

- How fast can we sort elements ?
- If we don't know the distribution or range of input elements a priori, we can derive lower bound based on comparisons only.
- Suppose we have input elements $a_1, a_2, a_3 \ldots .. a_n$
- Any comparison based algorithm can be reduced to a decision tree where in each node we check if $a_i \leq a_j$ for two elements a_i and a_j . Only 2 outcomes.
- Leaves of decision tree correspond to sorting order.
- Tree must have at least $n!$ leaves (permutations of n input elements) and must have height ≥ [log n!]
- Worst-case time complexity is Ω(h) where h is height of decision tree
- By Stirling's formula: n! ≈ $(\frac{n}{e})^n \rightarrow$ any sorting algorithm must have of # of comparisons ≥ $n \log n - 1.44n$



Decision tree for a 3-element sorting algorithm

Comparison based Sorting

- Criteria:

(a) # of comparisons

(b) # of data movements

(c) Additional space required

(d) in-memory vs external sorting

(e) stable sorting

- O(n^2) sorting algorithms

1. Bubble Sort (lot of comparisons + data movements),
2. Selection Sort (lot of comparisons + less data movements),
3. Insertion Sort (on average less comparisons + data movements); takes advantage of input ordering

- O(n log n) sorting algorithms

Heap sort, Merge Sort, Quick Sort

Selection Sort

- It sorts an array by making several passes through the array, selecting a next smallest item in the array each time and placing it where it belongs in the array

n	5
fill	
posMin	
next	

0	1	2	3	4
35	65	30	60	20

1. **for** `fill = 0` to `n - 2` do
2. Initialize `posMin` to `fill`
3. **for** `next = fill + 1` to `n - 1` do
4. **if** the item at `next` is less than the item at `posMin`
5. Reset `posMin` to next
6. Exchange the item at `posMin` with the one at `fill`

Source : Dr. Calvin & textbook

Analysis of Selection Sort

This loop is performed n-1 times

1. **for** `fill = 0` to `n - 2` do
2. Initialize `posMin` to `fill`
3. **for** `next = fill + 1` to `n - 1` do
4. **if** the item at `next` is less than the item at `posMin`
5. Reset `posMin` to next
6. Exchange the item at `posMin` with the one at `fill`

Analysis of Selection Sort (cont.)

There are n-1 exchanges

1. **for** `fill = 0` to `n - 2` do
2. Initialize `posMin` to `fill`
3. **for** `next = fill + 1` to `n - 1` do
4. **if** the item at `next` is less than the item at `posMin`
5. Reset `posMin` to next
6. Exchange the item at `posMin` with the one at `fill`

Analysis of Selection Sort (cont.)

This comparison is performed
(n – 1 - *fill*)
times for each value of *fill* and can be represented by the following series:
(n-1) + (n-2) + ... + 3 + 2 + 1

1. **for** `fill = 0` to `n - 2` do
2. Initialize `posMin` to `fill`
3. **for** `next = fill + 1` to `n - 1` do
4. **if** the item at `next` is less than the item at `posMin`
5. Reset `posMin` to next
6. Exchange the item at `posMin` with the one at `fill`

Analysis of Selection Sort (cont.)

For very large *n* we can ignore all but the most significant term in the expression, so the number of

- comparisons is $O(n^2)$
- exchanges is $O(n)$

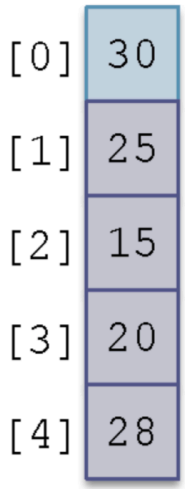
An $O(n^2)$ sort is called a *quadratic sort*

1. **for** `fill = 0` to `n - 2` do
2. Initialize `posMin` to `fill`
3. **for** `next = fill + 1` to `n - 1` do
4. **if** the item at `next` is less than the item at `posMin`
5. Reset `posMin` to next
6. Exchange the item at `posMin` with the one at `fill`

Insertion Sort

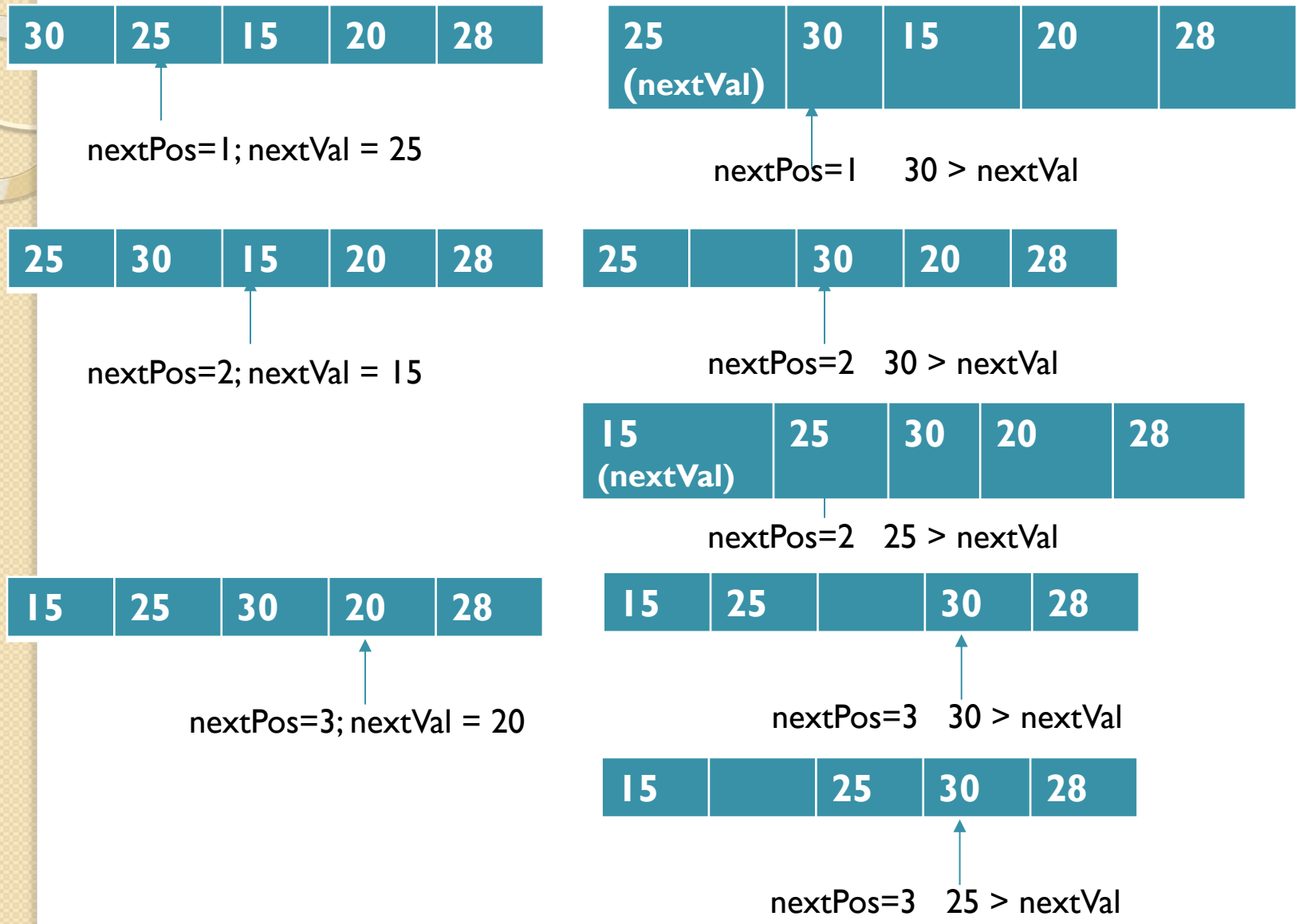
- Another quadratic sort, *insertion* sort, is based on the technique used by card players to arrange a hand of cards
  - The player keeps the cards that have been picked up so far in sorted order
  - When the player picks up a new card, the player makes room for the new card and then inserts it in its proper place

Insertion sort algorithm

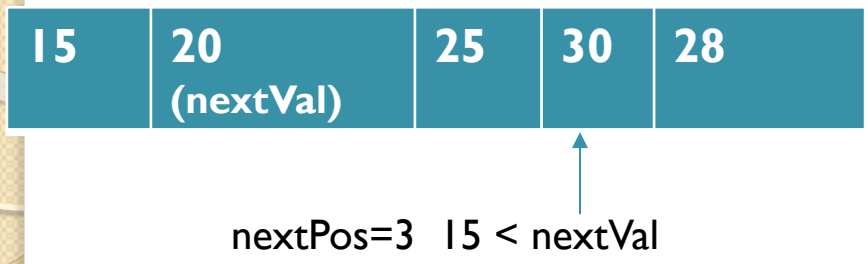


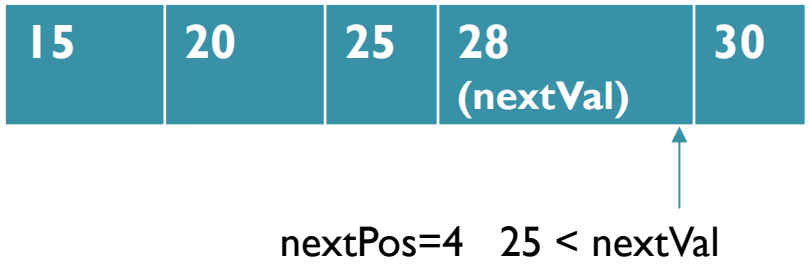
1. **for** each array element from the second (`nextPos = 1`) to the last
2. `nextPos` is the position of the element to insert
3. Save the value of the element to insert in `nextVal`
4. **while** `nextPos > 0` and the element at `nextPos - 1 > nextVal`
5. Shift the element at `nextPos - 1` to position `nextPos`
6. Decrement `nextPos` by `1`
7. Insert `nextVal` at `nextPos`

Insertion sort example



Insertion sort example (contd.)





Analysis of Insertion Sort

- The insertion step is performed $n - 1$ times
- In the worst case, all elements in the sorted subarray are compared to `nextVal` for each insertion
- The maximum number of comparisons will then be:

$$1 + 2 + 3 + \ldots + (n - 2) + (n - 1)$$

- which is $O(n^2)$

Comparison of $O(n^2)$ sorting methods

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Source : textbook

Heap sort

- Uses max-heap

```
static <E extends Comparable<? super E>>
void heapsort(E[] A) {
  // The heap constructor invokes the buildheap method
  MaxHeap<E> H = new MaxHeap<E>(A, A.length, A.length);
  for (int i=0; i<A.length; i++)  // Now sort
    H.removemax(); // Removemax places max at end of heap
}
```

```
/** Constructor supporting preloading of heap contents */
public MaxHeap(E[] h, int num, int max)
{ Heap = h;  n = num;  size = max;  buildheap(); }
```

Source : Dr. Calvin & textbook

Heapsort time complexity

Time complexity : O(n) for heap construction and O(log n) in each iteration of for loop

- Total time complexity : O(n log n) worst-case
- In-place sorting

Heap sort example

After heapify() to create max heap :

i=0; A[8]← removeMax(A[1..8])

i=1; A[7]← removeMax(A[1..7])

i=2; A[6]← removeMax(A[1..6])

i=3; A[5]← removeMax(A[1..5])

i=4; A[4]← removeMax(A[1..4])

i=5; A[3]← removeMax(A[1..3])

i=6; A[2]← removeMax(A[1..2])

Recursive Sort

RecursiveSort(S):

Input : Sequence S of n elements to be sorted

Output : Sorted sequence of S

```
if |S| =1
   return S
else
    (S1,S2) <- Split(S)  // split S into S1 and S2 subsequences of roughly equal size
    RecursiveSort(S1)
    RecursiveSort(S2)
 return  Join(S1,S2) // join S1 and S2
```

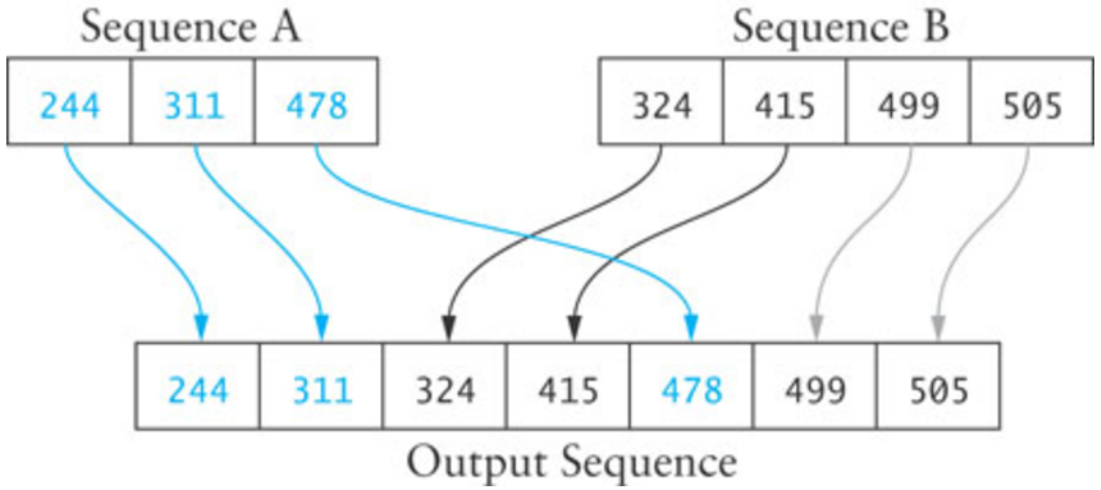
- In MergeSort, join step takes O(n) time
- In QuickSort, split step takes O(n) time
- Recurrence : $T(n) \leq 2T(n/2) + k\,n \Rightarrow T(n)$ is $O(n \log n)$

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*)

Code for Merge


```
/** Merge two sequences.
     pre: leftSequence and rightSequence are sorted.
     post: outputSequence is the merged result and is sorted.
     @param outputSequence The destination
     @param leftSequence The left input
     @param rightSequence The right input
 */
 private static < T
    extends Comparable < T >> void merge(T[] outputSequence,
                                         T[] leftSequence,
                                         T[] rightSequence) {
```

Code for Merge


```
int i = 0; // Index into the left input sequence.
    int j = 0; // Index into the right input sequence.
    int k = 0; // Index into the output sequence.

    // While there is data in both input sequences
    while (i < leftSequence.length && j <
              rightSequence.length) {
      // Find the smaller and
      // insert it into the output sequence.
      if (leftSequence[i].compareTo(rightSequence[j]) < 0) {
        outputSequence[k++] = leftSequence[i++];
      }
```

Code for Merge

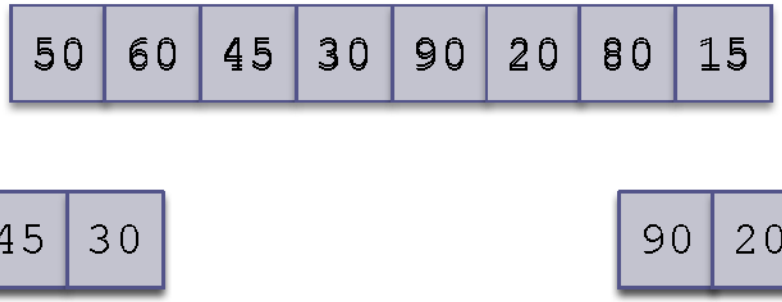

```
else {
        outputSequence[k++] = rightSequence[j++];
      }
    }
    // assert: one of the sequences has more items to copy.
    // Copy remaining input from left sequence to the output.
    while (i < leftSequence.length) {
      outputSequence[k++] = leftSequence[i++];
    }
    // Copy remaining input from right sequence into output.
    while (j < rightSequence.length) {
      outputSequence[k++] = rightSequence[j++];
    }  }   }
```

Merge Sort

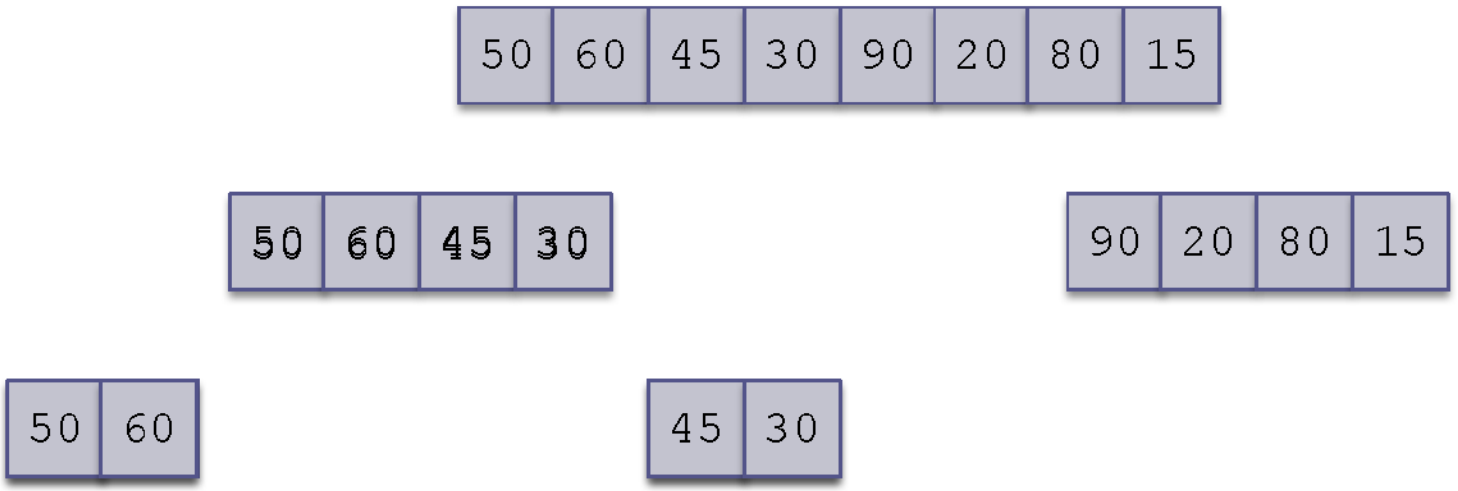
- We can modify merging to sort a single, unsorted array
  1. Split the array into two halves
  2. Sort the left half
  3. Sort the right half
  4. Merge the two

- This algorithm can be written with a recursive step

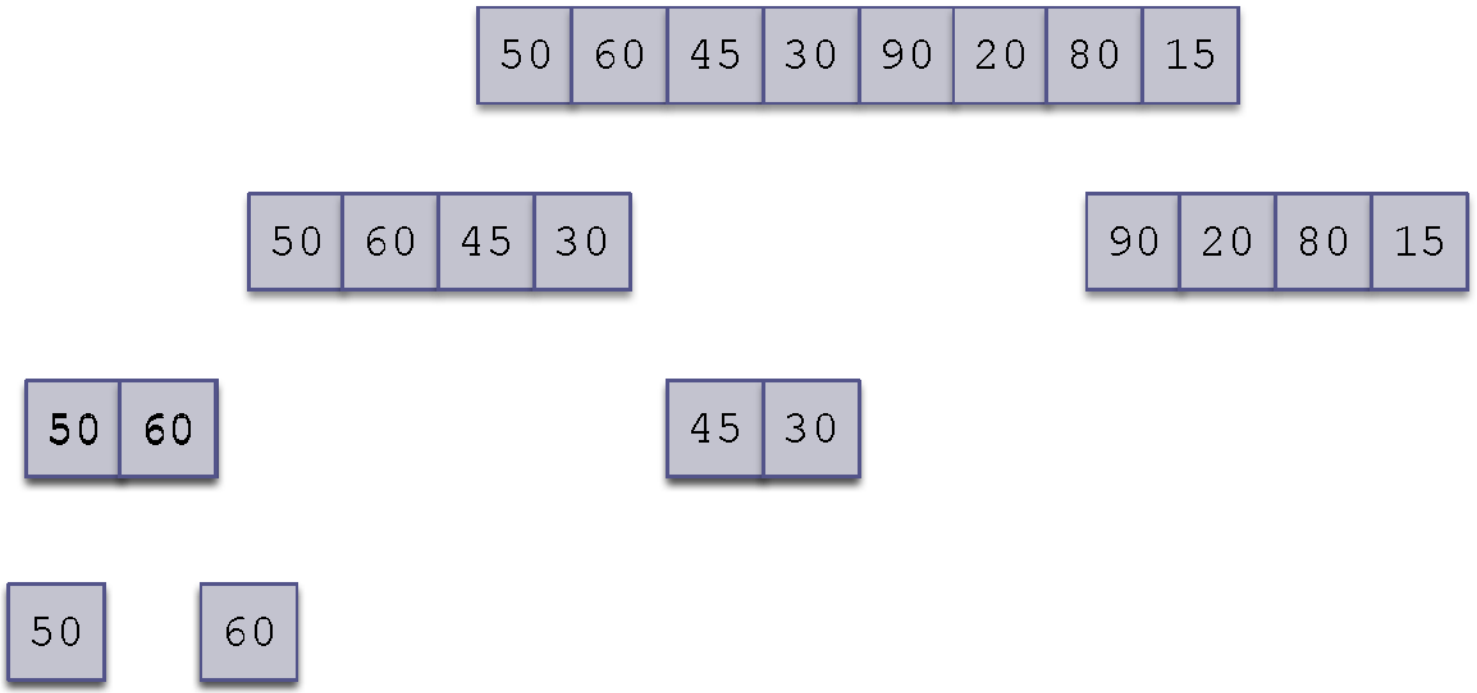
Trace of Merge Sort



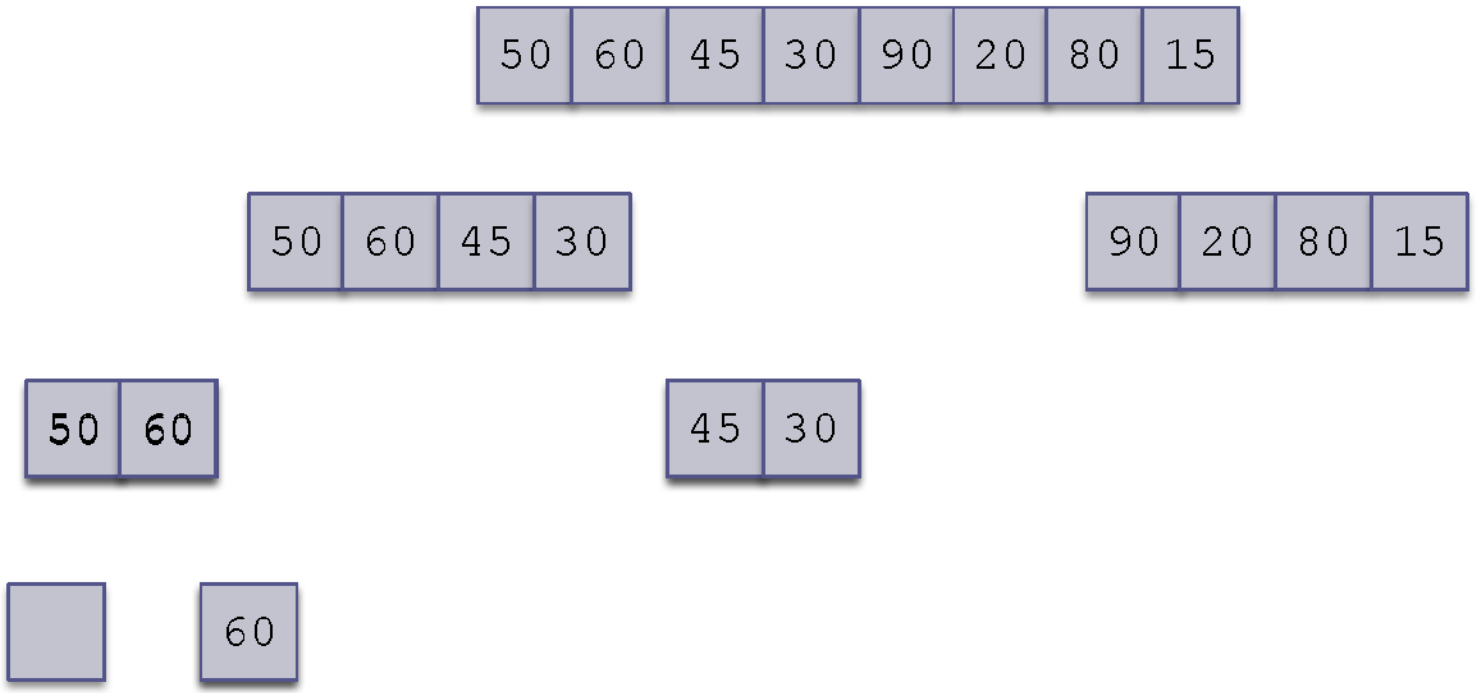
Trace of Merge Sort (cont.)



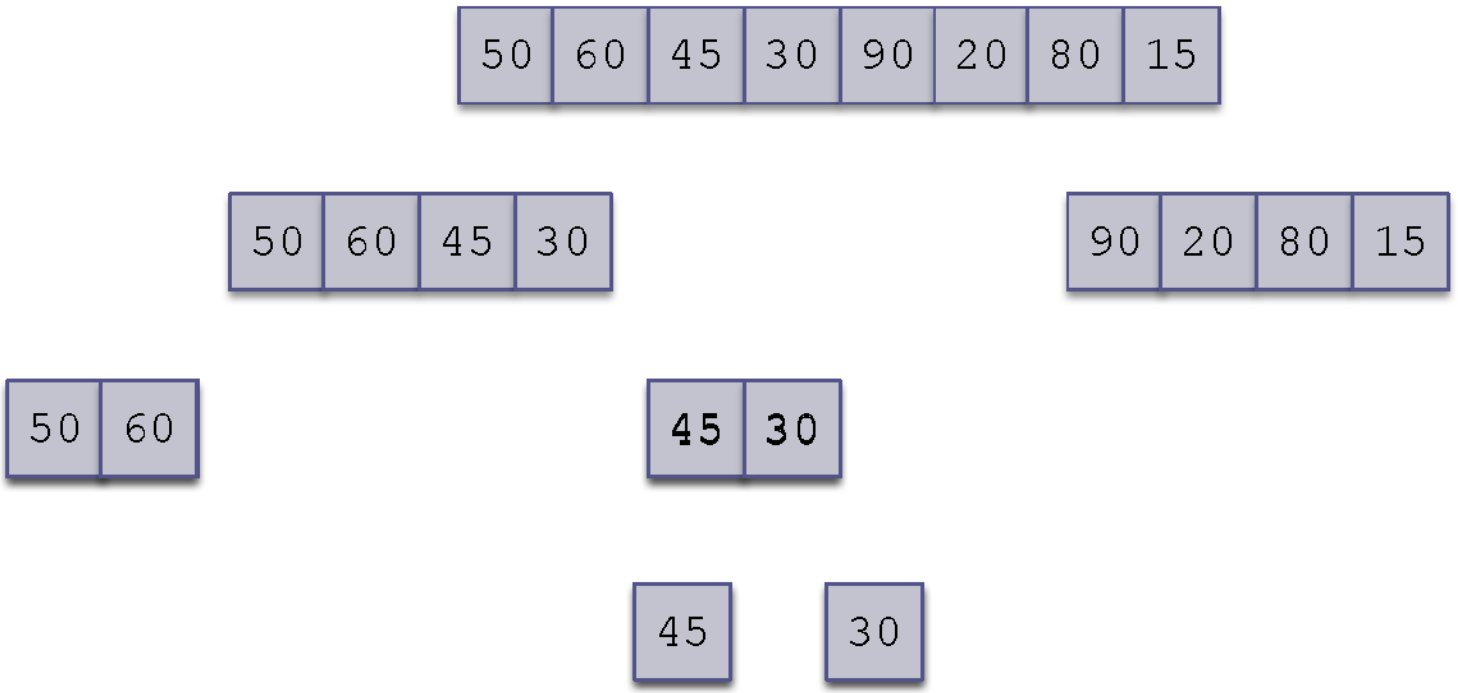
Trace of Merge Sort (cont.)



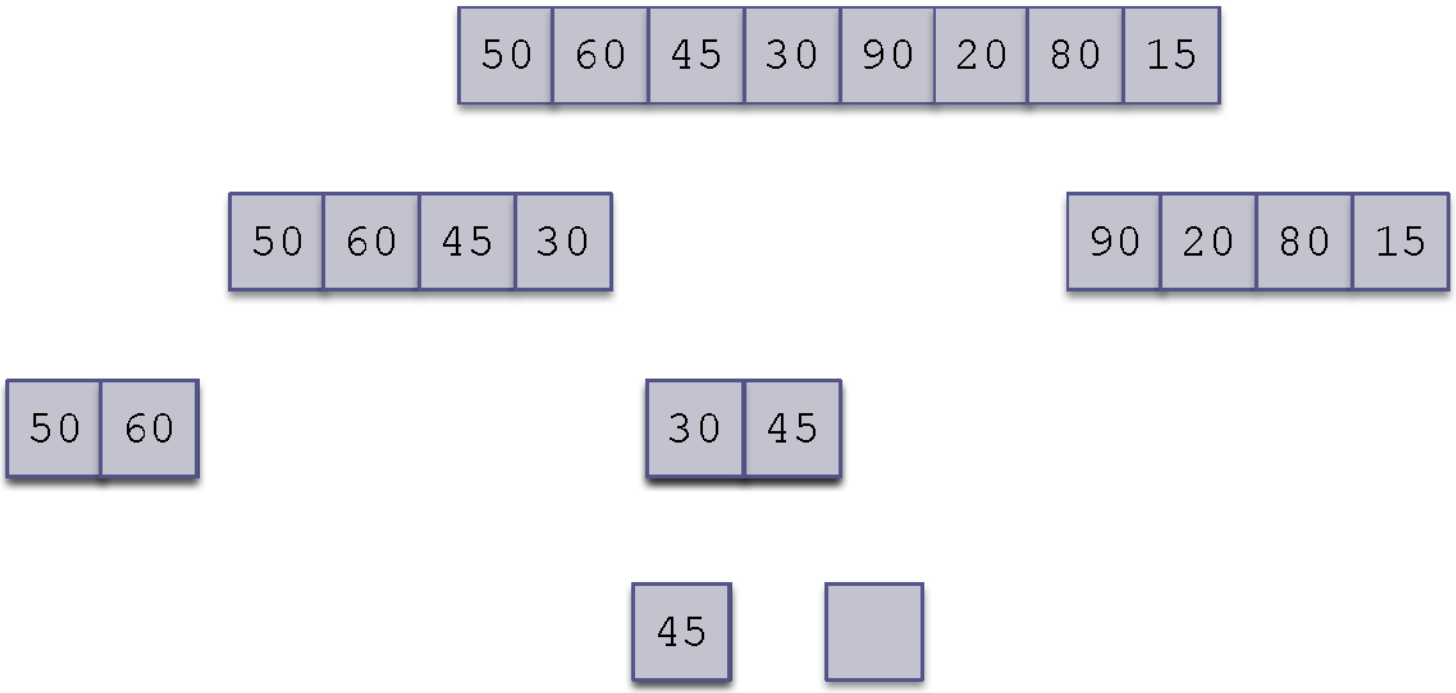
Trace of Merge Sort (cont.)



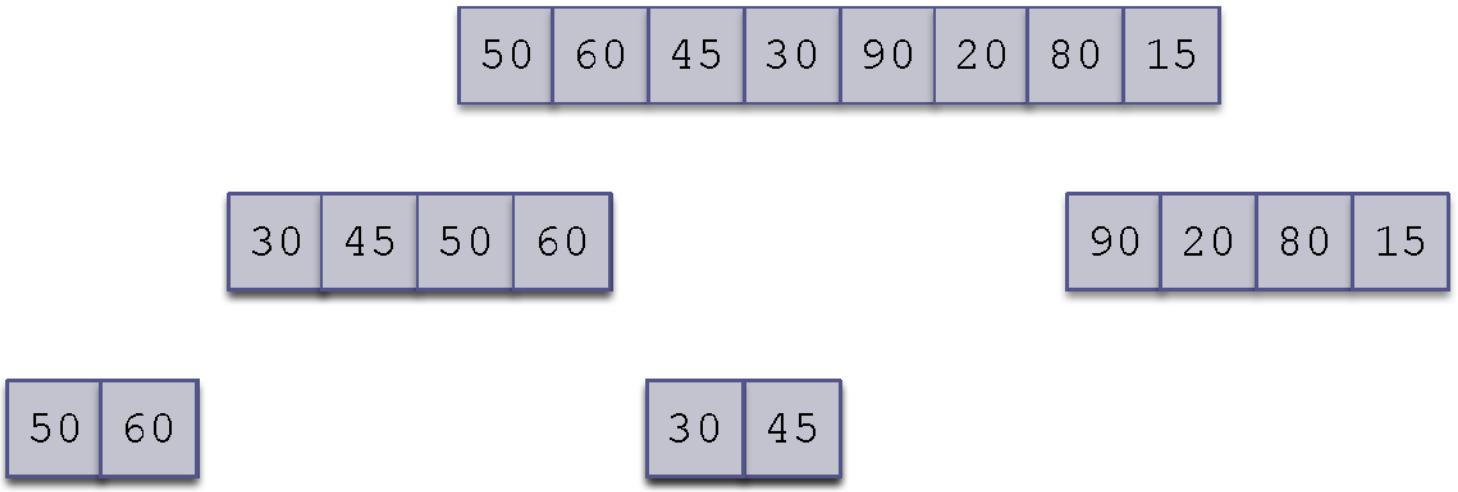
Trace of Merge Sort (cont.)



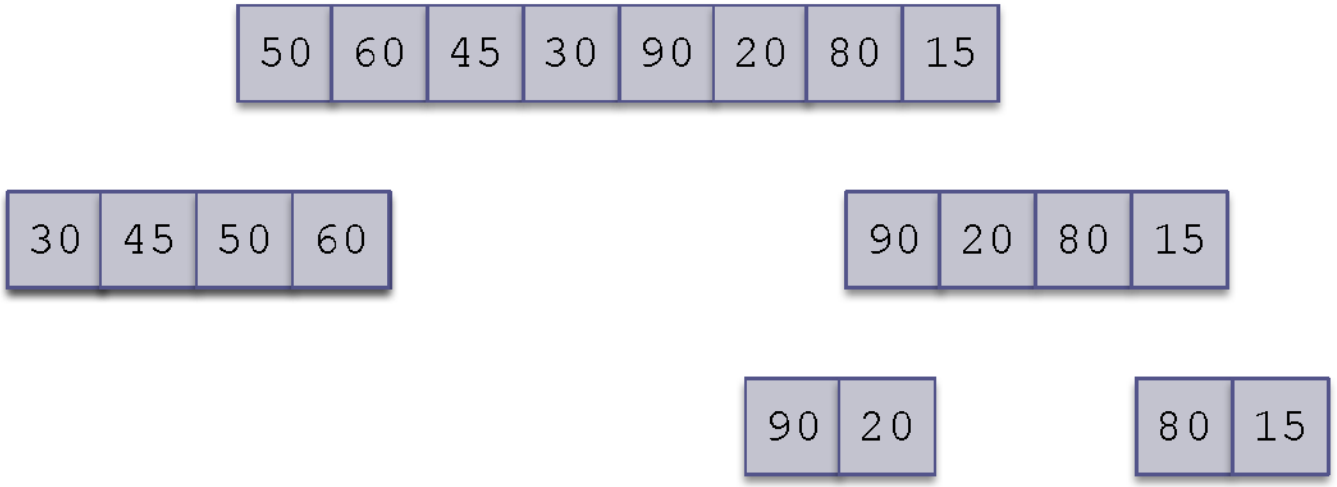
Trace of Merge Sort (cont.)



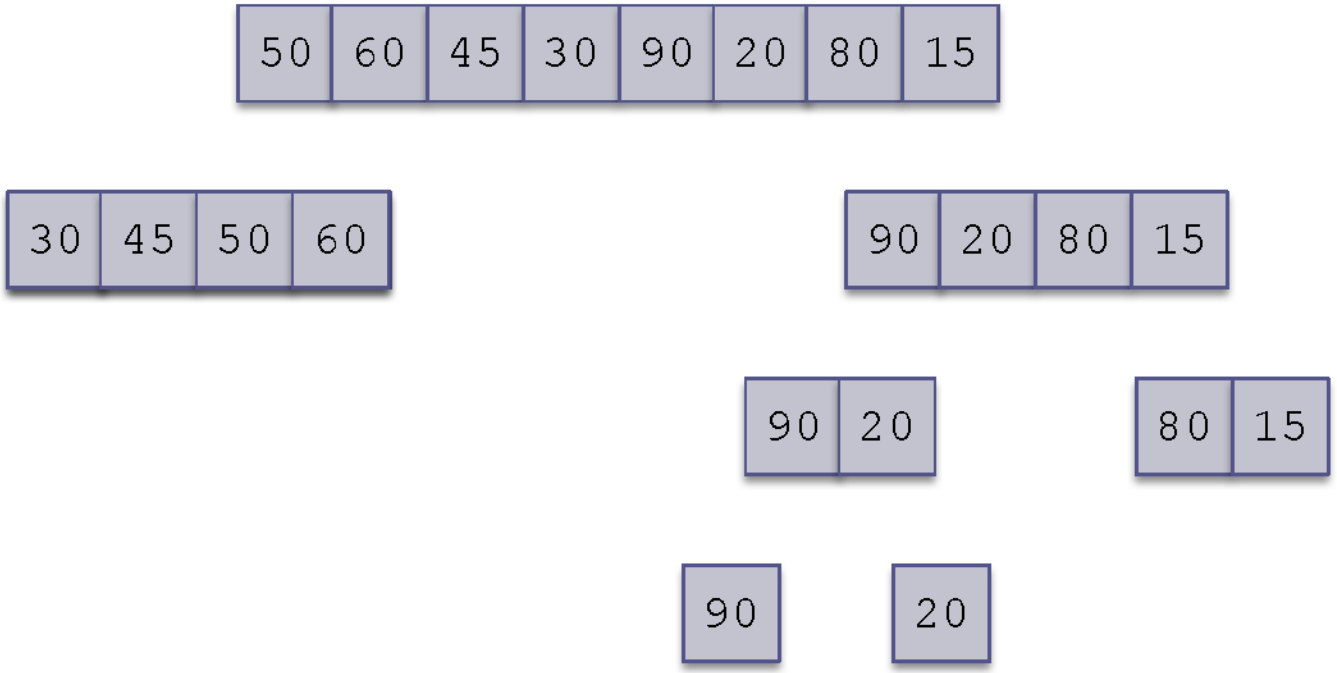
Trace of Merge Sort (cont.)



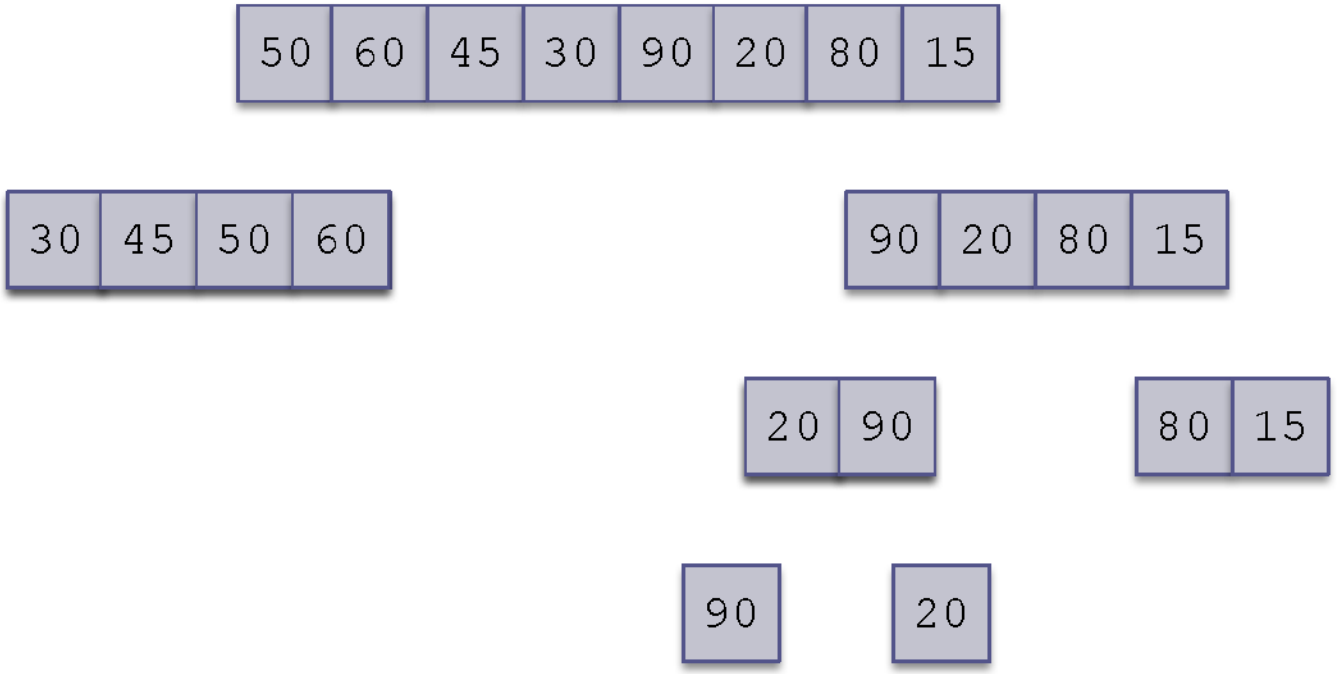
Trace of Merge Sort (cont.)



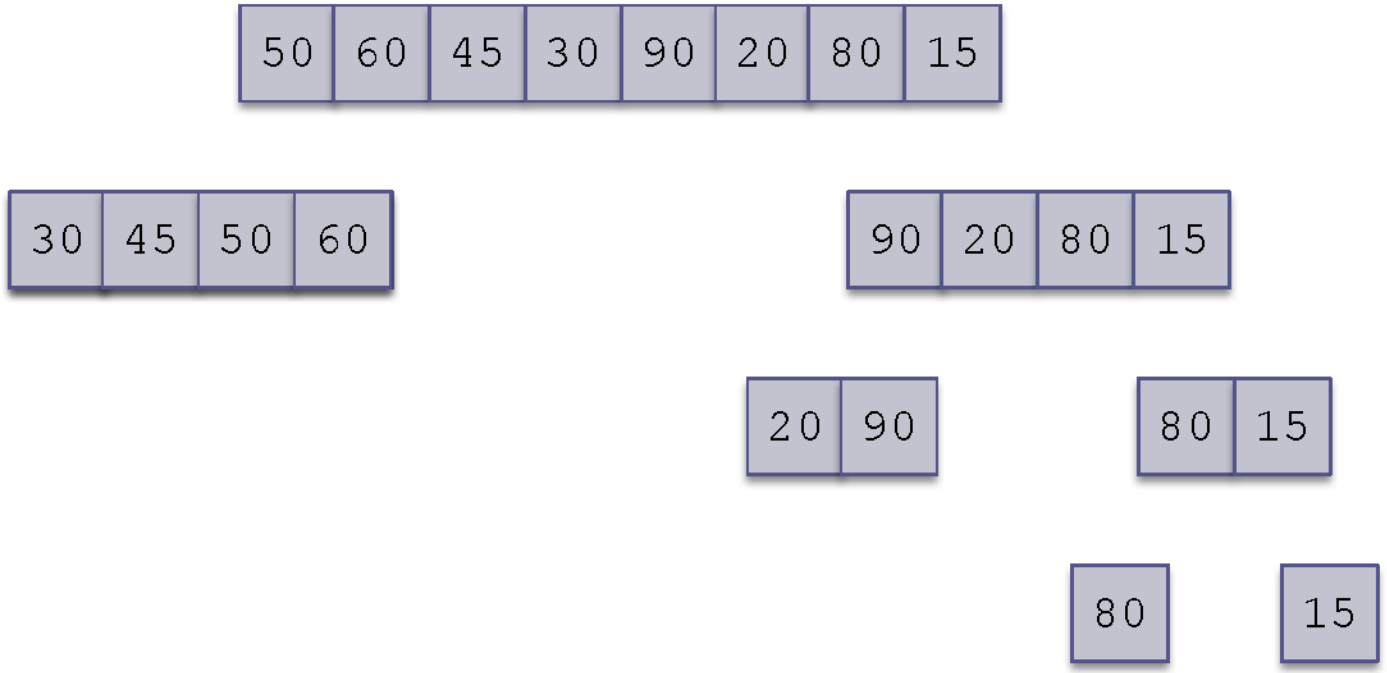
Trace of Merge Sort (cont.)



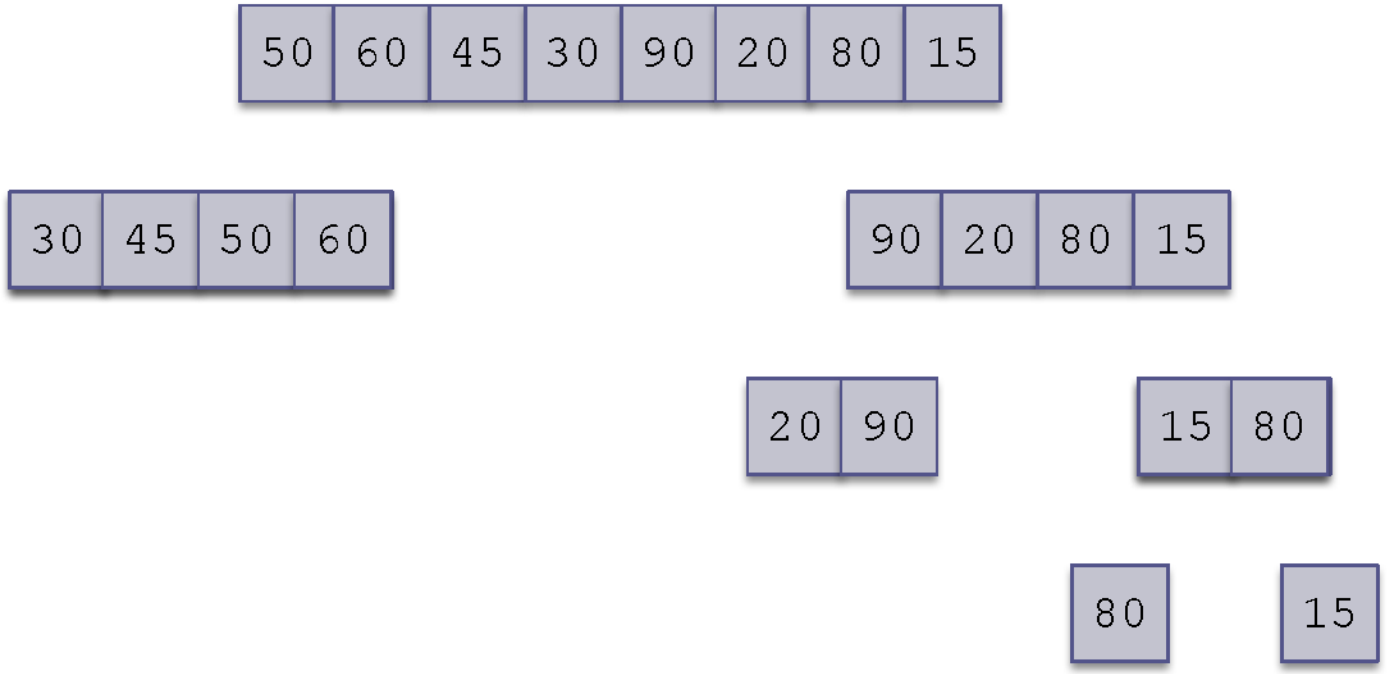
Trace of Merge Sort (cont.)



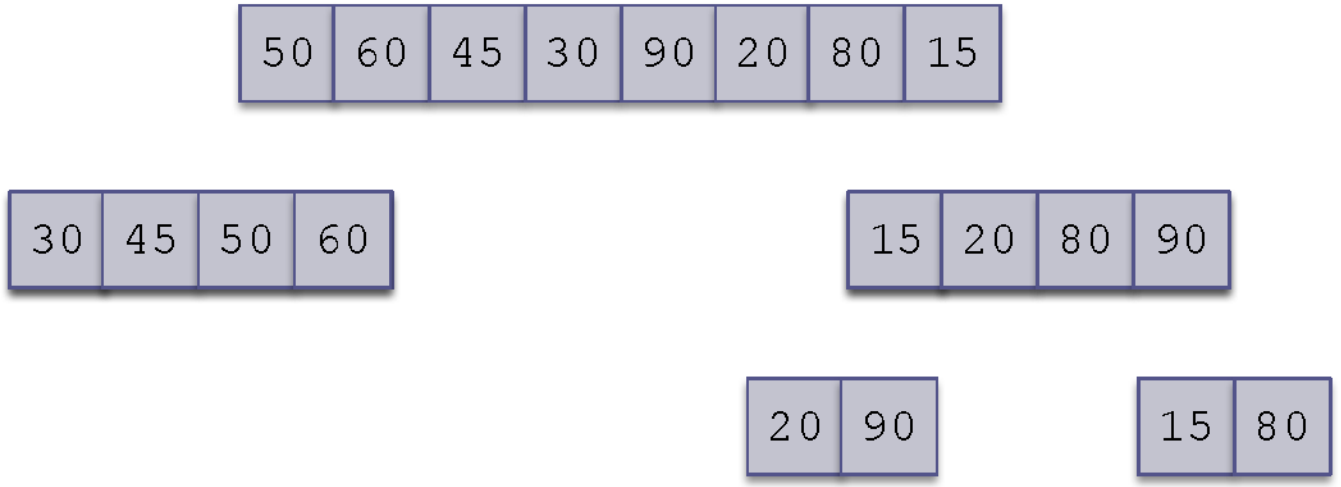
Trace of Merge Sort (cont.)



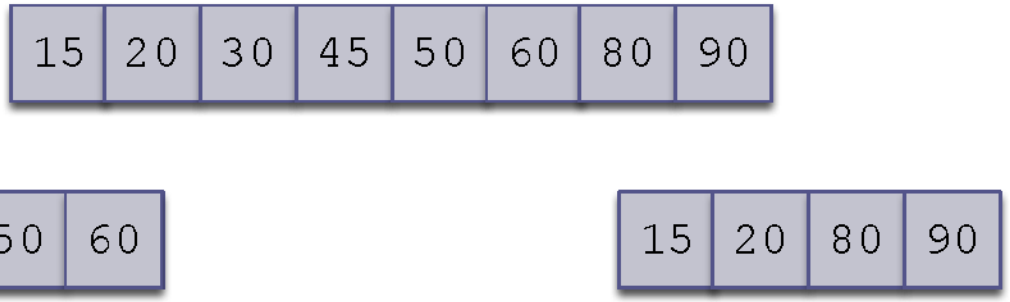
Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Distribution based sorting

- **Stable sorting** – if for two elements a_i , a_j in input sequence, i < j and $a_i = a_j$, in output sequence a_i should appear before a_j
- Allows sorting on multiple keys of elements easily.

- **Bucket-Sort (Bin-sort)** : Keys in range [0,N-1] (e.g. state abbreviations)

N buckets, each key id added to list of corresponding bucket, bucket lists are concatenated. Can be made stable.

Time complexity is O(n+N) where n is number of items to be sorted.

Radix Sort

- **Lexicographic** ordering for a multi-component key. For two keys (k_1, l_1) and (k_2, l_2), we say $(k_1, l_1) < (k_2, l_2)$ if either $k_1 < k_2$ V $(k_1 = k_2$ ^ $l_1 < l_2)$. We can extend this ordering to keys with more than 2 components.
- Radix sorting can be used to sort items lexicographically in most efficient manner if all components are from range [0,N-1] for some N.
- For an item $(k_{i1}, k_{i2}, .. k_{im})$, we refer to k_{im} least significant and k_{i1} most significant
- **Example** : Dictionary of words, m-digit numbers
- **Idea**:

(a) Sort by least significant component first using a stable bucket-sort strategy

(b) Sort the result list by next significant component using same strategy.

Stable sorting maintains least significant ordering of items which are in the same bucket in this step.

Radix sort algorithm

RadixSort(A):

Input : Sequence of m-component items $k_1, k_2, \ldots k_n$ where k_i is a m-tuple $(k_{i1}, k_{i2}, .. k_{im})$, where $k_{ij} \in [0, N-1]$

Output : Lexicographically sorted sequence of the items

l → copy of input sequence $k_1, k_2, \ldots k_n$ // output list

for t ← 0 to $N-1$

Q[t] ← {} // bucket lists

for j ← m down to 1 // each component

for i ← 1 to n

remove k_i from list l and add it to the end of the list Q[k_{ij}]

for t ← 0 to $N-1$

remove items from Q[t] add them to end of list "l"

Return l

```
static void radix(Integer[] A, Integer[] B,
                  int k, int r, int[] count) {
  // Count[i] stores number of records in bin[i]
  int i, j, rtok;

  for (i=0, rtok=1; i<k; i++, rtok*=r) { // For k digits
    for (j=0; j<r; j++) count[j] = 0;    // Initialize count

    // Count the number of records for each bin on this pass
    for (j=0; j<A.length; j++) count[(A[j]/rtok)%r]++;

    // count[j] will be index in B for last slot of bin j.
    for (j=1; j<r; j++) count[j] = count[j-1] + count[j];

    // Put records into bins, working from bottom of bin
    // Since bins fill from bottom, j counts downwards
    for (j=A.length-1; j>=0; j--)
      B[--count[(A[j]/rtok)%r]] = A[j];

    for (j=0; j<A.length; j++) A[j] = B[j]; // Copy B back
  }
}
```

Source : Dr. Calvin & textbook

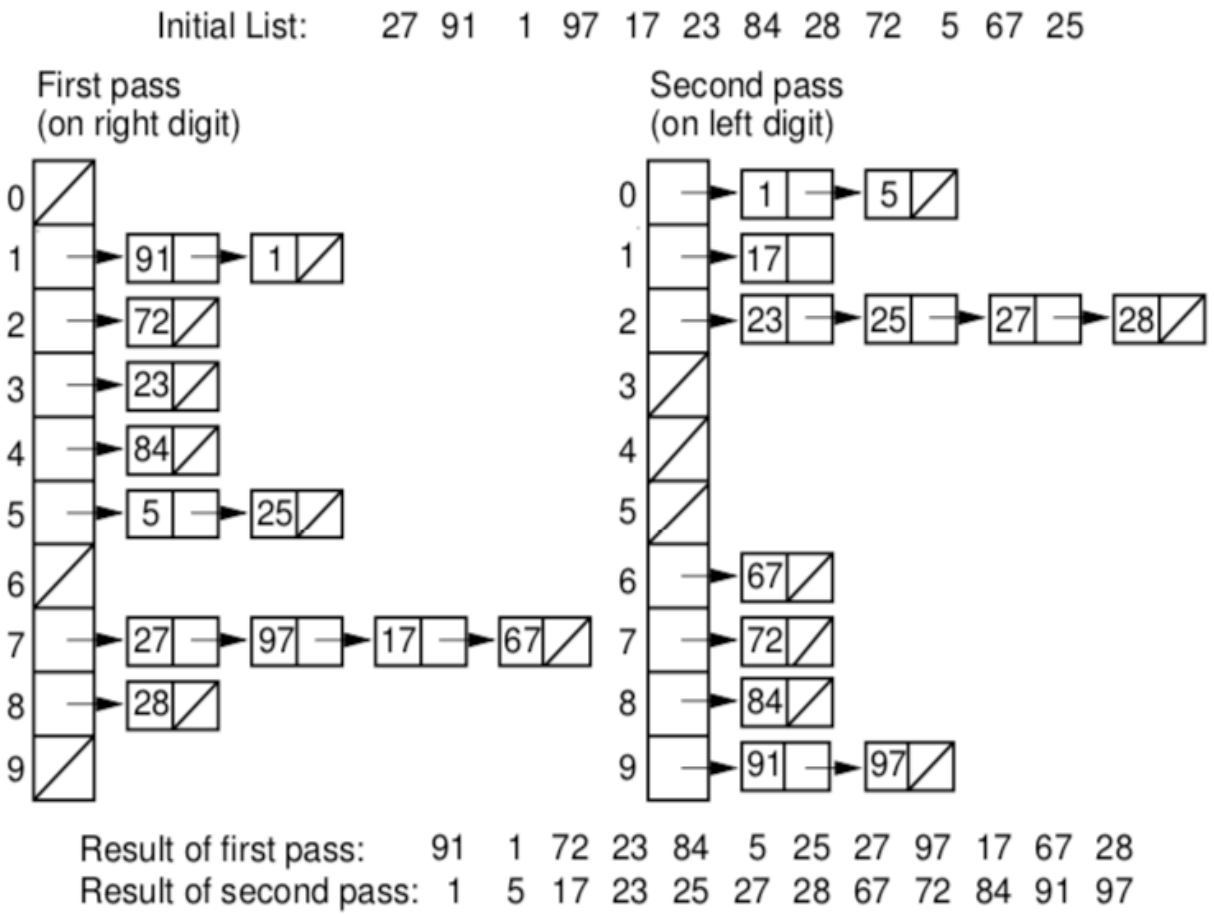


Figure 7.17 An example of Radix Sort for twelve two-digit numbers in base ten. Two passes are required to sort the list.

Source : Dr. Calvin & textbook

Radix sort time complexity

Time complexity : O(m(n+N)) where m is the number of digits, N is the number of possible values for a digit and n is the number of items to be sorted. For fixed value of m and N, complexity is O(n) best we can do.

Source : Dr. Calvin & textbook