CS 570: Data Structures

Sorting

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

CHAPTER 8
SORTING

Chapter Objectives

- To learn how to implement the following sorting algorithms:
  - selection sort
  - insertion sort
  - merge sort
  - quicksort
- To understand the differences in performance of these algorithms, and which to use for small, medium arrays, and large arrays

Week 14

- Reading Assignment: Koffman and Wolfgang, Sections 8.2, 8.3, 8.6, 8.9

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 $O(n \log n)$

Selection Sort

Section 8.2

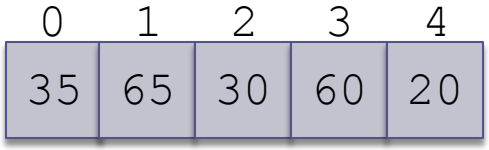
Selection Sort

- Selection sort is relatively easy to understand
- 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
  - While the sort algorithms are not limited to arrays, throughout this chapter we will sort arrays for simplicity
- All items to be sorted must be `Comparable` objects, so, for example, any `int` values must be wrapped in `Integer` objects

Trace of Selection Sort

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

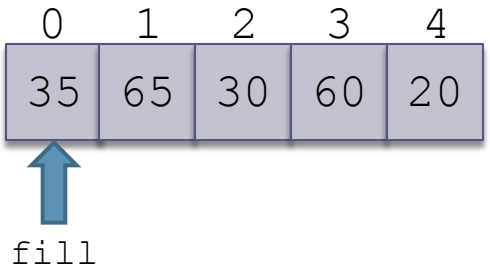


`n`	5
`fill`	
`posMin`	

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

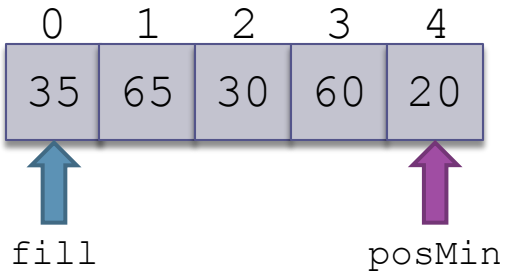


`n`	5
`fill`	0
`posMin`	

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

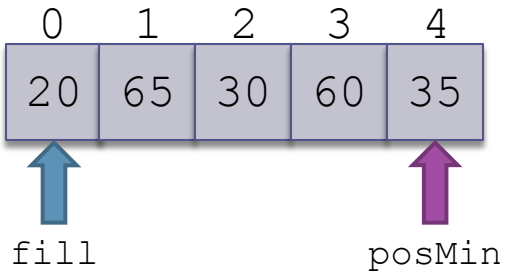


`n`	5
`fill`	0
`posMin`	4

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

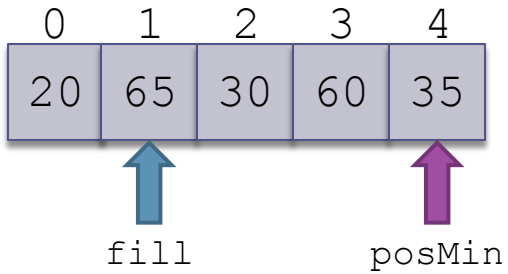


`n`	5
`fill`	0
`posMin`	4

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

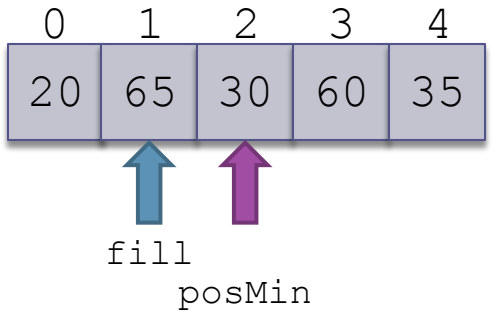


`n`	5
`fill`	1
`posMin`	4

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

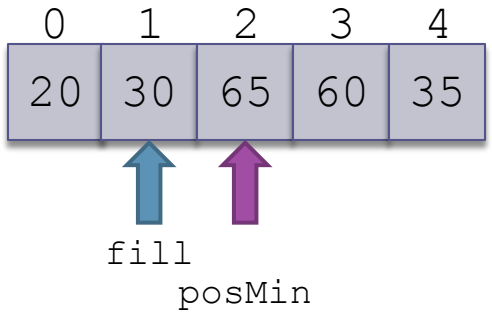


n	5
fill	1
posMin	2

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

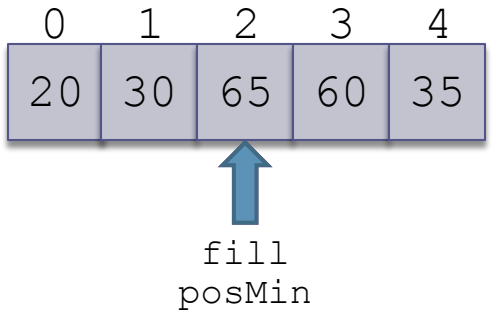


n	5
fill	1
posMin	2

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

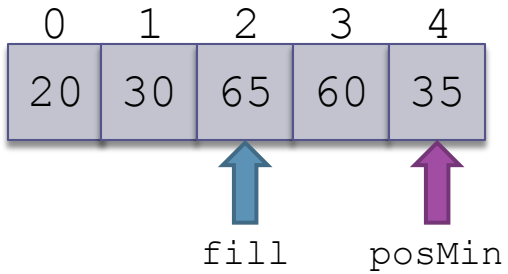


`n`	5
`fill`	2
`posMin`	2

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

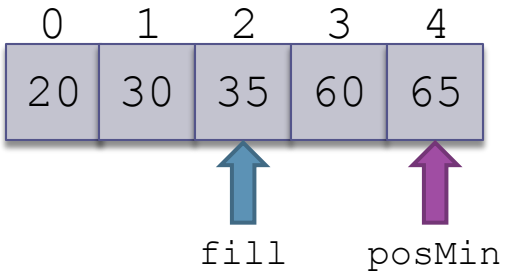


`n`	5
`fill`	2
`posMin`	4

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

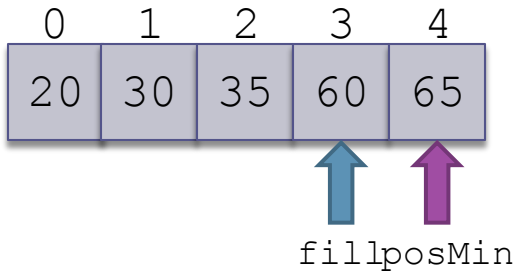


`n`	5
`fill`	2
`posMin`	4

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

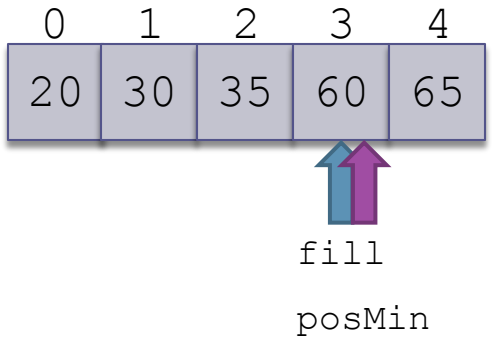


`n`	5
`fill`	3
`posMin`	4

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

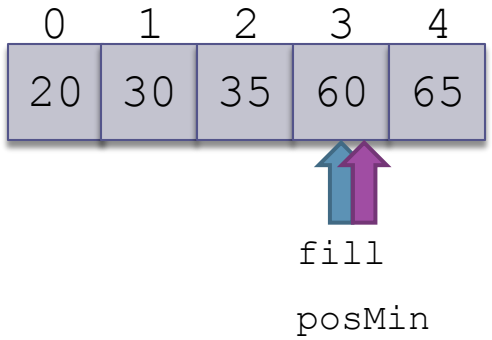


`n`	5
`fill`	3
`posMin`	3

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`

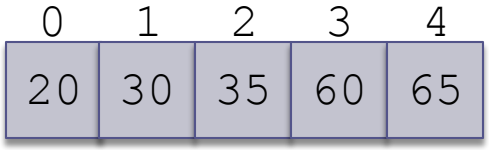


n	5
fill	3
posMin	3

Trace of Selection Sort (cont.)

`n` = number of elements in the array

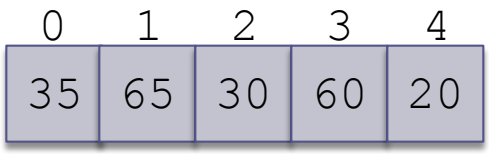
1. **for** `fill = 0` to `n - 2` do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`
3. Exchange the item at `posMin` with the one at `fill`



`n`	5
`fill`	3
`posMin`	3

Trace of Selection Sort Refinement

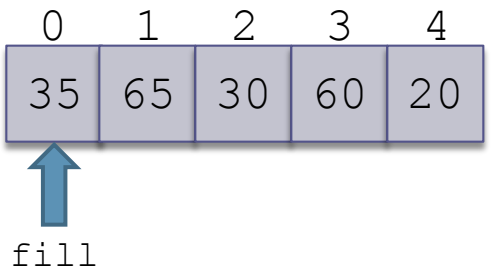
n	5
fill	
posMin	
next	



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`

Trace of Selection Sort Refinement (cont.)

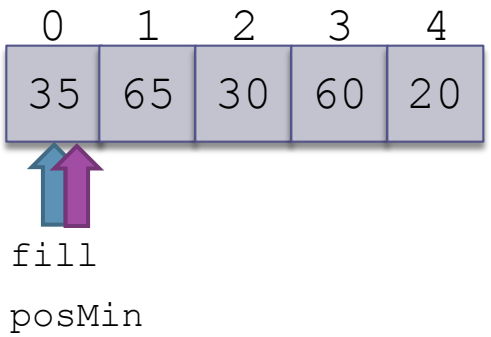
n	5
fill	0
posMin	
next	



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`

Trace of Selection Sort Refinement (cont.)

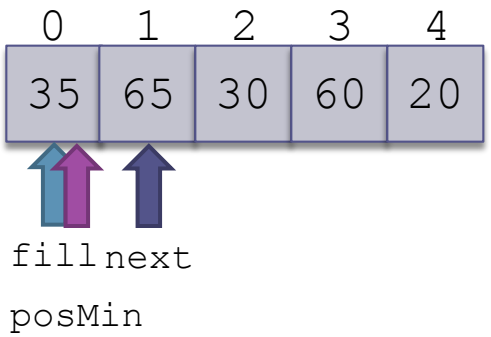
n	5
fill	0
posMin	0
next	



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`

Trace of Selection Sort Refinement (cont.)

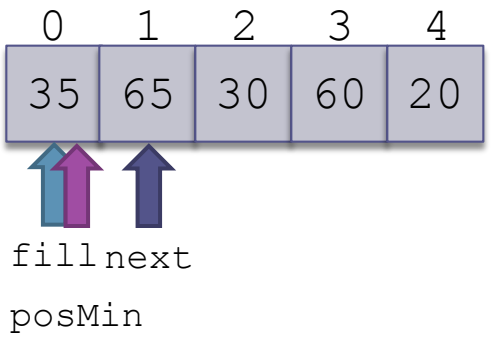
n	5
fill	0
posMin	0
next	1



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`

Trace of Selection Sort Refinement (cont.)

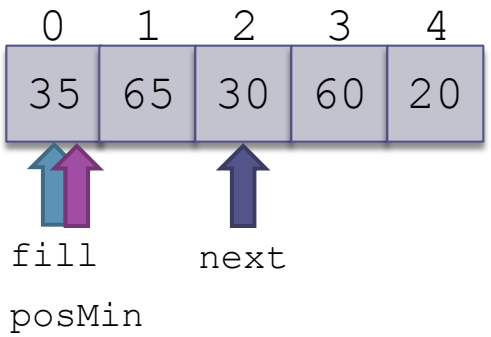
n	5
fill	0
posMin	0
next	1



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`

Trace of Selection Sort Refinement (cont.)

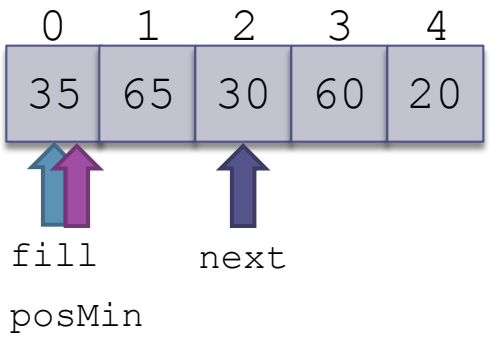
n	5
fill	0
posMin	0
next	2



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`

Trace of Selection Sort Refinement (cont.)

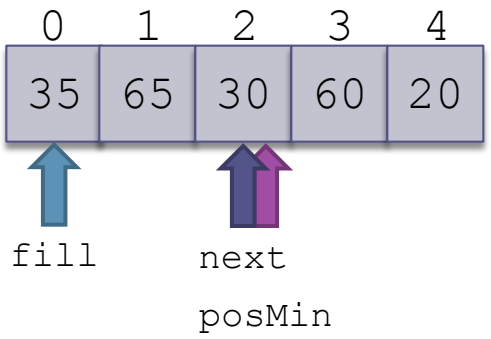
n	5
fill	0
posMin	0
next	2



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`

Trace of Selection Sort Refinement (cont.)

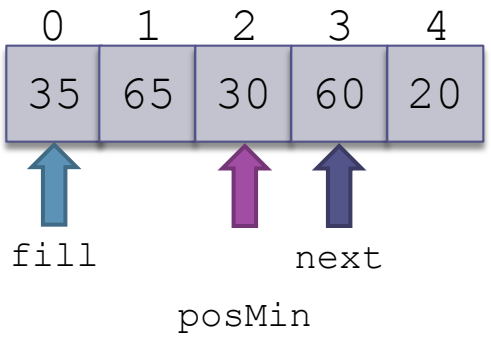
n	5
fill	0
posMin	2
next	2



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`

Trace of Selection Sort Refinement (cont.)

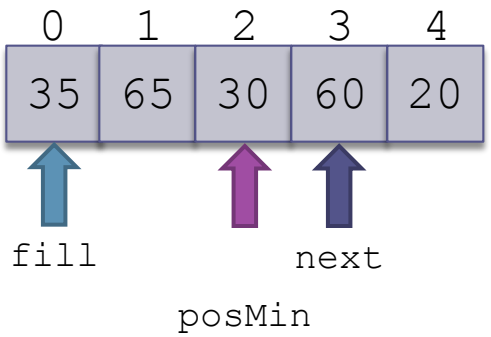
n	5
fill	0
posMin	2
next	3



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

Trace of Selection Sort Refinement (cont.)

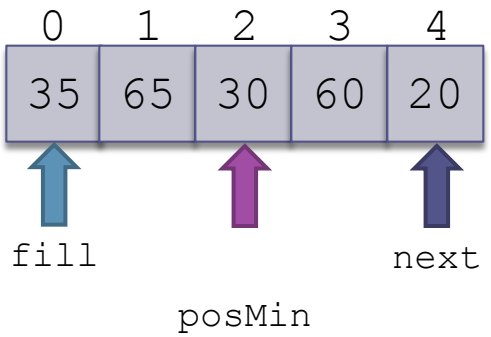
n	5
fill	0
posMin	2
next	3



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

Trace of Selection Sort Refinement (cont.)

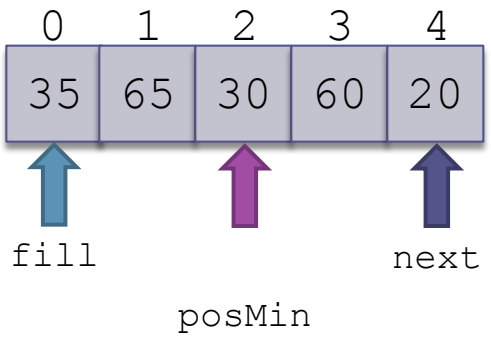
n	5
fill	0
posMin	2
next	4



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`

Trace of Selection Sort Refinement (cont.)

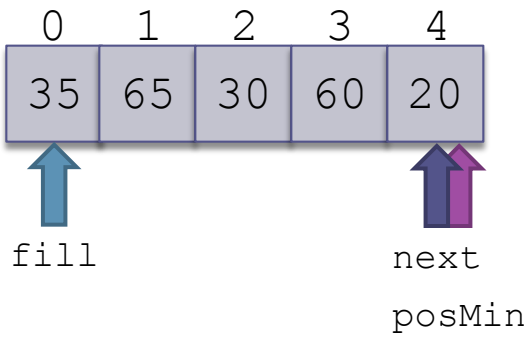
n	5
fill	0
posMin	2
next	4



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

Trace of Selection Sort Refinement (cont.)

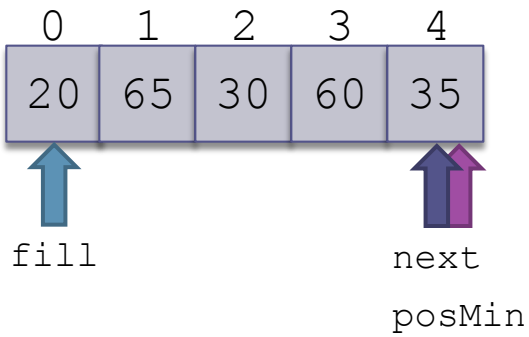
n	5
fill	0
posMin	4
next	4



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

Trace of Selection Sort Refinement (cont.)

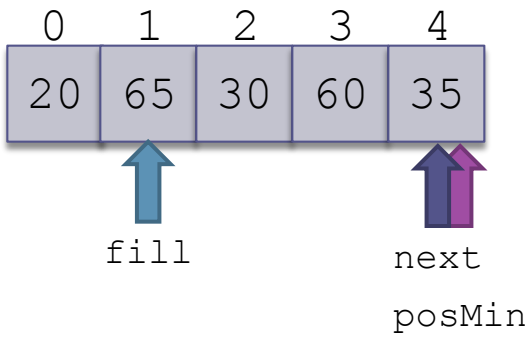
n	5
fill	0
posMin	4
next	4



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

Trace of Selection Sort Refinement (cont.)

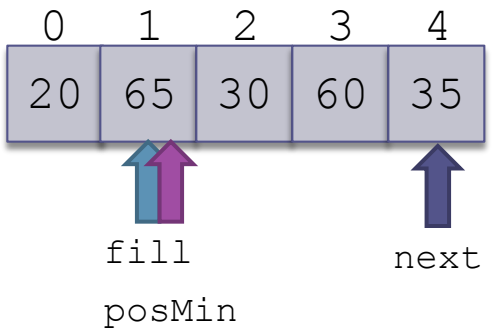
n	5
fill	1
posMin	4
next	4



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`

Trace of Selection Sort Refinement (cont.)

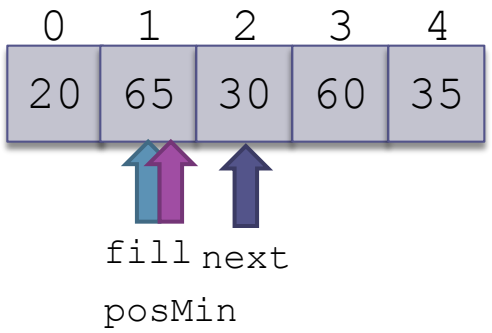
n	5
fill	1
posMin	1
next	4



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`

Trace of Selection Sort Refinement (cont.)

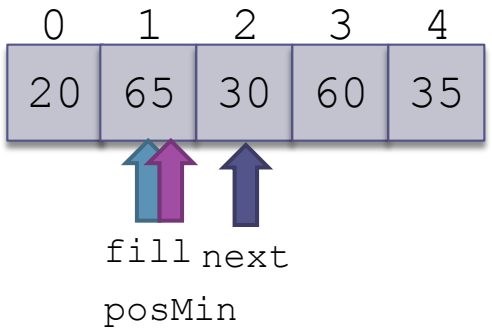
n	5
fill	1
posMin	1
next	2



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`

Trace of Selection Sort Refinement (cont.)

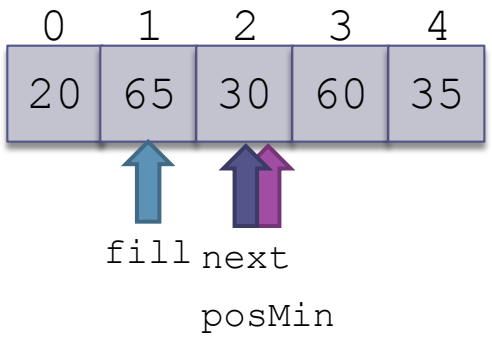
n	5
fill	1
posMin	1
next	2



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`

Trace of Selection Sort Refinement (cont.)

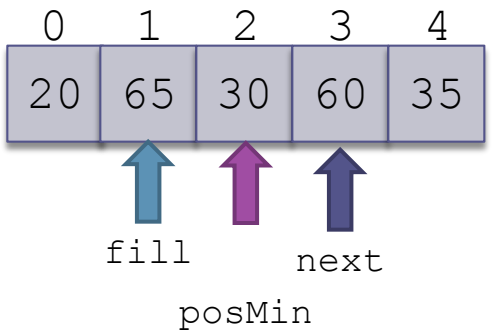
n	5
fill	1
posMin	2
next	2



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`

Trace of Selection Sort Refinement (cont.)

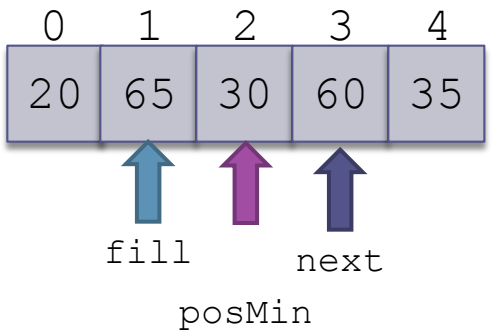
n	5
fill	1
posMin	2
next	3



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

Trace of Selection Sort Refinement (cont.)

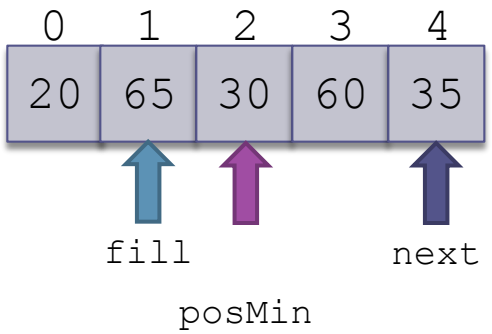
n	5
fill	1
posMin	2
next	3



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

Trace of Selection Sort Refinement (cont.)

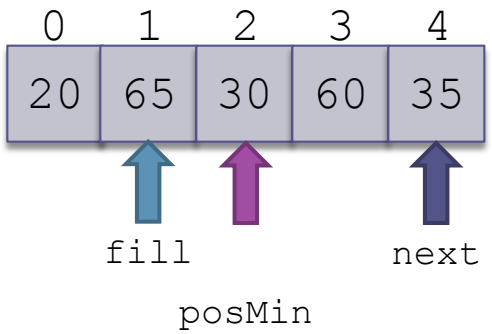
n	5
fill	1
posMin	2
next	4



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`

Trace of Selection Sort Refinement (cont.)

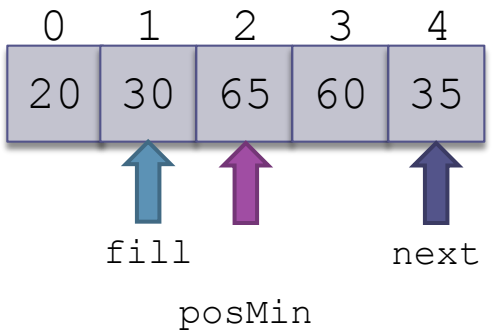
n	5
fill	1
posMin	2
next	4



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`

Trace of Selection Sort Refinement (cont.)

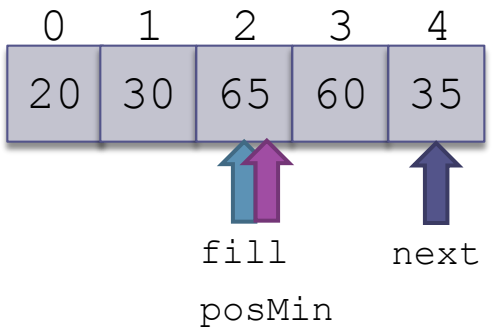
n	5
fill	1
posMin	2
next	4



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`

Trace of Selection Sort Refinement (cont.)

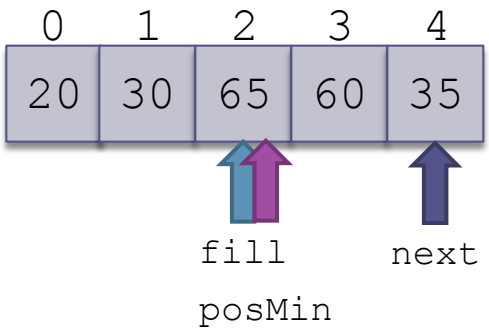
n	5
fill	2
posMin	2
next	4



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

Trace of Selection Sort Refinement (cont.)

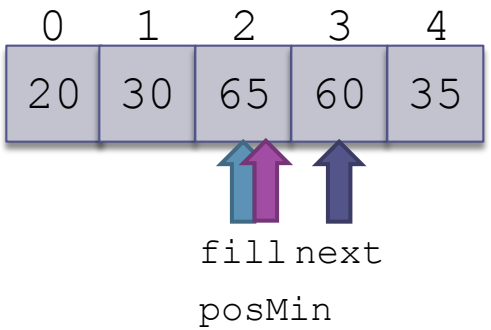
n	5
fill	2
posMin	2
next	4



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`

Trace of Selection Sort Refinement (cont.)

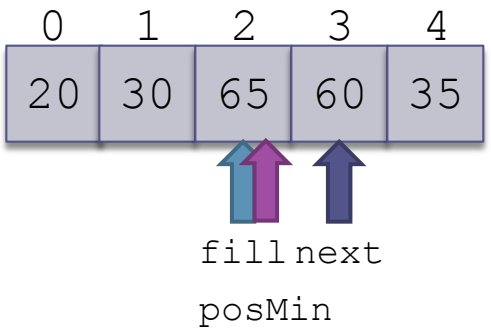
n	5
fill	2
posMin	2
next	3



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`

Trace of Selection Sort Refinement (cont.)

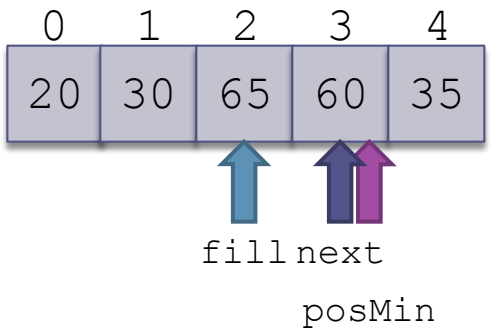
n	5
fill	2
posMin	2
next	3



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

Trace of Selection Sort Refinement (cont.)

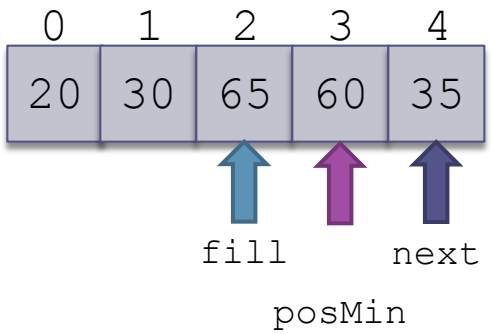
n	5
fill	2
posMin	3
next	3



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

Trace of Selection Sort Refinement (cont.)

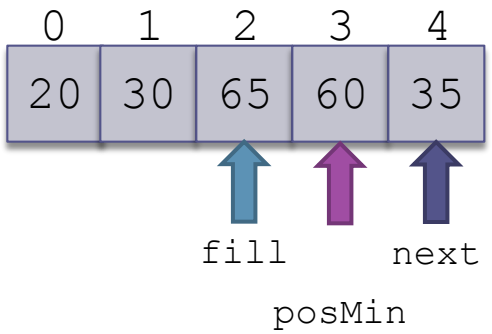
n	5
fill	2
posMin	3
next	4



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

Trace of Selection Sort Refinement (cont.)

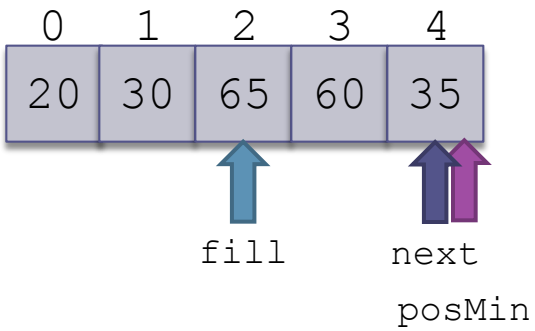
n	5
fill	2
posMin	3
next	4



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`

Trace of Selection Sort Refinement (cont.)

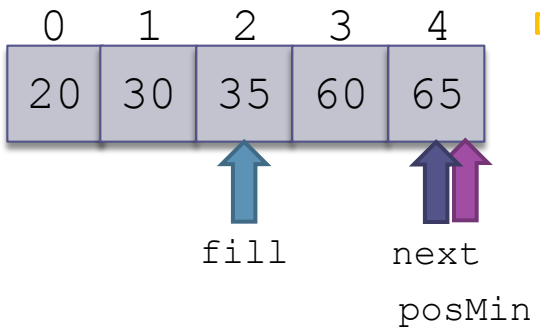
n	5
fill	2
posMin	4
next	4



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

Trace of Selection Sort Refinement (cont.)

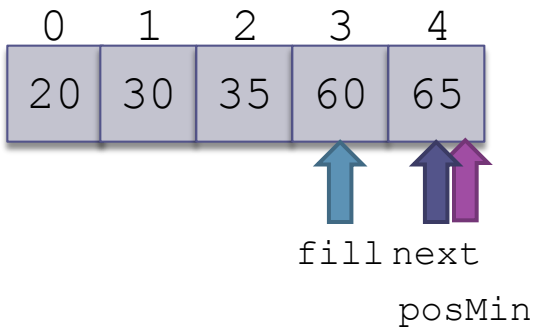
n	5
fill	2
posMin	4
next	4



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`

Trace of Selection Sort Refinement (cont.)

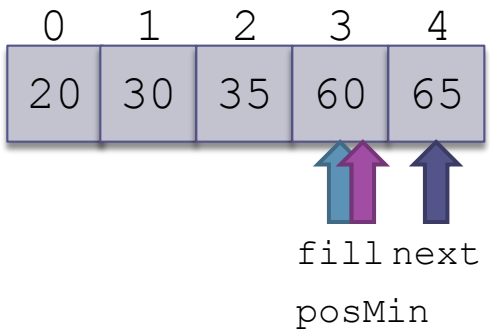
n	5
fill	3
posMin	4
next	4



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

Trace of Selection Sort Refinement (cont.)

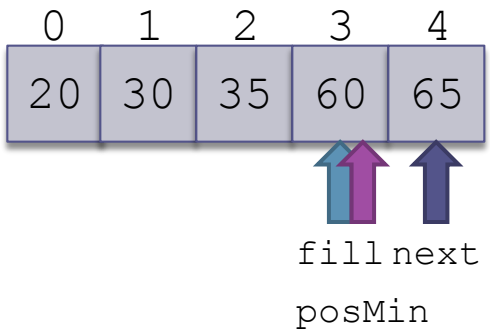
n	5
fill	3
posMin	3
next	4



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

Trace of Selection Sort Refinement (cont.)

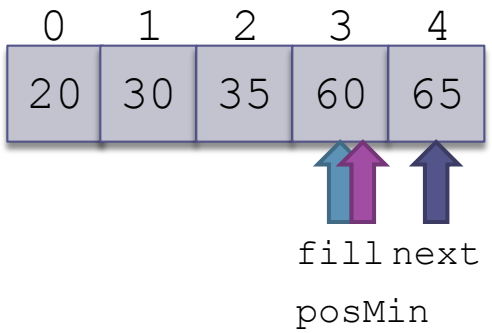
n	5
fill	3
posMin	3
next	4



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`

Trace of Selection Sort Refinement (cont.)

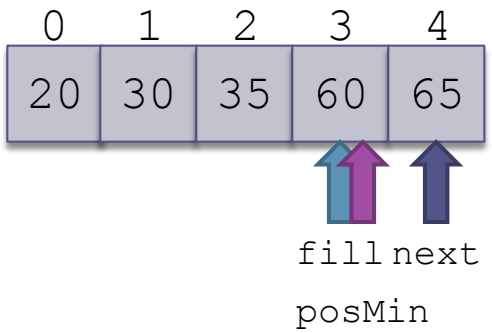
n	5
fill	3
posMin	3
next	4



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`

Trace of Selection Sort Refinement (cont.)

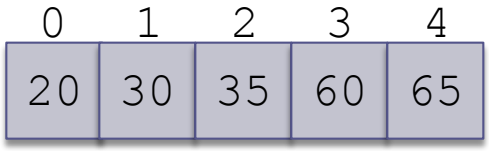
n	5
fill	3
posMin	3
next	4



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`

Trace of Selection Sort Refinement (cont.)

n	5
fill	3
posMin	3
next	4



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

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`

Code for Selection Sort


```
public class SelectionSort {
  /** Sort the array using selection sort algorithm.
      pre:  table contains Comparable objects.
      post: table is sorted.
      @param table The array to be sorted
   */
  public static void sort(Comparable[] table) {
    int n = table.length;
    for (int fill = 0; fill < n - 1; fill++) {
      // Invariant: table[0 . . . fill - 1] is sorted.
      int posMin = fill;
```

Code for Selection Sort


```
for (int next = fill + 1; next < n; next++) {
    // Invariant: table[posMin] is the smallest
    // item in
    // table[fill . . . next - 1].
  if (table[next].compareTo(table[posMin]) < 0)
  {
        posMin = next;
   }
  }
  // assert: table[posMin] is the smallest item
  // in table[fill . . . n - 1].
  // Exchange table[fill] and table[posMin].
```

Code for Selection Sort


```
        Comparable temp = table[fill];
        table[fill] = table[posMin];
        table[posMin] = temp;
        // assert: table[fill] is the smallest item in
        // table[fill . . . n - 1].
      }
      // assert: table[0 . . . n - 1] is sorted.
    }
}
```

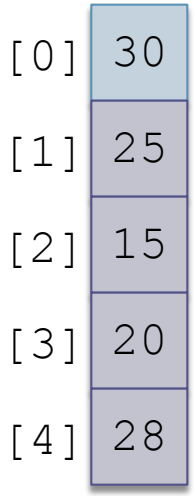
Insertion Sort

Section 8.3

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

Trace of Insertion Sort

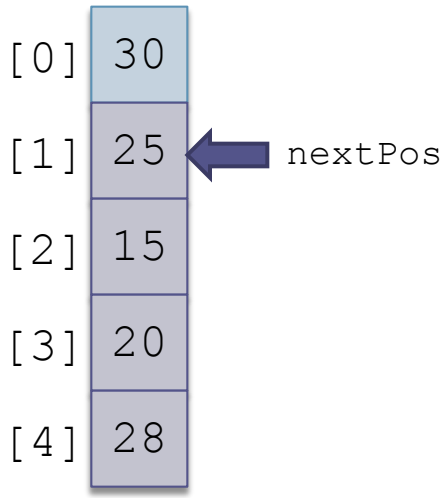


1. **`for`** each array element from the second (**`nextPos`** = 1) to the last
2. Insert the element at **`nextPos`** where it belongs in the array, increasing the length of the sorted subarray by 1 element

To adapt the insertion algorithm to an array that is filled with data, we start with a sorted subarray consisting of only the first element

Trace of Insertion Sort (cont.)

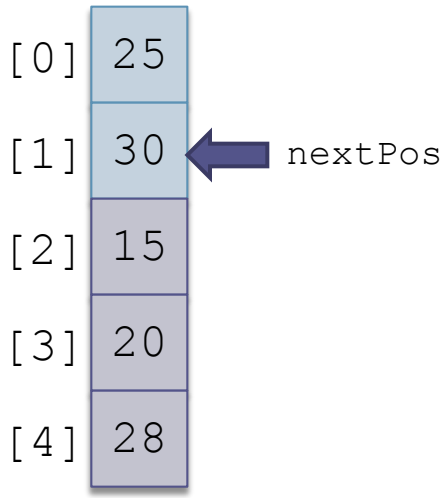
nextPos	1



1. **`for`** each array element from the second (**`nextPos`** = 1) to the last
2. Insert the element at **`nextPos`** where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

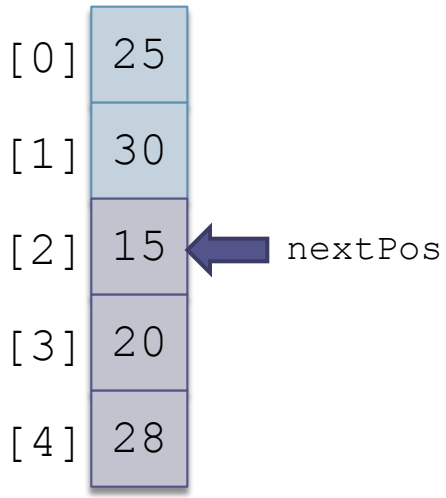
nextPos	1



1. **`for`** each array element from the second (**`nextPos`** = 1) to the last
2. Insert the element at **`nextPos`** where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

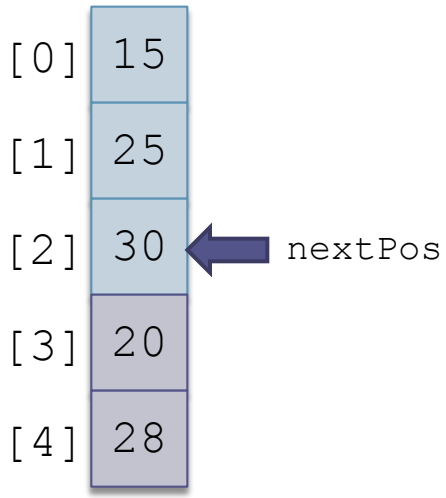
nextPos	2



1. **`for`** each array element from the second (**`nextPos`** = 1) to the last
2. Insert the element at **`nextPos`** where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

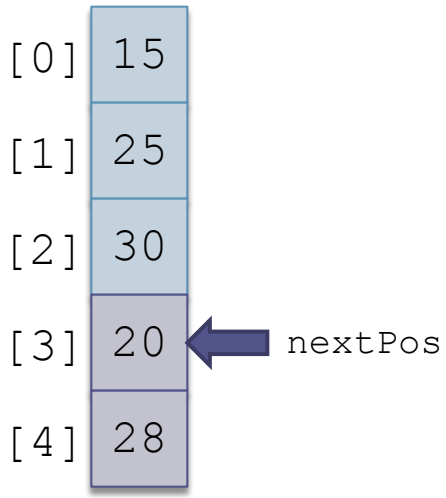
nextPos	2



1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

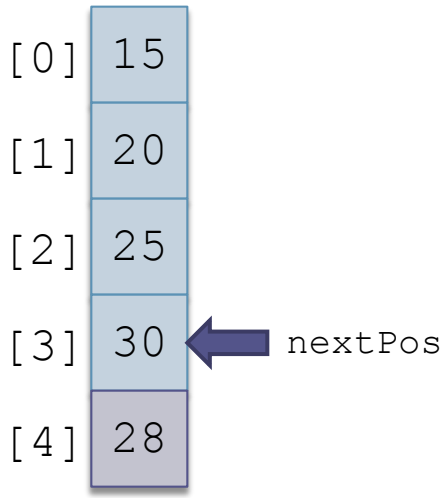
nextPos	3



1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

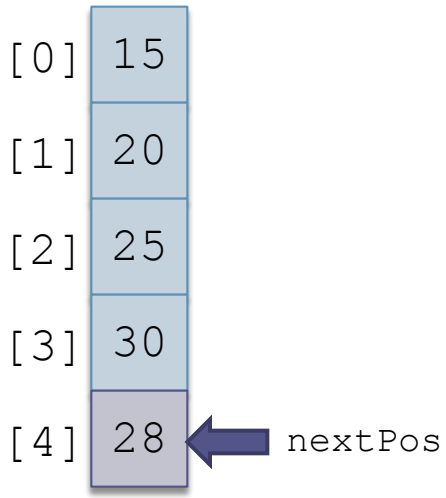
nextPos	3



1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

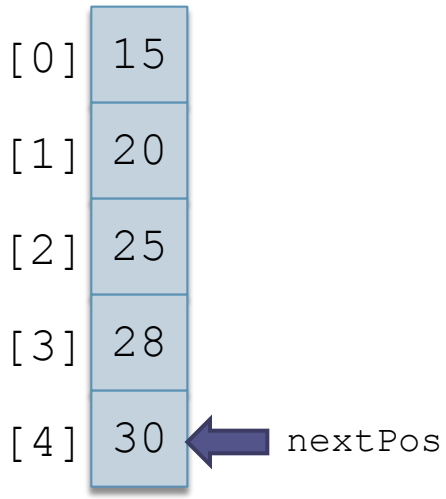
nextPos	4



1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

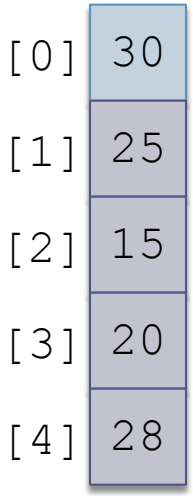
Trace of Insertion Sort (cont.)

nextPos	4



1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

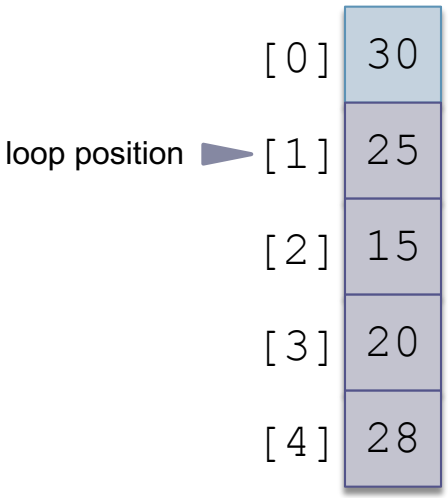
Trace of Insertion Sort Refinement



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`

Trace of Insertion Sort Refinement (cont.)

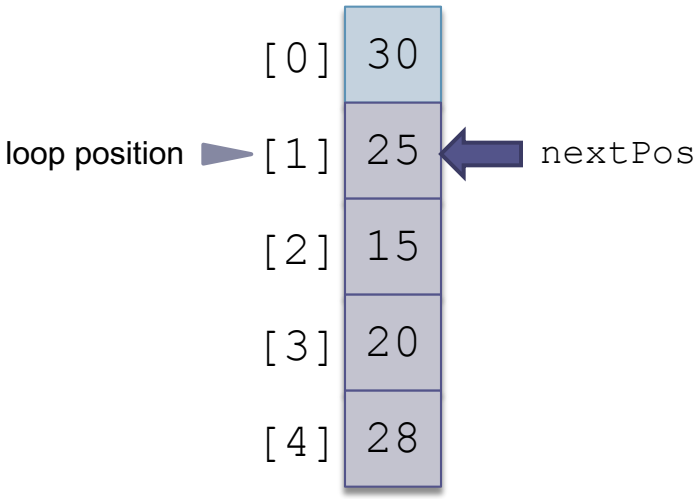
nextPos	1
nextVal	



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`

Trace of Insertion Sort Refinement (cont.)

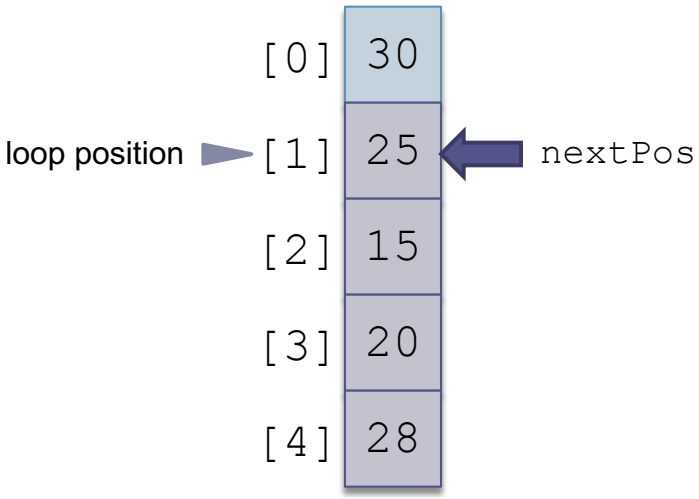
nextPos	1
nextVal	



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`

Trace of Insertion Sort Refinement (cont.)

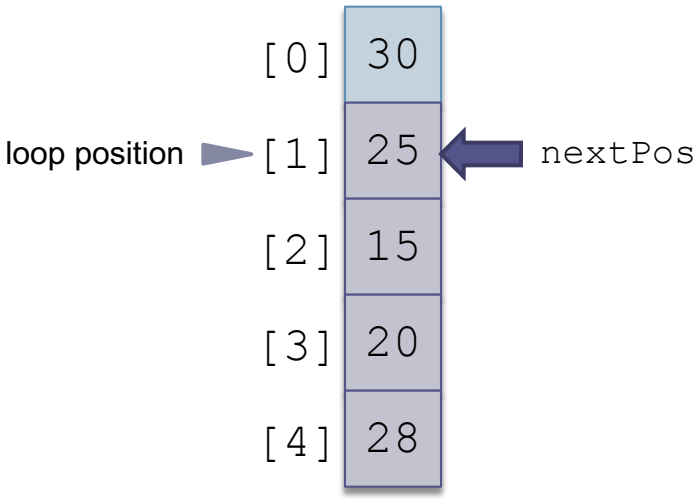
nextPos	1
nextVal	25



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`

Trace of Insertion Sort Refinement (cont.)

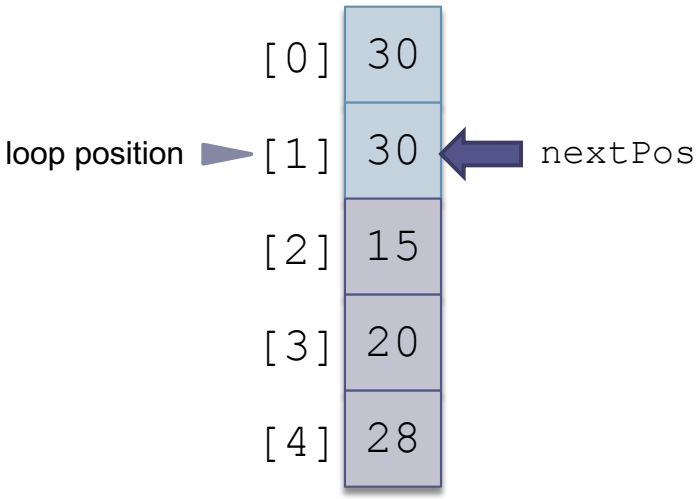
nextPos	1
nextVal	25



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`

Trace of Insertion Sort Refinement (cont.)

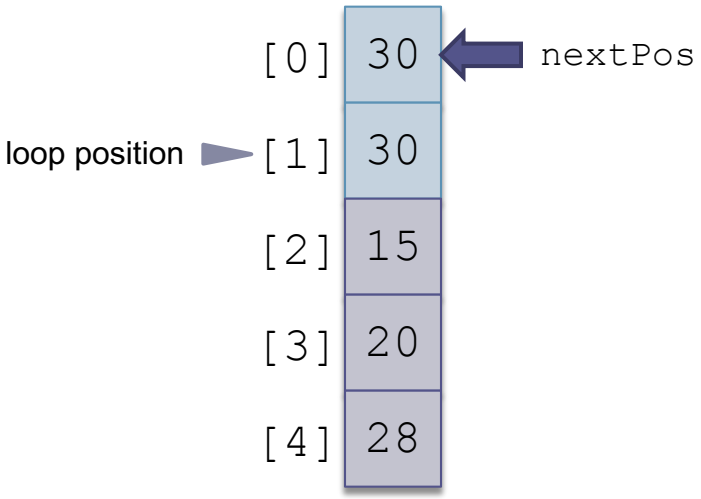
nextPos	1
nextVal	25



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`

Trace of Insertion Sort Refinement (cont.)

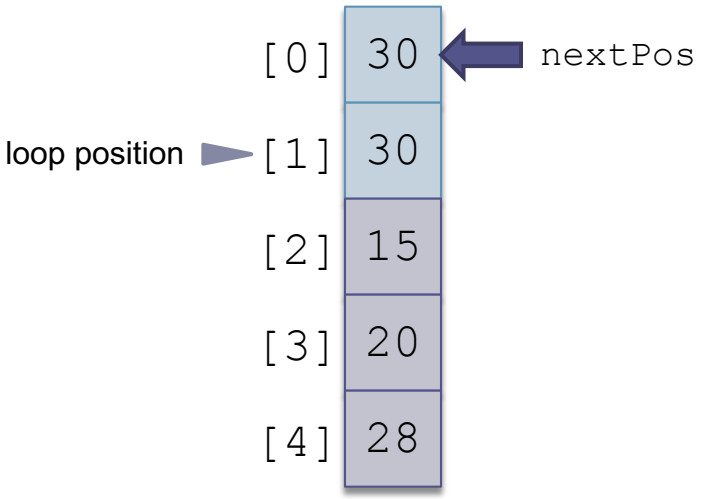
nextPos	0
nextVal	25



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`

Trace of Insertion Sort Refinement (cont.)

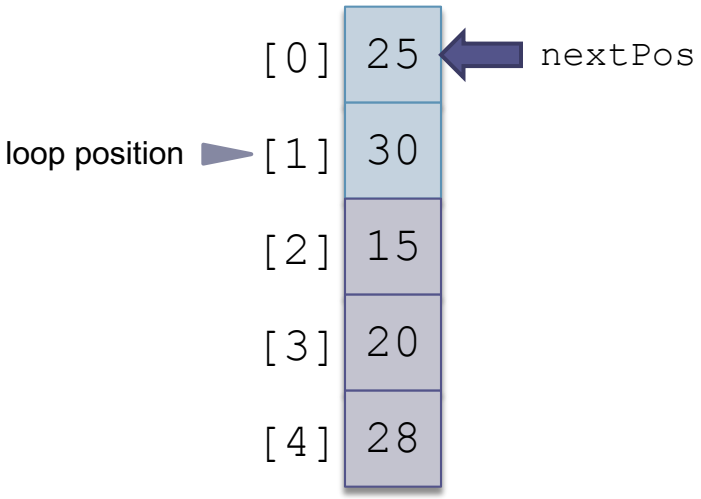
nextPos	0
nextVal	25



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`

Trace of Insertion Sort Refinement (cont.)

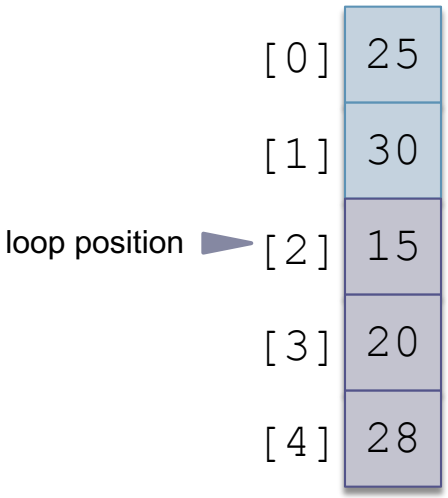
nextPos	0
nextVal	25



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`

Trace of Insertion Sort Refinement (cont.)

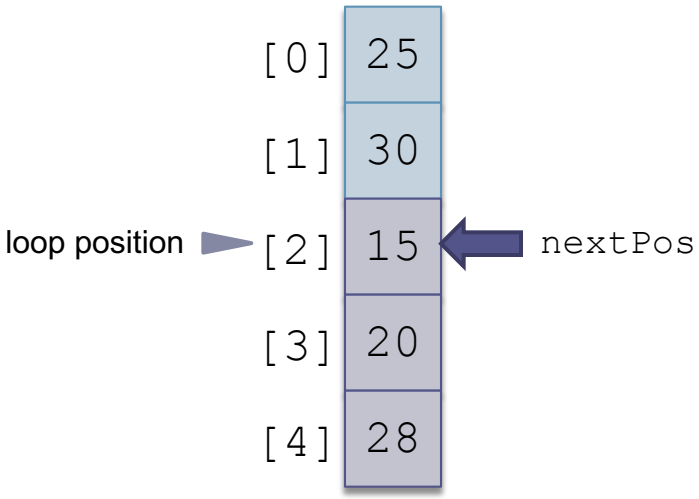
nextPos	0
nextVal	25



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`

Trace of Insertion Sort Refinement (cont.)

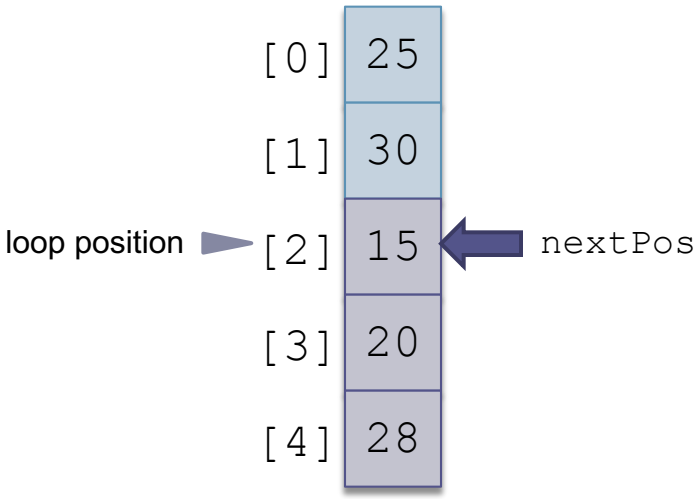
nextPos	2
nextVal	25



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`

Trace of Insertion Sort Refinement (cont.)

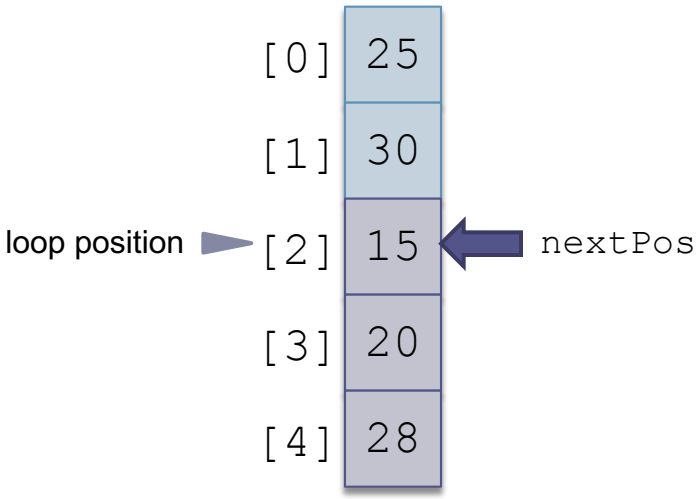
nextPos	2
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

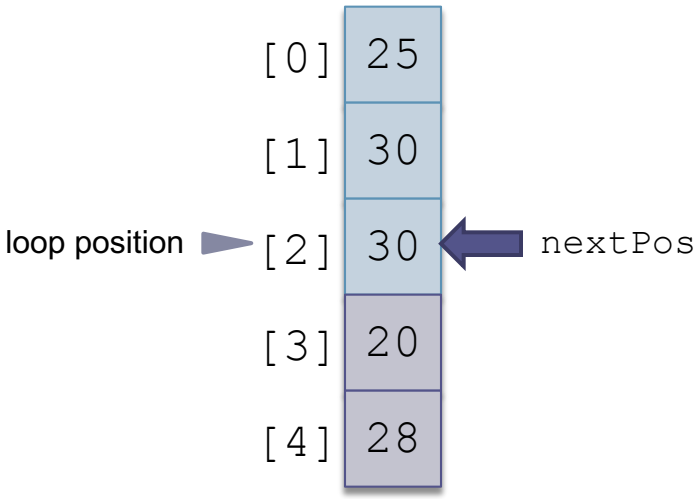
nextPos	2
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

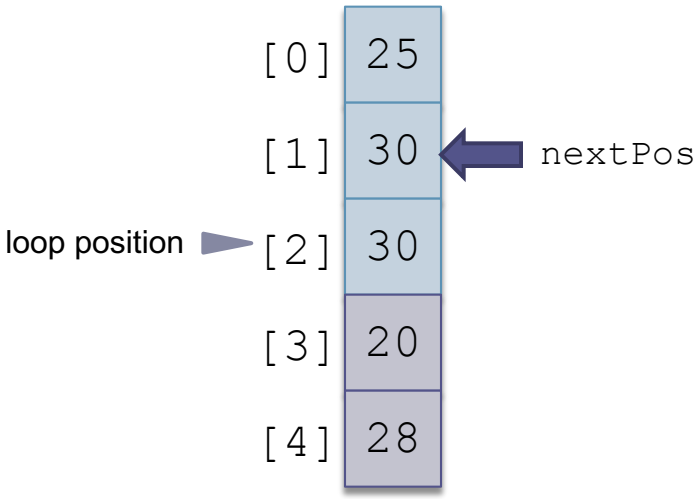
nextPos	2
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

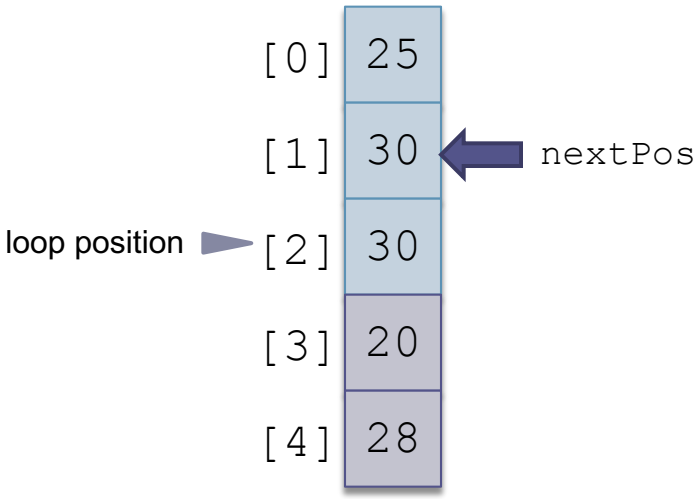
nextPos	1
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

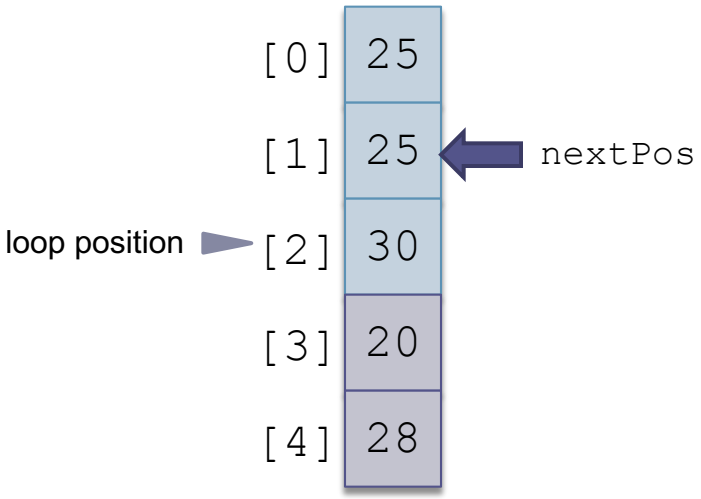
nextPos	1
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

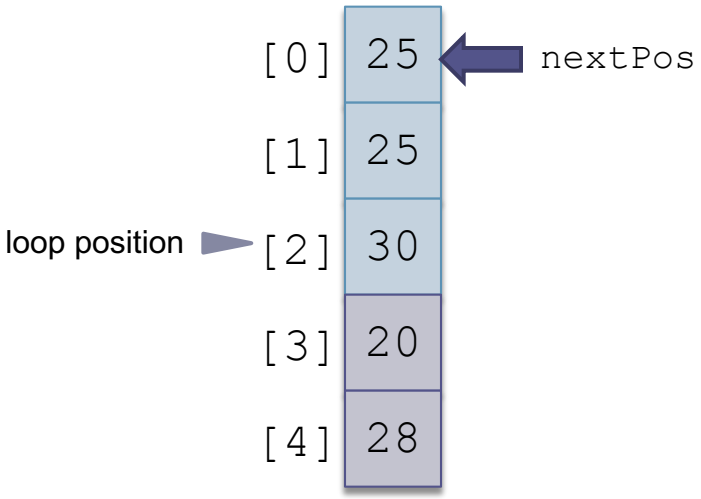
nextPos	1
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

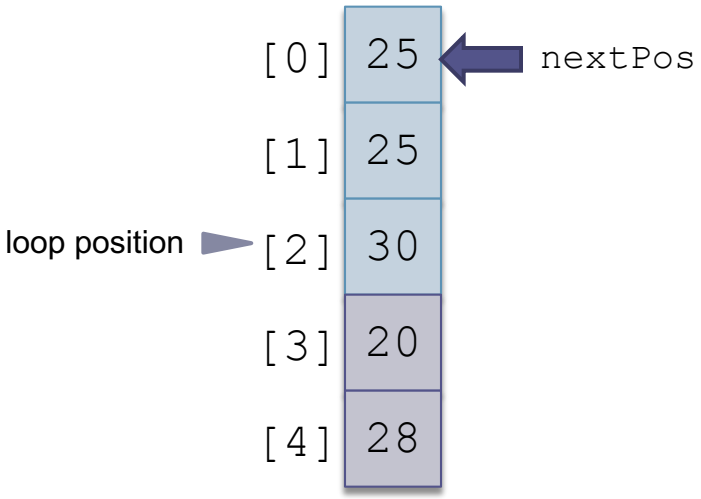
nextPos	0
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

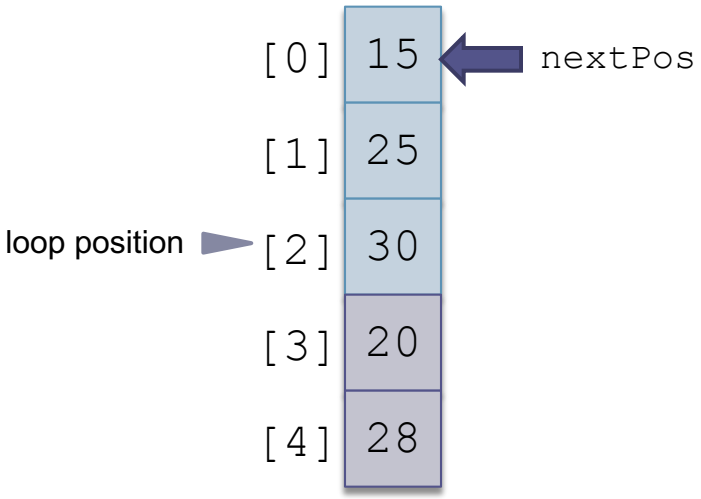
nextPos	0
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

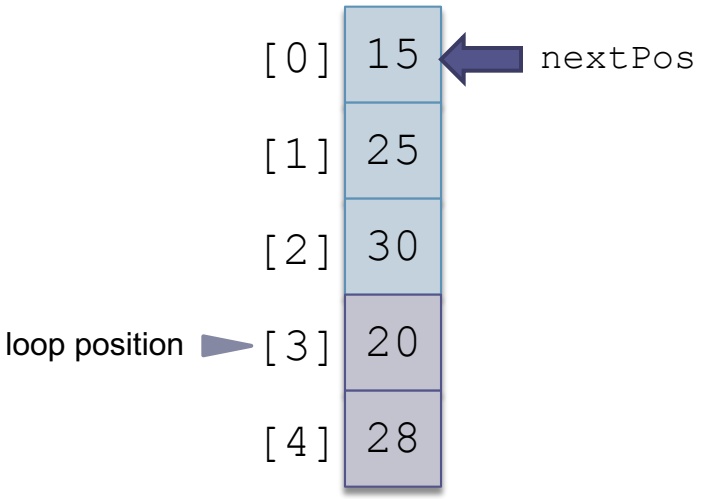
nextPos	0
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

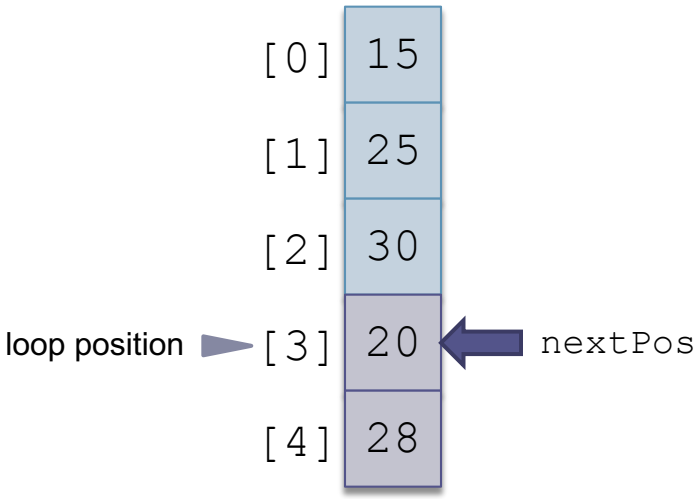
nextPos	0
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

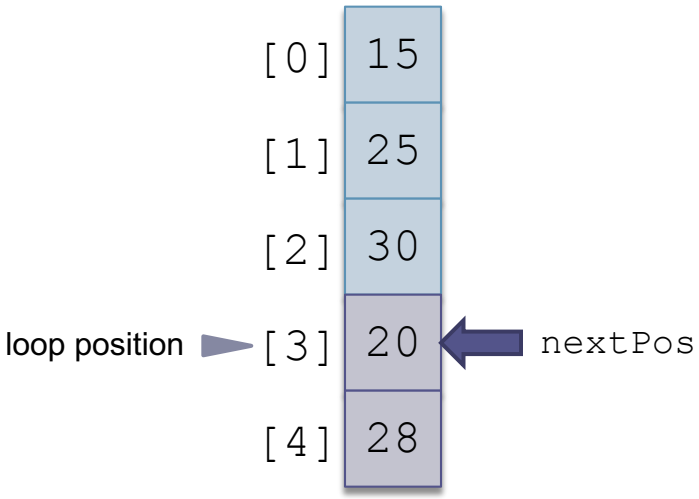
nextPos	3
nextVal	15



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`

Trace of Insertion Sort Refinement (cont.)

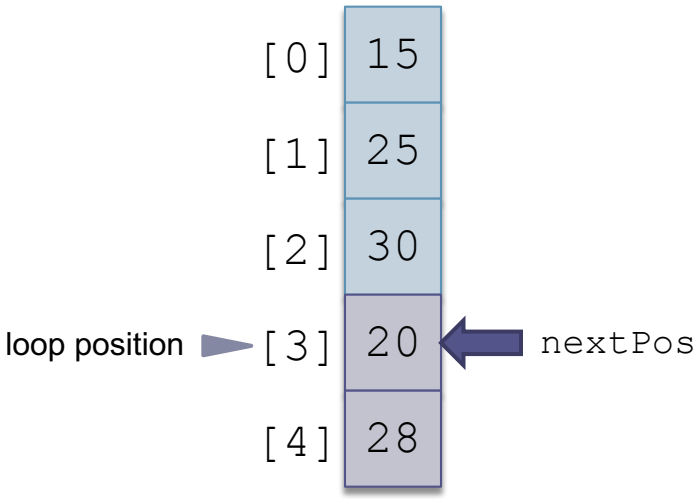
nextPos	3
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

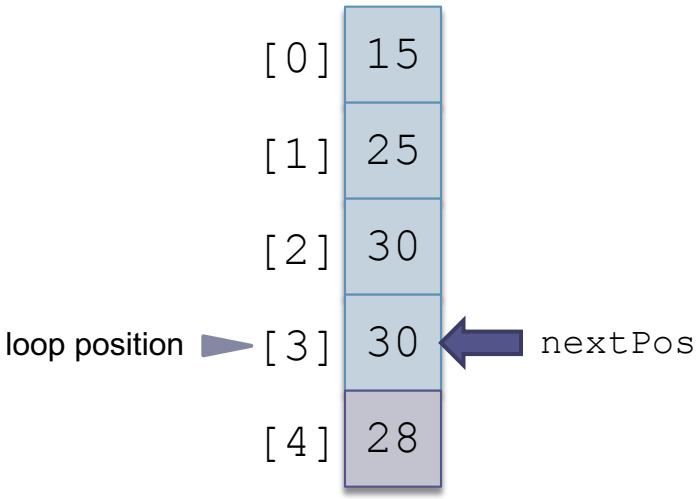
nextPos	3
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

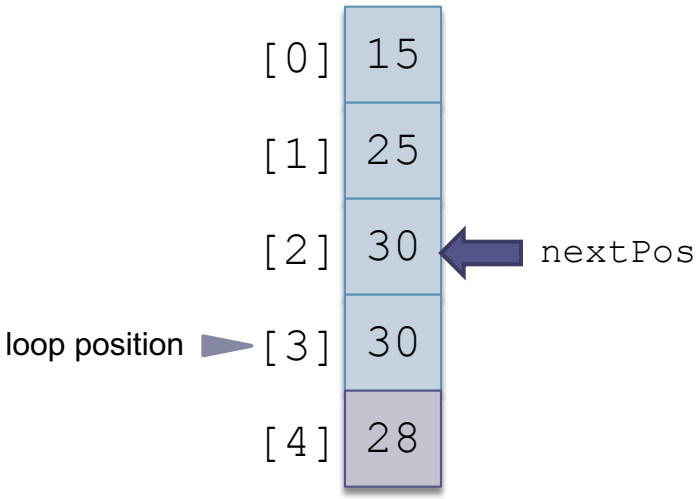
nextPos	3
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

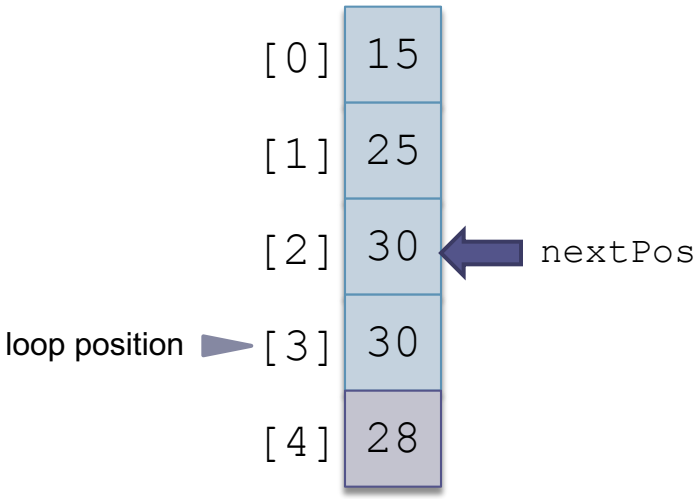
nextPos	2
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

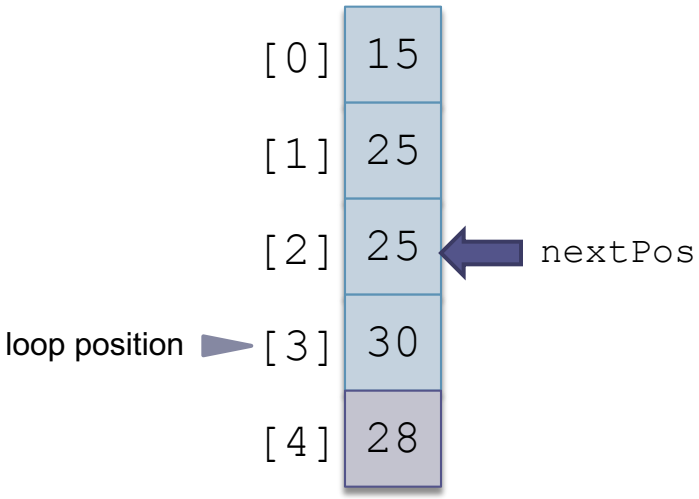
nextPos	2
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

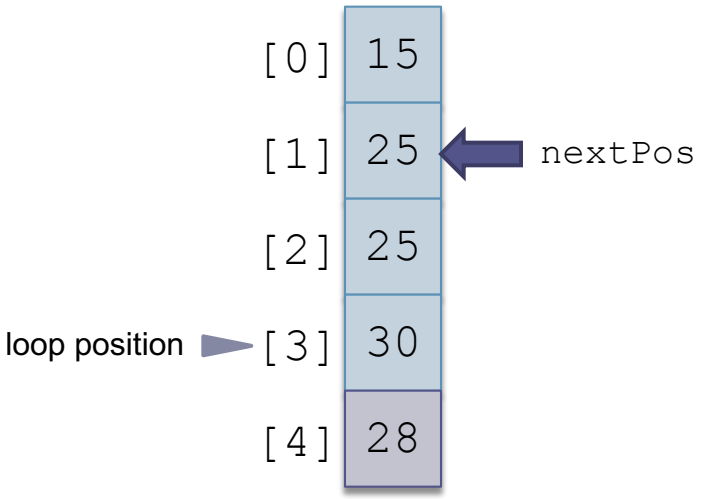
nextPos	2
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

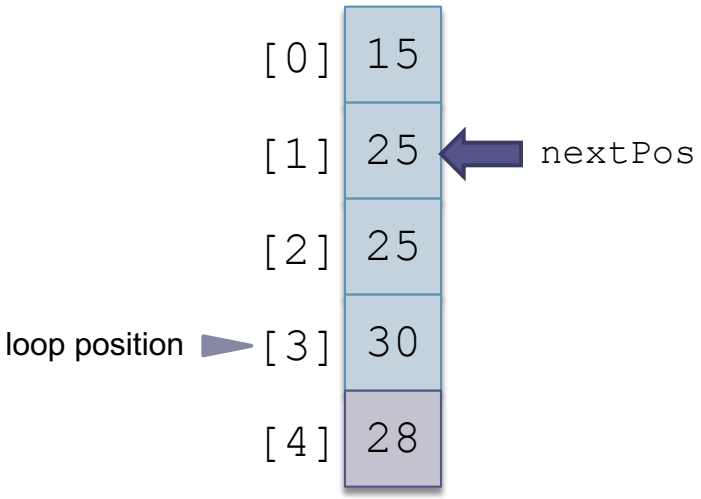
nextPos	1
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

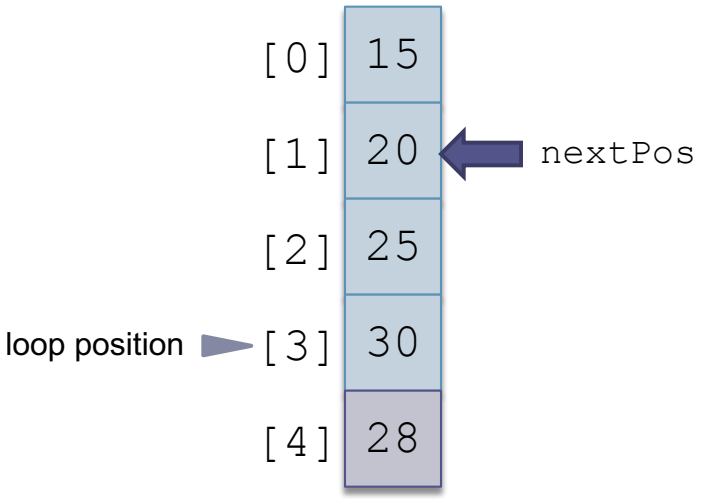
nextPos	1
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

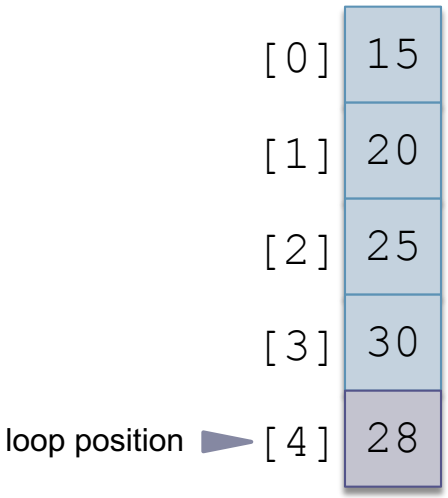
nextPos	1
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

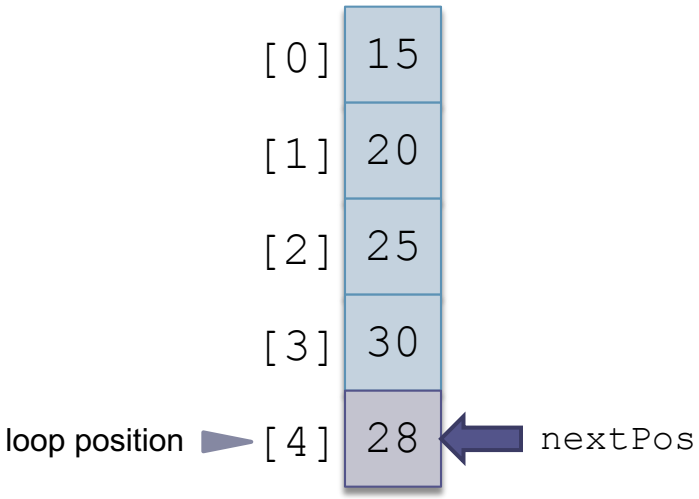
nextPos	1
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

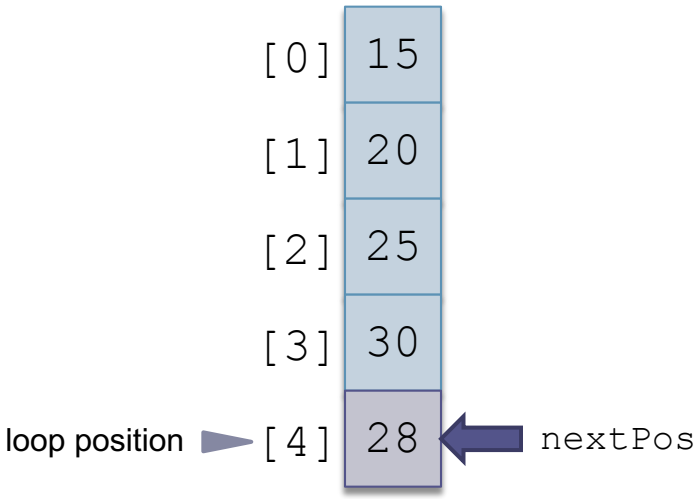
nextPos	4
nextVal	20



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`

Trace of Insertion Sort Refinement (cont.)

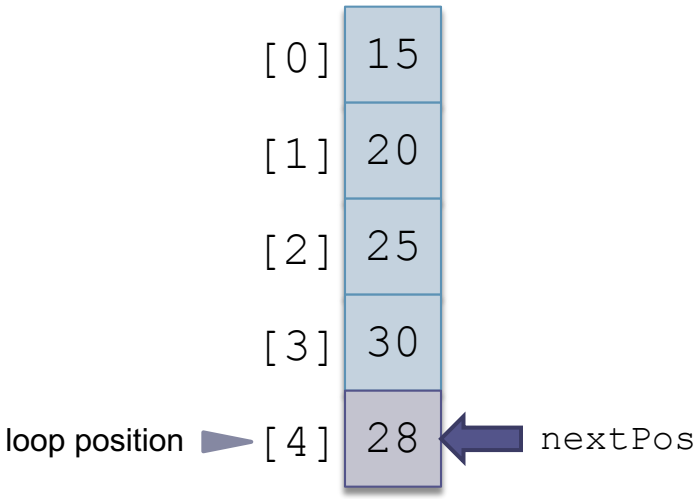
nextPos	4
nextVal	28



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`

Trace of Insertion Sort Refinement (cont.)

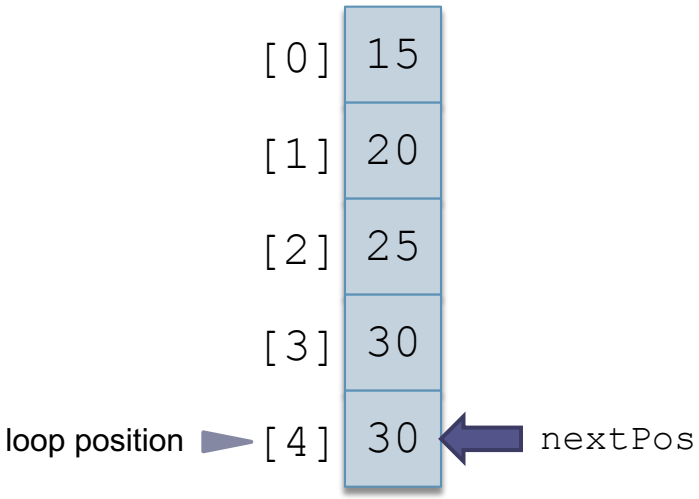
nextPos	4
nextVal	28



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`

Trace of Insertion Sort Refinement (cont.)

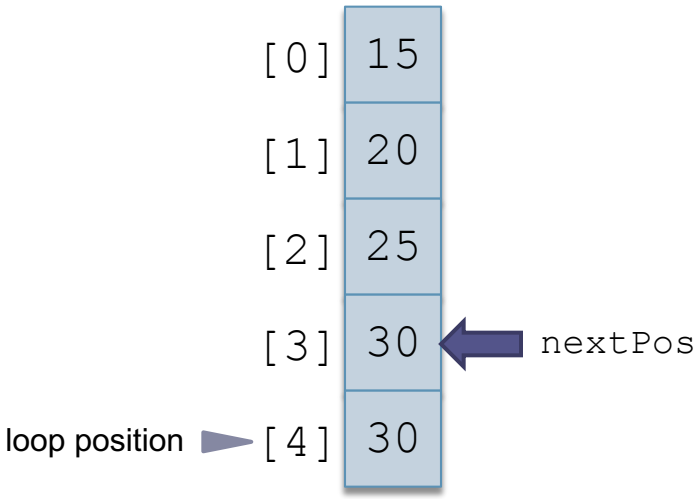
nextPos	4
nextVal	28



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`

Trace of Insertion Sort Refinement (cont.)

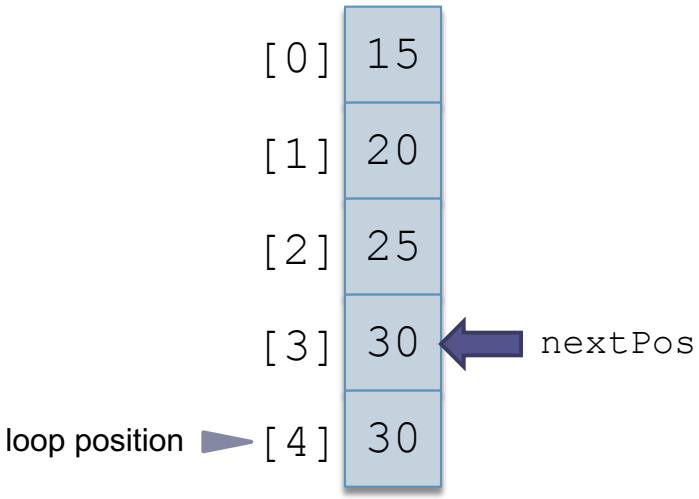
nextPos	3
nextVal	28



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`

Trace of Insertion Sort Refinement (cont.)

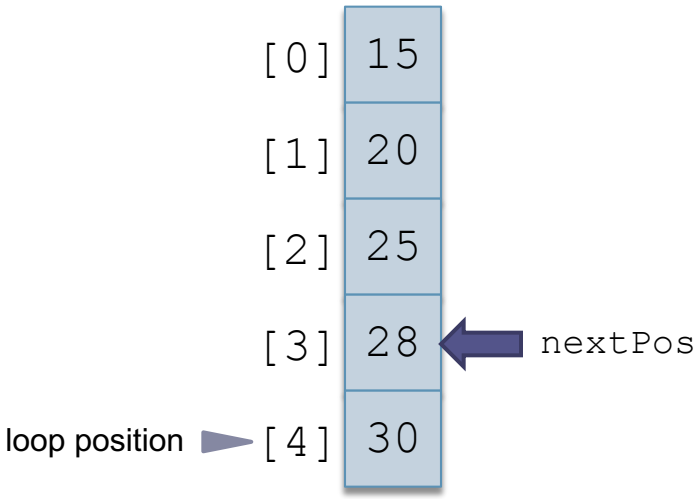
nextPos	3
nextVal	28



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`

Trace of Insertion Sort Refinement (cont.)

nextPos	3
nextVal	28



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`

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 $O(n^2)$

Analysis of Insertion Sort (cont.)

- In the best case (when the array is sorted already), only one comparison is required for each insertion
- In the best case, the number of comparisons is O(*n*)
- The number of shifts performed during an insertion is one less than the number of comparisons
- Or, when the new value is the smallest so far, it is the same as the number of comparisons

Analysis of Insertion Sort (cont.)

- A shift in an insertion sort requires movement of only 1 item, while an exchange in selection sort involves a temporary item and the movement of three items
  - The item moved may be a primitive or an object reference
  - The objects themselves do not change their locations

Code for Insertion Sort


```
public class InsertionSort {
  /** Sort the table using insertion sort algorithm.
      pre:  table contains Comparable objects.
      post: table is sorted.
      @param table The array to be sorted
   */
  public static < T
      extends Comparable < T >> void sort(T[] table) {
   for (int nextPos = 1; nextPos < table.length; nextPos++)
   {
      // Invariant: table[0 . . . nextPos - 1] is sorted.
      // Insert element at position nextPos
      // in the sorted subarray.
      insert(table, nextPos);
```

Code for Insertion Sort


```
    } // End for.
  } // End sort.
/** Insert the element at nextPos where it belongs
      in the array.
      pre:  table[0 . . . nextPos - 1] is sorted.
      post: table[0 . . . nextPos] is sorted.
      @param table The array being sorted
      @param nextPos The position of the element to
            insert
   */
```

Code for Insertion Sort


```
private static < T extends Comparable < T >>
      void insert(T[] table, int nextPos) {
    T nextVal = table[nextPos]; // Element to insert.
    while (nextPos > 0
          && nextVal.compareTo(table[nextPos - 1]) < 0)
  {
      table[nextPos] = table[nextPos - 1]; // Shift down.
      nextPos--; // Check next smaller element.
    }
    // Insert nextVal at nextPos.
    table[nextPos] = nextVal;
  }
}
```

Comparison of Quadratic Sorts (cont.)

Comparison of growth rates

n	n^2	$n \log n$
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

Comparison of Quadratic Sorts (cont.)

- Insertion sort
  - gives the best performance for most arrays
  - takes advantage of any partial sorting in the array and uses less costly shifts
- None of the quadratic search algorithms are particularly good for large arrays ($n > 1000$)
- The best sorting algorithms provide $n \log n$ average case performance

Comparison of Quadratic Sorts (cont.)

- All quadratic sorts require storage for the array being sorted
- However, the array is sorted in place
- While there are also storage requirements for variables, for large *n*, the size of the array dominates and extra space usage is O(1)

Comparisons versus Exchanges

- In Java, an exchange requires a switch of two object references using a third object reference as an intermediary
- A comparison requires an execution of a `compareTo` method
- The cost of a comparison depends on its complexity, but is generally more costly than an exchange
- For some other languages, an exchange may involve physically moving information rather than swapping object references. In these cases, an exchange may be more costly than a comparison
  - In some cases, writing to memory is much slower than reading

Merge Sort

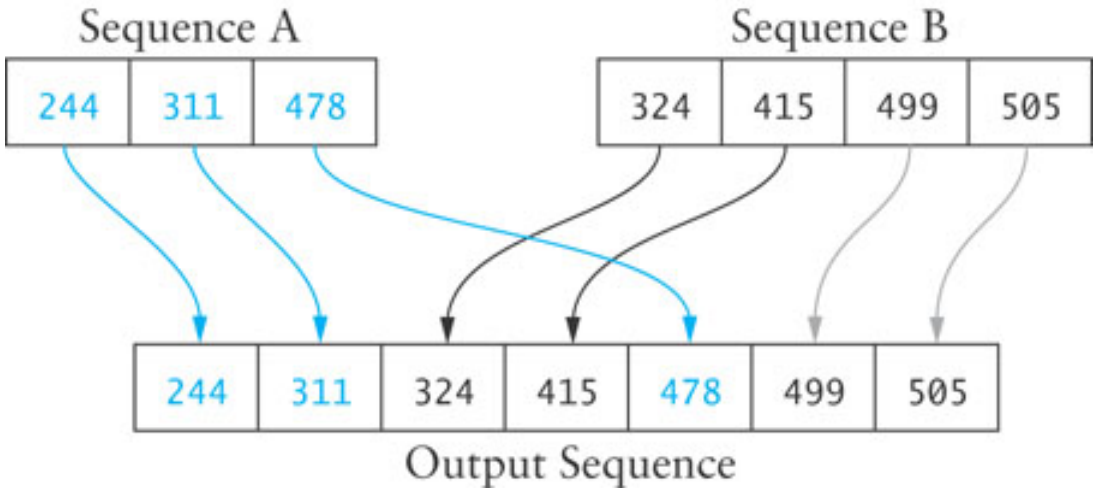
Section 8.6

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

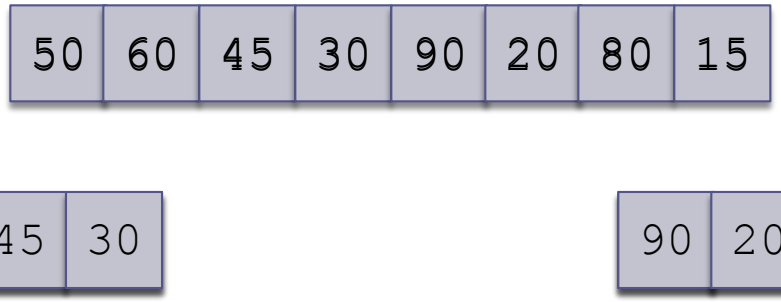
- This algorithm can be written with a recursive step

(recursive) Algorithm for Merge Sort

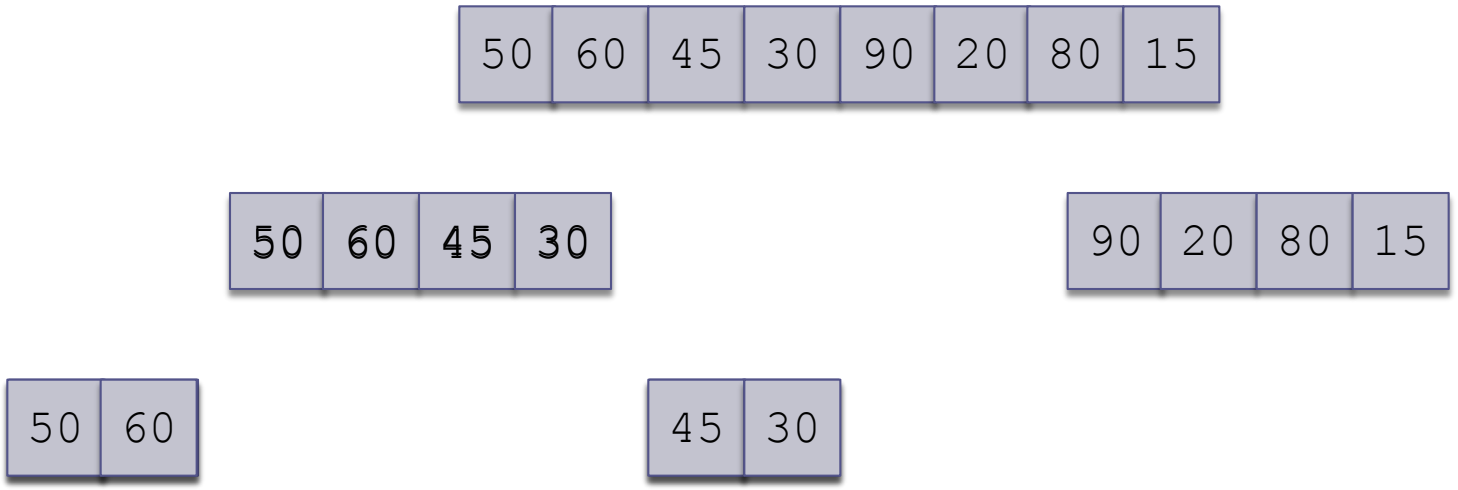
Algorithm for Merge Sort

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.

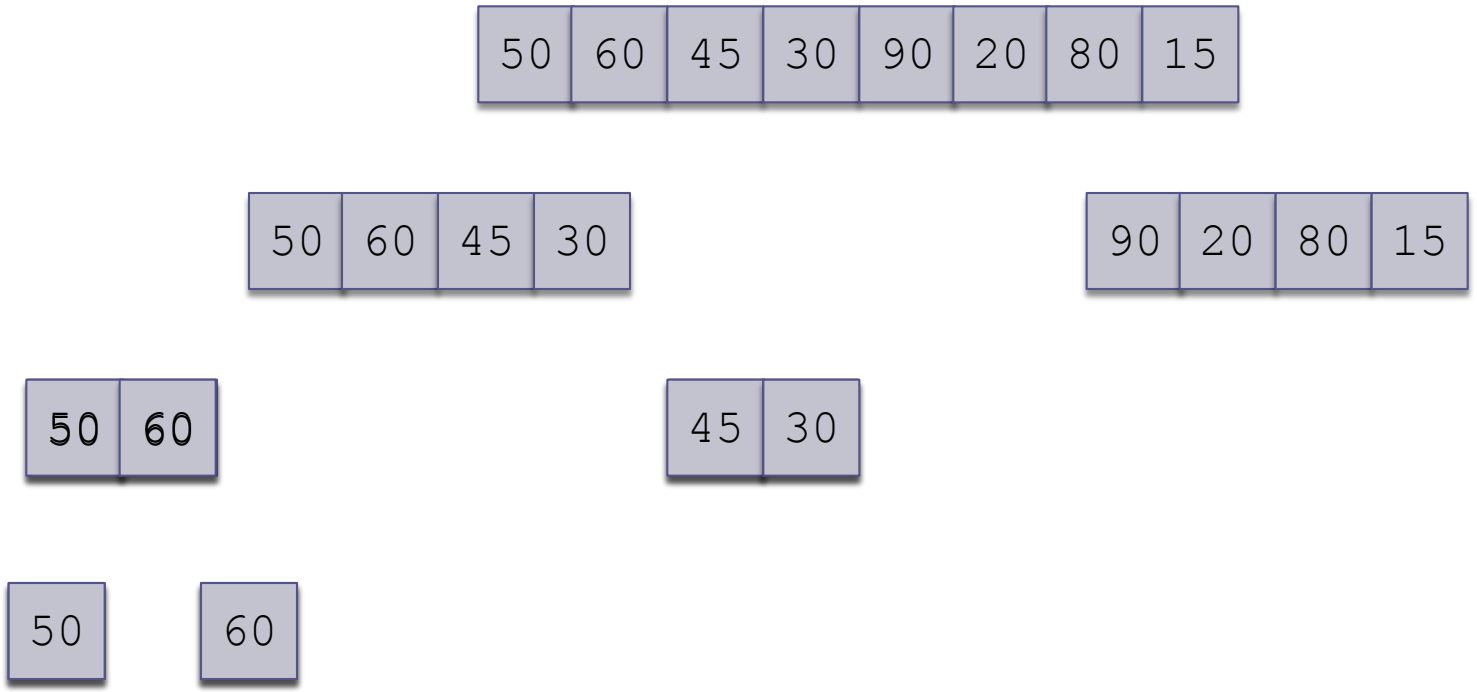
Trace of Merge Sort



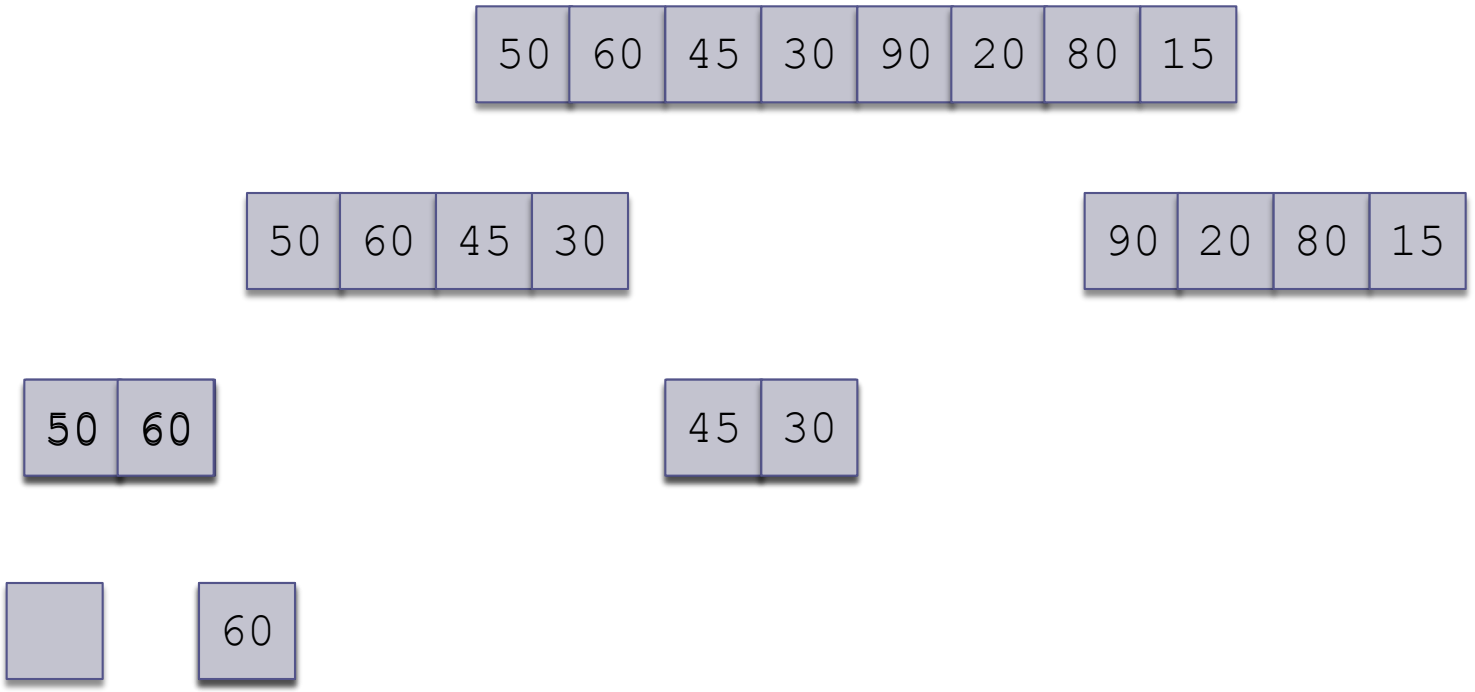
Trace of Merge Sort (cont.)



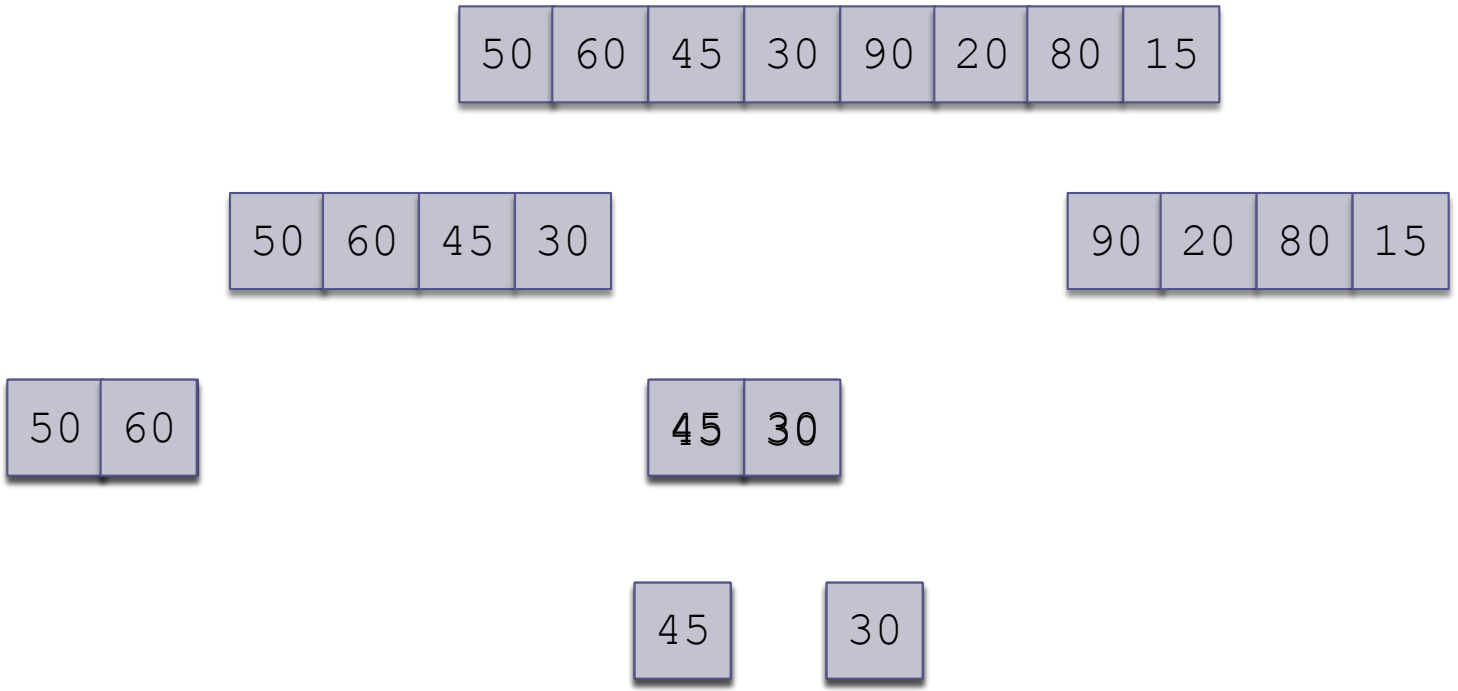
Trace of Merge Sort (cont.)



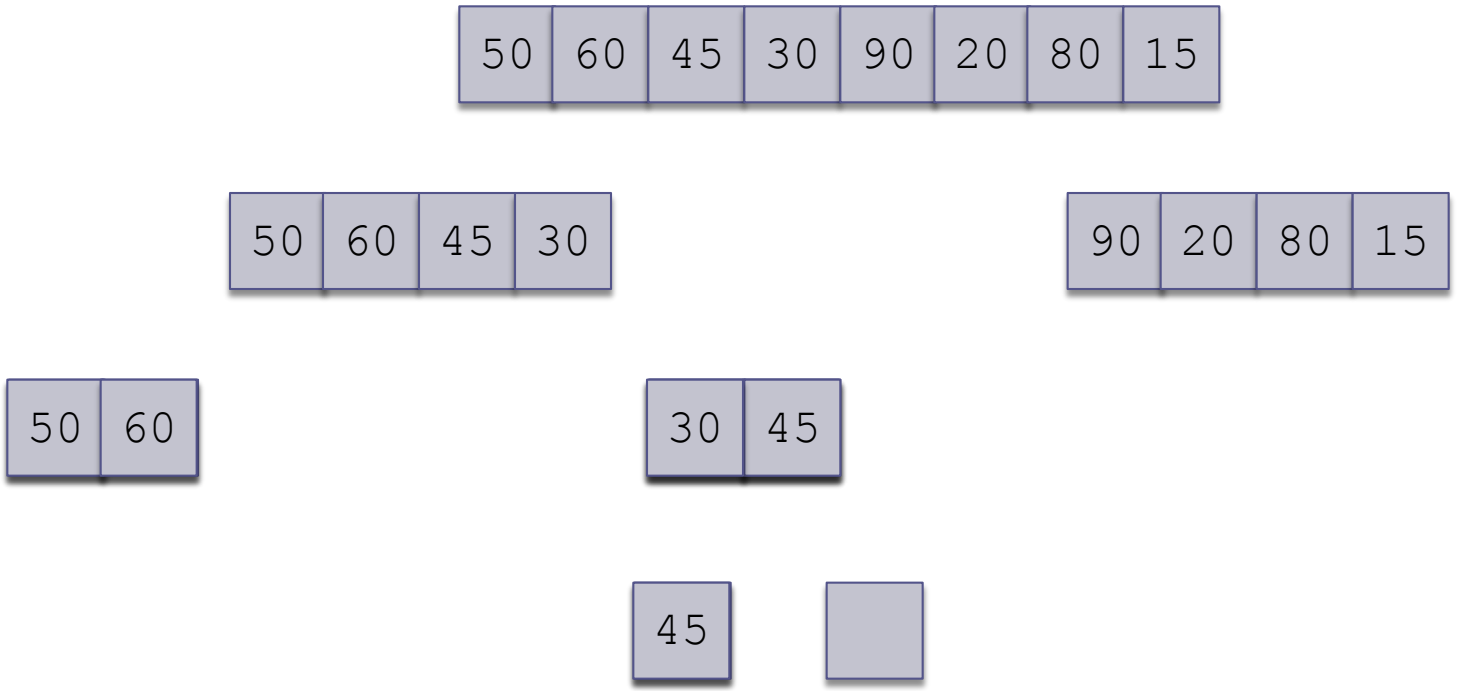
Trace of Merge Sort (cont.)



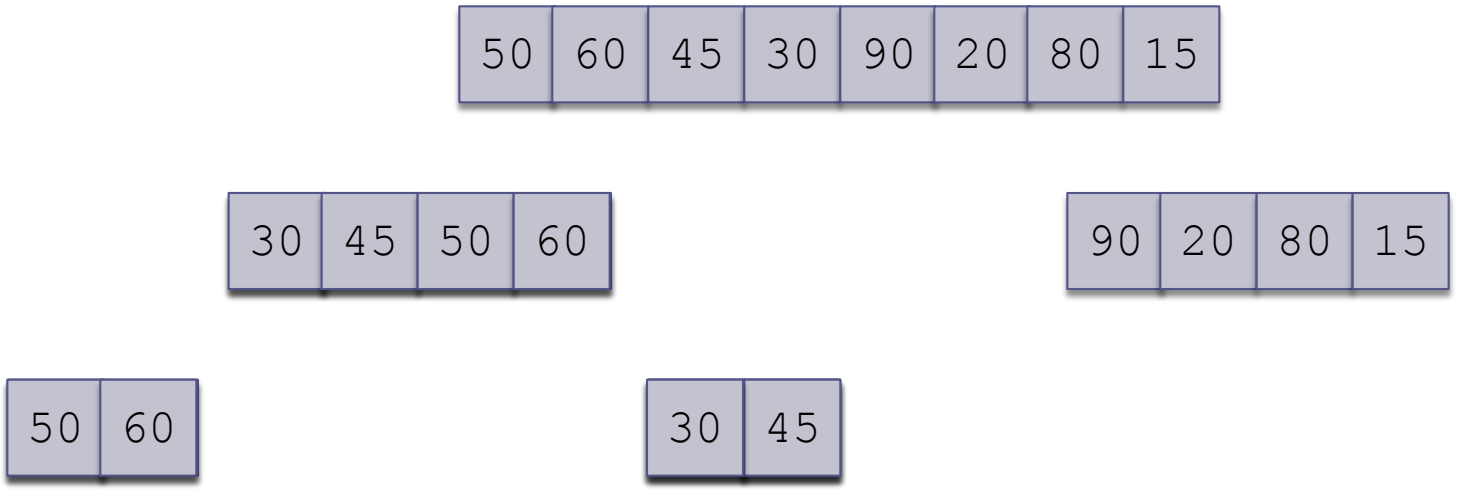
Trace of Merge Sort (cont.)



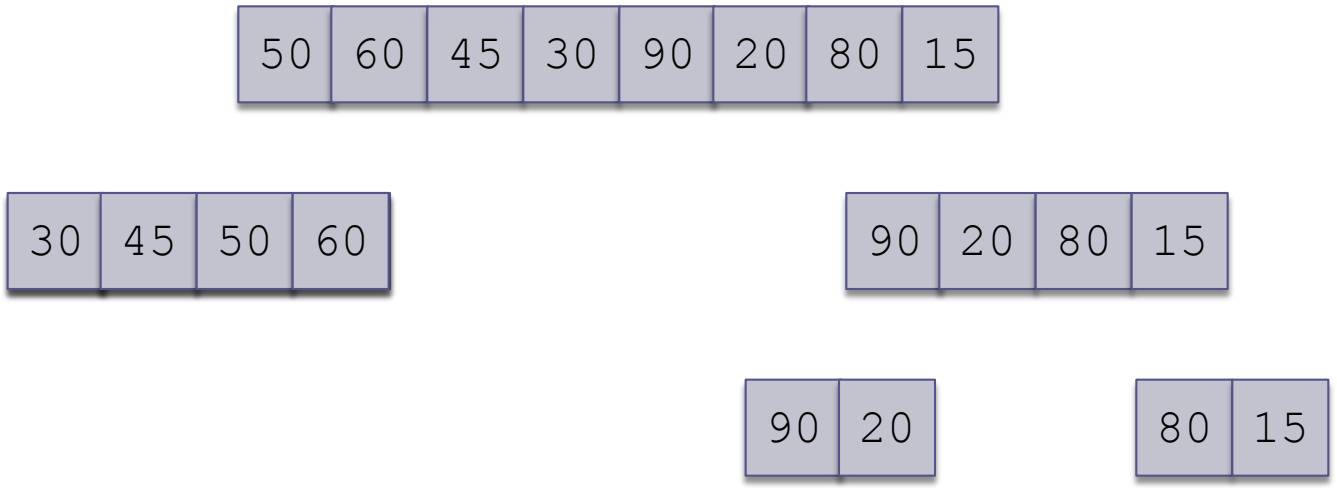
Trace of Merge Sort (cont.)



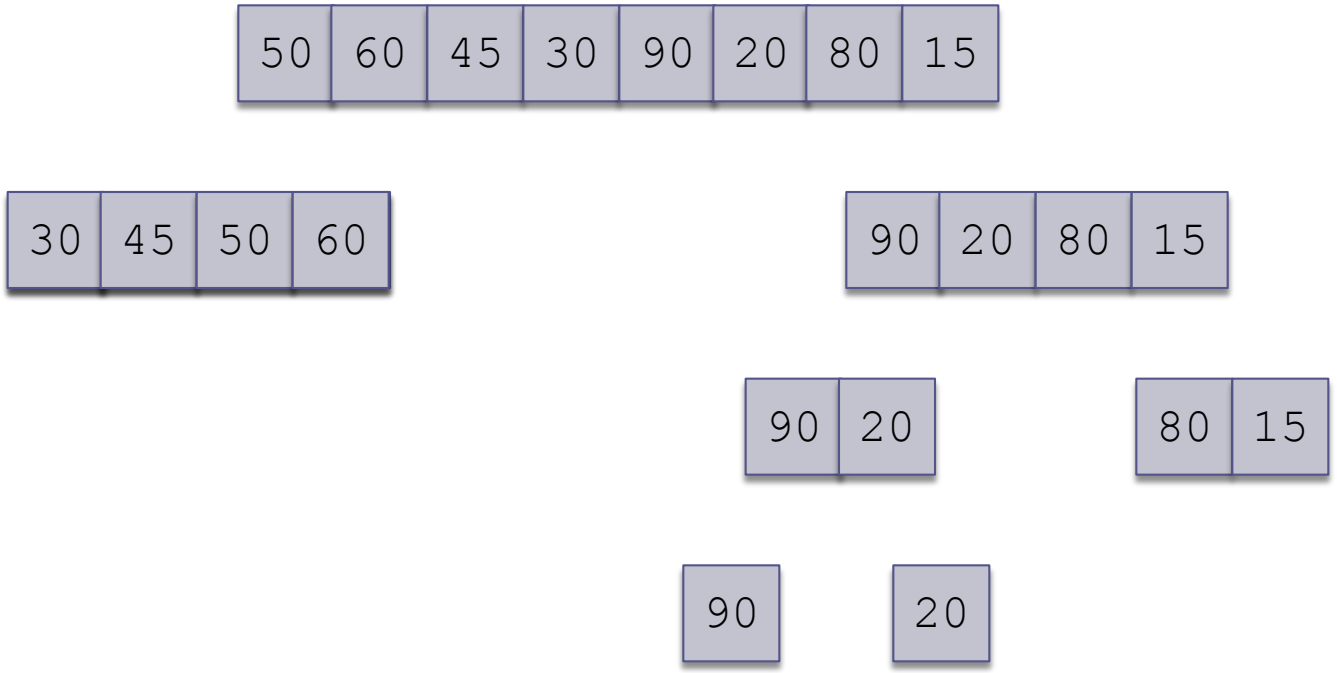
Trace of Merge Sort (cont.)



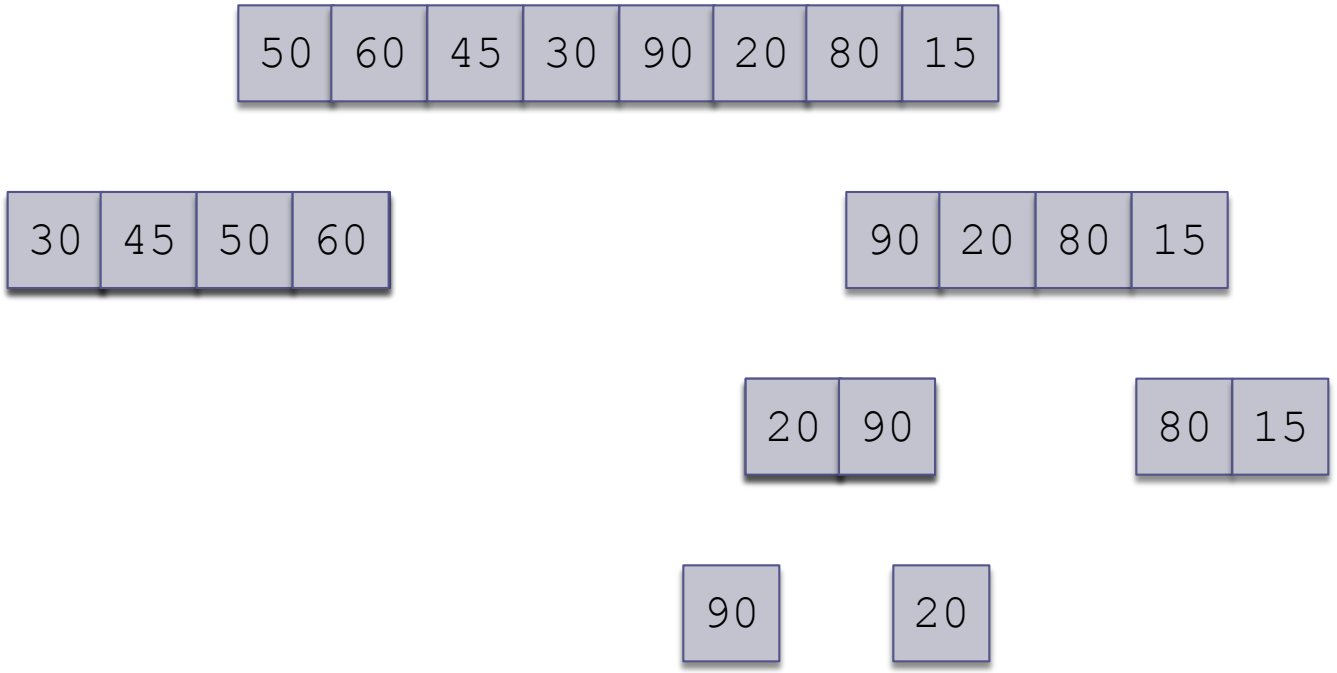
Trace of Merge Sort (cont.)



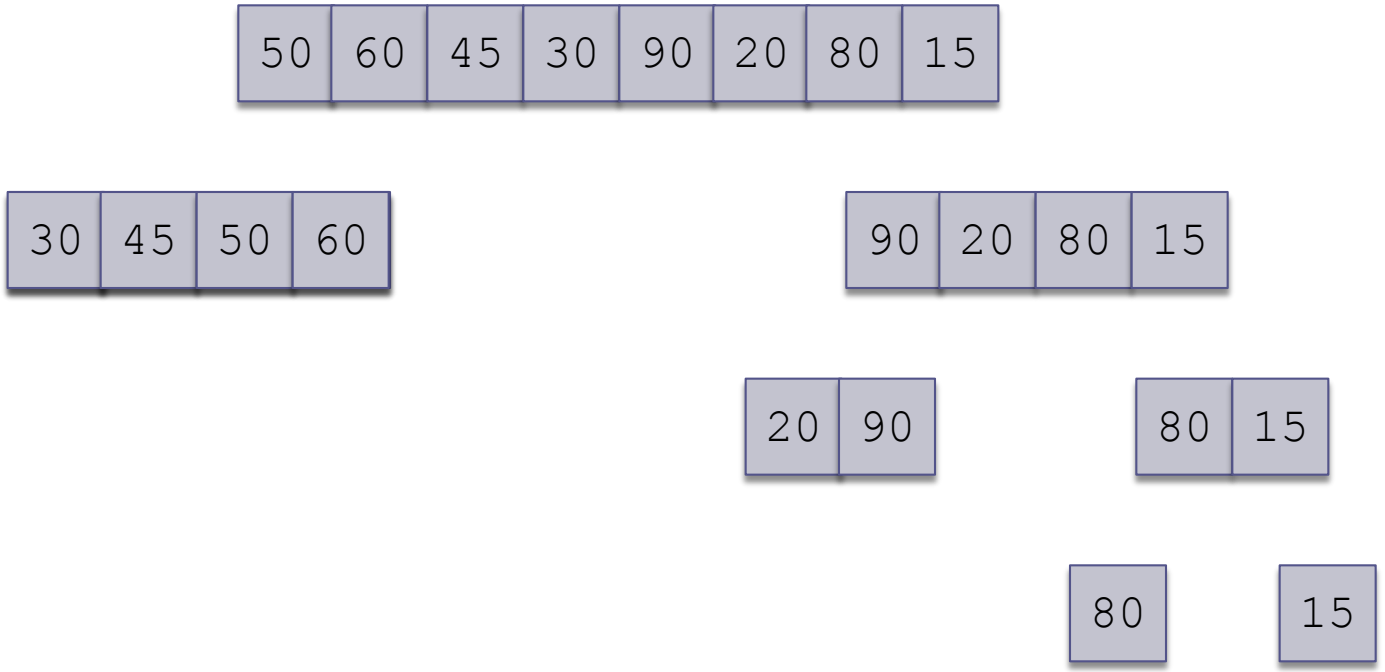
Trace of Merge Sort (cont.)



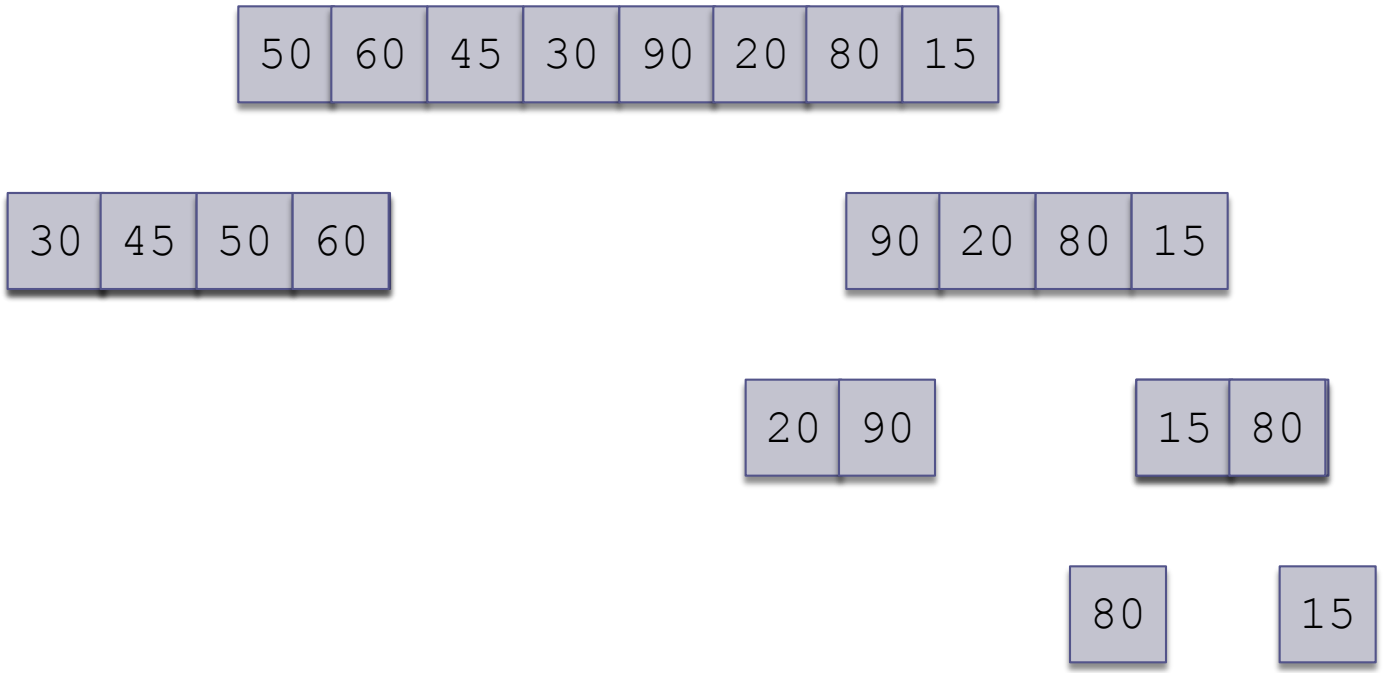
Trace of Merge Sort (cont.)



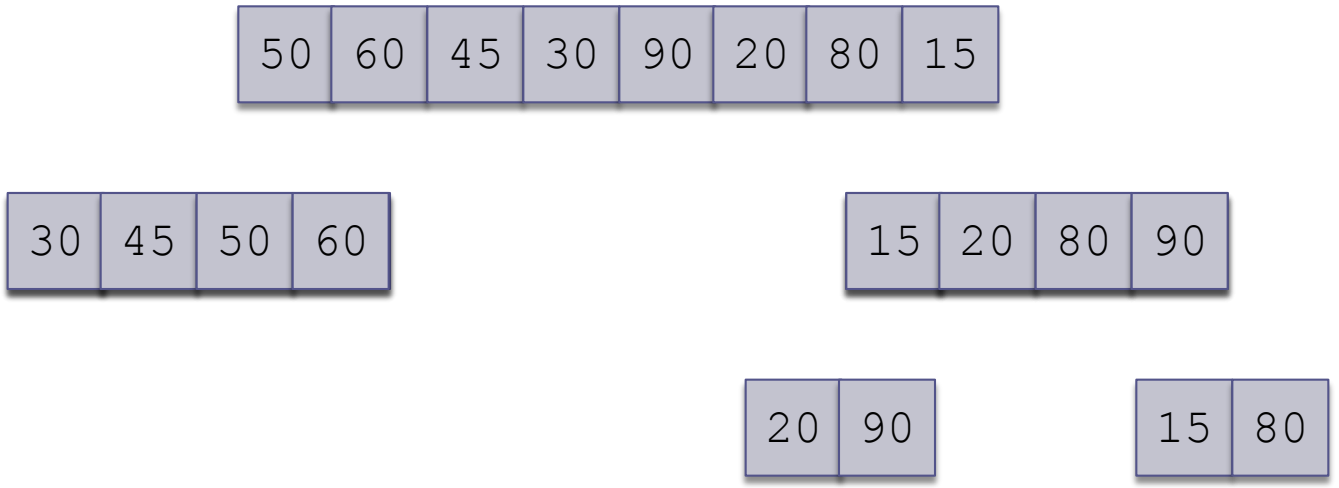
Trace of Merge Sort (cont.)



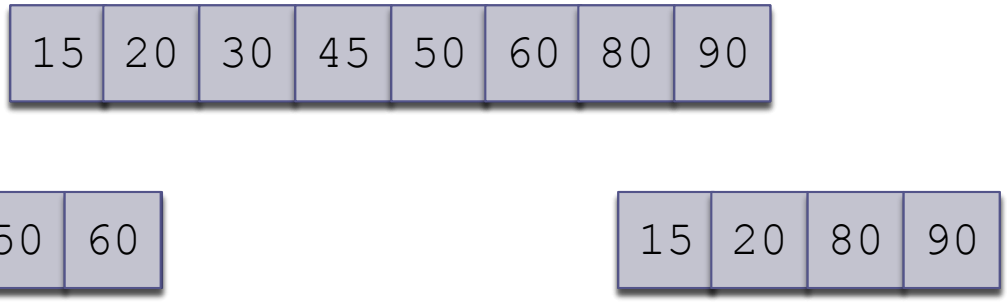
Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



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 lines which require merging at each step is log *n* because each recursive step splits the array in half
  - A line here refers to a subdivision of all current arrays
- The total effort to reconstruct the sorted array through merging is O(*n* log*n*)

Analysis of Merge Sort (cont.)

- Going down through the recursion chain, sorting the left tables, a sequence of right tables of size

$$\frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^k}$$

  is allocated

- Since

$$\frac{n}{2} + \frac{n}{4} + \ldots + 2 + 1 = n - 1$$

  a total of *n* additional storage locations are required

Code for Merge Sort


```
/** Implements the recursive merge sort algorithm. In this
   version, copies of the subtables are made, sorted, and
   then merged.
*    @author Koffman and Wolfgang
*/

public class MergeSort {
  /** Sort the array using the merge sort algorithm.
      pre: table contains Comparable objects.
      post: table is sorted.
      @param table The array to be sorted
   */
```

Code for Merge Sort


```
public static < T
      extends Comparable < T >> void sort(T[] table) {
    // A table with one element is sorted already.
    if (table.length > 1) {
      // Split table into halves.
      int halfSize = table.length / 2;
      T[] leftTable = (T[])new Comparable[halfSize];
      T[] rightTable =
          (T[])new Comparable[table.length - halfSize];
```

Code for Merge Sort


```
        System.arraycopy(table, 0, leftTable, 0, halfSize);
        System.arraycopy(table, halfSize, rightTable, 0,
                         table.length - halfSize);
        //Sort the halves.
        sort(leftTable);
        sort(rightTable);

        // Merge the halves.
        merge(table, leftTable, rightTable);
    }
}
```

Quicksort

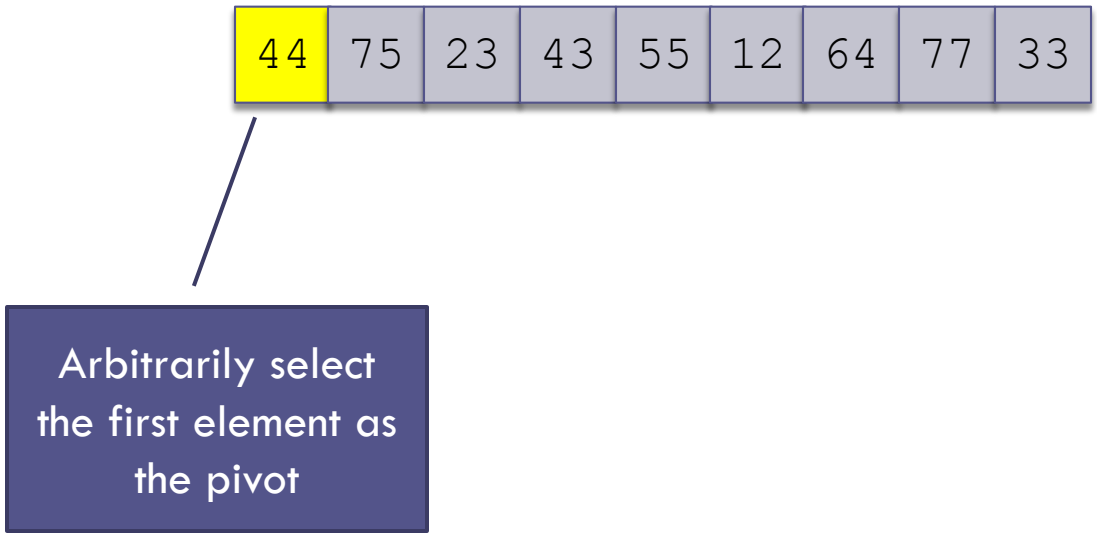
Section 8.9

Quicksort

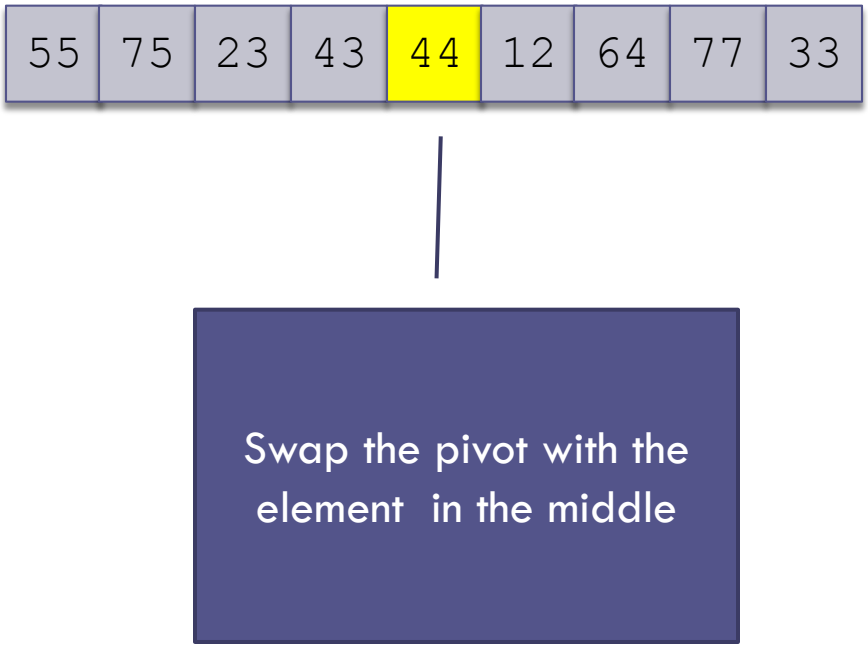
- Developed in 1962
- Quicksort selects a specific value called a pivot and rearranges the array into two parts (called *partioning*)
  - all the elements in the left subarray are less than or equal to the pivot
  - all the elements in the right subarray are larger than the pivot
  - The pivot is placed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

Trace of Quicksort

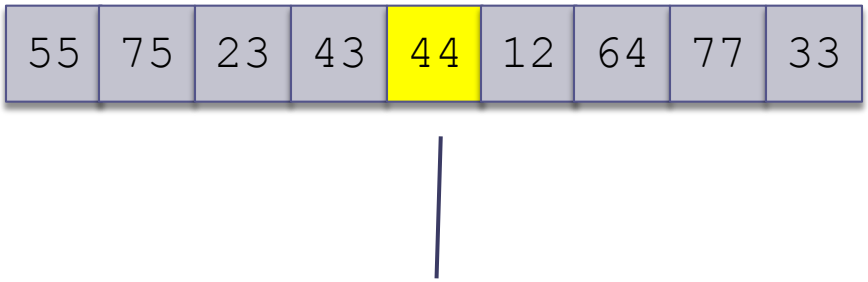
Trace of Quicksort (cont.)



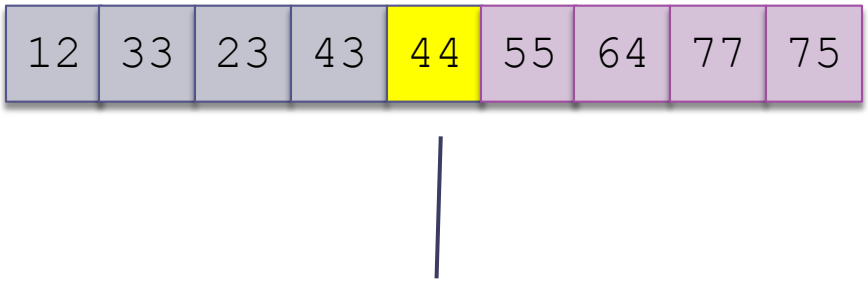
Trace of Quicksort (cont.)



Trace of Quicksort (cont.)

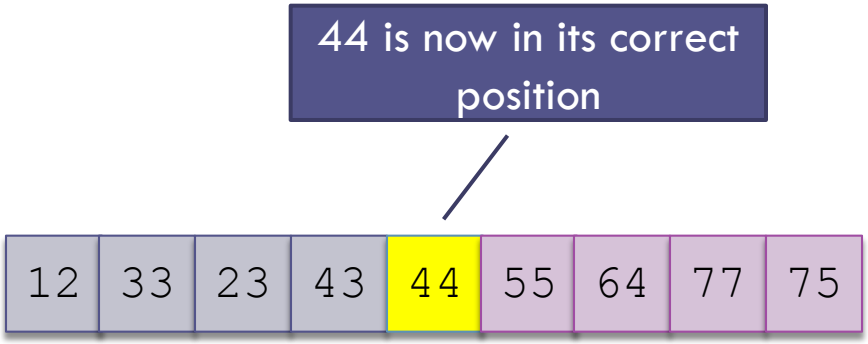


Trace of Quicksort (cont.)

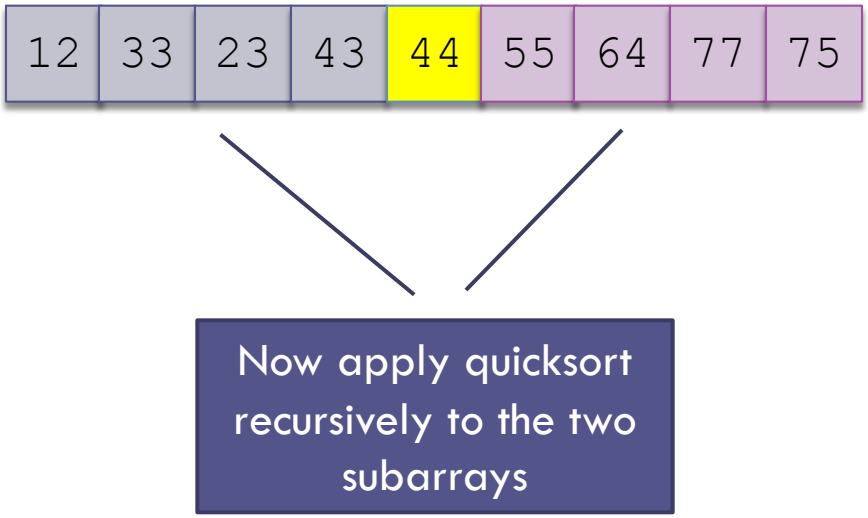


Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right

Quicksort Example(cont.)



Trace of Quicksort (cont.)



Trace of Quicksort (cont.)

Pivot value = 12

Trace of Quicksort (cont.)

Pivot value = 12

Trace of Quicksort (cont.)

Pivot value = 33

Trace of Quicksort (cont.)

Pivot value = 33

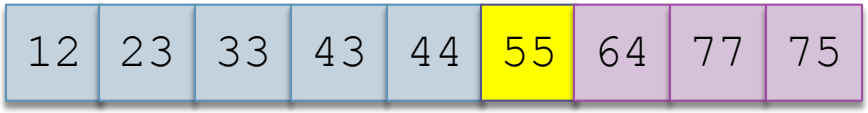
Trace of Quicksort (cont.)

Pivot value = 33

Left and right subarrays have single values; they are sorted

Trace of Quicksort (cont.)

Pivot value = 55



Trace of Quicksort (cont.)

Pivot value = 64

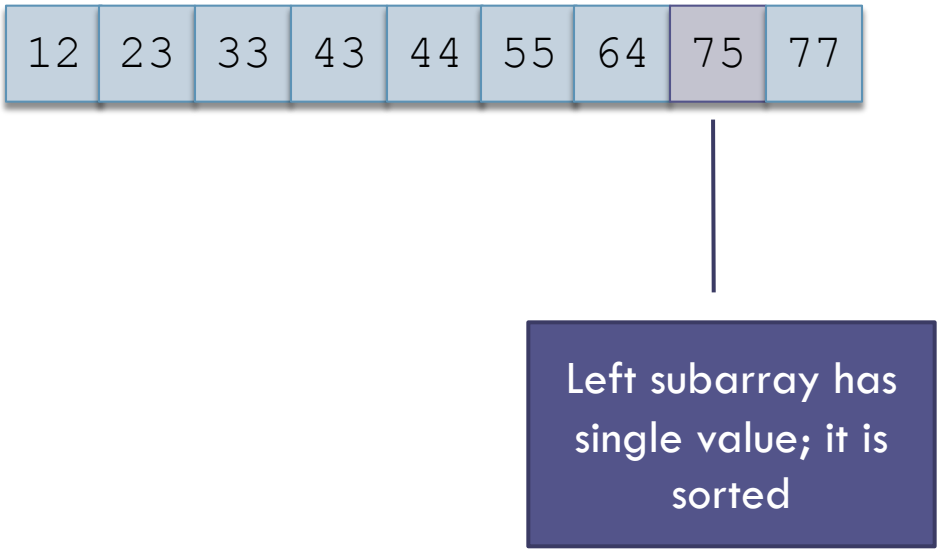
Trace of Quicksort (cont.)

Pivot value = 77

Trace of Quicksort (cont.)

Pivot value = 77

Trace of Quicksort (cont.)



Algorithm for Quicksort

- We describe how to do the partitioning later
- 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 bound 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
  - We'll discuss a solution later

Code for Quicksort


```
/** Implements the quicksort algorithm.
*   @author Koffman and Wolfgang
 */
public class QuickSort {
  /** Sort the table using the quicksort algorithm.
      pre: table contains Comparable objects.
      post: table is sorted.
      @param table The array to be sorted
   */
  public static < T
      extends Comparable < T >> void sort(T[] table) {
    // Sort the whole table.
    quickSort(table, 0, table.length - 1);
  }
```

Code for Quicksort


```
/** Sort a part of the table using the quicksort
algorithm.
    post: The part of table from first through last is
sorted.
    @param table The array to be sorted
    @param first The index of the low bound
    @param last The index of the high bound
 */
private static < T
    extends Comparable < T >> void quickSort(T[] table,
                                int first, int last) {
  if (first < last) { // There is data to be sorted.
    // Partition the table.
    int pivIndex = partition(table, first, last);
```

Code for Quicksort


```
      // Sort the left half.
       quickSort(table, first, pivIndex - 1);
       // Sort the right half.
       quickSort(table, pivIndex + 1, last);
     }
  }

  // Insert partition method
}
```

Algorithm for Partitioning

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript `first`

Trace of Partitioning (cont.)

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript `first`

Trace of Partitioning (cont.)

For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

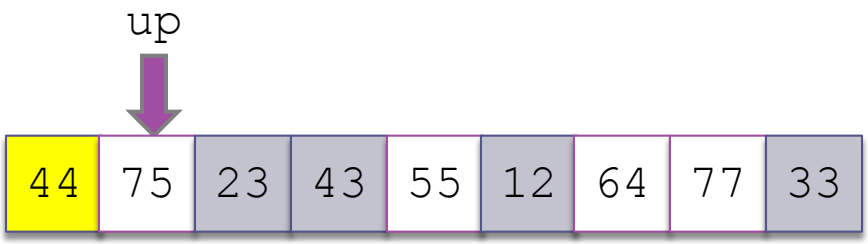
Trace of Partitioning (cont.)

For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

Trace of Partitioning (cont.)

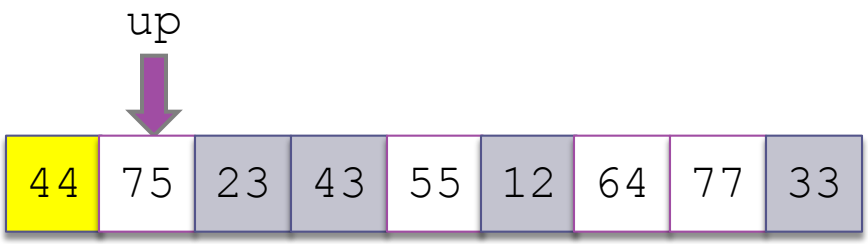
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



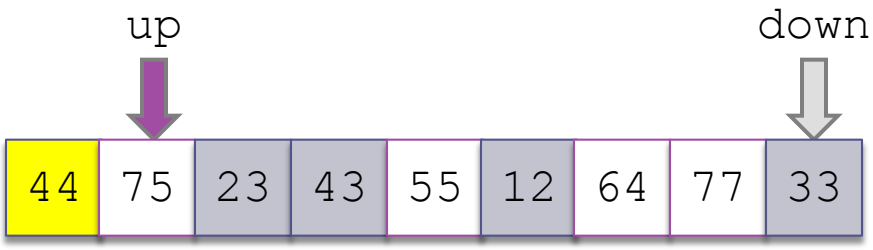
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



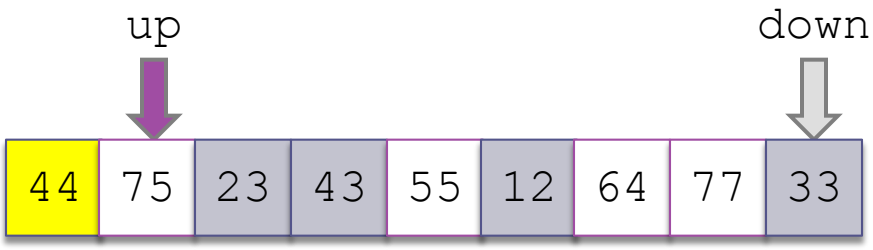
Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)



Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)



Exchange these values

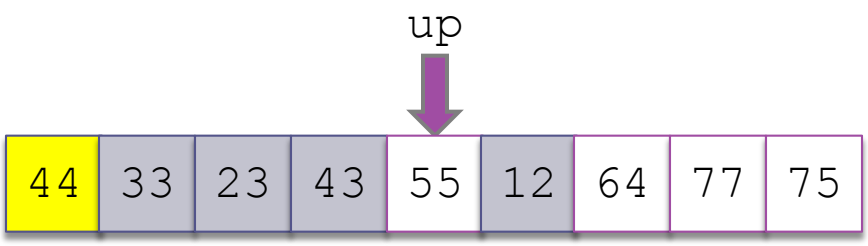
Trace of Partitioning (cont.)

Exchange these values

Trace of Partitioning (cont.)

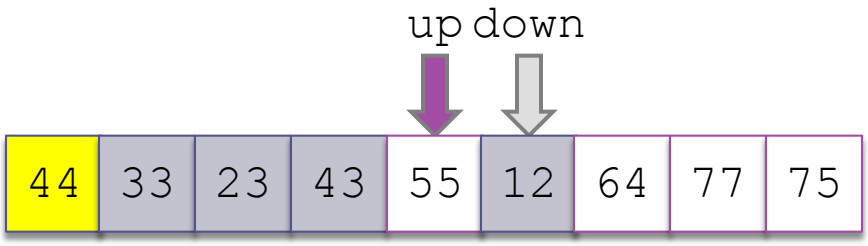
Repeat

Trace of Partitioning (cont.)



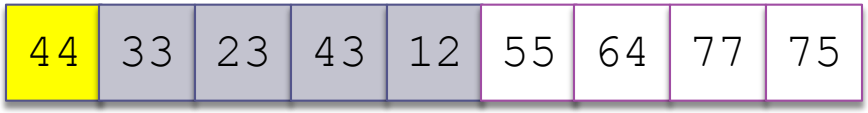
Find first value at left end greater than pivot

Trace of Partitioning (cont.)



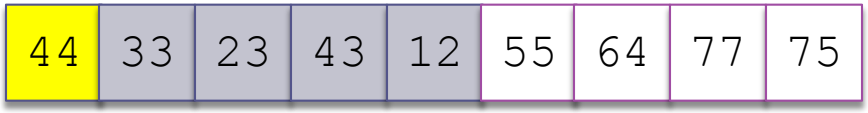
Find first value at right end less than or equal to pivot

Trace of Partitioning (cont.)



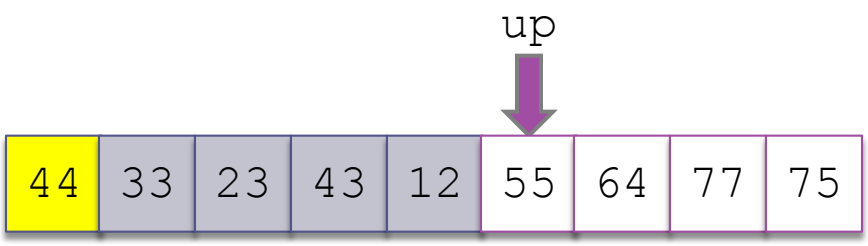
Exchange

Trace of Partitioning (cont.)



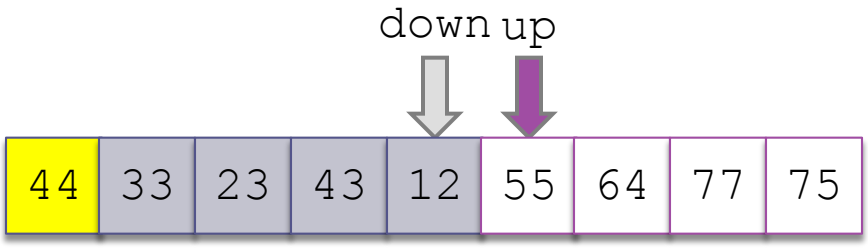
Repeat

Trace of Partitioning (cont.)



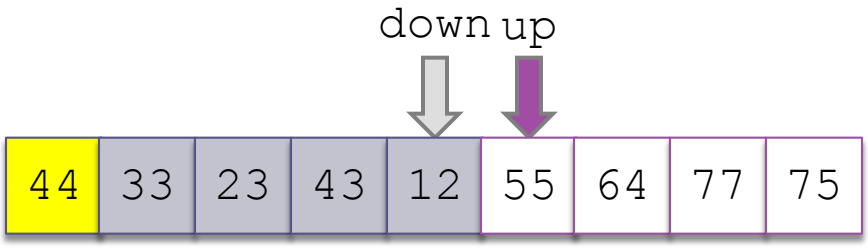
Find first element at left end greater than pivot

Trace of Partitioning (cont.)



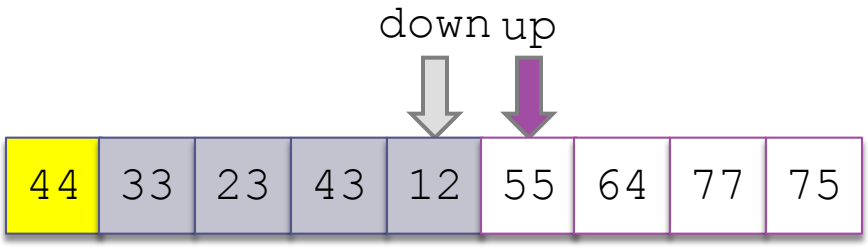
Find first element at right end less than or equal to pivot

Trace of Partitioning (cont.)



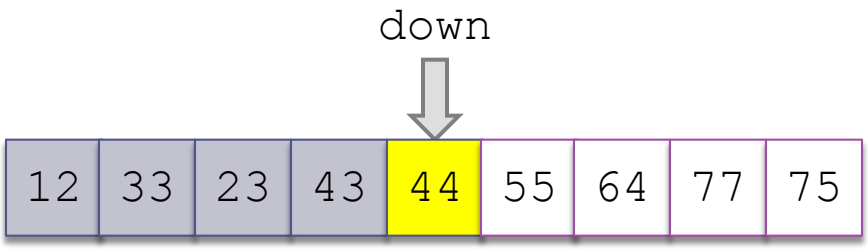
Since down has "passed" up, do not exchange

Trace of Partitioning (cont.)



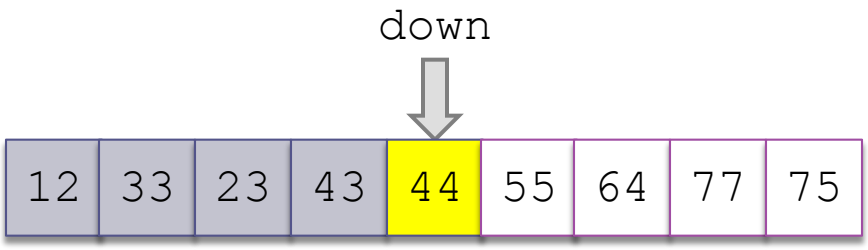
Exchange the pivot value with the value at `down`

Trace of Partitioning (cont.)



Exchange the pivot value with the value at `down`

Trace of Partitioning (cont.)



The pivot value is in the correct position; return the value of `down` and assign it to the pivot index `pivIndex`

Algorithm for Partitioning

Algorithm for partition Method

```
1. Define the pivot value as the contents of table[first]
2. Initialize up to first and down to last
3. do
4.     Increment up until it selects the first element
       greater than the pivot value or up has reached
       last
5.     Decrement down until it selects the first element
       less than or equal to the pivot value or up has
       reached first
6.     if up<down then
7.           Exchange table[up] and table[down]
8. while up is to the left of down
9. Exchange table[first] and table[down]
10. Return the value of down to pivIndex
```

Code for `partition` (cont.)


```
/** Partition the table so that values from first to pivIndex
      are less than or equal to the pivot value, and values
      from pivIndex to last are greater than the pivot value.
      @param table The table to be partitioned
      @param first The index of the low bound
      @param last  The index of the high bound
      @return The location of the pivot value
  */
 private static < T
     extends Comparable < T >> int partition(T[] table,
                              int first, int last) {
   // Select the first item as the pivot value.
   T pivot = table[first];
```

Code for `partition` (cont.)


```
int up = first;
 int down = last;
 do {
   /* Invariant:
      All items in table[first . . . up - 1] <= pivot
      All items in table[down + 1 . . . last] > pivot
    */
   while ( (up < last) && (pivot.compareTo(table[up]) >= 0))
    {
     up++;
    }
   // assert: up equals last or table[up] > pivot.
   while (pivot.compareTo(table[down]) < 0) {
     down--;
   }
```

Code for `partition` (cont.)

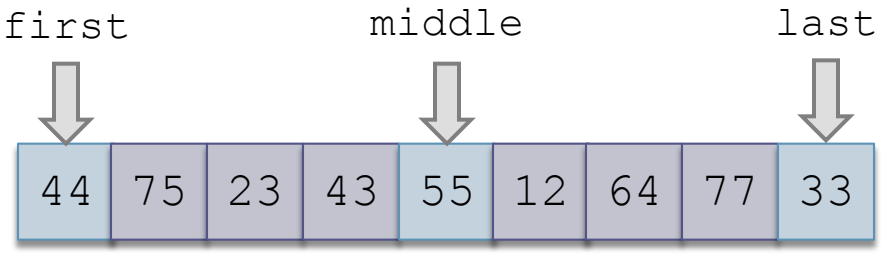

```
  // assert: down equals first or table[down] <= pivot.
    if (up < down) { // if up is to the left of down.
      // Exchange table[up] and table[down].
      swap(table, up, down);
    }
  }
  while (up < down); // Repeat while up is left of down.

  // Exchange table[first] and table[down] thus putting
  // the pivot value where it belongs.
  swap(table, first, down);
  // Return the index of the pivot value.
  return down;
}
```

Revised Partition Algorithm

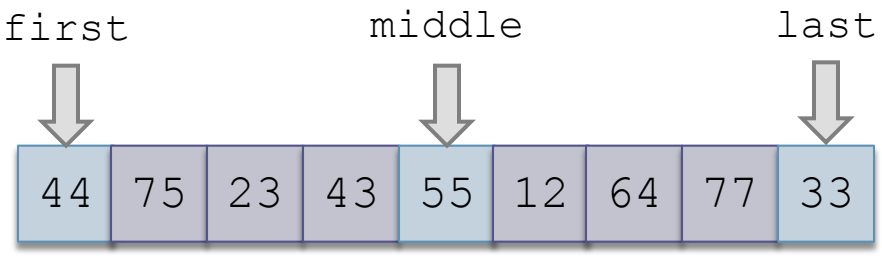
- Quicksort is $O(n^2)$ when each split yields one empty subarray, which is the case when the array is presorted
- A better solution is to pick the pivot value in a way that is less likely to lead to a bad split
  - Use three references: `first, middle, last`
  - Select the median of the these items as the pivot

Trace of Revised Partitioning

Trace of Revised Partitioning (cont.)

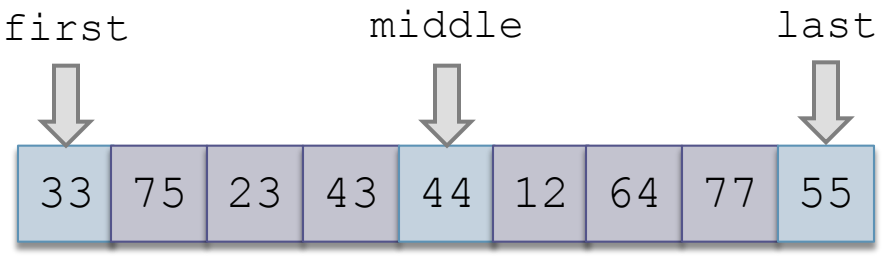


Trace of Revised Partitioning (cont.)



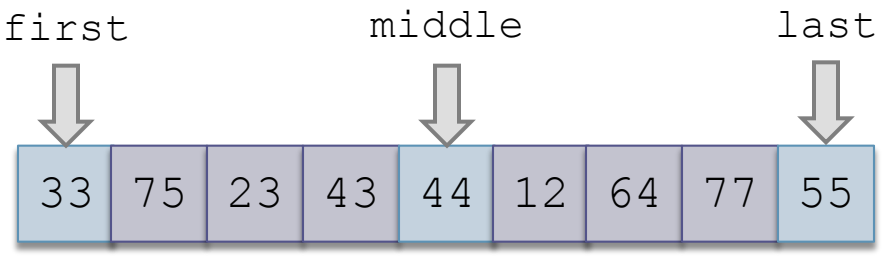
Sort these values

Trace of Revised Partitioning (cont.)



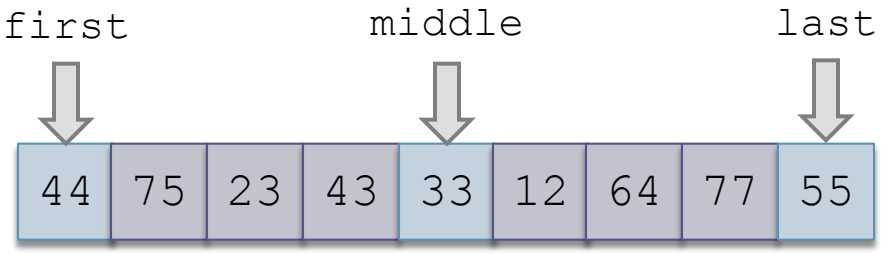
Sort these values

Trace of Revised Partitioning (cont.)



Exchange middle with first

Trace of Revised Partitioning (cont.)



Trace of Revised Partitioning (cont.)

Run the partition algorithm using the first element as the pivot

Algorithm for Revised `partition` Method

Algorithm for Revised partition Method

1. **Sort `table[first]`, `table[middle]`, and `table[last]`**

2. **Move the median value to `table[first]` (the pivot value) by exchanging `table[first]` and `table[middle]`.**

3. **Initialize `up` to `first` and `down` to `last`**

Algorithm for Revised `partition` Method

4. `do`
5. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`
6. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`
7. `if up < down` then
8. Exchange `table[up]` and `table[down]`
9. `while up` is to the left of `down`
10. Exchange `table[first]` and `table[down]`
11. Return the value of `down` to `pivIndex`

Sort Review

	Number of Comparisons		
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$