Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS 0445 slides.)

Announcements

- Upcoming Deadlines:
  - Lab 8: next Monday 11/14 @ 11:59 pm
  - Homework 8: next Monday 11/14 @ 11:59 pm
  - Midterm reattempts: tonight @ 11:59 pm

Today …

- Sorting Algorithms

Muddiest Points

- **Q: Request some guidance on Lab 8. Just like Lab 7 the provided PowerPoint and PDF provides very unclear (to no!) instruction.**
- The PowerPoint, the PDF, **and your recitation TA** should give you a clear idea of how to finish the lab in a short amount of time

Muddiest Points

- **Q: I was confused why you had to switch all your instances of Node to Node<T>**
- Since Node was a static class, it cannot use any non-static data and types, including the type parameter T of class SortingAlgorithms<T>
- So, we had to define another (static) type parameter for the static Node class
  - The type parameter could also have been named S or any other name

Muddiest Points

- **Q: Why the keyword "static" was tripping up your code? I don't think I have a firm understanding on when static should/is required to be used.**
- I had to use static because I was calling the sorting methods from the static method main
- Alternatively, I could have called the methods from the class constructor, in which case static won't be needed
  - SortingAlgorithms.java now uses that approach

Muddiest Points

- **Q: Can you post the code you did today in class? My code doesn't compile and for some reason it isn't working.**
- The code is always accessible from the Draft Slides and Code Handouts link on Canvas

Sorting Algorithms

- $O(n^2)$
  - Selection Sort
  - Insertion Sort
  - Shell Sort
- $O(n \log n)$
  - Merge Sort
  - Quick Sort
- $O(n)$ Sorting
  - Radix Sort

Sorting Algorithms

- For each algorithm
  - understand the main concept using an example
  - implement the algorithm
    - on an Array
      - iterative
      - recursive
    - on a linked list
      - iterative
      - recursive

Recursive Insertion Sort

- This pseudocode describes a recursive insertion sort.

```
Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (the array contains more than one entry)
{
    Sort the array entries a[first] through a[last - 1]
    Insert the last entry a[last] into its correct sorted position within the rest of the array
}
```

Recursive Insertion Sort

- Implementing the algorithm in Java

```
public static <T extends Comparable<? super T>>
       void insertionSort(T[] a, int first, int last)
{
   if (first < last)
   {
      // Sort all but the last entry
      insertionSort(a, first, last - 1);

      // Insert the last entry in sorted order
      insertInOrder(a[last], a, first, last - 1);
   } // end if
} // end insertionSort
```

Recursive Insertion Sort

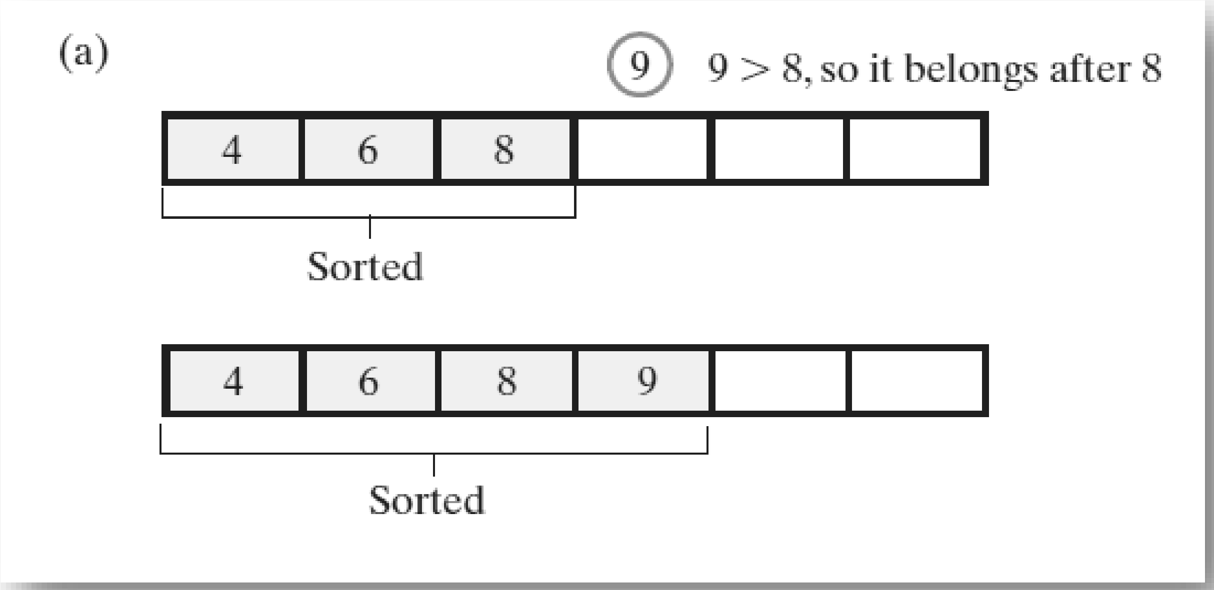
- First draft of **`insertInOrder`** algorithm.

```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// First draft.

if (anEntry >= a[end])
   a[end + 1] = anEntry
else
{
   a[end + 1] = a[end]
   insertInOrder(anEntry, a, begin, end - 1)
}
```

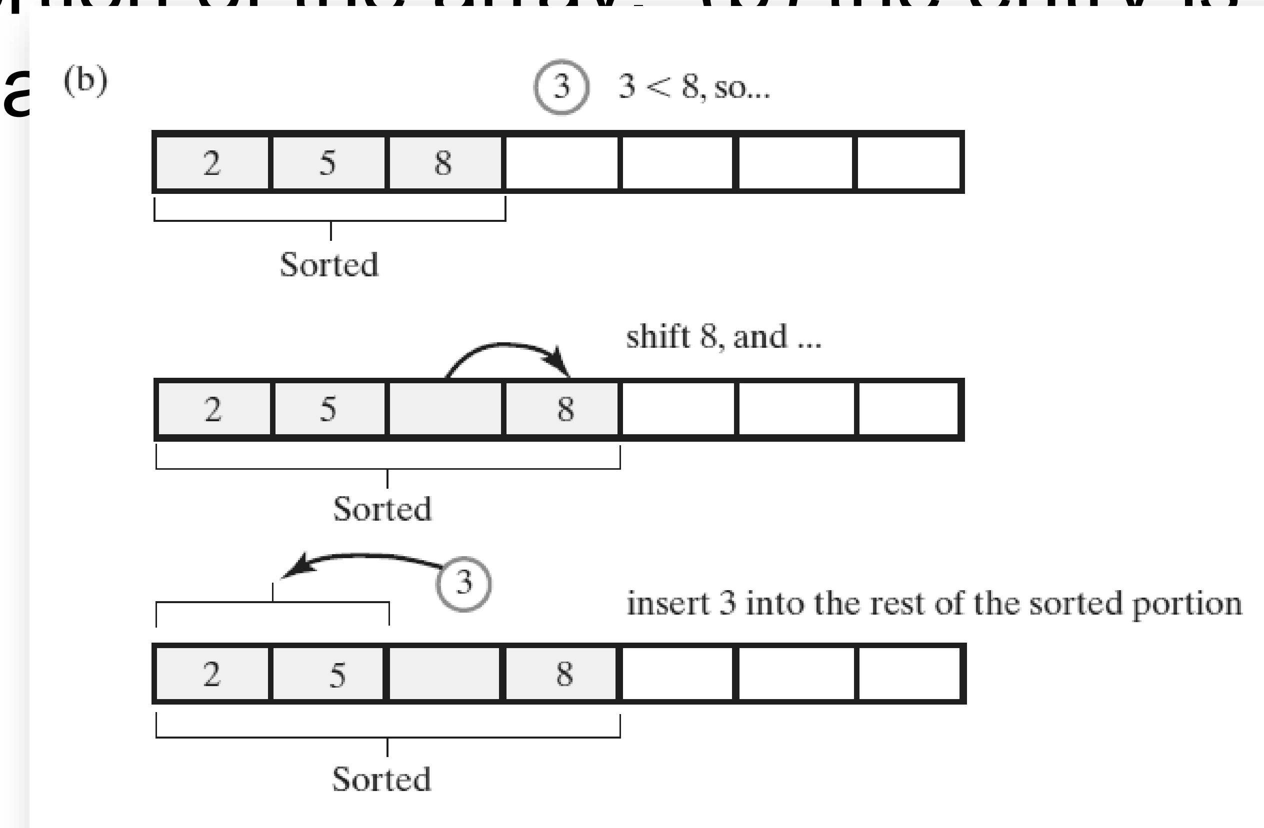
Recursive Insertion Sort

- FIGURE 8-8 Inserting the first unsorted entry into the sorted portion of the array. (a) The entry is greater than or equal to the last sorted entry



Recursive Insertion Sort

- FIGURE 8-8 Inserting the first unsorted entry into the sorted portion of the array. (b) the entry is smaller than the la



Recursive Insertion Sort

- The algorithm **insertInOrder**: final draft.

  Note: insertion sort efficiency (worst case) is $O(n^2)$

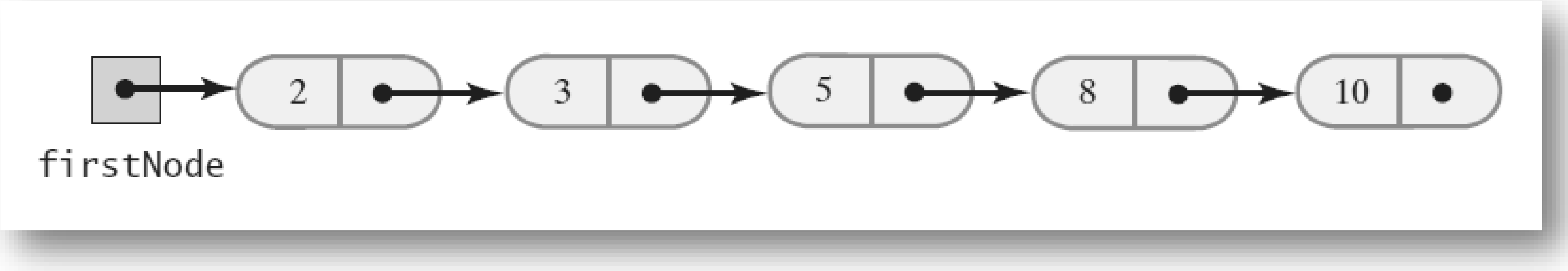
```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// Revised draft.

if (anEntry >= a[end])
   a[end + 1] = anEntry

   else if (begin < end)
   {
      a[end + 1] = a[end]
      insertInOrder(anEntry, a, begin, end - 1)
   }
   else // begin == end and anEntry < a[end]
   {
      a[end + 1] = a[end]
      a[end] = anEntry
   }
```

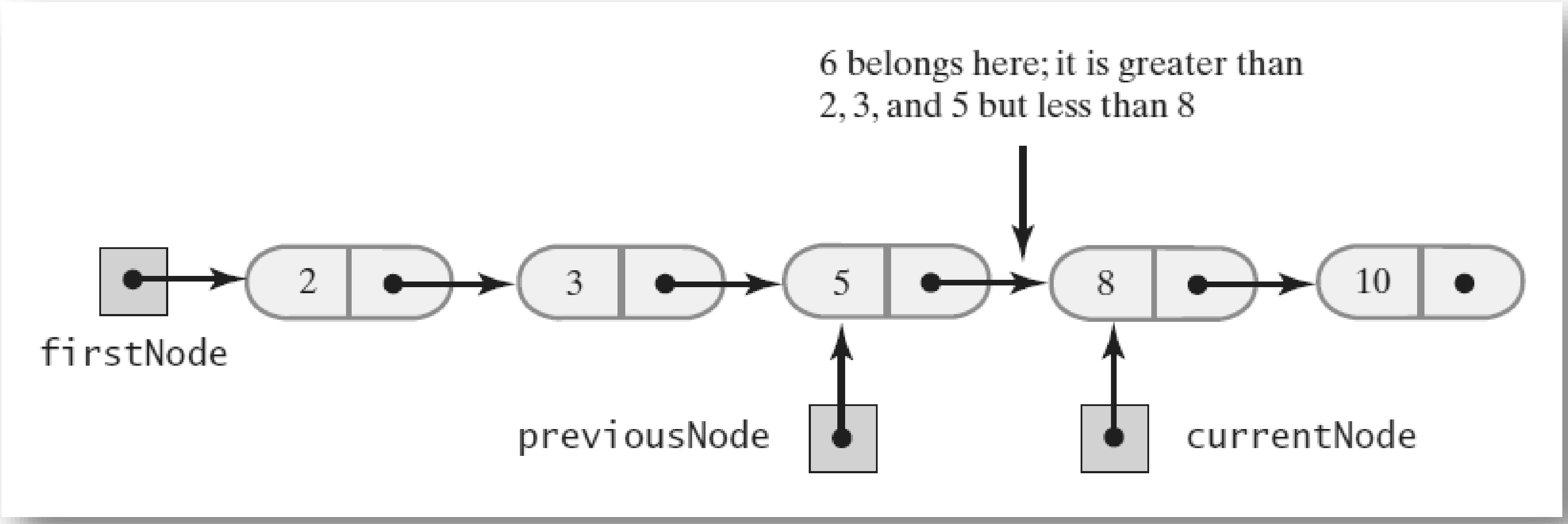
Insertion Sort of a Chain of Linked Nodes

- A chain of integers sorted into ascending order



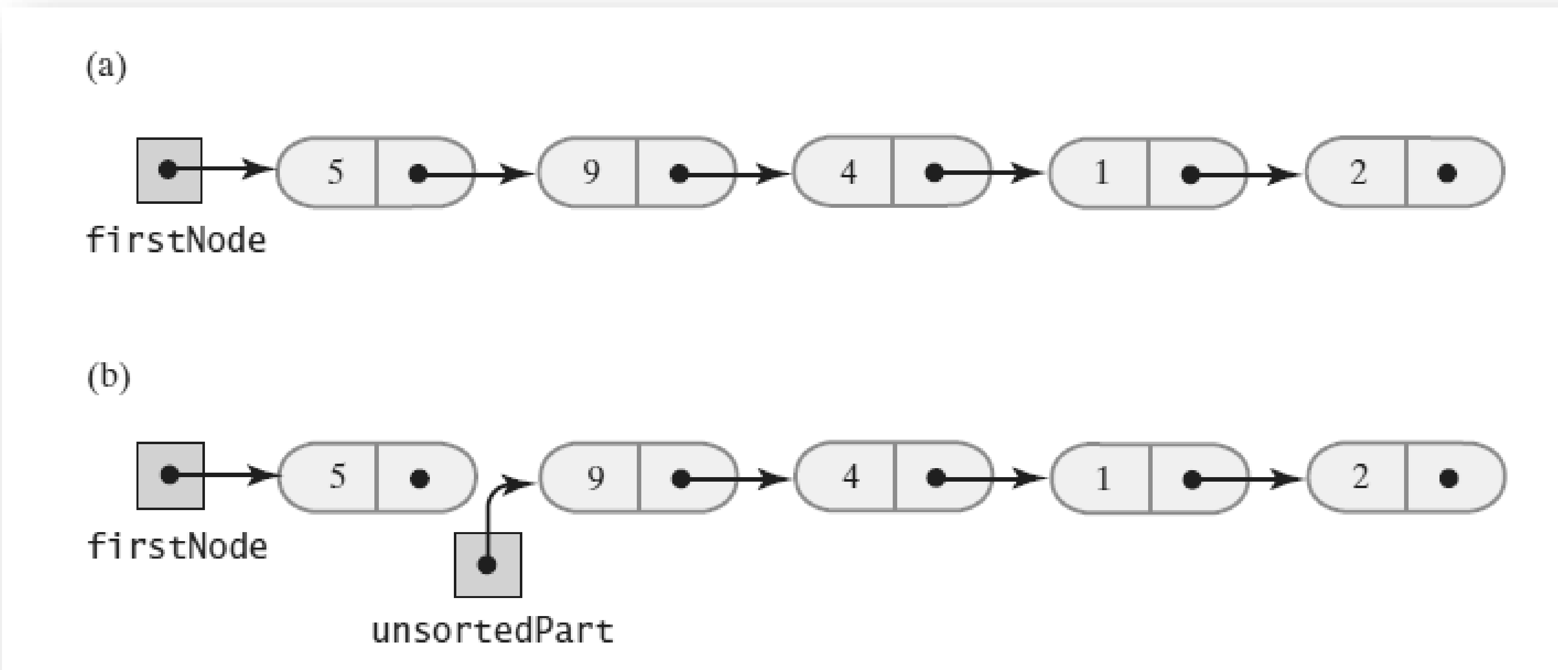
Insertion Sort of a Chain of Linked Nodes

- During the traversal of a chain to locate the insertion point, save a reference to the node before the current one



Insertion Sort of a Chain of Linked Nodes

- Breaking a chain of nodes into two pieces as the first step in an insertion sort: (a) the original chain; (b) the two pieces



Insertion Sort of a Chain of Linked Nodes

- Define an inner class **Node** that has set and get methods

```
private void insertInOrder(Node nodeToInsert)
{
   T item = nodeToInsert.getData();
   Node currentNode = firstNode;
   Node previousNode = null;

   // Locate insertion point
   while ( (currentNode != null) &&
           (item.compareTo(currentNode.getData()) > 0) )
   {
      previousNode = currentNode;
      currentNode = currentNode.getNextNode();
   } // end while

   // Make the insertion
```

Insertion Sort of a Chain of Linked Nodes

```
   } // end while
   // Make the insertion
   if (previousNode != null)
   { // Insert between previousNode and currentNode
      previousNode.setNextNode(nodeToInsert);
      nodeToInsert.setNextNode(currentNode);
   }
   else // Insert at beginning
   {
      nodeToInsert.setNextNode(firstNode);
      firstNode = nodeToInsert;
   } // end if
} // end insertInOrder
```

Insertion Sort of a Chain of Linked Nodes

- The method to perform the insertion sort.

```
public void insertionSort()
{
   // If zero or one item is in the chain, there is nothing to do
   if (length > 1)
   {
       assert firstNode != null;
       // Break chain into 2 pieces: sorted and unsorted
       Node unsortedPart = firstNode.getNextNode();
       assert unsortedPart != null;
       firstNode.setNextNode(null);

       while (unsortedPart != null)
       {
          Node nodeToInsert = unsortedPart;
          unsortedPart = unsortedPart.getNextNode();
          insertInOrder(nodeToInsert);
       } // end while
    } // end if
} // end insertionSort
```

Efficiency of Selection and Insertion Sorts

- Selection sort is $O(n^2)$ regardless of the initial order of the entries.
  - Requires $O(n^2)$ comparisons
  - Does only $O(n)$ swaps
- Insertion sort is $O(n^2)$ in the worst-case
  - Requires $O(n^2)$ comparisons and swaps
  - $O(n)$ in the best case

Some properties of Selection and Insertion Sorts

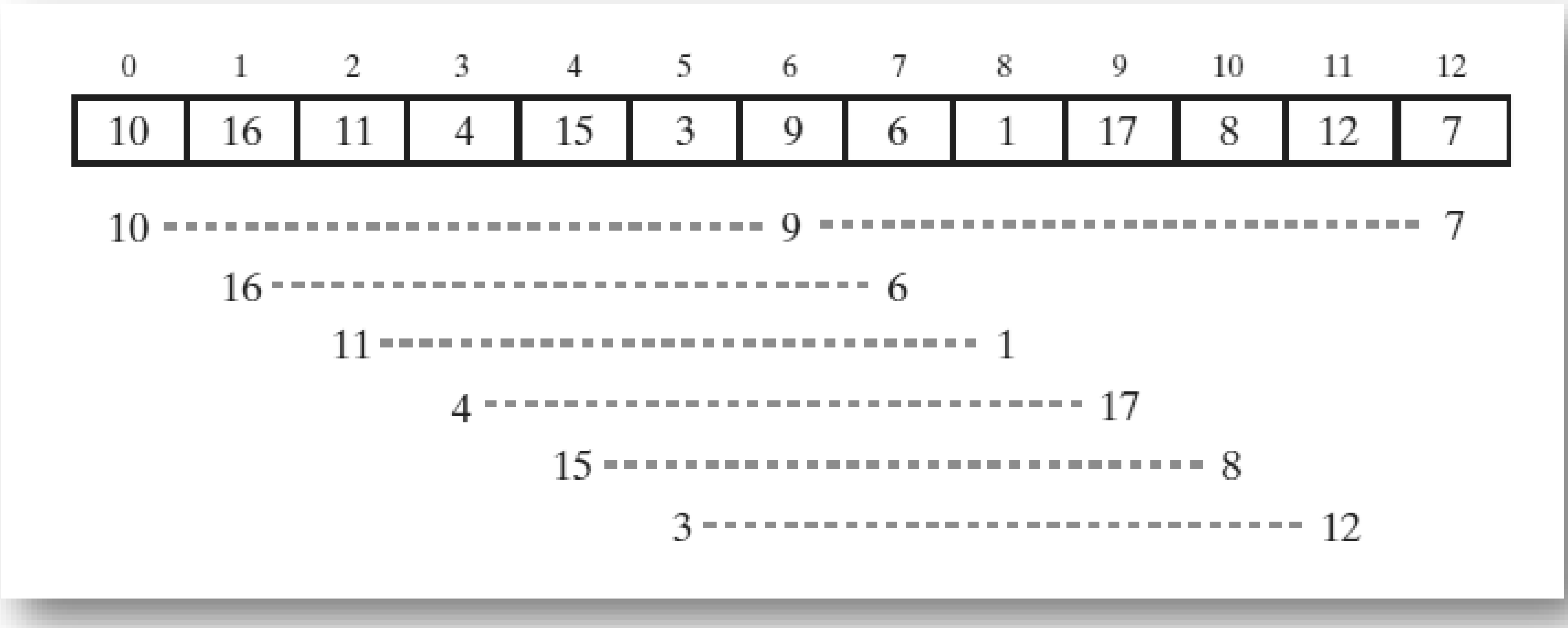
- Selection sort is
  - not stable
  - in-place
  - non-adaptive
  - provides partial solution when interrupted in the middle of execution
- Insertion sort
  - stable
  - in-place
  - adaptive
    - the more sorted an array is, the less work `insertInOrder` must do
  - very fast on small arrays
    - small constant factors

Shell Sort

- Algorithms seen so far are simple but inefficient for large arrays
- Improved insertion sort developed by Donald Shell

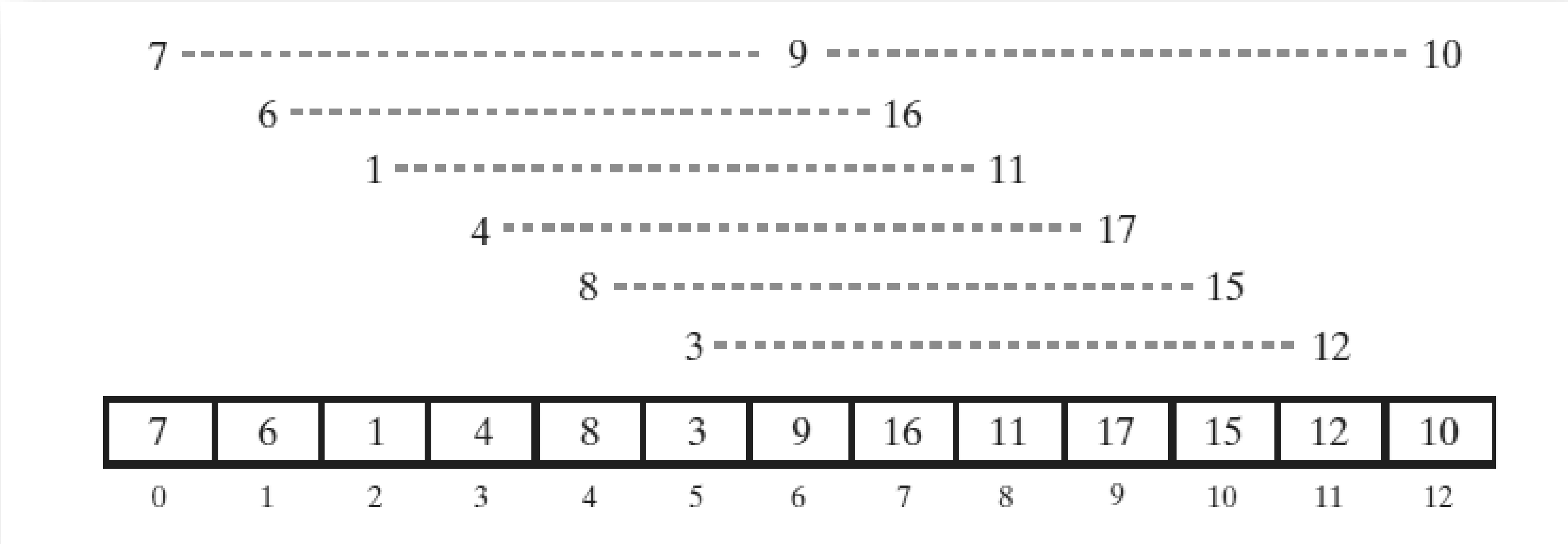
Shell Sort

- An array and the subarrays formed by grouping entries whose indices are 6 apart.



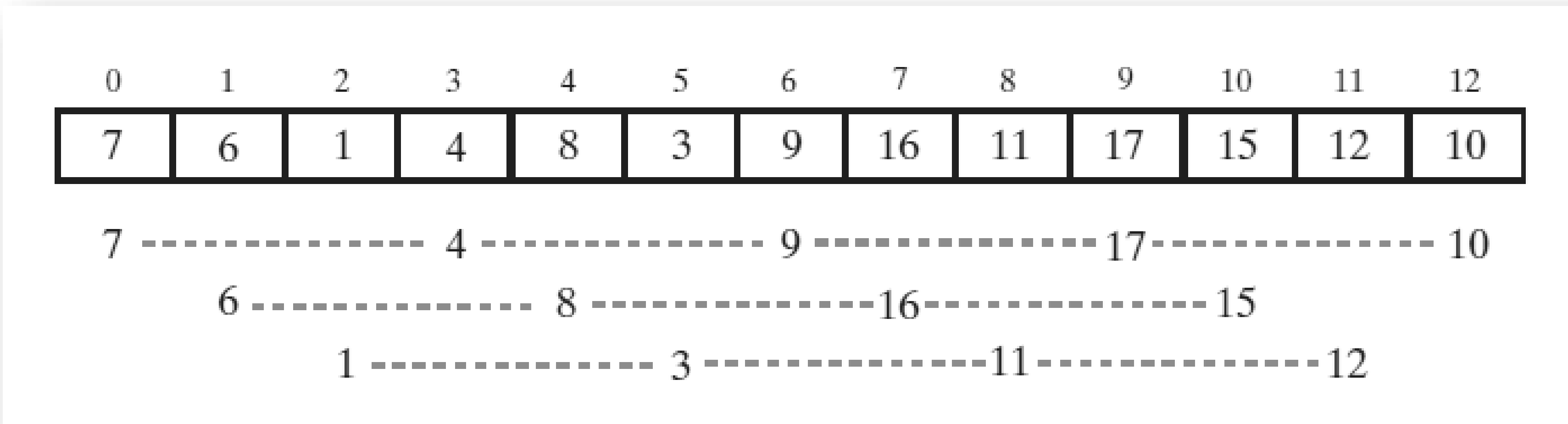
Shell Sort

- The subarrays of after each is sorted, and the array that contains them



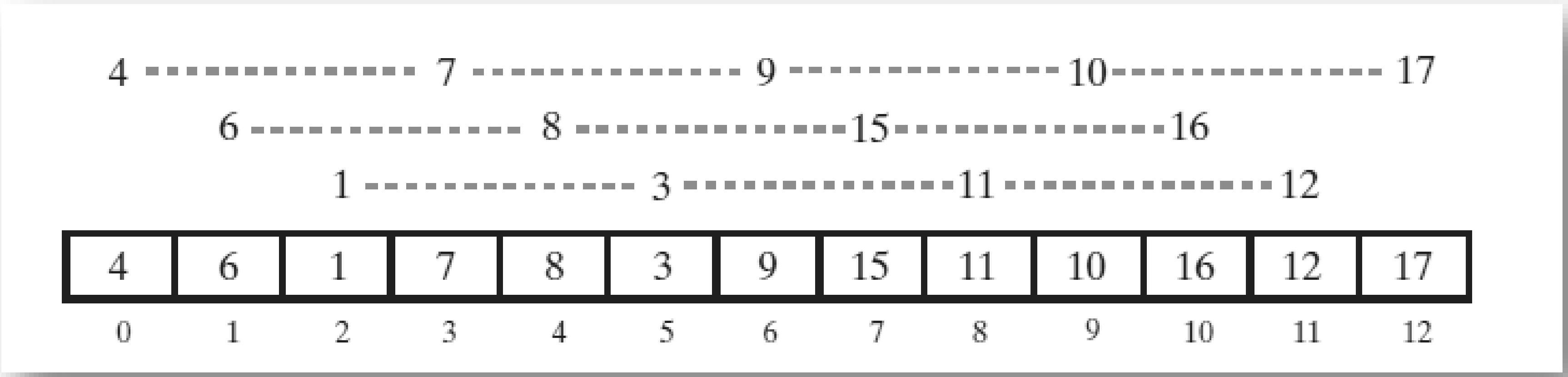
Shell Sort

- The subarrays formed by grouping entries whose indices are 3 apart



Shell Sort

- The subarrays after each is sorted, and the array that contains them



Shell Sort

- Algorithm that sorts array entries whose indices are separated by an increment of **space**.

```
Algorithm incrementalInsertionSort(a, first, last, space)
// Sorts equally spaced entries of an array a[first..last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length;
// space is the difference between the indices of the entries to sort.

   for (unsorted = first + space through last at increments of space)
   {
      nextToInsert = a[unsorted]
      index = unsorted - space
      while ( (index >= first) and (nextToInsert.compareTo(a[index]) < 0) )
      {
         a[index + space] = a[index]
         index = index - space
      }
      a[index + space] = nextToInsert
   }
```

Shell Sort

- Algorithm to perform a Shell sort will invoke **`incrementalInsertionSort`** and supply any sequence of spacing factors.

```
Algorithm shellSort(a, first, last)
// Sorts the entries of an array a[first..last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length.

   n = number of array entries
   space = n / 2
   while (space > 0)
   {
      for (begin = first through first + space - 1)
      {
         incrementalInsertionSort(a, begin, last, space)
      }
      space = space / 2
   }
```

Efficiency of Shell Sort

- Efficiency highly depends on the spacing
  - Average case O(n $\sqrt{n}$)
- Best case
  - O(n) when the number of spaces is constant
  - O(n log n) when space is divided by 2 in each iteration

Comparing the Algorithms

- The time efficiencies of three sorting algorithms, expressed in Big Oh notation

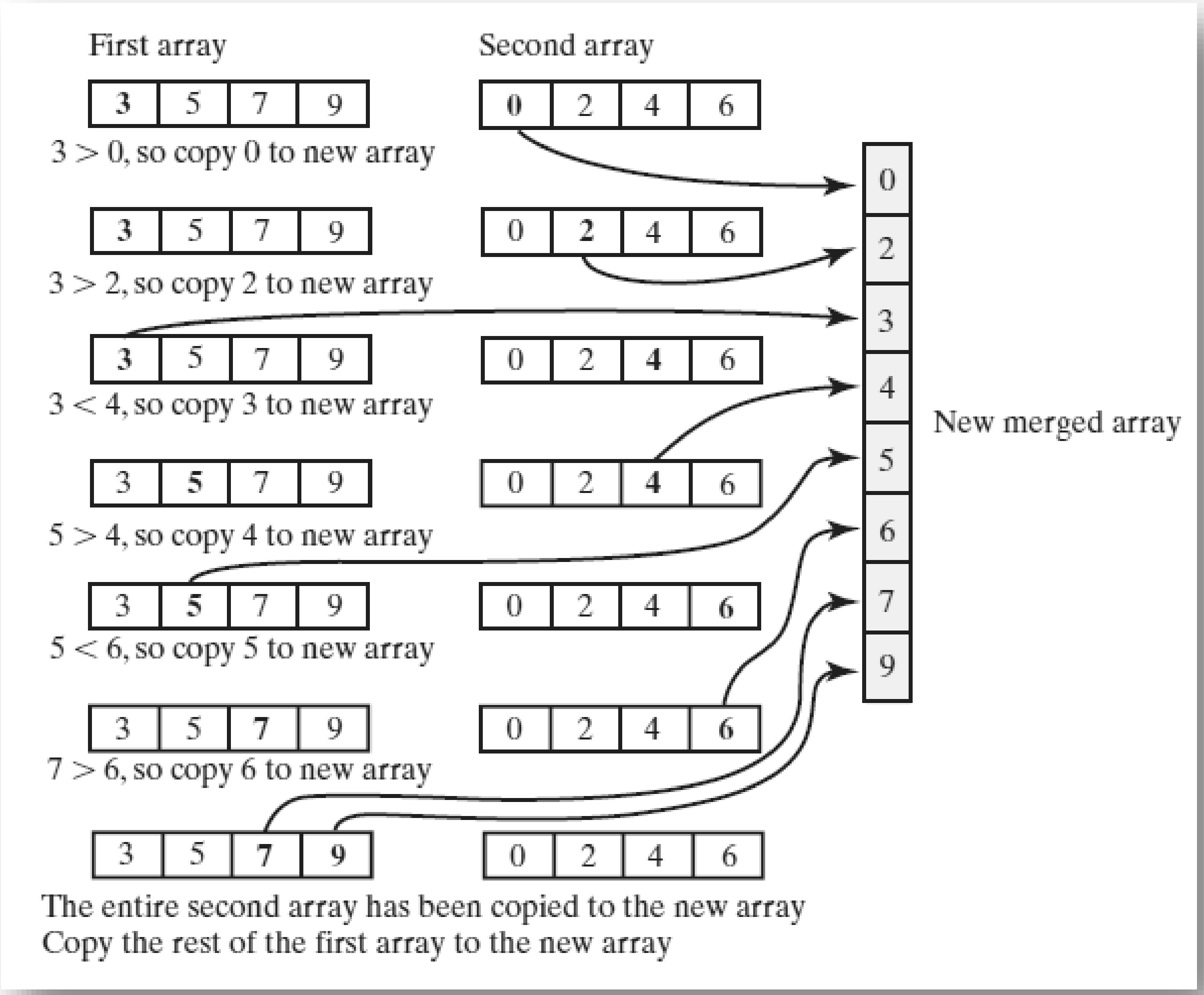
	Best Case	Average Case	Worst Case
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n)$	$O(n^{1.5})$	$O(n^2)$ or $O(n^{1.5})$

Merge Sort

- Divides an array into halves
- Sorts the two halves,
  - Then merges them into one sorted array.
- The algorithm for merge sort is usually stated recursively.
- Major programming effort is in the merge process

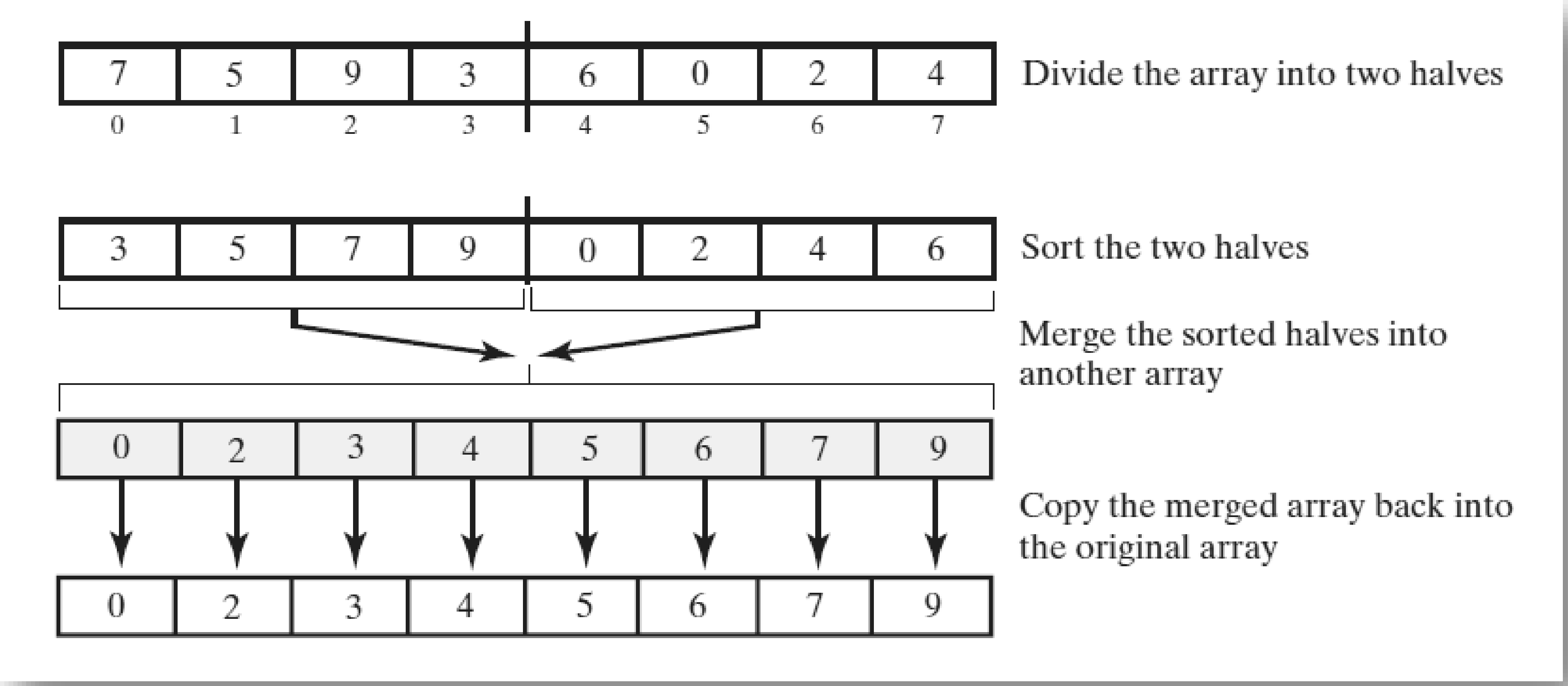
Merging Arrays

- Merging two sorted arrays into one sorted array



Recursive Merge Sort

- The major steps in a merge sort



Recursive Merge Sort

- Recursive algorithm for merge sort.

```
Algorithm mergeSort(a, tempArray, first, last)
// Sorts the array entries a[first] through a[last] recursively.
if (first < last)
{
   mid = approximate midpoint between first and last
   mergeSort(a, tempArray, first, mid)
   mergeSort(a, tempArray, mid + 1,last)
   Merge the sorted halves a[first..mid] and a[mid + 1..last] using the array tempArray
}
```

Recursive Merge Sort

- Pseudocode which describes the merge step.

```
Algorithm merge(a, tempArray, first, mid, last)
// Merges the adjacent subarrays a[first..mid] and a[mid + 1..last].
beginHalf1 = first
endHalf1 = mid
beginHalf2 = mid + 1
endHalf2 = last
// While both subarrays are not empty, compare an entry in one subarray with
// an entry in the other; then copy the smaller item into the temporary array
index = 0 // Next available location in tempArray
while ( (beginHalf1 <= endHalf1) and (beginHalf2 <= endHalf2) )
{
   if (a[beginHalf1] <= a[beginHalf2])
   {
      tempArray[index] = a[beginHalf1]
      beginHalf1++
   }
```

Recursive Merge Sort

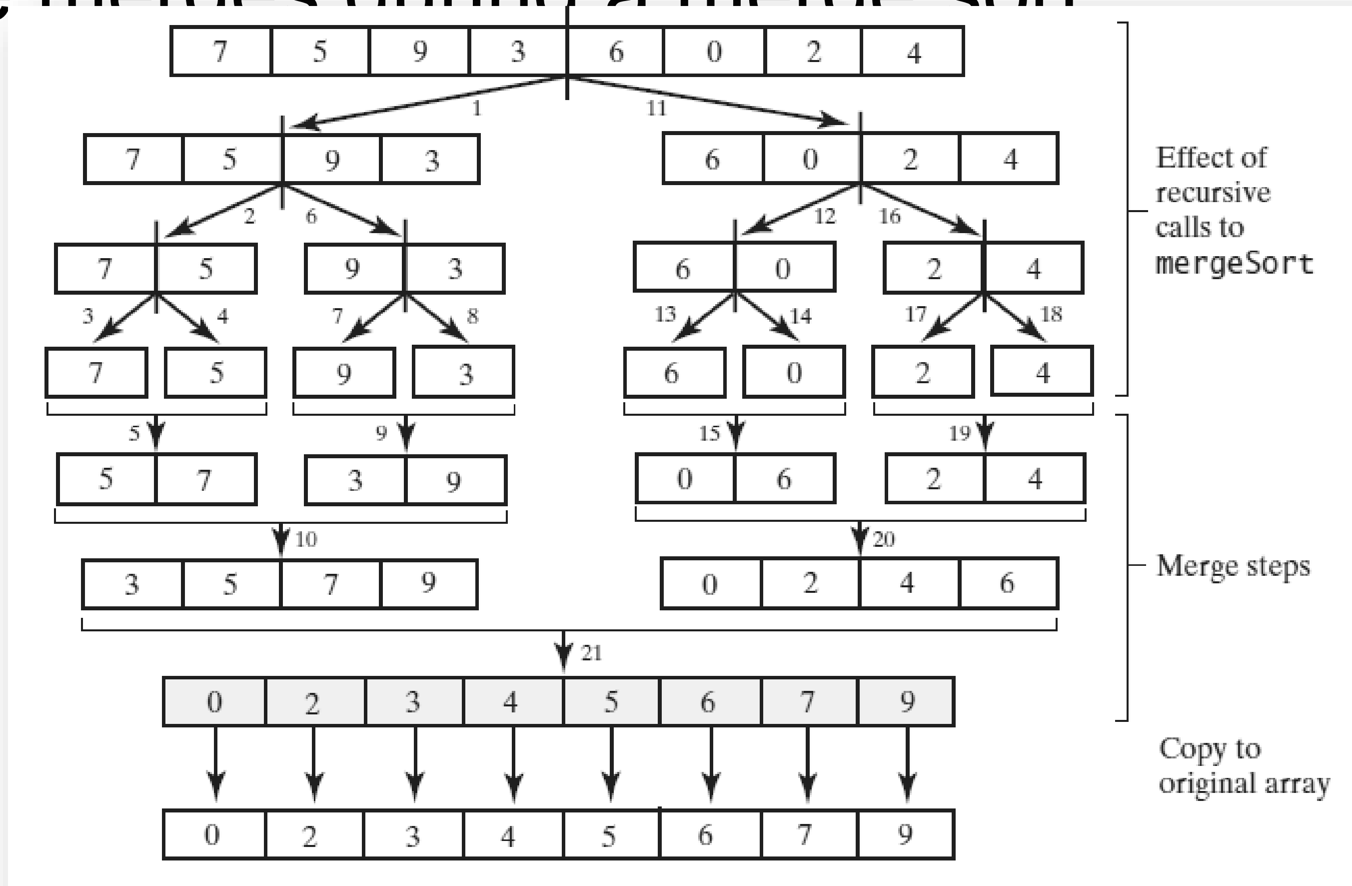
- Pseudocode which describes the merge step.

```
        tempArray[index] = a[beginHalf1]
        beginHalf1++
    }
    else
    {
        tempArray[index] = a[beginHalf2]
        beginHalf2++
    }
    index++
}
// Assertion: One subarray has been completely copied to tempArray.

Copy remaining entries from other subarray to tempArray
Copy entries from tempArray to array a
```

Recursive Merge Sort

- FIGURE 9-3 The effect of the recursive calls and the merges during a merge sort



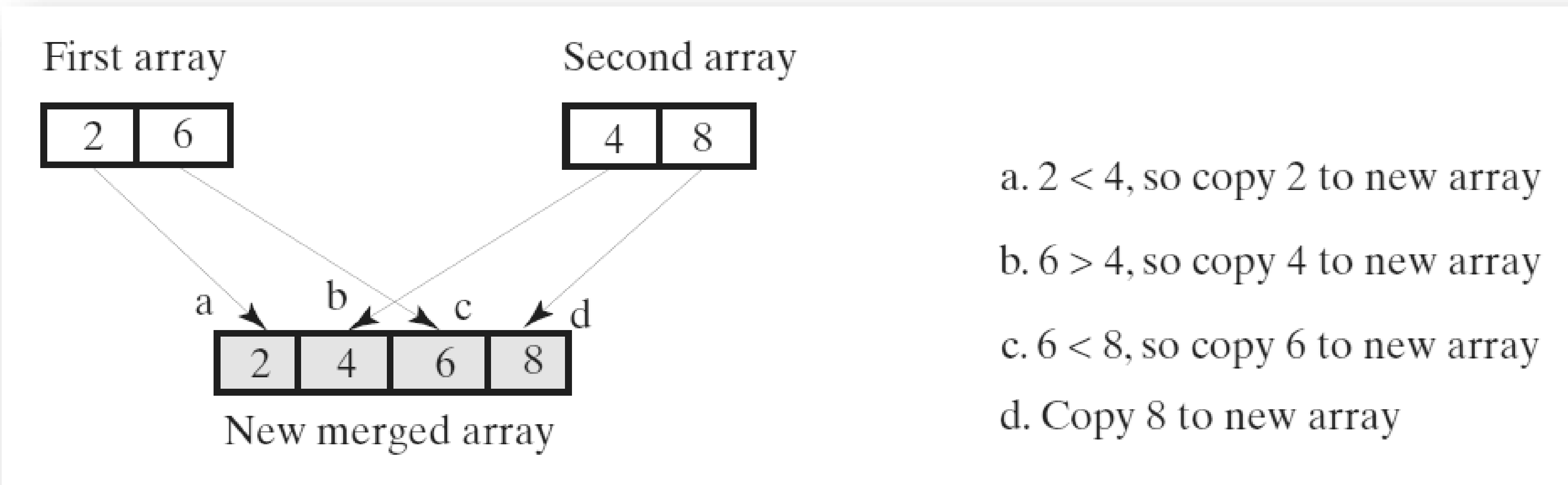
Recursive Merge Sort

- Be careful to allocate the temporary array only once.

```
public static <T extends Comparable<? super T>>
       void mergeSort(T[] a, int first, int last)
{
   // The cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] tempArray = (T[])new Comparable<?>[a.length]; // Unchecked cast
   mergeSort(a, tempArray, first, last);
} // end mergeSort
```

Efficiency of Merge Sort

- A worst-case merge of two sorted arrays.
- Efficiency is O(n log n).



Iterative Merge Sort

- Less simple than recursive version.
  - Need to control the merges.
- Will be more efficient of both time and space.
  - But, trickier to code without error.

Iterative Merge Sort

- Starts at beginning of array
  - Merges pairs of individual entries to form two-entry subarrays
- Returns to the beginning of array and merges pairs of the two-entry subarrays to form four-entry subarrays
  - And so on
- After merging all pairs of subarrays of a particular length, might have entries left over.

Merge Sort in the Java Class Library

- Class **`Arrays`** in the package **`java.util`** defines versions of a static method **`sort`**

```
public static void sort(Object[] a)

public static void sort(Object[] a, int first, int after)
```

Merge Sort Properties

- stable
- not in-place
- can be made adaptive

Quick Sort

- Divides an array into two pieces
  - Pieces are not necessarily halves of the array
  - Chooses one entry in the array—called the pivot
- Partitions the array into
  - <= pivot
  - >= pivot
  - places pivot in between the two parts
- Recursively sorts each part

Quick Sort

- When pivot chosen, array rearranged such that:
  - Pivot is in position that it will occupy in final sorted array
  - Entries in positions before pivot are less than or equal to pivot
  - Entries in positions after pivot are greater than or equal to pivot

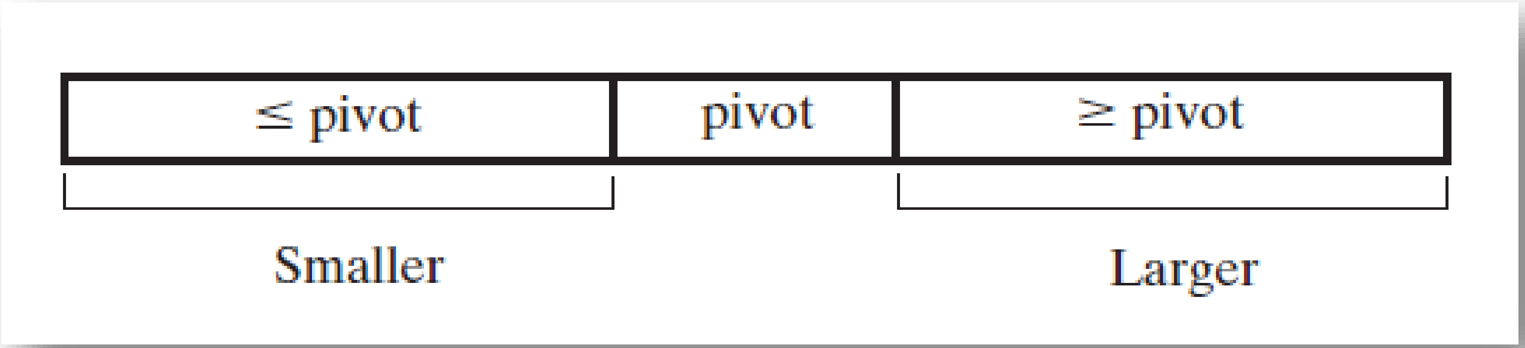
Quick Sort

- Algorithm that describes our sorting strategy:

```
Algorithm quickSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.
if (first < last)
{
    Choose a pivot
    Partition the array about the pivot
    pivotIndex = index of pivot
    quickSort(a, first, pivotIndex - 1) // Sort Smaller
    quickSort(a, pivotIndex + 1, last)  // Sort Larger
}
```

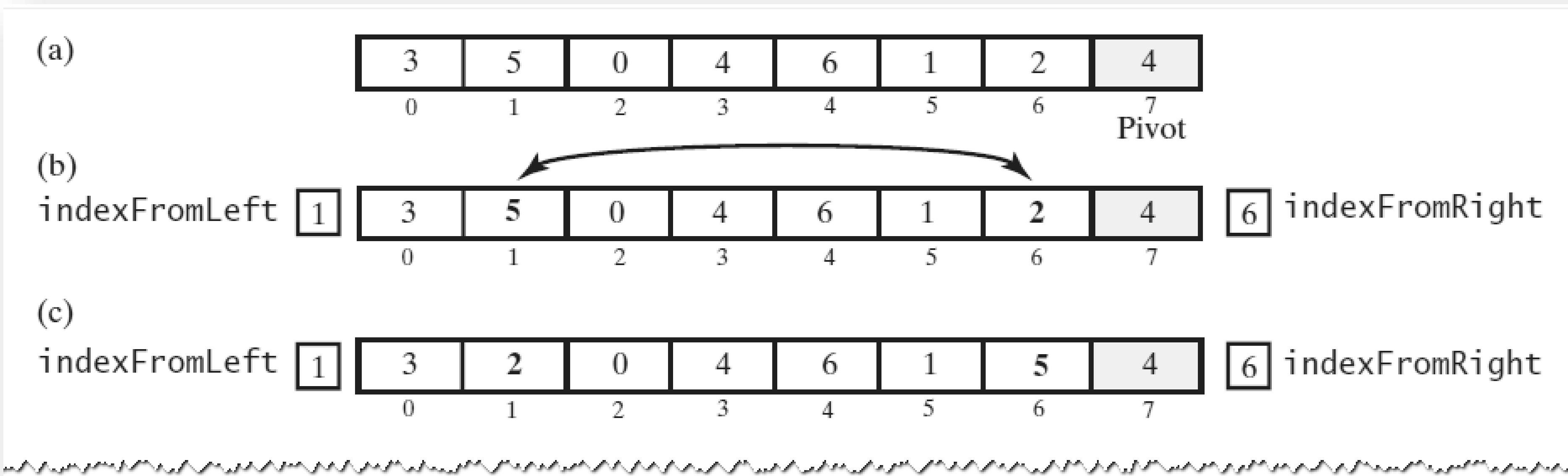
Quick Sort

- A partition of an array during a quick sort



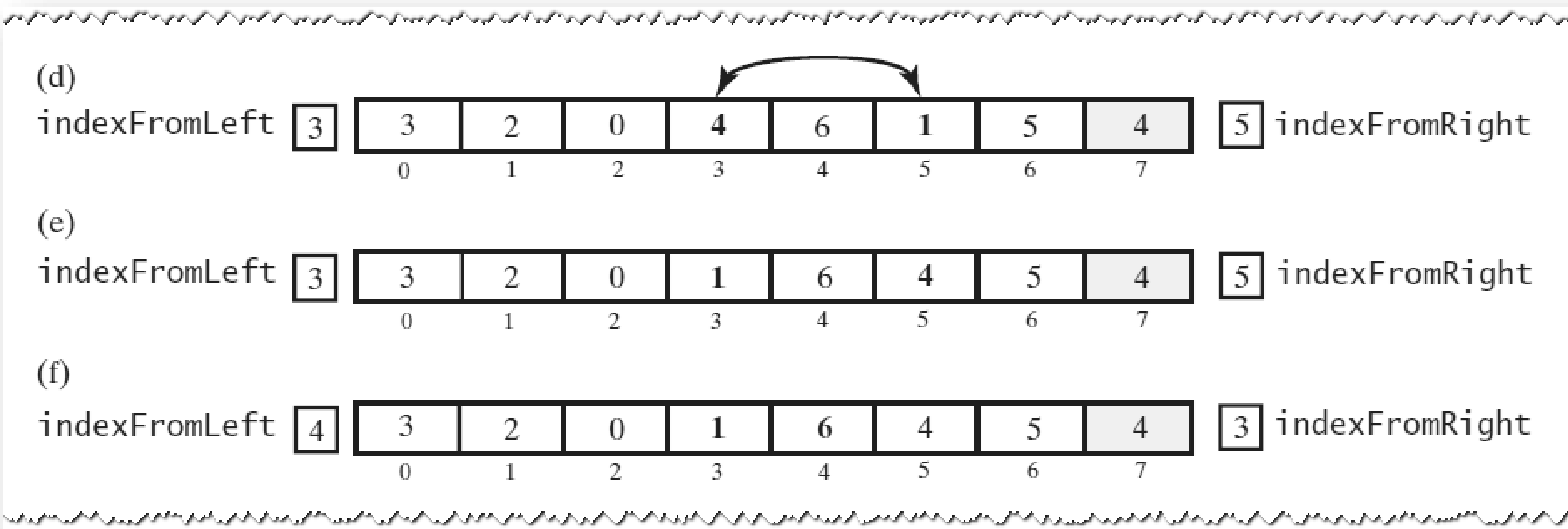
Creating the Partition

- A partitioning strategy for quick sort



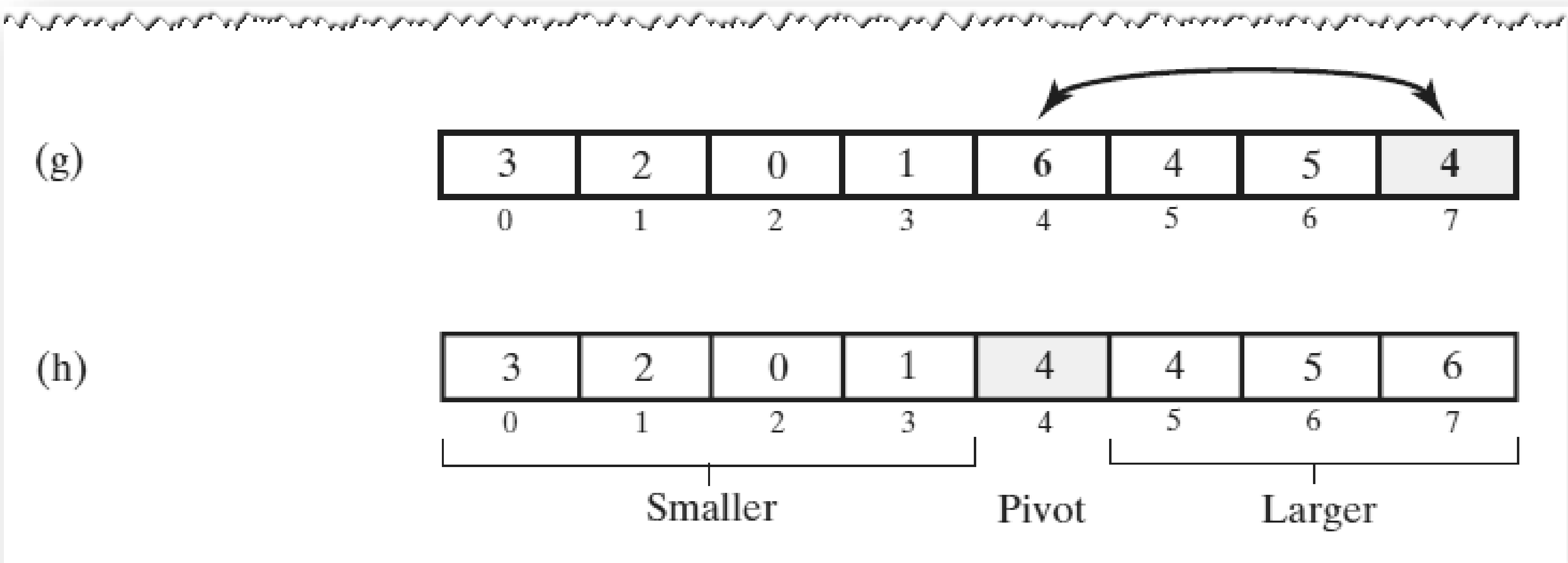
Creating the Partition

- A partitioning strategy for quick sort



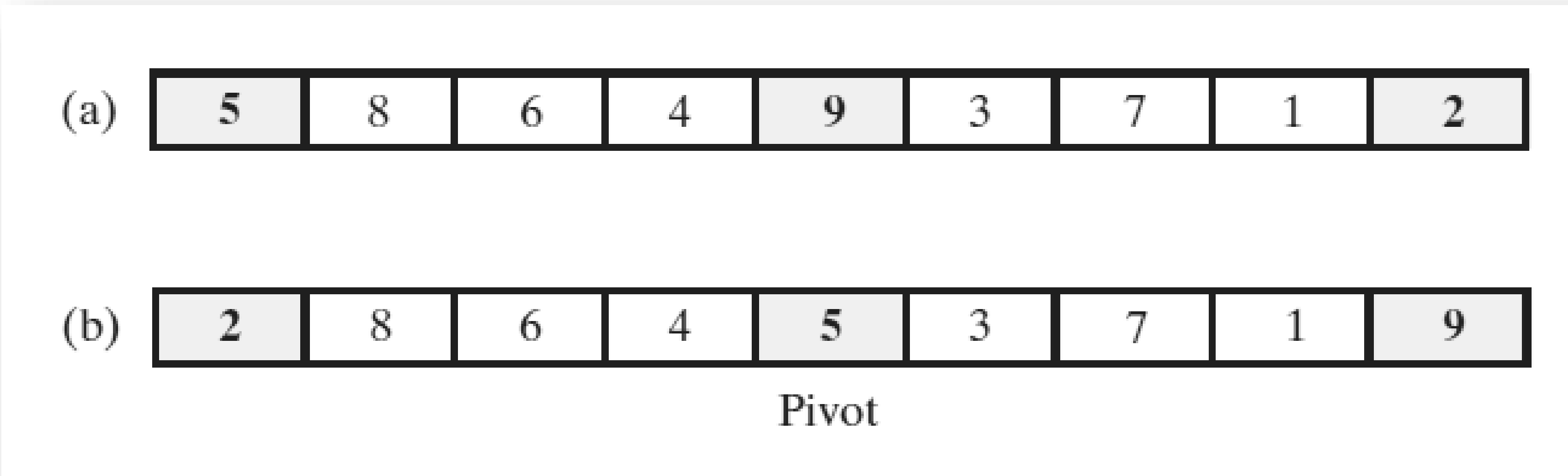
Creating the Partition

- A partitioning strategy for quick sort



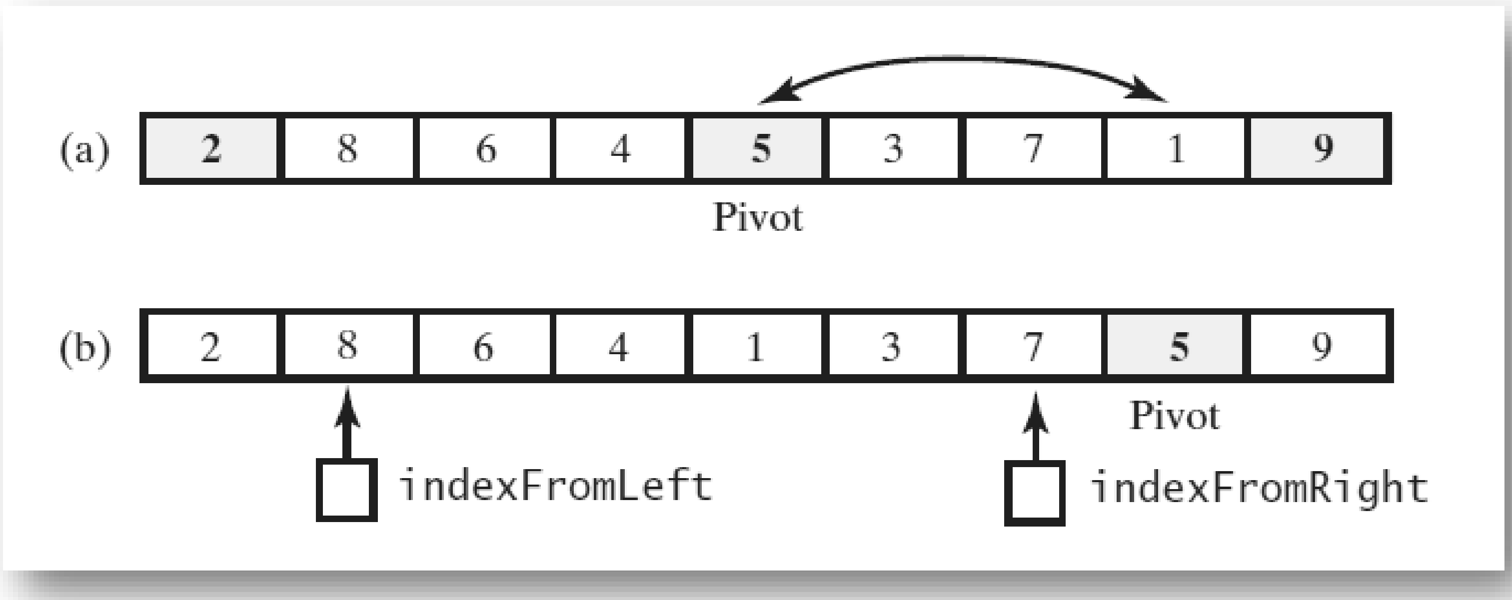
Creating the Partition

- Median-of-three pivot selection: (a) The original array; (b) the array with its first, middle, and last entries sorted



Adjusting the Partition Algorithm

- The array with its first, middle, and last entries sorted; (b) the array after positioning the pivot and just before partitioning



Adjusting the Partition Algorithm

```
Algorithm partition(a, first, last)
// Partitions an array a[first..last] as part of quick sort into two subarrays named
// Smaller and Larger that are separated by a single entry—the pivot— named pivotValue.
// Entries in Smaller are <= pivotValue and appear before pivotValue in the array.
// Entries in Larger are >= pivotValue and appear after pivotValue in the array.
// first >= 0; first < a.length; last - first >= 3; last < a.length.
// Returns the index of the pivot.

   mid = index of the array's middle entry
   sortFirstMiddleLast(a, first, mid, last)
   // Assertion: a[mid] is the pivot, that is, pivotValue;
   // a[first] <= pivotValue and a[last] >= pivotValue, so do not compare these two
   // array entries with pivotValue.

   // Move pivotValue to next-to-last position in array
```

Adjusting the Partition Algorithm

```
// Move pivotValue to next-to-last position in array
Exchange a[mid] and a[last - 1]
pivotIndex = last - 1
pivotValue = a[pivotIndex]

// Determine two subarrays:
//    Smaller = a[first..endSmaller] and
//    Larger  = a[endSmaller+1..last-1]
// such that entries in Smaller are <= pivotValue and
// entries in Larger are >= pivotValue.
// Initially, these subarrays are empty.
indexFromLeft = first + 1
indexFromRight = last - 2
done = false
while (!done)
```

Adjusting the Partition Algorithm

```
while (!done)
{
    // Starting at the beginning of the array, leave entries that are < pivotValue and
    // locate the first entry that is >= pivotValue. You will find one, since the last
    // entry is >= pivotValue.
    while (a[indexFromLeft] < pivotValue)
        indexFromLeft++

    // Starting at the end of the array, leave entries that are > pivotValue and
    // locate the first entry that is <= pivotValue. You will find one, since the first
    // entry is <= pivotValue.
    while (a[indexFromRight] > pivotValue)
        indexFromRight--

    // Assertion: a[indexFromLeft] >= pivotValue and
    //            a[indexFromRight] <= pivotValue
    if (indexFromLeft < indexFromRight)
```

Adjusting the Partition Algorithm

```
//          a[indexFromRight] <= pivotValue
    if (indexFromLeft < indexFromRight)
    {
        Exchange a[indexFromLeft] and a[indexFromRight]
        indexFromLeft++
        indexFromRight--

    }
    else
        done = true
}

Exchange a[pivotIndex] and a[indexFromLeft]
pivotIndex = indexFromLeft

// Assertion: Smaller = a[first..pivotIndex-1]
//            pivotValue = a[pivotIndex]
//            Larger = a[pivotIndex+1..last]
return pivotIndex
```

The Quick Sort Method

- Above method implements quick sort.

```
/** Sorts an array into ascending order. Uses quick sort with
    median-of-three pivot selection for arrays of at least
    MIN_SIZE entries, and uses insertion sort for smaller arrays. */
public static <T extends Comparable<? super T>>
       void quickSort(T[] a, int first, int last)
{
   if (last - first + 1 < MIN_SIZE)
   {
      insertionSort(a, first, last);
   }
   else
   {
      // Create the partition: Smaller | Pivot | Larger
      int pivotIndex = partition(a, first, last);

      // Sort subarrays Smaller and Larger
      quickSort(a, first, pivotIndex - 1);
      quickSort(a, pivotIndex + 1, last);
   } // end if
} // end quickSort
```

QuickSort in the Java Class Library

- Class **`Arrays`** in the package **`java.util`** uses a quick sort to sort arrays of primitive types into ascending order

```
public static void sort(type[] a)

public static void sort(type[] a, int first, int after)
```